Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

The Development of an Executable Graphical Notation for Describing Direct Manipulation Interfaces

A dissertation presented
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy in Computer Science
at Massey University

Christopher Henry Edwin Phillips

1993

1094080483

Abstract

The research reported in this thesis involves the development of an executable semi-formal graphical notation, Lean Cuisine+, for describing the underlying behaviour of event-based direct manipulation interfaces, and the application of the notation both in 'reverse engineering', and during the early design phase of the interface development life cycle.

A motivation for the research stems from the need for tools and techniques to support high level interface design. The research supports and brings together a number of views concerning the requirements of notations at this level. These are that a notation should be semi-formal, graphical, executable, and object-oriented, and that to be most effective it should be targeted at a specific category of interaction. The Lean Cuisine+ notation meets all these criteria, the underlying meneme model matching closely with the selection-based nature of direct manipulation interfaces.

Lean Cuisine+ is a multi-layered notation, and is a development of Lean Cuisine (Apperley & Spence, 1989). The base layer is a tree diagram which captures part of the behaviour of an interface in terms of constraints and dependencies between selectable dialogue primitives. Further constraints and dependencies associated with the dynamics of the interface are captured through overlays to the basic tree diagram. An orthogonal task layer captures any temporal relationships between primitive task actions, and provides a link with higher level functionality. Lean Cuisine+ is able to combine both static and dynamic modelling in a coherent manner, thus avoiding the necessity of employing separate and possibly disjoint models at the early design stage. A software support environment for the notation is also specified and partially prototyped.

The research demonstrates the advantages of a notation which can be executed to provide limited but valid early simulation of the dynamic behaviour of the interface under design. A mapping from Lean Cuisine+ to a dialogue implementation language, DAL (Anderson, 1993), is also developed in support of the view that a multi-notational approach to interface development is required, and that it must be possible to move easily from initial specification to prototyping and implementation.

The Lean Cuisine+ descriptions of aspects of the Apple Macintosh interface included in the thesis show the notation to be capable of handling a range of direct manipulation interaction styles and a variety of interface objects. A five stage methodology for the construction of Lean Cuisine+ specifications for new interfaces is also developed, and applied in two case studies.

Acknowledgements

I wish to thank Professor Mark Apperley, not only as my chief supervisor for his guidance, support and enthusiasm throughout this research, but also in his capacity as head of the Computer Science Department for creating the space which permitted its completion within a manageable time frame.

Thanks must also go to my second supervisor Dr Peter Kay for his painstaking reading and correction of the thesis, and to Philip Etheridge for the occasions on which he was called upon, often at short notice, to assist with the interfacing of Macintosh software. The assistance of Paul Anderson with regard to the mapping of Lean Cuisine+ into his Dialogue Activation Language is also appreciated.

The receipt of a grant from the Massey University Research Fund for the purchase of an A4 portrait monitor to support aspects of this research is also gratefully acknowledged.

On a more personal level I am grateful for the patience and support of my wife Carol throughout the research, and to my daughters for the many hours at home when because of the research I was out of circulation.

Finally, I wish to dedicate this thesis to my parents, Harry and Dorothy, for their vision in wanting me to have an education, and for supporting me through school and university at considerable personal cost.

Trademarks

Finder, HyperCard, Lisa, Macintosh and MacPaint are trademarks of Apple Computer Inc.

MacDraw is a trademark of Claris Corp.

Prototyper is a trademark of SmethersBarnes.

Star is a trademark of Xerox Corp.

Teamwork is a trademark of Cadre Technologies Inc.

Word is a trademark of Microsoft Corp.

Brief Contents

PART 1 SURVEY AND FRAMEWORK

Chapter 1 Introduction 3

Chapter 2 Direct Manipulation Interfaces 15

Chapter 3 A Review of Dialogue Models and Notations 33

Chapter 4 Extending Lean Cuisine 63

PART 2 DEVELOPMENT

Chapter 5 The Basic Lean Cuisine+ Notation 81

Chapter 6 Extending the Lean Cuisine+ Object Range 103

Chapter 7 Lean Cuisine+ in High Level Interface Design 125

Chapter 8 Software Support for Lean Cuisine+ 157

PART 3 REVIEW

Chapter 9 Conclusions and Further Work 185

REFERENCES AND GLOSSARY

References 195

Glossary 205

APPENDICES

Appendix A The Lean Cuisine Notation 227

Appendix B A Macintosh Task Analysis 231

Appendix C The Lean Cuisine+ Notation 237

Appendix D A Support Environment Prototype 245

Detailed Contents

Chapter 1 Introduction 3

1.1 Motivation for the research 3

Direct manipulation interfaces 4

The interface development life cycle (IDLC) 5

Existing interface development tools 6

The need for 'high level' tools 8

Formal vs informal specifications 10

Focus on a graphical representation 11

1.2 Objectives of the research 12

1.3 Approach adopted 13

1.4 Structure of the thesis 14

Chapter 2 Direct Manipulatiou Interfaces 15

2.1 The nature of direct manipulation 15

The object-action model 18

2.2 The desktop metaphor 20

Menus 21

Windows 21

2.3 The Macintosh interface 22

The desktop 23

Basic interaction techniques 25

Modality 26

The hierarchical file system 27

Other Macintosh applications 28

2.4 Modelling direct manipulation interfaces 29

Three views of the interface 30

2.5 Software architectures for direct manipulation interfaces 31

2.6 Review 32

Chapter 3 A Review of Dialogue Models and Notations 33

3.1 Specifying system behaviour 33

3.2 Dialogue notations 35

3.2.1 Sequence-based notations 37

State transition diagrams (STDs) 37

Event-decomposition graphs (EDGs) 41
Petri nets 43
Other sequence-based notations 46
3.2.2 State-based notations 47
Statecharts 47
3.2.3 Object-based notations 51
Lean Cuisine 51
Other object-based notations 55
3.3 Comparison and review 57
A direct manipulation dialogue model 59
A way forward 61

Chapter 4 Extending Lean Cuisine **63**
4.1 Analysis of Macintosh interaction tasks 63
Finder processes 66
MacPaint processes 68
A task taxonomy 70
4.2 Extending the scope of Lean Cuisine 72
4.3 A framework for Lean Cuisine+ 74

Chapter 5 The Basic Lean Cuisine+ Notation **81**
5.1 Introduction 81
5.2 The base layer 83
Object state variables 86
5.3 Further constraint layers 86
The selection trigger layer 86
The option precondition layer 90
The existence dependency layer 91
5.4 An extended icon object hierarchy 92
System directives 96
5.5 The addition of menus 99
5.6 Review 100

Chapter 6 Extending the Lean Cuisine+ Object Range **103**
6.1 External selection triggers 103
6.2 The Microsoft Word dialogue 103
Object state variables 107
Selection trigger layer 108
Option precondition layer 111

Existence dependency layer 111
6.3 The MacDraw dialogue 114
Object state variables 116
Selection trigger layer 119
Option precondition layer 119
6.4 The completed Finder-application linkage 121
6.5 The dialogue interlink diagram (DID) 121
6.6 Review 123

Chapter 7 Lean Cuisine+ in High Level Interface Design 125
7.1 The role of Lean Cuisine+ in interface design 125
7.2 Task analysis 126
7.3 The Lean Cuisine+ task layer 127
7.4 A methodology for constructing Lean Cuisine+ specifications 128
7.5 Case study I: the Finder document folder system 130
7.6 Case study II: an electronic mail browser 142
7.7 Review 154

Chapter 8 Software Support for Lean Cuisine+ 157
8.1 Requirements of a support environment 157
8.2 A support environment prototype 158
Execution of the specification 164
8.3 Prototyping the interface under design 170
Mapping to a dialogue specification language 171
The modified event response system (MERS) notation 172
Dialogue activation language (DAL) 173

Chapter 9 Conclusions and Further Work 185
9.1 Review of the Lean Cuisine+ notation 185
Visual aspects of the notation 186
Applicability of the notation 187
Executability of the notation 187
Mappability of the notation for prototyping and implementation 188
9.2 The wider contribution of the thesis 188
Support for the early design phase of the IDLC 188
Definition of a direct manipulation behavioural model 189
Analysis of Macintosh interaction tasks 189
9.3 General conclusions 190
9.4 Further work 191

References 195

Glossary 205

Appendix A The Lean Cuisine Notation 227

A1 Menu definitions 227
A2 Menu subgroup structures 227
A3 Lean Cuisine definitions 227
A4 Menu and meneme modifiers 228
A5 Special characters 228

Appendix B A Macintosh Task Analysis 231

B1 Macintosh Finder 232
B2 MacPaint application 235

Appendix C The Lean Cuisine+ Notation 237

C1 Basic features and definitions 237
C2 Additional constraints 242
C3 The task layer 243
C4 Execution of a Lean Cuisine+ specification 244

Appendix D A Support Environment Prototype 245

D1 Basic browsing mode 246
D2 Selection trigger mode 248
D3 Option precondition mode 250
D4 Existence dependency mode 252
D5 Object state variable (OSV) mode 253
D6 Task mode 255
D7 Execution mode 256

Figures and Tables

Figures

Figure 2.1: Macintosh document window (active state) 24
Figure 2.2: Example Macintosh desktop 25
Figure 2.3: Example dialogue box (Microsoft Word) 27
Figure 2.4: Example Macintosh document folder icon/window hierarchy 28
Figure 3.1: STD for MacWrite *Style* menu (Apperley & Spence, 1989) 39
Figure 3.2: STD for Finder document folder system 40
Figure 3.3: EDG for Finder document folder system 42
Figure 3.4: Petri net for Finder document folder system 45
Figure 3.5: Statechart for Finder document folder system 49
Figure 3.6: Lean Cuisine in action 52
Figure 3.7: Lean Cuisine diagram for Finder document folder system 53
Figure 4.1: *Get Value* subtask 65
Figure 4.2: Taxonomy of interaction tasks 71
Figure 4.3: Lean Cuisine+ framework 76
Figure 5.1: Finder: Lean Cuisine+ base layer tree diagram 83
Figure 5.2: *PrintDirectory* modal subdialogue tree 85
Figure 5.3: Finder: Selection trigger layer over greyed base layer diagram 88
Figure 5.4: Finder: Option precondition layer over greyed base layer diagram 90
Figure 5.5: Finder: Existence dependency layer over greyed base layer diagram 92
Figure 5.6: Extended Finder icon object hierarchy 93
Figure 5.7: Base layer tree diagram for extended Finder document folder system 95
Figure 5.8: *OpenDisk* subdialogue involving a system directive 96
Figure 5.9: Finder diskette-related selection triggers 97
Figure 5.10: Finder diskette-related option preconditions 98
Figure 5.11: Finder menus showing option preconditions 100
Figure 5.12: Lean Cuisine+ structural dialogue components 101
Figure 6.1: Finder icon object hierarchy showing external triggers 104
Figure 6.2: Microsoft Word: base layer tree diagram 106
Figure 6.3: Microsoft Word: Internal selection triggers 109
Figure 6.4: Microsoft Word: External selection triggers 110
Figure 6.5: Microsoft Word: Option preconditions 112

Figure 6.6: Microsoft Word: Existence dependencies 113
Figure 6.7: MacDraw: base layer tree diagram for level 2 115
Figure 6.8: MacDraw: selection triggers for level 2 118
Figure 6.9: MacDraw: option preconditions for level 2 120
Figure 6.10: The DID for part of the Macintosh interface 121
Figure 6.11: Finder icon object hierarchy showing all application links 122
Figure 6.12: Dialogue levels in Lean Cuisine+ 123
Figure 7.1: Task action sequence for *Open (Folder)* over greyed base layer diagram .127
Figure 7.2: Methodology for constructing a Lean Cuisine+ Specification 129
Figure 7.3: STAGE 1: Finder task decomposition 132
Figure 7.4: STAGE 2: Initial Finder subdialogue trees 135
Figure 7.5: STAGE 3: Initial Finder object hierarchy 136
Figure 7.6: STAGE 3: Refined Finder object hierarchy 136
Figure 7.7: STAGE 3: Refined Finder subdialogue trees 136
Figure 7.8: STAGE 3: Composite Finder dialogue tree 137
Figure 7.9: STAGE 4: Finder selection triggers showing Table 7.1 references 138
Figure 7.10: STAGE 4: Finder option preconditions showing Table 7.1 references ..139
Figure 7.11: STAGE 4: Finder existence dependency 140
Figure 7.12: STAGE 4: Finder task action sequence for *Empty Trash* 141
Figure 7.13: STAGE 5: Finder menus 141
Figure 7.14: STAGE 1: Mail browser task decomposition 143
Figure 7.15a: STAGE 2: Initial mail browser subdialogue trees 145
Figure 7.15b: STAGE 2: Initial mail browser subdialogue trees - contd 146
Figure 7.16: STAGE 3: Initial Mail browser object hierarchy 147
Figure 7.17: STAGE 3: Refined Mail browser object hierarchy 147
Figure 7.18: STAGE 3: Composite mail browser dialogue tree 148
Figure 7.19: STAGE 4: Mail browser selection triggers showing Table 7.3 refs 151
Figure 7.20: STAGE 4: Further mail browser constraints 152
Figure 7.21: STAGE 4: Mail browser *Open box* task action sequence 153
Figure 7.22: STAGE 5: Mail browser menus showing option preconditions 154
Figure 8.1: Support environment window format 159
Figure 8.2: Basic browsing mode showing base layer dialogue tree 160
Figure 8.3: Selection trigger overlay for *NewFolder* 161
Figure 8.4: OSV mode showing state variables for *FolderIcon* 162
Figure 8.5: Task overlay for task *Open Folder* 163
Figure 8.6: Finder simulation: example icon/window system 164
Figure 8.7: Finder simulation Stage #1: start-up state 167
Figure 8.8: Finder simulation Stage #7: folders B, C and F open 168
Figure 8.9: Finder simulation Stage #14: Trash folder open 169

Figure 8.10: Finder simulation Stage #7 under tandem execution 171
Figure 8.11a: Mapping Lean Cuisine+ to DAL 176
Figure 8.11b: Mapping Lean Cuisine+ to DAL - continued 177
Figure 8.11c: Mapping Lean Cuisine+ to DAL - continued 178
Figure 8.11d: Mapping Lean Cuisine+ to DAL - continued 179
Figure 8.11e: Mapping Lean Cuisine+ to DAL - continued 180

Tables

Table 2.1: Macintosh mouse techniques 26
Table 3.1: Classification of surveyed graphical dialogue notations 36
Table 3.2: Finder states: STDs vs. statecharts 50
Table 3.3: Required scope of a direct manipulation dialogue model 59
Table 4.1: Primitive virtual devices (Wallace, 1976) 64
Table 4.2: Interaction tasks (Foley *et al*, 1984) 65
Table 4.3: Macintosh mouse techniques 66
Table 4.4: Taxonomy of interaction tasks 71
Table 4.5: Required scope of a direct manipulation dialogue model 73
Table 4.6: Aspects of the dialogue model requiring Lean Cuisine extensions 74
Table 5.1: Initial set of Finder OSVs 87
Table 5.2: Application icon related OSVs 94
Table 5.3: Finder menu option preconditions 100
Table 6.1: OSVs for Microsoft Word 107
Table 6.2: OSVs for level 2 of MacDraw 117
Table 7.1: STAGE 1: Finder tasks 133
Table 7.2a: Initial set of Finder OSVs 138
Table 7.2b: Initial set of Finder OSVs - contd 139
Table 7.3: STAGE 1: Mail browser tasks 144
Table 7.4a: OSVs for mail browser 149
Table 7.4b: OSVs for mail browser - contd 150
Table 8.1: Finder simulation: initial OSV values 165
Table 8.2: Icon set and window stack at each stage of the simulation 166
Table 8.3: Feature analysis of Lean Cuisine+ and DAL 175

Part 1

Survey and Framework

Chapter 1 Introduction 3

Chapter 2 Direct Manipulation Interfaces 15

Chapter 3 A Review of Dialogue Models and Notations 33

Chapter 4 Extending Lean Cuisine 63

Chapter 1

Introduction

"The problem addressed here is not how to construct good interfaces; it is how to provide an environment in which good interfaces can be constructed."

Hartson & Hix, 1989

1.1 Motivation for the research

As human-computer interfaces have become easier to use they have become harder to create. Graphical interface software is large, complex, and difficult to debug and modify. The design of graphical interfaces has been likened to the design of buildings - partly an art, partly a science (Foley & Van Dam, 1984). An application's interface can account for a significant fraction of the code. Surveys of artificial intelligence applications, for example, report that 40 or 50 percent of the code and run-time memory are devoted to interface aspects (Bobrow, Mittal & Stefik, 1986). Smith (1986) points out that user interface software is not only critical to system performance but can also represent a sizeable investment. The production of good interactive software is therefore both difficult and expensive.

Rhyne & Watson (1987) consider that interface requirements are rarely as well defined and stable as requirements for computational algorithms. Farooq & Dominick (1988, p.481) are also of the view that the creation of a user interface requires special skills, tools and methodologies, "because it is an intrinsically different activity from the coding of computational algorithms". In the context of menu systems, Edmondson & Spence (1992, p.1) observe that "a designer has available, on the one hand, an embarrassingly large repertoire of interaction and presentation techniques, but on the other, a distressingly small toolkit of guidelines, notations, methodologies, frameworks and languages to provide the means for intelligent thought".

Hartson & Hix (1989, p.8) define dialogue "as the observable two-way exchange of symbols and actions between human and computer". In this thesis the term dialogue is used to refer to the *structure* of the human-computer interaction expressed in terms of constraints relating to dialogue primitives, and described through the employment of a formal or semi-formal dialogue model. Structural models of the human-computer interface serve as frameworks for understanding the elements of interfaces and for

guiding the dialogue developer in their construction (Hartson, 1989). Rhyne & Watson (1987) take the view that some symbolic specification of interface behaviour appears inescapable for the near future, but that the task is difficult, and can only be made easier through the development of better specification paradigms. A problem identified with some existing dialogue notations is that they describe only the 'surface behaviour' of the interface. New notations must go beyond this and must be capable of associating the functionality of the interface with objects, actions and states. Two concepts associated with dialogue design underpin the research reported. The first is the notion of 'dialogue independence' - the separation of the design of dialogue from the design of computational software. Most current approaches to interface development are based to some extent on dialogue independence. The second is the isolation of the *design* of interactive dialogues from the detail of their implementation.

Interface representation is currently achieved through a variety of notational schemes, including both textual and graphical representation languages. Control structures govern how sequencing among dialogue and computational components is designed and executed, the two basic kinds being 'sequential' and 'asynchronous' (Hartson, 1989). In sequential dialogue only one task is presented to the user at any one time, whereas in asynchronous dialogue more than one task or thread may be available. Asynchronous dialogue is also described as multi-threaded. In this thesis it is assumed to be multi-threaded dialogue *in which there is generally an absence of sequence within the separate threads or subdialogues*. Asynchronous dialogue is also referred to as 'event-based' dialogue (Hartson & Hix, 1989, p.11), "because end-user actions (e.g., clicking the mouse button on an icon) are viewed as input events". It should be noted that asynchronous does not imply concurrent. Concurrent dialogue is multi-threaded dialogue in which more than one thread can be executed simultaneously.

Direct manipulation interfaces

So-called 'direct manipulation' interfaces are asynchronous, with the objects of interest being displayed so that actions are directly in the problem domain. They are most easily understood by example, perhaps the best known being the Apple Macintosh interface (Apple, 1989a) which is used as the basis for the case studies in this thesis. This interface is based on the 'desktop' metaphor in which icons represent documents and folders and in which the mouse is the electronic extension of the hand. This is an example of the more general 'model world' metaphor defined by Hudson & King (1986). Many real direct manipulation interfaces include elements from several styles in a single interface, e.g. menus, dialogue boxes and windows; incorporate objects of a dynamic nature, e.g. windows and icons; and involve concurrency, conditions, and object inter-relationships. The nature of direct manipulation interfaces is examined in detail in Chapter 2 of this thesis.

The specification of direct manipulation interfaces presents particular problems not encountered in other software systems. Myers (1989, p.15) states that "direct manipulation interfaces popular on many modern systems are among the most difficult to implement....because they often provide elaborate graphics, many ways to give the same command, many asynchronous input devices, a mode-free interface (the user can give any command at virtually any time), and rapid semantic feedback". Rhyne & Watson (1987) highlight the difficulties of testing for potential ambiguities in the design of asynchronous dialogues, and identify a need for tools to assist the designer in building and debugging such dialogues. Hartson & Hix (1989, p.18) see research on structural modelling of asynchronous dialogue as still embryonic, because "such dialogue is less structured than sequential dialogue". Techniques are required for recording behavioural, structural and detailed representation of both visible and non-visible aspects of direct manipulation interfaces. Ideally the techniques should be independent of tools through which they may be implemented, and be complete in their ability to represent interfaces.

The 'object-oriented' paradigm, in which hierarchies of objects are manipulated and where immediate semantic feedback is required, seems to suit the implementation of direct manipulation interfaces (Barth, 1986; Sibert, Hurley & Bleser, 1986; Rhyne, Ehrich, Bennett, Hewett, Sibert & Bleser, 1987; Hartson, 1989; Foley, Kim, Kovacevic & Murray, 1989; Bass & Coutaz, 1991; Markopoulos, Pycock, Wilson & Johnson, 1992). In describing the development of the Xerox Star interface, Smith, Irby, Kimball, Verplank & Harslem (1982) argue that such interfaces are by their very nature object-oriented. Hartson takes the view that object orientation is effective for representing asynchronous dialogue, and for representing the behaviour of specific interface features (e.g. windows), because of its event-based nature. However, he also considers that a disadvantage of object orientation is its tendency to obscure temporal relationships in the high level sequencing behaviour in the application interface.

Event-based mechanisms are currently the primary underlying techniques upon which asynchronous dialogue is constructed. Reliable guidelines for the *design* of direct manipulation interfaces do not yet exist. There is a need for tools that can assist with their design and development.

The interface development life cycle (IDLC)

The interface development process, or life cycle, can be viewed as consisting of a number of stages. Shneiderman (1987) defines an eight stage life cycle, consisting of: collection of information; definition of requirements and semantics; design of syntax and support facilities; specification of physical devices; development of software; implementation and testing; evaluation; and revision. The need to view interface development as an integral part of the software engineering process is being recognised, and this life cycle bears broad resemblance to the more generally prescribed software life

cycle (e.g. Sommerville, 1985). Although this is a step-by-step description, the process is equally iterative. A distinction must be drawn between design representations and design methodologies. The former define the space of objects that can be designed within them and impose structure on them (Moran, 1981). A representation may be incorporated into a methodology, and it is important to show where such representations fit into the scheme of things.

A variety of models at various levels of abstraction have been employed in the early stages of the IDLC. These include user models, task models, and dialogue models. From an implementation viewpoint, Green (1986) broadly classifies the latter into state transition networks, formal grammars, and event-based techniques. User and task models are applicable to the first two stages of the life cycle, that is, to information collection and requirements definition. They form important inputs to the early phase of Shneiderman's third (design) stage, where dialogue models come into play, and where prototyping of the interface may be employed. Prototyping is also sometimes called dialogue simulation, and prototypes may involve executable specifications. Hartson & Hix (1989) view prototyping as an effective way to begin evaluation and testing earlier in the life cycle than was formerly the case. Dialogue models and notations, which are the focus of this thesis, are reviewed in Chapter 3.

Existing interface development tools

Many attempts have been made to develop tools to make the task of designing and implementing interfaces easier. These include user interface toolkits, and so-called 'user interface management systems' (UIMSs). Toolkits are libraries of pre-defined interaction techniques offering support for the implementation of limited interaction styles, but providing only minimal support for interface design. UIMSs reach further back into the IDLC by providing for the specification of sequencing and dialogue control. In UIMSs, the concept of dialogue independence is explicitly recognised and supported. Both UIMSs and toolkits are intended to speed up the development of interfaces. Hill (1987) defines the key goals of a UIMS as reducing the high cost of implementing user interfaces, and helping improve the quality of user interfaces by facilitating prototyping and experimentation. Hix & Hartson (1986) consider however that many UIMSs emphasise the execution-time aspects of interface management at the expense of interface design. Koiuvnen & Mantyla (1988, p.44) state that "the concept of UIMS is still also a vague one, and many mutually inconsistent approaches exist". They go on to produce a taxonomy of toolkits and UIMSs. Myers (1989) also provides a useful survey of both toolkits and UIMSs.

All UIMSs are restricted in the forms of user interfaces they can generate. A further limitation of many UIMSs is that they require the designer to specify interfaces in a textual, formal, programming-style language (Myers, 1987b). This has proved useful

and appropriate for textual command languages but difficult and clumsy for direct manipulation interfaces. A number of UIMSs allow the designer to use more graphical styles. Examples include Menulay (Buxton, Lamb, Sherman & Smith, 1983), Trillium (Henderson, 1986), and GRINS (Olsen, Dempsey & Rogge, 1985). These are still limited for the most part to using graphical techniques for specifying static aspects of the interface, i.e. to control the *placement* of interaction objects.

A number of problems have been identified with current UIMSs and interface toolkits:

- they are generally aimed at the programmer rather than the designer (Olsen, 1987a, 1987b; Myers, 1987a, 1989; Rhyne *et al*, 1987; Cockton, 1987, 1990b);
- they have borrowed models from other areas of computer science (Cockton, 1987);
- they are difficult to use (Rhyne *et al*, 1987);
- they support only part of the IDLC (Myers, 1989);
- they cannot support the development of direct manipulation interfaces (Myers, 1987a, 1989);
- they are unavailable commercially or not portable (Myers, 1989);
- they do not support evaluation (Myers, 1989).

Cockton (1990a, 1990b) considers models and architectures to be central to the development of tool-based design environments for interactive systems, which he calls 'interactive system design environments' (ISDEs). Cockton believes that current software practice is still dominated by a 'programming language' approach, where the designer's representation of the formalism is the implementation language. Olsen (1987b, p.73) goes further, stating that "A UIMS can be viewed as a tool for increasing programmer productivity". Myers (1987a) even argues for a UIMS classification based on the level of programming skills needed to use them, which he sees as closely tied to their ease of use. Olsen (1987c) considers that difficulties in expressing interfaces in 'borrowed models' have led to UIMSs not being used, in spite of large potential productivity gains. Rhyne *et al* (1987) see the difficulties in using current UIMSs as arising partly from the difficulty of the problem, partly from the considerable programming skills required by users of today's tools, and partly because tool interfaces are themselves badly designed.

Most automated tools support the developer in the *coding* phase of software production, not the design phase, and they generally lack tools to facilitate construction of human-computer interfaces (Hartson, 1989). Markopoulos *et al* (1992) take the view that recent generations of UIMSs have provided the designer with increasingly sophisticated design tools supporting rapid prototyping. They go on to argue that prototyping is largely a trial and error software engineering activity to support implementation,

providing no explicit theory beyond the particular implementation, and lacking principles for design and design decisions. Models and tools must be developed that provide support for the design process, to enable a designer to evaluate the consistency and completeness of the definition. Rhyne *et al* (1987, p.83) argue that "consistency is more likely to result when designers have a clear and spare mental model of the interface".

It must be recognised that direct manipulation interfaces are intrinsically complex, and may include multiple states, simultaneous interaction events, complex constraints, communication with the application, and elaborate feedback. Hudson (1987) argues that syntax (for direct manipulation interfaces) should be in terms of individual objects, should be as simple as possible, and should involve physical actions such as pointing or dragging instead of more linguistic concepts. This implies that previous approaches based on state transition networks or grammars which control the overall interface are not good candidates for direct manipulation interfaces. Jacob (1986, p.287) supports this, stating that "it is unnatural, though possible, to describe the user interface of a direct manipulation system as a conventional dialogue by means of a syntax diagram or other such notation". Cockton (1987) considers that the real issue is find a language that captures the *user's* view of a direct manipulation interface as perspicuously as possible, and with as few ad hoc features and extensions to the specification technique as possible. He also considers (Cockton, 1990b) that models are of limited use when we do not really understand the properties which component abstractions should satisfy. The identification of these properties for direct manipulation interfaces is central to this research.

The need for 'high level' tools

It is not enough for the designer or user to 'patch together' an interface using a toolkit. In supporting this view, Rhyne *et al* (1987, p.85) argue that "The complexity of the design problem at this level precludes the simple assembly of interaction technique modules". There is a need for 'high level' tools, models, and techniques for the designer, providing for both interface creation and interface analysis. Foley (1987) defines high level user interface representation as embodying information about the interface in terms of objects, actions, relations, attributes, and pre- and post-conditions associated with the actions. Alexander (1987) considers it important that interface designers are able to experiment with different ideas at the early stages of the interface development process.

With one or two exceptions (e.g. the UofA UIMS (Singh & Green, 1991)), there are few tools to support high level design of user interfaces. To some extent this is a consequence of the variety of individual methodologies in use, but Rhyne *et al* (1987) consider the greater cause to be the fact that few researchers have attempted to build such tools. Both Cockton (1990b) and Guindon (1990) also support this view. Guindon

observes that high level design has seldom been empirically studied, and that it is poorly supported by software tools and environments available today. Farooq & Dominick (1988, p.491) argue that higher level directly executable specification techniques are needed to avoid burying the interface designer in low level details, "especially during the experimentation associated with prototyping such interfaces". Olsen (1987a) identifies a goal for future research as being the development of designer oriented tools - in particular, tools that automate the design process, that involve high level representations of the user interface, and that allow designers to produce different styles of user interfaces. The high level design of interfaces is arguably the weakest link in the chain, and is an under-researched area.

Parallels can be drawn with two other areas of systems design:

- *The development of data models for database design*
  In the 1960s and early 1970s databases were designed using various implementation models, (hierarchical, network, or relational). Not until the mid 1970s was attention devoted to the development of independent conceptual data models, e.g. the E-R model (Chen, 1976), for use in the early stages of data base design.
- *The production of systems requirements specifications*
  Tse & Pong (1991) argue that a requirements specification should be independent of the design and implementation of the target system, and that the supporting language should be behaviour-oriented and non-procedural. It should provide a means of improving the conceptual clarity of the problems, should allow us to model logical and physical characteristics separately, and should be expressible in a precise notation with a unique interpretation. They also present a case for a hierarchical framework to permit downwards movement from higher levels of abstraction to more detailed descriptions.

Olsen (1987b) takes the view that the use of high level specifications should simplify the production of user interfaces. However, referring to existing models created for specifying interactive tasks, he states (Olsen, 1987c) that there are very few guidelines or principles to direct high level design of interfaces. He sees most current UIMS developments as being driven not by the task model but rather by syntactic and device oriented models. Smith (1986) argues the importance of distinguishing the underlying logic or architectural form of user interface functions from the detailed presentation of those functions to the user. Anderson & Apperley (1991, p.2) state that "initially what is needed is some form of abstraction that permits the basic dialogue structure of the interface to be defined, and then the details....to be added as required". In the context of menu systems, Apperley & Spence (1989) identify the need for a design methodology for the dialogue designer which should in itself provide a good interface to

the designer, and which should reflect the *structure* of the dialogue rather than the underlying program.

In order to characterise at a high level the interaction between a user and a system an explicit representation of the *behaviour* of the system itself is needed (Kieras & Polson, 1983). Edmondson (1991, p.8) notes that this is *underlying* behaviour "as distinct from the *superficial* behaviour we observe when looking at the mechanics of the interface". A representation of the behaviour of an interactive system should be independent of the implementation, and should reflect structural properties such as hierarchy, possible modes, and consistent patterns of interaction, in addition to being easy to define and understand. This representation will involve both semantic and syntactic aspects of the interface. Olsen (1987b, p.75) takes the view that the semantics of an application are expressed in "the behaviour of the objects and actions of the user interface and in the relationship between these objects and actions". Chakravarty & Kleyn (1990) observe that the interference of events internal to a task with the events in a higher level task making use of it, or with tasks that are interleaved in a multi-threaded execution, is a major problem associated with developing a specification of high level system behaviour.

Formal vs informal specifications

There is a tension between the merits of a fully formal specification, defined in a language which is mathematically precise and in which strict syntax and semantics are used, and the need in the early stages of design for a less formal and more intuitive approach as an aid to understanding the system being designed. Carey & Graham (1987) consider that any (design) method must be formal enough to provide benefits throughout the development cycle, and natural enough for designers to use it productively. Green (1986) argues that on the one hand design notations can be very informal, since their main purpose is to record the thoughts of the designer, but that on the other hand the notations used in the implementation of user interfaces must be formal, since they will be used to produce the implementation of the interface. Cockton (1987) sees the need for users to be able to comprehend and understand future dialogue models as more important than identifying a mathematically optimum model. Moran (1981) supports this view, arguing that a model can be too abstract, and thus difficult, for most people to grasp. Harbert, Lively & Sheppard (1990) consider the range of things that must be specified in a user interface to be too broad for any one form of specification.

Guindon (1990) considers that the early stages of design are best characterised as 'opportunistic'. He defines opportunistic design (p.336) as "design in which interim decisions can lead to subsequent decisions at various levels of abstraction in the solution decomposition". The results of this study have implications for methods and environments that support the early stages of design: representation languages should

support a smooth progression from requirements expressed informally to design decisions expressed formally or semi-formally, in code.

Docker (1989) in the context of the development of a structured analysis programming environment, quotes from Gehani & McGettrick (1986, p.vii): "Formal specifications do not render informal specifications obsolete or irrelevant; although they (formal specifications) can be checked to some degree for completeness, redundancy, and ambiguity, and can be used in program verification, they are often hard to read and understand. Consequently, informal specifications are still necessary as an aid to the understanding of the system being designed; informal and formal specifications complement each other". Docker also suggests the possibility of a spectrum of descriptions going from informal at one end to strictly formal at the other, and introduces the term 'semi-formal' to describe, for example, the class of techniques called 'structured systems analysis'. He sees these techniques as having been developed (p.8) "in an attempt to improve both the approach to analysis, and to place the emphasis more on the *graphical* presentation of information as a better method of communications".

Focus on a graphical representation

It has been argued (Tse & Pong, 1991) that graphical languages are better at the higher levels of abstraction, and textual languages are better for providing detailed description. In the context of specifying systems requirements, Tse & Pong argue that graphical representation of complex material is much more comprehensible than its textual counterpart, because it is two-dimensional rather than one-dimensional, and therefore provides an additional degree of freedom in presentation; because it can show naturally both hierarchy and concurrency; because it can be read selectively rather than linearly; because it reduces the number of concepts to be held in short-term memory; and because the reader can move naturally from higher to lower levels of detail. Harel, Lachover, Naamad, Pnueli, Politi, Sherman & Shutl-Trauring (1988) consider that languages for describing reactive systems ought to make it possible to move easily from the initial stages of requirements and specification to prototyping and design. They go on to suggest that specifications, including behavioural aspects, should be based to a large extent on 'visual formalisms' depending on a small number of carefully chosen diagrammatic paradigms. (Harel, 1988, p.528) supports the use of visual formalisms for representing computer-related systems "because they are to be generated, comprehended, and communicated by humans".

This suggests an approach to interface design in which a semi-formal high level graphical specification is augmented by non-graphical detail, and subsequently mapped into a more formal dialogue specification language for implementation; that is, an approach based upon multiple specifications. At the highest level of design what is needed is a visual vocabulary suitable for 'sketching' the interface, a representation able

to capture both user and system interaction with the visual display. In the words of Jacob (1985, p.52): "a visual representation chosen for this purpose needs to describe the external (user-visible) behaviour of the user interface of a system precisely, leaving no doubt as to the behaviour of the system for each possible input. It should separate function from implementation, describing the behaviour of a user interface completely and precisely without unduly constraining the way it will be implemented".

Chakravarty & Kleyn (1990) consider that compactness and readability are crucial to the usefulness of visual formalisms (which they term 'visualisms') applied to the description of the behaviour of reactive software systems. The control aspects of such systems comprise the generation and processing of external and internal events, the actions performed as a result of the events, the directions of control flow, and the concurrency of events, all of which contribute to the difficulty of specifying and understanding system behaviour. In any visual formalism there is a trade-off between the visual complexity of the graphical notation and the effort required to use it. As the visual complexity increases, the effort required to use the notation effectively also increases.

It is interesting to note that the most successful and widely adopted graphical representations in other fields of systems design have been simple notations with a few easily understood constructs, for example, the E-R notation (Chen, 1976) in high level data base design, and the data flow notation of structured systems analysis (Gane & Sarson, 1979). The event model, a candidate for describing direct manipulation interfaces, has to date lacked a graphical representation. Issues associated with graphical notations are explored further in Chapter 3 of this thesis.

1.2 Objectives of the research

The principal objective of this research is to develop a semi-formal graphical notation and associated methodology for use during the high level design phase of interface development. The notation should be targeted specifically at direct manipulation interfaces and should be extensively evaluated in the context of an existing interface. Implicit in this objective are the following secondary objectives:

- That visually, the notation should:
  - have a small number of simple concepts;
  - be suitably expressive, and in particular reflect the object oriented nature of direct manipulation interfaces;
  - be compact, and maintain locality between dialogue elements.
- That the notation should be suited both to the analysis of existing interfaces, and to the high level synthesis of component dialogues for new interfaces.

- That the notation should be executable, and thereby support a limited form of early prototyping of interfaces.
- That the notation should be mappable into a formal dialogue specification language, and thereby support interface prototyping and implementation.

1.3 Approach adopted

The approach to achieving the objectives set out in Section 1.2 can be summarised as follows:

- The nature of direct manipulation interaction is established, and the desktop metaphor is examined with reference to the Apple Macintosh. Three views pertinent to the development of direct manipulation interfaces are identified (Chapter 2).
- Existing dialogue notations are reviewed in the context of direct manipulation interfaces, with a focus on graphical representations for specifying underlying interface behaviour to support high level design. Some preliminary results of this research have been published (Phillips, 1991). Based on this analysis, the required scope of a direct manipulation dialogue model is established (Chapter 3).
- A task analysis of part of the Macintosh interface is undertaken in order to establish the fundamental nature of direct manipulation interaction. The results of this analysis have been published (Phillips & Apperley, 1991). A framework for developing and extending Lean Cuisine (Apperley & Spence, 1989), a tree-structured graphical notation for describing the behaviour of menu systems, is defined (Chapter 4).
- Lean Cuisine is developed incrementally into an object-based multi-layered notation, Lean Cuisine+, providing for specification of the full range of constraints and dependencies which exist between selectable dialogue primitives in a direct manipulation interface. This development is achieved through the description of the behaviour of part of the Macintosh interface (Chapters 5 and 6). Aspects of this research have been published (Phillips, 1992).
- The role of Lean Cuisine+ in interface design is considered, and an orthogonal task layer is added to the notation. A five stage design methodology, commencing with a task decomposition is developed and applied to an example interface (Chapter 7).
- A specification for a software environment to support the construction, browsing, and execution of Lean Cuisine+ specifications is developed. The extensions to support execution complete the Lean Cuisine+ notation. The software environment is partially prototyped using HyperCard. Mapping to a more formal dialogue specification language to support interface prototyping and implementation is considered (Chapter 8).

- The Lean Cuisine+ notation is reviewed and the contribution of the research examined critically (Chapter 9).

1.4 Structure of the thesis

The thesis is arranged in three parts. Part 1 includes this introductory chapter, Chapters 2 and 3, which present the major part of the literature survey, and Chapter 4, which establishes a framework for the notation developed in Part 2.

In Part 2, the basic Lean Cuisine+ notation is developed in Chapters 5 and 6, and extended in Chapters 7 and 8. Chapter 7 also presents a methodology for the construction of Lean Cuisine+ specifications, and Chapter 8 defines a software environment to support the notation.

Part 3 contains a single review chapter, Chapter 9, in which the contribution of the research is examined and further work identified.

Chapter 2

Direct Manipulation Interfaces

"Human dialogue is multi-dimensional and nulti-level, allowing communication to proceed along a number of paths, and threads of conversation to be started and restarted at will."

Baecker, 1980

The purpose of this chapter is to establish the nature of so-called 'direct manipulation' interfaces. Section 2.1 considers the *surface* nature of direct manipulation through a review of the literature, and in Section 2.2 the desktop metaphor is examined through its origin on the Xerox Star interface. Section 2.3 describes the implementation of this metaphor on the Macintosh computer, and introduces the applications used later in the thesis in the development of the Lean Cuisine+ notation. Section 2.4 identifies models useful in describing direct manipulation interfaces, and in particular outlines the requirements of a 'behavioural model', as a precursor to the review of dialogue models and notations in Chapter 3. Section 2.5 reviews issues relating to software architectures for the implementation of direct manipulation interfaces, while Section 2.6 briefly reviews the chapter.

2.1 The nature of direct manipulation

Human-computer interaction has traditionally been based on the 'conversational' metaphor, in which the interaction is seen as analogous to a conversation in which each participant speaks in turn in order to work step-by-step towards some goal. This metaphor is suited to the description of most text-based command language interfaces in which the objects being manipulated are referred to by name, but remain abstract and are neither directly represented to nor manipulated by the user. Their current state or status is displayed only on request. By contrast, Nelson (1980) was aware of user excitement when the interface was constructed by what he called the 'principle of virtuality' - a representation of reality that could be manipulated. Rutkowski (1982) embodied a similar concept in his 'principle of transparency' which states that where the user is able to apply intellect directly to a task, the tool itself seems to disappear. In support of these

principles the introduction of low-cost graphics hardware has led to the development of a new metaphor - the 'model world' metaphor, where the user acts by directly engaging object representations to accomplish some goal (Hudson & King, 1986).

In the model world, the user shows what to do by selecting and manipulating visual representations of objects, and has the illusion of directly acting upon the objects of interest without the intermediary of the system (Hudson, 1987). Hutchins, Hollan & Norman (1986) see this illusion as being characterised by three aspects:

- *direct engagement* - the feeling of communicating directly with the objects of interest;
- *articulatory distance* - the degree to which the form of communication with the system reflects the semantic objects and tasks involved; and
- *semantic distance* - the degree to which the semantic concepts used by the system are compatible with those of the task domain.

In the model world, objects can be moved to indicate an operation to be performed on them, and a command can be built up through the selection of options. In comparing the two worlds, Hartson (1989) views this selection activity as corresponding directly to the naming of a menu choice, command, or object by the user in the conversational world. The two metaphors correspond to the two basic kinds of dialogue control introduced in Chapter 1, the conversational world with sequential dialogue, and the model world with asynchronous dialogue.

Shneiderman (1983) coined the term 'direct manipulation' to describe interfaces that use the model world metaphor, and defined the central ideas as (p.57): "visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest". In the words of Jacob (1986, p.283): "With a direct manipulation interface, the user seems to operate directly on the objects in the computer instead of carrying on a dialogue about them". Sutcliffe (1988) defines seven essential features of direct manipulation interfaces, which are enunciated below by reference to examples drawn from current implementations and based partly on the desktop metaphor (see Section 2.2). These are:

- *explicit action:* e.g. the selection and dragging of an icon to a waste basket;
- *immediate feedback:* e.g. the highlighting of a selected icon;
- *incremental effect:* e.g. the continuous movement of an icon in response to mouse movement;
- *reversible actions:* e.g. the undoing of a paste operation during document preparation;
- *learning by exploration:* e.g. the browsing and trial of menu options;

- *intuitive interaction:* the use of pictures of familiar objects, which are a good match to the user's conceptual model e.g. the use of entity-relationship symbols and natural drawing action in CASE tools; and
- *pre-validation:* the removal of the possibility of syntax errors through the presentation of valid operations only in each interface state.

The central idea underpinning direct manipulation interaction is the replacement of complex command language syntax by direct manipulation of the objects of interest, although as Harrison & Dix (1990, p.130) observe "precisely what constitutes such a style of interactive behaviour is often unclear". Baecker & Buxton (1987) note that many real systems incorporate elements from several styles in a single interface. Harrison & Dix see the chief features as a close relationship between input and function (often one physical action per command) and the requirement for data manipulated by the user interface (and therefore the effects of all commands) to be immediately, consistently and unambiguously visible. In current direct manipulation interfaces, interaction is largely achieved through movement of a mouse or other pointing device. It is argued that novices and experts alike benefit from the simplicity and directness of this style of interface, and that all users have the benefit of a sense of direct contact with the data to produce a result. As pointed out by Rhyne & Watson (1987) however, a limitation of current direct manipulation interfaces is that their use is largely restricted to manipulation of concrete rather than abstract objects, e.g. particular files, as opposed to sets of files characterised by some common attribute.

Shneiderman (1983) analyses direct manipulation interaction in terms of the 'syntactic-semantic' model of user behaviour. Syntactic knowledge is system dependent whereas semantic knowledge is system independent. Thus, for example, the semantic activity of moving a sentence is generally available in text editors, but the syntax varies from editor to editor. Novices are more concerned with syntax, but as they gain experience may increasingly think in higher level semantic terms. In direct manipulation interfaces, objects of interest are displayed so that actions are directly in the higher (semantic) problem domain. There is little need for the mental decomposition of tasks into multiple commands with a complex syntactic form. The designers of direct manipulation interfaces are faced with the often difficult problem of choosing the right representations and operations. Shneiderman (1987) suggests that the content of graphic representations in a direct manipulation interface is a critical determinant of utility, with confusion resulting if the user is presented with the wrong information or a too cluttered layout. Users must learn the meaning of components of graphic representations, and problems may arise if they grasp the analogical representation but then make incorrect conclusions about permissible actions. Simple metaphors, analogies or models with a minimal set of concepts seem most appropriate.

Direct manipulation interfaces provide for the management of the complexity of functionality in application software through the employment of consistent graphical elements such as pull-down menus, buttons, scroll bars, and dialogue boxes. Applications also share an organisational similarity, for example standard menu layout for common functions, in order to reduce the need for relearning. Simplicity of a direct manipulation interface may however be difficult to achieve throughout in practice because system function may be so complicated at times that a single action-to-function mapping is inappropriate. Harrison & Dix (1990) see one mechanism for factoring this complexity as 'mode'. Tesler (1981) defines mode as a state of the user interface which lasts for a period of time, and which has no role other than to place an interpretation on operator input. Mode is used in direct manipulation interfaces to cluster operations in order to give an unambiguous and consistent view of part of the dialogue. Graceful transition between modes is required. Jacob (1986) sees direct manipulation interfaces as highly moded, but much easier to use than traditional moded interfaces because of the direct way in which the modes are displayed and manipulated. He considers the psychological advantages of direct manipulation interfaces to be (p.288) "that they make the mode so apparent and easy to change that it ceases to be a stumbling block".

The object-action model

Direct manipulation interaction is typically based on the 'object-action' model. Using the mouse or keyboard, users select an object from the screen. This might be text (ordered as characters, words or paragraphs) or an object such as an icon or scroll box. Once selected the object is a candidate for actions. Some objects can be viewed from more than one perspective and may require more than one representation. In this connection, Hutchins *et al* (1986, p.98) point out that "Even a system that gives the illusion that we are manipulating objects directly needs different representations when the objects are viewed from different perspectives". Thus a document might be represented on-screen at the same time by both an icon and a window.

Good direct manipulation interfaces are to a considerable extent non-deterministic - they assume little about the manner in which users will attempt to employ the capabilities of the software, especially in regard to the sequencing of the actions. Direct manipulation interfaces should include an 'undo' feature - at the very least users should be able to undo the last action performed. Nixdorf & Kiyooka (1992) are of the view that a good interface will feature multi-level *Undo* and *Redo* commands to allow users to retrace steps backwards or forwards to the last state of satisfaction.

Many of the operations in a direct manipulation interface are polymorphic e.g. *Cut*, *Copy* and *Paste* editing options. Rosenburg & Moran (1984) refer to such operations as 'generic commands', which they define as commands which are recognised in all contexts of a computer system, and which may be viewed as extremely general

operations which make minimal assumptions about their objects, the particular interpretation depending on the nature of the objects to which they are applied. Generic commands reduce the number of commands and command names a user has to know, and they increase the consistency and predictability of the system. Rosenburg & Moran observe that 'generic' is actually a relative rather than an absolute property of commands - generic commands are simply those commands which are at one end of this dimension of 'generality of application'. Generic commands also vary widely in the degree to which their meaning is determined by the characteristics of their objects. An important consequence of this object-centredness is that generic commands have potentially greater functionality.

The original characterisation of direct manipulation interaction (Shneiderman 1982, Hutchins *et al* 1986) explains it in terms of the user's reaction to a system. Wolf & Rhyne (1987) present a taxonomy for user interface techniques which provides another view. It is suggested that direct manipulation is a characteristic shared by a number of different interface techniques, including gestural interfaces, and that it is a *subjective* attribute which interfaces may have in varying degrees. The proposed taxonomy describes interface techniques in terms of the way in which actions and objects are specified. Interfaces which are commonly referred to as direct manipulation all employ a method of 'visual correlation' for object specification, with a range of methods employed for action specification. Visual correlation methods typically place a lesser burden on the user's memory, but since the available display space constrains the number of options which can be presented simultaneously, scrolling or a hierarchical menu structure may be employed. The 'analogous action' method of action specification requires the object to be visually present, and minimises the problems of remembering action names, although some time must be expended, for example, in finding and moving an icon. Interfaces also vary in the degree to which the analogy with the physical world is literal as opposed to abstract. It is suggested by Wolf & Rhyne that analogous action methods are most appropriate where the impact of screen space constraints is minimised and the analogy with the physical world is strong.

Early examples of direct manipulation interfaces included 'WYSIWYG' word processing systems, which attempted to model on the computer display exactly what the final printed copy would look like; computer-aided design (CAD) systems such as MINNIE (Spence & Apperley, 1977); and the Xerox Star office system (Smith *et al*, 1982) which led to the Apple Lisa and Macintosh systems. Some degree of directness was also provided by 'Query-by-example' systems. The *LOGO* language (Papert, 1980) offered students the opportunity to create line drawings easily with an electronic turtle displayed on the screen. In this environment, users derived rapid feedback about their programs, could easily determine what had happened, and could quickly spot and repair errors. These features are all characteristics of a direct manipulation interface.

2.2 The desktop metaphor

Direct manipulation principles have been applied in diverse fields. It has been argued (Edmondson, 1991) that true manipulation-based interaction is domain-specific. In support of this he argues that a simulator for aircraft will not also do for trains, and that (p.2) "*domain* action skills are required to operate the interface; the interaction is *in* the domain, or a representation of it". By far the widest and best known application of direct manipulation interaction however has been in the area of office automation systems, using the 'desktop' metaphor, where the applicability of the software is rather more general. The use of the term direct manipulation to describe such interfaces seems quite valid - objects are selected and manipulated by users in order to complete tasks. However, the 'articulatory distance' (Hutchins *et al*, 1986) of such interfaces is rather greater than in the case, for example, of those of flight simulators.

The first commercially available system based on the desktop metaphor was the Xerox Star (Smith *et al*, 1982). The Star interface adhered rigorously to a small set of design principles which made the system seem familiar and friendly and which simplified interaction. Examination of the Star interface continues to provide a useful introduction to the key features of the desktop metaphor. Several innovative aspects of the Star architecture were essential to the user interface, including a bit-mapped display screen, a memory bandwidth high enough to support adequate refreshing of the display, and a mouse. The design of the Star interface was approached using several principles derived from cognitive psychology (Bewley, Roberts, Schroit & Verplank, 1983), including:

- *A familiar user's conceptual model:* electronic counterparts to the physical objects in an office are created. Windows, which correspond to open documents, and which can be moved relative to a simulated desktop and to each other, are the principal mechanism for displaying and manipulating information. Windows appear as concrete embodiments on the desktop, which occupies the entire screen, and which continues to exist even when windows appear on the screen. Closed windows are represented as icons. Objects, including icons, can be selected by pointing to them with the mouse, and once selected can be moved, copied, or deleted.
- *Seeing and pointing rather than remembering and typing:* the display screen relieves the load on the user's short term memory by making objects and commands visible, on the premise that 'recognition is easier than recall'. All commands and effects have a visible representation. Menus in particular reduce memory load but raise a number of navigation issues.
- *Property and option sheets:* objects have properties, e.g. character style settings, which are displayed on two-dimensional 'forms', both to remind users of the current settings, and to permit settings to be changed by pointing and selecting. Property sheets can be thought of as an alternative representation for objects - the

screen shows the visible characteristics of the objects. Option sheets serve the same function for commands e.g. print options.

- *WYSIWYG:* the display screen portrays an accurate rendition of the printed page, permitting all composition to be carried out on the screen.
- *Universal or Generic commands:* a few commands such as *Move*, *Copy* and *Delete* can be used throughout the system on a variety of objects.
- *Consistency and simplicity:* the interface offers a relatively high degree of consistency and simplicity. Much of what the Star does can be thought of as editing, with 'copying' as a paradigm for 'creating', on the premise that it is easier to modify something than to create it out of nothing. Additional consistency is achieved through the use of classes and subclasses e.g. folder and document icons. Generally, operations e.g. *Open* (folder), can be applied at any level.
- *Modeless interaction:* this is not strictly true. Where modes are used however, as with property sheets, they are visible and the allowable actions are constrained.
- *User tailorability:* a limited amount of user tailorability is possible.

Rosenburg & Moran (1984) argue that the Star interface shows clearly how the design of the rest of the system is crucial to the success of generic commands. Chief among these are the replacement of specialised commands with specialised objects, and the extensive use of pointing. In addition to data objects, Star also has function objects such as printers and out-baskets, thus replacing specialised commands with combinations of function objects and the interpretive *Move* command. Pointing is used extensively as a selection method to eliminate the necessity for users to name every object and property in the system.

Menus

The now familiar style of pull-down menus arranged in a horizontal row across the screen is typically part of the desktop metaphor. Nixdorf & Kiyooka (1992) argue that the first principle of designing menus is to understand the menu system's role as a 'complexity browser' - the main menu bar and the choices available form a browser for the functionality that the application provides. Specifically most menu choices typically represent an operation that can be performed on the objects appearing in the main window. Users are able to examine available options in an attempt to gain insight into the capabilities of the software. While some variation according to mode is justifiable, Nixdorf & Kiyooka consider that the bulk of the menu structure should maintain a stable comfortable base within which users can operate.

Windows

Card, Pavel and Farrell (1984, p.241) suggest that "the display of a computer provides the possibility of giving the user an *external memory* that is an extension of the

user's own *internal memory*, one with which he can remember and keep track of more information than otherwise". Windows support this notion, and 'overlapping' windows are a fundamental component of the desktop metaphor, providing the possibility for several 'virtual screens', and therefore several documents, to be visible simultaneously. Card *et al* (p.239) define a window as "a particular view of some data object in the computer". The view may be of some portion of the object. Windows are generally rectangular in shape, comprising an 'application area' providing a view of the object of interest, together with a 'frame' providing information about the window e.g. title, and possibly containing control elements such as scroll bars. Most window systems offer 'two and a half' dimensional presentation i.e. they permit windows to be 'overlapped' on the display, thus enlarging the utilisable display surface. Windows permit multiple processes to time-share a single set of input devices (mouse, keyboard etc) and allow a user to interact with more than one source of information.

From the viewpoint of the window manager, Marcus and Mullet (1990, p.76) define a window as "any discrete area of the visual display that can be moved, sized and rendered independently on the display screen". Window management facilities permit the system to maintain spatial relationships between windows as they are moved, resized, and depth-arranged. Window systems generally provide some controls which can be manipulated directly, with immediate graphical feedback which provides information reflecting the current state of a number of system parameters. Generic window management commands might also be placed in a common action bar e.g. at the bottom of the screen, or displayed in a menu. Icons, which represent closed windows in many direct manipulation interfaces, permit display space to be conserved during periods of inactivity, while providing a visual reminder to the user of the existence of the application or document. They also permit previous window states to be "immediately restored through direct interaction with visible objects" (Marcus and Mullet, 1990, p.78).

Hartson (1989) points out that most window managers are toolkits in the sense that they contain libraries of window functions - and possibly other interface features - that programmers can evoke. In direct manipulation interfaces, the window system acts as a front end to the operating system and is viewed as the next level above the display in the user interface (Goodfellow, 1986). Card & Henderson (1987) warn that conflicts among tasks for the use of screen space may lead to high overheads as users move, reshape, or scroll windows. Window managers are reviewed in Williams (1986) and Myers (1988), and Macintosh windows and icons are considered further in Section 2.3.

2.3 The Macintosh interface

Hudson (1987) observes that direct manipulation interfaces are perhaps best understood by example. The Star system laid the groundwork for the Apple Lisa and

Macintosh interfaces. The Macintosh, which was introduced in 1984, was the first mass-market computer featuring a high resolution bit-mapped display and direct manipulation interaction style. It is a single-tasking system with a simple user model, although as Tanner (1987) observes, handling multiple inputs in a single-tasking environment brings its own problems. The Macintosh windowing system is built on top of a library of toolkit routines held in the ROM. The designers drew from the Star and Lisa experience but made some simplifying decisions while attempting to preserving adequate power for users. The hardware and software design supports rapid graphical interaction for pull-down menus, manipulation of windows and icons, graphics, and text editing.

The desktop

The Macintosh desktop is created and maintained through a system program called the Finder (Apple, 1989a). It is opened automatically when the Macintosh is turned on, and offers a range of commands and facilities for tailoring the desktop; for organising documents, folders, disks, and applications; and for shutting down and restarting the Macintosh. Version 6.1 is used in the examples throughout this thesis. The facilities of the Finder are utilised primarily through manipulation of a small group of standard components or objects which include pull-down menus, dialogue boxes, icons, and windows. These are examples of what Bass & Coutaz (1991) call 'syntactic computer objects', that is, objects which result from the use of computer technology and which are independent of the task domain. They are referred to throughout this thesis as 'syntactic' objects, their counterpart being 'semantic' objects, which model the task domain, e.g. text documents.

Icons are used in the Macintosh desktop environment as the basic representation of objects, which may be folders, disks, applications, or application documents. Icons can be created, renamed, copied, moved, and 'trashed' (deleted). Opening an icon instantiates an open window and provides access to the content of the object it represents. The standard Macintosh document window is made up of an application area and a number of control components contained in a frame area (Figure 2.1). All document windows for a given application share a single menu bar at the top of the display screen. Controls are provided in the window frame for immediate window control, that is, moving and sizing the window, and scrolling its contents. These controls are displayed only when a window is in a selected or 'active' state. The scrolling model is based on a moving window rather than moving data. The effect of scrolling is to change the visible portion of the object in the application area, permitting, for example, a text document to be browsed. Scrolling in both the horizontal and vertical axes is generally possible on the Macintosh, and is provided through scroll bars. At any time the position of a scroll box represents the position of the visible portion of the object in the application area relative to the entire object in that axis.

The opening and closing of windows, and manipulation of window content, can be undertaken through selections from the Finder and application *File* and *Edit* menus. Some Macintosh applications offer the user the option of being able to split a window into independently scrollable panes, thus providing for multiple concurrent views of objects in the application area.

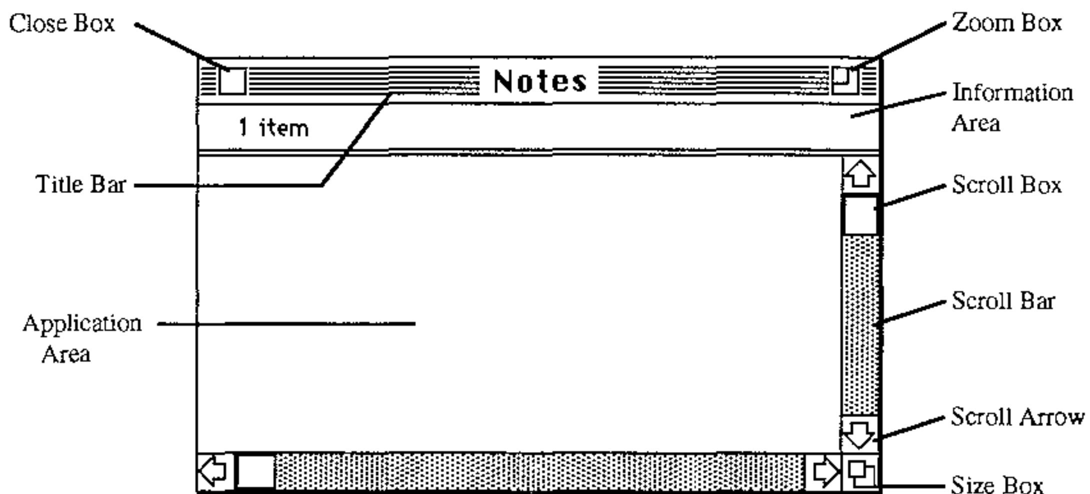


Figure 2.1: Macintosh document window (active state)

Part of a Macintosh desktop is shown in Figure 2.2. In this snapshot, a volume, *Cirrus 40*, and two folders, *Work*, and *Research*, are open. The highlighted *Close* command is about to be selected from the pull-down *File* menu. A single menu bar is provided at the top of the screen. *Research* is the active (currently selected) window, and will be the target of the *Close* command. The *Trash* folder icon is visible on the desktop, and other folder icons, e.g. *Teaching*, are visible within parent windows. Open windows are managed in the form of a stack exhibiting cyclic behaviour, with the active window at the top. Making a window active has three immediate consequences:

- The appearance of the window is altered. Its title bar is highlighted, and scroll bars and a size box appear.
- The window is 'moved' to the front plane of the desktop.
- The window has the 'keyboard focus', that is, it subsequently receives any keyboard input until another window is in turn made active.

Window focus and window depth are thus closely linked in the Macintosh model. Once a window is active, further selections are possible. In the context of the Finder these may involve either selection of window controls, or selection of one or more icons within the window area. Making a window inactive causes the visual changes

described above to be reversed, and any selected objects within the window to become deselected. It is possible to move an inactive window, but all other control commands are unavailable.

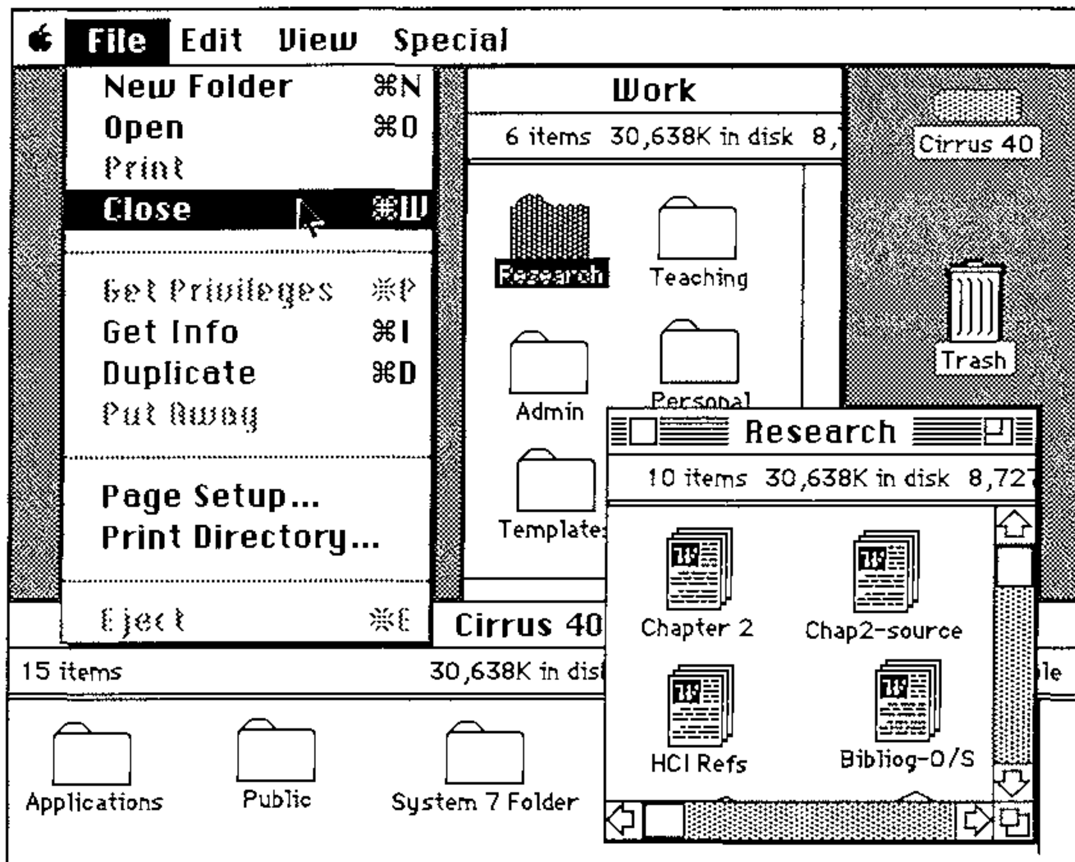


Figure 2.2: Example Macintosh desktop

Basic interaction techniques

The user interacts with the Macintosh through a single-button mouse and a keyboard, and works by selecting and moving objects, by selecting commands from menus, and by entering text and numeric values from the keyboard. The mouse is particularly well suited to accomplishing tasks that require spatial manipulation, such as menu navigation, and window sizing. The keyboard is more efficient for sequential tasks such as text entry. The interface is based on a 'single object-action' model - that is, at any time there is a single current object or group of objects on which a specified action, or group of actions, is carried out. A number of different mouse techniques are employed on the Macintosh to support this model. These techniques are summarised in Table 2.1.

Technique	Actions
Click	Press and quickly release the mouse button
Shift-click	Hold down the Shift key and click the mouse button
Double-click	Press and release the mouse button twice in quick succession
Press	Press and hold down the mouse button
Drag	Press and hold down the mouse button Move the mouse Release the mouse button

Table 2.1: Macintosh mouse techniques

Two cursors are employed on the Macintosh to display the 'current place':

- *a mouse cursor*: which represents the current mouse position, and which applies to all windows; and
- *a text cursor*: which represents the current keystroke position, and which applies to text windows only.

Together, these cursors determine the window, and therefore the 'client' application, to which input is directed.

Modality

'Modal subdialogues' are extensively employed on the Macintosh interface, primarily in order to cluster related operations and options. They are displayed in the form of double-bordered dialogue boxes, which provide both a visual and functional context. An example dialogue box from Microsoft Word is shown in Figure 2.3. This modal subdialogue permits printing options on a laser printer to be selected, no other aspects of the Word dialogue being available while this box is displayed. 'Check boxes' offer mutually compatible options, e.g. *Font Substitution* and *Text Smoothing*, whereas the 'radio buttons' provide mutually exclusive options, e.g. *US Letter* or *A4 Letter*. The subdialogue is terminated, and the dialogue box hidden, following selection of either the *OK* or *Cancel* options. Dialogue boxes can also offer text fields able to receive keyboard input.

Limited use is also made on the Macintosh of modeless dialogue boxes or 'floating windoids', which have a standard title bar with which they can be repositioned on the desktop.

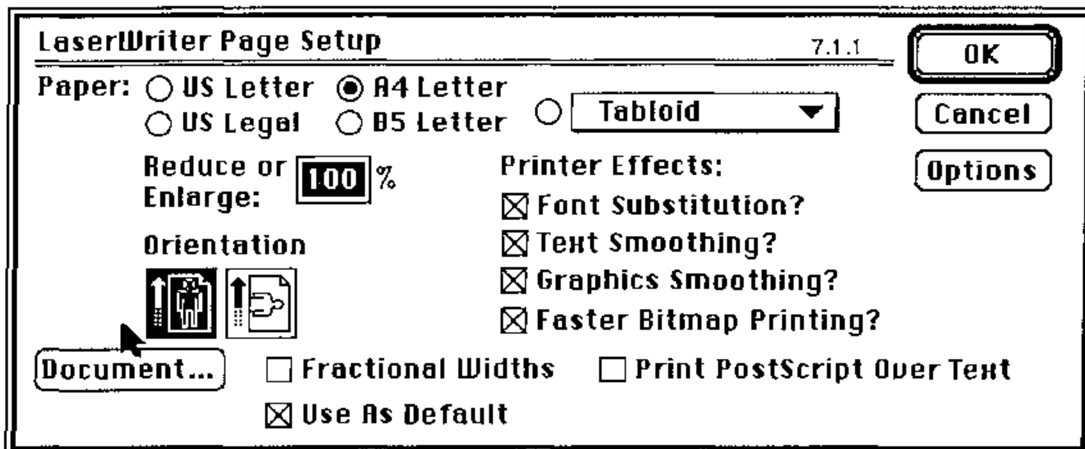


Figure 2.3: Example dialogue box (Microsoft Word)

The hierarchical file system

Document folders on the Macintosh provide for hierarchical organisation of documents and applications. Folders can be 'moved' into other folders, to 12 levels of nesting. The Finder provides a directory of the folders within each parent, which can be viewed in various formats once the parent folder is open. Figure 2.4 shows a small hierarchy of Macintosh folder icons and associated windows in which the directories are displayed 'by Icon'. Folder *A* represents the system disk. Folders *B* and *C*, which are on the desktop, contain respectively folders *F* and *G*, and *M* and *N*, and folder *F* contains folder *K*. *F* is the currently active window.

The complete document folder life cycle is supported by the Finder. The creation of a new folder, which is represented by an icon, also instantiates an associated window which is initially closed and unavailable for selection. Opening a folder makes available and selects the associated window. The icon remains available, and can be selected and moved following the opening of the window, when there are two representations of the same object on the display. An open window can be in an active (selected) or inactive (available) state. If another available document window is selected during the subsequent dialogue then the first window becomes deselected but remains available. If the physical state of the window is altered during the dialogue and the window closed, its state is saved and remains with the (closed) window. Closing a document window causes the next window on the stack (if any) to become active (the Macintosh does not permit a 'no listener' state). If a folder icon is trashed the associated window goes with it.

Icons and windows each exhibit a predefined behaviour. This behaviour is interlinked under certain conditions. For example, directly selecting a window (to make it active) causes the corresponding icon to be selected, and selecting an icon within an inactive parent window causes the parent to be selected (made the active window). The

Macintosh document folder system in general, and this example in particular, will be used in subsequent chapters in the development of the Lean Cuisine+ notation.

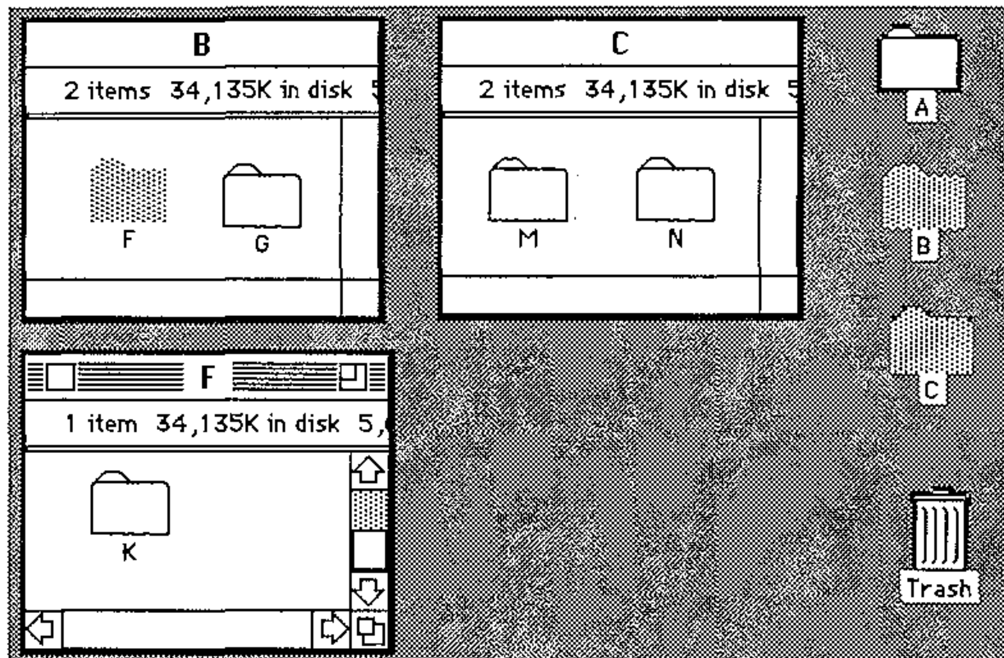


Figure 2.4: Example Macintosh document folder icon/window hierarchy

Other Macintosh applications

The Finder offers a dynamic environment involving instantiation of icons and windows, and provides a compact but relatively complex dialogue upon which to base the initial development of the Lean Cuisine+ notation in Chapter 5. Both the interaction tasks involved and the range of objects described are limited in scope however, with only syntactic objects involved. Two further classes of application are examined in order to broaden the scope and to introduce examples of semantic objects. These are employed both in Chapter 4 in the analysis of Macintosh interaction tasks, and in Chapter 6 in the further development of the notation, where the linkage between the Finder and other applications is also considered. These applications are:

- *A word processing application:* which introduces text documents as a class of semantic object. Aspects of Microsoft Word are considered.
- *Graphics applications:* which add graphical objects as classes of semantic objects, and also introduce genuine 'path' tasks in which the entire path taken by the mouse during an operation is significant. Aspects of both a bit-mapped package, in the form of MacPaint, and an object-oriented package, in the form of MacDraw, are

considered. Both applications provide palettes of tools, and both adhere to the Macintosh object-action model. MacPaint, in which the screen image is simply a collection of pixels, permits operations to be applied to selected *areas*, whereas MacDraw, in which the screen image is made up of a collection of discrete objects, provides for the selection and manipulation of these objects.

2.4 Modelling direct manipulation interfaces

Baecker (1980) stated that the development of interactive graphics systems was being hindered by the lack of an adequate conceptual framework for characterising graphical input and interaction. Since then, levels of abstraction have been identified in connection with interface design in the spirit of those used in data base design, where architectures involving several levels of abstraction are well established (Nijssen & Halpin, 1989). The three-schema 'ANSI/SPARC' architecture (Tsichritzis & Klug, 1978) has provided the basis for many of today's data base management systems. Top-down design methods such as CLG (Moran, 1981) and that of Foley and Van Dam (1984) are consistent with this view. The 'Seeheim model' (Green, 1985a), which is based on notions (semantics, syntax, lexicon) which are well understood by computer scientists, is perhaps the best known framework for the organisation of interface software. Green (1985b) states that the key to describing a user interface is the definition of a multi-layered model based on a number of conceptual levels in the spirit of data base schemas.

The frameworks of Moran, Foley & Van Dam, and Green differ in the number of levels proposed. Nevertheless there is significant correspondence between the approaches, with four levels of abstraction emerging:

1. *The user's view:* a high level functional interface description involving conceptual objects, their interrelationships, and the types of operations which can be performed on them. This corresponds with the 'task models' of Moran, and Green, and the 'conceptual level' of Foley and Van Dam.
2. *Functional level:* which is concerned with the *detailed* functionality of the dialogue in terms of objects and manipulations, and the information needed to carry out each operation. This corresponds with the 'semantic' components of Moran, and Foley and Van Dam, and the 'control model' of Green.
3. *Structural level:* which describes the structure and sequence of input and output aspects of the dialogue. This corresponds with the 'syntactic' components of Moran, and Foley and Van Dam, and (part of) the 'action model' of Green.
4. *Interaction level:* which is concerned with the physical actions of the user and display actions of the system. This corresponds with Moran's 'interaction' and

'physical' components, Foley & Van Dam's 'lexical' level, and (part of) the 'action model' of Green.

It should be noted that the boundaries between these levels are not hard, and that references to Moran's levels, Foley & Van Dam's components, and Green's models, represent a 'best fit' at each level. The first three levels are all concerned with the *underlying behaviour* of the interface at different levels of abstraction.

Three views of the interface

Despite the fact that they predate the widespread introduction of direct manipulation interfaces, the above levels are useful in identifying views for use in the development of direct manipulation interfaces. Three views emerge:

- *Conceptual view:* This is the user's mental view of the interface which defines the general form of the set of capabilities to be provided in terms of classes of objects, and processes involving user tasks and system feedback. It is essentially a functional view based on concepts familiar to the user e.g. the desktop metaphor of the Macintosh, and matches closely with level 1 above.
- *Behavioural view:* In direct manipulation interaction, where the user operates largely in the semantic plane, and where semantic and syntactic aspects of the interaction are closely interwoven, separation of the 'functional' and 'structural' levels identified above would be of doubtful value, even if it could be achieved. The 'behavioural view' represents the interaction in terms of objects being manipulated, user-generated input events, system-generated output events, interface states, and the relationships between these components (event sequences, object interactions etc.). The behavioural view is therefore concerned with both the semantics and syntax of a dialogue, and can be described using a 'dialogue model', as the latter is defined in this thesis.
- *Presentational view:* One or more 'presentational views' can be produced from this behavioural view - that is, an interface with a defined behaviour can be presented in more than one way. Different presentational views can be considered as members of a family, differing in presentation detail. Models to support this view are concerned with displayed objects, physical user actions and input methods (keyboard, mouse etc.), and the layout of the display. They describe the 'look and feel' of the interface. In both behavioural and presentational models, the objects available and user actions possible at any time may be constrained.

Through reference to the four levels identified above, these views have been indirectly matched with those of Green, Moran, and Foley and Van Dam. It is the behavioural model which is of interest in this thesis, although the point at which the

designer begins to make implementation decisions concerned with presentation may be difficult to pinpoint in what is a highly iterative process.

2.5 Software architectures for direct manipulation interfaces

In order to complete this examination of direct manipulation interfaces, issues relating to software architectures to support their implementation are briefly reviewed. Because of their special characteristics, direct manipulation interfaces do not fit well into some existing software architectures. The major architectural issues are:

- the desirability of separating management of the interface from the application, and the impact of this on semantic feedback; and
- the framework for the interface management component, particularly the levels of abstraction employed.

Rosenburg (1988) sees the key characteristic of direct manipulation interfaces as the rich feedback presented to the user. Dance, Granor, Hill, Hudson, Meads, Myers & Schulert (1987) observe that direct manipulation interfaces require that semantic information be used extensively for controlling feedback, generating default values, and error checking and recovery. This means that the application must constantly update the display to reflect the state of the application. Separation of the user interface from the application implies that semantic information must either be passed from the application via the UIMS to the user, or handled within the UIMS. The first approach places demands on the speed at which information must be passed between application and UIMS. The second approach is seen by some as inconsistent with the requirements of direct manipulation interfaces, in which syntax is hidden, and in which the user operates directly in the higher (semantic) problem domain.

In separating the interface from the application, the application and UIMS may thus require to communicate to an extent that brings the separation into question. Olsen (1987b, p.71) points out however that "to totally integrate the application code with the dialogue management is to forfeit some of the potential of a UIMS". Hartson (1989, p.65) states that "Achieving the ideal separation is neither always easy, nor in fact always possible". Bass & Coutaz (1991) identify the need for a high bandwidth between the dialogue controller and the application. An input may evolve in parallel with an output in order to provide immediate feedback, e.g. dragging an icon across the desktop involves a partial input expression until the mouse button is released (the user's action denotes the *Copy* command and a file name, but the recipient file name is undetermined until the mouse button is released). The system however is able to provide the user with semantic feedback in the form of an indication of potential recipients (presented in reverse video) as the icon is dragged.

The 'Seeheim' model (Green, 1985a) is based on a clear separation between an application and its interface, and defines three logical components that must appear in a UIMS: the presentation, dialogue control, and application interface components, which communicate by passing tokens. Partitioning an interface into sequential components, as characterised by the Seeheim model, can be seen as leading to poor semantic feedback, although both ALGAE (Flecchia & Bergeron, 1987), and the event language in the University of Alberta UIMS (Green, 1985c), are based on this model. Sibert *et al* (1986) consider that so-called 'linguistic' models encourage the designer to view each of the semantic, syntactic, and lexical levels in isolation, and have limitations in the areas of semantic error handling and feedback when used to specify direct manipulation interfaces. Hudson (1987) and Rhyne *et al* (1987) argue for an interface based on the notion of 'shared objects', with application objects being accessible to both the application and the dialogue controller.

Coutaz (1990) proposes the 'multi-agent' model as a way forward for direct manipulation interfaces. Cockton (1990b) sees this model as an improved view of the requirements of UIMSs, one which replaces the simple event-driven models of the late 1980s, which in turn replaced the linguistic models of the Seeheim vintage. This does not deny the principle of separation of an application from its interface, but goes further in applying the separation at various levels of abstraction, and distributing the responsibility for the dialogue among a set of co-operating agents. A number of multi-agent models and tools have been developed using object-oriented and event processing paradigms. Smalltalk's MVC (Goldberg & Robson, 1983) and PAC (Coutaz, 1990) are multi-agent models. GWUIMS (Sibert *et al*, 1986), Serpent (Bass, Hardy, Hoyt, Little & Seacord, 1988), and Sassafras (Hill, 1986) are run-time kernels and user interface generators based on multi-agent models.

2.6 Review

This chapter has identified both the class of interface under consideration in this research - the direct manipulation interface based on the desktop metaphor, and the particular current implementation of that class to be used in the development of the Lean Cuisine+ notation - the Apple Macintosh interface. Macintosh applications to be used in this development have been identified. The nature of the tasks performed at the Macintosh interface will be examined more closely in Chapter 4.

This chapter has also established a focus on models for supporting a behavioural view of the interface. The required scope of such a model will be determined through a review of existing dialogue models and notations in Chapter 3.

Chapter 3

A Review of Dialogue Models and Notations

" . . . the real problem is not just to find some way to describe the user interface (since, after all, assembly language can do that job), but to find a language that captures the user's view of a direct manipulation interface as perspicuously as possible. . ."

Jacob, 1986

The purpose of this chapter is to provide a context for the analysis of Chapter 4, and the development of Lean Cuisine+ which follows in subsequent chapters. Given the focus established in Chapters 1 and 2, of the development of a *graphical* representation for the *high level* design of direct manipulation interfaces, this review focuses on models and notations for specifying *underlying* system behaviour. In order to provide a broader context, Section 3.1 examines the related fields of system requirements specifications, and object-oriented analysis and design. Parallels are drawn between models and notations employed in these more general areas, and dialogue models. Section 3.2 reviews existing dialogue notations with reference to direct manipulation interfaces, and with emphasis on graphical representations. Some historical context is provided. In Section 3.3 these notations are compared and contrasted, and the detailed requirements of a dialogue model for describing interface behaviour are defined.

3.1 Specifying system behaviour

A variety of models at various levels of abstraction have been employed at the early stages of interface design. These include user models (Rich, 1983; Chin, 1989), task models (Annett, Duncan, Stammers & Gray, 1971; Moran, 1981; Card, Moran & Newell, 1983; Card *et al*, 1984; Johnson, Diaper & Long, 1984; Payne & Green, 1986; Johnson, Johnson, Waddington & Shouls, 1988; Markopoulos *et al*, 1992), and dialogue models, which are reviewed in Section 3.2. User and task models are human-oriented whereas dialogue models are computer-oriented.

There is considerable disagreement in the literature as to the distinction between user and task models. Kelly & Colgan (1992) define a task model as an abstract

representation of the procedures used, or knowledge required, to perform tasks interactively with a computer system. Task models are part of a number of methods for modelling user interactions, including CLG (Moran, 1981), GOMS (Card *et al*, 1983), TAG (Payne & Green, 1986), TKS (Johnson *et al*, 1988), and Adept (Markopoulos *et al*, 1992). All these methods employ a formal grammar to represent the task, i.e. the task is described using a special symbolic notation and associated rules. Task analysis is concerned with the analysis and decomposition of tasks, and their relationship with goals and procedures (Johnson, 1992). The methodology for constructing Lean Cuisine+ specifications described in Chapter 7 commences with a task decomposition. User models are primarily to do with the attributes of users, their abilities, etc. Kelly & Colgan consider that task models are subsumed within user models i.e., a user model is equivalent to a task model plus other information. Both user modelling and task modelling are major research areas in their own right.

Parallels between software design in general and interface software design in particular can be drawn. Davis (1988b) presents a taxonomy for the first three stages of the software life cycle, which are seen as: user needs analysis, solution space definition, and definition of external behaviour. These stages match closely with the first three phases of Shneiderman's interface development life cycle, outlined in Section 1.1. The function of the third stage system requirements specification (SRS) matches closely with that of a dialogue model in the early phase of interface design. It contains a complete description of *what* the software will do without describing *how* it will do it. It describes the expected *underlying* behaviour of the product to be built to solve the now understood problem. The behavioural requirements describe all inputs to and outputs from the system, as well as information concerning how the inputs and outputs interrelate (e.g. what sequences of inputs causes what outputs). Davis sees his third stage as a time to organise ideas, resolve conflicting views, and eliminate inconsistencies and ambiguities.

Davis (1988a) reviews and compares techniques and notations for use during the SRS phase of systems design - notations for describing the underlying behaviour of the system to be built. These include state transition networks (STNs), statecharts, and Petri nets, all of which are examined in Section 3.2 in the context of dialogue design. Davis considers that the understandability (of a SRS) appears to be inversely proportional to the level of complexity and formality present. Ideally, notations for use during the SRS phase should be understandable; should help organise the information in the SRS; should provide a basis for automated prototype generation (executable specifications); and should be able to serve as the basis for subsequent design and testing. Again a close match can be perceived with the general requirements of notations for use in the early stages of dialogue design, that is, notations for describing the *underlying* behaviour of human-computer interfaces.

Given the object-oriented nature of direct manipulation interfaces, which was established in Section 1.1, it is also fruitful to examine developments in the more general area of object-oriented analysis and design (OOAD). In evaluating current research in this area, Monarchi & Gretchen (1992) identify a need for methods and notations for the early phases of system development. They define two important approaches to OOAD as the 'combinative' approach, and the 'adaptive' approach. The combinative approach, which is the more complex, models separate views of a system using existing techniques, typically object-oriented, function-oriented, and dynamic-oriented views, which must then be integrated in some way. A major problem, given that these three views are generally independent of each other, is seen as the mapping of the data flow processes of the functional view, and the events and activities of the dynamic view with the objects in the object-oriented view. What is missing is a clear definition of what layers and views are important for representing and designing a system, and a means of integrating and balancing them. The adaptive approach, which is concerned with extending existing techniques to include object-orientation, may represent a better way forward.

In the context of methods and techniques, Monarchi & Gretchen attempt to distinguish object-oriented analysis (OOA) from object-oriented design (OOD), but concede that the distinction is a blurred one. OOA models the *problem* domain by identifying and specifying a set of semantic objects, including their structure, behaviour, and interrelationships. OOD, which precedes physical design, models the *solution* domain, which includes the semantic objects plus additional objects relating to the solution domain. These two classes of object match with the semantic and syntactic objects identified in Section 2.3. Solution domain objects include interface objects, which are not part of the problem domain but represent users' views of the semantic objects. Monarchi & Gretchen conclude that interface objects are rarely addressed in current methodologies, and that little is offered in the way of identifying or modelling interface objects.

3.2 Dialogue notations

A variety of notations have been proposed in recent years for representing dialogue models, including both textual and graphical representations. Some of these predate the introduction of direct manipulation interfaces. The emphasis in this review is on graphical notations suited to the early design phase of the interface life cycle, but other notations are briefly examined where they have something to contribute, and in order to provide some historical context. The review is intended to be representative, the aim being to identify the broad requirements of a notation for use in the high level design of direct manipulation interfaces, rather than to exhaustively compare and contrast existing notations (which would constitute a major research project in its own right).

Descriptions of a simplified version of the Finder document folder system are presented for each of the graphical notations, in order to facilitate analysis and discussion.

Green (1986) defines a dialogue model as an abstract model that is used to describe the *structure* of the dialogue between a user and an interactive computer system. In this thesis, structure is considered to be implicit in dialogue, and a dialogue model is viewed as an early design aid providing an abstract formal or semi-formal description of a dialogue, including its structure. Six & Voss (1990, p.383) view dialogue models as "an aid for the user interface designer to clarify and specify his ideas of the dialogue design". Myers (1989) observes that UIMSs can be classified according to the dialogue model they employ. Green distinguishes design notations, which can be informal, from implementation notations, which must be formal, and goes on to define three classes of dialogue model which can be used in the implementation of user interfaces - state transition networks (STNs), formal grammars, and event models. Green's taxonomy has been an influential one, but given his emphasis on notations for the *implementation* of user interfaces, it is less than ideal as a basis for the review which follows. Apperley & Spence (1989, p.58) conclude that (italics added) "they (the notations surveyed by Green) are better for describing *how* to achieve a particular behaviour rather than describing *what* the behaviour actually is".

Group	Notation	Event modelling
Sequence-based	STD	state-based event sequences
	EDG	task-based event sequences
	Petri net	state-based event sequences
State-based	Statechart	state-based event grouping
Object-based	Lean Cuisine	implicit and loosely object-based

Table 3.1: Classification of surveyed graphical dialogue notations

Dialogue control is concerned with reacting appropriately to the events which occur during the course of a dialogue. The five graphical notations included in the review which follows are classified according to the way in which they model events, as shown in Table 3.1. The three notations in the 'sequence-based' group model dialogue in terms of event sequences. The first two of these are based on the 'single event / single state' model, in which only a single state can be active at any time. The third, the Petri net,

provides for multiple active states, as do the notations in the other two groups. Statecharts feature alone in the 'state-based' category, in which events are grouped primarily around states, and Lean Cuisine (Apperley & Spence, 1989) is the only graphical notation in the third group, which is termed 'object-based' because events are grouped primarily around objects, with the notations being based to some extent on the object-oriented model. Other notations are briefly reviewed in connection with each group. The five graphical notations are compared through a simplified version of the Finder document folder system of Section 2.3, in which:

- for the purpose of selection, folder icons are considered to exhibit mutually exclusive behaviour, i.e., only one icon can be selected at any time;
- the *Trash* folder is ignored; and
- the creation and deletion of folders is omitted.

3.2.1 Sequence-based notations

State transition diagrams (STDs)

STDs are very well known in the broader context of a graphical notation for representing finite state machines, and over the past decade they and their derivatives have probably been the most widely used dialogue models (Parnas, 1969; Foley & Wallace, 1974; Kieras & Polson, 1983; Jacob, 1983, 1985, 1986; Wasserman, 1985; Koivunen & Mantyla, 1988). STDs basically consist of nodes, which represent states, interconnected by labelled directed arcs, which represent transitions between them. Each transition is triggered by an input event, and can have an associated system action (output event). STDs essentially show the state changes resulting from allowable sequences of input events. They differ in the way in which they represent user and system actions, and exist in several forms, including recursive and augmented. Recursive STDs provide for partitioning and increase the descriptive power by permitting sub-diagrams to call themselves. Augmented STDs extend the descriptive power still further by associating a set of registers (global variables) and a set of functions with the network. In the generalised transition networks (GTNs) of Kieras & Polson (1983), arcs are tested in a clockwise sequence, and if none is satisfied backtracking occurs.

Jacob (1985, p.52) considers that "One of the principal virtues of state diagram notations is that, by giving the transition rules for each state, they make explicit what the user can do at each point in a dialogue and what the effect will be". He goes on to define an extended STD model which he subsequently applies to the specification of direct manipulation interfaces (Jacob, 1986). Jacob views direct manipulation interfaces as a collection of many relatively simple individual dialogues, related to each other as a set of 'co-routines', each of which can be specified as a separate interaction object with an

independent dialogue description. The dialogue associated with each interaction object can be suspended and resumed with retained state, like a co-routine, with an executive to manage the overall flow of control. Given this structure, a direct manipulation interface is specified (p.289) "as a collection of individual, possibly mutually interacting objects... and the loci of remembered state in the dialogue".

The introduction of a structural hierarchy in order to overcome deficiencies in the STD model is not entirely satisfactory however, as it does not reduce the size of the description but merely fragments it, and fundamental shortcomings of STDs, when applied to asynchronous dialogues, remain. Myers (1989) sees STDs as largely unsuitable for the relatively modeless direct manipulation type of interface where the user has many choices at each point in a dialogue. Koiuvnen & Mantyla (1988) consider that STDs have significant shortcomings for dialogue description, especially the size of the STD for quite straightforward applications where the number of states can become large. Consequently the STD becomes difficult to define, comprehend or modify.

Apperley & Spence (1989) present a STD description of the underlying behaviour of a reduced MacWrite *Style* menu, reproduced in Figure 3.1, which clearly illustrates these shortcomings. In this menu, in which a valid choice must always be present, the *Plain Text* item is mutually exclusive with respect to each of the other items, whereas *Bold* and *Italic* are mutually compatible - either or both can be chosen from this pair. *Superscript* and *Subscript* form another mutually exclusive pair; only one of these two can be chosen, but that choice is compatible with the *Bold/Italic* pair. Each of the possible menu states (there are twelve in all) is represented by a state in the diagram, and each of the possible input actions in each state (there are five for each of the twelve states) is represented by an arc. States have been labelled using upper-case letters representing the selected options in each state, with *U* representing superscript, and *L* representing subscript. The corresponding lower-case letters are used to represent the selection of these options. The diagram has been simplified by using bi-directional arcs where possible. Apperley & Spence conclude that (p.54) "Although the diagram does completely and accurately describe the behaviour of this particular menu, it is neither simple to produce nor to understand, and its derivation from the known external behaviour of the menu is not obvious".

Figure 3.2 shows a STD for the Finder document folder system, which involves just four states: *no object selected*; *folder icon only selected* (the presumed start state); *window only selected*; and, *folder icon and window selected* (the pairing may involve the same or different folders). In the notation of this example, rounded rectangular areas rather than circles are employed to represent states (after statecharts (Harel, Pnueli, Schmidt & Sherman, 1987)). The labels on the arcs consist of two parts separated by a slash, to denote respectively the input events which trigger them, and any output (system) actions which are associated with them, and which may be conditional. Consider the

label *i/pw*(c_1): here the selection of an icon (input event *i*) triggers the selection by the system of the parent window (output action *pw*), resulting in a different *IWS* state instance, only if c_1 is true at the time, that is, if a parent window exists. This facility permits some dynamic links between folder icons and windows to be modelled.

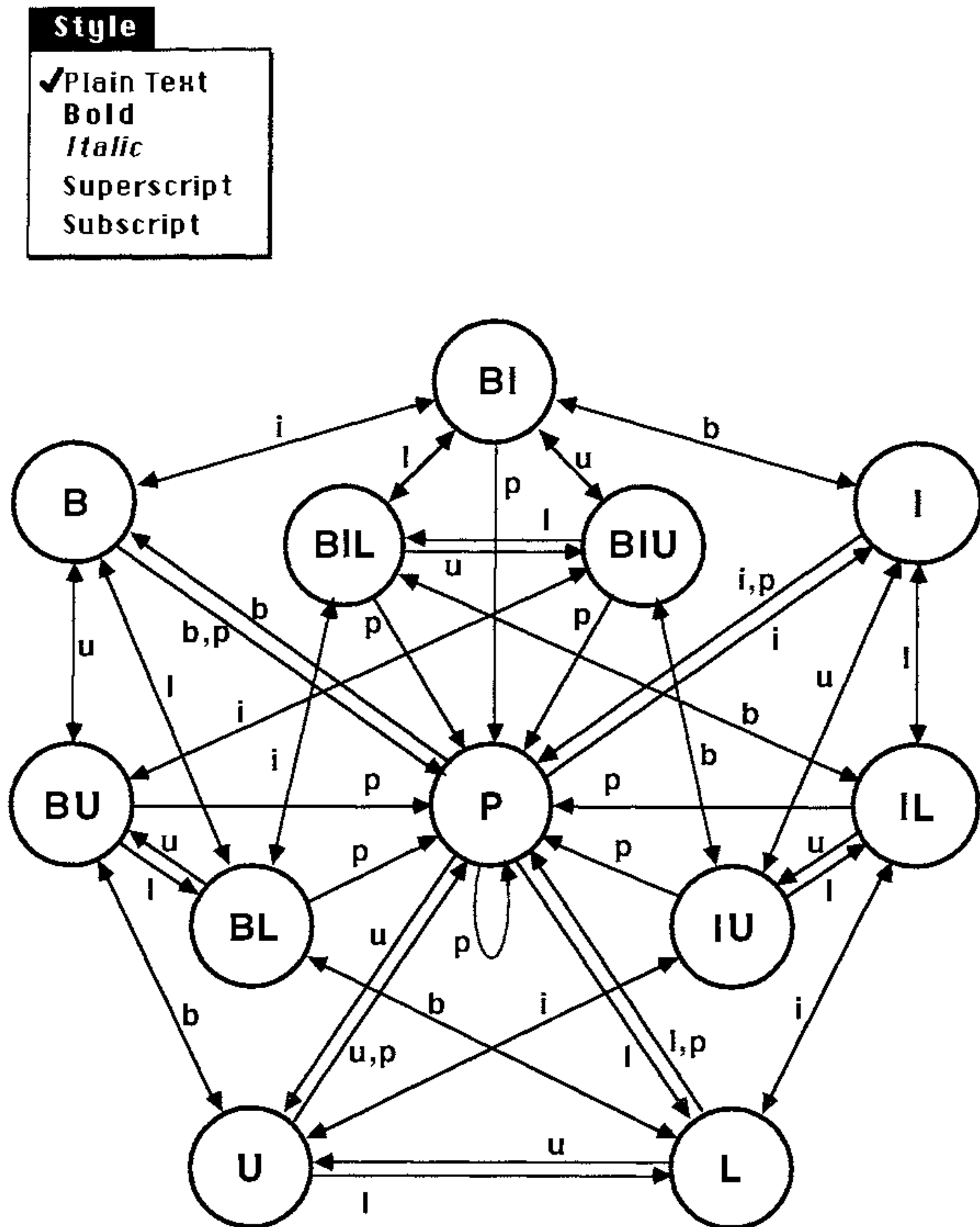
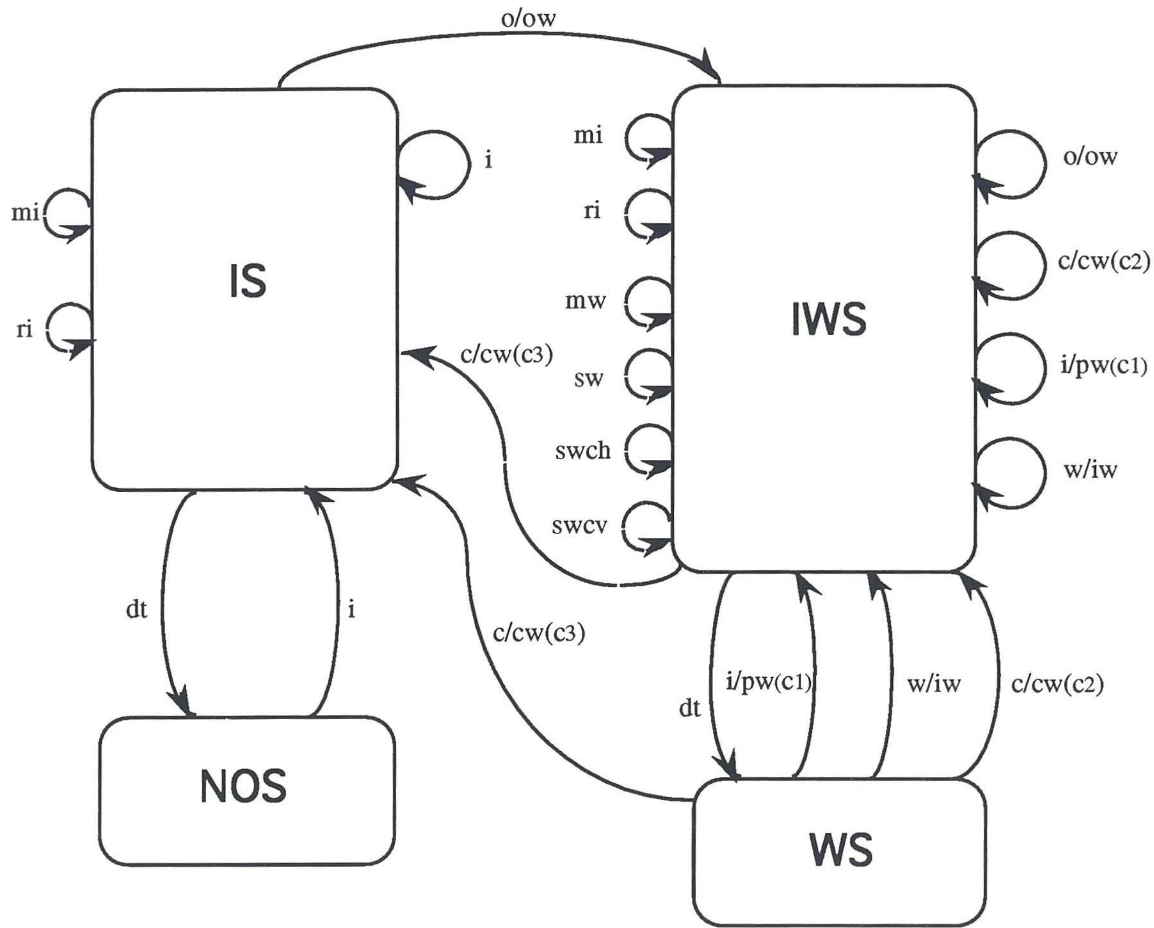


Figure 3.1: STD for MacWrite *Style* menu (Apperley & Spence, 1989)



Input events
c: selection of *Close (Window)*
dt : selection of position on desktop
i : selection of icon
mi : selection of *Move Icon*
mw : selection of *Move Window*
o : selection of *Open (Window)*
ri : selection of *Rename Icon*
sw : selection of *Size Window*
swch : selection of *Scroll Window Contents* (hor.)
swcv : selection of *Scroll Window Contents* (vert.)
w : selection of window

States:
IS : icon only selected*
IWS : icon and window selected
WS : window only selected
NOS : no object selected

* presumed start state

System actions
cw : selection of next window on stack
iw : selection of icon (to match window)
ow : addition of new window to stack
pw : selection of parent window

Conditions:
c1 : if parent window exists
c2 : if window stack > 1
c3 : if window stack = 1

Figure 3.2: STD for Finder document folder system

STDs provide an easily displayed and manipulated network representation based on a small number of constructs. Peterson (1977) observes that since the set of reachable

states is finite, it is possible to answer almost any question about a finite state model, and concludes that such a model therefore has very high 'decision power'. On the other hand he points out that the class of systems which can be modelled is severely limited, which means that such a model has very low 'modelling power'. STDs predate direct manipulation interfaces, and suffer from a number of shortcomings in this respect:

- They are based on the 'single event / single state' model, which can lead to an exponential growth in states and transitions in asynchronous dialogues, and therefore large and very complex diagrams. In particular they do not handle unexpected user actions well.
- They are inherently sequential in nature.
- Objects are represented only as the sum of their states, which means that objects and their behaviours are not well defined. Inter-object relationships are not represented at all.
- State representation is limited. In particular, STDs do not support state hierarchy or orthogonality.

Other dialogue models based on STDs include RAPID/USE (Wasserman, 1985), WindowNet (Grotchmann, 1987), AIM (Carey & Graham, 1987), which combines traditional use of STNs with more recent object-oriented techniques, and statecharts (Harel *et al*, 1987) which are examined in Section 3.2.2.

Event-decomposition graphs (EDGs)

In EDGs (Chakravarty & Kleyn, 1990), the description of the behaviour of a system is decomposed into temporal sequences of events and actions that are represented in an 'AND/OR' graph. Events are associated with particular task sequences. The notation is a derivative of the standard AND/OR graph used for representing problem space in the field of artificial intelligence, and is an attempt to address a difficulty in event-based systems of analysing and visualising the actual sequence of events that causes a particular action (describes a particular thread of functionality). AND/OR graphs have also been applied to describing goal substructures in the task analysis phase of interface analysis (Markopoulos *et al*, 1992).

In EDGs, all the events in a set of AND-linked nodes must occur in the order they are shown to complete the event of the parent node. Only one of the events of a set of OR-linked nodes need occur to complete the event of the parent node. The leaf nodes are associated with primitive interaction events e.g. opening a window, selecting a menu item. As a whole, an EDG represents the set of possible event sequences that accomplish the same task, where a task is some high level interaction such as scrolling a window, using a pop-up menu etc. An execution is the sequence of events of one particular instance of accomplishing the task.

EDGs, which are annotated with special nodes, boxes, and fonts, can be joined together, manipulated, and refined, to form more complex diagrams describing the behaviour of the system. AND and OR nodes are not themselves events, but an alternative to the more usual AND/OR graph arc notation for combining events, to permit easier on-screen editing. Cycles are indicated by repetition of nodes rather than links back, which preserves the tree structure but reduces the ability of EDGs to capture important dynamics visually. Leaf nodes in an EDG may represent: repetition of an existing label to indicate a cycle; primitive actions of the user (italic font) or the system (plain font); or may refer to a sequence of actions that are viewed as a primitive event at this level of refinement, and possibly described in another EDG.

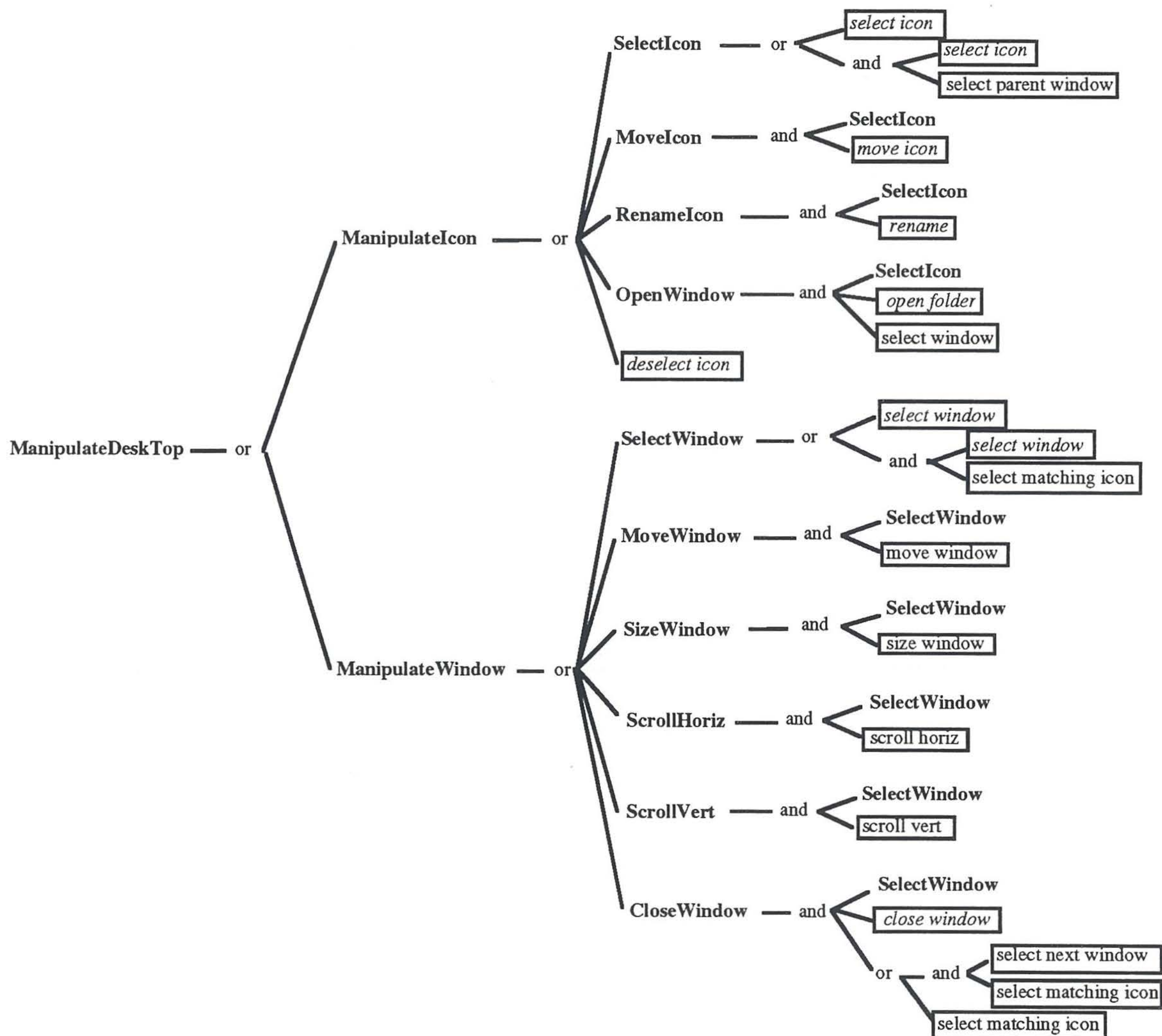


Figure 3.3: EDG for Finder document folder system

Figure 3.3 presents an EDG representation of the Finder document folder system. The event sequence for each higher level task, for example the opening of a window, requires a complete traverse of the tree. For this reason the notation fails to capture the asynchronous aspects of the dialogue, and would not lend itself to execution on any incremental basis. The notation does however provide a link between primitive task actions and higher level tasks. The representation involves much repetition at the lower levels in order to preserve the tree structure. While this is a laudable objective from the wider viewpoint of graphical notations (Harel, 1988), it could be achieved in other ways, for example by layering the notation.

The visual strengths of EDGs are seen by Chakravarty & Kleyn as in representing hierarchy and locality with a small number of constructs. On the debit side, EDGs provide only a limited (if useful) view of interface behaviour, and suffer from a number of shortcomings in the context of describing direct manipulation interfaces:

- They do not represent objects well. Inter-object relationships are not represented at all.
- They do not show states or substates.
- They do not show event preconditions.
- They do not handle interrupts/terminations well.

Petri nets

According to Peterson (1977) Petri nets were originally developed in response to the limited modelling power of finite state machines, in a search for more powerful methods for modelling and analysing the flow of information and control in discrete-event systems that may exhibit concurrent (parallel) activities. Petri nets are abstract virtual machines with a well defined behaviour, which have been used to specify process synchrony during the design phase of time-critical applications. Recent interest has been shown in their application during the system requirements phase (Davis, 1988a). Like EDGs, Petri nets define possible sequences of events in a modelled system. Additionally they show the system states (combinations of conditions) which result from the occurrence of these events. A Petri net is said to properly model a system if every sequence of actions in the modelled system is possible in the Petri net, and every sequence of actions in the Petri net represents a possible sequence in the modelled system.

Petri nets are represented as network graphs comprising two types of nodes: circles (places) and bars (transitions) connected by directed arcs. Petri net graphs model the static properties of a system, but also have dynamic properties that result from their execution. Places and transitions may be labelled to indicate the intent of the model, but Petri nets are uninterpreted models. During execution, tokens (represented by black

dots) move from place to place by the firing of the transitions of the net. A transition fires when: a) a clock pulse has arrived (the net is synchronised), and b) all arrows entering that transition come from places with tokens. In the case of a fan-out, the token will be replicated and a copy will follow every available path. The distribution of tokens at any time defines the state of the net and is called its 'marking'. In different markings of a net, different transitions may be enabled.

In the modelling of systems, places represent conditions, and transitions represent events. For an event to occur, it may be necessary for certain conditions to hold. These are referred to as its 'preconditions'. The occurrence of the event may cause the preconditions to cease to hold and may cause other conditions, 'postconditions', to become true. The inputs of a transition are the preconditions of the corresponding event; the outputs are the postconditions. The occurrence of an event corresponds to the 'firing' of the corresponding transition. The order of occurrence of events is one of possibly many allowed by the basic structure, reflecting the fact that in real life situations, in which several things are happening, the apparent order of occurrence of events is not unique, but rather any of a set of sequences may occur. Many of the properties of a Petri net are based on the properties of its 'reachability set', the set of reachable states. This can be expressed as a 'reachability tree' - a tree whose nodes represent markings of the Petri net and whose arcs represent the possible changes in state resulting from the firings of transitions. A reachability tree thus represents the entire 'state space' of the net. A path from the root (initial marking) to a node in the tree corresponds to an execution sequence.

Figure 3.4 shows a Petri net representation of the Finder document folder system. As with STDs and EDGs, the mutually exclusive behaviour of the folder icon and window dynamic instance sets has to be assumed (it is not obvious, for example, that the selection of a window to make it active causes deselection of any previously active window). The capturing of the constraints governing the interaction between icons and windows, for example those governing the selection by the system of a parent window for a user selected icon, appears clumsy. There would seem to be a need for extensions to the model to make it easier to use, for example a switch mechanism which could lead to alternative markings depending on its state. Peterson (1981) reports that Petri nets have been criticised as being too simple and limited to model real systems, and that extensions in the areas of constraints and switches have been proposed.

Mappings between Petri nets and other models have been defined (Peterson, 1981). Finite state machines are Petri nets which are restricted so that each transition has exactly one input and one output. A STD can be converted to a Petri net by simply making each state a place, and each state transition a Petri net transition. This underlines the superior modelling power of Petri nets, which is a function of their ability to support multiple active states. Davis (1988a) however argues that Petri nets are difficult to understand for non-computer experts.

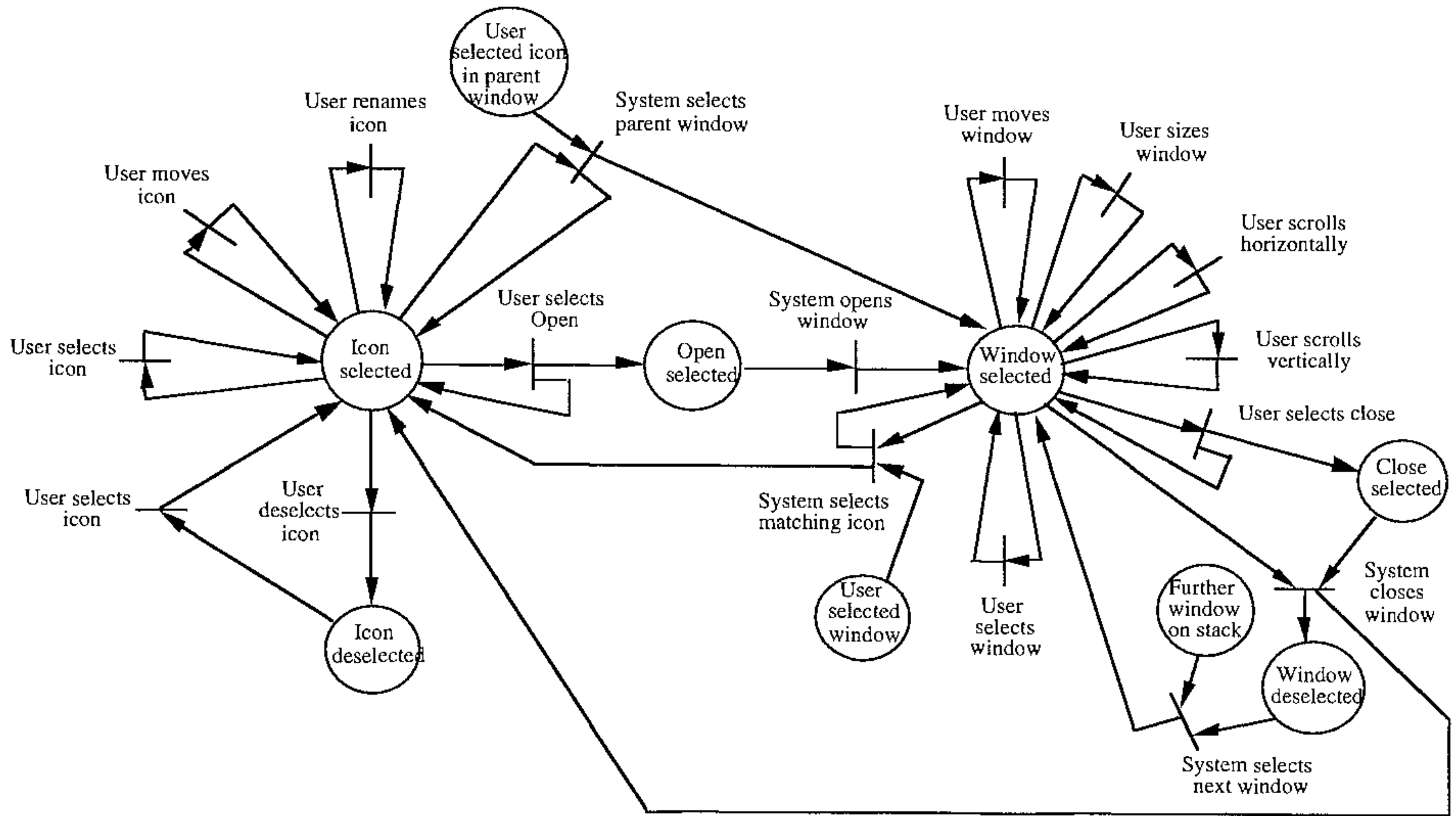


Figure 3.4: Petri net for Finder document folder system

Shortcomings of Petri nets are highlighted by the Finder example, in the context of describing direct manipulation interfaces:

- Objects are represented only indirectly by the sum of their states, which means that objects and their behaviours are not well defined. Inter-object relationships are not represented at all.
- The need to show all transitions explicitly clutters and detracts from the diagrams. This is especially problematic where a number of transitions relate to a particular object state, e.g. *window selected* in Figure 3.4.
- The latter problem is compounded to an extent by the requirement to represent separately, selected and deselected object states, which leads to additional (duplicate) transitions.
- Substates are not supported.

van Biljon (1988) examines Petri nets as a candidate for modelling dialogues. Labelled Petri nets are extended to nested Petri nets, allowing transitions to invoke subnets. The use of the extended notation is demonstrated by modelling a small hypothetical command language. The extensions proposed are not original, except in the context of the application of Petri nets to command language specification. There is no consideration of the suitability of the Petri net graph as a design representation of a dialogue.

Other sequence-based notations

Formal grammars

Formal grammars, which have been applied chiefly to the description of the syntax of languages, including programming languages, have also been used in the modelling of interactive dialogues (Wasserman, 1981; Reisner, 1981, 1982; Shneiderman, 1982; Bleser & Foley, 1982; Jacob, 1983). The best known form is Backus-Naur form (BNF) which is rule-based and produces a hierarchical description. SYNGRAPH (Olsen & Dempsey, 1983) is an example of a BNF-based UIMS. In the context of describing dialogues, grammars are a compromise as they describe only one half of the human-computer interaction - the syntax of the language employed by the user to communicate with the computer - and show the permissible user actions at each stage in a dialogue. In the context of the research reported in this thesis they suffer from the added disadvantage that they lack a graphical counterpart.

Grammars, like STDs, predate the dialogue control problem and are a 'borrowed' model. Jacob (1983) sees STDs and grammars as formally equivalent, but points out that their surface differences have an important effect on the comprehensibility of specifications, and argues that STDs capture the 'surface structure' of the user interface

more clearly, a view supported by Moran (1981). As with STDs, enhancements are needed in order to define the responses of the computer. Grammars also suffer from the shortcomings identified above for STDs, and although sequence-based do not show sequence well. Hix & Hartson (1986) go further, and argue that neither grammars nor STDs are models of interaction, because they are not descriptive of the general *structure* of interaction. A model of interaction should identify the generic constituents of a dialogue and the structural relationships between them.

Jackson structured design (JSD)

JSD is an entity life history oriented method which has been applied to the specification of human-computer interaction (Sutcliffe & Wang, 1991). JSD makes extensive use of structure diagrams (well known in computer science) which describe event sequences in terms of three control primitives: sequence, selection, and iteration, thereby expressing a process structure which becomes the template for program design. JSD attaches considerable importance to time ordering in specifications, and describes systems as an asynchronous network of concurrent processes. The major modelling notation of JSD, process structure diagrams (PSDs), are used to specify tasks in terms of time-ordered sequences of actions. Each action is assigned as a system or user action, or a joint action.

PSDs are said to be formally equivalent to STDs even though they record events rather than states. STDs are seen as an alternative design method or cross-check. In common with STDs however, PSDs are a 'borrowed' notation not designed for dialogue specification, and are sequence-based (hence their equivalence to STDs).

3.2.2 State-based notations

Statecharts

Statecharts (Harel *et al*, 1987) are a higraph-based extension of the STD formalism and are state-based. Areas represent states and labelled directed arcs represent transitions. They have been developed for describing the behaviour of complex reactive systems. Harel (1988) defines the behaviour of a reactive system as the set of allowed sequences of input and output events, conditions, and actions, although in statecharts event sequence is implicit, rather than explicit as in the case of STDs. A system is described in terms of the state changes resulting from events, with events grouped in connection with change of state. The concepts that statecharts add to basic STDs provide expressive power and reduce the size of the resulting specification. The extensions are:

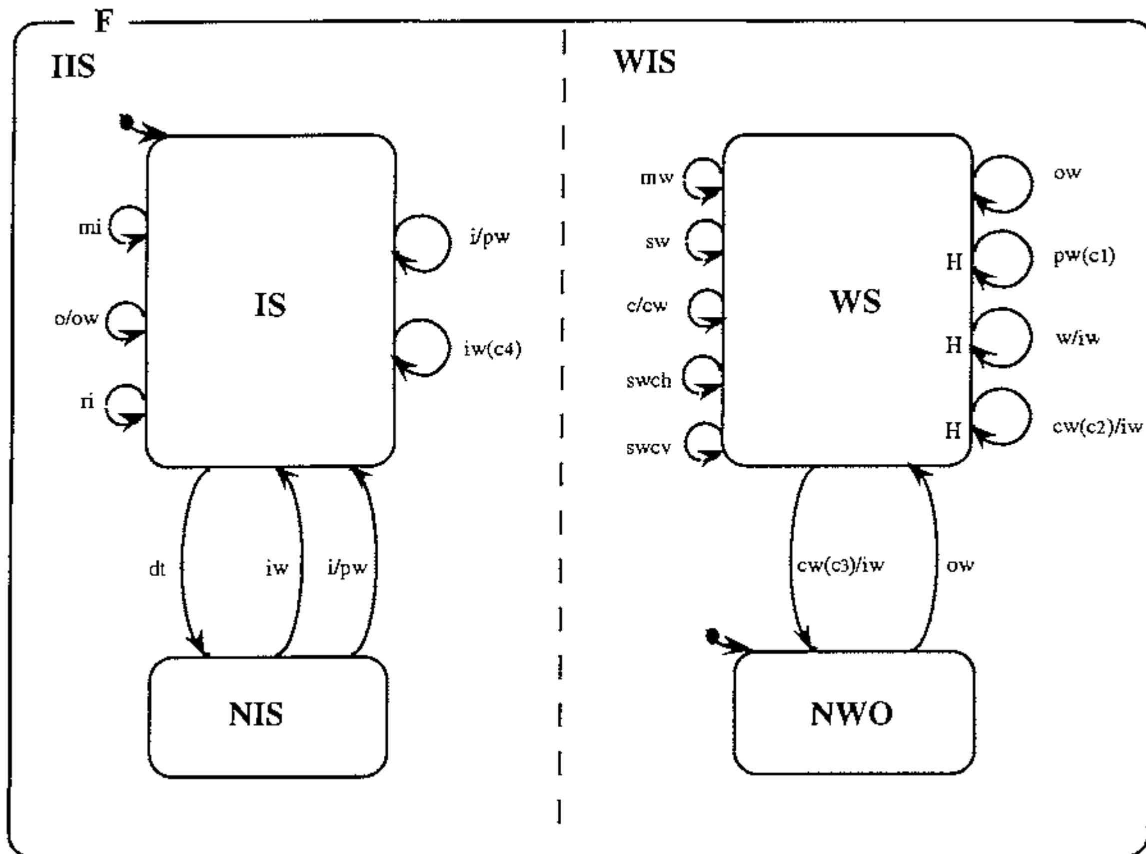
- *hierarchy*, the ability to group states into superstates to any level of nesting (which provides a basis for iterative refinement);
- *orthogonality*, the ability to describe the transition of several independent states concurrently; and

- *communication*, the ability to trigger several transitions by broadcasting an event to several states.

As a result, statecharts essentially consist of intercommunicating orthogonal STDs. A graphical notation based on enclosing areas, combining Euler/Venn diagrams and hypergraphs, is used for presentation of the formalism, together with the event/action notation of STDs. Hierarchy, which is introduced through an XOR construct, allows common transitions from states to be clustered and provides for default entries into substates. Statecharts can 'remember' a previous visit to a state, and provide for a ‘history’ entrance to a state. Concurrency is represented by an AND decomposition of states, and is modelled via output events, which can be attached as actions to triggering events along a transition. Actions are not merely sent to the 'outside world' but can affect the behaviour of the statechart itself in orthogonal components - this is achieved via a ‘broadcast’ mechanism and can lead to chain reactions of any length. A transition originating at the boundary of a state (set) applies to all substates in the set. Transitions are also permitted to merge and split.

The diagram in Figure 3.5 is a statechart representation of the Finder document folder system. The diagram incorporates Harel’s extensions to traditional STDs as outlined above, and also the rounded rectangular areas to represent states, introduced in connection with STDs in figure 3.2. The following notes relate to Figure 3.5:

- The default (or start) states, *IS* and *NWO*, are each denoted by a short arrow.
- The current *F (Finder)* state is made up of two orthogonal substates, *IIS* and *WIS*, related by AND. These states, and their substates, can exist concurrently, and are therefore mutually compatible. At any time, the state of *IIS* represents the state of the dynamic folder icon instance set, and the state of *WIS* represents the state of the dynamic window instance set.
- Both the *IIS* and *WIS* states are examples of XOR decompositions. *IIS* reflects the fact that at any time either a folder icon (from the available set of icons) is selected, or no folder icon is selected. *WIS* reflects the fact that at any time either a window (from the available set of windows) is active, or no window is open.
- Actions can be broadcast to orthogonal components. The dynamic links between icons and windows have been modelled using this facility. *pw*, for example, which originates in state *IS* and is triggered by input event *i*, is broadcast to state *WS*.
- A further feature of statecharts is their ability to remember a previous visit to a state. Three 'history' entrances (denoted by 'H') are shown into the *WS* state. If an already open window is re-entered either by direct selection, or through selection of one of its 'child' folder icons, or through closing the window immediately above it on the stack, its substates will be as they were in the most recent visit to it.



Events
c : selection of *Close (window)*
cw : selection of next window on stack *(Close)*
dt : selection of position on desktop
i : selection of icon
iw : selection of icon (to match window)
mi : selection of *Move Icon*
mw : selection of *Move Window*
o : selection of *Open (folder)*
ow : addition of new window to stack *(Open)*
pw : selection of parent window
ri : selection of *Rename Icon*
sw : selection of *Size Window*
swch : selection of *Scroll Window Contents* (hor.)
swcv : selection of *Scroll Window Contents* (vert.)
w : selection of window

States:
IS : icon selected
NIS : no icon selected
WS : window selected
NWO : no window open (selected)
IIS : state of icon instance set
WIS : state of window instance set
F : state of Finder window system

Conditions:
c1 : if parent window exists
c2 : if window stack > 1
c3 : if window stack = 1
c4 : if matching icon not selected

H : denotes 'history' entrance

Figure 3.5: Statechart for Finder document folder system

It is instructive to compare the STD and statechart representations of the simplified Finder document folder system in figures 3.2 and 3.5:

- The four generic Finder states identified earlier are represented differently, but can be shown to be equivalent (Table 3.2).

	STDs (Fig. 3.2)	statecharts (Fig. 3.5)
no object selected	NOS	NIS + NWO
folder icon only selected	IS	IS + NWO
window only selected	WS	WS + NIS
folder icon and window selected	IWS	IS + WS

Table 3.2: Finder states: STDs vs. statecharts

- Three further states appear in Figure 3.5: *F*, the current Finder state (which was implicit in Figure 3.2) and *IIS* and *WIS*. The latter two represent the current states of the icon and window dynamic object instance sets and provide stronger identities for these objects, although objects per se are not directly represented.
- The statechart representation contains significantly fewer transitions. This, coupled with the stronger object identity, provides a richer and clearer description of the behaviour of the system.

The origins of statecharts in STDs is both their strength and weakness. Their strength is that they describe clearly object states and transitions. Statecharts handle output events well, particularly with regard to scope, and make it clear which event triggers a transition, which objects are involved, and what the 'knock-on' effect may be. Thus they capture well the dynamics of the interaction. However, some shortcomings of statecharts in the context of describing the behaviour of direct manipulation interfaces are highlighted by this example. With the exception of substate representation, these compare closely with those identified earlier for Petri nets:

- In the example above, the states *IIS* and *WIS* provide some object identity by representing respectively the current states of the icon and window dynamic object instance sets, but objects are represented only as the sum of their states. The mutually exclusive behaviour of these object instance sets is entirely absent. The problem is compounded by the fact that this set behaviour leads to some implicit events. For example, *pw*, which explicitly triggers the selection of a parent window for a selected icon, also implicitly deselects the previously active window by virtue of the mutual exclusivity of window instances. Coleman, Hayes & Bear (1992) extend statecharts to support description of the life cycle behaviour of object classes. The states of the statechart represent the various stages that an object of a class may go through. Objects continue however to be represented by the sum of their states.

- The need to show transitions explicitly remains. This continues to pose problems despite the improvement over conventional STDs provided by the additional constructs. This can be particularly troublesome at the *intra-object* level where the need to show all transitions can clutter and detract from the diagrams.
- The latter problem is compounded to an extent by the requirement for 'null' states to be shown, e.g. *NIS* (no icon selected) in Figure 3.5, which in turn leads to additional transitions.

In reviewing statecharts, Chakravarty & Kleyn (1990) argue that events and states are not completely orthogonal, and that grouping together states does not necessarily mean that events among the states will not interfere with one another. Davies (1988a) sees statecharts as semantically rich and difficult for non-computer scientists. In the context of dynamic interfaces, such as that described above, the statechart notation would need to be further enriched to support mutually exclusive or mutually compatible generic states, an extension proposed by Phillips (1991). Wellner (1989) describes the implementation of a UIMS called Statemaster based on statecharts.

3.2.3 Object-based notations

Lean Cuisine

Lean Cuisine (Apperley and Spence, 1989) is a graphical notation based on the use of tree diagrams to describe systems of menus. A menu is viewed as a set of selectable representations, called 'menemes', of actions, parameters, objects, states and other attributes, in which selections may be logically related or constrained. A meneme is defined as having just two possible states, 'selected' and 'not selected', and is bistable under direct excitation. Its state may be changed either by direct excitation, or by indirect modification (that is to say as a result of the excitation of another meneme).

Menemes may be 'real' or 'virtual'. Associated with each real meneme is an implicit event which may trigger its selection or deselection. Real terminal (leaf) menemes represent specific actions or parameters. Real non-terminal menemes are headers to other menus, i.e. they refer to, and invoke, other menus. They are available for selection, and may be in a selected state regardless of the state of menemes within the menu they head. They do *not* become selected if selections are made in the menu which they head, and thus no propagation in either direction is implied. Virtual non-terminal menemes are used to partition a single menu, and are not directly accessible to the user. They can be indirectly selected, as a result of a valid selection having been made in the menu subsection they represent, thus permitting upwards propagation of selection. They may also be deselected as a result of the selection of a mutually exclusive meneme (see below), in which case any selected menemes in their subtree become deselected. Downwards propagation through a virtual meneme is thus also possible.

Lean Cuisine provides for the specification of certain constraints which impact meneme behaviour. Within a menu, menemes are grouped into subsets that are either mutually exclusive (1-from-N) or mutually compatible (M-from-N). Examples of these groupings and their graphical representation appear in Figure 3.6, which shows a Lean Cuisine representation of the simplified MacWrite *Style* menu introduced in Figure 3.1. Two virtual menemes, *Fancy Text* and *Indexed*, have been introduced. In the first branch, *Style* offers a mutually exclusive choice between *Plain Text* and the virtual meneme *{Fancy Text}*. This virtual meneme in turn offers the mutually compatible group of *Bold, Italic,* and *{Indexed}*, the latter another virtual meneme offering a mutually exclusive choice between *Superscript* and *Subscript*. *Style* is flagged as heading a 'required choice' group (§) in which *Plain Text* is the 'default choice' (*). Various other meneme modifiers are provided, and the default meneme type may be explicitly overridden by specifying one of the other types, which are denoted by means of special symbols. A definition of the Lean Cuisine notation appears in Appendix A.

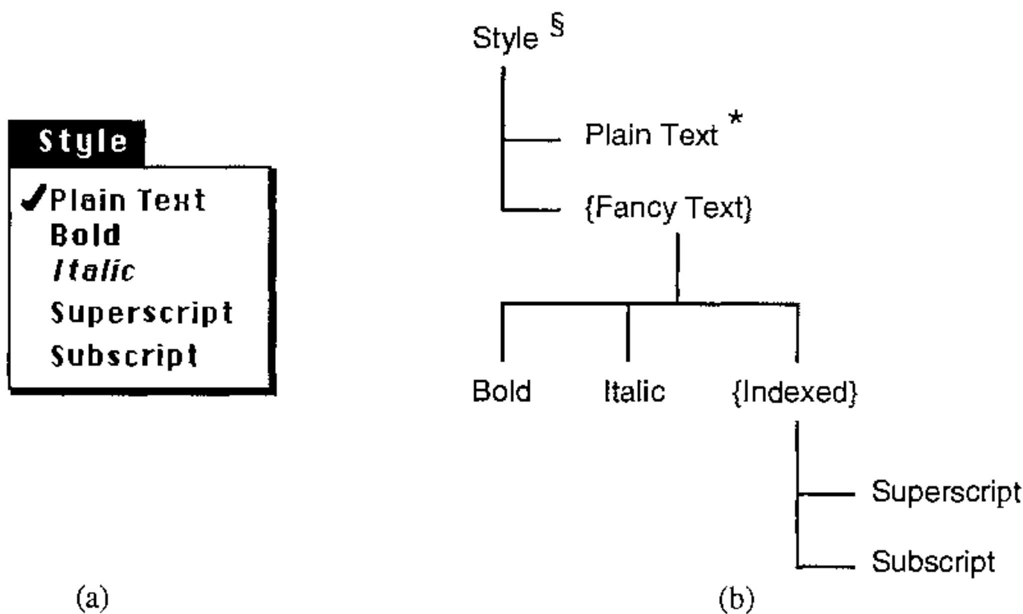


Figure 3.6: Lean Cuisine in action, showing
(a) the example MacWrite *Style* menu and
(b) its Lean Cuisine description

Lean Cuisine has been developed and applied in the context of menu systems. A menu is essentially a static object (although it is possible to think of examples which exhibit some dynamic properties e.g. a menu of document names). Other aspects of direct manipulation interfaces are intrinsically dynamic, e.g. the manipulation of windows and icons, and involve multiple states, conditions, and object interactions. The basic building block of Lean Cuisine, the meneme, although defined in the context of menus

may not be inappropriate for the description of other direct manipulation interactions. Figure 3.7 shows a Lean Cuisine representation of aspects of the simplified Finder document folder system.

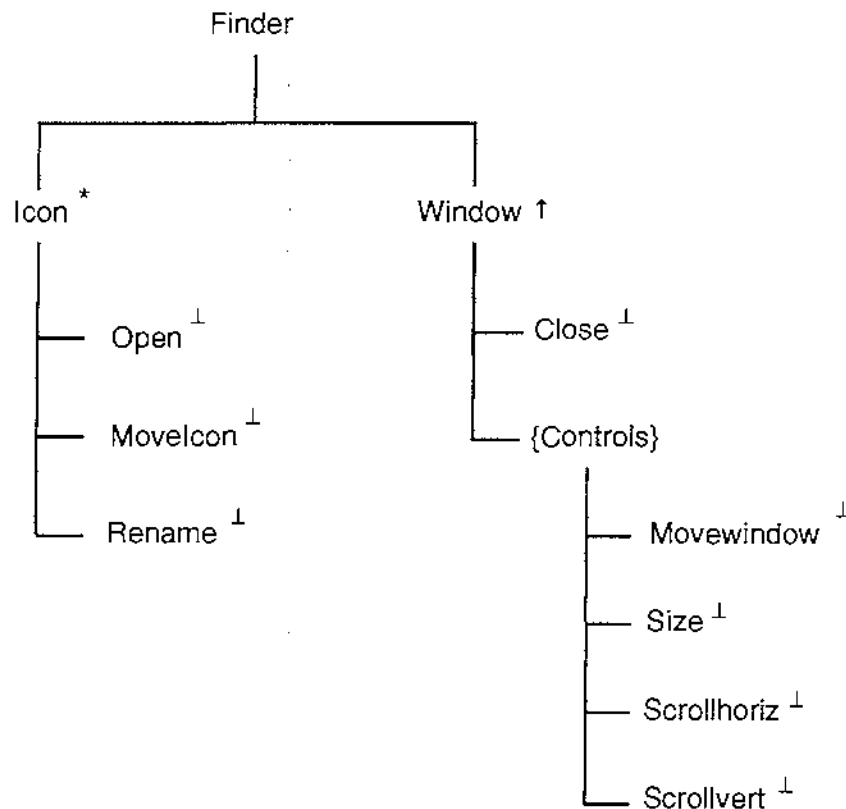


Figure 3.7: Lean Cuisine diagram for Finder document folder system

The *Icon* and *Window* objects are shown as being mutually compatible. Further mutually exclusive selectable options are then grouped beneath each of these dialogue objects. Non-terminal menemes act as subdialogue headers which are either real and selectable, as in the cases of *Icon* and *Window*, or virtual and unavailable for selection, as in the case of *{Controls}* which is used to further partition the available options in the *Window* subdialogue. Two further meneme modifiers are introduced: *Window* is shown as being 'select-only' (↑), to reflect the fact that an active window cannot be directly deselected, and several operations are represented by 'monostable' menemes (⊥), which capture the fact that they are of finite duration. Monostable menemes revert to an unselected state on completion of the operation represented by their selection.

The diagram fails to describe other interrelationships which exist between folder icons and windows (for example the fact that the selection of an icon could trigger the selection of a parent window) and does not represent the existence or behaviour of the icon and window dynamic object instance sets. In common with the other graphical notations considered, it would have to be assumed that an event associated with an icon

or window had an associated parameter denoting the instance. At present a Lean Cuisine diagram is executed only in the mind of the menu designer, and is not concerned with *how* the menemes are selected or with the dynamics of menu use.

In summary, Lean Cuisine offers a clear, concise, and compact graphical notation for describing asynchronous aspects of menu-based dialogues, but is not able to describe those aspects which include sequence. Events are not directly represented but are implicitly associated with selectable representations. Objects are directly represented but not visually distinguished from other elements, and the grouping of actions (and therefore events) is around objects. Execution of a Lean Cuisine diagram leads to a succession of permissible *states*, rather than permissible event sequences, underlining the essential nature of Lean Cuisine which shows the state changes resulting from object behaviour as (implicitly defined) events are applied to them.

Major extensions are required in order to support description of the *dynamics* of wider aspects of direct manipulation interfaces. These include:

- Extending the description of objects to include the definition of object types and type hierarchies, dynamic object instance set behaviour, and other inter-object relationships outside the present limited scope.
- Adding a mechanism for linking input and output events.
- Capturing a wider range of event preconditions.
- Incorporating a means of describing event sequences, and of relating event sequences to higher level functionality.

Apperley & Spence note that during a dialogue a meneme may be disabled (become unavailable for selection) and treat this as an 'apparent state' which is part of the presentation detail. Cockton (1990a, p.212) disputes this, arguing that this is not part of presentation detail but part of the interface dynamics reflecting "the changing state of the underlying application in response to user actions". Cockton views Lean Cuisine as a restricted form of object modelling capable of describing certain interaction-specific constraints (default values, mandatory selections) but modelling only a part of the functionality of an interactive system. He sees Lean Cuisine as suited to very early design with the advantage of forcing the designer to explicitly represent a model before continuing with the detail which will connect the user with the model. Edmondson & Spence (1992) consider that Lean Cuisine has a visual impact which renders it a powerful tool for early creative thought about menus. They present an account of a framework, Systematic Menu Design (SMD), for the design of menu-based interfaces which employs a framework of notational formalisms including Lean Cuisine, User Action Notation (Siochi & Hartson, 1989), and Event Response Systems (Hill, 1987), and which comprises both static and dynamic components.

Other object-based notations

Event response systems (ERSs)

The ERS (Hill, 1987) has been proposed as a model for removing the 'single event / single state' limits associated with STNs. The development of ERSs was influenced by CSP (which is described in the next section). Their basic structure was inspired by production systems (Hopgood & Duce, 1980), in which rules are fired when a condition is satisfied, and in which there is an absence of sequence. ERSs represent a major extension of this work, and form the basis of a technique for implementing direct manipulation interfaces in which events are processed by event handlers. ERSs show the state changes resulting from user events, and are considered to be loosely based on the object-oriented approach.

The main elements of ERSs are incoming events and flags. Flags are local (usually Boolean) variables used to encode the state of the system, and control execution. An ERS specification consists of a list of rules, each of which specifies a response to some external event (regular rule) or an action to be taken when some state is entered (ε-rule). Regular rules are of the form:

event: condition ⟶ action

where both the condition and the action are lists of flags. The event is omitted in the case of an ε-rule. A rule is fireable when all flags in the condition are on, and the event (if any) is at the head of the input queue. The response rules essentially say that given this event, and this set of flags (state), then we have a new set of flags (new state). The execution of an ERS involves a cycle which first processes the ε-rules until there are no longer any true conditions, then waits for an event and simultaneously processes all of the regular rules for which this is the event and for which the condition components are true. This cycle is repeated continuously.

Because of the rule-based nature of ERSs it is difficult to extract from a specification the *sequence* of actions which make up a specific operation, and ERSs are therefore not appropriate for interfaces with complex syntaxes. EDGs (Chakravarty & Kleyn, 1990), reviewed earlier, were proposed as a solution to this problem. Olsen (1990) points out that ERSs do not show which inputs are acceptable at a given point in a dialogue. ERSs also lack a graphical notation, but Anderson & Apperley (1990) have shown that a Lean Cuisine menu specification can be mapped into a modified version of ERS, which thus provides a path to the prototyping and implementation of interfaces described using Lean Cuisine, an issue developed further in Chapter 8. ERL, an ERS derivative which adds output events and a broadcasting scheme to transmit events, forms the basis of the Sassafras UIMS (Hill, 1986).

A number of other event-based models have been applied to the implementation of interactive dialogues (Borufka, Kuhlmann & Ten Hagen, 1982; van den Bos, Plasmeijer & Hartel, 1983; Bembasat & Wand, 1984; Cardelli & Pike, 1985; Green, 1985a; Anson, 1986; Flecchia & Bergeron, 1987; Hill, 1987). Flecchia & Bergeron (1987) describe ALGAE, a language for generating event handlers suitable for supporting the implementation of event driven asynchronous dialogues.

Communicating sequential processes (CSP)

CSP (Hoare, 1985) is a formal mathematically-based algebra capable of describing the behaviour of systems when seen as groups of communicating processes. A system is described by one or more processes, where each process represents the behaviour pattern of some object in the system. A process is made up of events, where events are atomic (cannot be decomposed) and represent some incident or activity in the system. Thus, processes describe the behaviour of objects in terms of events which affect them. CSP makes no distinction between user and system events, and an event can represent anything from a single keystroke to a high level function. CSP lacks a graphical notation, but like the ERS is a potential target language into which a graphical specification could be mapped in order to provide a route to prototyping and implementation.

eventCSP (Alexander, 1987, 1990), which is a subset of CSP, forms part of Specifying and Prototyping Interaction (SPI) tools. EventCSP can be executed through a simulator which takes a specification and simulates the behaviour of the described system. The events are defined in a second language called eventISL, which has a basic syntax, but which is tailored to its host environment. Events are specified in terms of the dialogue state. Each event has an associated condition, a predicate on the state which can be evaluated to decide whether or not the event can be selected. Events specify the inputs and outputs required and are translated into the host language. Executing both parts of the specification gives a prototype of the dialogue. eventCSP can handle direct manipulation interaction - each object is represented by a process whose initial events determine when that object is selected, and the entire display is represented as the parallel operation of the processes for all the objects it contains.

Dialogues for Window Applications (DIWA)

eventCSP has been incorporated into other models, including DIWA (Six and Voss, 1990), a model for object-oriented interface design. A dialogue description in DIWA comprises the dialogue hierarchy, the attributes relating to each dialogue object, and the dynamic dialogue behaviour. Dialogue behaviour is specified in an eventCSP-based abstract language, the main components of which are events, states, and sequences of actions that denote the reaction of the event handler on an event. The object structure of DIWA follows the hierarchical object idea of PAC (Coutaz, 1987), but is less

concerned with the implementation structure of the interface. Complex objects, such as menus, windows, and scroll bars, can be constructed from elementary objects, and both aggregation and set definition are supported. Class definition techniques involving inheritance can also be applied to the definition of objects. A distinction is made between passive (graphical) objects and active objects that are dialogue objects (dialogues) themselves.

3.3 Comparison and review

It was established in Section 2.1 that direct manipulation interfaces are typically based on the 'object-action' model. User-initiated events lead to the selection of objects and operations on these objects, which may in turn trigger system events. Events cause changes in the state of the interface and may impact the availability of selectable options. Multiple substates may be active at any time. The dialogue is largely asynchronous, that is, there is generally an absence of event ordering. However, event sequences may be of significance, for example in relating primitive task actions to higher level tasks. It should be noted in connection with event sequence that although users have no choice but to express their desires as a (time) sequence of selections, they may not *perceive* the task as sequential, and the task may indeed *not* be sequential. The five graphical notations reviewed in Section 3.2 are compared and contrasted below with regard to the three major components of the object-action model, namely objects, events, and states, with the objective of detailing the 'behavioural view' identified in Section 2.4, and hence defining the required scope of a direct manipulation dialogue model.

Objects

Objects are fundamental to direct manipulation interfaces. Users select objects which then become candidates for operations. With regard to object representation:

- STDs, statecharts, and Petri nets represent objects only as the sum of their states. The existence of objects in EDGs can be indicated only indirectly through judicious naming of events.
- Object instances, some attribute values, and associated operations, can be directly represented in Lean Cuisine, and can be grouped hierarchically, but objects are not visually distinguished, nor is any distinction made between syntactic and semantic objects.
- None of the notations supports the definition of object types in terms of attributes, nor the direct association of object types and operations.
- None of the notations is capable of directly representing the behaviour of object types, or of dynamic object instance sets as a dialogue progresses. This behaviour includes inter-instance behaviour (for example the mutually exclusive behaviour of

open windows on the Macintosh), and inter-object behaviour (for example the Macintosh icon-window interrelationship).

Events

In a direct manipulation interface, objects and operations are selected as a result of events. With regard to event representation:

- All the notations except Lean Cuisine explicitly represent events. STDs, EDGs, and statecharts visually distinguish input (user) events from output (system) events.
- Statecharts provide a mechanism for broadcasting output events to several states concurrently.
- STDs, EDGs, and Petri nets are based on possible event sequences. In the case of EDGs the event sequences are directly related to higher level functionality. Although Lean Cuisine imposes constraints on event sequence through its hierarchical submenu structure, event sequences are not represented per se.
- In STDs, Petri nets, and statecharts, events are visually grouped around states, whereas in EDGs, events are grouped according to higher level tasks. In Lean Cuisine each real meneme has an implicit event associated with it, which may trigger its selection or deselection unless modified, and to the extent that options are grouped around object instances, events are also object-centred.
- Event preconditions (state transition rules) are represented in STDs, Petri nets, statecharts, and to a limited extent in Lean Cuisine.

States

Events lead to changes in the state of the interface. In a direct manipulation interface, multiple substates may be active at any point in a dialogue. With regard to state representation:

- States are directly represented as network nodes in STDs, statecharts, and Petri nets. STDs are restricted to a single active state at any time, whereas statecharts and Petri nets support multiple active states, as does Lean Cuisine, which defines system state as the sum of the states of individual selectable representations (menemes). State is only indirectly represented in EDGs.
- Both statecharts and Lean Cuisine provide for hierarchies of states and substates, statecharts through enclosing areas, and Lean Cuisine via nested tree structures.
- State changes can be observed by 'executing' STD, statechart, Petri net, and Lean Cuisine representations.
- Global state variables are supported by some STD varieties but are not shown on the diagram. Lean Cuisine shows current selections (object instances, values etc.) explicitly as meneme states, and also the available options in each state. EDGs, Petri nets, and statecharts on the other hand are concerned primarily with control.

A direct manipulation dialogue model

Based on this analysis, the required scope of a dialogue model for specifying the behavioural view is defined in Table 3.3. The dialogue model must be capable of differentiating syntactic and semantic objects, and of describing the behaviour of both object types and object instance sets. In the Macintosh desktop environment, for example, an icon can be moved (part of the behaviour of the icon object type), and an icon can become deselected as a result of the selection of another icon (part of the behaviour of the icon object instance set). Objects may be related in either hierarchies or type hierarchies, and existence dependencies may occur between objects.

- **Objects:**
  syntactic and semantic objects
  associated attributes (state variables)
  associated options
- **User-generated (input) events:**
  selection of objects and operations
  input of textual data
- **System-generated (output) events:**
  selection of objects
  output of information, error messages...
- **Interface states and substates:**
  possibly orthogonal, and showing for each active state:
  - current selections
  - current state variable values
  - options available for selection
- **Relationships:**
  involving conditions and hierarchies, between:

objects:	object hierarchies, object type hierarchies, other inter-object and inter-instance relationships, including object dependencies
events:	event grouping, event interlinks
states:	state hierarchies, state orthogonality
events & states	event preconditions, state changes
events & tasks	event combinations & sequences

Table 3.3: Required scope of a direct manipulation dialogue model

In the context of developing a graphical notation to support this model it is also instructive to compare and contrast the *surface* characteristics of each of the reviewed graphical notations, in terms of their building blocks, their graph structure, and their complexity. The characteristics of Lean Cuisine and statecharts are compared in Phillips (1991). The following points emerge from the analysis of this chapter:

- STDs and Petri nets are based solely on network graphs, constructed from simple building blocks, but are liable to be large and therefore difficult to comprehend for asynchronous dialogue as all possible event sequences must be represented.
- Statecharts are a mix of network graph and enclosing area notations, making them visually 'busier' and intrinsically more complex than their STD ancestor, which offsets some of the gains in expressive power and reduced diagram size.
- EDGs and Lean Cuisine are both based on tree structures, which according to Touretsky (1986) should render them easier for humans to comprehend. Lean Cuisine has a single all-embracing building block, the meneme, which can be grouped in subtrees, whereas EDGs are built from a mix of nodes distinguished by boxes and fonts. EDGs can be large because all possible event sequences must be shown.

Chakravarty & Kleyn (1990) argue that in any visual formalism there is a trade off between the visual complexity of the graphical notation and the effort required to use it. As the visual complexity increases, the effort required to use the notation effectively also increases. They go on to define criteria for measuring the effectiveness of graphical notations, which can be matched against the reviewed notations:

- *Expressiveness:* this is the ability to handle hierarchy and orthogonality. Statecharts and Lean Cuisine provide for both dialogue hierarchy and state orthogonality. EDGs support dialogue hierarchy only, and Petri nets offer state orthogonality only. STDs offer neither in their basic form, but some extended forms offer hierarchy.
- *Locality:* this is the ability to maintain local adjacency between dialogue elements. With regard to events (and therefore operations), STDs, Petri nets, and statecharts group events by state, EDGs by task, and Lean Cuisine by object.
- *Refinement:* this is concerned with support for stepwise refinement. All the notations are capable of supporting this in either their basic or extended forms.
- *visual compactness*. Although the Lean Cuisine representation of the Finder document folder system is incomplete, it is apparent that the notation is very compact. STDs, Petri net, and EDG representations are weak in this respect because of the need to represent *all* event sequences. Statecharts fall somewhere in between, and are complicated by the need to show transitions explicitly.

None of the notations reviewed is capable of fully describing the underlying behaviour of a direct manipulation interface. The shortcomings are particularly apparent in connection with describing object types and object relationships, and also in relation to events, where none of the notations captures both sequential and asynchronous aspects of a dialogue. STD, EDG and Petri net representations tend to be large and complex because of the need to show all event sequences. Statecharts and Lean Cuisine support both hierarchy and state orthogonality and lend themselves to describing asynchronous dialogue. Event grouping in statecharts is state rather than object-based, and objects are not represented. Statecharts are a mix of two notations which renders them more complex visually, and the need to show all transitions from a given state can lead to large diagrams. Lean Cuisine is object-based, and because events are implicit is a concise and compact notation, although its structural simplicity masks shortcomings in its scope when applied outside the area of menu systems.

A way forward

Either of the 'combinative' and 'adaptive' approaches of Monarchi & Gretchen, (1992), identified in Section 3.1 could be adopted as a way forward. On the one hand the combination of two or more notations could be attempted in a 'multi-view' approach, which brings problems of mapping and integration. On the other hand the adaptation and extension of a single notation could be pursued in an attempt to meet the identified requirements of the dialogue model, which raises issues of notational complexity. The approach adopted in this research is an adaptive one in relation to the Lean Cuisine notation.

Apperley & Spence (1989) suggest that the approach taken in Lean Cuisine, of describing menu systems in terms of selectable representations (menemes), could be extended to cover the other direct manipulation interactions, and that with further development the Lean Cuisine notation could become an early design tool for direct manipulation interfaces. This is based on the proposition that other aspects of direct manipulation interaction are selection-based and could therefore be mapped into an extended meneme model. This will be examined in Chapter 4 through an analysis of Macintosh interaction tasks.

Chapter 4

Extending Lean Cuisine

"It is suggested that not only are menu systems fundamental to the direct manipulation model, but that the approach taken in Lean Cuisine could be extended to cover the other interactions of a direct manipulation system."

Apperley & Spence, 1989

This chapter defines a framework for the extension of Lean Cuisine. Section 4.1 presents the results of an analysis of Macintosh interaction tasks which establishes their selection-based nature. Section 4.2 identifies the scope of the extensions required to enable Lean Cuisine to meet the requirements of the direct manipulation behavioural model defined in Section 3.3, and Section 4.3 proposes an object-based multi-layered development of Lean Cuisine, called Lean Cuisine+, in which a direct manipulation interface is described in terms of the constraints and dependencies which exist between selectable dialogue primitives.

4.1 Analysis of Macintosh interaction tasks

Direct manipulation interfaces such as that of the Macintosh involve a range of interactions undertaken using a mouse and keyboard, and such styles as menu selection, dragging, drawing, form filling, and text entry. An important step in defining the requirements of a notation for representing these interfaces at a high level of abstraction is an understanding of the underlying interaction tasks as viewed from a *user* perspective. This section reviews previous attempts to classify interaction tasks, examines them in the context of the Macintosh interface through close examination of the Finder and the MacPaint application, and proposes a taxonomy of direct manipulation interaction tasks. Particular attention is paid to tasks involving repeated actions, including dragging operations and the manipulation of text.

Interaction with a graphical interface, whether through sequential or asynchronous dialogue, can be decomposed into a number of basic interaction tasks which can be viewed independently of their implementation or physical form. The notion of 'virtual (logical input) devices' was first proposed by Newman (1968) and developed by Foley and Wallace (1974), and Wallace (1976), who proposed a set of five 'primitive

virtual devices' the characteristics of which are summarised in Table 4.1. Wallace also identified four 'non-primitive devices' including *stroke*, the sampling output of which is a series of screen locations. Stroke also appears in the GKS standard (ISO, 1983) as one of six 'logical input devices', although it is interesting to note that stroke was considered but not included in an earlier draft version (Hopgood, 1982). The other five GKS devices match exactly with those in Table 4.1. The motivation for the identification of these virtual devices was hardware and software oriented, and the gains in reducing a potentially large number of actual graphics input devices to a few primitive virtual devices were seen as being primarily in the area of programming and program portability.

Device	Sampling output
Locator	a current location on the screen corresponding to the position of the device
Valuator	a real vector corresponding to the current input of the valuator device
Button	an identifier indicating the code of the button currently being depressed
Pick	a reference to the graphical object currently being pointed at
Keyboard	the entire content of a block of concatenated characters

Table 4.1: Primitive virtual devices (Wallace, 1976)

Foley, Wallace & Chan (1984) identify six fundamental types of interaction tasks from a 'user perspective' in that they represent actions performed by the user. In this respect they differ from, but are related to, the virtual input devices described above, as shown in Table 4.2. These tasks are independent of application and hardware devices, and form the basic building blocks from which more complex interaction tasks and complete dialogues are assembled. The six tasks are summarised and matched with Wallace's virtual devices in Table 4.2. The *text* task is concerned with text which becomes part of the information in the computer, as distinct from the input of a character string which forms part of another interaction task, for example a command name or coordinate position. The *path* task, which is equivalent to Wallace's non-primitive stroke device, is considered by Foley *et al* to be a 'fundamental task' (p.22) "even though it consists of other primitive tasks (*position* or *orient*), because another fundamental

dimension - time - is involved". Just as the *text* task is concerned with a vector of character values, so the *path* task returns a vector of positions.

Task	Purpose	Equivalent virtual device
Position	indicate a position on the display	Locator
Orient	rotate an entity in 2-D or 3-D space	Locator
Quantify	specify a value to quantify a measure	Valuator
Select	select from a set of alternatives	Button, Pick
Text	input a series of characters	Keyboard
Path	generate a series of positions over time	Stroke

Table 4.2: Interaction tasks (Foley *et al*, 1984)

An underlying generic subtask *Get Value* is proposed, which can be used as a basic building block in connection with each of the tasks in Table 4.2. The structure of this subtask is shown in Figure 4.1. *Get Value* generates a representation of a single value each time it is performed, in the form expected by the interaction task. This representation is selected in one of two ways:

- *directly:* in a single selection action from the display or keyboard, or
- *indirectly:* in the form of a character string built by the repeated selection of characters from the keyboard or display.

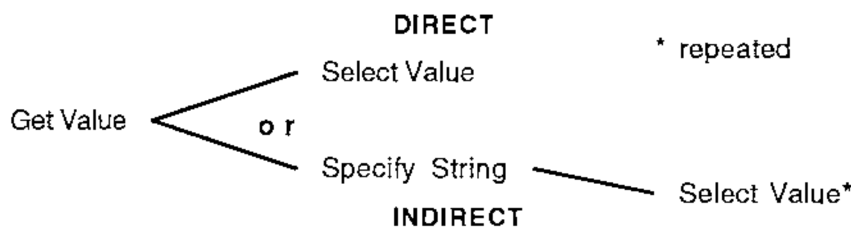


Figure 4.1: *Get Value* subtask

The input of a character string representing some other value e.g. a command, and the input of text are handled differently. Text is built through repetition of the *Get Value* subtask, each repetition generating a directly selected character representation, whereas a character string representing a value is constructed within *Get Value*, and the string returned as the value representation.

Phillips and Apperley (1991) examine the utility of this subtask through a Macintosh task analysis involving aspects of the Finder and MacPaint applications. This analysis appears in Appendix B and a summary of the findings follows. The mouse techniques referenced in the analysis were introduced in Section 2.3, and are summarised again in Table 4.3. The first four columns of the analysis in Appendix B - *Object*, *Action*, *Effect* and *Method* - describe physical processes; the remaining columns classify each process in terms of the tasks listed in Table 4.2. They identify the value type generated each time the *Get Value* subtask is performed, and the underlying representation. Compound tasks are delineated. The references in the analysis which follows are to the processes as numbered in Appendix B. Significant findings emerge in connection with processes involving repeated tasks.

Technique	User action(s)
Click	press and quickly release the mouse button
Shift-click	hold down the Shift key and click the mouse button
Double-click	press and release the mouse button twice in quick succession
Press	press and hold down the mouse button
Drag	press and hold down the mouse button move the mouse release the mouse button

Table 4.3: Macintosh mouse techniques

Finder processes

Appendix B1 examines a representative selection of Finder processes grouped under 6 headings: *Icons, Windows, Menus, Dialogue boxes, Commands* and *Text*. The distinction between the last two groups is that between the depression of a single character key representing a command, and the entering and editing of text. Those processes concerned with dragging or text input, or otherwise involving repeated tasks, merit fuller analysis, and are examined below.

Dragging

Consider the following processes from the analysis in Appendix B1, each of which involves the mouse being dragged:

Group 1:

1.2 Dragging over a group of icons to make them current.

3.2 Dragging down over a pull-down menu to select an item.

4.2 Dragging over a list of names to select several items from the list.

6.7 Dragging over text in order to make it current.

Group 2:

1.5 Dragging icon(s) across the desktop.

2.5 Dragging on a size box to change window size.

2.6 Dragging a window across the desktop.

Group 3:

2.9 Dragging on a scroll box to scroll a window.

At first appraisal these may appear to be *path* tasks, because the mouse is dragged. However, a fundamental property of *path* tasks is that every intermediate position is of significance and must be recorded. In Group 1, dragging the mouse results in a series of discrete selections as the mouse cursor passes over the objects concerned. There is no absolute path or sequence of positions - the consequence is a set of selections. In process 6.7, a piece of text is made current through selection of its start and finish positions. The processes in Group 1 translate therefore into one or more discrete *select* tasks.

In Group 2 each of the processes essentially involves the selection of a single new final position, with the path taken by the cursor as the mouse is dragged to this position being of no lasting significance. Each of the processes in this group translates into a single *position* task. The process in Group 3 involves the selection of a single numeric quantity which is then used by the window manager to position the file being displayed in the window. This process translates into a single *quantify* task.

It should be noted in connection with dragging that although the intermediate path taken by the mouse in achieving the desired effect can vary and may be wholly transient, the path actually selected will have implications at the presentation level in terms of the semantic feedback provided to the user in a continuous representation environment. For example, dragging a document icon across the desktop (process 1.5) may require folder icons to be temporarily highlighted as they are encountered, to show that they are possible recipients of the document.

Text and character input

In the context of the Finder, there are two basic types of keyboard input:

- Single character input in which characters represent commands (as in processes 5.1, 5.2, 5.3, and 6.5). Each of these processes maps directly into a *select* task.
- Single or multiple character input or editing, in which the characters represent themselves (as in all remaining processes in group 6). These processes map variously into *character, position*, or *select* tasks, with the exception of process 6.6, which is concerned with the input of text which may be lengthy and highly structured. The latter process is mapped into a compound *text* task comprising a related set of *character* tasks. The character task is thus a repeating unit in connection with the input of text.

Get Value, as defined earlier, can meet each of these needs, as in connection with the first requirement it is capable of generating representations of values resulting from the input of character strings (of length 1 in the case of Finder commands), and in the case of the second requirement it is capable of generating single character values selected from a finite set.

Other repeated tasks

There is a further group of processes in the analysis which involve task repetition:

2.7 Pressing the mouse button on a scroll arrow to scroll a window.
2.8 Pressing the mouse button on the scroll bar grey area to scroll a window.
4.5 Pressing the mouse button on a dialogue box scroll arrow to set a value.

Each of these processes repeatedly returns a value, and therefore involves the repeated execution of the *quantify* task. In process 2.7 for example, the effect of holding down the mouse button in order to achieve continuous scrolling is of repeatedly generating a numeric value which is used by the window manager to position the window.

MacPaint processes

The above analysis has revealed no *path* tasks, but this is perhaps to be expected as the Finder includes no drawing or painting facilities. MacPaint (Apple, 1983) is included here in order to provide a more complete analysis of the Macintosh interface, one which includes *path* tasks. MacPaint is a Macintosh painting application providing a set of tools which can be used to build pictures on a 'page' which is viewed through a fixed screen window. In addition to the usual pull-down menus providing a range of *File, Edit* and *Style* options, MacPaint offers three on-screen menus providing for:

- the selection of drawing tools;
- the selection of line and border widths; and
- the selection of 'fill in' patterns.

The MacPaint interface analysis in Appendix B2 is based on Version 1.5, and is limited to those processes which differ from and extend the range of those considered for the Finder. Processes relating to the input of text, to cutting and pasting, to menu selection, and to window management are thus excluded. The processes included relate to the use of the MacPaint drawing tools. Those concerned with *path* tasks, or otherwise involving repeated tasks are analysed below.

Path tasks

The analysis shows that in 15 of the 18 processes considered, dragging on the mouse is involved. Only four *path* tasks emerge however:

1. Drawing a freehand line using the brush, pencil or spraycan.
4. Selecting a freehand drawn area using the lasso.
10. Erasing by moving the eraser over the page.
14. Drawing a hollow or filled freehand shape.

The common thread here is that each of these processes involves freehand movement in which the entire path taken by the tool (in a single dragging operation) is significant, that is, each process is a compound task consisting of a *continuous* series of tasks all of which are of permanent significance. The processes in this group thus translate into a *path* task comprising a related set of *position* tasks.

Other repeated tasks

The following processes, arranged in two groups, also involve repeated tasks:

Group 1:

3. Selecting a rectangular area to make it current.
7. Copying of a selected area repeatedly.
12. Drawing a hollow or filled rectangle (with square or rounded corners).
13. Drawing a hollow or filled oval.
15. Drawing a hollow or filled polygon.

Processes 3, 12, and 13 involve the return of two discrete positions which define a diagonal of a rectangle. Process 7 involves a series of discrete copies at selected positions, and process 15 involves a series of discrete positions joined by straight lines. None of these actions is a *path* task, and each has been translated into a repeated *position* task. The selection of a rectangular area (in 3) should be contrasted with the selection of a freehand drawn area (in 4 above, under *path* tasks). These two actions highlight the

distinction between *path* tasks, in which the entire path is significant, and other tasks which involve only the return of positions at nodal points.

Group 2:

16. Setting dots to black or white to create a "fill in" pattern.
17. Setting dots to black or white in detailed drawing.

Both actions involve repeated discrete *select* tasks in which the state of a bistable dot is flipped from black to white and vice versa.

A task taxonomy

The above analysis has clarified the nature of processes which involve repeated tasks. In this context, two classes of process have emerged:

1. Those in which the repeated tasks form part of a related whole, and in which every user action is significant and recorded. Here, a continuous series of tasks is viewed as being a single compound task. Both *path* and *text* tasks fall into this group. *Path* tasks, identified only in MacPaint, return vectors of positions, and the *text* task has been defined in terms of an underlying *character* task.
2. Those which involve the repetition of a set of discrete subtasks, which generally form a sub set of the physical actions performed by the user. Examples involving both dragging and pressing actions on the mouse have been identified. Actions in this category are considered as a set of discrete tasks rather than a single compound task.

The analysis has also shown that the tasks of Table 4.2 decompose into underlying tasks which involve either the direct selection of a value, or the indirect specification of a value (representing a command, position etc.) via a character string input from the keyboard. In the case of selection from the display there is an implicit assumption that some other value is associated with the position, for example, a numeric quantity. The *path* and *text* tasks differ from the other four in requiring the repetition of simpler tasks, and they are viewed as 'compound' tasks.

In the light of the analysis, a taxonomy of interaction tasks can be produced for a Macintosh-like direct manipulation interface. A new *character* task, involving a single selection action, is required in connection with the compound *text* task. In performing this task the user selects a character from the available set. Given the non-appearance of the *orient* task in the analysis, and in the light of the earlier taxonomy of virtual devices (Wallace, 1976), the *orient* task will be subsumed within the *position* task. The resulting taxonomy is shown in Table 4.4. It includes two compound tasks, *path* and *text*, in addition to *position, quantify, select* and *character* tasks, all of which map into the generic *Get Value* subtask.

Compound task	Task	Subtask	Value represents
	Position	Get Value	Position
	Quantify	Get Value	Numeric quantity
	Select	Get Value	Entity e.g. command
	Character	Get Value	Character
Path	Position*	Get Value	Position
Text	Character*	Get Value	Character

* repeated

Table 4.4: Taxonomy of interaction tasks

With the addition of the detail of the *Get Value* subtask from Figure 4.1, the final taxonomy can be represented graphically as shown in Figure 4.2.

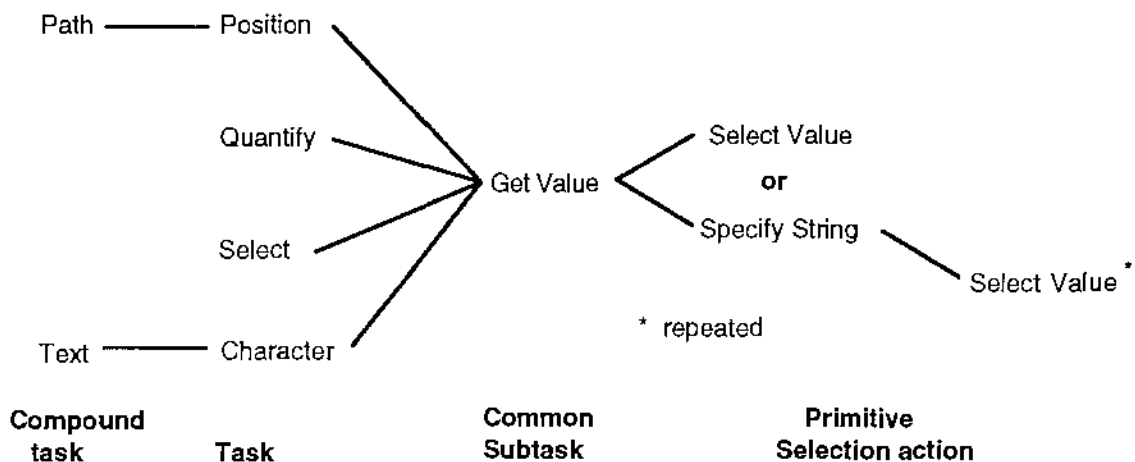


Figure 4.2: Taxonomy of interaction tasks

Figure 4.2 shows quite clearly that at the most primitive level all tasks reduce to one or more 'selection' actions involving the mouse or keyboard. This has implications for the types of tools and techniques needed to describe and implement direct

manipulation interfaces. In particular it supports the proposition of Apperley & Spence (1989) that more general direct manipulation interactions such as positioning and dragging could also be regarded as 'selection', and could therefore be mapped into the Lean Cuisine meneme model. With suitable development and extension, the meneme model should thus be capable of specifying the behaviour of direct manipulation interfaces in terms of selectable dialogue primitives.

4.2 Extending the scope of Lean Cuisine

The capabilities of Lean Cuisine, a graphical notation for describing systems of menus, were examined in Chapter 3. The behaviour of a menu is described using selectable representations, called 'menemes', of actions, parameters, objects, states, and other attributes, in which selections may be logically related or constrained through mutually compatible and mutually exclusive meneme groupings, and through the use of meneme modifiers. In the wider context of a direct manipulation interface, Lean Cuisine has the following capabilities with regard to objects, events and states:

Objects

- Object instances, some attribute values, and associated operations, can be directly represented, and can be grouped hierarchically. Objects are not visually distinguished however, nor is any distinction made between syntactic and semantic objects.
- Lean Cuisine does not support the definition of object *types* in terms of attributes, nor the direct association of object types and operations.
- Lean Cuisine is not capable of representing inter-object behaviour, nor the behaviour of *dynamic* object instance sets.

Events

- Each real meneme has an *implicit* event associated with it, which may trigger its selection or deselection. Events may be user generated or system generated, this distinction also being an implied one.
- Lean Cuisine imposes constraints on event sequence through its hierarchical submenu structure. However, event sequences *per se* are not represented, and it is not possible to link event sequences with higher level tasks.
- Event preconditions (state transition rules) can be represented to a limited extent in Lean Cuisine, through its hierarchical structuring, meneme grouping, and meneme modification capabilities.
- To the extent that options are grouped around object instances, events are object-centred.

States

- System state is defined as the sum of the states of individual selectable representations (menemes).
- Lean Cuisine provides for hierarchies of states and substates via nested tree structures, and for state orthogonality through mutually compatible subtrees.
- Lean Cuisine shows current selections (object instances, values etc.) explicitly as meneme states, and within the limitations identified above for event preconditions shows the available options in each state.
- 'Execution' of a Lean Cuisine diagram leads to a succession of permissible states, underlining the essential nature of Lean Cuisine which shows the state changes resulting from meneme behaviour as (implicitly defined) events are applied to them. At present, a Lean Cuisine diagram is executed only in the mind of the designer.

In Table 3.3 the required scope of a direct manipulation dialogue model was defined. This is presented in Table 4.5 under *Objects, Events*, and *States.*

- **Objects:**

  syntactic and semantic objects; dynamic object instance sets;
  associated attributes (state variables), attribute values and options;
  object hierarchies; object type hierarchies;
  inter-object and inter-instance relationships, including object dependencies.

- **Events:**

  user generated (input) events:
      selection of objects and operations, input of text;
  system generated (output) events:
      selection of objects, output of information and error messages;
  event interlinks; event preconditions; event combinations & sequences.

- **States:**

  states and substates, showing for each state:
      current selections and state variable values,
      options available for selection;
  state hierarchy; state orthogonality;
  execution to show state change, current selections and option availability.

Table 4.5: Required scope of a direct manipulation dialogue model

In Table 4.6 those items in Table 4.5 which require Lean Cuisine extensions are shown underlined. With the exception of event preconditions, in which area Lean Cuisine has limited capability as identified above, these are new requirements of the notation.

- **Objects:**
  syntactic and semantic objects; dynamic object instance sets;
  associated attributes (state variables), attribute values and options;
  object hierarchies; object type hierarchies;
  inter-object and inter-instance relationships, including object dependencies.
- **Events:**
  user generated (input) events:
  selection of objects and operations, input of text;
  system generated (output) events:
  selection of objects, output of information and error messages;
  event interlinks; event preconditions; event combinations & sequences.
- **States:**
  states and substates, showing for each state:
  current selections and state variable values,
  options available for selection;
  state hierarchy; state orthogonality;
  execution to show state change, current selections and option availability.

Table 4.6: Aspects of the dialogue model requiring Lean Cuisine extensions

4.3 A framework for Lean Cuisine+

The approach to extending Lean Cuisine must now be considered, and a framework for Lean Cuisine+ defined. In essence, a Lean Cuisine+ description will present a set of dialogue primitives, represented by menemes, and a set of constraints over the selection of these primitives, where a primitive can be an object, an operation, a state, or a value. In its executable form it will show the state changes resulting from meneme behaviour as events are applied to them, thus providing for the exploration of the *dynamics* of interface behaviour.

Direct manipulation interfaces are first and foremost object-oriented, and Lean Cuisine+ will be therefore be grounded in an object-oriented approach, with selectable representations grouped primarily around objects. In the broader context of systems

design, Burns & Lister (1991) argue that such an approach minimises the structural clash between architectural specification, design, and implementation. Objects are seen as an adequate modelling tool for the *functional* requirements of a system because (p.174) "they can be used to provide traceability through all stages of the design process to implementation and execution". Burns & Lister go on to define 'logical' and 'physical' architectures which match closely with the behavioural and presentational interface views developed in Section 2.4. Logical architecture is concerned with defining a set of object classes and their interfaces and relationships, while the physical architecture is primarily concerned with object instances of the logical object classes and the mapping of these onto the target environment.

Lean Cuisine offers a clear, concise, and compact graphical notation, based on tree structures. A major issue in extending the notation is that of how to retain these aspects whilst at the same time considerably enriching it. As reported earlier, Touretsky (1986) considers that hierarchy is by far the way humans prefer to structure things, which suggests that the basic tree diagram of Lean Cuisine, which captures some of the constraints which exist between dialogue primitives, should be retained. The problem then becomes one of how to add further detail without compromising the appeal of the basic diagram. In connection with the development of the Pasta-3 direct manipulation interface for browsing knowledge-base schemas, Kuntz & Melchert (1990) argue strongly that the display of both entity interrelationships *and* inheritance, which requires two types of arcs between nodes, should not be undertaken in the same diagram because it would be confusing and could lead to users misinterpreting the graph. Their solution is two separate diagrams, which introduces problems of consistency.

An alternative approach, and the one adopted in Lean Cuisine+ in order to preserve the simplicity of the basic tree notation, is to *layer* the notation (Phillips, 1992). In a Lean Cuisine+ specification, the behaviour of a direct manipulation interface is described in terms of a basic dialogue tree overlayed by additional constraints and dependencies, with event sequences captured in a further orthogonal layer. In the support environment described in Chapter 8 these layers can be switched in and out during the construction, browsing, and execution of a specification. The Lean Cuisine+ notation is based on the following framework, which is also shown in Figure 4.3:

Base layer	Consists of a dialogue tree augmented by additional primitives providing for the description of object types and type hierarchies, the delineation of semantic and syntactic objects, and the behaviour of dynamic object instance sets. Options are grouped primarily around objects, which can have state variables attaching to them. Stepwise refinement of tree diagrams is supported.

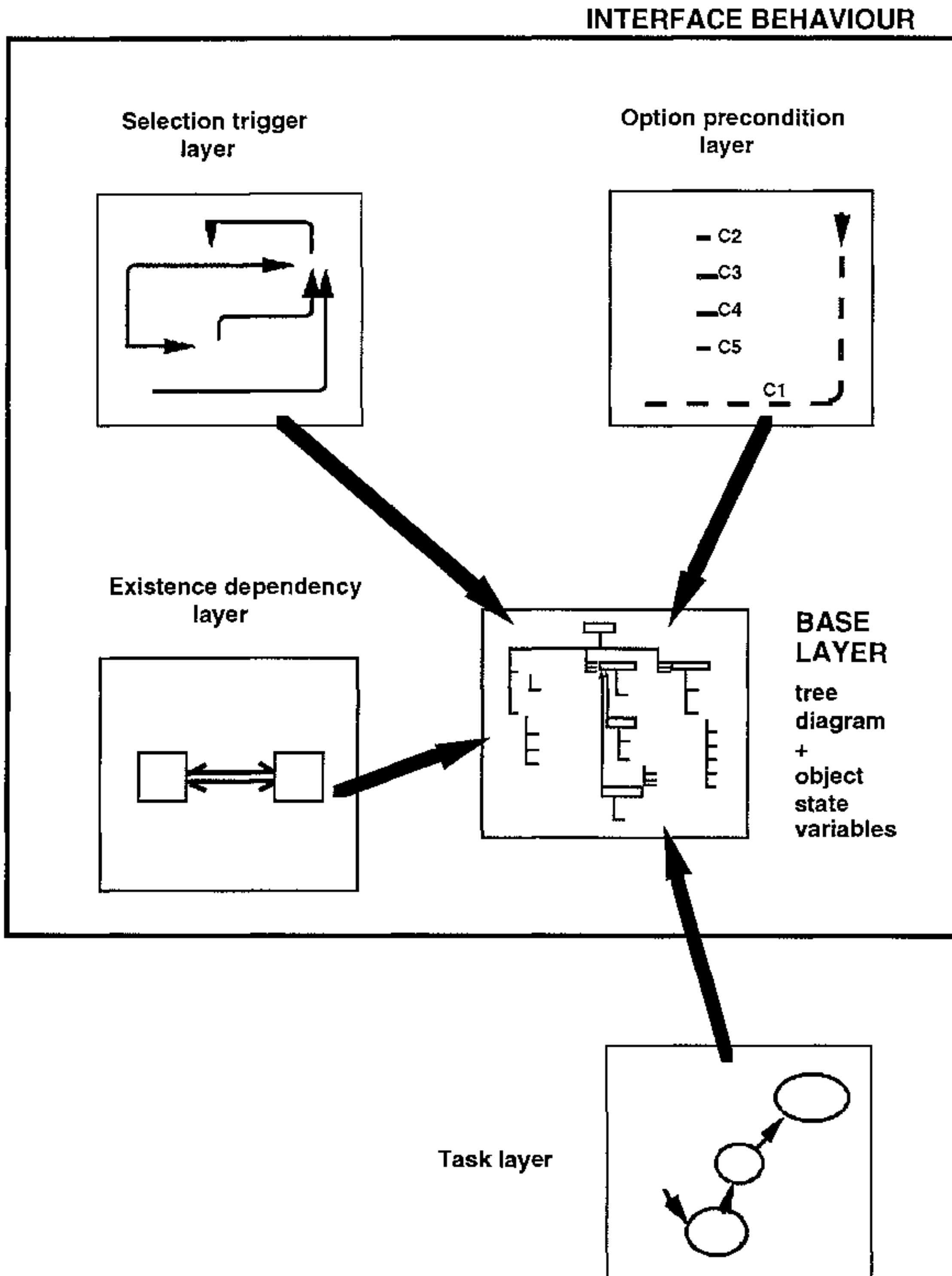


Figure 4.3: Lean Cuisine+ framework

Selection trigger layer Annotated selection triggers between menemes are overlayed on the basic dialogue tree. They provide for the description of inter-object relationships, and of system responses to user events (event interlinks) involving object

selection. They capture the fact that the selection or deselection of an object or option may trigger (possibly conditionally) the selection or deselection of objects and/or options in the same or different dialogue. Triggers may also create or delete object instances.

Option precondition layer Option (availability) preconditions are overlayed on the basic dialogue tree, and provide for the description of event preconditions. The availability of an option for selection may depend upon the state of a meneme in another part of a dialogue, or on the existence of some other state or condition, or both.

Existence dependency layer Existence dependencies are overlayed on the basic dialogue tree. The existence of an object may depend upon the existence of another object or dialogue. The dependency may be mutual.

Task layer Sequences and/or groups of primitive selection actions (and associated events) relating to higher level user tasks are overlayed on the basic dialogue tree. This layer, which is orthogonal to the basic behavioural model, provides a link between object-oriented and task-oriented views of the dialogue, and captures any temporal relationships between primitive actions. Selection and deselection actions are shown, and user and system actions are delineated.

Lean Cuisine+ is thus an object-based multi-layered development of Lean Cuisine providing for the specification of direct manipulation interfaces in terms of the constraints and dependencies which exist between selectable dialogue primitives. The state of an interface at any time is represented by the sum of the individual meneme states plus any associated state variables (e.g. names, positions etc.). The basic behavioural model, involving the base layer plus selection trigger, option precondition, and existence dependency layers, is developed in Chapter 5 and further extended in Chapter 6. The orthogonal task layer, which can be generated automatically from a task decomposition provided enough information is captured about task action relationships, is developed in Chapter 7. A Lean Cuisine+ specification can be executed to produce a succession of permissible states, showing current selections and available options in each state, with unavailable options greyed[1]. This aspect of Lean Cuisine+ is developed in Chapter 8.

[1] 'Greying' and 'ungreying' are used from here on to denote the dimming and undimming of part or all of a Lean Cuisine+ dialogue tree diagram during the browsing or execution of the specification. Greying denotes unavailability during execution.

Part 2

Development

Chapter 5	The Basic Lean Cuisine+ Notation	81
Chapter 6	Extending the Lean Cuisine+ Object Range	103
Chapter 7	Lean Cuisine+ in High Level Interface Design	125
Chapter 8	Software Support for Lean Cuisine+	157

Chapter 5

The Basic Lean Cuisine+ Notation

In this chapter the basic Lean Cuisine+ dialogue notation is developed. In Section 5.1 the structural requirements of the notation are identified, and the Finder document folder system which was introduced in Chapter 2, and which is used here in the development of the notation, is briefly reviewed. Section 5.2 defines the Lean Cuisine+ base layer comprising the dialogue tree plus object state variables, and Section 5.3 describes three further layers which capture additional constraints. The Finder case study is extended in Section 5.4 to include application programs and documents, and to handle the insertion and ejection of diskettes. Menus are considered in Section 5.5, and aspects of the notation are reviewed in Section 5.6.

5.1 Introduction

The base layer of a Lean Cuisine+ dialogue specification consists of a hierarchical arrangement of menemes, grouped into subdialogues. A subdialogue headed by a real non-terminal meneme in a Lean Cuisine+ diagram consists of all of the real menemes with which the header is connected by downward directed branches, either directly or via virtual menemes. A subdialogue is available for selection only if its parent dialogue header is in a selected state (i.e. its parent dialogue is active). In grouping menemes in the base layer tree diagram, and in applying meneme modifiers, certain constraints are placed on their behaviour. Further constraints can be added in the form of overlays to the base layer diagram.

The values of variables associated with the system state may be referenced and manipulated through appropriate subdialogues. 'State variable' values may be directly represented by menemes in the dialogue tree. Alternatively, state variables may exist in the form of parameters to menemes representing operations, particularly in the early stages of developing a dialogue specification. Thus a subdialogue *SetFontStyle,* capable of changing the value of the state variable *fontstyle* for a *character* object, may be defined initially with *fontstyle* as a parameter. Subsequently this may be expanded, with selectable *fontstyle* values represented as menemes. This direct representation of state variable values will tend to occur as the lower levels of the dialogue tree are developed, and during the addition of a menu system in the final stage of development of a specification (as described in the methodology in Chapter 7). The Lean Cuisine+

notation must be capable of supporting state variables at both levels of abstraction during stepwise refinement of a specification, and of representing the following dialogue components:

- *The root dialogue:* this is the top level dialogue, which when made available through selection of the root node provides access to the subdialogue(s) and/or state variable values immediately below it, subject to any constraints applying.
- *Subdialogues:* these consist of logical groupings of menemes within the body of the dialogue, each headed by a real meneme representing an object or operation. When made available through selection of its header, a subdialogue provides access to the subdialogue(s) and/or state variable values immediately below it, subject to any constraints applying. Options in the parent dialogue remain available, again subject to any constraints applying.
- *Modal Subdialogues:* these are subdialogues which when selected exclude all other parts of the dialogue.
- *State variable values:* these are terminal nodes representing values available for selection or deselection within a particular subdialogue. Values may remain set after termination of the subdialogue, in which case they may be referenced in other subdialogues, subject to any scoping restrictions within object type hierarchies.

In the sections which follow the basic Lean Cuisine+ notation is developed through description of aspects of the Macintosh Finder document folder system. A full definition of the notation appears in Appendix C. The purpose of the document folder system, a fuller description of which appeared in Section 2.3, is to provide for the hierarchical organisation of application programs and documents to facilitate subsequent access to them. In Sections 5.2 and 5.3, the following simplifications are assumed:

- folder icon instances are considered to exhibit mutually exclusive behaviour, i.e., only one icon can be selected at any time (a switch mechanism capable of changing this to mutually compatible behaviour is introduced in Section 6.3);
- folder (and therefore window) names are restricted to one character;
- application programs and documents are excluded (they are added in Section 5.4); and
- the insertion and ejection of diskettes have been omitted (they can be handled by the notation, and are also dealt with in Section 5.4).

The complete document folder life cycle is described: a document folder instance can be created; the icon representing it can be moved and renamed; the folder can be opened to make its contents available via a window, which can be moved and resized and its contents scrolled; the window can be closed; and the folder can be deleted.

5.2 The base layer

The Lean Cuisine+ base layer consists of a dialogue tree diagram plus object state variables. The notes which follow relate to the tree diagram in Figure 5.1:

- The *Finder* root dialogue provides access to the mutually exclusive *NewFolder*, *EmptyTrash* and *PrintDirectory* subdialogues and to the mutually compatible *FolderIcon* and *Window* subdialogues, the selection of either of which provides access to further options. The virtual meneme *Controls* has been used to further group options within the *Window* subdialogue.
- Modal subdialogues are represented in one of two ways. Simple modal subdialogues such as *NewFolder*, involving a primitive operation, are represented by 'monostable' menemes, shown thus (⊥). Monostable menemes remain selected only for the duration of the operation, and revert to an unselected state on its completion through system action (exceptionally following some user intervention - see Section 5.4). Otherwise, modal subdialogues are designated by a bar across the arc above the subdialogue header, as in the case of *PrintDirectory*.

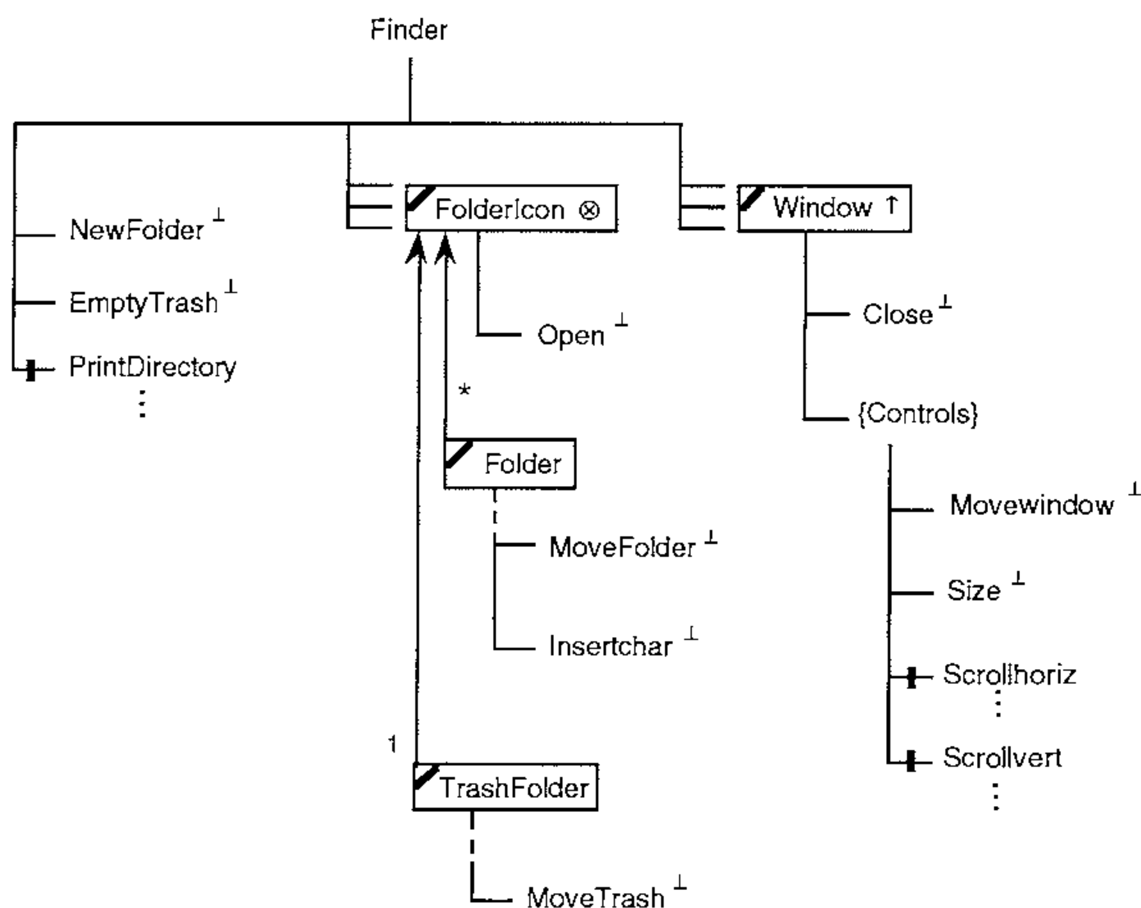


Figure 5.1: Finder: Lean Cuisine+ base layer tree diagram

- The notation distinguishes menemes representing objects, which are shown boxed, and also delineates 'semantic' and 'syntactic' objects (which were introduced in Section 2.3). All the objects in the Finder example are syntactic objects, which carry a diagonal in the top left hand corner. Semantic objects appear in the examples in Chapter 6. Objects are also described in terms of state variables, which are discussed in the next subsection. The fork symbol shows that *FolderIcon* represents one of a dynamic object instance set of folders, including the Trash folder, which exhibit mutually exclusive behaviour. *Window* similarly represents one of a mutually exclusive dynamic instance set of windows.
- The dialogue involves two object types - *FolderIcon*, and *Window*, and two subtypes - *Folder* and *TrashFolder*. *TrashFolder* is shown as having a cardinality of one (the default is many, as in the case of *Folder*). A folder object instance is represented by a *FolderIcon/Folder* meneme pair, and the Trash folder object by the *FolderIcon/TrashFolder* meneme pair. Each meneme in these pairs represents the same object at a different level in the type hierarchy and heads a subdialogue at that level. In general selection is propagated in either direction within the type hierarchy. In this dialogue the user is restricted to selecting at the subtype level through the employment of a meneme modifier at the supertype level (see next paragraph). Selection of either subtype causes selection of the *FolderIcon* parent type, and access to its state variables and subdialogue options (in this case the *Open* operation). In effect the subtype 'inherits' the characteristics of its parent type and extends them. In this example the options of the parent dialogue are extended by additional mutually exclusive options in the case of both subtypes. This is shown by the broken arcs in the subdialogue trees immediately below *Folder* and *TrashFolder* (generally enough structural information must be provided within the subdialogues at both levels to show how the behaviour inherited from the dialogue of the parent type is extended). Selection propagation, and therefore subtype inheritance, can be inhibited - an example appears in Section 6.3.
- Four meneme modifiers appear in Figure 5.1. The monostable modifier (⊥) has already been considered. *Window* is flagged as 'select-only' (↑), reflecting the fact that once available (open) a window can be selected but not directly deselected by the user. *FolderIcon* is represented by a 'passive' meneme (⊗), and is only indirectly selectable through selection by the user of one of its subtypes. One of the instances of the *Folder* set is flagged as the 'default choice' (*) at the commencement of the dialogue (on starting up the Finder the icon representing the system volume is in a selected state).
- Stepwise refinement is supported. A vertical ellipsis below a meneme indicates that further development of that subdialogue takes place in another diagram. The *PrintDirectory* modal subdialogue in Figure 5.1 is further developed in Figure 5.2.

This subdialogue in turn provides access to further modal subdialogues, including *Help* which would be developed in a further diagram. At the lower levels of the diagram, state variable values such as *FirstPage* are directly represented, with default choices flagged (*). The notation provides for display (in parentheses) of ranges and default values associated with objects such as *Copies*. The simple modal subdialogues *Insertcop*, *Insertfrom*, and *Insertto* add a digit to a displayed value associated with their respective parent objects each time they are selected. The *PrintDirectory* subdialogue header is flagged as select-only because deselection can be achieved only indirectly through selection by the user of either *OK* or *Cancel*, which deselect *PrintDirectory* via selection triggers. (The fact that the *PrintDirectory* subdialogue is represented on the Macintosh as a dialogue box is ignored at this stage of the design, but could be added later. This would require recognition of the fact that the dialogue box itself could be moved.)

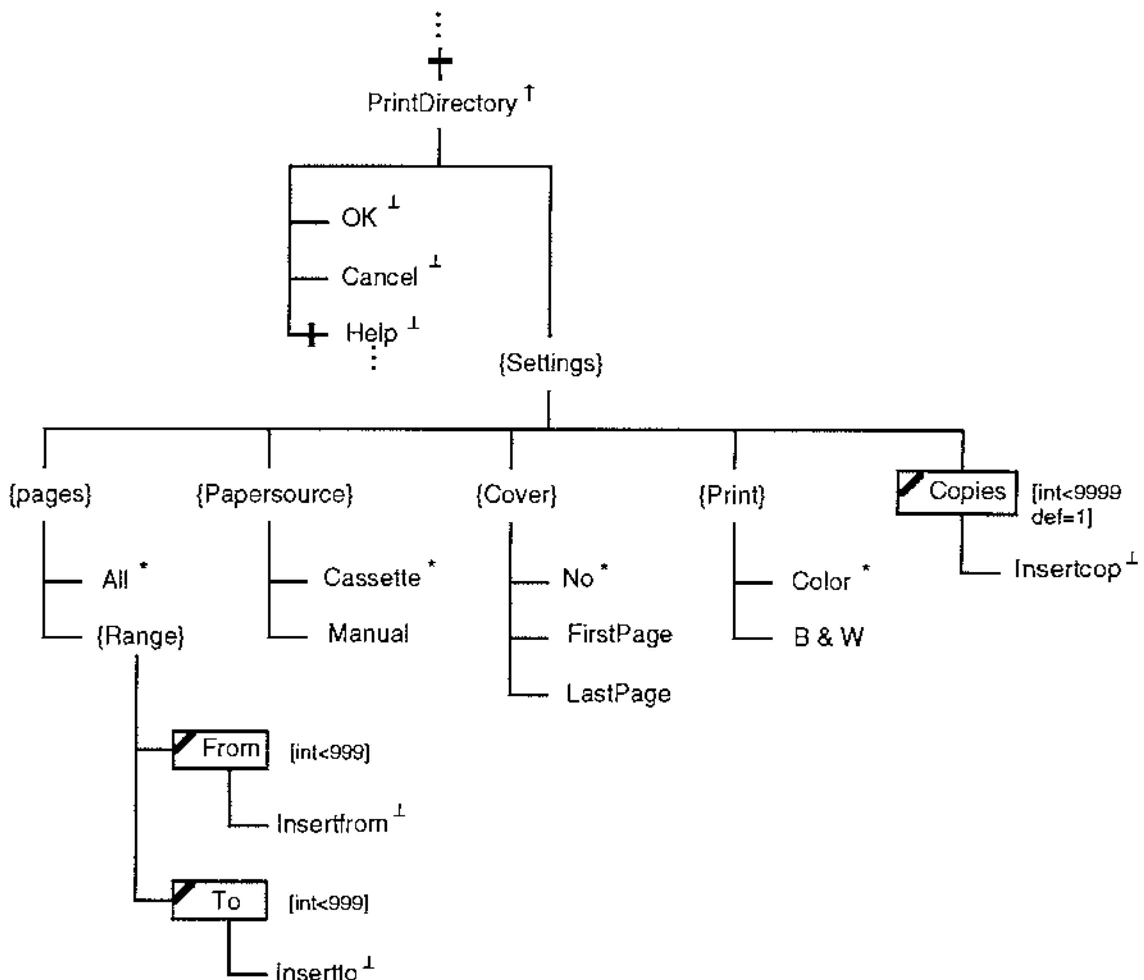


Figure 5.2: *PrintDirectory* modal subdialogue tree

- With regard to the life cycle of a document folder, operations occur within several subdialogues. A folder (and its associated window) is created by the *NewFolder* subdialogue, and may subsequently be opened within the *FolderIcon* subdialogue, moved or renamed within the *Folder* subdialogue, and closed within the *Window* subdialogue. Deleting a document folder is rather more subtle and involves two stages: (1) moving the folder into the Trash folder (that is, moving it to a position where its parent becomes the Trash folder), and (2) selecting the *EmptyTrash* subdialogue. The latter deletes both the folder and its associated window via a selection trigger (see Figure 5.3).

Object state variables

Each meneme representing an object can have attributes attaching to it. These are referred to as 'object state variables' (OSVs), and form part of the entire set of state variables associated with the dialogue, which would be defined in the system dictionary. Within type hierarchies specific state variables attach to each level. With normal selection propagation within the hierarchy, where user selection of a subtype instance also selects a supertype instance, subtypes effectively 'inherit' the state variables of their supertype. Where selection propagation within the hierarchy is inhibited, state variables are not available for manipulation outside the level at which they occur (an example appears in Section 6.3). OSVs also exist as parameters to options which may modify them as a dialogue progresses, e.g. *MoveFolder (iposition, fparent).*

The values of OSVs form part of the total system state, both during the execution of a Lean Cuisine+ specification and in the subsequent prototyping of the interface, and may be tested in event preconditions. At the early design stage they are intended to be indicative, with additional attributes and further detail being added at subsequent stages of design. An initial set of OSVs relating to the Finder document folder system, grouped by object type and subtype, is shown in Table 5.1. The exact form of presentation of OSVs on a Lean Cuisine+ base layer tree diagram would be dependent upon the nature of the software support environment. This issue is addressed further in Chapter 8.

5.3 Further constraint layers

Additional constraints, in the form of selection triggers, option preconditions, and existence dependencies, are captured in three further layers which can be superimposed on the base layer tree diagram.

The selection trigger layer

Selection triggers are directed links between menemes such that the selection or deselection of a meneme by the user triggers (possibly conditionally) the selection or

deselection of one or more other menemes in the same or different dialogues (with concomitant selection of dialogue and/or subdialogue headers as and if necessary). A selection trigger represents a system response to a user action. Triggers may also create or delete object instances, and are either 'unconstrained', in which case further 'knock on' effects are possible, or 'constrained', in which case the behaviour of the selected or deselected object is modified, inhibiting further triggers and preventing side effects such as cycles.

FolderIcon:	#iname	object name
	iobclass	object class = syntactic
	iobtype	object type = folder icon
	iselstat	object selection status (selected, unselected)
	ifostat	folder open status (open, closed)
	iposition	current icon position
Folder:	**Subtype of FolderIcon**	
	#iname	object name
	fobsubtype	object subtype = document folder icon
	favstat	object availability status (available, not available)
	fparent	parent folder name
TrashFolder:	**Subtype of FolderIcon**	
	#iname	object name
	tobsubtype	object subtype = Trash folder icon
Window:	#wname	object name
	wobclass	object class = syntactic
	wobtype	object type = window
	wavstat	object availability status (open, closed)
	wselstat	object selection status (active, inactive)
	wposition	current window position
	wwidth	current window width
	wdepth	current window depth
	wdwidth	current overall directory width
	wddepth	current overall directory depth

Table 5.1: Initial set of Finder OSVs

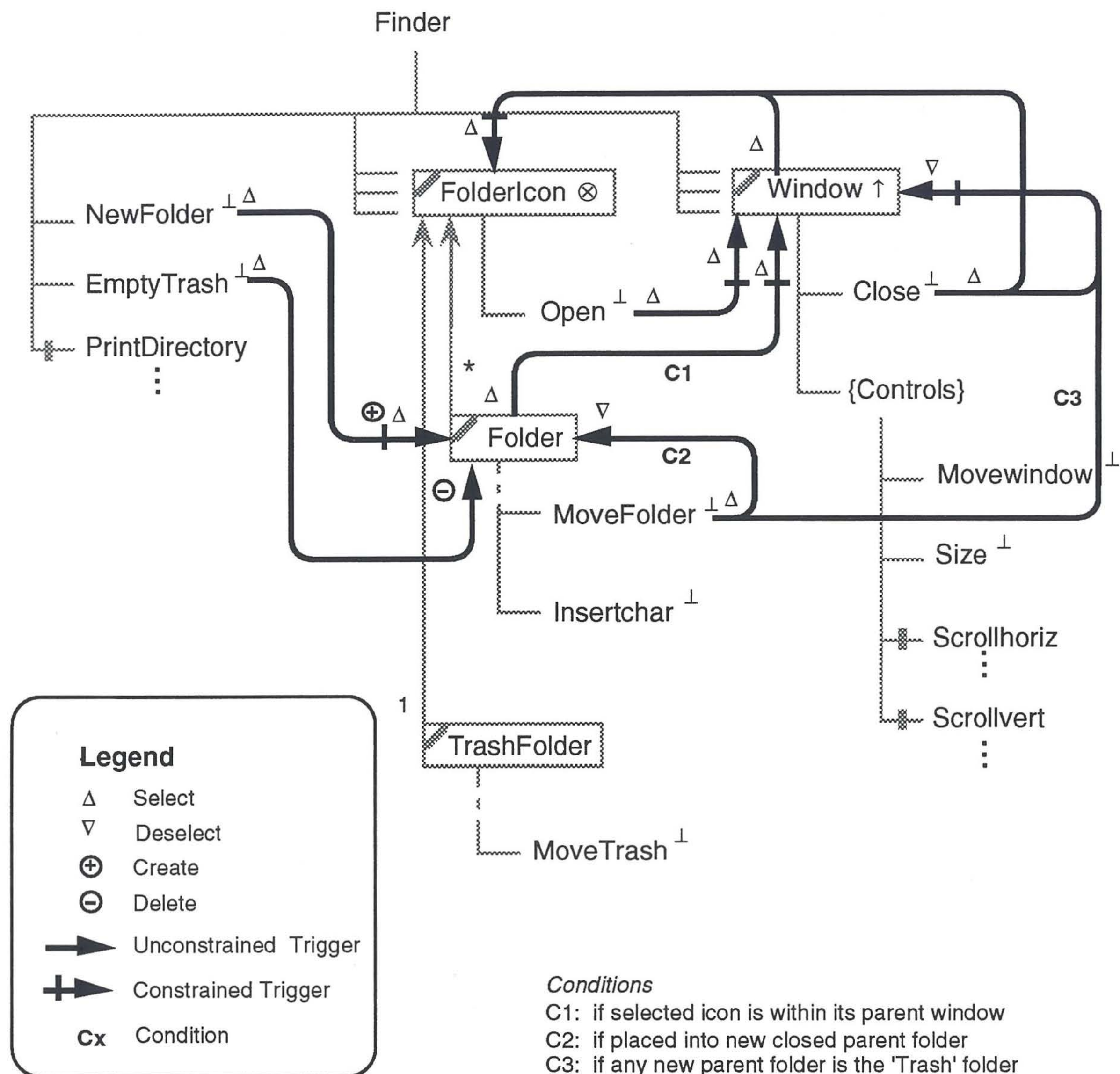


Figure 5.3: Finder: Selection trigger layer over greyed base layer diagram

The following notes refer to specific triggers, and should be read in conjunction with Figure 5.3 which shows the selection triggers for the Finder document folder system, three of which are conditional. Several of the triggers are constrained in order to prevent possible cycles involving knock on selections of folders and windows. The base layer tree is shown greyed. The annotation is defined by the legend.

- *NewFolder -> Folder trigger:* Selection of *NewFolder* triggers the creation and selection of a new folder instance (represented by a *FolderIcon/Folder* meneme pair), and also the creation of a new window instance (as a result of a mutual existence dependency - see Figure 5.5) which is neither available (open) nor selected (active). Any previously selected folder instance is deselected by virtue of the

mutual exclusivity of the folder instance set. A precondition for the availability of *NewFolder* is defined in the option precondition layer.

- *Folder -> Window trigger:* Selection of a folder triggers selection of the parent window, if the icon is within it (condition C1). It should be noted that selection of a folder within a parent window is equivalent to the following: selection of the parent window, which would trigger selection of the corresponding folder icon (see next trigger below) followed by selection of the child folder (which would deselect the parent folder icon by virtue of the mutual exclusivity of the dynamic folder instance set).
- *Window -> FolderIcon trigger:* Selection of a window causes selection of the corresponding folder icon. The link is at the *FolderIcon* type level, as it applies to all folders, and it results in the selection of a meneme pair at the type and subtype levels through selection propagation.
- *Open -> Window trigger:* Selection of *Open* makes available and selects a new window instance, which becomes the active window at the top of the window stack. (Opening the folder satisfies the precondition for availability for selection of the window instance. This is described in the option precondition layer.) Any previously active window is deselected by virtue of the mutual exclusivity of the dynamic window instance set. The behaviour associated with *Open*, at the *FolderIcon* level, is 'inherited' by both the *Folder* and *TrashFolder* icon subtypes.
- *Close -> Window / FolderIcon triggers:* Selection of *Close* makes unavailable and deselects the currently active window instance. It also selects the matching *FolderIcon/Folder* or *FolderIcon/TrashFolder* meneme pair through selection of *FolderIcon*. If at least one further open window exists (window stack > 1 on selection of *Close*) then the next window on the stack is selected and becomes active, because a selection from the open window set is required. (The capturing of the latter requirement also involves option preconditions, and is discussed at the end of the next subsection).
- *MoveFolder -> Folder / Window triggers:* Moving a folder icon may place it into a new closed parent folder (condition C2), in which case the folder icon is deselected. If the new parent is the Trash folder (condition C3), and the window corresponding to the moved folder is open, then this window is closed. (It should be noted that moving a folder into a new window area (i.e. into a new open parent folder) has the effect only of changing state variable values.)
- *EmptyTrash -> Folder trigger:* Emptying the Trash folder triggers the deletion of folder instances (and associated window instances). A precondition for the availability of *EmptyTrash* is defined in the option precondition layer.

The option precondition layer

An option precondition is a condition for the availability (for selection) of a meneme, which may involve the state of a meneme in another part of the dialogue, or the existence of some other state or condition, or both. An option precondition is thus indirectly an event precondition. The following notes, which refer to specific preconditions, should be read in conjunction with Figure 5.4 which shows the option preconditions for the Finder document window system over a greyed base layer tree. The annotation is defined by the legend.

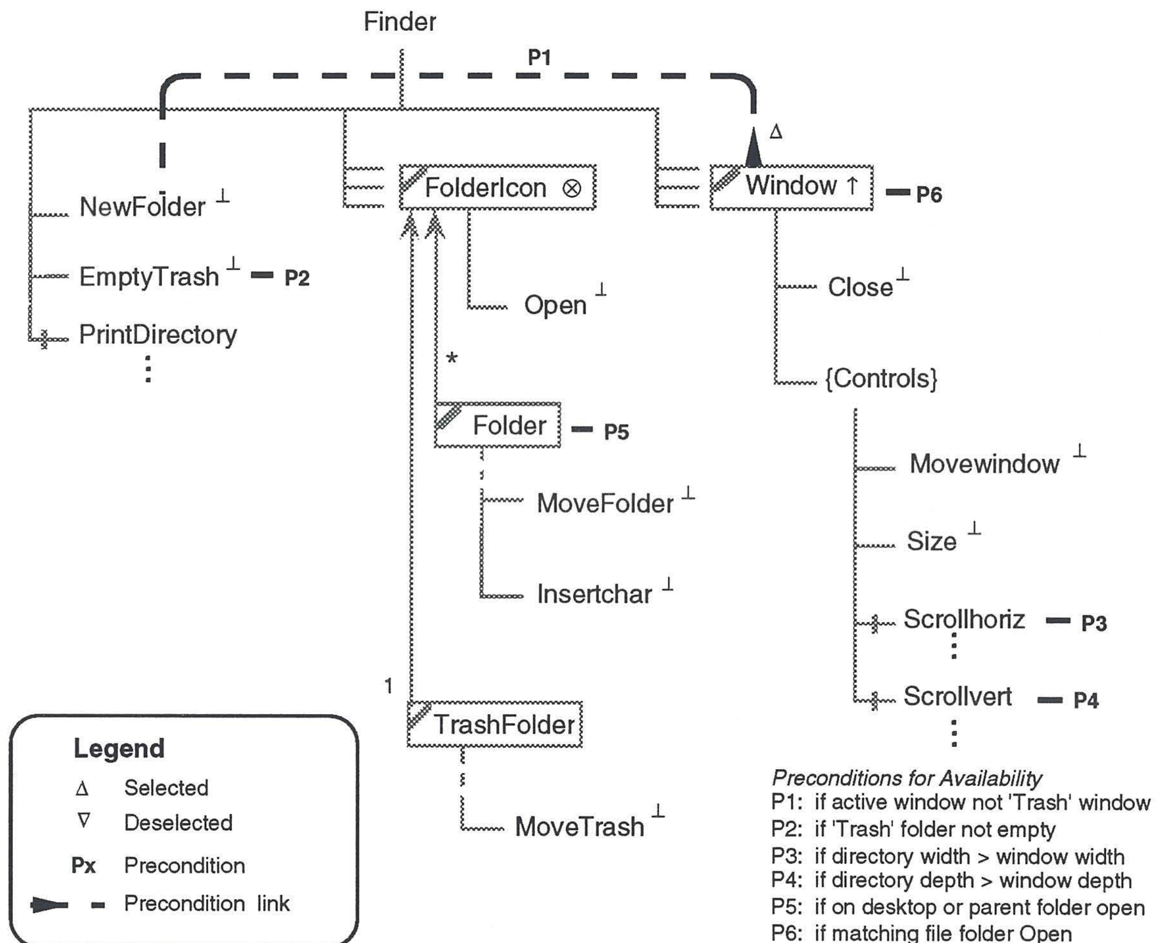


Figure 5.4: Finder: Option precondition layer over greyed base layer diagram

- *NewFolder:* This option is available only if an active window exists, i.e. a window instance is currently selected. A further constraint (P1) also applies, in that *NewFolder* is not available if the selected window is the Trash folder window.
- *EmptyTrash:* This option is available only if precondition P2 applies, that is, if the Trash folder is not empty.

- *Scrollhoriz:* This option is available only if precondition P3 applies, that is, if the width of the file directory is greater than that of the open window. (At the presentation level this determines whether the horizontal scroll box appears).
- *Scrollvert:* This option is available only if precondition P4 applies, that is, if the depth of the file directory is greater than that of the open window. (At the presentation level this determines whether the vertical scroll box appears).
- *Folder:* A particular instance of this generic object is available (for user selection) only if precondition P5 applies, that is, if the instance is on the desktop or its parent folder is open (additional spatial aspects would apply in a prototype or implementation).
- *Window:* A particular instance of this object is available (for user selection) only if precondition P6 applies, that is if the matching folder instance is in an open state.

Open windows

A note is required about open windows, which form a subset of the dynamic window instance set. This subset can be null and if not null carries the need for a required choice (as noted earlier). This information is captured by a combination of aspects of the specification in Figures 5.1, 5.3 and 5.4, as follows:

Base layer tree (Figure 5.1):

- Window instances are shown as being mutually exclusive. Selecting an available (open) window makes it active, and also causes the previously active window to be deselected.
- *Window* is designated 'select-only'. In particular, it is not possible for the user to deselect the only open window.

Selection trigger (Figure 5.3):

- Opening a folder selects the associated window. Windows are opened in an active state.

Option preconditions (Figure 5.4):

- All windows may be unavailable, i.e. no folders may be open.
- Opening a folder causes the associated window to be made available for selection.

Windows can thus be unavailable for selection (closed), available for selection (open), or selected (active).

The existence dependency layer

The existence of an object may depend upon the existence of another object in the same or different subdialogue, or upon the existence of a dialogue. Figure 5.5 shows the single mutual existence dependency in the Finder document folder system, between the

FolderIcon and *Window* object types. A *Window* instance cannot exist without a corresponding *FolderIcon* instance, and vice versa. The base layer diagram is shown greyed. Further examples of existence dependencies appear in connection with Microsoft Word in Section 6.2.

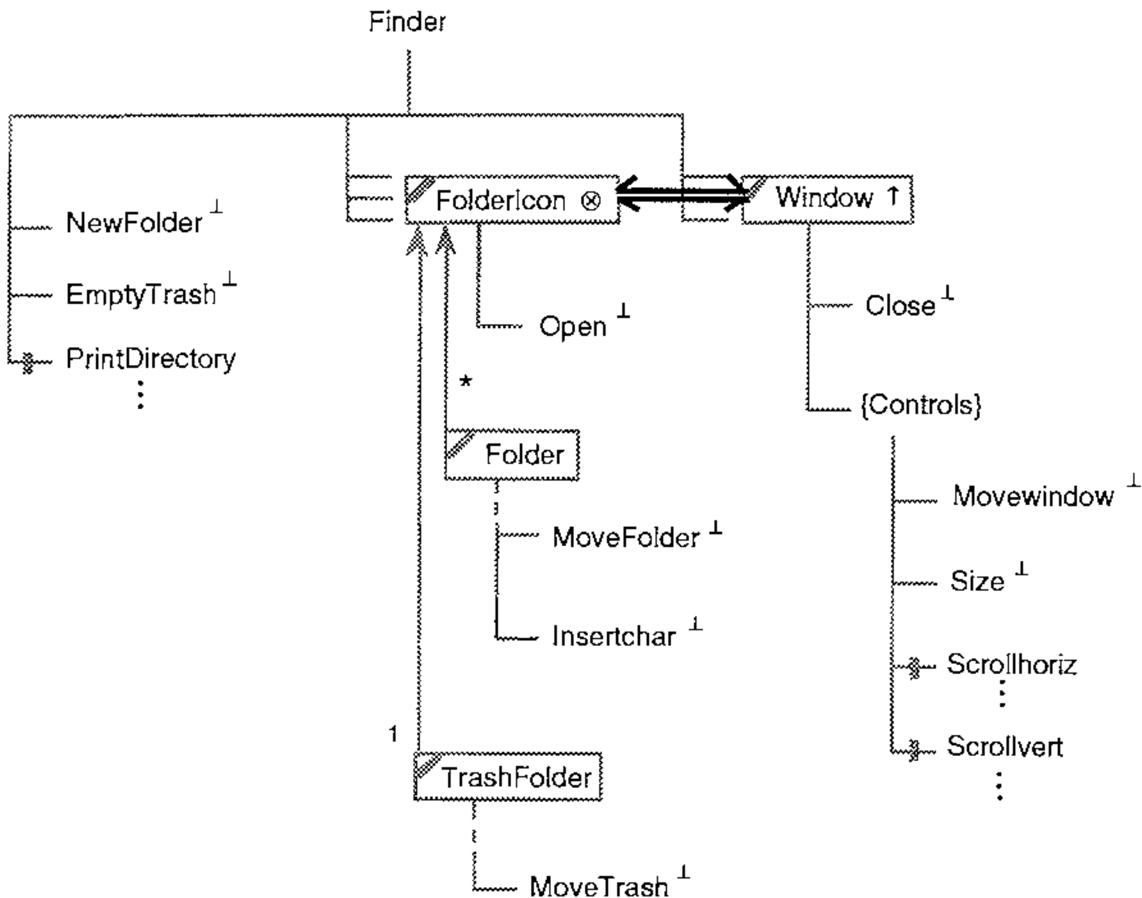


Figure 5.5: Finder: Existence dependency layer over greyed base layer diagram

5.4 An extended icon object hierarchy

The Finder system used above is limited to icons representing document folders, including the Trash folder. The primary purpose of the document folder system is to provide for the hierarchical organisation of application programs and documents to facilitate subsequent access to them. Figure 5.6 shows an extended Finder icon object hierarchy which includes application programs and application documents. Some renaming of icon types and subtypes has been undertaken in order to better describe the extended model, which paves the way for an examination of two Macintosh applications, Microsoft Word and MacDraw, in Chapter 6. With reference to this diagram:

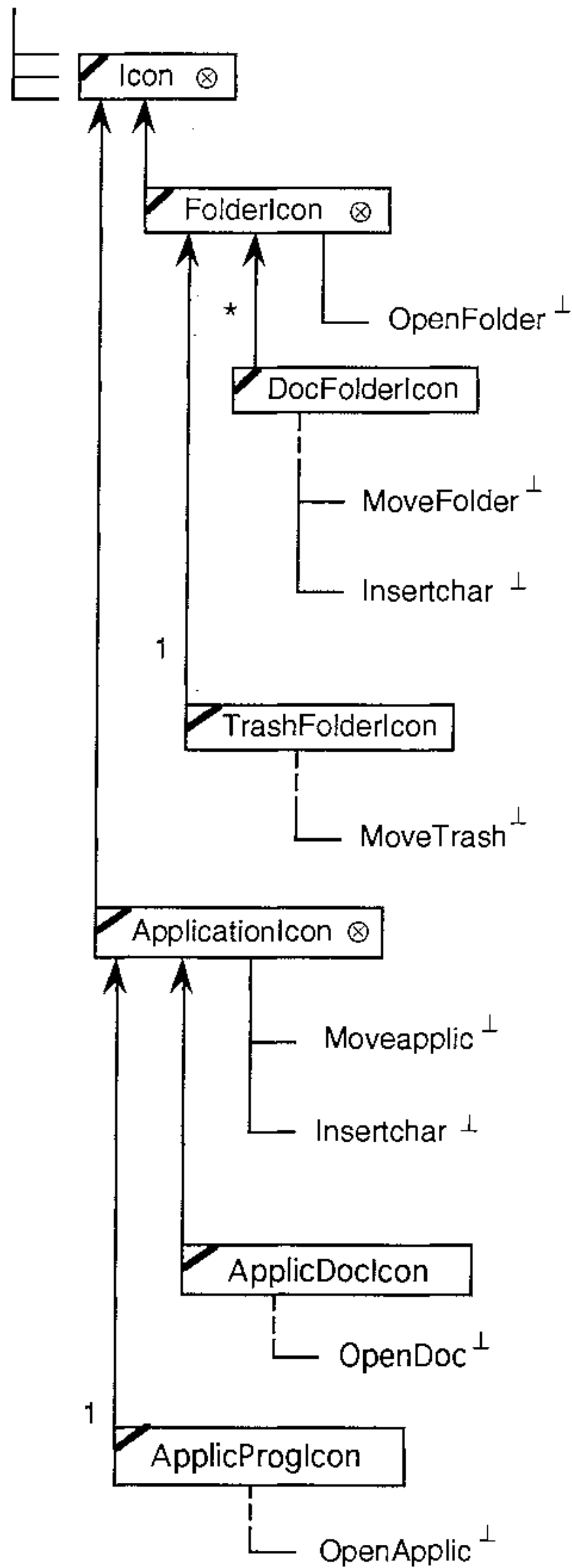


Figure 5.6: Extended Finder icon object hierarchy

- An additional icon subtype *ApplicationIcon* has been added. This subtype and the sub-subtypes *ApplicProgIcon* and *ApplicDocIcon* are further examples of syntactic objects. While many instances of *ApplicDocIcon* may exist for a given application, *ApplicProgIcon* is shown with a cardinality of one. The initial OSVs attaching to these subtypes are shown in Table 5.2.
- The opening of either *ApplicProgIcon* or *ApplicDocIcon* triggers selection of the application concerned. These links are considered in Section 6.1.
- Only terminal object subtypes are selectable by the user in this example, with selection being propagated up the object type hierarchy. The selection of *ApplicDocIcon*, for example, would also select *ApplicationIcon* and *Icon*, thus providing access to the options in the subdialogues at each level. *Icon*, *FolderIcon* and *ApplicationIcon* are represented by passive menemes.
- It is possible to draw an alternative tree with *ApplicationIcon* and *DocFolderIcon* as sub-subtypes of a new subtype *FileIcon*, as they have some options (e.g. *Move* and *Insertchar)* and state variables (e.g. availability status and parent folder name) in common. In this alternative version *TrashFolderIcon* would be a subtype on its own. The representation in Figure 5.6 is preferred because it groups icons more logically, even though overall there is slightly more duplication.

ApplicationIcon:		**Subtype of Icon**
	#iname	object name
	aobsubtype	object subtype = application icon
	aavstat	object availability status (available, not available)
	aparent	parent folder name
ApplicDocIcon:		**Subtype of ApplicationIcon**
	#iname	object name
	adobsubtype	object subtype = application document icon
ApplicProgIcon:		**Subtype of ApplicationIcon**
	#iname	object name
	apobsubtype	object subtype = application program icon

Table 5.2: Application icon related OSVs

The icon object hierarchy can be further extended to incorporate diskettes. Figure 5.7 shows the Lean Cuisine+ base layer tree for an extended document folder system which includes both application and diskette icons.

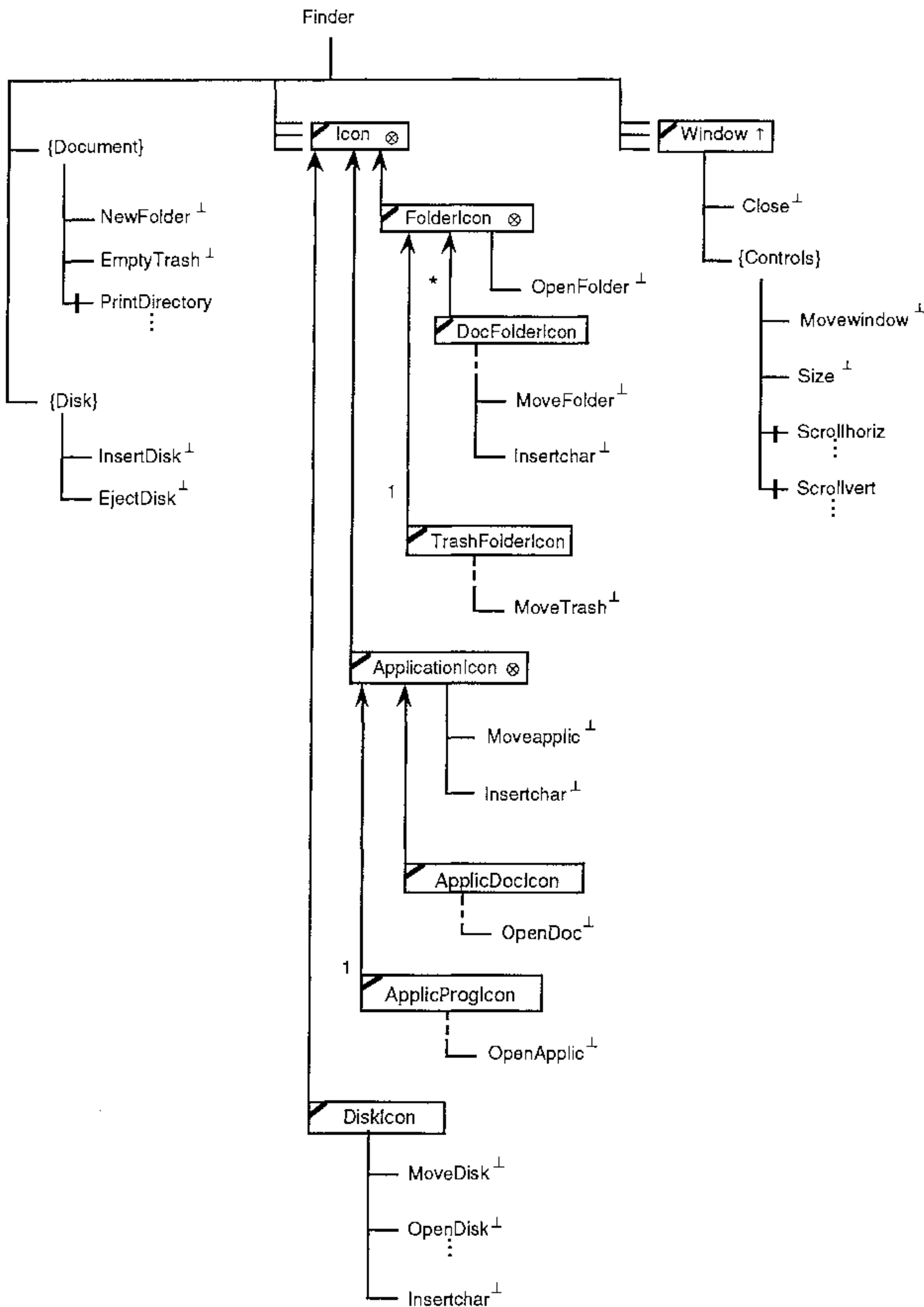


Figure 5.7: Base layer tree diagram for extended Finder document folder system

In connection with diskettes, a further *Icon* subtype, *DiskIcon*, and associated subdialogue have been added to the icon object hierarchy, and two modal subdialogues, *InsertDisk* and *EjectDisk*, have been added to the root dialogue. The *OpenDisk* subdialogue is shown in full in Figure 5.8 and discussed more fully below. Figure 5.9 shows the additional diskette-related selection triggers, and Figure 5.10 the additional diskette-related option preconditions. The selection of *InsertDisk* (through disk insertion) triggers the creation and selection of an instance of *DiskIcon*. Moving an instance of *DiskIcon* may place it 'into' another disk folder, triggering the creation of an equivalent hierarchy of *DocFolderIcon*, *ApplicDocIcon*, and *ApplicProgIcon* instances within the target disk folder, and the deselection of the *DiskIcon* instance.

System directives

The *OpenDisk* modal subdialogue, shown in Figure 5.8, offers two possible outcomes including one which involves a conditional system directive to the user:

1. Provided a diskette is currently inserted, selection of *OpenDisk* makes available and selects a new *Window* instance. Here *OpenDisk* behaves as a standard simple modal subdialogue terminated by the system on completion of the operation.
2. If the diskette is not currently inserted a further modal subdialogue, *ReInsertDisk*, is triggered, which results in the output of a system directive (⇒). This prevents the *OpenDisk* subdialogue from completing, pending user action. The deselection of *ReInsertDisk* by the user (through insertion of the disk) completes the modal subdialogue, permitting *OpenDisk* to proceed to a successful conclusion.

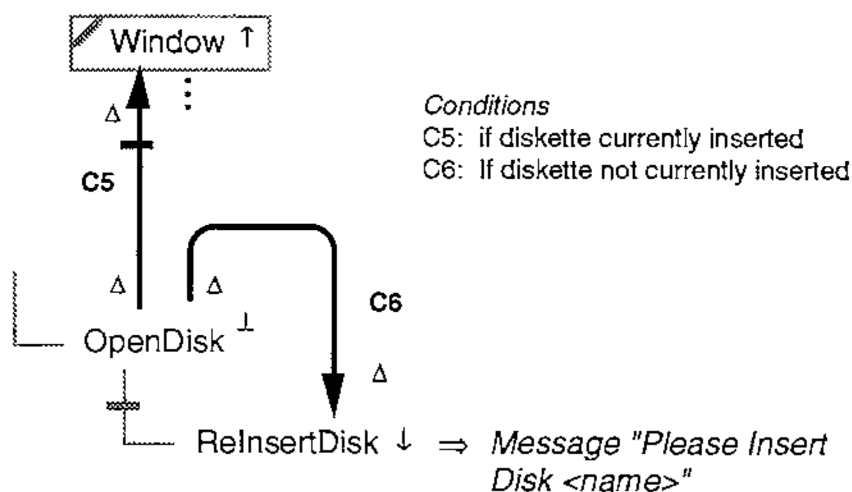


Figure 5.8: *OpenDisk* subdialogue involving a system directive

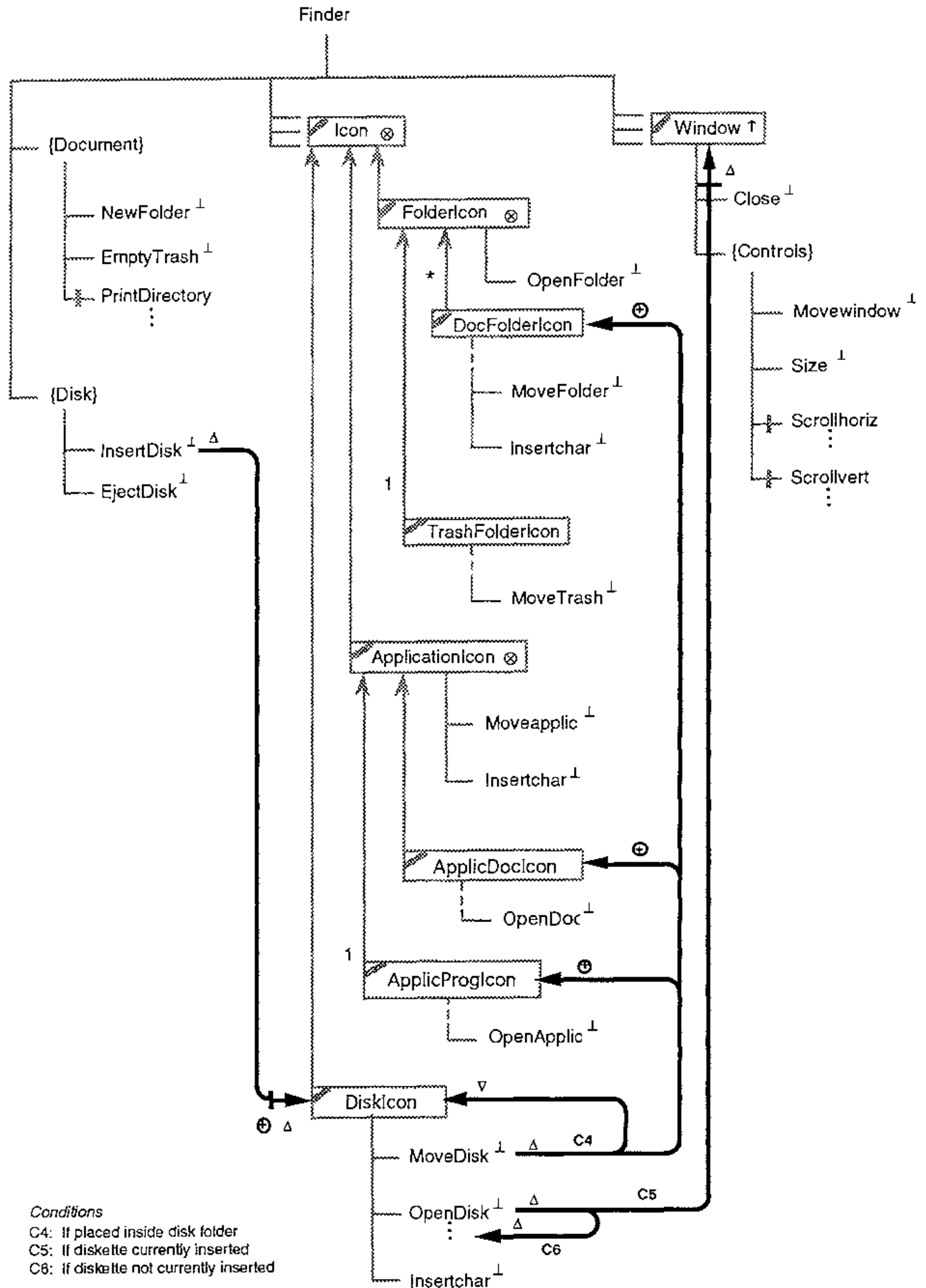


Figure 5.9: Finder diskette-related selection triggers

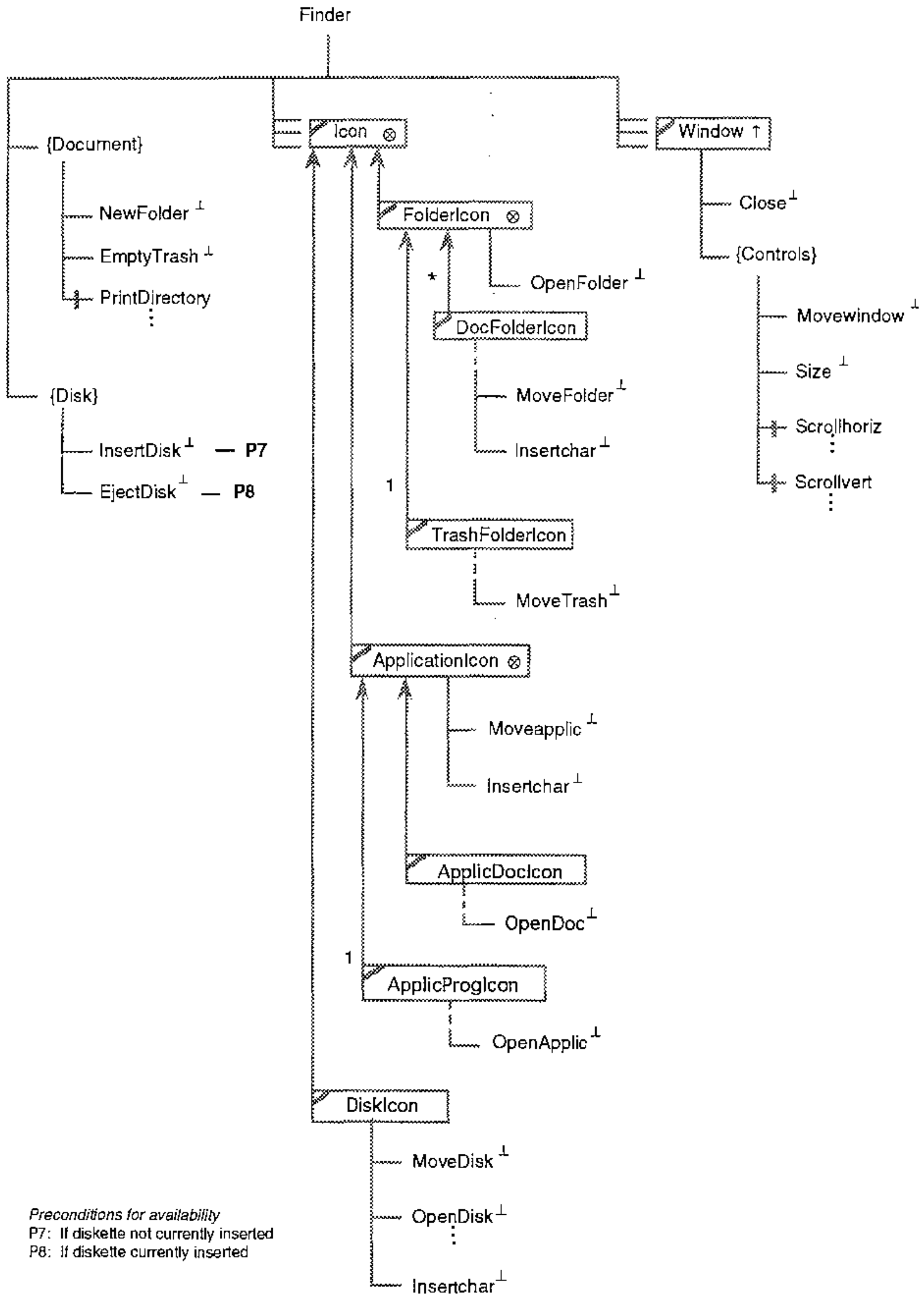


Figure 5.10: Finder diskette-related option preconditions

This example extends the usage of monostable menemes, which thus far have been used only for simple subdialogues at terminal nodes, and provides for the description of subdialogues which normally behave as simple modal subdialogues, but which under certain conditions present further dialogue options accompanied by system directives.

System directives may also be associated with errors and warnings. On the assumption that in a direct manipulation interface only valid options are presented for selection in each state, error messages and warnings will be associated with either:

- invalid keyboard input from the user;
- computation errors; or
- system errors e.g. space problems, disk read errors.

These can be handled by Lean Cuisine+ within subdialogues triggered when certain conditions hold, as shown in the above example for diskette insertion.

5.5 The addition of menus

Menus have not been considered so far in the Finder example, but were the focus of the original Lean Cuisine notation. Menus provide an orthogonal grouping of the selectable options present in an interface to facilitate navigation and control. The grouping of menemes within menus may be based on a functional rather than an object-oriented view (Lewis, 1991) as an aid to navigation, and may include selectable representations of state variable values. The behaviour of the Finder menu system can be described in a further base layer Lean Cuisine+ tree diagram, with menus represented as syntactic objects and individual options represented by terminal menemes. A meneme can thus belong to more than one interface object, e.g. *Open* can belong both to an *Icon* and to the *File* menu, and can appear in more than one tree diagram. This would be captured in the system dictionary.

Constraints already defined for menemes in the Finder interface specification described in Sections 5.2 and 5.3 apply in the specification of the menu system. The availability for selection of the menemes which form part of the Finder *File* and *Special* menus is specified partly in the base layer tree diagram (Figure 5.1), and partly in the option precondition layer (Figure 5.4), as shown in Table 5.3. The ordering and grouping of these menemes within the Finder menus is shown in Figure 5.11. Additional syntactic objects in the form of *File* and *Special* menus have been introduced, and the option preconditions listed above are shown, with the addition of designators P9 and P10.

Option	Precondition	Figure	Designation
Open	if a folder icon is selected	5.1	implicit
Close	if a window is active	5.1	implicit
NewFolder	if the active window is not 'Trash'	5.4	P1
EmptyTrash	if the 'Trash' folder is not empty	5.4	P2
PrintDirectory	None		

Table 5.3: Finder menu option preconditions

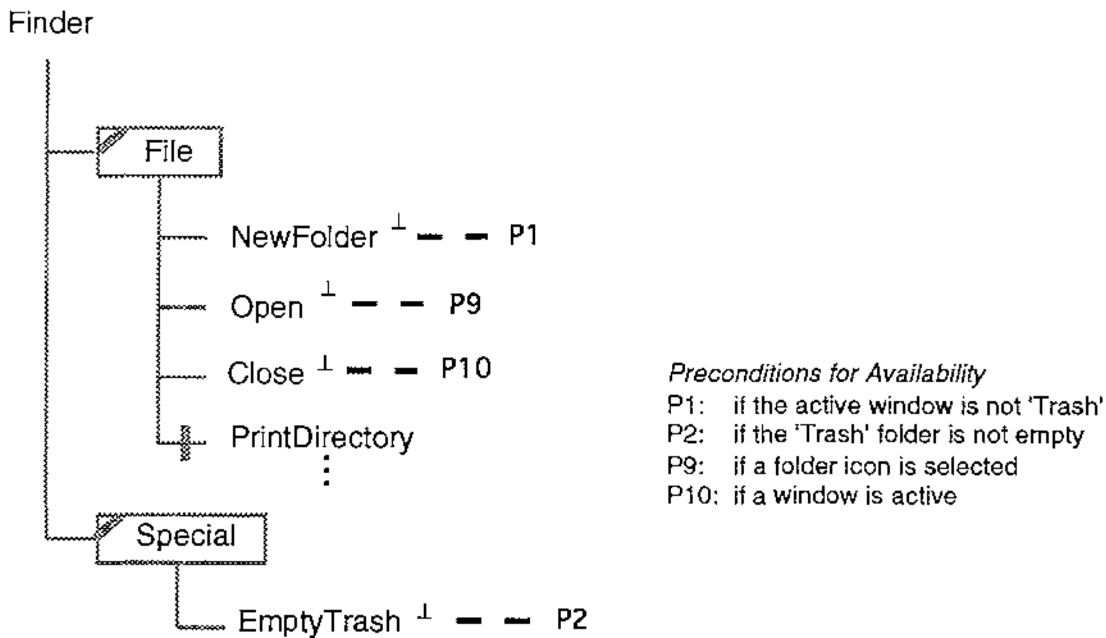


Figure 5.11: Finder menus showing option preconditions

5.6 Review

Through description of a simplified version of the Finder document folder system, the basic layered Lean Cuisine+ notation has been developed, and some important issues have been addressed. The object range encompassed by the notation, which is limited at this stage, is extended in Chapter 6. A number of dialogue components were identified in Section 5.1 as necessary in the construction of a hierarchical dialogue description. These have been implemented in the Lean Cuisine+ notation as described below, and as shown in Figure 5.12:

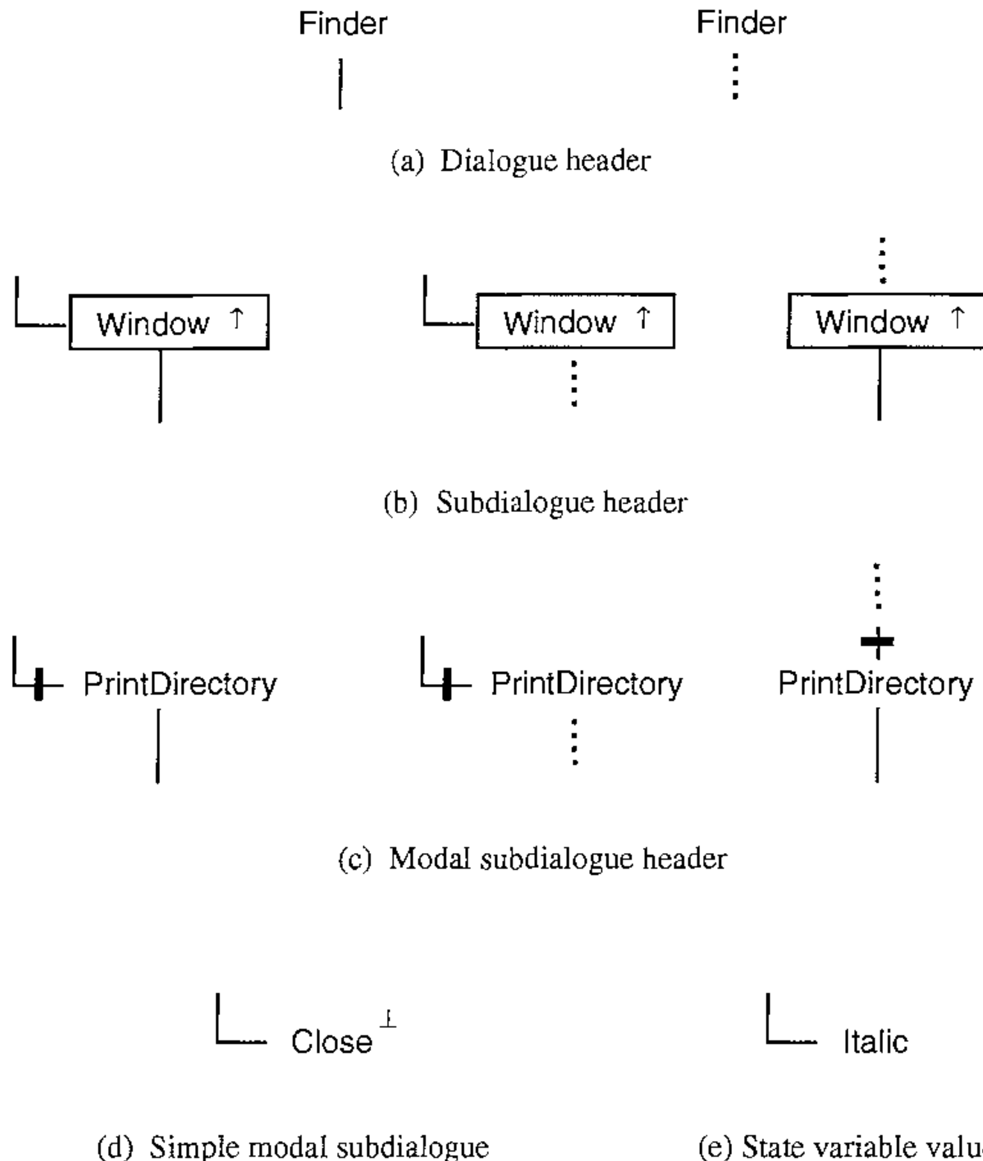


Figure 5.12: Lean Cuisine+ structural dialogue components

(a) *Dialogue header:* this is the root node in a Lean Cuisine+ dialogue tree. When selected, a dialogue header provides access to the subdialogue(s) and/or state variable values immediately below it, subject to any constraints applying. To support stepwise refinement, the tree below the root node may be represented by an ellipsis to indicate that the detail of the dialogue is described in a separate diagram.

(b) *Subdialogue header:* this is an intermediate node heading a subtree, and representing either an object or operation. When selected, a subdialogue header provides access to further dialogue options. Options in the parent dialogue remain available, subject to any constraints applying. As in the case of dialogues, the subtree below the subdialogue node may be represented by an ellipsis to indicate

that the detail is described in a separate diagram. The lower level diagram is then headed by an ellipsis in order to complete the association.

(c) *Modal subdialogue header:* this is a subdialogue header which when selected provides access to subdialogue options, and additionally makes options in all other parts of the dialogue unavailable. This is indicated by a bar across the arc above the subdialogue node.

(d) *Simple modal subdialogue:* this is a node involving a primitive operation, and represented by a monostable meneme, shown thus (⊥). Monostable menemes remain selected only for the duration of the operation, and revert to an unselected state on its completion. Simple modal subdialogues are normally terminal nodes in which the subdialogue is system terminated on completion of the operation, but may exceptionally have a further modal subdialogue associated with them which is triggered under certain conditions, and accompanied by a system directive. In such cases the operation becomes suspended pending user termination of the further subdialogue, when the operation completes in the usual way.

(e) *State variable value:* this is a terminal node representing an object state variable value which can be selected or deselected within that subdialogue. The value may remain set after termination of the subdialogue, and may be referenced in other subdialogues, subject to any scoping restrictions within object type hierarchies.

Two further structural issues are worthy of mention:

1. *Selection propagation within the object type hierarchy.* This has been considered in this chapter in relation to the Finder example, in which bi-directional propagation occurs and in which subtypes 'inherit' the state variables and behaviour of their parent type. An example which involves the inhibition of selection propagation appears in Section 6.3 in connection with the MacDraw dialogue.
2. *The use of virtual menemes.* In developing the original Lean Cuisine notation for menus, Apperley & Spence (1989) headed each meneme subgroup with a virtual meneme. Subgroup headers are not however essential and can be omitted without changing the meaning of the tree diagram. The approach adopted in Lean Cuisine+, and used in the examples in this thesis, has been to limit the use of virtual menemes. They are used only to support stepwise refinement, and to improve clarity. It is possible that group headers could be required in subsequent mapping to an implementation notation, but the view taken is that it is better to use them sparingly to avoid cluttering Lean Cuisine+ diagrams, and to introduce them as and if necessary at a later design stage.

Chapter 6

Extending the Lean Cuisine+ Object Range

In this Chapter aspects of two Macintosh applications are described in Lean Cuisine+ in order to extend the object range encompassed by the notation. In Section 6.1, triggers from the Finder dialogue of Chapter 5 to these application dialogues are defined. Section 6.2 describes aspects of Microsoft Word, and Section 6.3 part of MacDraw. In Section 6.4 the Finder-application linkage is completed, while Section 6.5 adds a top level dialogue interlink diagram (DID). Section 6.6 reflects on the notation and defines three levels of abstraction based on an object classification.

6.1 External selection triggers

In Section 5.4 an extended Finder icon object hierarchy incorporating application programs and documents was developed. This is reproduced in Figure 6.1 with the addition of selection triggers. The opening of either *ApplicProgIcon* or *ApplicDocIcon* triggers a link between the Finder and the application concerned. These external selection triggers are named (A1 and A2) and broken, to indicate that they relate to separate dialogues, and are directed to the root node of the target dialogue. Opening *ApplicProgIcon* triggers the opening of the application and the creation of a new unnamed document, whereas opening *ApplicDocIcon* triggers the opening both of the application and the selected application document, with the document name being passed as a parameter. Both triggers will cause selections to occur within the body of the target dialogue, as shown in Figure 6.4 for Microsoft Word. An extended version of Figure 6.1, which includes a trigger from the application to the Finder, appears in Figure 6.11.

6.2 The Microsoft Word dialogue

Microsoft Word (Microsoft, 1991) is a comprehensive word processing application available on the Macintosh. In this section certain basic aspects of the Microsoft Word dialogue (version 5.0), involving both syntactic and semantic objects, are described in Lean Cuisine+. Many features are omitted as they would add complexity to the description without revealing further aspects of the Lean Cuisine+ notation. The dialogue is considered at two levels:

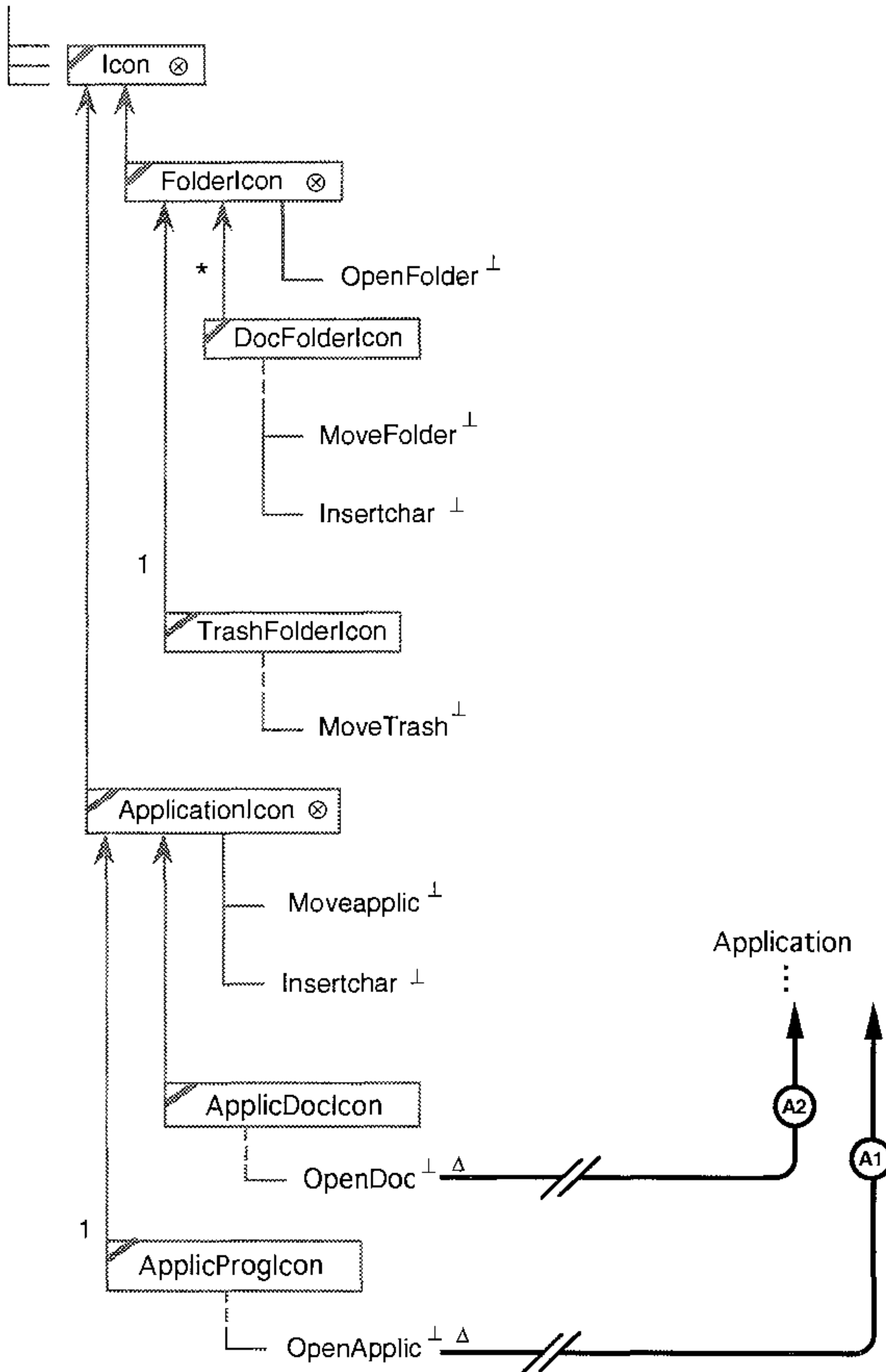


Figure 6.1: Finder icon object hierarchy showing external triggers

- *Level 1:* provides for the manipulation of text documents, which can be created and opened in the form of document windows. Documents can be saved or printed, and windows can subsequently be closed. A range of window controls similar to those in the Finder exists.
- *Level 2:* applies Lean Cuisine+ to the description of objects and options relating to the *content* of the document. These semantic objects exist *within* the working space provided by the document and its associated window.

Figure 6.2 shows a base layer tree diagram describing certain aspects of the Microsoft Word dialogue. The two levels defined above are clearly shown. With regard to this diagram:

- *Level 1:* The *Word* root dialogue provides access to the mutually exclusive group of *Open* and *New* modal subdialogues, and to the *DocWindow* subdialogue, headed by a syntactic object, which is mutually compatible with this group. The fork symbol shows that *DocWindow* represents one of a mutually exclusive dynamic object instance set of document windows. The set carries an unassigned 'dynamic default' (•), implying that one of the set is in a selected state at the commencement of the dialogue. The *DocWindow* subdialogue provides access to further options, including *Close* and a range of window controls. Also associated with each *DocWindow* instance is a semantic *Document* object. This is represented by a 'default choice' and 'passive' meneme, which means that the *Document* meneme is selected whenever the *DocWindow* meneme is selected, and that it remains selected while the window is open (as it cannot be deselected by the user independently of the window). An open window thus always has a selected document associated with it. At Level 1, the *Document* subdialogue provides access to the mutually exclusive *Print* and *Save* subdialogues, i.e., a document can be saved or printed.
- *Level 2: Document* is shown as a composite object consisting of a mutually compatible set of semantic *Character* objects. One or more characters may thus be selected at any time. It must be assumed that the set has a sequence associated with it. A *Document* is opened with an initial insertion point which can subsequently be directly altered through selection of *Changeinsertpt*, and indirectly altered through selection of any of *Insertchar, Inserttab, Insertnewline, Insertpagebrk, Paste*, or *Deletelastchar*. Characters can be added one at a time to the document via *Insertchar* (with the character value as a parameter). With reference to the task analysis of Section 4.1, *Insertchar* is the basic 'character' task, the repetition of which makes up the compound 'text' task. Selected characters can be directly *Cut* or *Copied*, and indirectly cut and replaced by the subsequent selection of any of *Insertchar, Inserttab, Insertnewline, Insertpagebrk* or *Paste*.

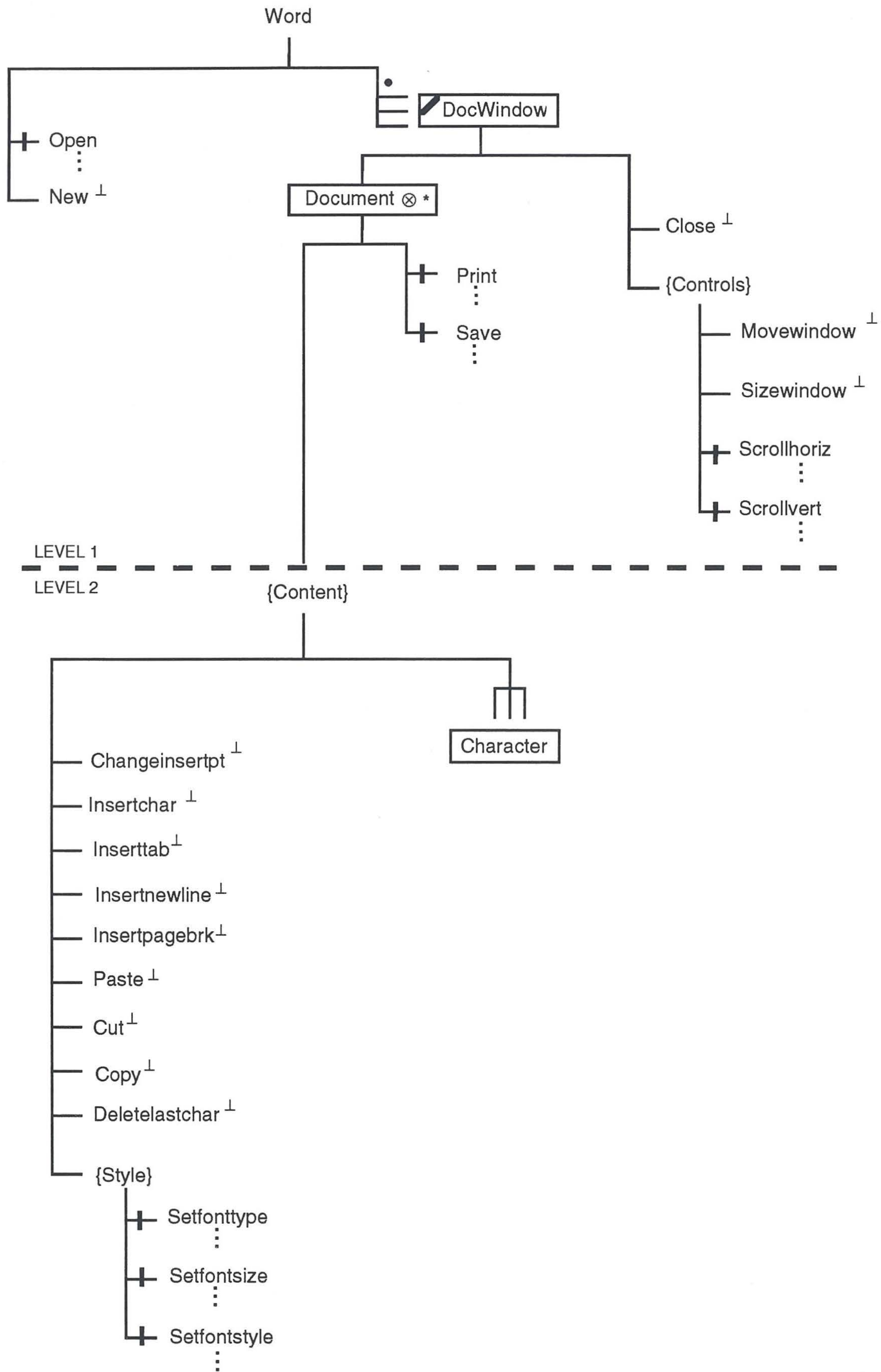


Figure 6.2: Microsoft Word: base layer tree diagram

Modal subdialogues to alter the type, size and style of characters are shown but not developed at this stage. These subdialogues can either be applied to selected characters, or used to change the style in force at the current insertion point.

Object state variables

An initial set of OSVs relating to Microsoft Word, grouped by object type, is shown in Table 6.1. They are intended to be indicative at this stage.

DocWindow:	#dwname	object name (=#dname)
	dwobclass	object class = syntactic
	dwobtype	object type = document window
	dwavstat	object availability status (open, closed)
	dwselstat	object selection status (active, inactive)
	dwposition	current window position
	dwwidth	current window width
	dwdepth	current window depth
	dwdwidth	current overall directory width
	dwddepth	current overall directory depth
Document:	#dname	object name
	dobclass	object class = semantic
	dobtype	object type = document
	davstat	object availability status (open, closed)
	dselstat	object selection status (active, inactive)
	dinsertpoint	current insertion point
	dsize	current document size
	dfonttype	font type at current insertion point
	dfontsize	font size at current insertion point
	dfontstyle	font style at current insertion point
Character:	#cname	object name
	cobclass	object class = semantic
	cobtype	object type = character
	cselstat	object selection status (selected, unselected)
	cfonttype	character font type
	cfontsize	character font size
	cfontstyle	character font style

Table 6.1: OSVs for Microsoft Word

Selection trigger layer

Figures 6.3 and 6.4 show selection triggers for the Microsoft Word dialogue. Figure 6.3 shows triggers *internal* to the dialogue, while Figure 6.4 displays named *external* triggers associated with the interface between Word and the Finder.

The following notes refer to Figure 6.3. It should be noted that the deselection of all currently selected objects which results from the deselection of the *Word* root node (i.e. from termination of the dialogue) is not shown:

- *Open -> DocWindow link:* Selection of *Open* causes a modal subdialogue to be initiated. A trigger within it causes the corresponding *DocWindow* instance to be made available (opened) and selected (made active i.e. placed at the top of the window stack). The corresponding *Document* is also selected by virtue of being a default choice (*) within the *DocWindow* subdialogue. Any previously selected *DocWindow* instance is deselected by virtue of mutual exclusivity.
- *New -> DocWindow link:* Selection of *New* triggers the creation of a new untitled *DocWindow* instance (and also of a new *Document* instance, as a result of an existence dependency - see Figure 6.6), which is available (open) and selected (active). Any previously selected *DocWindow* instance is deselected by virtue of the mutual exclusivity of the *DocWindow* instance set.
- *Close -> DocWindow link:* Selection of *Close* makes unavailable and deselects the currently active *DocWindow* instance. It also deselects the associated *Document* instance, and any selected characters. If at least one further open window exists then the next window on the stack is selected and becomes active, because a selection from the open window set is required.
- *Insertchar/Inserttab/Insertnewline/Insertpagebrk -> Character links:* Selection of any of these subdialogues adds a character to the document at the current insertion point. Additionally, any previously selected characters are deselected and deleted.
- *Paste -> Character link:* Selection of *Paste* inserts text characters from the Clipboard. Additionally, any previously selected characters are deselected and deleted. *Paste* is only available when the Clipboard is not empty (the precondition is defined in the option precondition layer).
- *Cut -> Character link:* Selection of *Cut* deselects and deletes the currently selected *Character* instance(s).
- *Deletelastchar -> Character link:* Selection of *Deletelastchar* deletes the character immediately preceding the insertion point.

The following notes refer to the external triggers shown in Figure 6.4. The two triggers *from* the Finder appeared in Figure 6.1, and *all* Finder-related external triggers are shown in Figure 6.11.

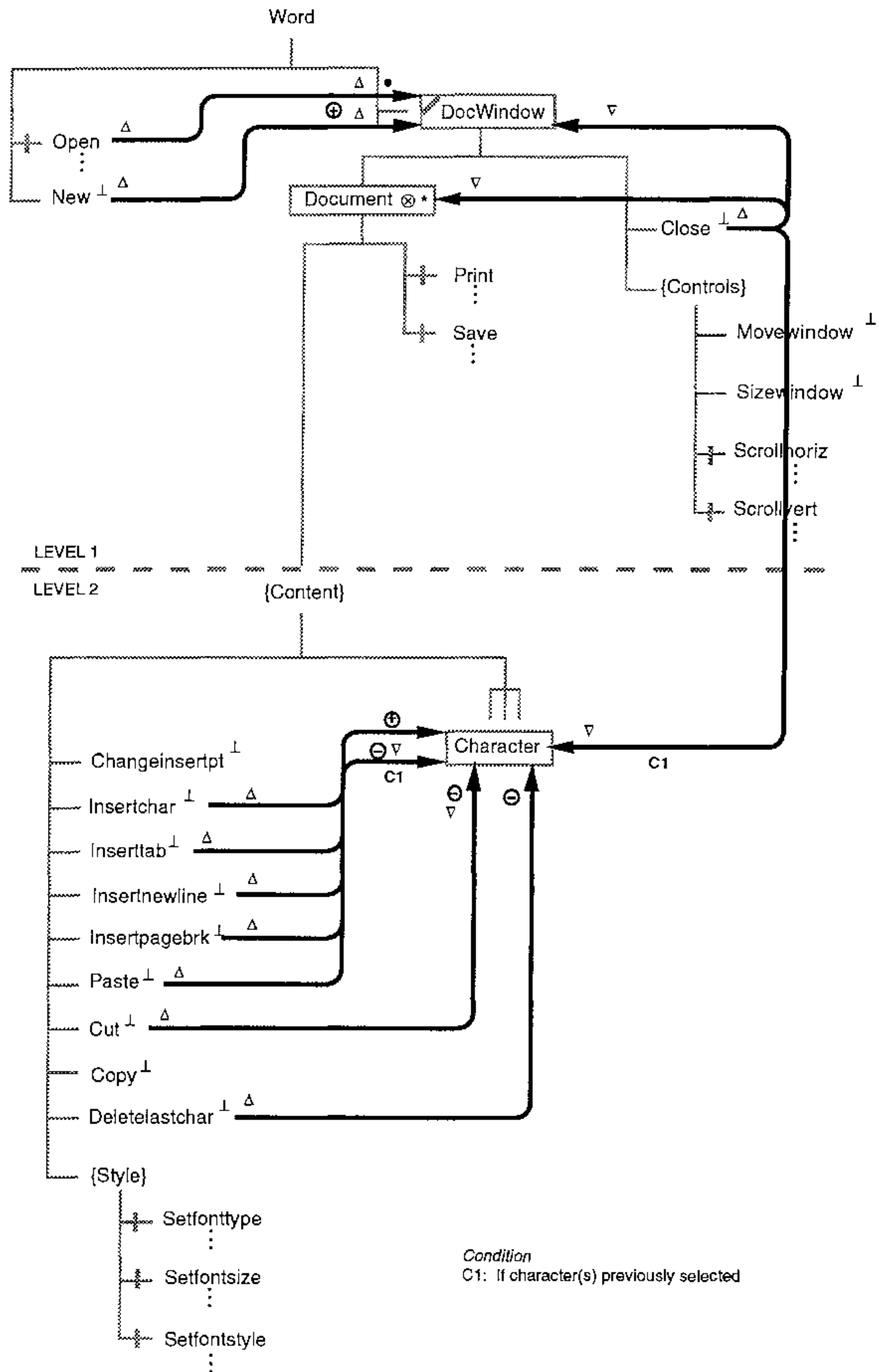


Figure 6.3: Microsoft Word: Internal selection triggers

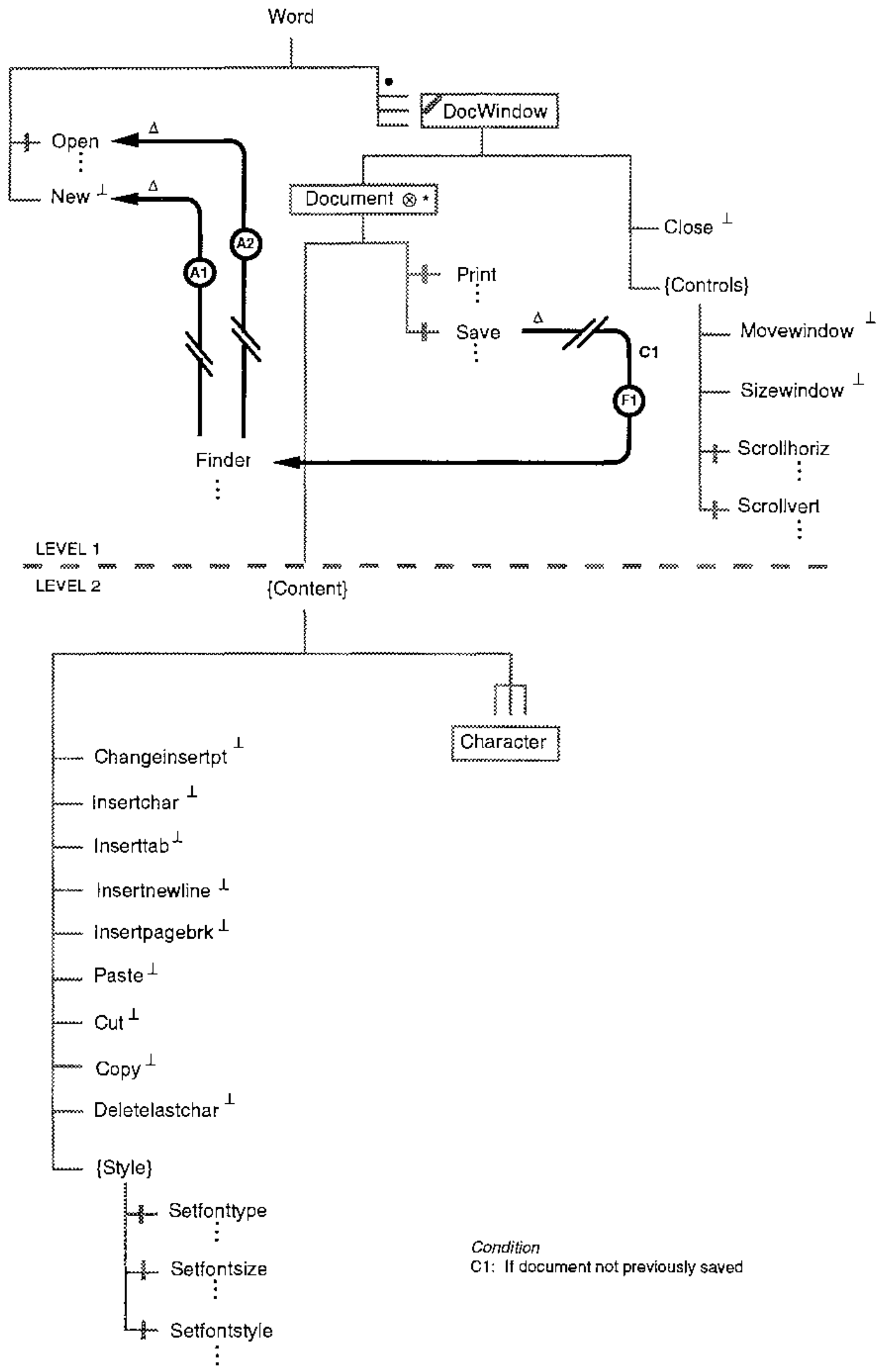


Figure 6.4: Microsoft Word: External selection triggers

- *Finder (OpenApplic) -> New link:* This link (A1) from the *Finder* dialogue triggers selection of the *New* subdialogue, which in turn triggers the creation of a new unnamed document window.
- *Finder (OpenDoc) -> Open link:* This link (A2) from the *Finder* dialogue triggers selection of the *Open* subdialogue, with the document name being passed as a parameter. This in turn triggers the opening of the corresponding document window.
- *Save -> Finder link:* This conditional link (F1) to the *Finder* subdialogue triggers the creation of a corresponding instance of *ApplicDocIcon* (see Figure 6.11) the first time a *Save* command is issued.

It should be noted that:

(i) The two Finder-initiated triggers also select the *Word* root dialogue header (which starts up the application). The selection of ancestors by triggers is considered to be implicit (see Section 5.3) and is not therefore shown on the diagram.

(ii) Quitting the application by deselecting the *Word* root dialogue header triggers the selection of the *Finder* dialogue. In the interests of hiding unnecessary detail, this is not shown here. The linkages between dialogue headers (and therefore between dialogues) are shown in a separate top level dialogue interlink diagram (see Section 6.5).

Option precondition layer

Figure 6.5 shows option preconditions for the Microsoft Word dialogue. The preconditions are listed on the diagram, and are self explanatory.

Existence dependency layer

Figure 6.6 shows existence dependencies for the Microsoft Word dialogue. External dependencies are shown between *ApplicProgIcon* and *Word*, and between *ApplicDocIcon* and *DocWindow*. The icons appear in the *Finder* dialogue (see Figure 6.1). The first of these is a mutual dependency, but the second is not, as a new *DocWindow* instance can exist independently until saved for the first time. A mutual dependency internal to the dialogue, between *DocWindow* and *Document*, is also shown.

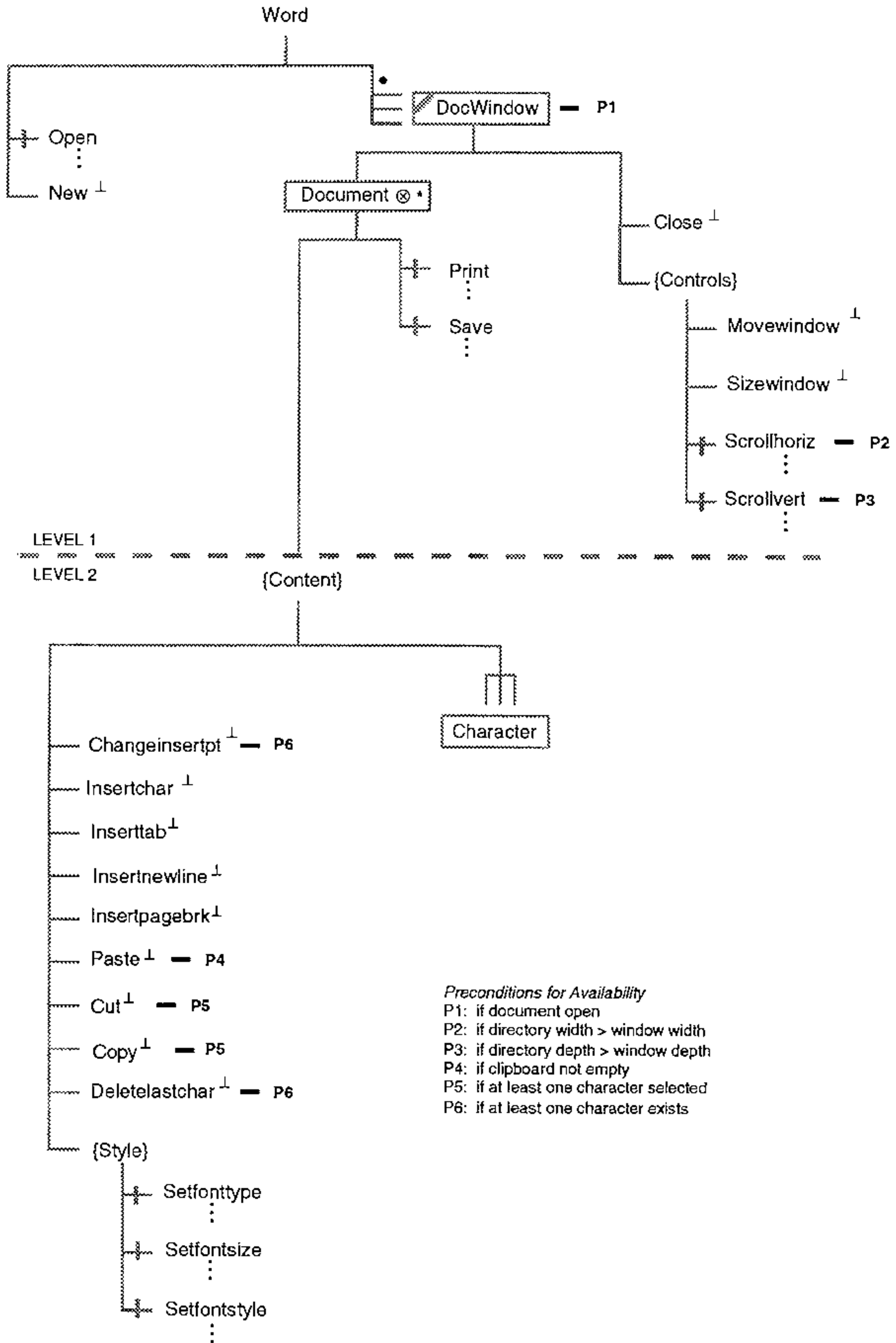


Figure 6.5: Microsoft Word: Option preconditions

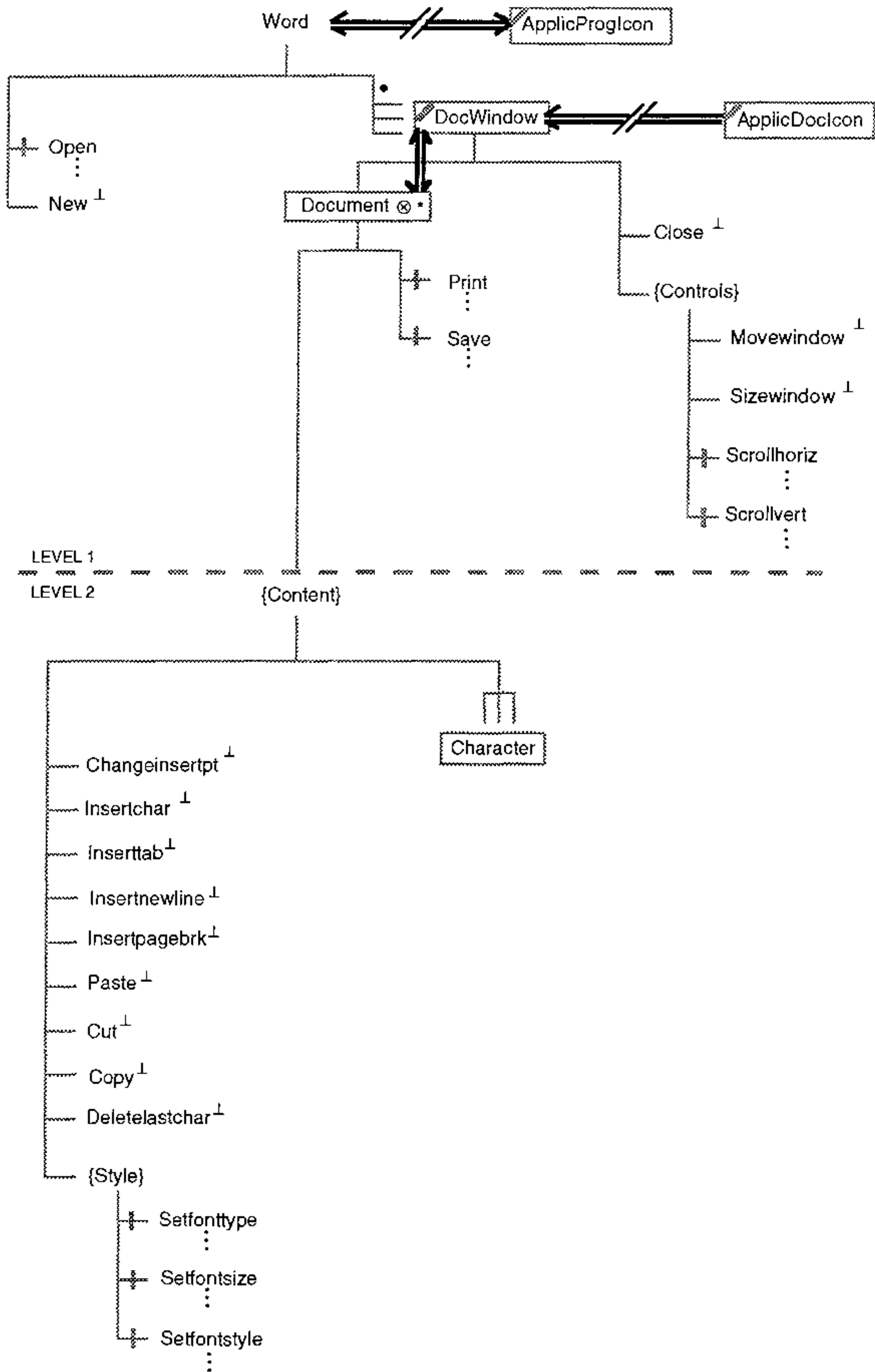


Figure 6.6: Microsoft Word: Existence dependencies

6.3 The MacDraw dialogue

MacDraw II (Claris, 1988) is a sophisticated object-based drawing application available on the Macintosh, providing a set of tools which can be used to build diagrams. In addition to the usual Macintosh pull-down menus offering a range of *File*, *Edit* and *Style* options, MacDraw offers on-screen menus providing for the selection of drawing tools and 'fill in' patterns. Version 1.1 is considered here.

As in the case of Microsoft Word, the base layer tree representing a MacDraw dialogue can be considered at two levels. The level 1 behaviour is similar to that of Microsoft Word, that is, documents can be created, opened, and displayed in windows, which when active offer a range of coutrols. Documents can also be saved and printed. Level 1 is not considered further here. Aspects of level 2 of the MacDraw dialogue, which extends Lean Cuisine+ to the description of *graphical* objects, are considered below. Other features are omitted, as they would add complexity to the description without revealing further aspects of the Lean Cuisine+ notation.

With regard to level 2 of the base layer tree diagram, shown in Figure 6.7:

- A mutually exclusive set of tools is offered, each of which places a particular interpretation on user input (i.e. represents a particular mode of operation). Three of these tools, *Selectobj*, *Writetext*, and *Drawline* are included, and are shown as heading modal subdialogues. A choice is required at all times (§) from the *{Tool}* group. Each subdialogue header is represented by a select-only meneme (↑), reflecting the fact that tools can be deselected only indirectly.
- The diagram includes a single object type *Graphobj*, and two of its subtypes *Textobj* and *Lineobj*. Selection propagation (and therefore subtype 'inheritance' within the type hierarchy) is inhibited, as denoted by the bars on the subtype links. As a result, the selection of *Graphobj* provides access only to the options at the supertype level. Similarly selection of either *Textobj* or *Lineobj* provides access only to options at the subtype level. The behaviour of a graphical object in MacDraw is thus dependent upon the context in which it is selected. Each of the subdialogues described below is concerned with object manipulation at either the supertype or subtype level.
- *Selectobj* is the default mode (*). Graphical objects can be selected from the current instance set of such objects, represented at this level by *Graphobj*. The basic <u>mutually exclusive</u> relationship between objects, in which only one object can be selected at any time, can be made <u>mutually compatible</u> by setting switch *S1* ON. Once an object has been selected at this level it can be *Moved*, *Cut*, *Copied*, or *Resized*. *Resize* is only available following selection of a single object, whereas the other options can apply to groups of selected objects.

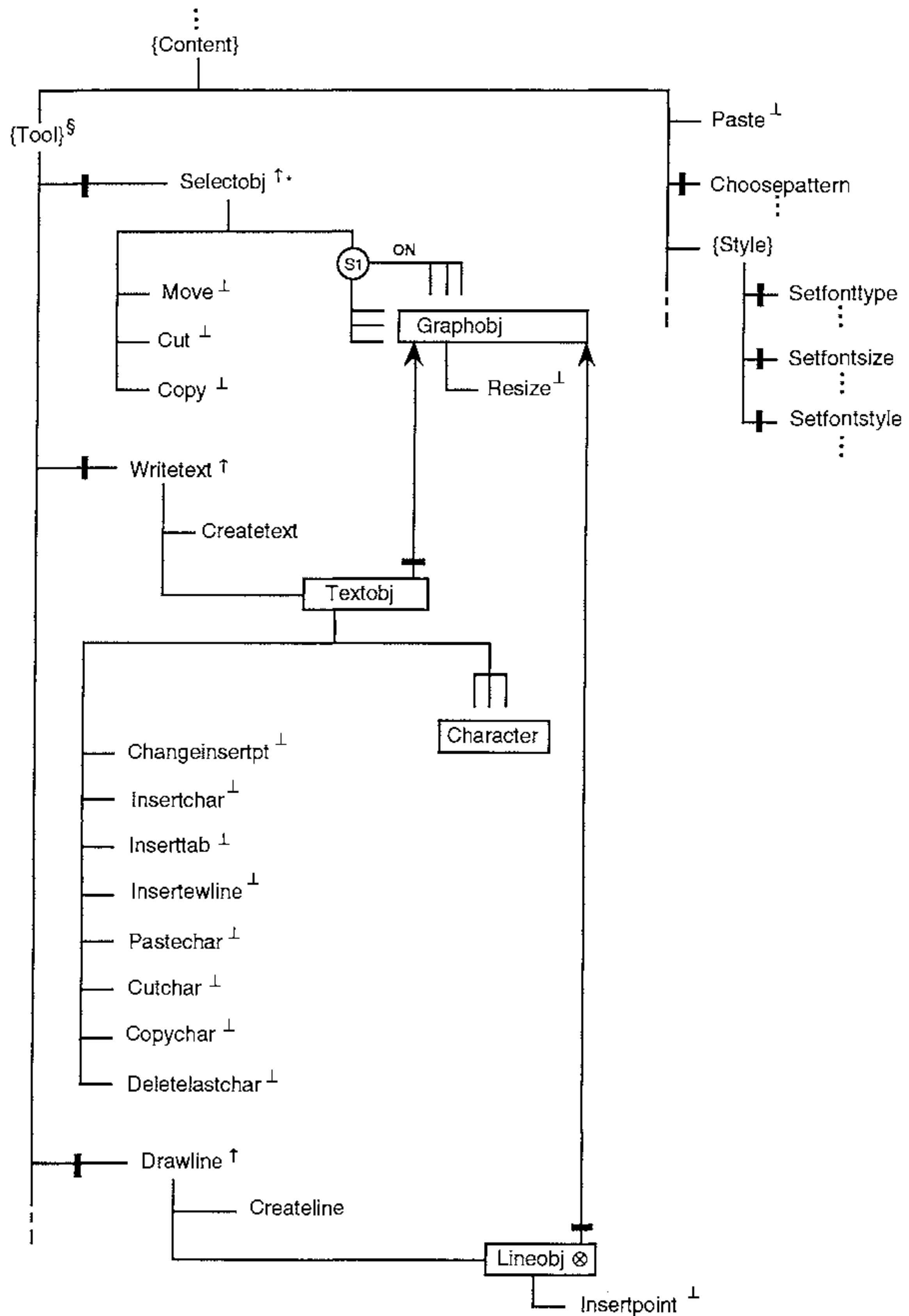


Figure 6.7: MacDraw: base layer tree diagram for level 2

- Text is handled in a manner similar to that in Microsoft Word. Selection of the *Writetext* subdialogue provides access to two options:
    1. *Createtext* provides for the creation of a new *Graphobj/Textobj* instance. A range of options is subsequently available for the manipulation at the *Textobj* level of the characters which comprise the newly created and selected object.
    2. An existing instance of *Graphobj/Textobj* can be selected at the *Textobj* level. The point of selection of the object determines the initial position of the insertion point in the text. Once selected the object can be extended either through *Pastechar*, or through repeated selection of *Insertchar* (which represents the insertion of any of the set of character values), reduced through selection of *Cutchar* or *Deletelastchar*, or modified through *Insertnewline* or *Inserttab*. The position of the insertion point can also be directly changed through *Changeinsertpt*.
- One further tool, *Drawline,* is included. This provides for the drawing of freehand lines, and is an example of a 'path' task. Path tasks involve repetition of 'position' tasks, as identified in the task analysis of Chapter 4 and shown in Table 4.4. Here, *Createline* creates an instance of *Graphobj/Lineobj*, and defines the start position of the line at the *Lineobj* level, which is then drawn through repeated selection of *Insertpoint*. Deselection of *Lineobj* completes the line.
- During any of the above subdialogues, an object instance can be created and selected at the *Graphobj* level through the *Paste* command.
- Subdialogues relating to fill in patterns and character style are shown but not further developed. These can either be applied to selected objects, or used to change the current settings in force. Other sets of options, also mutually compatible with drawing tool modes, have been omitted as they would add complexity to the design without revealing further aspects of the Lean Cuisine+ notation.

It should be noted that although both text and freehand line objects are produced via compound tasks (designated 'text 'and 'path' respectively in the task analysis of Chapter 4), an asymmetry exists in MacDraw in the manner in which the resulting composite text and line objects can be manipulated. Whereas the primitive characters of text objects can be individually selected and operated upon, the primitive points which define a freehand line exist only as part of the composite line object. This is reflected both in the way they are represented in Figure 6.7, and in the OSVs shown for these object subtypes in the next subsection.

Object state variables

OSVs relating to level 2 of the MacDraw dialogue, as far as it is developed in Figure 6.7, are shown in Table 6.2. They are grouped by object type and subtype. They are intended to be indicative at this early design stage. Other aspects of the dialogue

would lead to further object subtypes being defined, and subsequent refinement would require, for example, the exact form of an object's position and size to be determined.

Graphobj:	#gname	object name
	gobclass	object class = semantic
	gobtype	object type = graphical object
	gselstat	object selection status (selected, unselected)
	gposnTL	object position - top left
	gposnBR	object position - bottom right
	glayer	object layer
Textobj:	**Subtype of Graphobj**	
	#gname	object name
	tsubtype	object subtype = text object
	tselstat	object selection status (selected, unselected)
	tinsertpoint	current insertion point
	tsize	current size (characters)
	tfonttype	font type at current insertion point
	tfontsize	font size at current insertion point
	tfontstyle	font style at current insertion point
Lineobj:	**Subtype of Graphobj**	
	#gname	object name
	lsubtype	object subtype = freehand line
	lselstat	object selection status (selected, unselected)
	lpointposn*	point position (* repeated)
Character:	#cname	object name
	cobclass	object class = semantic
	cobtype	object type = character
	cselstat	object selection status (selected, unselected)
	cfonttype	character font type
	cfontsize	character font size
	cfontstyle	character font style

Table 6.2: OSVs for level 2 of MacDraw

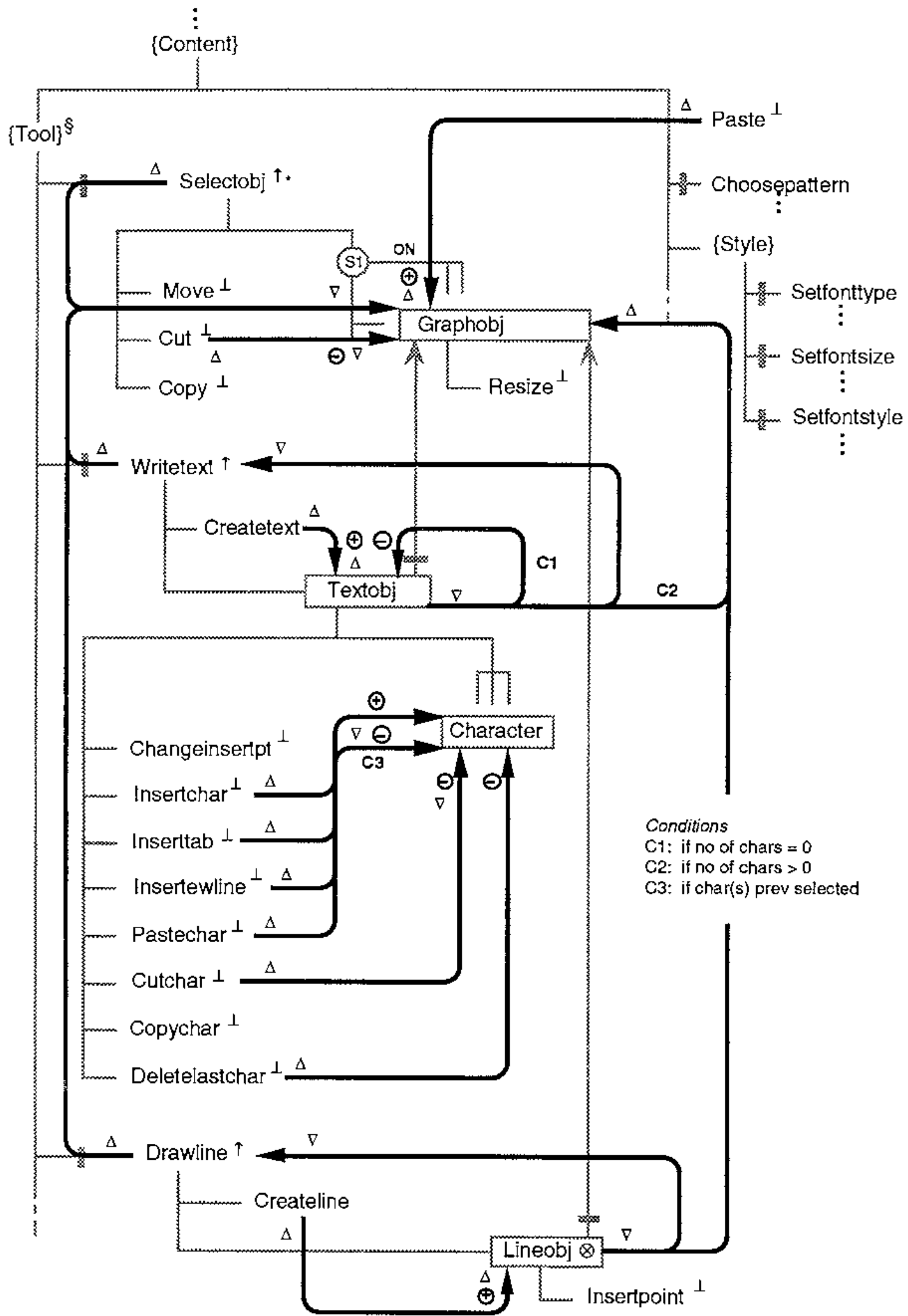


Figure 6.8: MacDraw: selection triggers for level 2

Selection trigger layer

Figure 6.8 shows selection triggers for level 2 of the MacDraw dialogue. In the following discussion, points of note relating to these triggers are grouped according to the tool subdialogues and their associated objects:

- *Selectobj subdialogue:*
  - The selection of *Paste* creates an object instance and selects it at the *Graphobj* level. (It also triggers system selection of the dialogue header *Selectobj*.)
  - The selection of *Selectobj* deselects any previously selected object instance.
  - The selection of *Cut* both deselects and deletes a selected object instance.

- *Writetext subdialogue:*
  - The selection of *Writetext* deselects any previously selected object instance.
  - The selection of *Createtext* creates a new *Graphobj/Textobj* instance, and selects it at the *Textobj* level. This in turn causes deselection (and therefore termination) of the mutually exclusive *Createtext* option. The newly created object instance remains available (selected) for further manipulation at the *Textobj* level.
  - *Character* instances can be created through the *Insertchar* trigger, and manipulated through triggers associated with *Insertchar*, *Inserttab*, *Insertnewline*, *Pastechar*, *Deletelastchar*, and *Cutchar*.
  - The deselection of the current object instance at the *Textobj* level, triggers the deselection of *Writetext*, which in turn causes the mutually exclusive default subdialogue *Selectobj* to become selected. Depending on the number of characters remaining in the object, one of two further triggers result:
    1. if the number of characters is zero (condition C1), the *Graphobj/Textobj* instance is deleted;
    2. otherwise (condition C2), the object instance is selected at the *Graphobj* level, and is available for further editing at that level.

- *Drawline subdialogue:*
  The selection of *Createline* creates a new *Graphobj/Lineobj* instance, and selects it at the *Lineobj* level. This in turn causes deselection (and therefore termination) of the mutually exclusive *Createline* option. Deselection of the object at the *Lineobj* level completes the line, causing *Drawline* to be deselected, *Selectobj* to become selected, and the newly created instance to be selected at the *Graphobj* level and made available for further editing at that level.

Option precondition layer

Figure 6.9 shows option preconditions for the MacDraw dialogue. The preconditions are listed on the diagram, and are self explanatory.

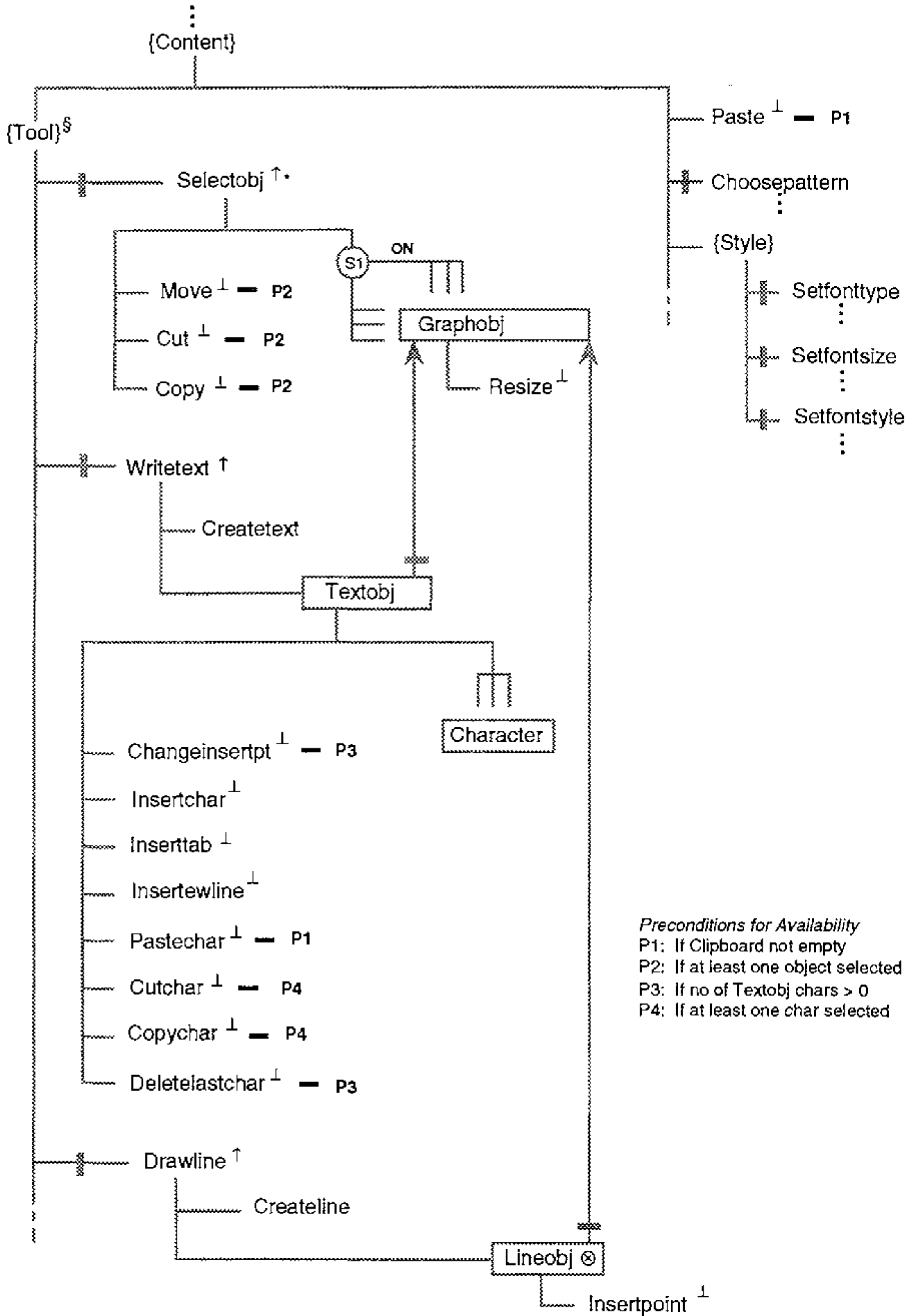


Figure 6.9: MacDraw: option preconditions for level 2

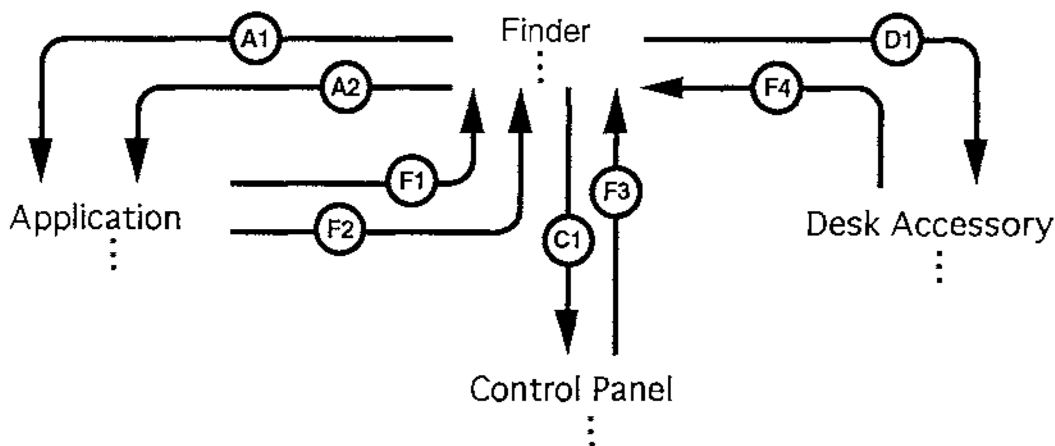
6.4 The completed Finder-application linkage

Section 6.1 introduced named external selection triggers between dialogues and specifically described two such triggers between the Finder and application dialogues. In Section 6.2 a further external trigger (F1) was introduced between the Microsoft Word and Finder dialogues, and shown in Figure 6.4.

This link triggers the creation of an instance of *ApplicDocIcon* in the Finder dialogue on the first occasion on which a *Save* command is issued within the Word dialogue. Although not shown, a similar link exists between the MacDraw and Finder dialogues. Figure 6.11 is an extension of Figure 6.1 which shows the full set of triggers between the Finder and application dialogues.

6.5 The dialogue interlink diagram (DID)

The descriptions of the Microsoft Word and MacDraw applications have introduced a requirement for named selection triggers *between* dialogues. These links have been detailed on the separate dialogue specifications. A top level dialogue interlink diagram (DID) showing the root nodes of the various dialogue trees and the named triggers between them provides a useful addition to the overall interface description. An example for part of the Macintosh interface is shown in Figure 6.10. The links would be defined in the system dictionary, and are briefly described below the diagram. In this diagram, *Application*, *Desk Accessory*, and *Control Panel* represent generic types of dialogues.



A1: Open application program
A2: Open application document
C1: Open control panel application
D1: Open desk accessory
F1: Save application document
F2: Quit application program
F3: Quit control panel application
F4: Quit desk accessory

Figure 6.10: The DID for part of the Macintosh interface

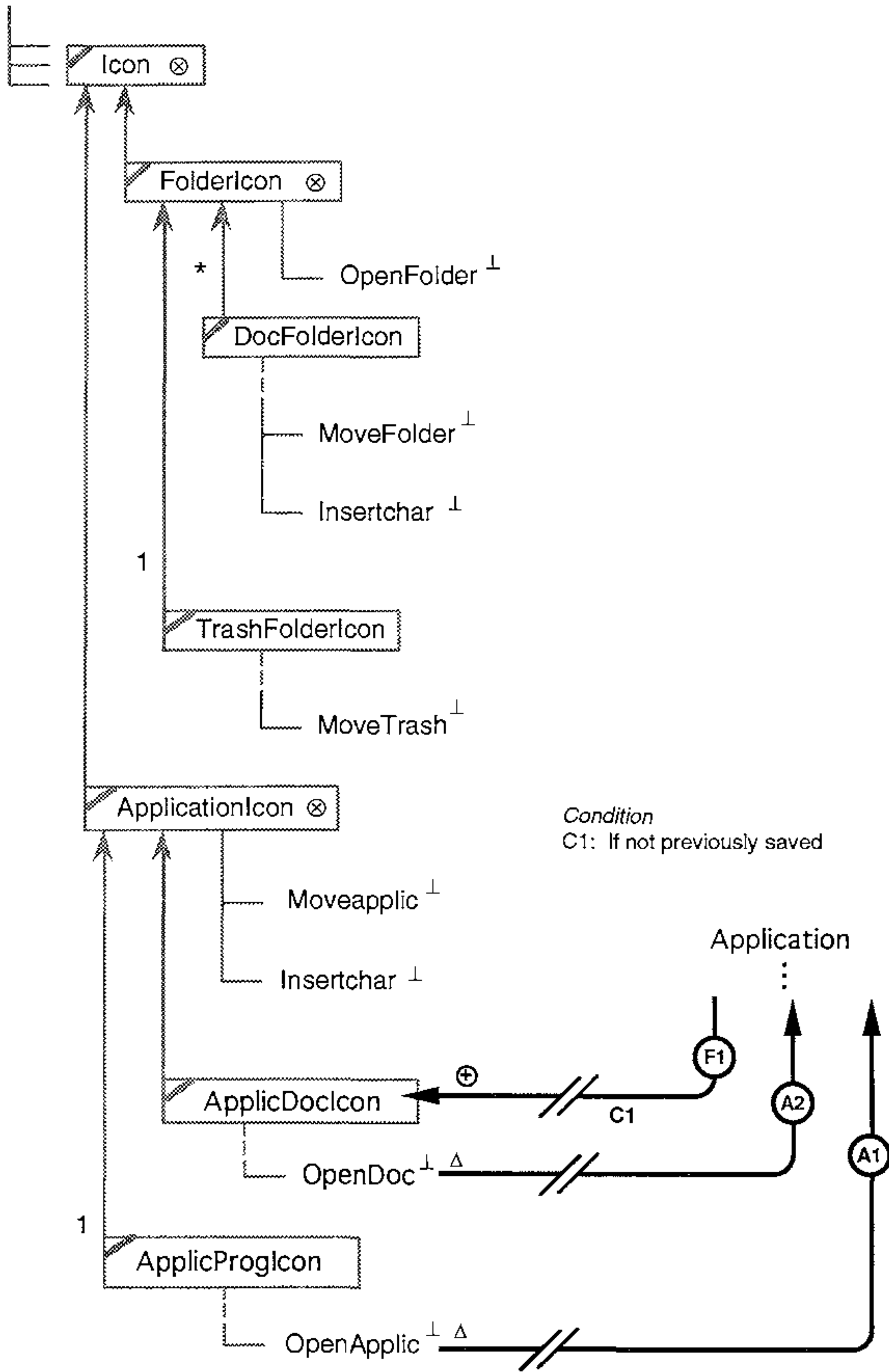


Figure 6.11: Finder icon object hierarchy showing all application links

6.6 Review

Lean Cuisine+ dialogue descriptions are based primarily on the grouping of selectable options around objects. Object types and object type hierarchies are supported. Through the specification of two application dialogues in this chapter, the object range has been extended and inter-dialogue linkage has been more fully considered. The notation has shown itself to be capable of encompassing a range of direct manipulation interaction objects, including icons, windows, menus, documents, text characters, and graphical objects.

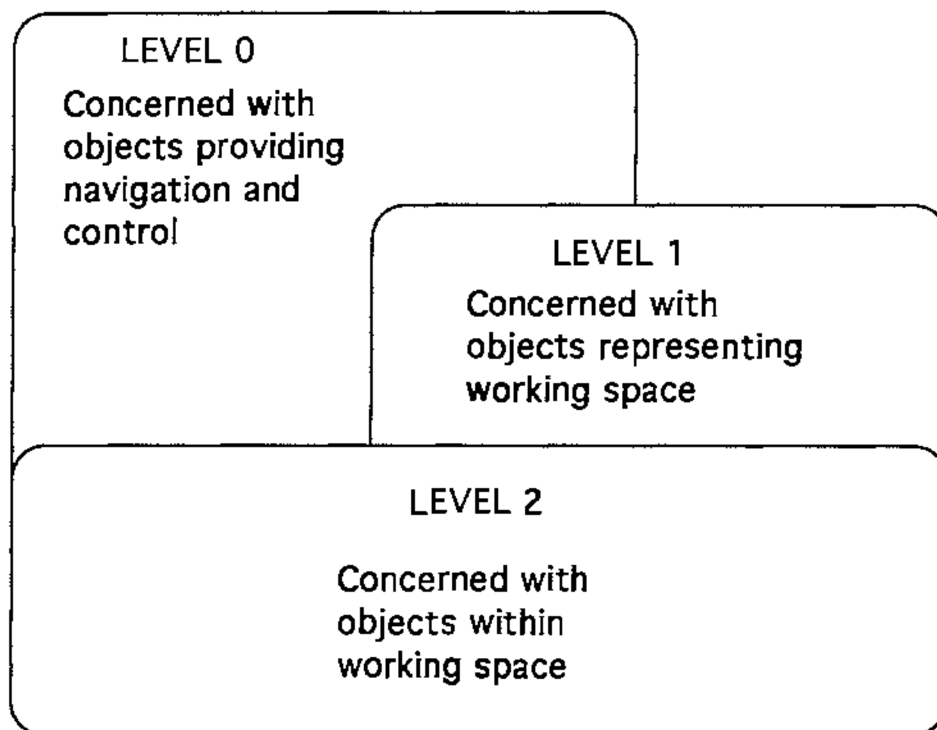


Figure 6.12: Dialogue levels in Lean Cuisine+

In describing the Microsoft Word and MacDraw dialogues, two levels of abstraction based on objects have been introduced into the base layer tree diagram:

Level 1: which is concerned with objects such as icons, windows, and documents which represent working space on the desktop; and

Level 2: which is concerned with objects *within* working space, such as text characters and graphical objects.

In Section 5.5, menus were defined as an orthogonal grouping of selectable interface options providing for navigation and control. A given set of options can be arranged in different ways in menu hierarchies without impacting the underlying

behaviour of the interface. This grouping of options into menus can be viewed as a further level of abstraction in the dialogue description which sits above both levels 1 and 2, and leads to:

Level 0: which is concerned with the grouping around menu objects of selectable options which exist in levels 1 and 2, in connection with their subsequent presentation to the user. Additional options in the form of selectable state variable values may also be included.

The relationship between these levels is shown diagrammatically in Figure 6.12. Levels 1 and 2 describe the underlying behaviour of the interface in terms of working space and objects within working space, while level 0 describes the behaviour of the menu system, which offers objects and options to the user for selection.

Chapter 7

Lean Cuisine+ in High Level Interface Design

Section 7.1 identifies possible roles for Lean Cuisine+ in interface design and establishes a focus for the chapter. In Section 7.2 the purpose of task decomposition in high level interface design is examined, and Section 7.3 adds an orthogonal task layer to the Lean Cuisine+ notation which captures any temporal relationships between primitive task actions. Section 7.4 describes a five stage methodology for constructing Lean Cuisine+ specifications which commences with a task decomposition. This methodology is then applied to the high level design of two example interfaces in Sections 7.5 and 7.6, and is briefly reviewed in Section 7.7.

7.1 The role of Lean Cuisine+ in interface design

In Chapters 5 and 6, the Lean Cuisine+ notation has been developed through the specification of part of an existing direct manipulation interface - that of the Apple Macintosh. The approach so far has thus been analytical, and based on so-called 'reverse engineering'. A primary objective of the research, as set out in Section 1.2, is that the notation should be applicable in the high level design phase of interface development. In pursuit of this goal, this chapter sets out to develop a methodology for constructing Lean Cuisine+ specifications for new interfaces. If supported by a software system, Lean Cuisine+ could become an executable specification providing for simulation of the behaviour of the interface under design. In Chapter 8 a software environment to support the construction, browsing, and execution of Lean Cuisine+ specifications is defined, and a prototype of part of this environment is described.

Two other possible roles can be identified for Lean Cuisine+:

1. Use as a prototyping and implementation tool. This would require the mapping of Lean Cuisine+ specifications into a dialogue specification language, and could provide for the linking of the specification and the prototype interface in on-screen tandem execution.
2. Use in documenting an interface being constructed in some other way, for example via interactive interface generators such as HyperCard or Prototyper on the Macintosh.

The first of these is examined in Chapter 8. The second is listed under further work in Chapter 9.

7.2 Task analysis

Task analysis, which includes the decomposition of the tasks within a system, was identified in Section 3.1 as a technique for use at the requirements stage of interface design. Both human and computer tasks are involved. A number of interface design methodologies include some form of task analysis at this stage, e.g. CLG (Moran, 1981), GOMS (Card *et al*, 1983), TKS (Johnson *et al*, 1988), and Adept (Markopoulos *et al*, 1992), which uses a graphical notation based on AND/OR graphs. Techniques have also been borrowed from the requirements phase of systems analysis. The approaches of DeMarco (1978) and Gane & Sarson (1979) for example both involve stepwise refinement, which is the essence of task decomposition. Successive refinement provides an increasingly detailed view of the interface under design, and when the units are sufficiently small they can be described by actions, which are the primitive building blocks of tasks. A given task is composed of a group of actions which achieve a goal.

Task analysis in relation to a complete interface can be difficult and time consuming. Sutcliffe (1988) observes that methods of interface task analysis have not been well defined. According to Bass & Coutaz (1991), decisions must be made in relation to the following:

- *Task ordering:* the amount of freedom the user has to switch between tasks.
- *Task anticipation:* the amount of information to be provided about the next tasks allowed, once a particular task has been specified.
- *Explanations:* the amount of domain-dependent information required when errors occur.
- *Assistance:* the amount of assistance to be provided with error recovery.

Bass & Coutaz (p.19) see the result of this analysis as "a set of specific tasks and concepts that the operator will perform and manipulate with the aid of the computer system". In a direct manipulation interface the concepts are represented by objects. The task graph which results from task decomposition can be validated to ensure that it is semantically correct (meets user requirements), and revised if necessary. It can be examined for alternative task decompositions, common tasks, and non-determinancy (when a single task has multiple subtasks which can occur in any order). At the wider level, Johnson (1992) considers that task analysis can be used by designers to generate ideas about the interface, and that tools and methods that focus on task analysis are needed. In the methodology in Section 7.4, only task decomposition is addressed.

7.3 The Lean Cuisine+ task layer

Lean Cuisine+ provides for the specification of direct manipulation interfaces in terms of the constraints and dependencies which exist between selectable dialogue primitives. The notation developed in Chapters 5 and 6 is a layered one. A further layer, the task layer, can be added. It provides for the description of higher level user tasks in terms of primitive actions (and associated events), capturing any temporal relationships between them, and providing a link between object-oriented and task-oriented views of a dialogue. Sequences or combinations of task actions relating to each higher level task can be overlayed on the base layer tree diagram. Each of these overlays forms part of the task layer, which is orthogonal to the constraint-based behavioural model of the interface provided by the tree diagram and constraint overlays. In Section 7.4, a task decomposition is used as a starting point for constructing Lean Cuisine+ specifications for new interfaces, and can subsequently be used as a basis for generating the task overlays (automatically if supported by a software support system).

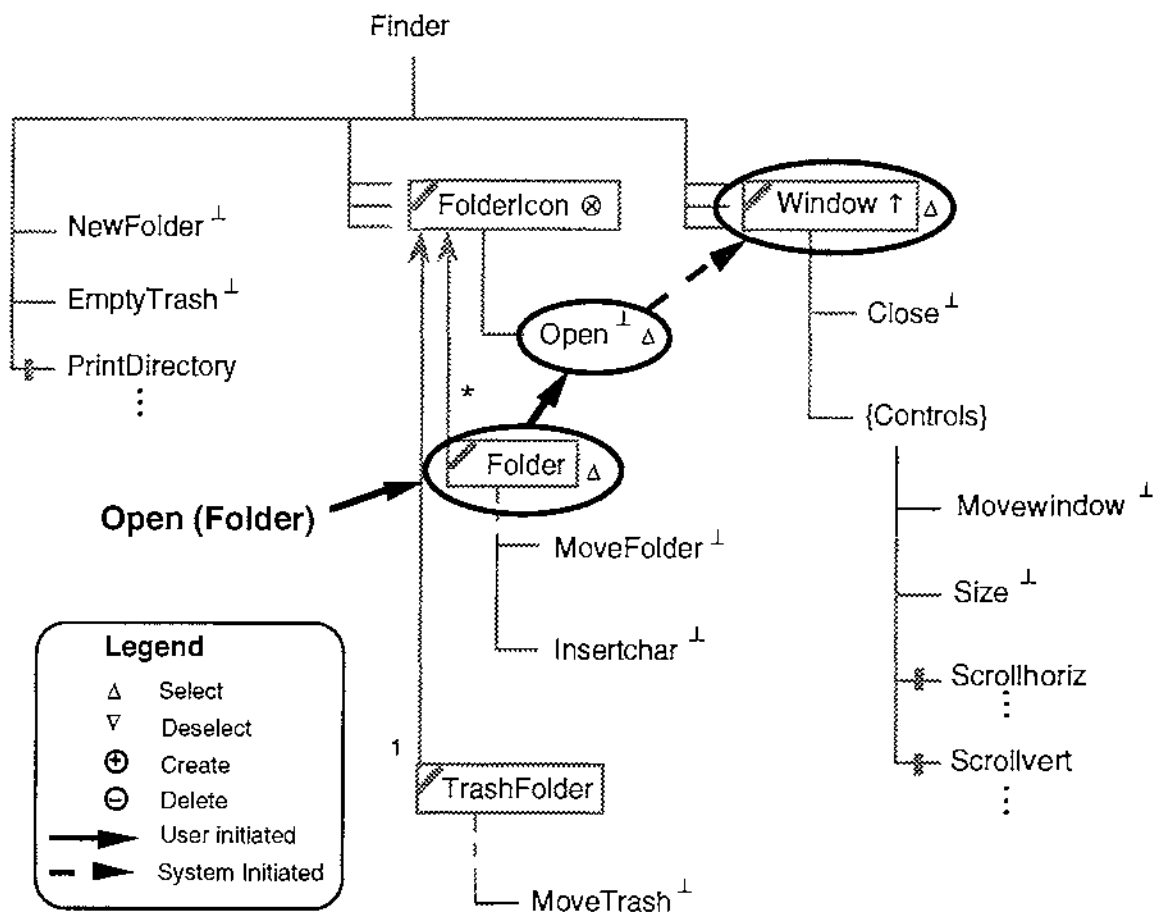


Figure 7.1: Task action sequence for *Open (Folder)* over greyed base layer diagram

In Figure 7.1, the task action sequence for opening a folder is overlaid on the base layer Lean Cuisine+ diagram, shown greyed. The annotation is defined by the legend. The notation distinguishes user initiated and system initiated events by means of the arcs connecting selections. A plain arc between two successive selections denotes that the new selection is the result of a user action, whereas a dashed arc indicates that the new selection is a system response triggered by the last user action. In this example, the user selects a *Folder* instance (which also selects the parent *FolderIcon* through propagation up the type hierarchy) and then selects *Open*, and the system responds by making available (opening) and selecting (making active) the corresponding *Window* instance. The link which triggers the system response was included in the selection trigger layer of Figure 5.3.

7.4 A methodology for constructing Lean Cuisine+ specifications

A five stage approach for the construction of Lean Cuisine+ specifications for new interfaces is presented. The methodology is based on a logical progression from task decomposition through object identification to construction of Lean Cuisine+ layers. It is shown diagrammatically in Figure 7.2 and described below. Some iteration could be expected but is omitted from the diagram. Details of the design would be entered into a system dictionary as it was constructed.

STAGE 1: *Decomposition:* Task decomposition is undertaken and a task hierarchy constructed. Higher level user tasks associated with the interface under design are identified and decomposed through successive levels into actions (primitive subtasks) involving selection of interface objects or operations. In some cases the *sequence* of selections is significant and must be captured. In other instances, it is only the *combination* of selections which matters, i.e. there is non-determinancy in the task graph. Side effects relating to selections, and preconditions relating to option availability are noted. The groupings of task actions are evaluated, and validated against user requirements, and changes fed back to the higher level if appropriate, in an effort to improve the decomposition.

STAGE 2: *Translation:* Lean Cuisine+ subdialogue trees are constructed. Below the root dialogue these are headed primarily by objects. This stage involves identification of the basic logical inter-relationships between options (mutually exclusive or mutually compatible, default etc) and the grouping of options. Modal subdialogues are identified. The structural correctness of the subtrees is verified.

STAGE 3: *Synthesis:* The Lean Cuisine+ subtrees are combined into a single dialogue tree diagram capturing levels 1 and 2 of the dialogue as defined in Section 6.6. This involves consideration of the overall object hierarchy, specific object subtype hierarchies, and inter-object relationships, and the introduction where necessary of further generic objects representing homogeneous object instance sets. Object state variables are identified. Further meneme modifiers may also be added. The structural correctness of the dialogue model, including any type hierarchies, is verified.

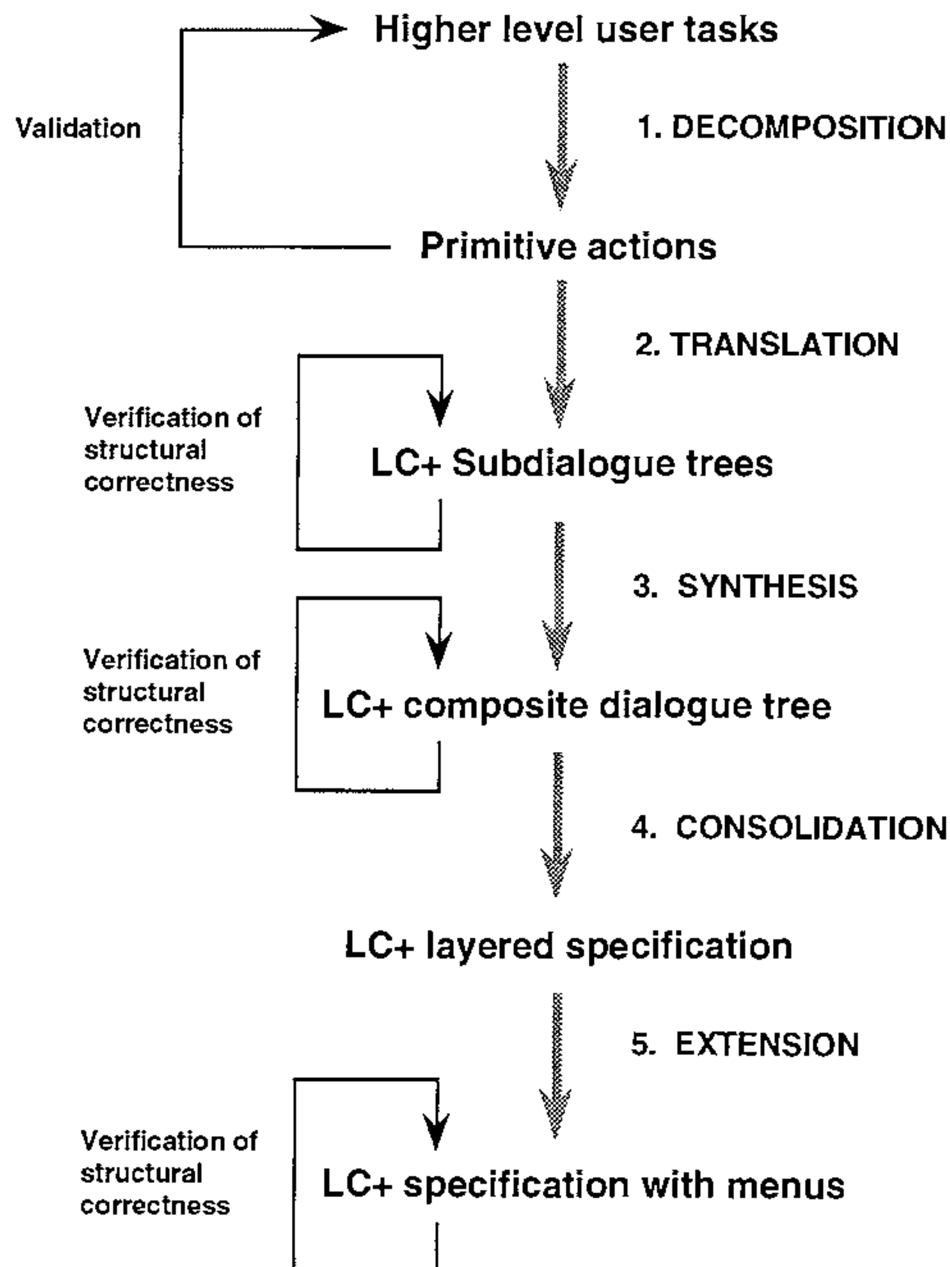


Figure 7.2: Methodology for constructing a Lean Cuisine+ Specification

STAGE 4: *Consolidation:* Constraints and conditions identified at Step 1 are captured in the specification through the addition of Lean Cuisine+ layers involving selection triggers, option preconditions, and existence dependencies. Task action groupings identified at Step 1 are captured in task overlays.

STAGE 5: *Extension:* Lean Cuisine+ representations of menus, involving both the regrouping of menemes defined at stages 2 and 3, and the introduction of additional menemes representing state variable values, are constructed and added to the specification. Each menu subdialogue is headed by a syntactic menu object, and constraints already defined for menemes at stage 3 apply. Where menemes belong to more than one object, e.g. an icon and a menu, they are replicated in the visual model, but appear only once in the system dictionary, where information regarding the parent objects is recorded. Menus, which relate to level 0 of the dialogue as defined in Section 6.6, provide an orthogonal grouping of the selectable options present in an interface to facilitate navigation and control.

7.5 Case study I: the Finder document folder system

In this section the above methodology is applied to the specification of the Macintosh Finder document folder system of Chapter 5. Although this is somewhat artificial, as the interface (and a Lean Cuisine+ specification of it) already exists, it demonstrates the applicability of the methodology, and by using a known example keeps the focus on the methodology rather than the problem. The method is applied to a new interface in case study II in Section 7.6. The following simplifications to the Finder system, introduced in Chapter 5, also apply here:

- folder icon instances are considered to exhibit mutually exclusive behaviour, i.e., only one icon can be selected at any time;
- folder (and therefore window) names are restricted to one character;
- application programs and documents are excluded; and
- the insertion and ejection of diskettes have been omitted.

The functional requirements of this system, which supports the complete document life cycle, can be defined as follows:

- Document folders are represented by icons, and are considered to be available for selection if they are either on the desk top or their parent folder is open. The document folder life-cycle is as follows: a document folder can be created (in a selected state) within the active (currently selected) window; a selected folder can be renamed, and moved (which may place it inside another folder, when it becomes

deselected); the folder can be opened, when its contents become available via an associated window; when active, a window can be moved and resized, and the directory area within it scrolled; the active window (and associated folder) can be closed; the folder can be deleted by placing it into a 'Trash' folder (which also causes the associated window to be closed), and subsequently emptying this folder. The Trash folder can be moved, but cannot be renamed or nested, nor can folders be created within it. The current hierarchical directory of folders can be printed.

- The behaviour of icons and windows is related. Each icon, which represents one of a mutually exclusive set of document folders, has a window associated with it *at all times* whether or not it is in an open state, and vice versa. The available (open) window set may be a subset of the available icon set. The creation of a new folder also instantiates an associated window which is initially closed and unavailable for selection. Opening a folder makes available and selects (makes active) the associated window. The icon remains available for selection or deselection following the opening of the window, when there are two representations of the same object on the display. Closing a window deselects it, and causes the associated icon to be selected (presentationally, a window 'closes into' its folder). If an icon is deleted the associated window goes with it. Selecting a window (to make it active) causes the corresponding icon to be selected; selecting an icon within an inactive parent window causes the parent to be selected (made the active window); and selecting an icon, either directly or indirectly, causes any previously selected icon to be deselected by virtue of the mutual exclusivity of the set.
- Open windows are managed in the form of a stack exhibiting cyclic behaviour, with the currently active window at the top. An open window can be in an active (selected) or inactive (available) state. In the active state, if the directory area is bigger than the window area then scrolling facilities are available in horizontal and/or vertical planes. If another available window is selected during the subsequent dialogue then the first window becomes deselected by virtue of the mutual exclusivity of the set, but remains available. A window cannot be directly deselected. If the physical state of the window is altered during the dialogue and the window closed again, its state is saved. Closing a document window causes the next window on the stack (if any) to become active.
- Both intra-object and inter-object relationships require to be described, the former resulting from the fact that two representations of the same object (an icon and a window) can be in existence at any time, and the latter from the hierarchical parent-child nature of the relationship between icon and window objects. It is necessary to represent both available and unavailable instantiations of icons and windows.

The methodology described in Section 7.4 is now applied to this interface:

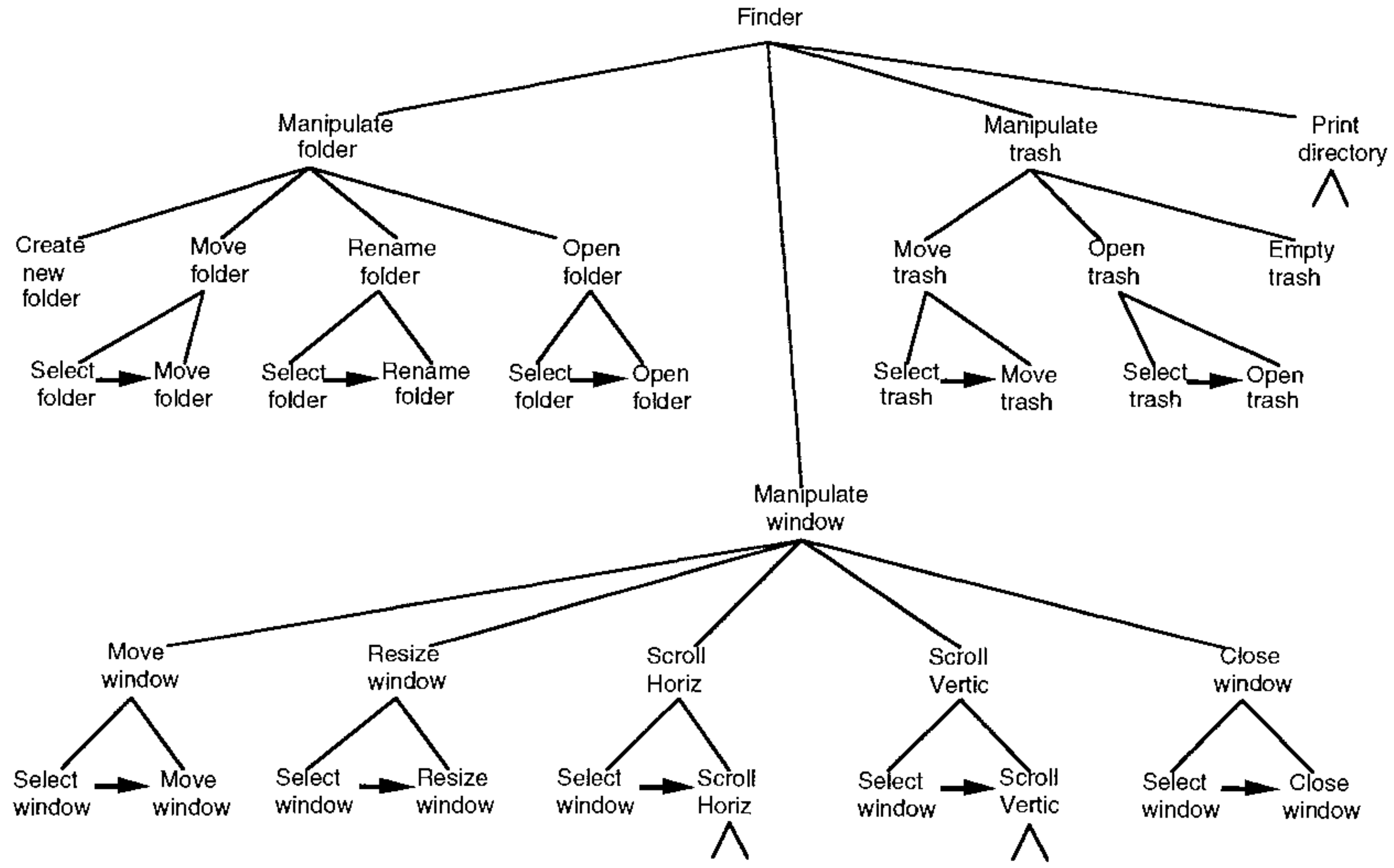


Figure 7.3: STAGE 1: Finder task decomposition

	User Task	Task Actions		Side effects (S) or Preconditions (D)
		Action	Object	
1.	Create new folder	Create folder	(folder) (window)	S1: Select folder & Create window D1: window selected & not Trash window
2.	Rename folder	Select folder	folder (window)	S2: Select parent window if folder in window D2: On desktop or parent folder open
		Enter character	folder	
3.	Move folder	Select folder	folder	see 2 above
		Move folder	folder (window)	S3: Select new parent & Deselect folder if dictated by move & Close assoc. window if moved to Trash
4.	Open folder/Trash	Select folder/Trash	folder/Trash	see 2 above
		Open folder/Trash	folder/Trash	S4: Select assoc. window
5.	Move window	Select window	window (folder)	S5: Select matching folder D5: matching folder open
		Move window	window	
6.	Resize window	Select window	window	see 5 above
		Size window **or**	window	
7.	Scroll horiz.	Select window	window	see 5 above
		Scroll horiz.	window	D7: dir width > window
8.	Scroll vert.	Select window	window	see 5 above
		Scroll vert.	window	D8: dir depth > window
9.	Close window	Select window	window	see 5 above
		Close window	window (folder)	S9: Deselect window & Select matching folder Select next window on stack (if any)
10.	Move Trash	Select Trash	Trash	
		Move Trash	Trash	
11.	Empty Trash	Empty Trash	(folder) (window)	S11: Delete folder Delete window D11: Trash not empty
12.	Print directory	Print directory	(folder)	

Table 7.1: STAGE 1: Finder tasks

STAGE 1: *Decomposition:* Figure 7.3 shows a task decomposition for the Finder document folder system. The higher and intermediate level user tasks have been identified by considering the document folder life cycle. Where more than one subtask makes up a task at the next level, any sequencing is indicated through node connection at the subtask level. The task actions *Select folder* and *Select window* occur repeatedly. Window scrolling actions have been omitted.

Table 7.1 expands on the diagram by identifying object(s) associated with each task action (objects indirectly impacted are shown bracketed), plus any associated side effects and/or option preconditions, which for reference are numbered according to the task to which they relate. Three object types have been identified within the dialogue - *folder*, *Trash (folder),* and *window*. Most of the primitive task actions are directly associated with one of these object types, through being involved in either the selection or direct manipulation of an existing object instance, the exceptions being *Create folder*, *Empty Trash*, and *Print directory*.

STAGE 2: *Translation:* Figure 7.4 shows the initial Lean Cuisine+ subdialogue trees. With the exceptions of *NewFolder, EmptyTrash*, and *PrintDirectory,* which are attached to the *Finder* root dialogue, options are attached to the objects identified in Table 7.1. All the operations are represented as modal subdialogues.

STAGE 3: *Synthesis:* A first pass object hierarchy, comprising two mutually compatible generic object types - *Folder* and *Window* - is shown in Figure 7.5. *Folder* is mutually exclusive with respect to *TrashFolder*, and this group is mutually compatible with respect to *Window*. The object hierarchy is refined in Figure 7.6 through the introduction of a type hierarchy for folders. *Folder* and *TrashFolder* have identical *Open* options but different *Move* options (it is not possible to move the Trash folder into another folder). In addition a *Folder* instance can be renamed. In the revised object hierarchy of Figure 7.6 a supertype, *FolderIcon*, has been introduced. The resulting subdialogue trees are shown in Figure 7.7, and the composite tree in Figure 7.8 now emerges.

Further meneme modifiers have been added in Figure 7.8: the *Window* meneme has been flagged select-only (↑) to reflect the fact that windows cannot be directly deselected by the user; the passive modifier (⊗) has been added to the *FolderIcon* supertype to indicate that it is user selectable only through its subtypes; and a default choice (*) is shown against the *Folder* set to indicate that at the commencement of the dialogue one of the set is selected. The composite tree now matches with the base layer Lean Cuisine+ tree developed in Section 5.2 and shown in Figure 5.1.

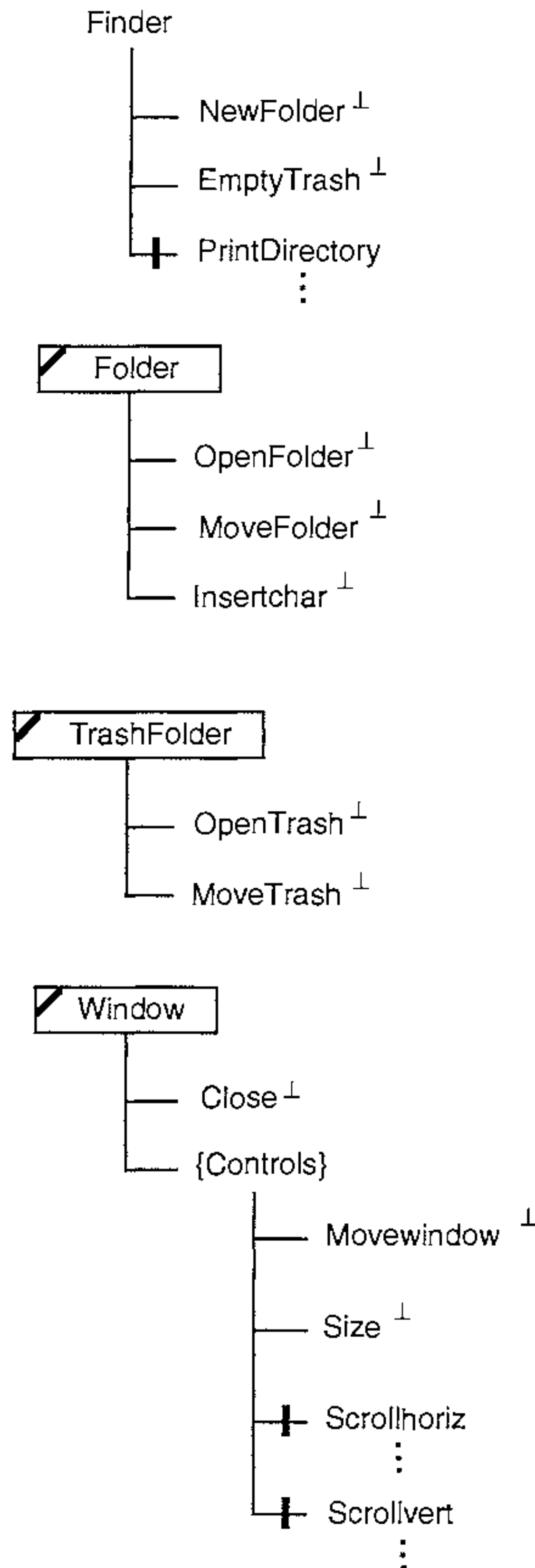


Figure 7.4: STAGE 2: Initial Finder subdialogue trees

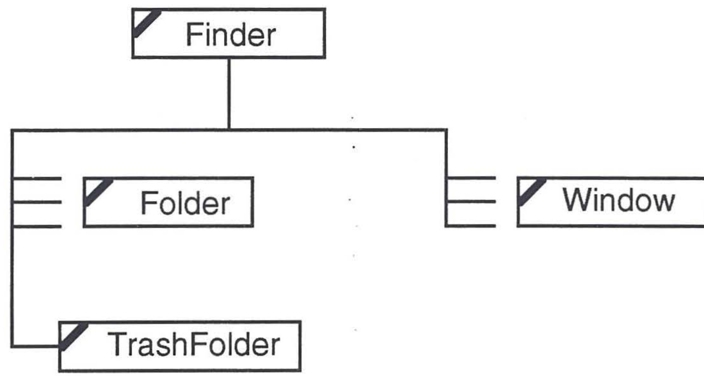


Figure 7.5: STAGE 3: Initial Finder object hierarchy

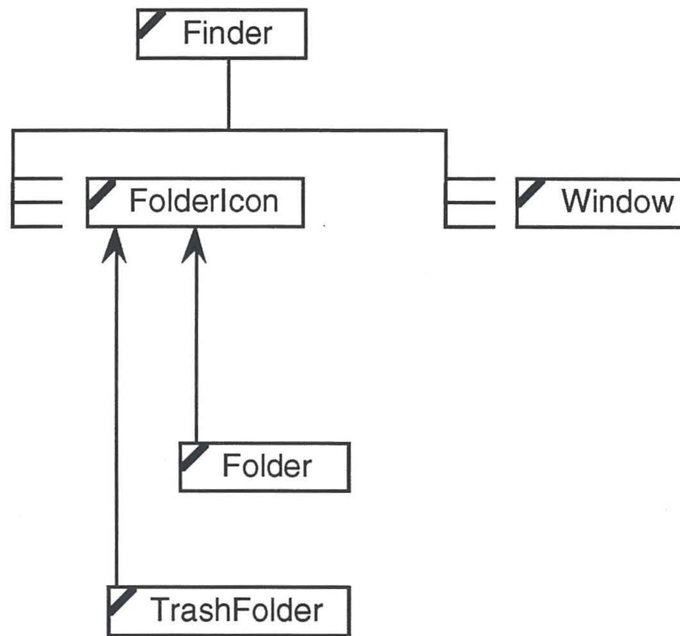


Figure 7.6: STAGE 3: Refined Finder object hierarchy

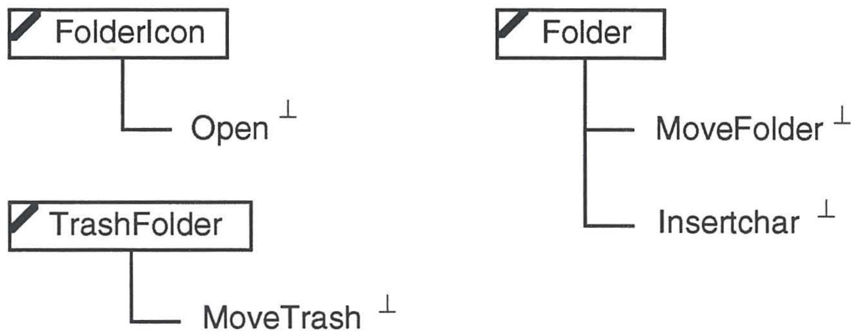


Figure 7.7: STAGE 3: Refined Finder subdialogue trees

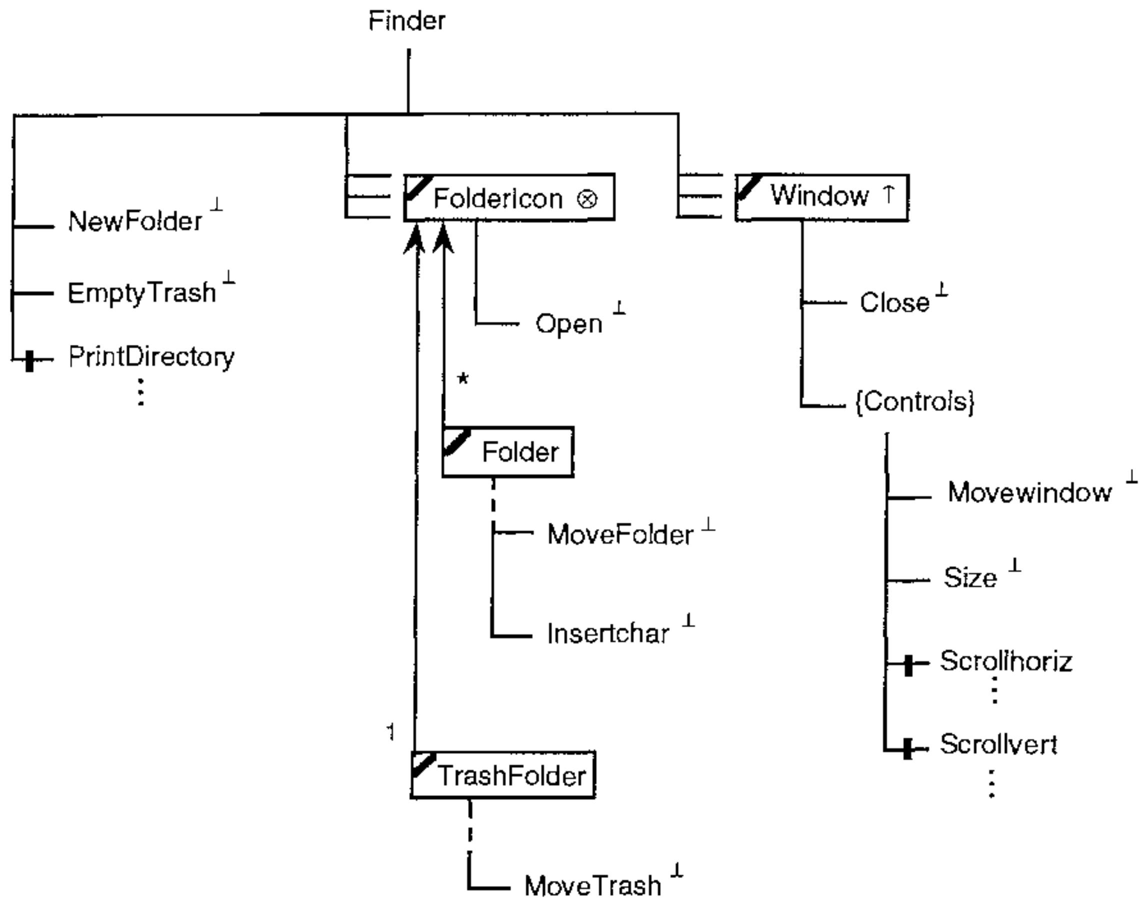
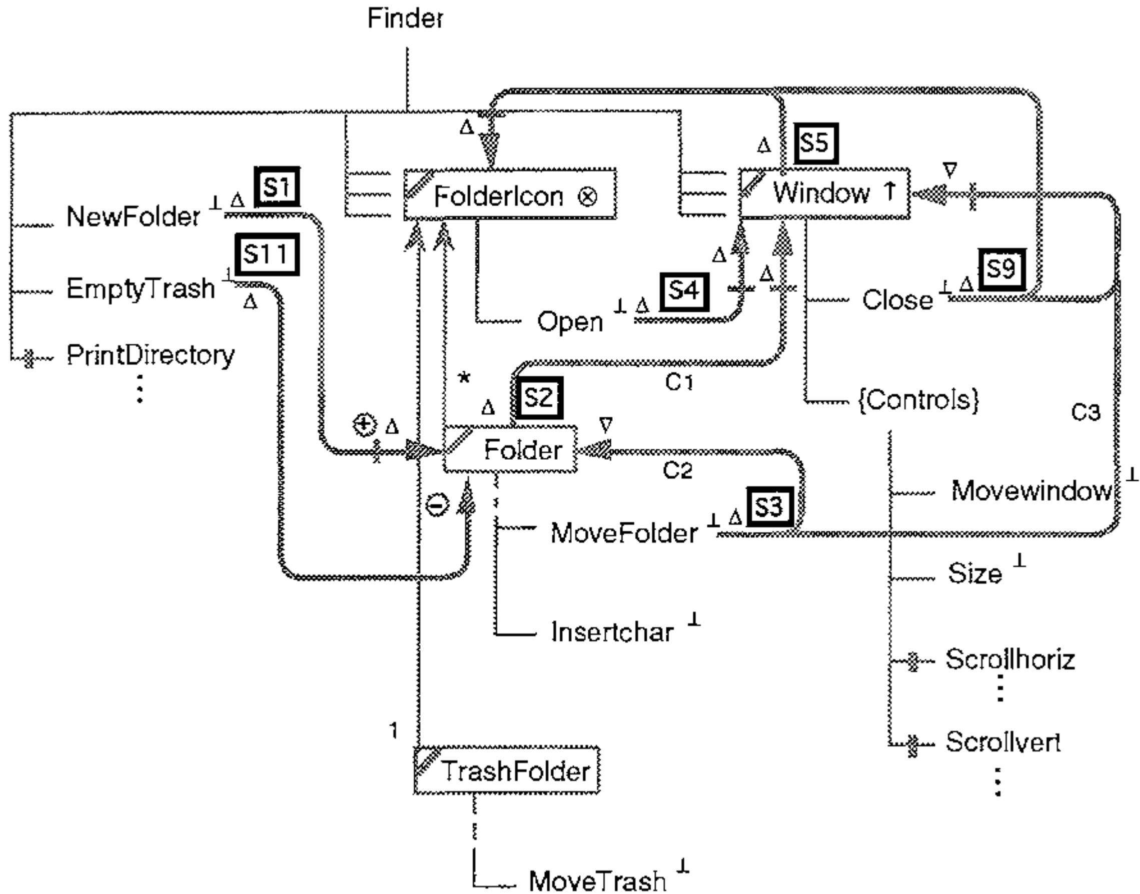


Figure 7.8: STAGE 3: Composite Finder dialogue tree

An initial set of state variables is now defined for each object and added to the system dictionary. These are shown in Table 7.2. Each object is given a unique identifier, is classified as being either semantic (from the task domain) or syntactic, and is allocated an object type classification. Object selection and availability indicators are included. Each folder carries the name of its parent, and both folders and windows carry their current position. The remaining window variables are connected with window and directory sizes.

STAGE 4: *Consolidation:* The selection side effects and option preconditions identified at Step 1, and listed in Table 7.1, are now added to the dialogue description in the form of overlays. Figure 7.9 shows selection triggers, with the source of each trigger referenced to a side effect listed in Table 7.1. Triggers are designated 'constrained' where side effects such as cycles may result. For this first case study, the whole diagram has been greyed in order to highlight these references, which are shown boxed. Option preconditions are handled in a similar manner in Figure 7.10, with the sources of preconditions also referenced to Table 7.1.



Conditions
C1: if selected icon is within its parent window
C2: if placed into new closed parent folder
C3: if any new parent folder is the 'Trash' folder

Figure 7.9: STAGE 4: Finder selection triggers showing Table 7.1 references

FolderIcon:	#iname	object name
	iobclass	object class = syntactic
	iobtype	object type = folder icon
	iselstat	object selection status (selected, unselected)
	ifostat	folder open status (open, closed)
	iposition	current icon position
Folder:	**Subtype of FolderIcon**	
	#iname	object name
	fobsubtype	object subtype = document folder icon
	favstat	object availability status (available, not available)
	fparent	parent folder name

Table 7.2a: Initial set of Finder OSVs

TrashFolder:	**Subtype of FolderIcon**	
	#iname	object name
	tobsubtype	object subtype = Trash folder icon
Window:	#wname	object name
	wobclass	object class = syntactic
	wobtype	object type = window
	wavstat	object availability status (open, closed)
	wselstat	object selection status (active, inactive)
	wposition	current window position
	wwidth	current window width
	wdepth	current window depth
	wdwidth	current overall directory width
	wddepth	current overall directory depth

Table 7.2b: Initial set of Finder OSVs - contd.

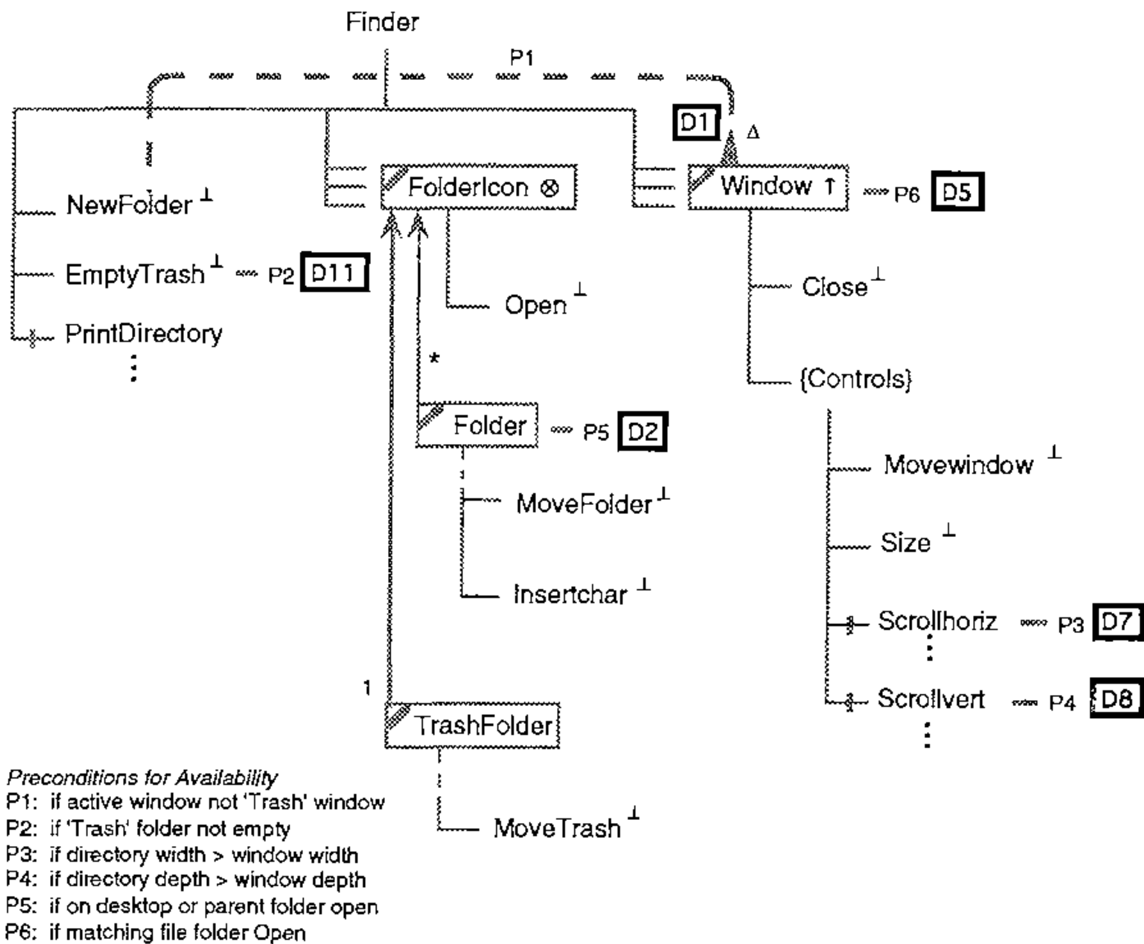


Figure 7.10: STAGE 4: Finder option preconditions showing Table 7.1 references

The existence dependency between the *FolderIcon* and *Window* objects is captured in the further layer shown in Figure 7.11.

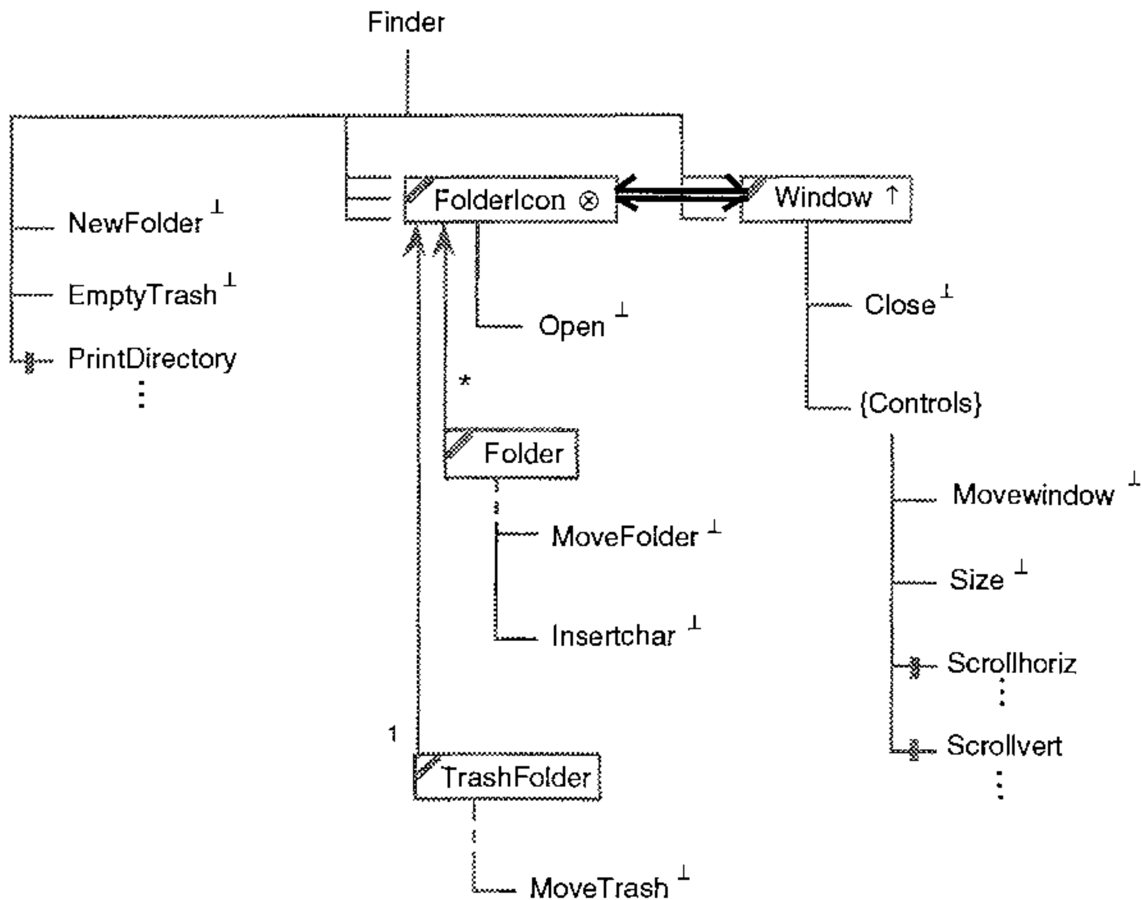


Figure 7.11: STAGE 4: Finder existence dependency

The task layer provides for the display of sequences or groups of actions relating to each of the higher level user tasks shown in Table 7.1, in the form of overlays to the dialogue tree. The example which appeared in Figure 7.1 for *Open (Folder)* is equivalent to Task 4 in Table 7.1. It comprises two user actions, *Select folder* and *Open folder*, and a trigger leading to the system selection (indicated by the dashed link) of the associated window. *Open* is an example of a higher level user task consisting of a <u>sequence</u> of primitive task actions. A further example for the task *Empty Trash* (Task 11 in Table 7.1) appears in Figure 7.12. This task comprises a single user action which triggers a system action. Task overlays are generated from information captured in the task decomposition of stage 1. Generation could be automatic in a software support environment.

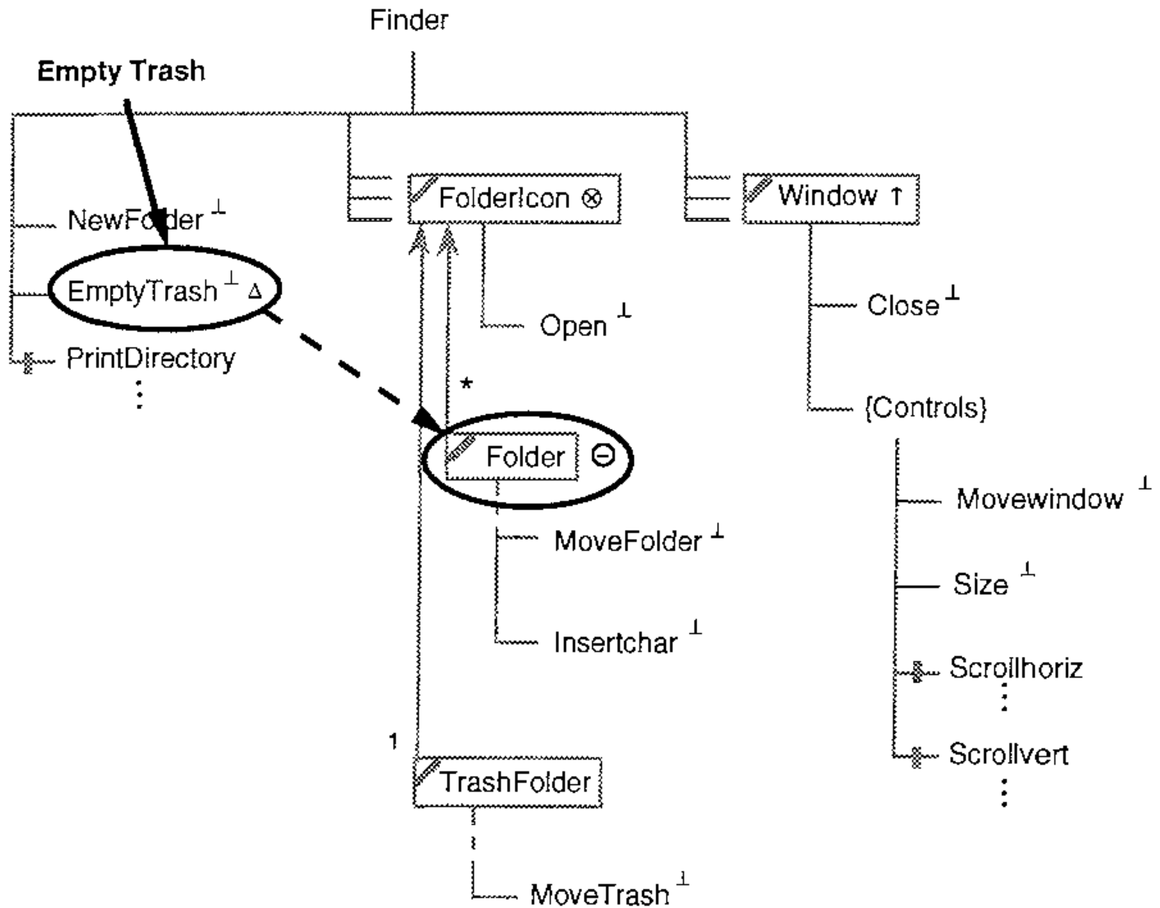


Figure 7.12: STAGE 4: Finder task action sequence for *Empty Trash*

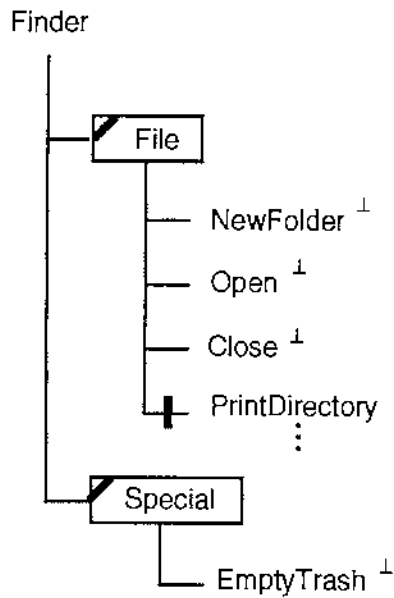


Figure 7.13: STAGE 5: Finder menus

STAGE 5: *Extension:* A menu system to provide for navigation and control of the Finder interface is now added, and described in a separate Lean Cuisine+ tree diagram. Menus are represented as syntactic objects, and individual options are represented by terminal menemes. A Lean Cuisine+ tree diagram showing those aspects of the Finder *File* and *Special* menus which appear in the simplified document folder system of the case study is shown in Figure 7.13. Additional syntactic objects in the form of *File* and *Special* menus have been introduced. Option preconditions would also apply (see Figure 5.11).

7.6 Case study II: an electronic mail browser

In this section the methodology of Section 7.4 is applied to the specification of an electronic mail browser (exhibiting functionality typical of current mail applications). Mail messages sent and received are held in mailboxes, any number of which may exist at any time. Messages comprise a header, which identifies them, and a body, which contains the contents of the message. Specifically the browser must support the functions listed below in a window environment. It should be noted that facilities for the creation of messages and the manipulation of message content are not included:

- The creation and deletion of mailboxes. A mailbox must be empty to be deletable.
- The opening and closing of mailbox windows, and the browsing of message headers within the active window. Mailbox windows can be moved.
- The transfer of messages between mailboxes, and the deletion of messages, both via manipulation of their headers in mailbox windows.
- The opening and closing of message windows, and the browsing and printing of message content within the active window. Message windows can also be moved.

Open windows of both types are managed in the form of a single stack with the currently selected (active) window at the top. Only one window can be active at any time. Closing a window causes the next window on the stack (if any) to become active.

The methodology described in Section 7.4 is now applied to this interface.

STAGE 1: *Decomposition:* Figure 7.14 shows a task decomposition for the mail browser. Any sequencing is indicated through node connection at the subtask level. Table 7.3 expands on the diagram by identifying the object(s) associated with each task action, plus any associated side effects and/or option preconditions, which are numbered for reference according to the task to which they relate. Window browsing actions have been omitted in order to simplify the object hierarchy.

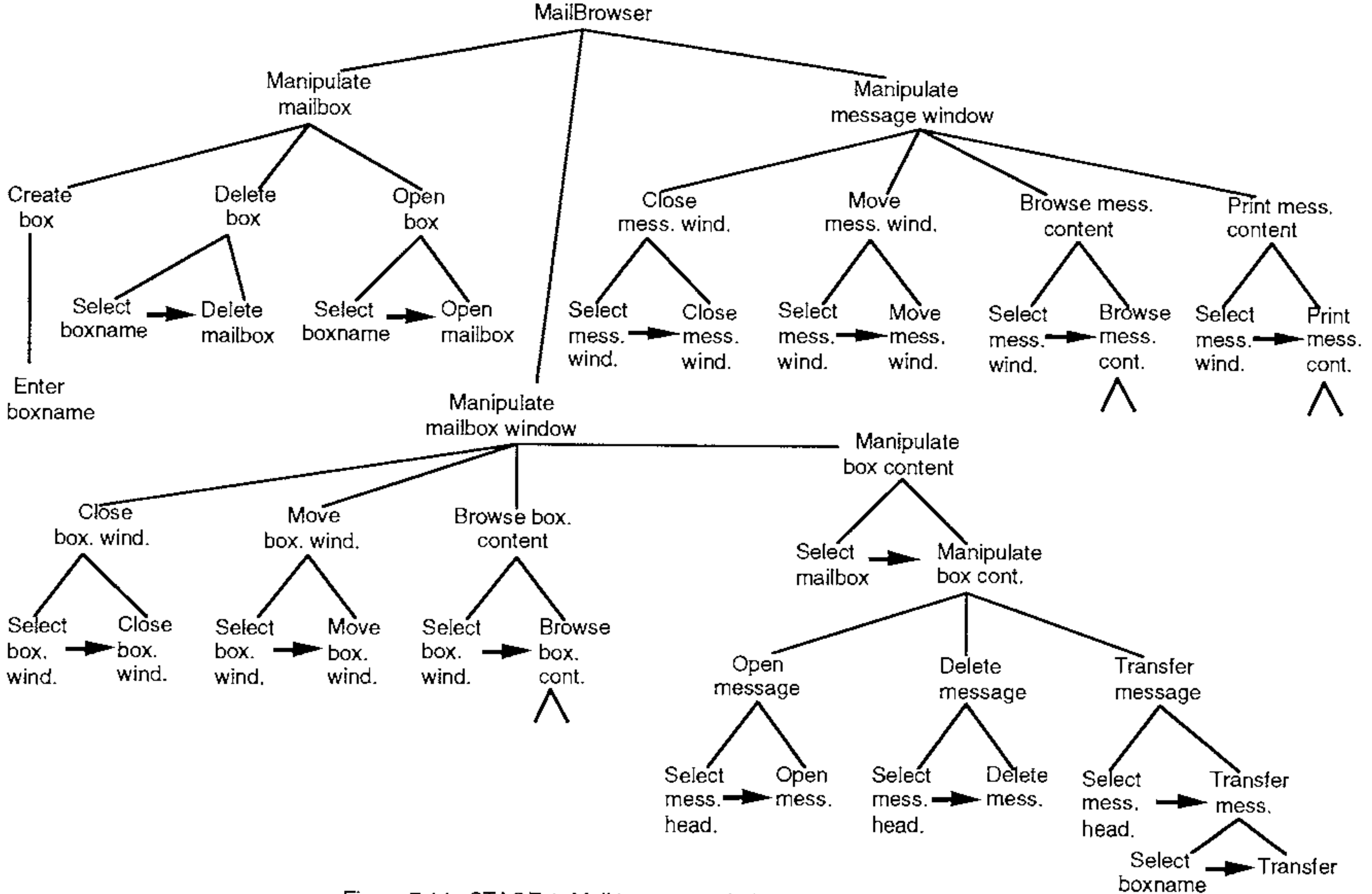


Figure 7.14: STAGE 1: Mail browser task decomposition

User Task	Task Actions Action	 Object	Side effects (S) or Preconditions (D)	
1. Create box	Enter box name Create box	 (box wind.) (mailbox)	 S1:	 Create box wind. Create mailbox
2. Delete box	Select box name Delete box	(mailbox) (box wind.) (mailbox)	 S2: D1:	 Delete box wind. Delete mailbox Mailbox empty
3. Open box	Select box name Open box	(mailbox) (box wind.) (mailbox)	 S3:	 Select box wind. Select mailbox
4. Close box wind.	Select box wind. Close box wind.	box wind. box wind.	 S4:	 Deselect box wind. Select next window on stack (if any)
5. Move box wind.	Select box wind. Move box wind.	box wind. box wind.		
6. Browse box cont.	Select box wind. Browse box cont.	box wind. box wind. (mailbox)		
7. Open message	Select mess. head. Open mess	mess. header (mess. wind.) (mess. body)	 S5:	 Select mess. wind Select message
8. Delete message	Select mess. head. Delete mess.	mess. header (mess. wind.) (mess. body)	 S6:	 Delete mess. wind Delete message
9. Transfer message	Select mess. head. Select box name Transfer	mess. header (mailbox) mess. header (mailbox)		
10. Close mess. wind.	Select mess. wind. Close mess. wind.	message wind. message wind.	 S7:	 Deselect mess. wind. Select next window on stack (if any)
11. Move mess. wind.	Select mess. wind. Move mess. wind.	message wind. message wind.		
12. Browse message	Select mess. wind. Browse mess. cont.	message wind. message wind. (mess. body)		
13. Print message	Select mess. wind. Print mess. cont.	message wind. (mess. body)		

Table 7.3: STAGE 1: Mail browser tasks

Five object types have been identified within the dialogue - *Mailbox, box window, message header, message window* and *message body*. Most of the primitive task actions are directly associated with one of these object types, through being involved in the selection of an existing object instance, the exceptions being *Create box, Delete box*, and *Open box*.

STAGE 2: *Translation:* Figure 7.15 shows the initial Lean Cuisine+ subdialogue trees. With the exceptions of *Createbox, Deletebox*, and *Openbox*, which are attached to the *MailBrowser* root dialogue, options are attached to objects identified in Table 7.3.

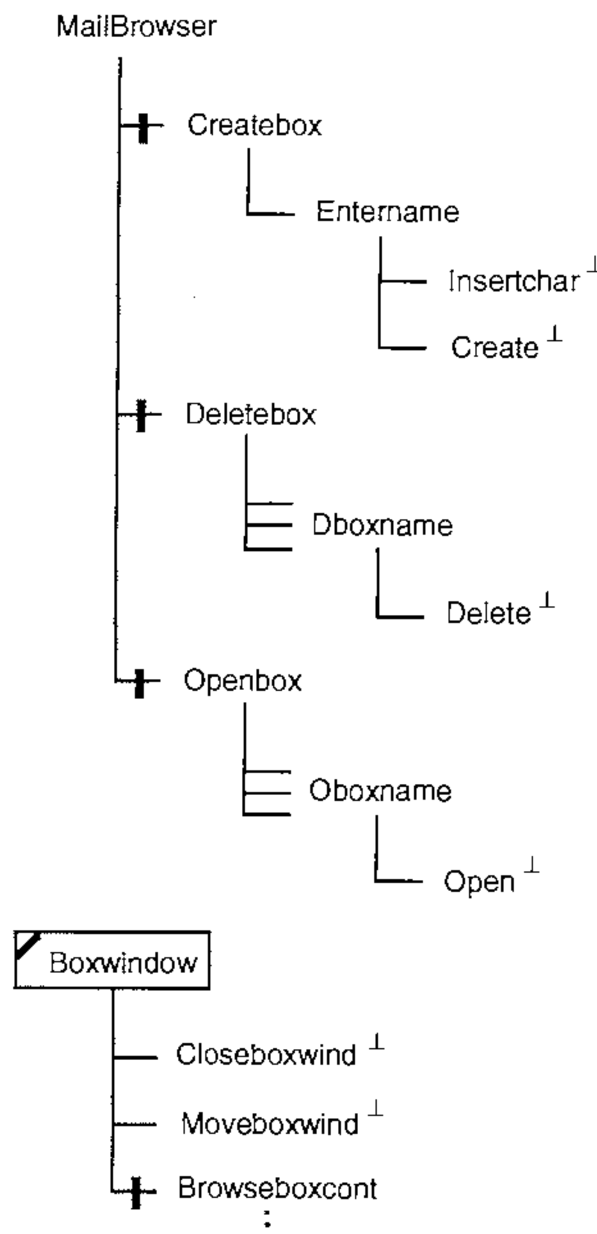


Figure 7.15a: STAGE 2: Initial mail browser subdialogue trees

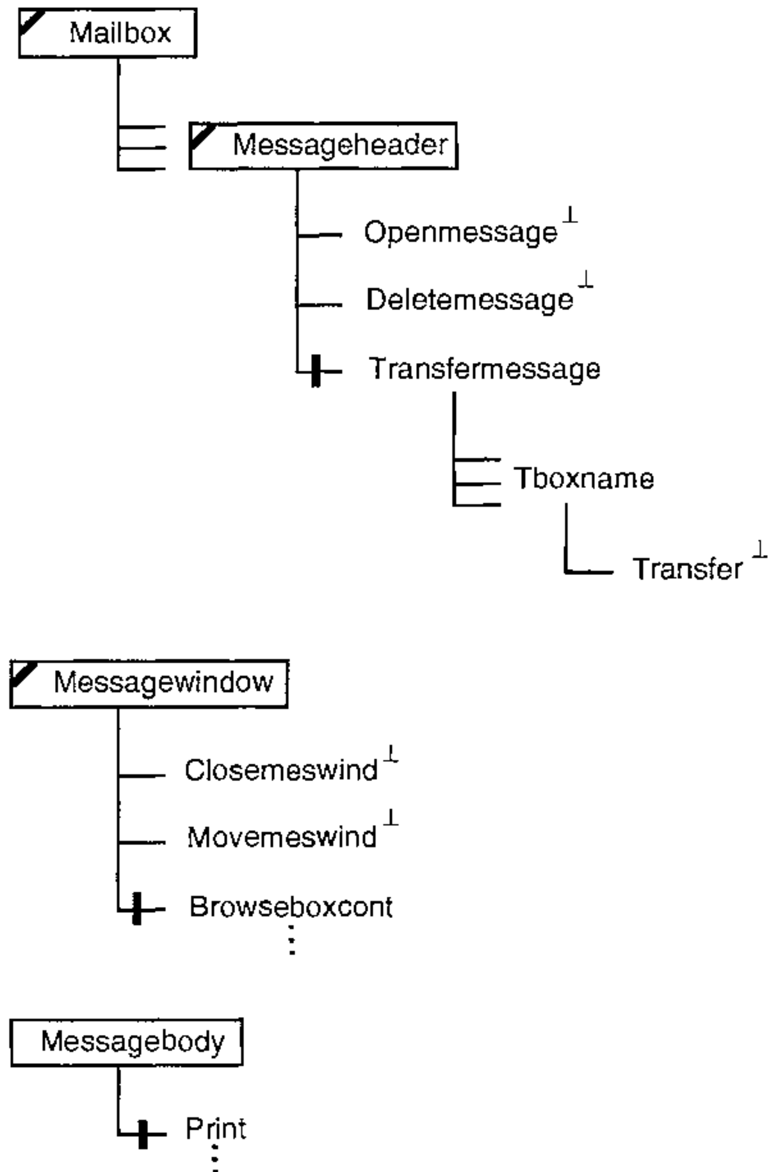


Figure 7.15b: STAGE 2: Initial mail browser subdialogue trees - contd.

STAGE 3: *Synthesis:* A first pass object hierarchy is shown in Figure 7.16. Two mutually exclusive generic objects *Messagewindow* and *Boxwindow* have been defined, each representing a dynamic object instance set in which window instances exhibit mutually exclusive behaviour (thus only one window instance from one of the sets can be selected at any time). Associated with each *Boxwindow* instance is a *Mailbox* object, which in turn is the parent of a dynamic instance set of *Messageheader* objects, again exhibiting mutually exclusive behaviour. Associated with each *Messagewindow* instance is a *Messagebody* object. This object hierarchy is refined in Figure 7.17 through the introduction of a window type hierarchy.

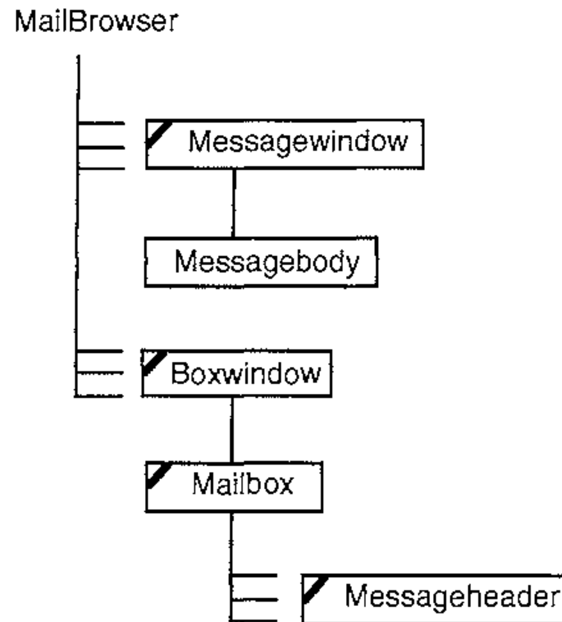


Figure 7.16: STAGE 3: Initial Mail browser object hierarchy

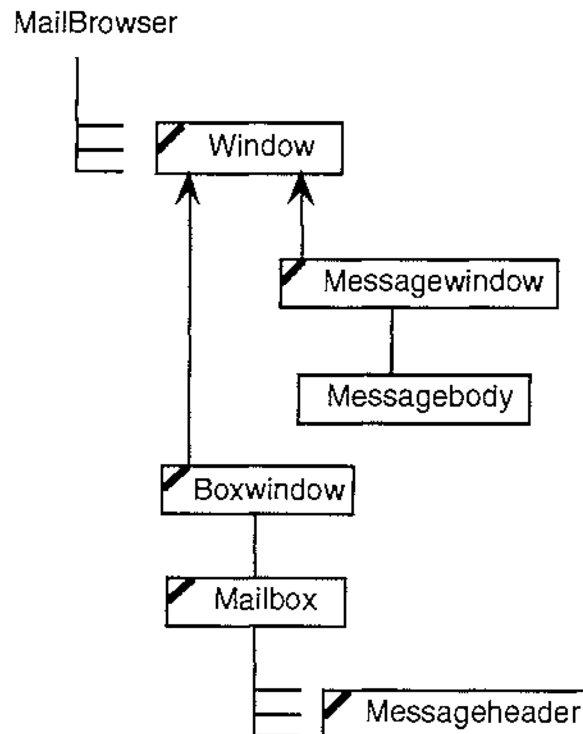


Figure 7.17: STAGE 3: Refined Mail browser object hierarchy

A supertype *Window*, abstracting common *Boxwindow* and *Messagewindow* options, has been introduced in Figure 7.17. The resulting composite dialogue tree in Figure 7.18 now emerges. Through the use of broken arcs in the diagram, the *Close*, *Move* and *Browse* group of options in

the *Window* subdialogue at the supertype level is shown as being mutually compatible with the options in the *Boxwindow* and *Messagewindow* subdialogues at the subtype level. Dialogue levels 1 and 2 (as defined in Chapter 6) are show on the diagram.

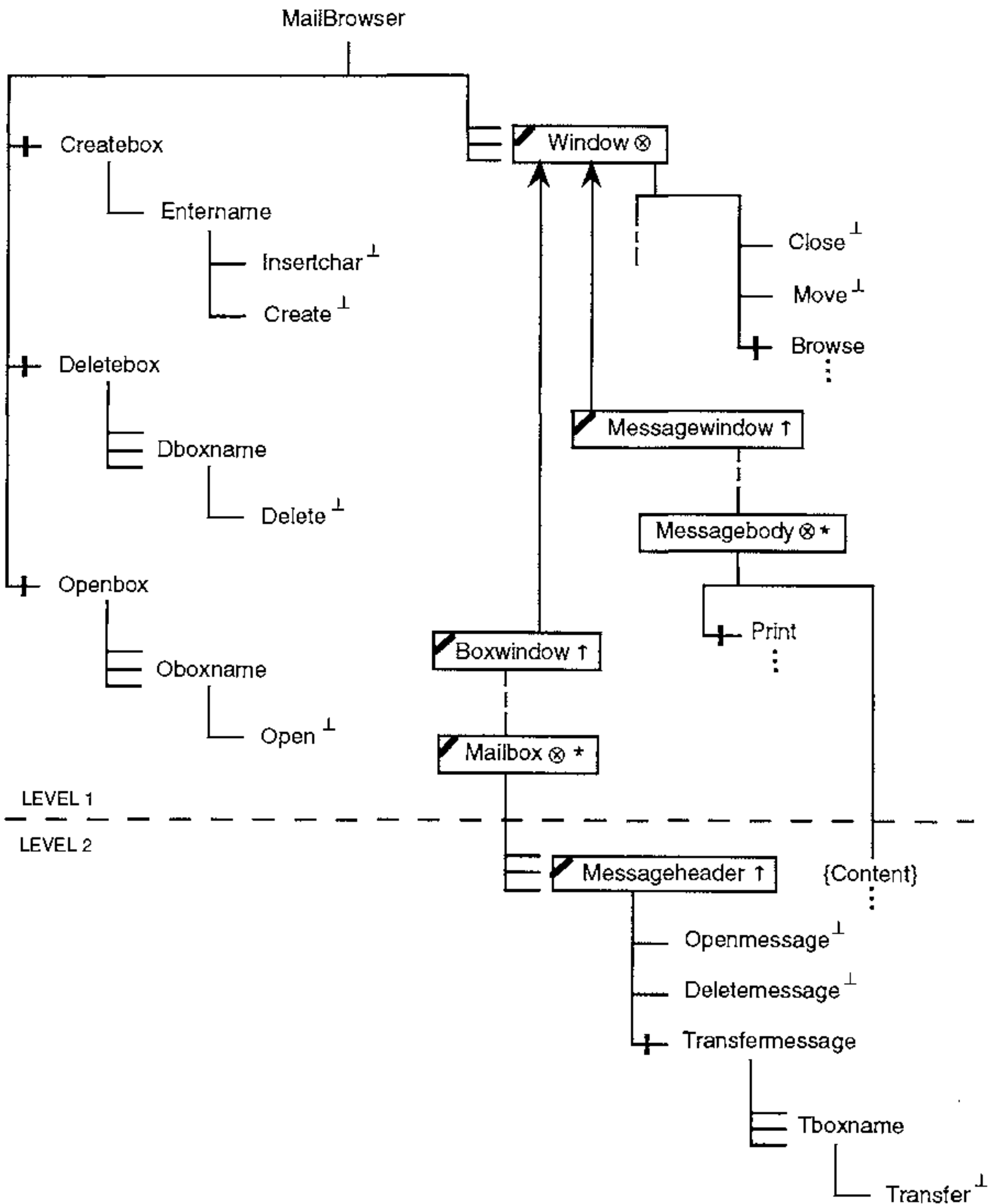


Figure 7.18: STAGE 3: Composite mail browser dialogue tree

Further meneme modifiers have been added in Figure 7.18. The *Window* meneme has been flagged passive (⊗) to indicate that it is user selectable only through its subtypes. Both the *Boxwindow* and *Messagewindow* subtypes are flagged select-only (↑) to reflect the fact that windows cannot be directly deselected by the user. *Messageheader* is flagged select-only for the same reason. The *Mailbox* and *Messagebody* menemes are both flagged as default choice (*) and passive, which means that they are selected whenever their respective parents are selected, and that they cannot subsequently be deselected by the user independently of the parent.

An initial set of state variables is now defined for each object and added to the system dictionary. These are shown in Table 7.4. Each object is given a unique identifier, is classified as being either semantic (from the task domain) or syntactic, and is allocated an object type classification. Object selection and availability indicators are included. The *Messagebody* object is only partially defined, reflecting the fact that manipulation of message content is not being considered in this case study.

Window:	#wname	object name
	wobclass	object class = syntactic
	wobtype	object type = window
	wavstat	object availability status (open, closed)
	wselstat	object selection status (active, inactive)
	wposition	current window position
	wwidth	current window width
	wdepth	current window depth
Boxwindow:	**subtype of Window**	
	#wname	object name (= #bname)
	bwobstype	object subtype = mailbox window
Mailbox	#bname	object name
	bobclass	object class = syntactic
	bobtype	object type = mailbox
	bavstat	object availability status (open, closed)
	bselstat	object selection status (active, inactive)

Table 7.4a: OSVs for mail browser

Messagewindow:	**subtype of Window**	
	#wname	object name (= #mname)
	mwobstype	object subtype = message window
Messageheader:	#mhname	object name (= #mname)
	mhobclass	object class = syntactic
	mhobtype	object type = message header
	mhselstat	object selection status
	mhdirection	message direction (in, out)
	mhaddress	message address (from, to)
	mhdate	message date
	mhsubject	message subject
Messagebody:	#mname	object name
	mobclass	object class = semantic
	mobtype	object type = message
	mselstat	object selection status
	maddfrom	address from
	maddto	address to
	mdate	message date
	msubject	message subject
	msize	message size

Table 7.4b: OSVs for mail browser - contd.

STAGE 4: *Consolidation:* The selection side effects and preconditions identified at Step 1, and listed in Table 7.3, are now added to the dialogue description in the form of overlays. Figure 7.19 shows selection triggers. Triggers relating to side effects listed in Table 7.3. are cross referenced, with the references shown boxed. Other triggers associated with subdialogue termination have been identified within the *Createbox, Deletebox, Openbox,* and *Transfer-message* subdialogues. It should be noted that the effects of the triggers associated with *Create* and *Delete* extend beyond *Boxwindow* to *Mailbox*, as a result of a mutual existence dependency between these objects (shown in Figure 7.20). Thus for example, the creation of a mailbox window also creates an associated mailbox. *Mailbox* is also selected following the opening of a mailbox window, in this case because it is shown as a default choice

within the *Boxwindow* subdialogue. The triggers associated with *Openmessage* and *Deletemessage* similarly impact both the *Messagewindow* and *Messagebody* objects, and *Deletemessage* also deletes the associated *Messageheader* as a result of a further mutual existence dependency (see Figure 7.20).

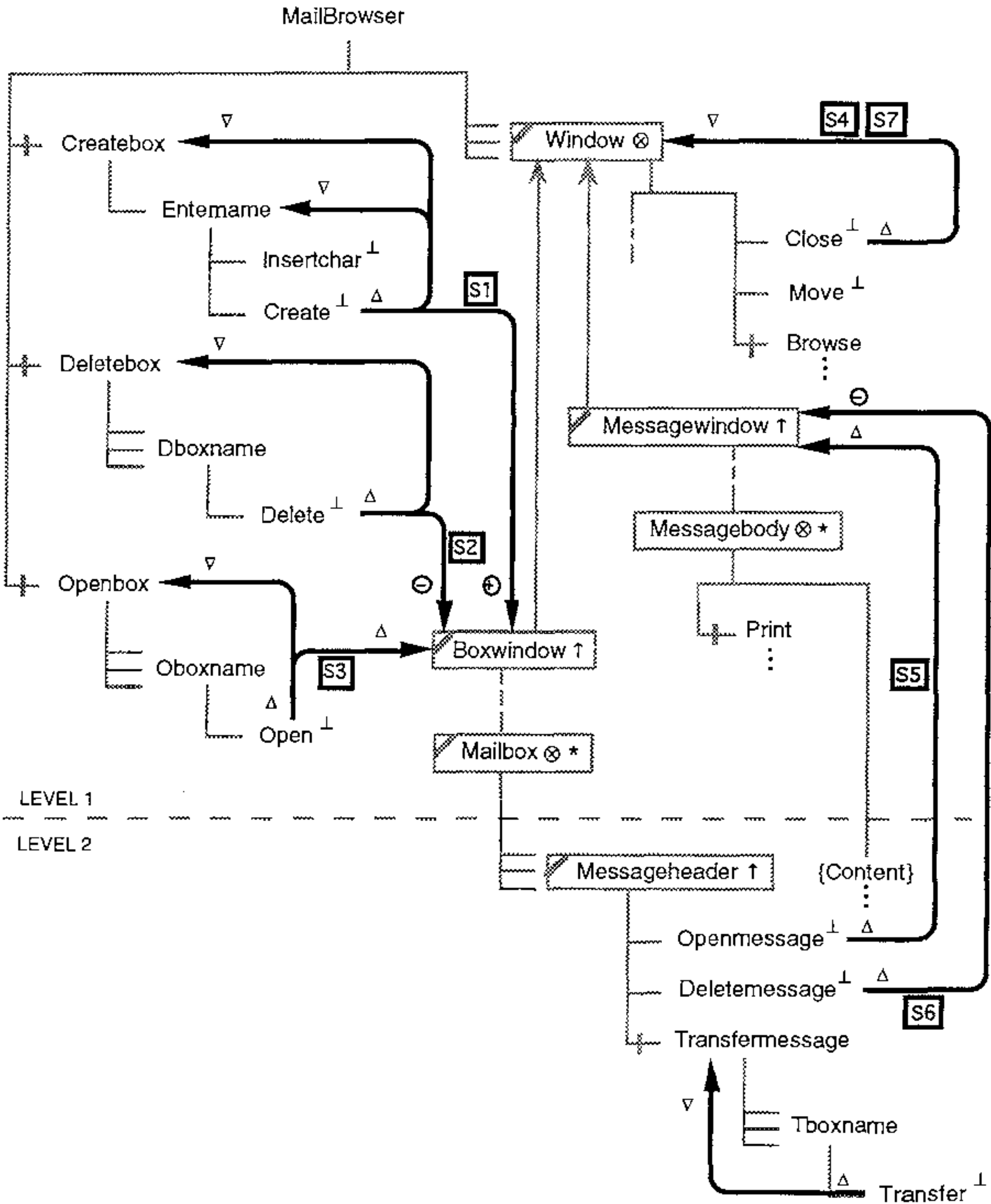


Figure 7.19: STAGE 4: Mail browser selection triggers showing Table 7.3 references

Figure 7.20 shows the two further constraint overlays in a single diagram. The single option precondition, relating to the availability of mailbox names for deletion, is also referenced to Table 7.3. Three mutual existence dependencies involving different object representations relating to mailboxes and messages are also shown.

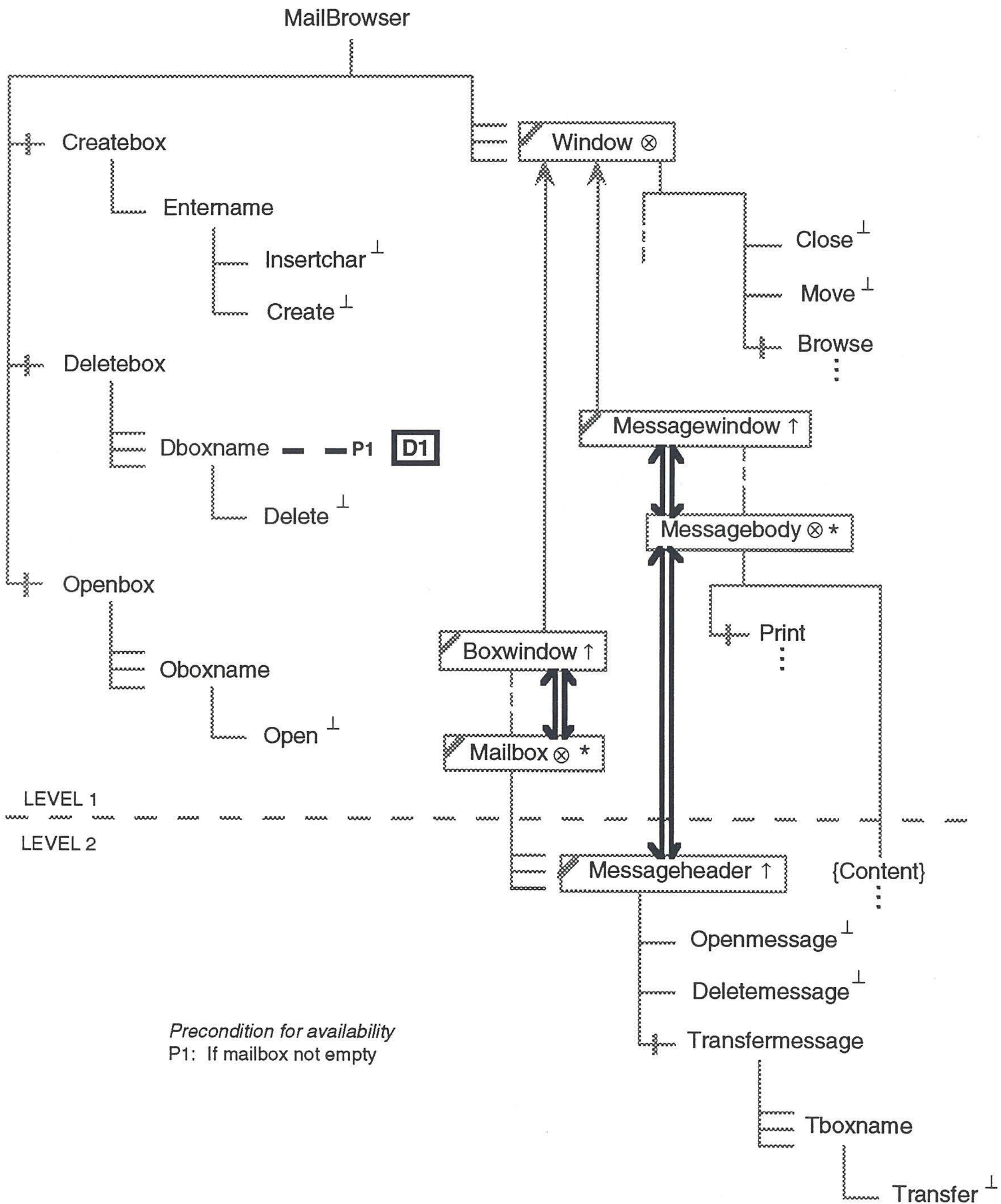


Figure 7.20: STAGE 4: Further mail browser constraints

In Figure 7.21 the task action sequence relating to the higher level *Open box* task is shown. The user selects *Openbox*, then an instance of *Oboxname*, then *Open*, and the system responds by selecting the instance of *Boxwindow* corresponding with *Oboxname*, and also deselecting (terminating) *Openbox*.

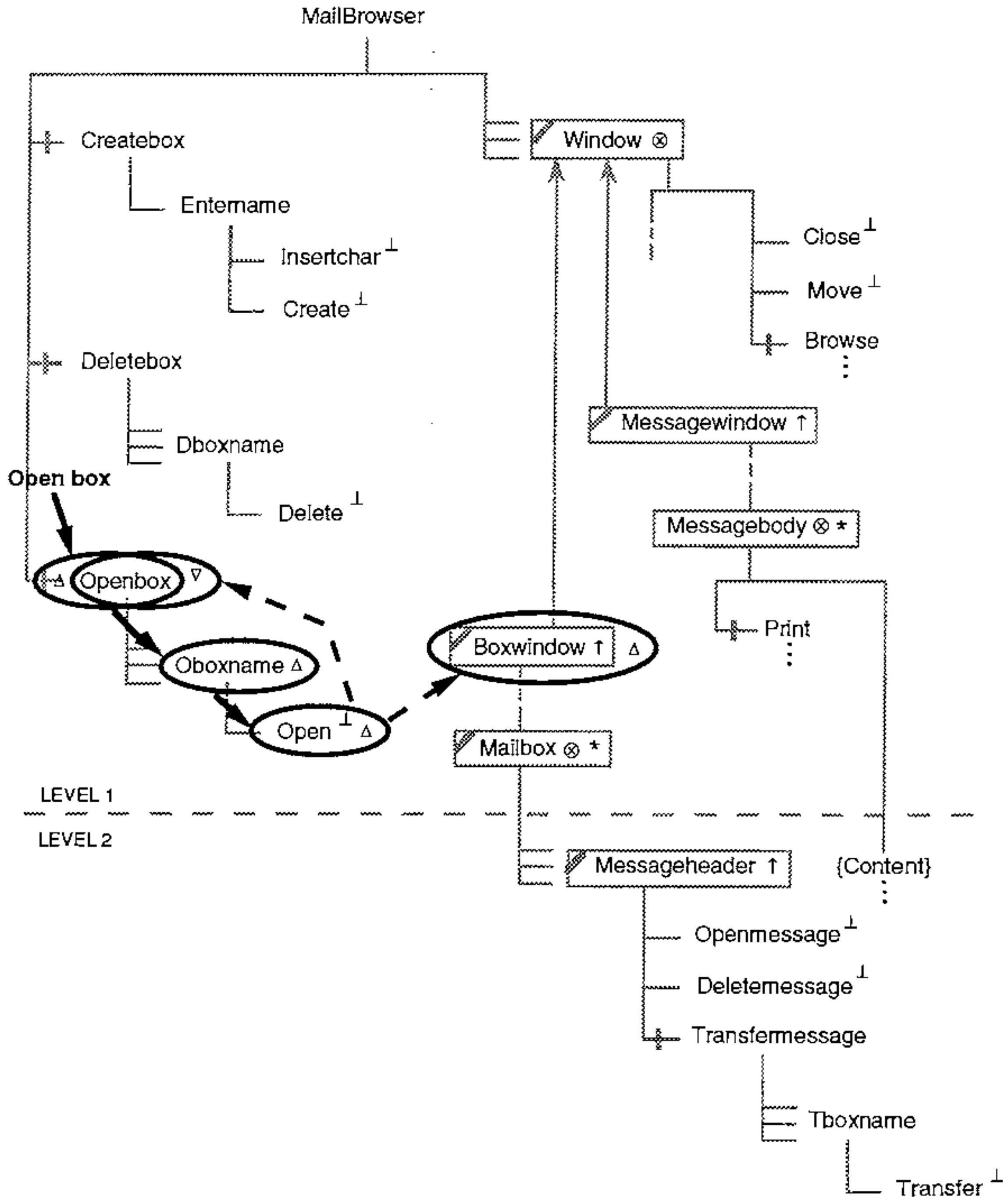


Figure 7.21: STAGE 4: Mail browser *Open box* task action sequence

STAGE 5: *Extension:* A menu system to provide for navigation and control of the mail browser interface is now added, and described in a separate Lean Cuisine+ tree diagram. Menus are represented as syntactic objects, and individual options are represented by terminal menemes. A Lean Cuisine+ tree diagram showing menus relating to mailboxes and messages is shown in Figure 7.22. Option preconditions resulting from constraints previously defined in the mail browser dialogue model are shown.

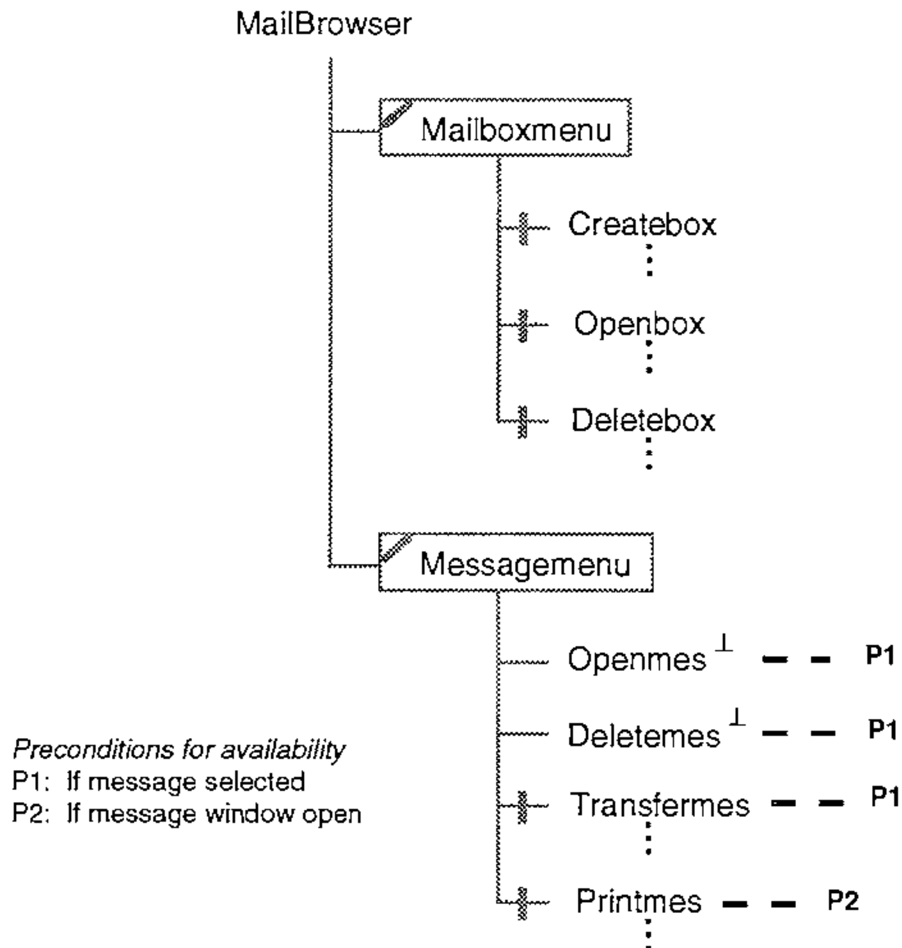


Figure 7.22: STAGE 5: Mail browser menus showing option preconditions

7.7 Review

In this chapter a five stage methodology for constructing Lean Cuisine+ specifications has been developed. Stage 1 comprises a task decomposition to identify primitive task actions involving the selection of objects and operations. Stage 2 involves translation from a task-oriented view to an object-oriented view, and the construction of Lean Cuisine+ subdialogue trees. Stage 3 involves the synthesis of these subtrees to

form a single Lean Cuisine+ dialogue tree. At stage 4, further constraints identified at stage 1 are incorporated in the form of overlays to the dialogue tree, and task action overlays relating to the higher level tasks identified at stage 1 are generated. At stage 5, a menu system facilitating navigation and control is added to the specification. Two case studies involving the application of the methodology to high level interface design have been developed.

Chapter 8

Software Support for Lean Cuisine+

"There is a need for behavioural representation techniques coupled with supporting interactive tools to give a user-centred focus to the interactive development process."

Hartson, Siochi & Hix, 1990

This chapter is concerned with software support for the Lean Cuisine+ notation. In Section 8.1 the requirements of a software environment to support the construction, browsing, and execution of Lean Cuisine+ specifications are established, and in Section 8.2 a prototype of part of this environment is described. The extensions to support execution complete the Lean Cuisine+ notation. Section 8.3 considers a mapping from Lean Cuisine+ into a more formal dialogue specification language in order to support interface prototyping and implementation.

8.1 Requirements of a support environment

A software support environment for Lean Cuisine+ should offer the following functionality:

- Support for the construction, browsing and display of Lean Cuisine+ specifications, through the provision of diagram editors supported by a system dictionary. The designer must be able to switch easily between a Lean Cuisine+ dialogue tree, the overlays showing additional constraints, and the contents of the underlying system dictionary. In terms of functionality these requirements could be compared with those of a CASE tool, e.g., Teamwork (Cadre, 1990). The various components require to be fully integrated, and a variety of display options and printed reports offered. Facilities for structural verification of the specification could also be incorporated, with regard to graph structure and type hierarchy structure.
- Support for the execution of Lean Cuisine+ specifications, providing a simulation of meneme behaviour as 'selections' are made. This must take cognisance of specified constraints, including selection triggers and option preconditions. Execution of a dialogue specification may involve activation of other dialogues or subdialogues

through selection of their headers. In terms of the simulation, this will involve 'switching in' other diagrams. The state of the interface at any stage will be described primarily in terms of meneme states, delineated visually. In order to support the simulation of dynamic interfaces involving variable numbers of menemes, the simulator must also provide for the instantiation of generic objects as a dialogue progresses and selections are made, and for the initialisation and updating of significant object state variables. In a full simulation spatial aspects of the interaction would also have to considered. This issue is addressed in Section 8.3.

The support environment is explored in the next section through an example system of document folders under the Finder.

8.2 A support environment prototype

A prototype of part of a support environment for Lean Cuisine+, exhibiting limited functionality, has been constructed using HyperCard (Apple, 1989b; Goodman, 1990). The Finder document folder system of previous chapters has been used as the example. The prototype simulates the following aspects of the support environment:

- Browsing of the Lean Cuisine+ dialogue tree which forms the basis of a specification, including subdialogues developed in lower level diagrams.
- The switching in and out of further constraints in the form of selection triggers, option preconditions, and existence dependencies. These are overlayed on greyed dialogue tree diagrams. Within a particular class of constraint, either the constraint subset relating to a specified meneme, or *all* constraints can be displayed.
- Display of state variables, either in relation to a specified object, or as parameters to an operation which may modify them. These are shown in a separate window.
- Display of task overlays. The sequence or combination of meneme selections relating to a specified higher level user task can be overlayed on a greyed dialogue tree. This provides a decomposition of the higher level task into primitive task actions, capturing any temporal relationships between task actions, and forming a link between object-oriented and task-oriented views of the dialogue.
- Execution of a specification. This is presented in the form of a specific session for the Finder document folder system. A particular start-up state, in terms of icon and window instances, is assumed, i.e. the initialisation of the desktop is ignored. Window and directory sizes are also ignored. Menemes are selected by 'pointing and clicking'. As the session progresses, meneme states are delineated visually: normal presentation implies that a meneme is unselected, reverse video that a meneme is in a selected state, and the greying of either of these representations that a meneme is currently unavailable for selection.

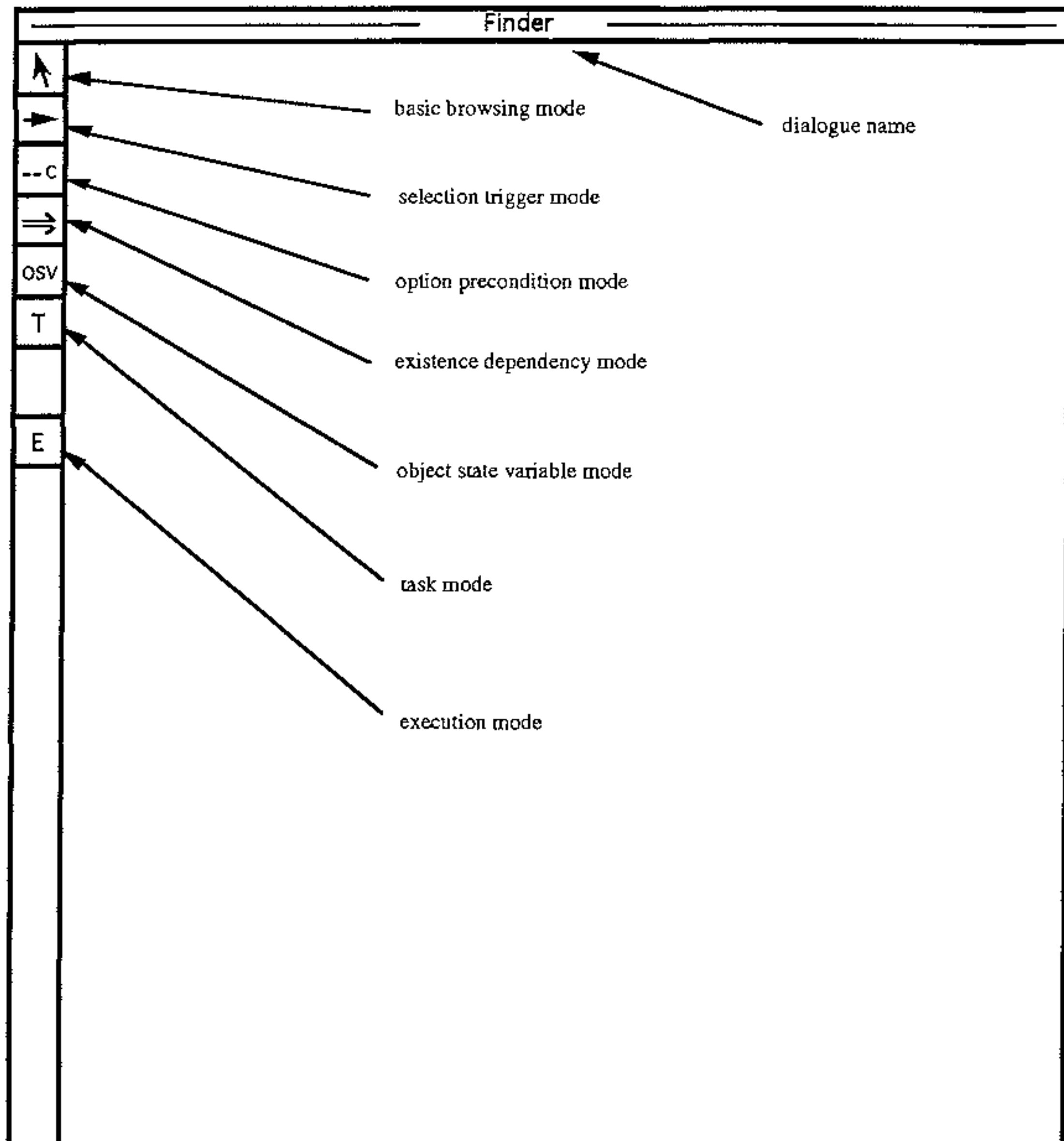


Figure 8.1: Support environment window format

The support environment is displayed in a window offering a palette of selectable options, each option relating to a particular 'mode' within the environment, as shown in Figure 8.1. Seven modes are provided, each represented iconically in the palette:

- a basic browsing mode for exploring the dialogue tree;
- three modes relating to the display of further constraints: selection triggers, option preconditions, and existence dependencies;
- a mode relating to the display of object state variables;
- a task mode concerned with the display of task action sequences / combinations; and

- an execution mode, supporting simulation of the behaviour of the interface during interaction.

The HyperCard mock-up of the support environment, which is based on the Finder document folder system, provides for:

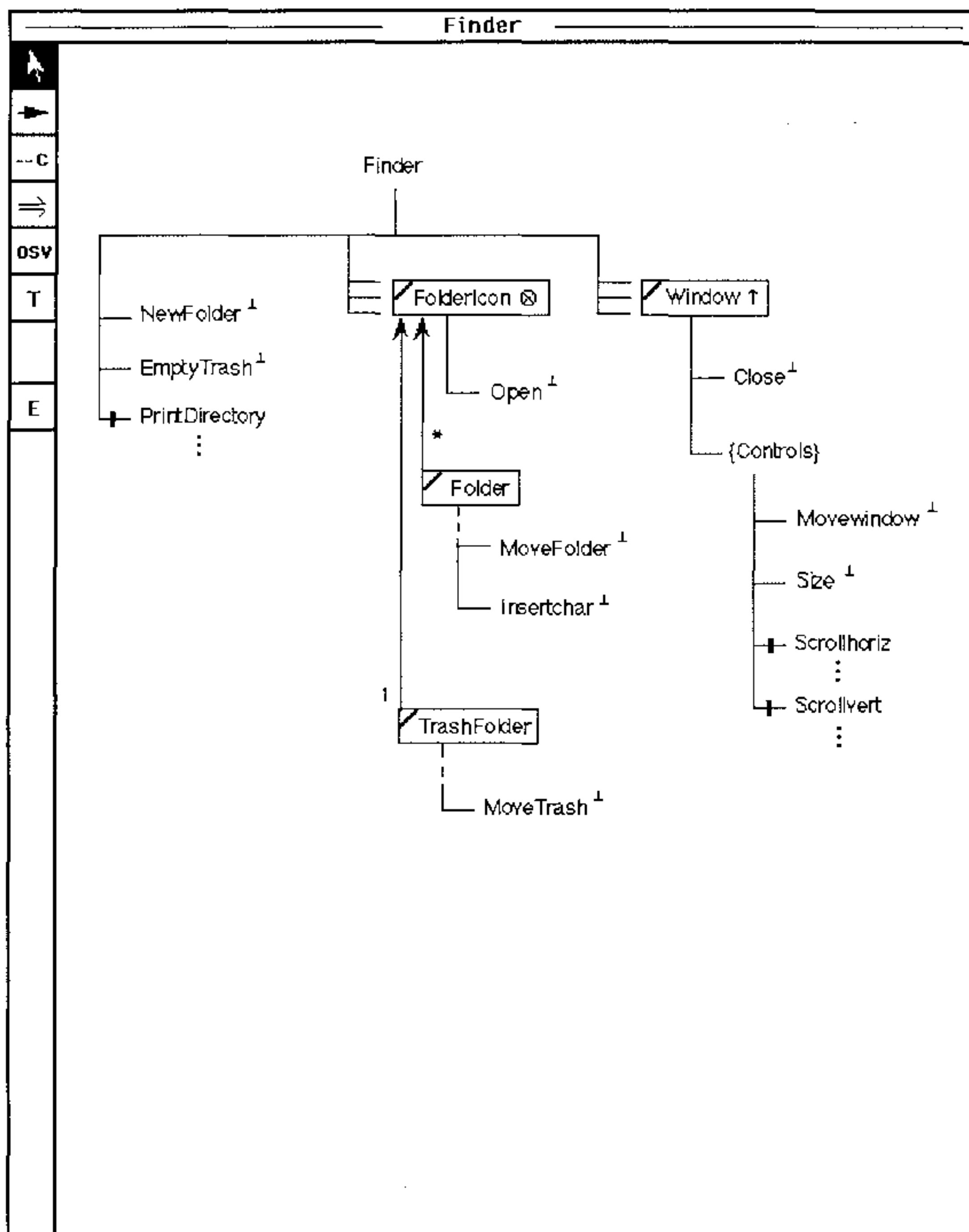


Figure 8.2: Basic browsing mode showing base layer dialogue tree

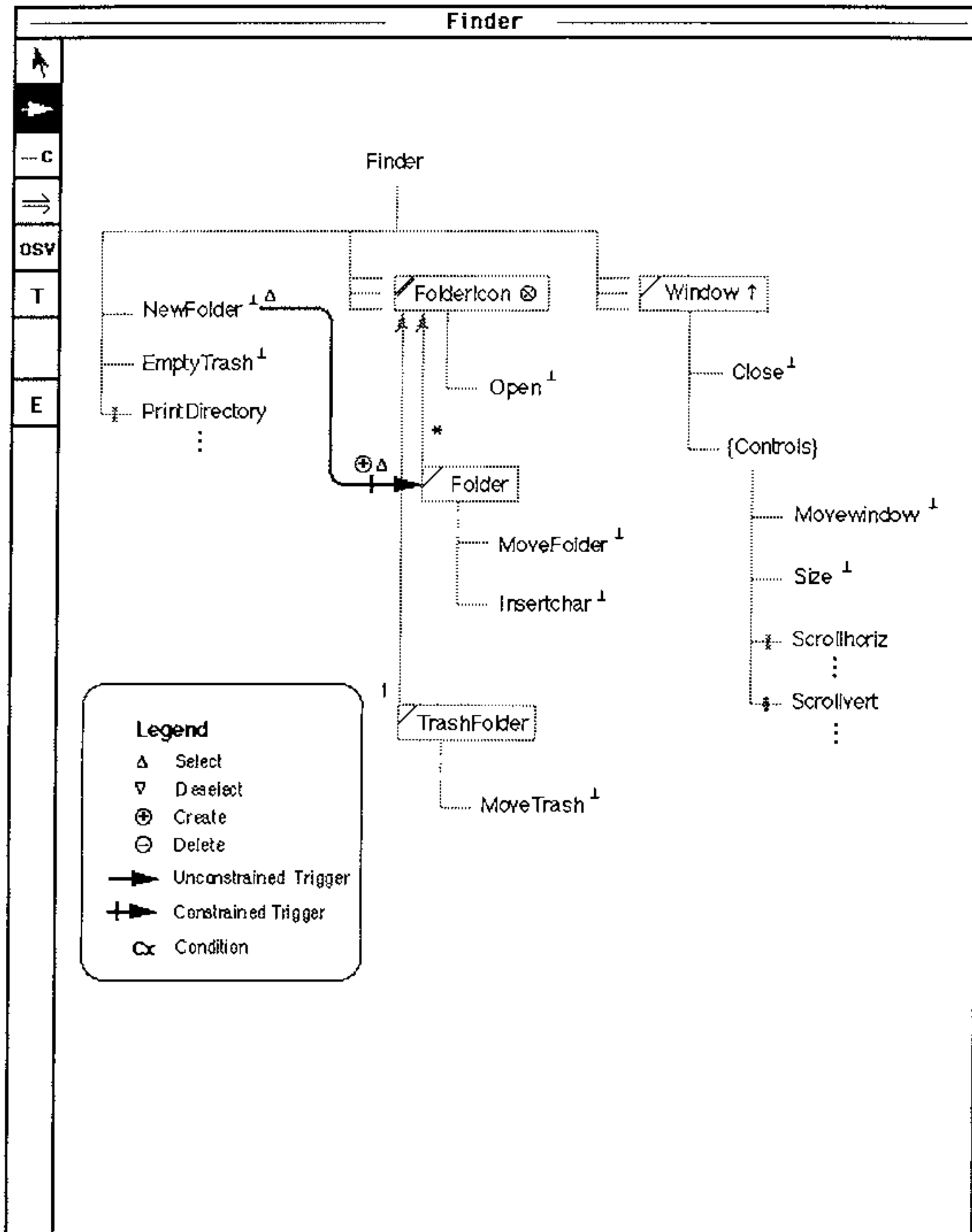


Figure 8.3: Selection trigger overlay for *NewFolder*

- the browsing of parts of the Lean Cuisine+ description of this system, including the base layer dialogue tree and constraint overlays, and
- an execution of the specification based on a specific sequence of interaction tasks.

The full simulation is presented in detail in Appendix D, with representative examples shown here. Figure 8.2 shows the display of the Lean Cuisine+ dialogue tree diagram in basic browsing mode. Tree diagrams for modal subdialogues such as *PrintDirectory* are displayed by pointing and clicking on the appropriate meneme, thus providing for navigation of the multi-level specification.

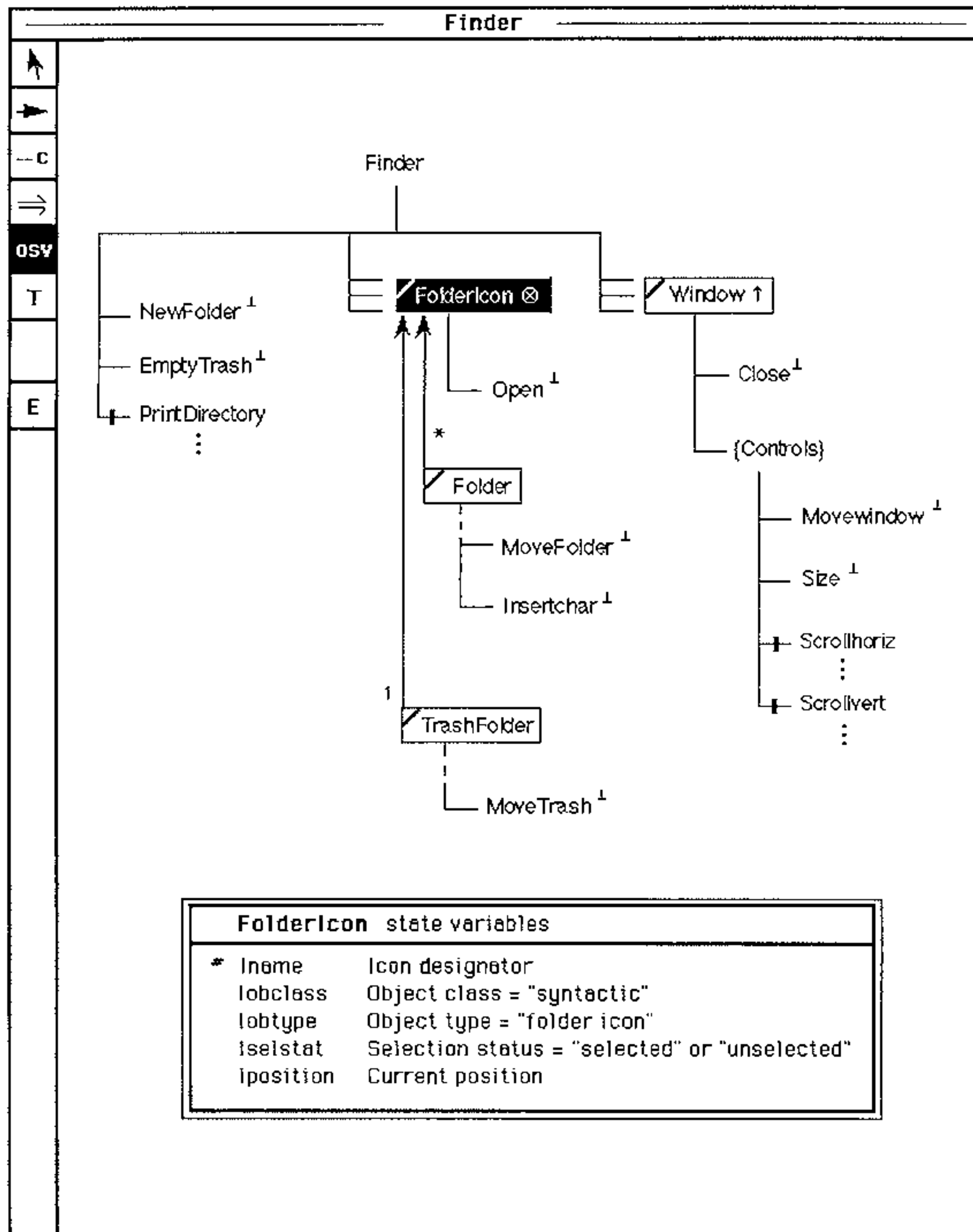


Figure 8.4: OSV mode showing state variables for *FolderIcon*

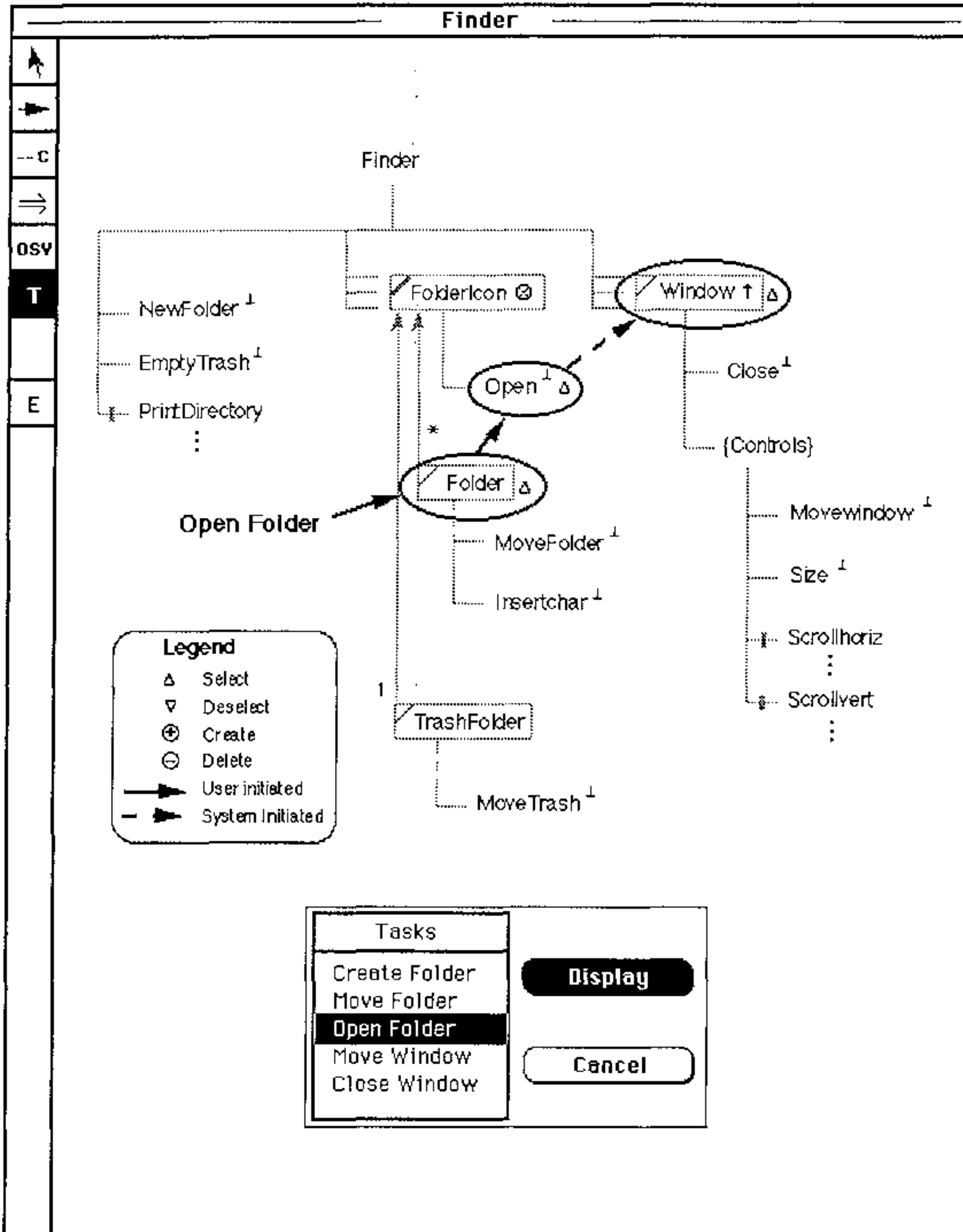


Figure 8.5: Task overlay for task *Open Folder*

Figure 8.3 shows the display of a selection trigger over a greyed tree diagram for the *NewFolder* option. Object state variables for the *FolderIcon* object are shown displayed in Figure 8.4, and the task action sequence for the *Open Folder* task is shown in Figure 8.5. In each case the icon representing the mode is highlighted in the palette.

Execution of the specification

The execution of the specification is based on the small system of icons and windows introduced in Figure 2.4, and reproduced below in Figure 8.6. For the purposes of this simulation, folder *A* represents the system disk. Folders *B* and *C*, which are on the desktop, contain respectively folders *F* and *G*, and *M* and *N*, and folder *F* contains folder *K*.

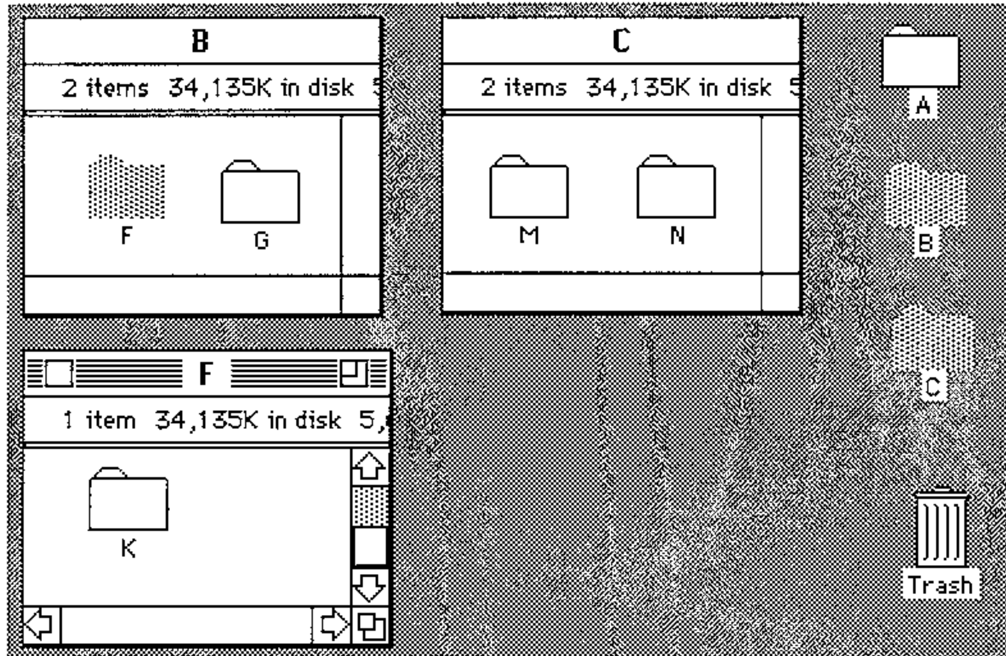


Figure 8.6: Finder simulation: example icon/window system

A sequence of operations is undertaken, commencing with folders *A*, *B* and *C* in a closed state on the desktop, and with folder *A* selected. The initial OSV values, which are a subset of those defined in Section 5.2, are shown in Table 8.1. A folder is deemed to be available if its parent folder is open, i.e., spatial considerations are ignored. *iselstat* represents the current *FolderIcon* object selection status (s = selected, u = unselected). *ifostat* represents the current *FolderIcon* status (o = open, c = closed). *favstat* represents the current *Folder* object availability status (a = available, n = not available). *wavstat* represents current *Window* object availability (c = closed, o = open) and *wselstat* represents current *Window* object selection status (a = active, i = inactive).

The input of these values and the initial populating of the interface with object instances are ignored in the simulation which follows. These values determine initial meneme availability and impact the appearance of the Lean Cuisine+ specification in execution start-up state. In general, the initial dialogue state would also be impacted by any default or system selected menemes.

***FolderIcon/Folder* object instances:**

#iname	A	B	C	F	G	K	M	N
iobtype	ic	ic	ic	ic	ic	ic	ic	ic
iselstat	s	u	u	u	u	u	u	u
ifostat	c	c	c	c	c	c	c	c
fobsubtype	fo	fo	fo	fo	fo	fo	fo	fo
favstat	a	a	a	n	n	n	n	n
fparent	-	A	A	B	B	F	C	C

***FolderIcon/TrashFolder* object instance:**

#iname	T
iobtype	ic
iselstat	u
ifostat	c
tobsubtype	tr

***Window* object instances:**

#wname	A	B	C	F	G	K	M	N	T
wobtype	wi	wi	wi	wi	wi	wi	wi	wi	wi
wavstat	c	c	c	c	c	c	c	c	c
wselstat	i	i	i	i	i	i	i	i	i

Table 8.1: Finder simulation: initial OSV values

Table 8.2 shows the states of the icon set and window stack at each stage of the simulation, and Appendix D contains a related series of diagrams with accompanying narrative. The major aspects of this dialogue, with reference to the stage numbers in Table 8.2, are described below. Extensions to the Lean Cuisine+ notation to support execution are introduced with reference to diagrams reproduced from Appendix D, and are summarised at the end of this section.

- The Finder is started (Stage#1). The state of the dialogue at this stage is shown in Figure 8.7. With reference to this diagram:
  - Three menemes are initially selected, the selected state being represented by reverse video. The *Finder* dialogue header remains selected throughout. The system disk icon, *A*, is the initial default folder choice.
  - The names of the currently selected folder and window instances (if any) are displayed in 'tabs' above the respective object boxes. These tabs appear only in execution mode, and the single character restriction on names introduced earlier remains. *A* is shown as the currently selected folder instance. The arrows to

the right of these boxes, which also appear only in execution mode, provide a means of accessing the current sets of folders and open windows (see Figures 8.8 and 8.9).

Stage	Operation or State	Folder icon Set	Trash Icon	Window Stack
#0	Start-up Finder			
#1	At start-up	**A** B C	T	
#2	Select folder B	**A** B C	T	
	Folder B selected	A **B** C	T	
#3	Open folder B	A **B** C	T	
	Window B open	A **B** C F G	T	**B**
#4	Select folder C	A **B** C F G	T	**B**
	Folder C selected	A B **C** F G	T	**B**
#5	Open folder C	A B **C** F G	T	**B**
	Window C open	A B **C** F G M N	T	**C** B
#6	Select folder F	A B **C** F G M N	T	**C** B
	Folder F selected	A B C **F** G M N	T	**B** C
#7	Open folder F	A B C **F** G M N	T	**B** C
	Window F open	A B C **F** G K M N	T	**F** B C
#8	Folder F deselected	A B C F G K M N	T	**F** B C
#9	Select window B	A B C F G K M N	T	**F** B C
	Window B active	A **B** C F G K M N	T	**B** F C
#10	Create new folder in B	A B C F **G** K M N	T	**B** F C
	New folder # created	A B C F G K M N **#**	T	**B** F C
#11	Rename new folder as D	A B C F G K M N **#**	T	**B** F C
	Folder renamed D	A B C **D** F G K M N	T	**B** F C
#12	Move folder D to Trash	A B C **D** F G K M N	T	**B** F C
	Folder D in Trash	A B C D F G K M N	T	**B** F C
#13	Trash folder selected	A B C D F G K M N	**T**	**B** F C
#14	Open Trash folder	A B C D F G K M N	**T**	**B** F C
	Trash window open	A B C D F G K M N	**T**	**T** B F C
#15	Empty Trash (of D)	A B C D F G K M N	**T**	**T** B F C
	Trash folder emptied	A B C F G K M N	**T**	**T** B F C
#16	Close Trash window	A B C F G K M N	**T**	**T** B F C
	Trash window closed	A B C F G K M N	**T**	**B** F C
#17	Close window B	A B C F G K M N	**T**	**B** F C
	Window B closed	A **B** C K M N	T	**F** C
#18	Close window F	A **B** C K M N	T	**F** C
	Window F closed	A B C M N	T	**C**
#19	Close window C	A B C M N	T	**C**
	All windows closed	A B **C**	T	
#20	Shut down Finder	A B **C**	T	

Table 8.2: Icon set and window stack at each stage of the simulation

Notes:
1. Currently selected instances are shown bold underlined at each stage.
2. # is the default new folder name.

- Available options are displayed normally. At this stage *PrintDirectory* is available in the root dialogue, and *Open*, *MoveFolder*, or *Insertchar* could be selected for application to the current folder. The behaviour of monostable menemes such as *Open*, which remain selected only until the operation they represent is complete, is demonstrated in the full simulation which appears in Appendix D.
- Unavailable options are greyed. The preconditions for availability of these menemes are defined in the option precondition overlay of the Lean Cuisine+ dialogue specification. At this stage, the entire *Window* subdialogue tree is greyed (it has been assumed that no windows were left open at the last shutdown), as are the *NewFolder* and *EmptyTrash* options at the root level, and *MoveTrash* in the *TrashFolder* subdialogue.

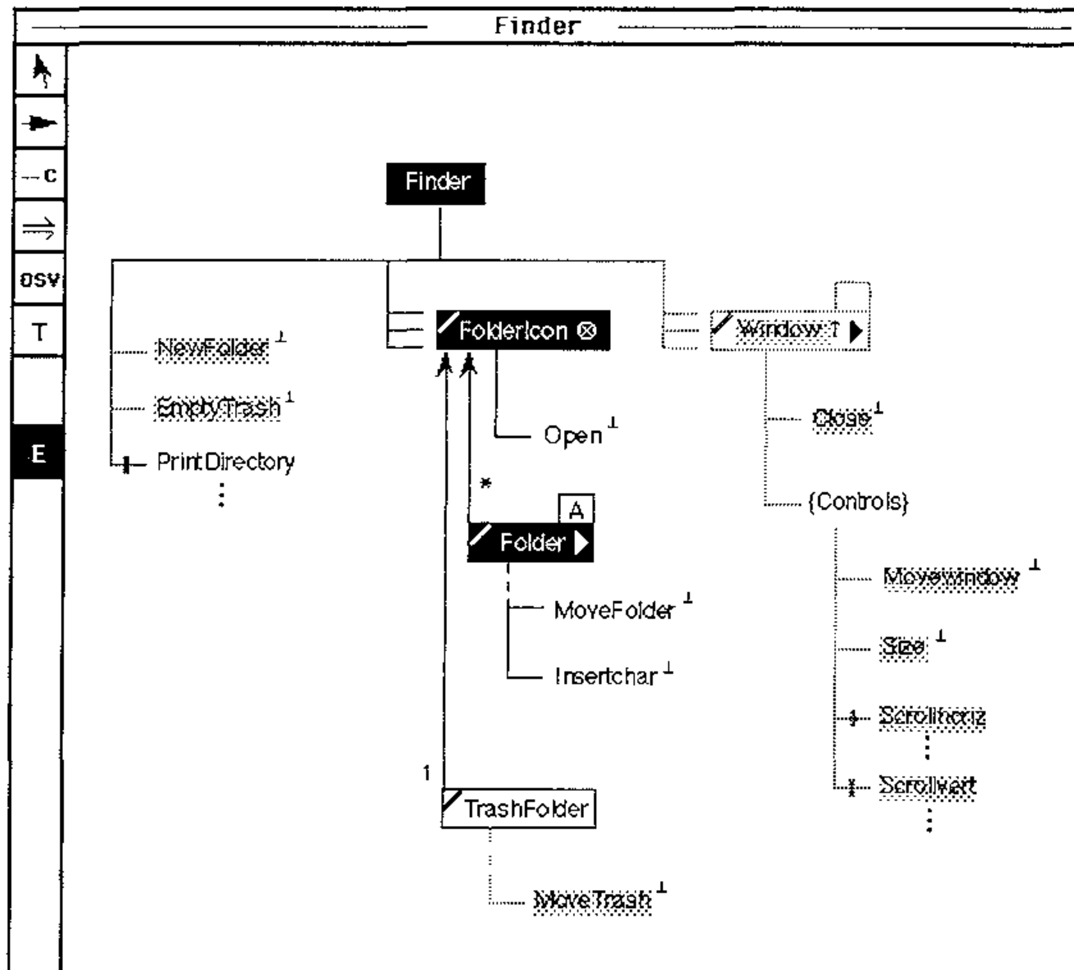


Figure 8.7: Finder simulation Stage #1: start-up state

- Folder instances *B*, *C* and *F* are successively selected and opened to produce the state shown in Figure 8.8 (Stage #7). With reference to this diagram:
  - Folder *F* and window *F* are the currently selected object instances.
  - The *Window* subdialogue tree is no longer greyed. Both the *Close* command, and a range of controls are available for selection for window *F*. Because a window (other then the Trash window) is active, the *NewFolder* option at the root level is also now ungreyed and available.
  - Part of the currently available folder set has been displayed through a series of pull-down menus which reflect the hierarchical parent-child folder structure. For each parent the subset is shown alphabetically. Open folders, providing access to their children, are denoted by means of a right arrow. The current folder instance can be changed through selection from these menus.
- Some folder icon and window swapping is undertaken, leaving *B* as the active window (Stage #9).

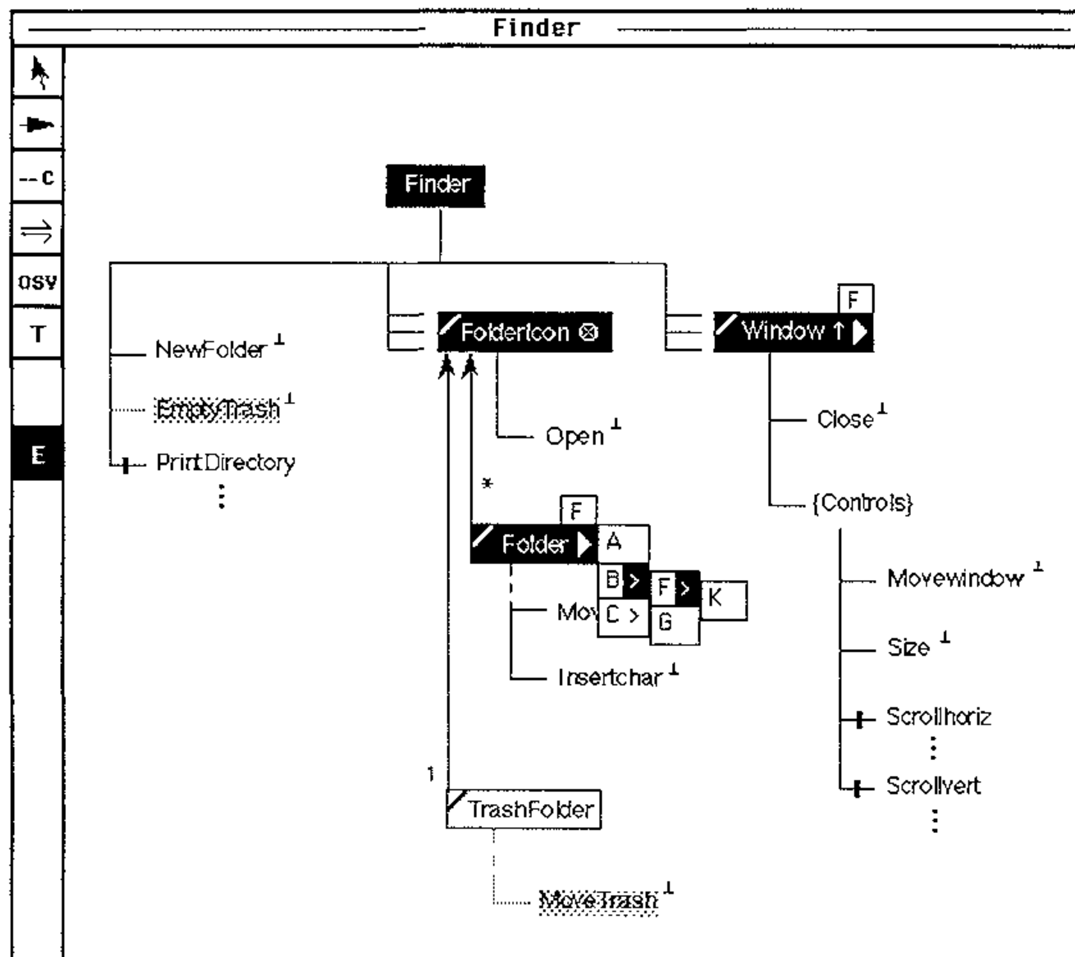


Figure 8.8: Finder simulation Stage #7: folders B, C and F open

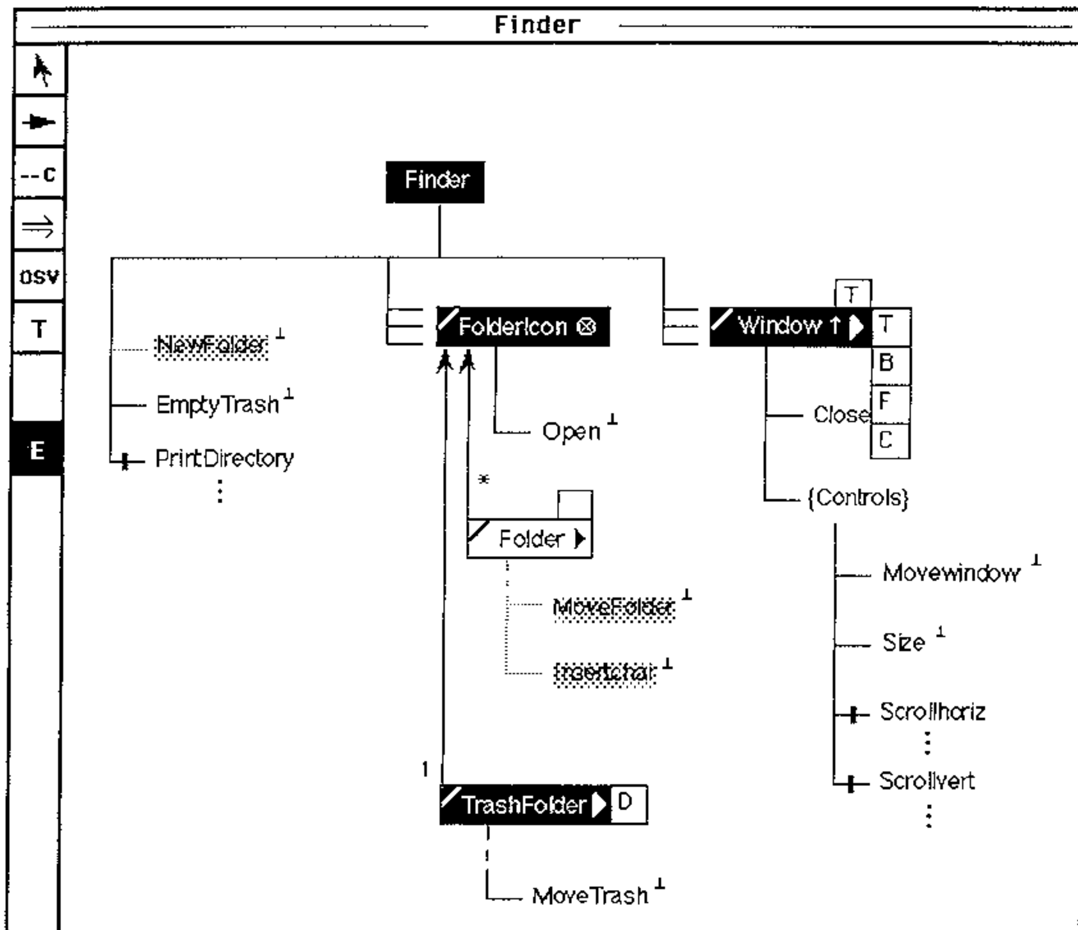


Figure 8.9: Finder simulation Stage #14: Trash folder open

- A new folder instance is created and renamed *D* (Stage #11).
- *D* is moved to the Trash folder, *T*, and then *T* is opened, producing the state shown in Figure 8.9 (Stage #14). With reference to Figure 8.9:
  - The Trash folder *(T)* and window *T* are the currently selected object instances.
  - *NewFolder* is now greyed at the root level, as it does not apply to window *T*. *MoveFolder* and *Insertchar* are greyed as no folder instance is currently selected.
  - Because the Trash folder is no longer empty, *EmptyTrash* is now ungreyed and is available at the root level.
  - The current window stack has been displayed in a pull-down menu. The names are displayed in stack order from the top down. The current (active) window can be changed through selection from this menu.
  - The contents of the Trash folder (currently folder *D*) have been displayed in a pull-down menu.
- The Trash folder, *T*, is emptied (of *D*) and then closed (Stage#16).

- The four open windows, *T*, *B*, *F*, and *C* are successively closed (Stage #19).
- The Finder is shut down (Stage #20).

A number of extensions to the Lean Cuisine+ notation in connection with its execution have been made in the above simulation. These are:

- The use of reverse video and greying to indicate respectively selected and unavailable menemes in any state.
- The addition of tabs to object boxes in the case of generic objects to permit display of the currently selected instance.
- The addition of right arrows to object boxes to provide access to current instantiations of generic objects.

These extensions form part of the Lean Cuisine+ notation, which is fully defined in Appendix C.

8.3 Prototyping the interface under design

In the above simulation, spatial aspects of the interaction are ignored with one exception - the moving of folder *D* into the Trash folder. Moving a folder involves the selection of a new position on the desktop, and may also lead to the selection of a new parent folder as in this instance. This is achieved in the simulation, as shown in Appendix D (Stage #12), through selection of *T* from an array of possible parent names. The simulation works least well in this connection. To fully simulate the behaviour of the system the positions of folder icons and windows, and the relative sizes of windows and directories, would have to be defined and subsequently maintained as commands such as *MoveFolder* were executed. This would however remain an 'arms length' approach.

A better solution would be to extend the support environment to provide for the generation of a prototype implementation of the interface under design. This would require a means of mapping from Lean Cuisine+ into a dialogue specification language. This is considered below. The support environment could then provide for manipulation of both the Lean Cuisine+ specification and the associated prototype in tandem in separate windows. A mock-up involving Stage #7 of the Finder simulation is shown in Figure 8.10. Under such a regime, the designer could either make selections in the Lean Cuisine+ specification which would be reflected in the prototype, or manipulate the prototype, and bring about changes (including selections and deselections) in the Lean Cuisine+ model. Spatially based operations, such as moving a folder, would be better undertaken in the prototype. The environment could be yet further extended to become a full user interface development environment (UIDE). This extension is listed under further work in Chapter 9.

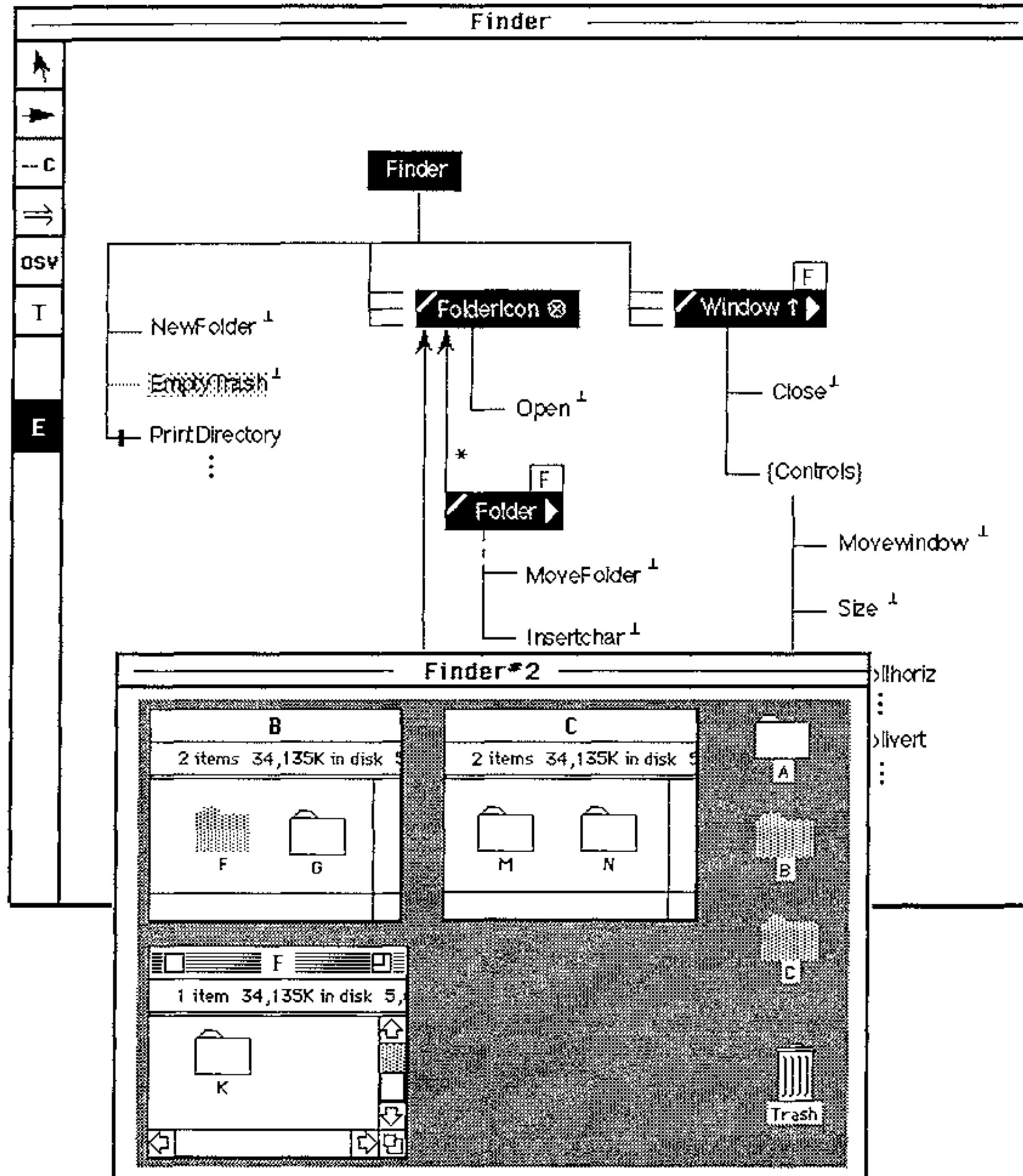


Figure 8.10: Finder simulation Stage #7 under tandem execution

Mapping to a dialogue specification language

Apperley (1988) applies the Lean Cuisine notation to the design of a menu system for a simple interactive application, a combined Text and Paint program called TaP. The design is taken through a number of stages from specification through preliminary analysis to the final design, with Lean Cuisine being used as a diagrammatic design tool. Examples of a textual form of Lean Cuisine which could be used for a direct implementation of the interface are also included.

Apperley & Spence (1989) go further and examine the relationship between the Lean Cuisine notation and the event response system (ERS) model (Hill, 1987) for menu systems (the ERS was briefly reviewed in Section 3.2.3). Lean Cuisine is considered to be closer to the menu designer and ERS closer to the implementor. Both are based on the notion of an asynchronous set of events and the effect of these on a set of flags which define a total state. The mutually compatible and mutually exclusive groupings of menu options are explicit in Lean Cuisine but implicit in ERS. In contrast, events are implicit in Lean Cuisine and explicit in ERS. With its event bias, ERS clearly expresses the effect of each possible user action with a minimum of context, in terms of the incremental effect on the individual flags. Because it is more formal, the ERS model is seen by Apperley & Spence as offering a possible route to the implementation of the Lean Cuisine notation for menus.

Further to this, two approaches to prototyping interfaces described using the Lean Cuisine notation have been developed more recently at Massey University:

- the modified event response system (MERS) notation (Anderson & Apperley, 1990), which has been applied to menu systems; and
- dialogue activation language (DAL) (Anderson, 1993), a dialogue specification language which supports dynamic interfaces, and which could be extended to support prototyping from Lean Cuisine+ specifications.

These approaches are considered further below, and an outline mapping into DAL is developed for the Lean Cuisine+ specification of the Finder example.

The modified event response system (MERS) notation

Event response systems have lacked a graphical notation. Anderson & Apperley (1990) modify Hill's ERS notation in order to accommodate Lean Cuisine concepts, and show that for non-dynamic menu-based interfaces, Lean Cuisine diagrams can be mapped into a modified ERS from which a prototype interface can be generated.

As a consequence of the bistable nature of menemes, two rules are associated with each meneme, corresponding to setting and clearing respectively. Side effects concerning propagation of selection via virtual menemes are also considered. A dummy start-up event is required to cope with default settings, and a means of setting default choice menemes in response to the clearing of all flags within a required choice group is necessary. A prototyping environment is implemented on the Macintosh as two programs: the first takes a Lean Cuisine specification in textual form and outputs an interface definition file (IDF); an interpreter then takes the IDF and simulates the interface that it describes. The menus are implemented as pull-down menus attached to the standard Macintosh menu bar. The control layer is implemented as an MERS interpreter, with an event response loop that repeatedly fires all fireable ε-rules until none is left. The

next event is then obtained by calling a subroutine within the event response loop. The value returned from this subroutine identifies the meneme selected and all regular rules matching the event and current meneme states are marked and subsequently fired. The event response loop is then repeated.

The MERS system is not applicable to dynamic interfaces involving object instantiation, and cannot handle the enabling and disabling of menemes in response to interactions. In the latter connection further development involving a second binary flag is needed. Performance is seen as an issue in larger and more complex interfaces, as a result of the problems of recognising an event and identifying the appropriate meneme.

Dialogue activation language (DAL)

Anderson & Apperley (1991) propose a direct manipulation dialogue model based on a hierarchy of dialogue cells, and the notion of 'dialogue activation' taken from a user action viewpoint. An active dialogue is one with which the user is able to directly interact. A dialogue cell becomes active only if its parent dialogue cell is active, and it receives one of a set of possible activating events. A second set of deactivating events can also exist. In this way a dialogue can be specified in terms of both a user's actions and the sequences in which those actions might be carried out. Each dialogue cell has an associated behaviour in addition to its role of restricting access to dialogue cells further down the hierarchy, and is referred to as a micro-dialogue or 'μdialogue'.

Based on these concepts of dialogue activation, a notation called Dialogue Activation Language (DAL) has been developed (Anderson, 1993). DAL consists of three components:

1. User actions and the context of these actions are described using a variation of User Action Notation (UAN) (Siochi & Hartson, 1989), a task-oriented representation of user action sequences in relation to screen objects for asynchronous interaction. The user actions for a task are represented by a sequence of symbols, and can be annotated to describe feedback and semantic connections. Two restrictions are introduced within DAL: that a μdialogue can have at most one display object (widget) associated with it; and that a user action event can only specify a context local to the μdialogue within which it is specified. Since by definition any ancestors of a μdialogue must be active, the local context of a μdialogue includes any of its ancestor contexts.
2. Valid user action sequences are specified by arranging μdialogues hierarchically according to the notion of dialogue activation. Within the body of a μdialogue, new system events can be generated in response to user or system events received.
3. Concurrent threads within a dialogue are supported by superimposing sequence hierarchies on a visual framework based on Lean Cuisine. In DAL, the nodes in a

tree diagram are μdialogues rather than menemes. Mutually compatible groups of μdialogues can all be active together, and action events which arrive at their parent μdialogue are broadcast to all of them. In a mutually exclusive grouping only one μdialogue can be active at any time, and the activation of a μdialogue in the group will cause any other active μdialogue in the group to deactivate.

Other features of DAL include variables, which can be used to pass information between μdialogues; guard conditions which determine whether or not a μdialogue is available for activation; and a system event broadcasting mechanism. In the spirit of object orientation, DAL also supports the definition of μdialogue templates or classes that define a generic behaviour and appearance, with inheritance of parent class behaviour by derived (sub)classes. The creation and deletion of μdialogue instances in response to events is also supported, thus providing for the specification of dynamic interfaces. In order to prove the suitability of DAL for dialogue design, a UIMS based around the language and associated methodology has been constructed. To support this, DAL includes a presentation attachment, which specifies the relationship with the display and provides for the definition of attributes of widgets, and an application attachment, which specifies the relationship with the semantics of the application through a function linkage. Consistency checking is provided in order to prevent the specification of dialogues in which subdialogues are unreachable or non-deterministic.

Table 8.3 matches DAL features with those of Lean Cuisine+. There is close correspondence between the two notations, with the following exceptions:

- *Modal subdialogues:* In DAL, modality is supported only at the top level of a μdialogue hierarchy because the ancestors of a current subdialogue cannot be deactivated, whereas in Lean Cuisine+ it is supported at any level. This issue could be resolved through the addition of a control layer to the DAL environment, which provided for the suspension or 'locking' of other active dialogues.
- *System events:* In Lean Cuisine+, *conditional* system events are modelled directly via conditions attached to selection triggers. In DAL, an incoming conditional event associated with a particular μdialogue is handled by a nested guarded μdialogue which is activated by the event only if the guard condition is true.

Notwithstanding the mismatch on modality, and assuming an extension to accommodate nested μdialogues within class definitions to handle conditional events, Figure 8.11 shows an incomplete but reasonably detailed DAL representation of the Lean Cuisine+ specification of the Finder dialogue, as exemplified by the icon/window system of Figure 8.6. Explanatory notes follow the example, and should be read in conjunction with it.

Feature	Lean Cuisine+	DAL
subdialogue hierarchy	hierarchy of subdialogue trees	hierarchy of µdialogues
subdialogue activation and deactivation	subdialogue header selection & deselection events	µdialogue activation & deactivation events
concurrency	mutually compatible subdialogues	mutually compatible µdialogues
modal subdialogue	supported at any level in the subdialogue hierarchy	supported only at the top level in the µdialogue hierarchy
object	meneme	µdialogue
generic object	fork symbol plus meneme	µdialogue template
object instance	generic object instance	µdialogue instance based on template
object subtype	meneme plus subtype link	derived µdialogue template
object attribute	object state variable	attribute
object creation and deletion	selection trigger	New & Delete statements
system event	selection trigger (scope explicit) - possibly conditional	generated event received by another (possibly guarded) µdialogue
operation	meneme	µdialogue
subdialogue availability precondition	option precondition	µdialogue guard condition
variable	state variable	variable
existence dependency	explicit existence dependency	implicit dependency via µdialogue hierarchy
action sequence	explicit action sequence in task layer	explicit action sequence via µdialogue hierarchy
other constraints	meneme designators	on activation/deactivation statements & guard conditions

Table 8.3: Feature analysis of Lean Cuisine+ and DAL

```
attribute
    pfolder
        iname         S,
        iselstat      S,
        ifostat       S,
        iposition     DD,
        favstat       S,
        fparent       S,
        //other attribute types...//
    ;

    ptrash
        iname         S,
        //other attribute types...//
    ;

    pwindow
        wname         S,
        wavstat       S,
        wselstat      S,
        wposition     DD,
        wwidth        D,
        wdepth        D,
        wdwidth       D,
        wddepth       D,
        //other attribute types...//
    ;
    //other widgets...//


class
    foldericon ([ ]Mv, [ ]MvM^,
        select_match_folder,                     //<= foldericon 1//
        do_close)                                //<= foldericon 2//

        ([ ]Mv) {
        event hilite, dehilite;
        on activation {
            hilite;
            iselstat = s;
        }
        on deactivation {
            dehilite;
            iselstat = u;
        }
        do_open <- open_selected;                //=> window 3//
    }

    trashfolder : foldericon ( ) ( ) {
        event movetrash;
        display type ptrash;
        movetrash [iposition] <- cursor_moved;
    }
```

Figure 8.11a: Mapping Lean Cuisine+ to DAL

```
folder : foldericon |favstat = a|                                    //--P5//
        ( ) (do_nest, do_trash) {
        event  do_insertchar, grab, drop, do_nest;
        display type  pfolder;
        new  folder [iposition] <- do_newfolder;                    //<= folder 1//
        delete  folder <-  do_empty;                                //<= folder 2//
        on new  {
                create mfolder, nfolder, tfolder;
                iname = #;
                iselstat = s;
                ifostat = c;
                favstat = a;
                fparent = activewname;
                //set other initial atttribute values//
        }
        do_insertchar [character] <- char_typed;
        choose_parent <- [ ]Mv | MvM^;                              //=> window 4//
        grab <- [ ]Mv;
        Grouping ME;
        mfolder (grab) (drop) {
                event  movefolder;
                movefolder [iposition] <- cursor_moved;
                drop <- [ ]M^;
        }
        nfolder |if closed recipient folder indicated|
                (drop) ( ) {
                do_nest;                                            //nest this folder//
        }
        tfolder |if closed recipient folder indicated AND recipient is Trash|
                (drop) ( ) {
                do_trash;                   //trash this folder and => window 5//
        }
}

window |ifostat = o|                                                 //--P6//
        ([ ]Mv, [ ]MvM^, do_open, choose_parent,
                do_trash, next_wind)
        (do_close) {
        event  show_active, show_inactive, move_window,
                size_window, wdir, close_it;
        display type  pwindow;
        on deactivation {
                show_inactive;
                wselstat = i;
        }
        new  [wposition] window <- do_newfolder;                    //<= window 1//
        delete  window <-  do_empty;                                //<= window 2//
        on new {
                create wdirect, wopen, wparent, wclose, wnext,
                        wscrollhoriz, wscrollvert;
                wname = #;
                wavstat = c;
                wselstat = i;
                //set other initial attribute values//
        }
```

Figure 8.11b: Mapping Lean Cuisine+ to DAL - continued

```
        wdir <- [ ]Mv | [ ]MvM^;
        close_it <- close_selected;
        move_window;                                          //detail undefined//
        size_window;                                          //detail undefined//

        Grouping ME;
        wdirect (wdir) ( ) {
                show_active;
                enable_scroll;
                wselstat = a;
                activewname = wname;
                select_match_folder;                          //=> foldericon 1//
        }
        wopen (do_open) ( ) {                                 //<= window 3//
                show_active;
                enable_scroll;
                wselstat = a;
                wavstat = o;
                //update windstack, and favstat for each child folder//
                activewname = wname;
        }
        wparent |if child folder is within window|
                (choose_parent) ( ) {                         //<= window 4//
                show_active;
                enable_scroll;
                wselstat = a;
                activewname = wname;
        }
        wclose (close_it, do_trash) ( ) {                     //<= window 5//
                wselstat = i;
                wavstat = c;
                //update windstack, and favstat for each child folder//
                next_wind;
                do_close;                                     //=> foldericon 2//
        }
        wnext (next_wind) ( ) {
                show_active;
                enable_scroll;
                wavstat = a;
                activewname = wname;
        }
        wscrollhoriz |wdwidth > wwidth|                          //--P3//
                (enable_scroll) ( ) {
                event do_scrollhoriz;
                do_scrollhoriz;                               //detail undefined//
        }
        wscrollvert |wddepth > wdepth|                           //--P4//
                (enable_scroll) ( ) {
                event do_scrollvert;
                do_scrollvert;                                //detail undefined//
        }
}
//other classes//
```

Figure 8.11c: Mapping Lean Cuisine+ to DAL - continued

```
system
    finder ( )( ) {
        var windstack, trashempty, activewname;
        event do_newfolder, do_empty, do_printdirectory,
            select_match_folder, do_close, choose_parent,
            do_open, do_trash;
        on activation {
            trashempty = TRUE;
            //initialise windstack//
            //perform other startup tasks, set defaults//
        }
        on deactivation {
            //shutdown tasks//
        }


        document ( )( ) {
            grouping ME;
            newfolder |windstack > 0 and activewname not = T|          //--P1//
                ( ) ( ) {
                do_newfolder <- newfolder_selected;      //=> folder 1
                                                          => window 1//
            }
            emptytrash |NOT trashempty| ( ) ( ) {                      //--P2//
                do_empty <- emptytrash_selected;         //=> folder 2
                                                          => window 2//
            }
            printdirectory ( )( ) {
                do_printdirectory <- printdirectory_selected;
            }
            //other Finder facilities//
        }


        icon_group ( )( ) {
            grouping ME;
            A_icon : folder ( )( ){
                display type  pfolder,
                    iname  A,
                    iselstat s,
                    ifostat c,
                    iposition 560 100,
                    favstat a,
                    fparent,
                    //other attribute values....//;
            }
            B_icon : folder ( )( ) {
                display type  pfolder,
                    iname  B,
                    //other attribute values....//;
            }
            //other icon instantiations....//
        }
```

Figure 8.11d: Mapping Lean Cuisine+ to DAL - continued

```
            window_group ( )( ) {
                    grouping ME;
                    A_window : window ( )( ) {
                            display type  pwindow,
                                    wname  A,
                                    wavstat c,
                                    wselstat i,
                                    wposition 10 350,
                                    wwidth 200,
                                    wdepth 150,
                                    wdwidth 230,
                                    wddepth 3000,
                                    //other attribute values....//;
                    }
                    B_window : window ( )( ) {
                            display type  pwindow,
                                    wname  B,
                                    //other attribute values....//;
                    }
                    //other window instantiations....//
            }
    }
```

Figure 8.11e: Mapping Lean Cuisine+ to DAL - continued

The following notes highlight significant aspects of the DAL representation shown in Figure 8.11, with reference to Lean Cuisine+ where appropriate:

- The *attribute* section defines important attributes types for the *folder*, *trashfolder*, and *window* interface objects. Each is defined as being of either type string (S) or type digit (D).
- The *class* section defines µdialogue templates, including two µdialogue types, *foldericon* and *window*, and two derived types, *folder* and *trashfolder*, based on *foldericon*. Mutually exclusive nested µdialogues are contained within both the *folder* and *window* classes. Lists of possible activating and deactivating events associated with each µdialogue appear in parentheses following the µdialogue name, and may be preceded by a guard condition, shown thus: |condition|. Events local to each µdialogue are listed via the *event* statement. Events and other statements associated with the instantiation of µdialogue instances are handled via *new* and *on new* statements; those concerned with the deletion of µdialogue instances via the *delete* statement; and those with the activation and deactivation of µdialogue instances via *on activation* and on *deactivation* statements. Move related folder operations are defined in detail, whereas window related operations associated with moving, sizing, and scrolling have been left undefined.

- The *system* section defines the structure of the *finder* dialogue in terms of three mutually compatible μdialogues: *document, icon_group* and *window_group*, which in turn comprise mutually exclusive groupings of further μdialogues. Only sample entries are shown for the folder icon and window instances. Global variables and events with global scope are defined.
- System events can be generated in response to received events. Thus for example, the statement *do_empty <- emptytrash_selected;* generates the system event *do_empty* in response to the user generated event *emptytrash_selected* (selection of the *EmptyTrash* option in the Finder dialogue).
- Events associated with the implementation of Lean Cuisine+ selection triggers are annotated and numbered (//...//). Each trigger involves both the export (annotated =>) and the import (annotated <=) of an event, the latter being handled by a nested guarded μdialogue where the trigger is conditional. Thus for example, the event *choose_parent* is exported from the *foldericon* μdialogue, and imported by the *window* μdialogue, where it is handled by the nested *wparent* μdialogue only if the guard condition is true, i.e. if the folder instance which exported the event is located within its parent window.
- Guard conditions associated with the implementation of Lean Cuisine+ option preconditions are matched through annotation with the precondition numbers of the Lean Cuisine+ Finder specification, e.g., //--P2//.
- Some of the events would trigger the execution of system procedures, associated for example with presentation issues. Thus for example, the event *movefolder* would trigger a procedure which highlighted potential recipient folders as an icon was dragged across the desktop.
- Some state variable changes are shown explicitly, e.g, *iselstat* = *s*. Others are hidden within as yet undefined procedures triggered by events, e.g. the setting of *trashempty* to *TRUE* in a procedure triggered by the event *do_empty.*
- Some licence has been taken in the definition of gnard conditions, but the intentions have been made clear.
- Not all required user actions (expressed in UAN) have been included. The emphasis throughout is on *structural* aspects of the mapping between the two notations, and on event handling.

The event sequence associated with the simulation defined in Table 8.2, using the values in Table 8.1 as a starting point, has been successfully taken through the above DAL dialogue representation in a dry run.

The example points up the closeness of the match between the Lean Cuisine+ and DAL models, and demonstrates that with further development DAL could offer a route to the prototyping and implementation of interfaces designed using Lean Cuisine+.

Part 3

Review

Chapter 9 Conclusions and Further Work 185

Chapter 9

Conclusions and Further Work

In this chapter the research reported in this thesis is reviewed, and further research is identified. Section 9.1 reviews the Lean Cuisine+ notation in the light of the objectives defined in Section 1.2, and Section 9.2 examines the contribution of the thesis in a broader context. Section 9.3 contains some general conclusions, while in Section 9.4 further work is identified.

9.1 Review of the Lean Cuisine+ notation

So called 'event-based' dialogue has to date lacked a suitably expressive visual model. The research reported in this thesis has involved the development of a semi-formal graphical notation, Lean Cuisine+, for describing the underlying behaviour of event-based asynchronous direct manipulation interfaces, and has explored the use of the notation both in 'reverse engineering' and during the high level design phase of the interface development life cycle. To this extent the principle objective of the research as set out in Section 1.2 has been met.

Lean Cuisine+ is a multi-layered notation. The base layer is a tree diagram which captures part of the behaviour of a direct manipulation interface in terms of constraints and dependencies between selectable primitives (menemes). Associated with each meneme is an implicit event. User manipulation of a meneme can be restricted through the use of meneme modifiers. Further constraints and dependencies associated with the dynamics of the interface are captured through overlays, where events are explicitly shown. The orthogonal task layer goes further in defining sequences or related groups of events, and provides a task-oriented view of the interface. Lean Cuisine+ is able to combine both static and dynamic interface modelling in a coherent manner, thus avoiding the necessity of employing separate and possibly disjoint models in high level design.

A Lean Cuisine+ specification fits with the notion of a 'problem space', as defined by Newell (1980). A problem space consists of a set of symbolic structures (the states of the space) and a set of operators over the space. Each operator takes a state as input and produces a state as output, and may be defined as partial (i.e. not defined for all states). More than one problem space may be active, i.e. applying an operator in one problem space may require passing into another problem space. These characteristics apply equally to Lean Cuisine+ dialogue specifications.

The dialogue descriptions produced so far have shown Lean Cuisine+ to be capable of handling a range of direct manipulation interaction styles and a variety of interface objects. Although software architecture is not an issue at the early stages of dialogue design, it is interesting to note that the two dialogue levels identified in the Microsoft Word and MacDraw dialogues of Chapter 6 correspond with two of the software abstraction levels defined by Bass & Coutaz (1991), which represent windows and interaction objects respectively. In the context of the 'Seeheim' model (Green, 1985a), which was considered in Section 2.5, the Lean Cuisine+ dialogue model provides for description of the 'dialogue control' component.

Secondary objectives for the research were also set down in Section 1.2:

- That visually, the notation should:
  - have a small number of simple concepts;
  - be suitably expressive, and in particular reflect the object oriented nature of direct manipulation interfaces;
  - be compact, and maintain locality between dialogue elements.
- That the notation should be suited both to the analysis of existing interfaces, and to the high level synthesis of component dialogues for new interfaces.
- That the notation should be executable, and thereby support a limited form of early prototyping of interfaces.
- That the notation should be mappable into a formal dialogue specification language, and thereby support interface prototyping and implementation.

Each of these objectives is examined below:

Visual aspects of the notation

In extending Lean Cuisine a number of objectives for visual notations have been pursued, including those of Chakravarty & Kleyn (1990) which were described in Section 3.3. Important amongst these are simplicity, expressiveness, compactness and locality. These objectives have largely been met by the notation:

Simplicity

As originally defined, Lean Cuisine was a simple notation based on hierarchical groupings of selectable representations. The simplicity and elegance of the Lean Cuisine menu notation has been retained in the base layer Lean Cuisine+ diagram, which remains a hierarchical structure based on a small number of simple concepts. The more complex dynamic aspects of interface behaviour, which cut across the hierarchy, are captured through additional and separate layers which are superimposed on the base layer diagram. Through consistent usage of constructs across the different layers, the complexity of the notation has been minimised.

Expressiveness

The base layer Lean Cuisine+ tree diagram can handle both hierarchy and orthogonality, and supports stepwise refinement. Objects are clearly delineated, thus reflecting the underlying nature of direct manipulation interfaces. The addition of further constraint layers has produced a richer but more complex notation. Complexity has been managed through the discrete layer structure. The trade-off between expressiveness and simplicity has been carefully weighed at each stage of development of the notation.

Compactness

Lean Cuisine+ is an economical, compact and concise notation when matched against some other graphical dialogue notations which have been applied to direct manipulation interfaces e.g. state transition diagrams (Jacob, 1986), and event decomposition graphs (Chakravarty & Kleyn, 1990), both of which were reviewed in Section 3.2.

Local Adjacency

In Lean Cuisine+ task actions are grouped around 'syntactic' or 'semantic' *objects*. This seems a natural grouping in the context of direct manipulation interfaces, which are squarely object-based. An alternative approach is to group actions around states. Both statecharts (Harel, 1988) and Petri nets (Peterson, 1977), which were examined in Section 3.2, offer such an approach. However, objects per se are not represented in either of these notations, but exist as the sum of their states, which adds considerable visual complexity.

Applicability of the notation

Lean Cuisine+ has been developed through the description of the behaviour of aspects of an existing direct manipulation interface, that of the Apple Macintosh. This represents an *analytical* approach - the application of the notation in so-called 'reverse engineering'. In describing an existing interface, inconsistencies or quirks relating more to presentation than underlying functionality may impact the description. This does not invalidate reverse engineering as a means of developing a notation, but rather points up side-effects of adopting the approach. On the positive side this should enable Lean Cuisine+ to point up inconsistencies present in existing interfaces.

By contrast, in Chapter 7 Lean Cuisine+ has been applied in the synthesis of component dialogues in the early stages of the design of new interfaces. This is further considered in Section 9.2 below.

Executability of the notation

The Lean Cuisine+ notation has been shown to be executable. The designer running a simulation in Lean Cuisine+ acts as the environment, choosing which event

will happen next and observing the results of that choice on meneme states in the base layer tree diagram(s), and on state variables. This can be considered as a valid if limited form of prototyping in which the dynamic behaviour of an interface can be examined step-by-step and the simulation stopped and restarted at any point. Partially completed specifications can be exercised in this way. Execution of a Lean Cuisine+ specification leads to a succession of permissible interface states, with full cognisance being taken of constraints, including those in the constraint layers. By executing the specification as the prototype, the designer (and the user) can receive early feedback as to its acceptability. Execution at this stage also provides for early validation of the design.

Simulation of interface behaviour through execution of the Lean Cuisine+ notation works least well in connection with spatial aspects of the interaction, for example the moving of objects, where execution offers a somewhat artificial 'arms length' approach. Simulation of spatial aspects would be better approached through production of a prototype interface.

Mappability of the notation for prototyping and implementation

The notion of mapping from Lean Cuisine+ to DAL (Anderson, 1993) has been examined, and an outline mapping developed for an example interface. The two notations match closely in most respects, and with further development, including the addition of a control layer to handle modality, DAL could provide a route to the development of a software environment to support prototyping of interfaces specified in Lean Cuisine+. This approach could support manipulation of both the Lean Cuisine+ specification and the associated prototype interface in tandem in separate windows (as described in Section 8.3).

9.2 The wider contribution of the thesis

In the context of systems requirements specifications, Avison & Wood-Harper (1991) state that "all information systems development methodologies have limitations....and none is all-inclusive. The methodologies address a moving target in that the technology, along with the techniques and tools supporting it, develops relentlessly." This is equally applicable to techniques and tools to support the design of interfaces. In this section the work undertaken in this thesis is reviewed from a wider perspective.

Support for the early design phase of the IDLC

Reference was made in Section 1.1 to the interface development life cycle (IDLC) as defined by Shneiderman (1987). The first three stages of Shneiderman's eight stage life cycle are: collection of information; definition of requirements and semantics; and design of syntax and support facilities. User and task models are applicable to the first

two stages, which are concerned with the definition of functional requirements. Lean Cuisine+ is relevant to stage 3, the early design phase, which is concerned with structural requirements and with specifying the underlying behaviour of the interface.

A parallel can be drawn with software development, in which connection Lane (1990) distinguishes functional and structural design spaces. *Functional* design space, which represents the results of requirements analysis, matches with stage 2 of Shneiderman's interface life cycle, and with user and task analysis. On the other hand the dimensions that describe structural choices make up the *structural* design space, which represents the results of initial system decomposition. This matches stage 3 of Shneiderman's life cycle, and with the use of notations such as Lean Cuisine+ for specifying underlying system behaviour. Structural choices are made after the functional requirements of a system have been defined, and therefore after some design decisions have been made, for example regarding the basic kind of interaction to be supported.

The Lean Cuisine+ task layer defined in Section 7.3 captures temporal relationships between primitive task actions, and provides for a mapping between task and dialogue models, there being a relationship between the objects of the two models and the actions in which they participate. This mapping was used as the basis for the five stage methodology developed in Section 7.4 for the construction of Lean Cuisine+ specifications for new interfaces, to support the early design phase of the IDLC.

Definition of a direct manipulation behavioural model

Three views relevant to the development of direct manipulation interfaces, the 'conceptual', 'behavioural' and 'presentational' views, were constructed in Section 2.4 and related to other frameworks. The required scope of the behavioural model was subsequently established through the analysis of dialogue notations in Section 3.2, and set out in detail in terms of objects, user and system events, states, and relationships. This model provided the basis for the extensions to Lean Cuisine reported in this thesis and could serve as a template for future developments in the area of direct manipulation models as the technology develops.

Analysis of Macintosh interaction tasks

Section 4.1 reviewed previous attempts to classify interaction tasks, examined them in the context of the Macintosh interface, and proposed a taxonomy of direct manipulation interaction tasks. Particular attention was paid to tasks involving repeated actions, including dragging operations and the manipulation of text, in which connection a new basic 'character' task was identified. The analysis showed quite clearly that at the most primitive level all tasks reduce to one or more 'selection' actions.

This analysis is not only an important research contribution in its own right, but also has general implications for the types of tools and techniques needed to describe and implement direct manipulation interfaces. In the context of this thesis it lends support to

the proposition of Apperley & Spence (1989) that more general direct manipulation interactions such as positioning and dragging could also be regarded as 'selection', and could therefore be mapped into the Lean Cuisine meneme model.

9.3 General conclusions

In addition to the specific contributions discussed above, some more general conclusions can be drawn from this work:

- The research supports the view (Moran, 1981; Green, 1986; Gehani & McGettrick, 1986; Carey & Graham, 1987) that 'semi-formal' notations are appropriate at the early stage of design.
- The research confirms the view (Harel, 1988; Chakravarty & Kleyn, 1990; Tse & Pong, 1991) that graphical notations supporting visual modes of reasoning are effective at the higher levels of the design of complex systems.
- The research confirms the view (Burns & Lister, 1991) that object-oriented modelling is appropriate at the early stages of design because objects can be used to provide traceability through all stages of the design process to implementation and execution. The object-oriented aspects of Lean Cuisine+ render the notation particularly relevant to direct manipulation interfaces, because of their event-based nature.
- The research supports the notion (Edmondson & Spence, 1992) that to be most effective notations should be targeted at a specific category of interaction. The underlying meneme model of Lean Cuisine+ matches closely with the selection-based nature of direct manipulation interfaces.
- The research supports the notion (Kieras & Polson, 1983; Smith, 1986; Davis, 1988b; Apperley & Spence, 1989; Anderson & Apperley, 1991; Edmondson, 1991) that the description of the underlying behaviour of an interface should not be constrained by the details of its implementation. Lean Cuisine+ has been targeted at the 'behavioural' level.
- The research supports the view that new notations should capture more than just the 'surface' nature of human-computer interaction, and that they should be capable of associating lower level activity with the functionality of an interface. The task layer of Lean Cuisine+ provides a link between higher level functionality, and objects, primitive task actions, and states.
- The research demonstrates the advantages of a notation which can be executed to provide early simulation of the dynamic behaviour of the system under design.
- The research supports the view (Harel *et al*, 1988; Harbert *et al*, 1990) that a multi-notational approach to interface development is required, and that it should be

possible to move easily from initial specification to prototyping and implementation. The mapping from Lean Cuisine+ to DAL shows promise in this respect.

9.4 Further work

Over the period of the reported research, a number of areas for further related research have been identified:

- The application of Lean Cuisine+ to the description of the behaviour of interfaces produced via interface generators (such as HyperCard or Prototyper on the Macintosh) which provide no separate description of the behaviour of the interface constructed (it exists and its behaviour can be uncovered through exploration). In general these tools support the construction of interfaces from objects with predefined behaviours and links between these objects, which suggests that Lean Cuisine+ should be an appropriate graphical notation for describing interface behaviour, with the further possibility of Lean Cuisine+ descriptions being generated automatically. This is currently under exploration at Massey University as an honours level project.
- The development of the prototyping environment described in Section 8.3, involving mapping from Lean Cuisine+ into an extended DAL. Cooperative research is planned in this area.
- Extension of this environment to become a full UIDE, supporting verification of the design at a more formal level and qualitative analysis of the interface under development in areas such as consistency and completeness.
- The extension of the Lean Cuisine+ notation to handle existence dependencies of a more complex nature. In the Microsoft Word dialogue of Section 6.2, for example, 'splitting' a window, would create a 1:N dependency between the *Folder* and *Window* objects.
- The addition of attribute dependency graphs (Barth, 1986) to the Lean Cuisine+ support environment. Attribute dependencies provide for the linking of attributes such that when an attribute is updated, dependent attributes are updated via dependencies specified in a graphical dependency lattice.
- The application of the Lean Cuisine+ notation to other manipulation-based interfaces, for example those involving domain specific interaction, as typified by flight simulators.
- The undertaking of a fuller exploration of the benefits of producing a behavioural specification unconstrained by presentational issues, including the notion of developing multiple versions of an interface designed using Lean Cuisine+, for example to assist handicapped users.

References and Glossary

References 195
Glossary 205

References

Alexander, H. (1987): Executable Specifications as an aid to Dialogue Design, *Interact '87 Proc.*, Elsevier, Amsterdam, IFIP, 739-744.

Alexander, H. (1990): Structuring Dialogues using CSP, in *Formal Methods in Human-Computer Interaction*, Harrison, M. & Thimbleby, H. (eds), CUP, Cambridge, England, 273-295.

Anderson, P.S. (1993): *Dialogue Activation: An Approach to User Centred Constructional Modelling of Direct Manipulation Interfaces*, PhD thesis, to be published, Massey University, New Zealand.

Anderson, P.S. & Apperley M.D. (1990): An interface prototyping system based on Lean Cuisine, *Interact. with Comput.*, **2**, 2, 217-226.

Anderson P.S. & Apperley M.D. (1991): *Dialogue Activation: A Design Approach for Direct Manipulation Interfaces*, Inf. Sci. Rep. 91/2, Massey University, New Zealand.

Annett, J., Duncan, J.D., Stammers, R.B. & Gray, M.J. (1971): *Task Analysis*, Training Information Number 6, HMSO, London.

Anson, E. (1986): The device model of interaction, *Siggraph '86 Proc.*, in *Comput. Graph.*, **20**, 4, 107-114.

Apperley, M.D. (1988): *TaP: A Menu Interface Design Study Using the Lean Cuisine Notation*, Inf. Eng. Rep. #88/2, Imperial College, London.

Apperley, M.D. & Spence, R. (1989): Lean Cuisine: A Low-Fat Notation for Menus, *Interact. with Comput.*, **1**, 1, 43-68.

Apple (1983): *MacPaint*, Apple Computer Inc, Cupertino, CA.

Apple (1989a): *Macintosh System Software Users Guide*, Apple Computer Inc, Cupertino, CA.

Apple (1989b): *HyperCard Users Guide*, Apple Computer Inc, Cupertino, CA.

Avison, D.E. & Wood-Harper, A.T. (1991): Information Systems Development Research: An Exploration of Ideas in Practice, *The Comput. J.*, **34**, 2, 98-111.

Baecker, R.M. (1980): Towards an Effective Characterisation of Graphical Interaction, In *Methodology of Interaction*, Guedj, R.A. *et al* (eds), Elsevier, Amsterdam, 127-147.

Baecker, R.M. & Buxton, W.A. (1987): *Readings in Human-Computer Interaction - A Multi-disciplinary Approach*, Morgan Kaufmann, California.

Barth, P.S. (1986): An Object-oriented Approach to Graphical Interfaces, *ACM Trans. Graph.*, **5**, 2, 142-172.

Bass, L. & Coutaz, J. (1991): *Developing Software for the User Interface*, Addison-Wesley, Reading, Mass.

Bass, L., Hardy, E., Hoyt, R., Little, R. & Seacord, R. (1988): *The Serpent run time architecture and dialogue model*, Carnegie Mellon Tech. Rep., CMU/SEI-88-TR-6.

Benbasat, I. & Wand, I. (1984): A structured approach to designing human-computer dialogues, *Int. J. Man-Mach. Stud.*, **21**, 105-126.

Bewley, W.L., Roberts, T.L., Schroit, D. & Verplank, W.L. (1983): Human Factors Testing in the Design of Xeroxs' 8010 "Star" Office Workstation, *CHI '83 Proc.,* ACM, New York, 72-77.

Bleser, T.W. & Foley, J.D. (1982): Towards specifying and evaluating the human factors of user-computer interfaces, *Proc. Conf. on Human Factors in Comput. Sys.*, ACM, New York, 309-314.

Bobrow, D.G., Mittal, S. & Stefik, M.J. (1986): Expert Systems: Peril and Promise, *Commun. ACM*, Sept. 1986, 880-894.

Borufka, H.G., Kuhlmann, H.W. & Ten Hagen, P.J.W. (1982): Dialogue Cells: A method for defining interactions, *IEEE Comput. Graph. and Applic.*, **2**, 5, 25-33.

Burns, A. & Lister, A.M. (1991): A Framework for Building Dependable Systems, *The Comput. J.,* **34**, 2, 173-181.

Buxton, W.A., Lamb, M.R., Sherman, D. & Smith, K.C. (1983): Towards a comprehensive user interface management system, *Comput. Graph.*, **17**, 3, 35-42.

Cadre (1990): *Teamwork User Guide: Release 4.0*, Cadre Technologies Inc., Rhode Island, USA.

Card, S.K. & Henderson, A. (1987): A Multiple, Virtual-Workspace Interface to Support User Task Switching, *CHI + GI '87 Proc.*, ACM, New York, 53-59.

Card, S.K., Moran, T.P. & Newell, A. (1983): *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Assoc., New Jersey.

Card, S.K., Pavel, M. & Farrell, J.E. (1984): Window-based Computer Dialogues, *Interact '84 Proc.,* Elsevier, Amsterdam, 239-243.

Cardelli, L. and Pike, R. (1985): Squeak: a language for communicating with mice, *Comput. Graph.,* **19**, 3, 205-213.

Carey, T.T. & Graham, C.H. (1987): Modular Specification Methods for User Interfaces, *Interact '87 Proc.,* 403-408.

Chakravarty, I. & Kleyn, M.F. (1990): Visualisms for Describing Interactive Systems, in *Engineering for Human-Computer Interaction*, Cockton, G. (ed), Elsevier, Amsterdam, IFIP, 333-356.

Chen, P.P. (1976): The Entity-Relationship Model - Towards a Unified View of Data, *ACM Trans. Data Base Sys.,* **1**, 1, 9-36.

Chin, D. (1989): KNOME: Modelling what the user knows in UC, in *User Models in Dialog Systems*, Kobska, A. & Wahlster, W. (eds), Springer-Verlag, Berlin, 74-107.

Claris (1988): *MacDraw II*, Claris Corp., Mountain View, CA.

Cockton, G. (1987): Interaction ergonomics, control and separation: open problems in user interface management, *Inf. and Softw. Tech.*, **29**, 4, 176-191.

Cockton, G. (1990a): Lean Cuisine: no sauces, no courses!, *Interact. with Comput.*, **2**, 2, 205-216.

Cockton, G. (ed) (1990b): *Engineering for Human-Computer Interaction*, Elsevier, Amsterdam, IFIP, 3-8.

Coleman, D., Hayes, F. & Bear, S. (1992): Introducing Objectcharts or How to Use Statecharts in Object-oriented Design, *IEEE Trans. Softw. Eng.*, **18**, 1, 9-18.

Coutaz, J. (1987): PAC, an Object Oriented Model for Dialogue Design, *Interact '87 Proc.*, Elsevier, Amsterdam, IFIP, 431-436.

Coutaz, J. (1990): Architecture Models for Interactive Software: Failures and Trends, in *Engineering for Human-Computer Interaction*, Cockton, G. (ed), Elsevier, Amsterdam, IFIP, 137-151.

Dance, J.R., Granor, T.E., Hill, R.D., Hudson, S.E., Meads, J., Myers, B.A. & Schulert, A. (1987): The Run-time Structure of UIMS-supported Applications, *Siggraph '87 Proc.*, in *Comput. Graph.*, **21**, 2, 97-101.

Davis, A.M. (1988a): A Comparison of Techniques for the Specification of External System Behaviour, *Commun. ACM*, **31**, 9, 1098-1115.

Davis, A.M. (1988b): A taxonomy for the early stages of the software development life cycle, *J. Syst. Soft.*, **8**, 4, 297-311.

DeMarco, T. (1978): *Structured Analysis and System Specification*, Prentice Hall, New Jersey.

Docker, T.W.G. (1989): *SAME: Structured Analysis Modelling Environment*, PhD thesis, Massey University, New Zealand.

Edmondson, W.H. (1991): Interaction Taxonomy, Presented at *The Second Venaco Workshop on the Structure of Multimodal Dialogue*, Acquafredda di Maratea, Italy, Sept.

Edmondson, W.H. & Spence, R. (1992): Systematic Menu Design, in *Proc. HCI '92*, Monk, A., Draper, D. & Harrison, M. (eds), CUP, Cambridge, England, 209-226.

Farooq, M.U. & Dominick, W.D. (1988): Survey of formal tools and models for developing user interfaces, *Int. J. Man-Mach. Stud.*, **29**, 479-496.

Flecchia M.A., & Bergeron, R.D. (1987): Specifying Complex Dialogues in ALGAE, *CHI + GI '87 Proc.*, ACM, New York, 229-234.

Foley, J.D. (1987): Transformations on a Formal Specification of User-Computer Interfaces, *Siggraph '87 Proc.*, in *Comput. Graph.*, **21**, 2, 109-113.

Foley, J.D., Kim, W.C., Kovacevic, S. & Murray, K. (1989): Defining Interfaces at a High Level of Abstraction, *IEEE Softw.,* Jan., 25-32.

Foley, J.D. & Van Dam, A. (1984): *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA.

Foley, J.D. and Wallace V.L. (1974): The Art of Natural Graphic Man-machine Conversation, *Proc. IEEE,* **62**, 4, 462-471.

Foley, J.D., Wallace, V.L. and Chan, P. (1984): The human factors of computer graphics interaction techniques, *IEEE Comp. Graph. and Applic.,* **4**, 11, 13-48.

Gane, C. & Sarson, T. (1979): *Structured Systems Analysis: tools and techniques*, Prentice Hall, New Jersey.

Gehani, N. & McGettrick, A.D. (eds) (1986): *Software Specification Techniques*, Addison-Wesley, Wokingham, England.

Goldberg, A. & Robson, D. (1983): *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA.

Goodfellow, M.J. (1986): WHIM, the Window Handler and Input Manager, *IEEE Comp. Graph. and Applic.*, **6**, 5, 46-52.

Goodman, D. (1990): *The Complete HyperCard 2.0 Handbook*, Bantam Books Inc., New York.

Green, M. (1985a): Report on Dialogue Specification Tools, in *User Interface Management Systems,* G. Pfaff (ed), Springer-Verlag, Berlin, 9-20.

Green, M. (1985b): Design Notations and User Interface Management Systems, in *User Interface Management Systems,* G. Pfaff (ed), Springer-Verlag, Berlin, 89-107.

Green, M. (1985c): The University of Alberta UIMS, *Comput. Graph.,* **19**, 3, 205-213.

Green, M. (1986): A Survey of Three Dialogue Models, *ACM Trans. Graph.,* **5**, 3, 244-275.

Grotchmann, M. (1987): Windownet - A Formal Notation for Window-based User Interfaces, *Interact '87 proc.,* Elsevier, Amsterdam, 437-442.

Guindon, R. (1990): Designing the Design Process: Exploiting Opportunistic Thoughts, *HCI,* **5**, 305-344.

Harbert A., Lively W. & Sheppard S. (1990): A Graphical Specification System for User-Interface Design, *IEEE Softw.,* July, 12-19.

Harel, D. (1988): On Visual Formalisms, *Commun. ACM*, **31**, 5, 514-530.

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R. & Shutl-Trauring, A. (1988): Statemate: a Working Environment for the Development of Complex Reactive Systems, *Proc. Tenth Int. Conf. on Softw. Eng.*, Singapore, IEEE, April, 396-406.

Harel, D., Pnueli A., Schmidt J.P. & Sherman R. (1987): On the Formal Semantics of Statecharts, *IEEE Proc. 2nd Symp. on Logic in Comput. Sci.,* 54-64.

Harrison, M. & Dix, A. (1990): A State Model of Direct Manipulation in Interactive Systems, in *Formal Methods in Human-Computer Interaction*, Harrison, M. & Thimbleby, H. (eds), CUP, Cambridge, England, 129-151.

Hartson, H.R. (1989): User-Interface Management Control and Communication, *IEEE Softw.,* Jan., 62-70.

Hartson, H.R. & Hix D. (1989): Human-Computer Interface Development: Concepts and Systems for its Management, *ACM Comp. Surv.*, **21**, 1, 5-92.

Hartson, H.R., Siochi, A.C. & Hix, D. (1990): The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs, *ACM Trans. Inf. Sys.*, **8**, 3, 181-203.

Henderson, D.A. (1986): The Trillium User-Interface Design Environment, *CHI '86 Proc.*, ACM, New York, 221-227.

Hill, R.D. (1986): Supporting concurrency, communication, and synchronisation in HCI. The Sassafras UIMS, *ACM Trans. Graph.*, July, 179-210.

Hill, R.D. (1987): Event Response Systems - A Technique for Specifying Multi-threaded Dialogues, *CHI + GI '87 Proc.,* ACM, New York, 241-248.

Hix, D. and Hartson, H.R. (1986): An Interactive Environment for Dialogue Development: Its Design, Use, and Evaluation, *CHI '86 Proc.,* ACM, New York, 228-234.

Hoare, C.A.R. (1985): *Communicating Sequential Processes*, Prentice-Hall, New Jersey.

Hopgood, F.R.A. (1982): *GKS - the first graphics standard,* Sci. & Eng. Research Council Rep. RL-82-007, England.

Hopgood, F.R.A. & Duce, D.A. (1980): A Production System Approach to Interactive Graphic Program Design, in *Methodology of Interaction,* Guedj, R.A. *et al* (eds), Elsevier, Amsterdam, 247-263.

Hudson, S.E. (1987): UIMS Support for Direct Manipulation Interfaces, *Comput. Graph.,* **21**, 2, 120-124.

Hudson, S.E. & King, R. (1986): A Generator of Direct Manipulation Office Systems, *ACM Trans. Office Inf. Sys.*, **4**, 2, 132-163.

Hutchins, E.L., Hollan, J.D. & Norman, D.A. (1986): Direct Manipulation Interfaces in *User Centered System Design*, Norman, D.A. & Draper, S.W. (eds), Lawrence Erlbaum Assoc., Hillsdale, New Jersey, 118-123.

ISO (1983): *Information processing - Graphical Kernel System (GKS) - functional description*, Draft International Standard ISO/DIS 7942.

Jacob, R.J.K. (1983): Using Formal Specifications in the Design of a Human-Computer Interface, *Commun. ACM,* **26**, 4, 259-264.

Jacob, R.J.K. (1985): A State Transition Diagram Language for Visual Programming, *IEEE Comput.,* **18**, 51-59.

Jacob, R.J.K. (1986): A Specification Language for Direct Manipulation User Interfaces, *ACM Trans. Graph.*, **5**, 4, 283-317.

Johnson, P. (1992): *Human-Computer Interaction: Psychology, Task Analysis and Software Engineering*, McGraw -Hill, London.

Johnson, P., Diaper, D. & Long, J.B. (1984): Tasks, skills and knowledge, in *Interact '84 Proc.*, Shackel, B. (ed), Elsevier, Amsterdam, 499-503.

Johnson, P., Johnson, H., Waddington, R. & Shouls, A. (1988): Task related knowledge structures: analysis, modelling and application, in *People and Computers IV*, Jones, D.M. & Winder, R. (eds), CUP, Cambridge, England, 35-62.

Kelly, C. & Colgan, L. (1992): User Modelling and User Interface Design, in *Proc. HCI '92*, Monk, A., Draper, D. & Harrison, M. (eds), CUP, Cambridge, England.

Kieras, D. & Polson, P.G. (1983): A generalised Transition Network Representation for Interactive Systems, *CHI '83 Proc.*, ACM, Boston, 103-106.

Koiuvnen, M.R. & Mantyla, M. (1988): HutWindows: An Improved Architecture for a User Interface Management System", *IEEE Comput. Graph. & Applic.*, Jan., 43-52.

Kuntz, M. & Melchert, R. (1990): Pasta-3s Requirements, Design, and Implementation: A Case Study in Building a Large, Complex Direct Manipulation Interface, in *Engineering for Human-Computer Interaction*, Cockton, G. (ed), Elsevier, Amsterdam, IFIP, 43-60.

Lane, T.G. (1990): *Studying Software Architecture Through Design Space and Rules*, Carnegie Mellon Tech. Rep. CMU/SEI-90-TR-18, Pittsburgh, Pennsylvania.

Lewis, S. (1991): Cluster Analysis as a technique to guide interface design, *Int. J. Man-Mach. Stud.*, **35**, 251-265.

Marcus, A. & Mullet, K. (1990): *Graphical Human-Computer Interface Design for Window Management Systems*, CHI '90 Tut. 20, ACM, New York.

Markopoulos, P., Pycock, J., Wilson, S. & Johnson, P. (1992): Adept - A task based design environment, *Proc. Hawaii Int. Conf. on Sys. Sci.*, **2**, IEEE Comp. Soc. Press.

Microsoft (1991): *Microsoft Word*, Microsoft Corp., Redmond, WA.

Monarchi, D.E. & Gretchen, I.P. (1992): A Research Typology for Object-Oriented Analysis and Design, *Commun. ACM*, **35**, 9, 35-47.

Moran, T.P. (1981): The Command Language Grammar: a representation for the user interface of interactive computer systems, *Int. J. Man-Mach. Stud.*, **15**, 3-50.

Myers, B.A. (1987a): Gaining General Acceptance for UIMSs, *Siggraph '87 Proc.*, in *Comput. Graph.*, **21**, 2, 130-134.

Myers, B.A. (1987b): Creating Dynamic Interaction Techniques by Demonstration, *CHI + GI '87 Proc.*, ACM, 271-278.

Myers, B.A. (1988): A Taxonomy of Window Manager User Interfaces, *IEEE Comput. Graph. and Applic.*, **8**, 5, 65-84.

Myers, B.A. (1989): User-Interface Tools: Introduction and Survey, *IEEE Softw.*, Jan., 15-23.

Nelson, T. (1980): Interactive systems and the design of virtuality, *Creative Comput.*, **6**, 11, Nov., 56ff.

Newell, A. (1980): Reasoning, problem solving and decision processes: the problem space as a fundamental category, in *Attention and Performance VIII*, R.S.Nickerson (ed), Erlbaum, Hillsdale, New Jersey, 693-718.

Newman, W.M. (1968): A system for interactive graphical programming, *Proc SJCC AFIPS Conf.*, **32**, 47-54.

Nijssen, G.M. & Halpin, T.A. (1989): *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*, Prentice Hall, New Jersey.

Nixdorf, T. & Kiyooka, G. (1992): Substance & Style: GUI design and culture, *Comput. Lang.*, Feb., 43-58.

Olsen, D.R. (mod.) (1987a): Whither (or wither) UIMS, *CHI '87 Proc.*, ACM, New York, 311-314.

Olsen, D.R. (chair) (1987b): ACM Siggraph Workshop on Software Tools for User Interface Management, *Comput. Graph.*, **21**, 2, 71-78.

Olsen, D.R. (1987c): Larger Issues in User Interface Management, *Siggraph '87 Proc.*, in *Comput. Graph.*, **21**, 2, 134-137.

Olsen, D.R. (1990): Propositional Production Systems for Dialogue Description, *CHI '90 Proc.*, ACM, New York, 57-63.

Olsen, D.R. & Dempsey, E.P. (1983): Syngraph: A Graphic user interface generator, *Siggraph '83 Proc.*, in *Comput. Graph.*, **17**, 3, 43-50.

Olsen, D.R., Dempsey, E.P. & Rogge, R. (1985): Input/output Linkage in a user interface management system, *Siggraph '85 Proc.*, in *Comput. Graph.*, **19**, 3, 191-197.

Papert, S. (1980): *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books Inc., New York.

Parnas, D. (1969): On the use of transition diagrams in the design of a user interface for an interactive computer system, *Proc. ACM Nat. Conf.*, ACM, New York, 379-385.

Payne, S.J. and Green, T.R. (1986): Task-Action Grammars: A model of the mental representation of task languages, *HCI*, **2**, 93-133.

Peterson, J.L. (1977): Petri Nets, *ACM Comput. Surv.*, **9**, 3, 223-252.

Peterson, J.L. (1981): *Petri Net Theory and the Modeling of Systems*, Prentice Hall, New Jersey.

Phillips, C.H.E. (1991): Towards a Notation for Describing the Behaviour of Direct Manipulation Interfaces, *New Zealand J. Comput.*, **3**, 1, 11-25.

Phillips C.H.E. (1992): *Extending Lean Cuisine: A Case Study in Reverse Engineering*, Inf. Sci. Rep. 92/1, Massey University, New Zealand.

Phillips, C.H.E. & Apperley, M.D. (1991): Direct Manipulation Interaction Tasks: A Macintosh-based Analysis, *Interact. with Comput.*, **3**, 1, 9-26.

Reisner, P. (1981): Formal Grammar and Human Factor Design of an Interactive Graphics System, *IEEE Trans. Softw. Eng.*, SE-7, 229-240.

Reisner, P. (1982): Further developments towards using formal grammar as a design tool, *Proc. Conf. on Human Factors in Comput. Sys.*, ACM, New York, 309-314.

Rhyne, J.R., Ehrich, R., Bennett, J., Hewett, T., Sibert, J.L. & Bleser, T.W. (1987): Tools and Methodology for User Interface Development, *Comput. Graph.*, **21**, 2, 78-87.

Rhyne, J. & Watson, T.J. (1987): Dialogue Management for Gestural Interfaces, *Siggraph '87 Proc.*, in *Comput. Graph.*, **21**, 2, 137-142.

Rich, E. (1983): Users are individuals: individualizing user models, *Int J. Man-Mach. Stud.*, **18**, 199-214.

Rosenberg, J.K. (Mod.) (1988): UIMSs: Threat or Menace?, *CHI '88 Proc.*, ACM, New York, 197-200.

Rosenburg, J.K. and Moran, T.P. (1984): Generic Commands, *Interact '84 Proc.*, Elsevier, Amsterdam, IFIP, 245-249.

Rutkowski, C. (1982): An Introduction to the Human Applications Standard Computer Interface, Part1: Theory and Principles, *Byte*, **7**, 11, 291-310.

Shneiderman, B. (1982): Multiparty Grammars and related features for designing interactive systems, *IEEE Trans. on Sys. Mgmt. and Cyber.*, SMC-12, 148-154.

Shneiderman, B. (1983): Direct Manipulation: A Step Beyond Programming Languages, *IEEE Comput.*, Aug., 57-69.

Shneiderman, B. (1987): *Designing the User Interface*, Addison-Wesley, Reading, MA.

Sibert, J.L., Hurley, W.D. & Bleser, T.W. (1986): An Object-oriented User Interface Management System, *Siggraph '86 Proc.*, in *Comput. Graph.*, **20**, 4, 259-268.

Singh, G. & Green, M. (1991): Automating the Lexical and Syntactic Design of Graphical User Interfaces: The UofA UIMS, *ACM Trans. Graph.*, **10**, 3, 213-254.

Siochi, A.C. & Hartson, H.R. (1989): Task-oriented Representation of Asynchronous User Interfaces, *CHI '89 Proc.*, ACM, New York, 183-188.

Six, H.W. & Voss, J. (1990): DIWA: A Hierarchical Object-oriented Model for Dialog Design, in *Engineering for Human-Computer Interaction*, Cockton, G. (ed), Elsevier, Amsterdam, IFIP, 383-402.

Smith, C.D., Irby, C., Kimball, R., Verplank, B. & Harslem, E. (1982): Designing the Star User Interface, *Byte*, **7**, 4, 242-282.

Smith, S.L. (1986): Standards Versus Guidelines for Designing User Interface Software, *Behav. and Inf. Tech.*, **5**, 1, 47-61.

Sommerville, I. (1985): *Software Engineering*, Addison-Wesley, Wokingham, England.

Spence, R. & Apperley, M.D. (1977): The Interactive-Graphic Man-Computer Dialogue in Computer-Aided Circuit Design, *IEEE Trans. Circ. & Sys.*, **24**, 2, 49-61.

Sutcliffe, A.G. (1988): *Human-Computer Interface Design*, McMillan, London.

Sutcliffe, A.G. & Wang, I. (1991): Integrating Human Computer Interaction with Jackson System Development, *The Comput. J.*, **34**, 2, 132-142.

Tanner, P.T. (1987): Multi-Thread Input, *Siggraph '87 Proc.*, in *Comput. Graph.*, **21**, 2, 142-145.

Tesler, L. (1981): The Smalltalk environment, *Byte*, **6**, 6, 90-147.

Thimbleby, H. (1990): *User Interface Design*, Addison-Wesley, Wokingham, England.

Touretsky, D.S. (1986): *The Mathematics of Inheritance Systems*, Pitman, London.

Tse, T.H. & Pong, L. (1991): An Examination of Requirements Specification Languages, *The Comput. J.*, **34**, 2, 143-152.

Tsichritzis, D. & Klug, A. (1978): The ANSI/X3/SPARC DBMS Framework Report of the Study Group on DBMS, *Inf. Sys.*, **3**, 173-191.

van Biljon, W.R. (1988): Extending Petri Nets for Specifying Man-Machine Dialogues, *Int. J. Man-mach. Stud.*, **28**, 4, 437-455.

van den Bos, J., Plasmeijer, M.J. & Hartel, P.H. (1983): Input-output tools: A language facility for interactive and real-time systems, *IEEE Trans. Softw. Eng.*, **9**, 3, 247-259.

Wallace, V.L. (1976): The semantics of graphic input devices, *Comput. Graph.*, **10**, 1, 61-65.

Wasserman, A.I. (1981): User Software Engineering and the Design of Interactive Systems, *Proc. 5th Conf. on Softw. Eng.*, IEEE, 387-393.

Wasserman, A.I. (1985): Extending State Transition Diagrams for the Specification of Human-Computer Interaction, *IEEE Trans. Softw. Eng.*, **11**, 8, 699-713.

Wellner, P.D. (1989): Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation, *CHI '89 Proc.*, ACM, New York, 177-182.

Williams, A. (1986): A Comparison of Some Window Managers, in *Methodology of Window Management*, Hopgood F.R.A. *et al* (eds), Springer-Verlag, Berlin, 15-32.

Wolf, C.G. and Rhyne, J.R. (1987): A Taxonomic Approach to Understanding Direct Manipulation, *Proc. Hum. Fact. Soc.*, 31st Annual Mtg., 576-580.

Glossary

In compiling this glossary, reference has been made to Bass & Coutaz (1991), Hartson & Hix (1989), Johnson (1992), Myers (1989), and Thimbleby (1990). It should be noted that as Lean Cuisine+ is a super set of Lean Cuisine, definitions and terms pertaining to Lean Cuisine also apply to Lean Cuisine+.

action a primitive operation, performed by the user or the computer, which generates an event.

active dialogue a current dialogue or subdialogue.

active state a current interface state or substate.

active window the Macintosh window with the current keyboard focus.

adaptive approach an approach to system modelling in which an existing model is modified to suit a new requirement.

analogous action a method of action specification in a direct manipulation interface in which actions are demonstrated in an analogous manner.

analytical notation a dialogue notation which is suited to the analysis of existing interfaces.

AND/OR graph a directed tree graph built using AND and OR nodes, which represents all possible sequences of nodes. See also *EDG*.

application a user required software program or suite of programs.

application area the area of a Macintosh window which presents a view of the object(s) of interest.

application document a Macintosh document which relates to an application.

application interface component the Seeheim model component which defines the semantics of the application.

application program see *application.*

articulatory distance the degree to which the form of communication with a model world-based interface reflects the semantic objects and tasks involved.

asynchronous dialogue a dialogue in which more than one task or thread may be available at any time, and in which there is generally an absence of sequence within the separate threads or subdialogues. Also referred to as *event-based.* See also *multi-threaded.*

attribute a property of an object.

augmented STD a STD with an associated set of registers (global variables).

Backus-Naur form (BNF) a rule-based formal grammar which has been used in the modelling of dialogues.

base layer the bottom layer of the Lean Cuisine+ notation structure, comprising the dialogue tree plus object state variables.

base layer tree diagram the Lean Cuisine+ dialogue tree in which selectable representations (menemes) are grouped. (See Section 5.2)

behaviour the underlying behaviour of a direct manipulation interface, as distinct from its surface behaviour or presentation.

behavioural view a view of a direct manipulation interface concerned with both the semantics and syntax of the dialogue. It can be described using a dialogue model. (See Section 2.4)

bit-mapped display a high resolution pixel-based computer display.

BNF see *Backus-Naur form.*

button a primitive virtual input device the sampling output of which is an identifier indicating the code of the button currently being depressed.

CASE tool a software support environment for computer aided software engineering providing diagram editors for constructing system models, linked to a central dictionary holding a system description.

character 1. one of the available set of keyboard characters.
2. an interaction task the purpose of which is to generate a character from the available set.
3. the primitive object from which text documents are constructed.

character set the available set of keyboard characters.

character string a concatenated string of characters representing a value.

check boxes a set of buttons in a Macintosh dialogue box providing a means of presenting a set of mutually compatible selectable options.

Clipboard a holding area for data on the Macintosh, associated with copying and pasting.

close the operation of closing a window in a direct manipulation interface.

combinative approach an approach to system modelling which involves multiple separate models.

command a request from the user for the computer to perform an operation.

command language interface a text-based interface in which users issue commands expressed in a concise notation with a strict syntax to invoke operations, and in which there is little information displayed.

communicating sequential processes (CSP) a formal algebra capable of describing the behaviour of systems when seen as groups of communicating processes. (see Section 3.2.3)

compactness an informal measure of the size of a dialogue specification in a graphical dialogue notation.

composite object an interface object constructed from other objects.

compound task an interaction task made up of a continuous series of repetitions of a simpler task.

computer display a physical output device in the form of a (usually) rectangular screen.

computer event an internal or output event. See also *system directive.*

conceptual view a functional view of a direct manipulation interface which is based on concepts familiar to the user. (See Section 2.4)

concurrency the ability of a dialogue notation to represent concurrent subdialogues.

concurrent dialogue a multi-threaded dialogue in which more than one thread or subdialogue can be executed simultaneously.

consistency the employment of consistent graphical elements in a direct manipulation interface.

consolidation the fourth stage of a methodology for constructing Lean Cuisine+ specifications, in which constraints are captured in the form of overlays to the base layer dialogue tree. (See Section 7.4)

constrained trigger a Lean Cuisine+ selection trigger in which the behaviour of the selected or deselected object is modified, inhibiting further links, and preventing side effects such as cycles. (See Section 5.3)

constraint a condition, interrelationship or dependency involving one or more dialogue primitives.

conversational metaphor a model of interaction which is based on the notion of a conversation in which the two participants - the user and the computer - speak in turn in order to work step-by-step towards some goal.

Copy an editing operation on the Macintosh in which selected data is copied to the Clipboard.

CSP see *communicating sequential processes.*

Cut an editing operation on the Macintosh in which currently selected data is deleted.

DAL see *dialogue activation language.*

decomposition the first stage of a methodology for constructing Lean Cuisine+ specifications, which involves a task decomposition. (See Section 7.4)

default choice a Lean Cuisine meneme modifier which indicates a meneme that is to be in the selected state at the commencement of a dialogue or subdialogue.

deselection event an implicit event associated with a real meneme, unless over-ridden.

deselect-only meneme a Lean Cuisine meneme which can be deselected, but not directly selected, by the user.

design methodology a set of procedures and principles for the development of an interface.

design notation a dialogue notation for use in high level design.

design representation an interface design expressed in terms of a dialogue notation.

desktop the Macintosh environment created and maintained through the Finder. (see Section 2.3)

desktop metaphor a model of interaction which is based on electronic counterparts to the physical objects in an office. (See Section 2.2)

device-oriented model a low level interface model which is close to the physical input and output devices.

dialogue 1. the structure of the observable two-way exchange of symbols and actions between user and computer, expressed in terms of constraints relating to dialogue primitives.
2. (in Lean Cuisine+) a set of selectable primitives arranged in a tree structure, and a set of constraints over these primitives.

dialogue activation a direct manipulation dialogue model based on the notion of a hierarchy of dialogue cells, each of which can be activated and made available to the user only when its parent cell is active. (See Section 8.3)

dialogue activation language (DAL) a notation based on the concepts of dialogue activation. DAL incorporates UAN; a hierarchical arrangement of µdialogues; and concurrency expressed visually in a derivative of Lean Cuisine.

dialogue box the display of a modal subdialogue on the Macintosh in the form of a two-dimensional option or property sheet. (See Section 2.3)

dialogue cell See *µdialogue.*

dialogue control the control of the interaction taking place at an interface.

dialogue control component the Seeheim model component which defines the structure of the dialogue between the user and the application program.

dialogue header the root node in a Lean Cuisine+ dialogue tree. When selected, a dialogue header provides access to the subdialogue(s) and/or state variable values immediately below it, subject to any constraints applying. (See Section 5.6)

dialogue independence the separation of the design of dialogue from the design of computational software.

dialogue interlink diagram (DID) a top level Lean Cuisine+ diagram showing the root nodes of dialogue trees and named selection triggers between them. (See Section 6.5)

dialogue model an abstract formal or semi-formal model which is used to describe the dialogue between user and computer.

dialogue notation a graphical or textual notation associated with a dialogue model.

dialogue primitive the smallest unit of exchange in a dialogue. See also *action*.

dialogue specification the description of a dialogue in a dialogue notation.

dialogue specification language a formal language which supports the implementation of a dialogue.

DID see *dialogue interlink diagram*.

direct engagement the feeling of communicating directly with the objects of interest in an interface based on the model world metaphor.

direct manipulation interface an event-driven asynchronous interface which is based on the model world metaphor. (See section 2.1)

directory the content of a Macintosh Finder window, which shows the folders and/or documents available.

document the Macintosh representation of an application file.

document folder a Macintosh Finder repository which provides for hierarchical organisation of documents and applications in the desktop environment. (See Section 2.3)

dragging a mouse technique for selecting and moving objects on the computer display, which involves moving the mouse with the button held down.

drawing a freehand operation, undertaken using a pointing device such as a mouse, which generates a continuous series of positions on the display.

drawing tool a MacPaint mode representing a particular type of drawing tool, in which user input is interpreted accordingly. (See Section 4.1)

dynamic behaviour the state changes at an interface which result from the occurrence of events.

dynamic default a Lean Cuisine meneme modifier which may be associated with a meneme (assigned) or a meneme group (unassigned), and which indicates a default choice which takes on the value of the last user selection from that group.

dynamic instance set see *object instance set*.

dynamic interface a direct manipulation interface in which object instances can be created and deleted as a dialogue progresses.

EDG see *event decomposition graph*.

Edit menu a Macintosh pull-down menu in the main menu bar which offers a set of options for editing text.

ellipsis the symbol used to indicate stepwise refinement in a Lean Cuisine+ dialogue tree diagram. (See Section 5.2)

ERS see *event response system*.

Euler/Venn diagram a graphical notation which uses enclosing areas to represent sets and subsets.

event an occurrence which results from a user or computer action.

event-based dialogue see *asynchronous dialogue*.

eventCSP an executable subset of CSP which forms part of Specifying and Prototyping Interaction (SPI) tools. (See Section 3.2.3)

event decomposition graph (EDG) a dialogue notation which comprises temporal sequences of events and actions, represented as an AND/OR graph. (See Section 3.2.1)

event interlink a link between two events such that the occurrence of the first triggers the second, possibly conditionally.

eventISL a language which forms part of SPI tools, and which specifies events in terms of dialogue state. (See Section 3.2.3)

event model an implementation model for asynchronous dialogue which is based on a set of event handlers for processing discrete events.

event postcondition a condition which applies after an event has occurred.

event precondition a condition which must apply for an event to occur. See also *state transition rule*.

event response loop a program loop in an ERS implementation which continuously responds to incoming events.

event response system (ERS) a rule-based dialogue model which describes the state changes resulting from input events, and form the basis of a technique for implementing direct manipulation interfaces. (See Section 3.2.3)

event sequence a temporal sequence of input and/or output events. Dialogue models can be classified according to their representation of event sequence. (See Section 3.2)

executable specification a dialogue specification which provides for simulation of dynamic aspects of the interaction.

execution mode a Lean Cuisine+ support environment mode which provides for simulation of the behaviour of an interface. (See Section 8.2)

existence dependency a relationship between two interface objects, or between an object and a dialogue, in a direct manipulation interface, such that the existence of one is dependent upon the existence of the other. Can be mutual.

existence dependency layer the Lean Cuisine+ overlay which shows existence dependencies. (See Section 5.3)

expressiveness an informal measure of the ability of a dialogue notation to handle hierarchy and orthogonality.

extension the final stage of a methodology for constructing Lean Cuisine+ specifications, in which menus are added. (See Section 7.4)

external event see *input event*.

external selection trigger a selection trigger in Lean Cuisine+ which involves menemes from more than one dialogue. (See Section 6.1)

feedback a system response to a user action.

File menu a Macintosh pull-down menu in the main menu bar which offers a set of options for manipulating files.

Finder the system program on the Macintosh which supports the desktop. (see Section 2.3)

flag a local (usually Boolean) variable which is used to encode the state of a dialogue and control execution in an ERS.

font size the current point size of text characters in Macintosh applications.

font style the current style of text characters in Macintosh applications.

font type the current font type of text characters in Macintosh applications.

fork symbol the Lean Cuisine+ symbol used to denote a homogeneous set of objects or options. (See Section 5.2)

formal grammar a formal notation for describing language syntax which has been used in the modelling of dialogues. The best known form is Backus-Naur form (BNF).

formal model a dialogue model defined in a language which is mathematically precise, and in which strict syntax and semantics are used.

form filling a style of interaction which involves the user in selecting options and entering values into fields in two-dimensional property or option sheets, in response to prompts.

frame area the area surrounding a Macintosh window which contains the window controls.

functional design space the software development design space which represents the results of requirements analysis.

functional view a high level view which defines the general form of the set of capabilities of a system.

generic command a polymorphic command in a direct manipulation interface which can be applied to a range of objects and is recognised in all modes of the interface.

Get Value An underlying generic subtask which can be used as a basic building block in connection with interaction tasks, and which generates a representation of a single value each time it is performed, in the form expected by the interaction task. (See Section 4.1)

GKS see *Graphical Kernel System.*

goal a state to be achieved; the purpose for which a task is undertaken.

graphical interface an interface in which representations of objects of interest are displayed and manipulated graphically. See *model world metaphor.*

Graphical Kernel System the ISO standard providing a standard set of functions for graphics programming.

graphical language see *graphical notation.*

graphical notation a dialogue notation which is based on graphical representation of information.

graphical object a class of semantic computer object.

greying the dimming of the presentation of part or all of a Lean Cuisine+ dialogue tree diagram during the browsing or execution of the specification. Denotes unavailability during execution.

guard condition a condition placed on a DAL μdialogue which is used to determine whether or not it is available for activation. (See Section 8.3)

hierarchy the ability of a dialogue notation to support nesting of the dialogue specification.

high level design the early design phase of interface design at a high level of abstraction.

human-computer interface see *interface.*

HyperCard an object-based Macintosh application which supports the prototyping and generation of interfaces.

icon a compact pictorial representation of an object, such as a document folder or disk, under the desktop metaphor.

IDF see *interface definition file.*

IDLC see *interface development life cycle.*

inactive window an open window without the keyboard focus on the Macintosh.

incremental action a user action in a direct manipulation interface which is one of a series of actions working towards some goal.

input device a physical device for use in generating input events.

input event an occurrence, resulting from a user action, which generates a value.

insertion point the current point of insertion in a text document.

interaction the observable two-way exchange of symbols and actions between a user and a computer.

interaction event see *event.*

interaction task a task undertaken by a user.

interaction technique a way of using a physical input device.

interactive computer system a software system supporting interaction between user and computer.

interface the part of a computer system which provides the user with access to the facilities offered.

interface definition file an intermediate form of a Lean Cuisine specification in the MERS environment. (See Section 8.3)

interface development life cycle (IDLC) the collection of related activities which are involved in the production of an interface. Frequently expressed in terms of a number of discrete stages, but highly iterative. (See Section 1.1)

interface framework a conceptual framework, based on levels of abstraction, which provides different views of an interface. (See Section 2.4)

interface model an abstract model which is used to describe a particular view of an interface.

interface toolkit see *toolkit*.

internal event an occurrence which results from a computer action.

internal selection trigger a selection trigger in Lean Cuisine+ which involves menemes from a single dialogue.

keyboard 1. a physical input device used to enter characters.
2. a virtual input device the sampling output of which is the entire content of a block of concatenated characters.

keyboard focus the status of the window which is currently connected to the keyboard input device in a direct manipulation interface.

layer a stratum in the lean Cuisine+ notation. The notation comprises a base layer plus constraint and task overlays.

Lean Cuisine a graphical notation which is based on the use of tree diagrams to describe systems of menus. The behaviour of a menu is described using selectable representations called menemes, in which selections may be logically related or constrained through mutually compatible and mutually exclusive meneme groupings, and through the use of meneme modifiers. (See Section 3.2.3)

Lean Cuisine+ an object-based multi-layered development of Lean Cuisine in which a direct manipulation interface is described in terms of the constraints and dependencies which exist between selectable dialogue primitives. (See Section 4.3)

level a level of abstraction in a Lean Cuisine+ specification, based on an object classification. (See Section 6.6)

linguistic model an interface framework which is based on semantic, syntactic and lexical levels of interaction.

locality the ability of a dialogue notation to maintain local adjacency between dialogue primitives.

locator a primitive virtual input device the sampling output of which is a current location on the screen corresponding to the position of the device.

logical input device see *virtual device*.

MacDraw an object-based Macintosh drawing application providing a set of tools which can be used to construct diagrams. (See Section 6.3)

MacPaint a pixel-based Macintosh painting application providing a set of tools which can be used to construct pictures. (See Section 4.1)

main menu bar a bar along the top of the computer display in a direct manipulation interface which provides access to the main menus.

meneme an individual selectable representation of an object, operation, state or value, in a Lean Cuisine dialogue description.

meneme grouping see *mutually exclusive group* and *mutually compatible group*.

meneme modifier a designator attached to a meneme in a Lean Cuisine diagram which modifies its behaviour.

menu a set of selectable representations (menemes) in which selection may be constrained.

menu selection a style of interaction which involves the user in making selections from menus.

MERS see *modified event response system.*

micro-dialogue see *μdialogue*

Microsoft Word a word processing application available on the Macintosh. (See Section 6.2)

modal subdialogue a subdialogue within which all other parts of the dialogue are inaccessible.

mode a substate of a user interface which places particular interpretation on user input.

model of interaction the user's mental view of an interface, which is based on a metaphor for interaction. See *conversational metaphor* and *model world metaphor*.

model world metaphor a model of interaction in which the user has the illusion of acting directly upon the objects of interest without the intermediary of the system.

modified event response system (MERS) a modified event response system (ERS) notation which has been applied to menu systems. (See Section 8.3)

monostable meneme a Lean Cuisine meneme which following selection reverts to an unselected state on completion of the operation it represents. Reversion is through system action (exceptionally following user intervention - see Section 5.4).

mouse a physical input device in a direct manipulation interface which is used to point, select, drag etc.

mouse cursor the cursor in a direct manipulation interface which represents the current on-screen position relating to the mouse.

move the operation of moving an object on the computer display in a direct manipulation interface.

multi-agent model an implementation model for asynchronous dialogue which applies the separation of an application from its interface at various levels of abstraction, and distributes dialogue control among a set of co-operating agents.

multi-threaded dialogue a dialogue in which more than one thread (subdialogue) may be available at any time.

mutually compatible group a group of Lean Cuisine menemes in which one or more menemes may be in a selected state at any time.

mutually exclusive group a group of Lean Cuisine menemes in which only one meneme may be in a selected state at any time, and in which the selection of one meneme causes any previously selected meneme in the group to become deselected.

non-determinancy the situation in which a single task has multiple subtasks which can occur in any order.

object either an object type or object instance, in general discussion.

object-action model a model of interaction in which a user selects an object on the screen. The selected object is then a candidate for actions. (See Section 2.1)

object-based a dialogue model in which events are grouped around objects.

object box the Lean Cuisine+ graphical symbol which is used to denote a meneme representing an object.

object hierarchy a hierarchical structure relating objects together.

object instance set the set of all current instantiations of an object. The set may be dynamic and exhibit a behaviour. (See Section 5.2)

object instantiation an instance of an object. Used where it is important to distinguish an instance from a type. See also *object.*

object-oriented a system in which hierarchies of objects are manipulated.

object state variable (OSV) an attribute of an object type in Lean Cuisine+ which forms part of the total system state. (See Section 5.2)

object subtype an object subclass which inherits the state variables and subdialogue behaviour associated with its parent class, unless inhibited. (See Section 5.2)

object type an entity type with a set of defining attributes. See also *object.*

object type hierarchy a hierarchical structure relating objects and their subtypes, sub-subtypes etc., with attribute inheritance. (See Section 5.2)

open the operation of opening a window in a direct manipulation interface.

operation see *task.*

option see *meneme.*

option precondition a precondition for the availability (for selection) of a meneme in the Lean Cuisine+ notation, which may involve the state of a meneme in another part of a dialogue, or the existence of some other state or condition, or both. Indirectly an event precondition.

option precondition layer the Lean Cuisine+ overlay which shows event preconditions. (See Section 5.3)

option sheet see *property sheet.*

orthogonality the ability of a dialogue notation to support multiple active states.

OSV see *object state variable.*

output device a physical device for use in presenting computer output.

output event an occurrence resulting from a computer action, which generates an output value.

palette a menu of tools.

parent dialogue the dialogue immediately above the subdialogue under consideration, in the dialogue hierarchy.

parent window an open window under the Macintosh Finder containing other document folders and/or documents.

passive meneme a Lean Cuisine meneme which is unavailable for selection or deselection by the user.

Paste an editing operation on the Macintosh which inserts data from the Clipboard at the current insertion point.

path an interaction task the purpose of which is to generate a series of positions over time.

Petri net an abstract virtual machine which is represented as a network graph comprising two types of nodes: circles (places) and bars (transitions) connected by directed arcs. In the modelling of systems, places represent conditions, and transitions represent events. (See Section 3.2.1)

pick a primitive virtual input device, the sampling output of which is a reference to the graphical object currently being pointed at.

pointing and clicking a means of selecting an object or operation using the mouse, in a direct manipulation interface.

position an interaction task the purpose of which is to indicate a position on the display.

presentation the appearance and surface behaviour of a direct manipulation interface. Its 'look and feel'.

presentation component the Seeheim model component which is responsible for the external presentation of the user interface.

presentational view a view of a direct manipulation interface which is concerned with displayed objects, physical user actions, interaction techniques, and the layout of the computer display.

pressing a mouse technique for generating repeated selections, which involves pressing and holding down the button.

pre-validation the removal of the possibility of syntax errors in a direct manipulation interface, through the presentation of only valid operations in each interface state.

problem domain see *task domain.*

problem space a possible way in which to represent a task so as to obtain a solution. Consists of a set of symbolic structures (the states of the space) and a set of operators over the space. (See Section 9.1)

process a related group of tasks.

process structure diagram (PSD) a graphical notation which is used to specify tasks in terms of time-ordered sequences of actions, and which is based on the three control structures of sequence, selection, and iteration. (See Section 3.2.1)

programming language approach an approach adopted in many UIMSs in which the designer is required to specify interfaces in a textual, formal, programming-style language.

property sheet a form which is used to display of the attributes (or properties) of an object in a direct manipulation interface.

Prototyper an object-based Macintosh application which supports the prototyping and generation of interfaces.

prototyping a 'trial and error' software engineering activity which simulates the behaviour and appearance of an interface under design, in order to support early evaluation.

PSD see *process structure diagram.*

pull-down menu a menu in a direct manipulation interface, the content of which appears on the computer display below its header, following selection of the header.

quantify an interaction task the purpose of which is to specify a value to quantify a measure.

radio buttons a set of buttons in a Macintosh dialogue box providing a means of presenting a set of mutually exclusive selectable options.

reactive system an event driven system.

real meneme a Lean Cuisine meneme which is selectable unless modified. See *meneme.*

recursive STD a STD which permits sub-diagrams to call themselves.

required choice a Lean Cuisine meneme modifier associated with a subdialogue or subgroup header, which indicates a meneme group from which a valid choice is always required.

reverse video the technique used to visually highlight a selected meneme during the browsing or execution of a Lean Cuisine+ specification.

reversible action a user action in a direct manipulation interface, the effect of which can be reversed through some form of 'Undo' operation.

right arrow a visual indication in a generic object box of a mechanism for selecting instantiations during execution of a Lean Cuisine+ specification.

root dialogue the top level dialogue in a Lean Cuisine+ tree, which when made available through selection of the dialogue header (root node) provides access to the subdialogue(s) and/or state variable values immediately below it, subject to any constraints applying. (See Section 5.1)

rule a component of an ERS specification which specifies either a response to an external event (regular rule) or an action to be taken when some state is entered (ε-rule).

scroll the operation in a direct manipulation interface of changing the visible portion of the object in the viewing area of a window, by manipulating the scroll box.

scroll bar the horizontal or vertical plane within which a scroll box can be moved in the frame area of a window.

scroll box the slider within a scroll bar in the frame area of a window.

Seeheim model an interface framework which is based on a clear separation between an application and its interface. (See Section 2.4)

select an interaction task the purpose of which is to select from a set of alternatives.

selection action the primitive user action underlying all direct manipulation tasks.

selection event an implicit event associated with a real meneme, unless overridden.

selection trigger a directed link between menemes in the Lean Cuisine+ notation, such that the selection or deselection of a meneme by the user triggers (possibly conditionally) the selection or deselection of one or more other menemes in the same or different dialogues. A selection trigger represents a system response to a user action.

selection trigger layer the Lean Cuisine+ overlay which shows selection triggers. (See Section 5.3)

select-only meneme a Lean Cuisine meneme which can be selected, but not deselected, by the user.

semantic distance the degree to which the semantic concepts used by a model world-based interface are compatible with those of the task domain.

semantic feedback the presentation of feedback in terms of the task domain, as provided in a direct manipulation interface.

semantic object a direct manipulation interface object which models the task domain.

semi-formal model a class of model on a spectrum of models going from informal at one end to strictly formal at the other. Can be applied to a dialogue model such as Lean Cuisine+ for use in high level design, which is intuitive rather than strictly formal.

sequence-based a dialogue model which represents a dialogue in terms of event sequences.

sequential dialogue a dialogue in which only one task is presented to the user at any one time.

side effect an event triggered as a result of a meneme selection in a Lean Cuisine+ specification which leads to a further meneme selection or deselection.

simple modal subdialogue a Lean Cuisine+ modal subdialogue normally involving a primitive operation at a terminal node, and represented by a monostable meneme. Under certain conditions a further modal subdialogue requiring user action may be triggered. (See Section 5.4)

simplicity the employment of simple graphical elements in a direct manipulation interface.

single event/single state model a dialogue model in which only a single state can be active at any time, and which suits sequential dialogue.

size the operation in a direct manipulation interface of resizing an open window.

size box a Macintosh button which provides a means of resizing windows.

software system an automated system that is implemented in executable code.

solution domain the combination of the semantic and syntactic knowledge pertaining to a problem area.

SRS see *system requirement specification.*

state the current set of conditions at a point in a dialogue. Dialogues can be classified according to whether they support single or multiple active states.

state-based a dialogue model in which events are grouped around states.

state change the change in state of an interface which results from the performance of a task or primitive action. Can be observed through the 'execution' of some dialogue specifications.

statechart a higraph-based extension of the STD formalism developed for describing the behaviour of complex reactive systems. A system is described in terms of the state changes resulting from events, with events grouped in connection with change of state. (see Section 3.2.2)

state hierarchy a hierarchical structure relating states and substates.

state orthogonality the circumstance in which two more independent states can be active at the same time.

state transition diagram (STD) a graphical notation for representing finite state machines, consisting of nodes, which represent states, interconnected by directed labelled arcs, which represent transitions between them. (See section 3.2.1)

state transition rule one or more event preconditions, relating to a task or primitive action, which must apply for a state change to occur.

state variable a variable whose value is associated with the current state of an interface.

state variable value a Lean Cuisine+ terminal meneme representing a value available for selection or deselection. (See Section 5.6)

STD see *state transition diagram.*

stepwise refinement the decomposition of the functional elements of a system.

stroke a non-primitive virtual input device the sampling output of which is a series of screen locations.

structural design space the software development design space which represents the results of initial system decomposition.

subdialogue 1. a discrete component part of a larger dialogue.
2. a logical grouping of menemes within the body of a Lean Cuisine+ dialogue, headed by a real meneme representing an object or operation. When made available through selection of its header, a subdialogue provides access to the subdialogue(s) and/or state variable values immediately below it, subject to any constraints applying. (See Section 5.1)

subdialogue header an intermediate real meneme heading a Lean Cuisine+ subdialogue.

subgroup a logical grouping of menemes in the body of a Lean Cuisine+ dialogue or subdialogue, headed by a virtual meneme.

subgroup header a virtual meneme heading a Lean Cuisine+ subgroup.

substate a discrete component part of a larger state.

subtask a discrete component part of a larger task.

subtree a discrete component part of a larger Lean Cuisine+ dialogue tree.

subtype link the Lean Cuisine+ arc linking an object subtype with its parent type.

subtype link bar a bar across a Lean Cuisine+ subtype link which inhibits selection propagation and subtype inheritance.

surface behaviour the visible behaviour of an interface.

switch mechanism a mechanism employed in some dialogue notations which leads to different states depending on the setting of a switch.

switch meneme a special type of Lean Cuisine+ meneme which operates as a modifier on meneme group behaviour.

syntactic object a direct manipulation interface object which results from the use of computer technology and which is independent of the task domain.

syntactic-semantic model a model of interaction which is based on a separation between syntactic and semantic knowledge. Semantic knowledge models the task domain, whereas syntactic knowledge is dependent upon the computer system used to support the interaction.

synthesis the third stage of a methodology for constructing Lean Cuisine+ specifications, in which subdialogue trees are combined into a single dialogue tree. (See Section 7.4)

synthetic notation a dialogue notation which is suited to the design of new interfaces.

system a connected set of procedures.

system dictionary an organised repository carrying a description of the components of a Lean Cuisine+ specification.

system directive a system initiated event. See also *computer event.*

system requirement specification (SRS) a functional specification of a software system.

tab a visual indicator above an object box containing the name of the current object instantiation during execution of a Lean Cuisine+ specification.

tandem execution the execution of a Lean Cuisine+ specification in tandem with the manipulation of a prototype of the interface. (See Section 8.3)

task an activity, involving actions and objects, which when undertaken to achieve a goal results in a change of state of the interface. Also referred to as an *operation.*

task analysis a study concerned with the analysis and decomposition of tasks, and their relationship with goals and procedures. (See Section 7.2)

task decomposition a major component of task analysis which is concerned with a hierarchical decomposition of tasks.

task domain the semantic knowledge pertaining to a problem area.

task graph the graph resulting from a task decomposition.

task layer the Lean Cuisine+ overlay which provides for the description of higher level user tasks in terms of primitive actions (and associated events), and which captures any temporal relationships between actions. (See Section 7.3)

task model an abstract model which is used to describe interactive tasks.

text an interaction task the purpose of which is to generate characters.

text cursor the cursor which represents the current on-screen keystroke position. It applies to text documents only.

text document a Macintosh document containing textual data. A text document is a composite object made up of one or more character objects.

text entry a style of interaction which involves the user in entering data made up of characters selected from the keyboard.

toolkit a library of predefined interaction techniques.

transition the movement from one interface state to another.

translation the second stage of a methodology for constrncting Lean Cuisine+ specifications, in which subdialogue trees are constructed. (See Section 7.4)

Trash folder a special document folder on the Macintosh desktop into which documents are placed prior to their deletion.

UAN see *user action notation.*

UIMS see *user interface management system.*

unconstrained trigger a Lean Cuisine+ selection trigger in which 'knock on' effects are possible.

undo an operation in a direct manipulation interface which provides for reversal of the last action performed.

ungreying the undimming of the presentation of part or all of a Lean Cuisine+ dialogue tree diagram during the browsing or execution of the specification. Denotes availability during execution.

universal command see *generic command.*

user a person or group interacting with a computer.

user action notation (UAN) a task-oriented representation of user action sequences in relation to screen objects for asynchronous interaction, incorporated into DAL. (See Section 8.3)

user event see *input event.*

user input a user generated event.

user interface see *interface.*

user interface management system (UIMS) an integrated set of tools which assists with the design, implementation and management of user interfaces.

user model a model describing the attributes of users, their abilities, etc.

user's view see *conceptual view.*

user tailorability the degree to which an interface can be adapted to the needs of a particular user.

validation the analysis of a dialogue specification to ensure that it is semantically correct (meets user requirements).

valuator a primitive virtual input device the sampling output of which is a real vector corresponding to the current input of the valuator device.

value a command, number, location or name, as represented by a character string.

verification the analysis of a dialogue specification to ensure that it is syntactically correct.

view a level of abstraction at which an interface can be usefully considered. A view can be described in terms of an interface model.

virtual device a logical (hardware independent) input device.

virtual meneme a non-selectable Lean Cuisine meneme which is used to partition a dialogue.

visual correlation a method of object specification in a direct manipulation interface which involves performing a selection operation in the vicinity of the object.

visual formalism a graphical representation of a formal model.

widget an interaction object.

window a discrete area of a computer display providing a view of one or more objects. Overlapping windows are fundamental to the desktop metaphor. (See Section 2.2)

window stack the stack structure which is used to maintain the order of open windows on the Macintosh. The window at the top of the stack is active.

WYSIWYG 'What You See Is What You Get'. An acronym denoting a generic class of word processing application which presents on the computer display the final printed form of a text document.

μdialogue a discrete DAL micro-dialogue. (See Section 8.3)

μdialogue activation the activation of a DAL μdialogue. A dialogue cell becomes active only if its parent dialogue cell is active, and it receives one of a set of possible activating events.

μdialogue deactivation the deactivation of a DAL μdialogue. An active dialogue cell becomes deactivated if it receives one of a set of possible deactivating events.

μdialogue instance an instance of a DAL μdialogue class.

μdialogne template a DAL μdialogue class that defines a generic behaviour and appearance. Any derived subclasses inherit parent class behaviour.

Appendices

Appendix A	The Lean Cuisine Notation	227
Appendix B	A Macintosh Task Analysis	231
Appendix C	The Lean Cuisine+ Notation	237
Appendix D	A Support Environment Prototype	245

Appendix A

The Lean Cuisine Notation

The following summary is based on Apperley (1988).

A1. Menu definitions:

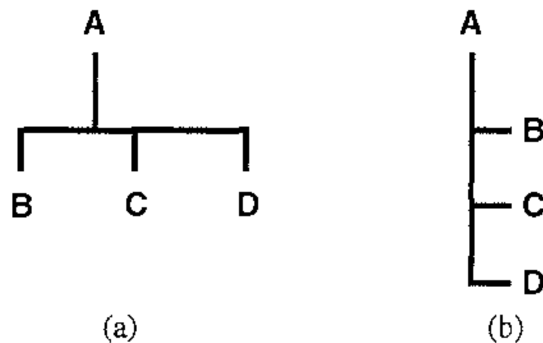
A *menu* is a set of selectable representations of actions, parameters, objects (which may be other menus), states, and other attributes, in which selections may be logically related and/or constrained.

A *meneme* is defined as an individual selectable representation (of an action, parameter, object, menu, or other attribute) within a menu, the minimum or basic unit of information in the two-way dialogue between the user and the application.

A meneme has just two possible *states*, selected and not selected, and the state may be changed either by direct excitation, or by indirect modification.

A2. Menu subgroup structures:

Within a menu, menemes are clustered into syntactic subgroups that are either *mutually exclusive* (1-from-N) or *mutually compatible* (M-from-N). In the Lean Cuisine notation these structures are represented diagrammatically as follows:



(a) The tree diagram for a mutually compatible group, and
(b) that for a mutually exclusive group.

A3. Lean Cuisine definitions:

Menemes may be *terminal* (leaf) menemes, in which case they represent specific actions or parameters, or they may be *non-terminal* menemes, in which case they are themselves headers to other menus.

Virtual menemes are used to partition a single menu into constituent syntactic subgroups. They correspond to nodes in the Lean Cuisine tree, but are not presented to, nor directly accessible to, the user.

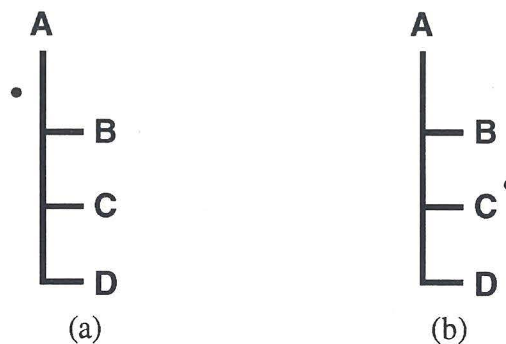
The menu corresponding to a *real* non-terminal meneme in a Lean Cuisine diagram consists of all of the real menemes with which the target meneme is connected by downward directed branches, either directly or via virtual menemes. If there is a sequence associated with the menemes, it is as scanned from left to right and from top to bottom.

A4. Menu and meneme modifiers:

A menu or subgroup header may be tagged as a *required choice* group; this places the additional constraint on the relationships between the menemes of that group, that a valid selection is always required. An initial default choice must be shown under these circumstances.

An *initial default* choice indicates a meneme that is to be initially in the selected state. It may subsequently be deselected, directly or indirectly, according to the constraints and interrelationships that apply.

A *dynamic default* is a default which takes on the value of the last user selection from that group. It may have an initial assignment, or it may be initially unassigned, in which case the first user choice from that group becomes its first value (see figure below).



(a) An unassigned dynamic default applying to the subgroup BCD, and
(b) a dynamic default applying to the same group, but initially assigned to C.

Menemes may be *bistable*, *select-only*, *deselect-only*, *monostable* or *passive*. The type is normally determined by the subgroup constraints. However, the default type may be explicitly overridden by specifying one of the other types (see special characters).

A5. Special characters:

The following special characters are used in the Lean Cuisine diagrammatic notation:

{ } *Virtual meneme:* the meneme name appears between braces.

§ *Required choice:* normally associated with a menu or subgroup header.

*	*Default choice:* normally associated with a meneme.
•	*Dynamic default:* may be associated with a meneme (assigned) or a group (unassigned)
↑↓	*Bistable meneme:* used to explicitly override the default type.
↑	*Select-only meneme:* used to explicitly override the default type.
↓	*Deselect-only meneme:* used to explicitly override the default type.
⊥	*Monostable meneme:* used to explicitly override the default type.
⊗	*Passive meneme:* used to explicitly override the default type.

Appendix B

A Macintosh Task Analysis

This appendix presents the detailed results of a task analysis of a representative selection of Macintosh processes, arranged as shown below. The implications of this analysis are discussed in Section 4.1.

Appendix B1: Macintosh Finder

1. Icons
2. Windows
3. Menus
4. Dialogue boxes
5. Commands
6. Text

Appendix B2: MacPaint application

B1: Macintosh Finder

Object	Action	Effect	Method	Compound task	Task	Value type generated	Value represents
1. ICONS							
1. Icon	Select	Make icon current	click mouse on icon		Select	position	icon
2. Icon group	Select	Make group of icons current	drag mouse over group, or shift-click mouse on each icon		Select* Select*	position position	icon icon
3. Icon / group	(de)Select	Deselect icon(s)	click mouse outside group, or shift-click mouse on each icon		Select	position	<u>not</u> icon
4. Icon	Select/Open	Make icon current and open application / document etc	double-click mouse on icon		Select+	position	icon
5. Icon / group	Move	Move curreut icon(s) on desktop	drag mouse on icon(s)		Position	position	position
2. WINDOWS							
1. Wiudow	Select	Make wiudow active	click mouse in window		Select	position	window
2. Zoom box	Select	Make window fill screen	click mouse on zoom box		Select	position	Zoom-out commd.
3. Zoom box	(de)Select	Return window to former size	click mouse on zoom box		Select	position	Zoom-iu command
4. Close box	Select	Close window	click mouse on close box		Select	position	Close command
5. Size box	Move	Alter size of window	drag mouse on size box		Position	position	BRH corner posn.
6. Title bar	Move	Move window on desktop	drag mouse on title bar		Position	positiou	window position
7. Scroll arrow	Select	Scroll window by one line	click mouse on scroll bar arrow or press mouse to repeat		Quantify Quantify*	position position	numeric quantity numeric quantity
8. Scroll bar	Select	Scroll window by windowful	click mouse on scroll bar area or press mouse to repeat		Quantify Quantify*	position position	numeric quantity numeric quantity
9. Scroll box	Move	Scroll window	drag mouse on scroll box		Quantify	position	numeric quantity

* repeated task
+ Select plus implied system action

	Object	Action	Effect	Method	Compound task	Task	Value type generated	Value represents
3. MENUS								
1.	Menu title	Select	Pull down menu	press mouse on menu title		Select	position	menu
2.	Menu item	Select	Choose menu alternative	drag mouse down menu and release on highlighted item		Select	position	menu item
4. DIALOGUE BOXES								
1.	Name	Select	Choose named item from list	click mouse on name		Select	position	item
2.	Name group	Select	Choose several items from list	drag mouse over names, or shift-click mouse on each name		Select* Select*	position position	item item
3.	Name	Select/Open	Choose and open item	double-click mouse on name		Select+	position	item
4.	Button	Select	Choose command / option etc	click mouse on button		Select	position	option
5.	Arrow	Select	Set value from scale	click mouse on scroll arrow or press mouse to repeat		Quantify Quantify*	position position	numeric quantity numeric quantity
6.	Parameter	Enter	Enter value	hit character keys: numeric string or text string		Quantify Select	char. string char. string	numeric quantity item name
5. COMMANDS								
1.	Command	Select	Choose command	press apple key & hit char. key		Select	character	command
2.	Command	Confirm	Confirm command or entry	hit return or enter key		Select	character	command
3.	Action	Modify	Modify action	press control key & hit char. key		Select	character	command

* repeated task

Object	Action	Effect	Method	Compound task	Task	Value type generated	Value represents
6. TEXT							
1. Character	Enter	Enter lower-case character	hit character key		Character	character	character
2. Character	Enter	Enter upper-case character	press shift & hit char. key, **or** engage Caps Lock & hit char. key		Character	character	character
3. Character	Enter	Enter optional character	press option or option/shift keys, & hit char. key		Character	character	character
4. Position	Select	Choose text insertion point	click mouse button on desktop		Position	position	position
5. Position	Select	Choose text insertion point	hit tab (next stopping place) **or** hit return (beg. of next line)		Select Select	character character	tab command newline command
6. Text	enter	enter text	hit character keys	Text	Character*	character	character
7. Text	Select	Make a piece of text current	drag mouse **or** click mouse then shift-click mouse		Select*	position	start or finish position
8. Word	Select	Make a word current	double-click mouse		Select	position	character string

* repeated task

B2: MacPaint application

	Tool	Object	Action	Effect	Method	Compound task	Task	Value type generated	Value represents
1.	Brush/ Pencil/ Spray	Line	Draw	Draw a freehand line	drag mouse over document	Path	Position*	position	position
2.	Paintcan	Area	Fill	Fill selected area with current pattern	click mouse on enclosed area		Select	position	area
3.	Selection Rectangle	Area	Select	Make selected rectangular area current	drag mouse over document		Position*	position	position
4.	Lasso	Area	Select	Make object(s) within selected freehand drawn area current	drag mouse over document	Path	Position*	position	position
5.	Sel. Rect/ Lasso	Current Area	Move	Move selected area over document	drag mouse on selected area		Position	position	position
6.	Sel. Rect/ Lasso	Current Area	Copy	Copy selected area once	press Option Key **and** drag mouse on selected area		Position	position	position
7.	Sel. Rect/ Lasso	Current Area	Copy	Copy selected area repeatedly	press Option & Cmd. keys **and** drag mouse on sel. area		Position*	position	position
8.	Sel. Rect/ Lasso	Current Area	Stretch	Alter one dimension of selected area	press Command key **and** drag mouse on selected area		Position	position	position
9.	Sel. Rect/ Lasso	Current Area	Clear	Erase selected area	hit Backspace (Delete) key		Select	character	erase command
10.	Eraser	Area	Erase	Move eraser over document erasing area on path	drag mouse over document	Path	Position*	position	position
11.	Eraser	Window	Erase	Erase document under window	double-click mouse on eraser icon		Select+	position	erase command

* repeated task
\+ Select plus implied system action

	Tool	Object	Action	Effect	Method	Compound task	Task	Value type generated	Value represents
12.	Rectangle	Rectangle	Draw	Draw hollow or filled rectangle	drag mouse over document		Position*	position	position
13.	Oval	Oval	Draw	Draw hollow or filled oval	drag mouse over document		Position*	position	position
14.	Freehand Shape	Freehand Shape	Draw	Draw hollow or filled freehand shape	drag mouse over document	Path	Position*	position	position
15.	Polygon	Polygon	Draw	Draw hollow or filled polygon	drag mouse over document		Position*	position	position
16.	Drawing Tool	Dot(s)	Set	Set dot(s) to black/white in detailed drawing	click mouse on dot(s) or drag mouse over dot(s)		Select*	position	dot
17.	Drawing Tool	Dot(s)	Set	Set dot(s) to black/white to create "fill in" pattern	click mouse on dot(s) or drag mouse over dot(s)		Select*	position	dot
18.	Grabber	Document	Move	Move document under window	drag mouse on document		Position	position	position

* repeated task

Appendix C

The Lean Cuisine+ Notation

The Lean Cuisine+ notation is presented in the form of a concise tutorial.

C1. Basic features and definitions:

Dialogue
A *dialogue* is described by a set of selectable primitives, *menemes*, arranged in a tree structure, and a set of *constraints* over these primitives. Some constraints are captured in the dialogue tree diagram, and others in the form of overlays to the diagram (see Section 2). The total dialogue state at any time is the sum of all the meneme states plus the values of any *state variables.*

The top level or *root dialogue*, when made available through selection of its *dialogue header* (the root node), provides access to the *subdialogue(s)* and/or *state variable values* immediately below it, subject to any constraints applying.

Menemes
The *meneme* is the dialogue primitive, and is an individual selectable representation of an object, operation, state, or value. Menemes may be real or virtual. The notation is object-based and distinguishes menemes representing objects (see *Objects* below).

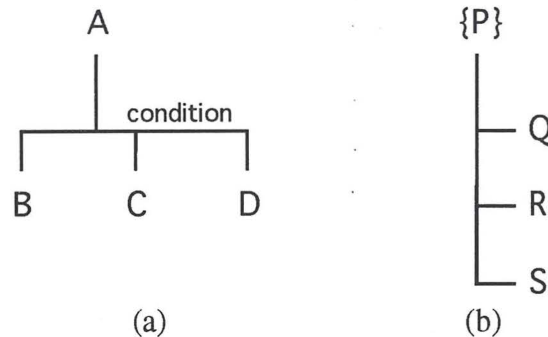
Real menemes represent specific selectable options, and may be terminal or non-terminal. Non-terminal real menemes are headers to further subdialogues. *Virtual* non-terminal menemes can be used to partition a dialogue or subdialogue into further constituent syntactic meneme groupings. They are not available for selection.

In each dialogue state, a meneme is either *available* (for selection) or *unavailable*, and either *selected* or *not selected.* This gives rise to four possible meneme states. The state may be changed either by direct excitation, or by indirect modification, subject to any constraints applying. Menemes may be *bistable* (default), *select-only*, *deselect-only*, *monostable* or *passive*. The type may be determined by subgroup constraints (see *Subdialogues* below). Alternatively, the default type may be explicitly overridden by specifying one of the other types through the use of *meneme designators* (which are summarised at the end of this section).

Subdialogues
Selectable primitives may be grouped into *subdialogues* to any level of nesting. A subdialogue is a logical grouping of menemes within the body of a Lean Cuisine+ dialogue, headed by a real meneme representing an object or operation. The subdialogue corresponding to a non-terminal real meneme in a Lean Cuisine+ dialogue tree consists of all of the real menemes with which the *subdialogue header* is connected by downward directed branches, either directly or via virtual menemes.

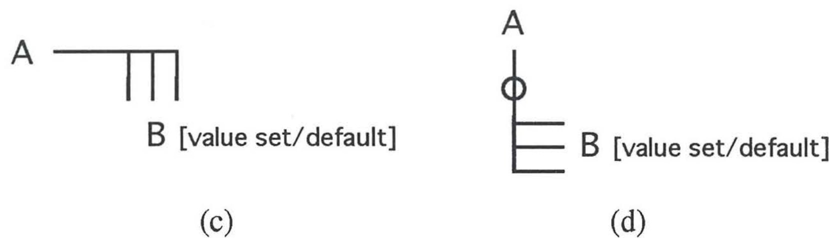
When made available through selection of its header, a subdialogue provides access to the subdialogue(s) and/or state variable values represented by these menemes, subject to any constraints applying. A subdialogue is available for selection only if its parent dialogue is active. Options in the parent dialogue remain available for selection, again subject to any constraints applying.

The grouping of menemes within a subdialogue places constraints on their behaviour. The grouping may be either *mutually exclusive (ME)* (1-from-N) or *mutually compatible (MC)* (M-from-N). In the latter case, selection may be governed by a condition. In the Lean Cuisine+ notation these structures are represented graphically as follows:



(a) A mutually compatible meneme group within subdialogue A.
(b) A mutually exclusive meneme group within subgroup P, which is headed by a virtual meneme.

Where a group of options constitute a *homogeneous set*, they may be represented graphically using a *fork* symbol. A set which may be null is indicated as shown in Figure (d) below. A specification of the range, value set and/or default value may optionally appear in braces:



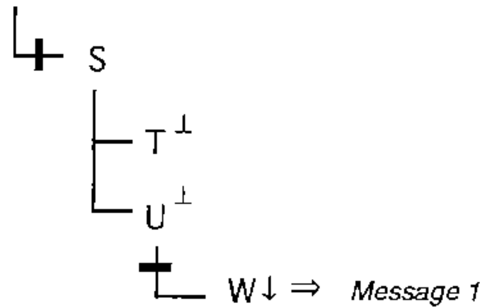
(c) (d)

(c) The tree diagram for a mutually compatible homogeneous group, and
(d) that for a mutually exclusive homogeneous group (which may be null).

Modal subdialogues
A subdialogue may be *modal*, in which case, following its selection, options in all other parts of the dialogue become unavailable. A modal subdialogue which involves a subtree is distinguished by a bar across the arc above the meneme which heads it. An example (S) appears in Figure (e).

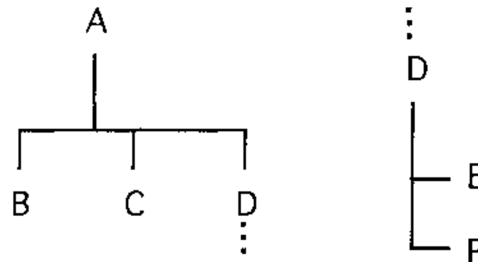
A modal subdialogue may be *simple*, in which case it is represented by a *monostable* meneme, shown thus ($\perp$). An example (T) appears in Figure (e). Following selection, a meneme of this type reverts to an unselected state on completion of the operation it represents, through system action.

Simple subdialogues are normally terminal nodes. Alternatively they may have explicit modal subdialogues associated with them which are triggered under certain conditions. In such cases the operation represented by the monostable meneme becomes suspended pending user termination of the further modal subdialogue, when it completes in the usual way. An example (U) appears in Figure (e). Here, the selection of meneme W (a deselect-only meneme) by the system is accompanied by the output of a directive (Message 1) to the user concerning appropriate action.



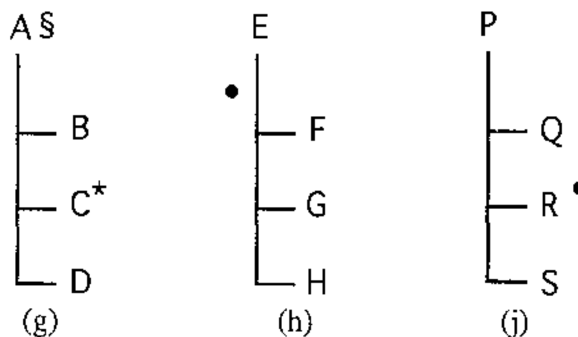
(e) S represents a modal subdialogue, and T a simple modal subdialogue. U represents a simple modal subdialogue which under certain conditions triggers a further modal subdialogue W requiring user action, as detailed in output Message 1.

Stepwise refinement
To support *stepwise refinement,* a dialogue or subdialogue tree may be represented visually by an ellipsis to indicate that the detail is developed in a separate diagram. In the case of a subdialogue, the lower level diagram is headed by an ellipsis in order to visually complete the association:



(f) Meneme D heads a subdialogue which is separately developed.

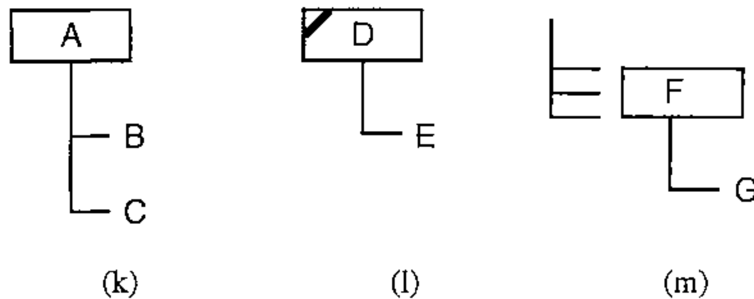
Further subdialogue constraints
A subdialogue header may be tagged to indicate a *required choice* group (§). This places the additional constraint on the relationships between the menemes of that group that a valid selection is always required. An initial *default choice,* which indicates a meneme that is to be initially in the selected state, must be shown (*) under these circumstances (see Figure (g) below). A *dynamic default* is a default which takes on the value of the last user selection from that group (•). It may have an initial assignment, or it may be initially unassigned, in which case the first user choice from that group becomes its first value (see Figures (h) and (j) below).



(g) A is a required choice group, and C is the initial default.
(h) An unassigned dynamic default applying to the subgroup FGH.
(j) A dynamic default applying to the group QRS, initially assigned to R.

Objects

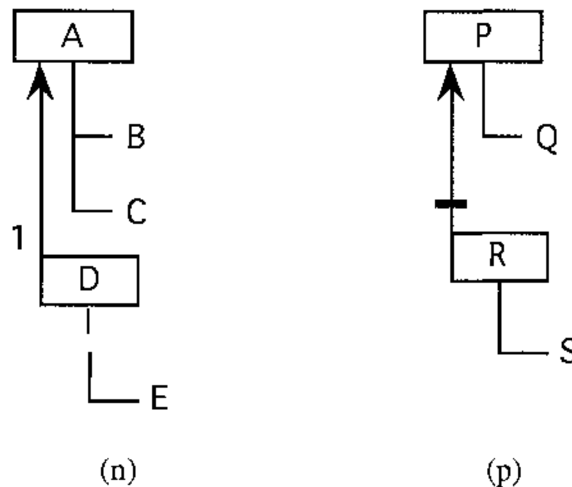
The notation is object-based and distinguishes menemes representing *object types,* which are shown boxed. The notation distinguishes *semantic* objects, which model the task domain, from *syntactic* objects, which form part of the computer domain. Syntactic objects carry a diagonal in the top left hand corner. *Homogeneous object instance sets,* which may be dynamic, are indicated by means of the fork symbol introduced above. Examples of objects are shown in Figures (k) to (m).



(k) Meneme A represents a semantic object, while
(l) Meneme D represents a syntactic object, and
(m) Meneme F represents one of a homogeneous set of semantic object instances.

Each meneme representing an object can have attributes attaching to it. These are referred to as *object state variables* (OSVs), and are a subset of the entire set of state variables associated with the dialogue. OSVs can also exist as parameters to options which may modify them as a dialogue progresses. The exact form of presentation of OSVs on a Lean Cuisine+ tree diagram is implementation dependent.

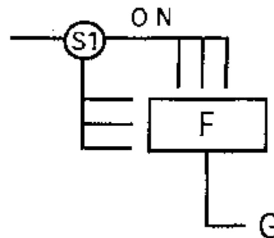
Object *subtypes* are supported. Each level in the type hierarchy is represented by a meneme which heads a subdialogue at that level. Selection is normally propagated in either direction in the hierarchy but can be inhibited. Specific state variables attach to each level. A subtype 'inherits' the state variables and subdialogue behaviour of its parent type (i.e. has access to the options in the parent dialogue) unless propagation is inhibited. In Figure (n) selection of subtype D also selects A, with access to dialogue options B and C. In Figure (p) selection of R does not select P, and option Q is unavailable. This is denoted by the bar across the subtype link.



(n) D is a subtype of A, with a cardinality of 1. D 'inherits' ME options B and C from A, and adds its own ME option E.
(p) R is a subtype of P, but does not 'inherit' option Q.

State variable values
A terminal node in a Lean Cuisine+ tree may represent a *state variable value* available for selection or deselection within that subdialogue. The value may remain set after termination of the subdialogue, in which case it may be referenced in other subdialogues, subject to any scoping restrictions within object type hierarchies.

Switch menemes
The behaviour of meneme groups can be changed through selection of a special type of meneme which operates as a modifier or *switch*, as shown below. The behaviour of the group can be switched between mutually exclusivity and mutually compatibility:



(q) The behaviour of object instance set F is mutually compatible when S1 is ON (i.e. selected), otherwise mutually exclusive.

Meneme designators
The following *meneme designators* are used in the Lean Cuisine+ notation.

{ } *Virtual meneme:* a non-selectable meneme which is used to partition a dialogue or subdialogue. The meneme name appears between braces.

⊥ *Monostable meneme:* a meneme which, following user selection, reverts to an unselected state on completion of the simple modal subdialogue it represents, through system action (exceptionally following some user intervention).

↑ *Select-only meneme:* a meneme of this type cannot be directly deselected by the user, but may be deselected through selection of another mutually exclusive option.

↓ *Deselect-only meneme:* a meneme of this type cannot be directly selected by the user, but may become selected as an indirect result of some other action.

⊗ *Passive meneme:* a meneme of this type is unavailable for selection or deselection by the user, and can only be selected by the system.

* *Default choice:* indicates a meneme that is to be in the selected state at the commencement of a dialogue or subdialogue.

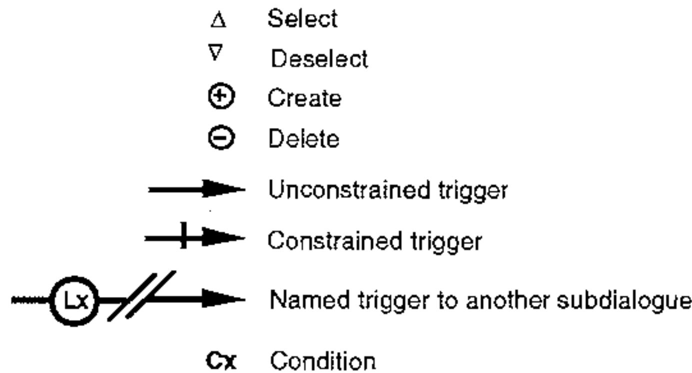
§ *Required choice:* is associated with a subdialogue or snbgroup header, and indicates a meneme group from which a valid choice is always required.

• *Dynamic default:* may be associated with a meneme (assigned) or a meneme group (unassigned), and indicates a default choice which takes on the value of the last user selection from that group.

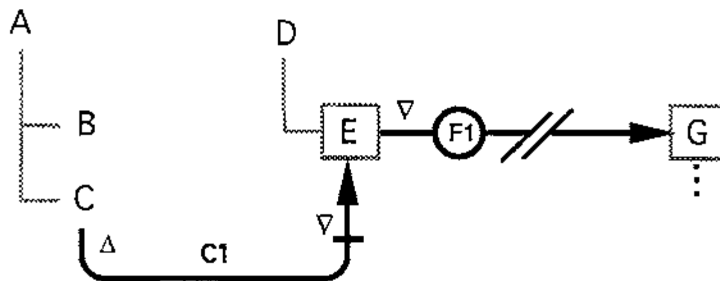
C2. Additional Constraints

Additional constraints, in the form of *selection triggers*, *option preconditions*, and *existence dependencies*, are captured in three further layers which are superimposed on the base layer dialogue tree diagram.

Selection triggers
A selection trigger is a directed link between menemes such that the selection or deselection of a meneme by the user triggers (possibly conditionally) the selection or deselection of one or more other menemes in the same or different dialogues (with concomitant selection of dialogue and/or subdialogue headers as and if necessary). A selection trigger represents a system response to a user action. Object instances may be created or deleted by selection triggers, which are annotated as shown below. An example appears in Figure (r):

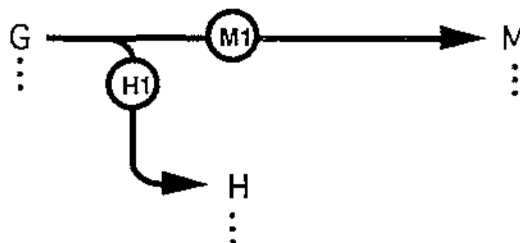


Triggers are either *unconstrained*, in which case further 'knock on' effects are possible, or *constrained*, in which case the behaviour of the selected or deselected object is modified, inhibiting further links, and preventing side effects such as cycles.



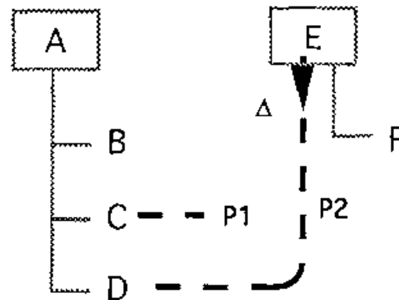
(r) Selection (Δ) of C causes deselection (∇) of E provided condition C1 is true. The selection of G is not triggered because the first trigger is constrained.

A separate top level *Dialogue Interlink Diagram* (DID) showing dialogue root nodes, and the (named) triggers between them, can also be constructed, as shown below.



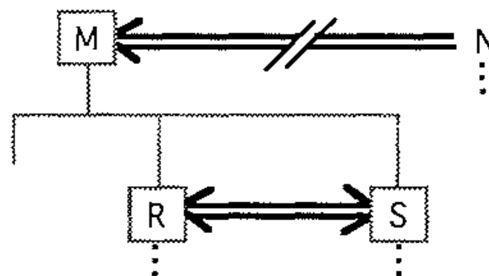
(s) DID diagram showing triggers between dialogue G, and H and M.

Option preconditions
An option precondition is a precondition for the availability (for selection) of a meneme in the Lean Cuisine+ notation, which may involve the state of a meneme in another part of a dialogue, or the existence of some other state or condition, or both. An option precondition is indirectly an event precondition.



(t) D is available for selection when E is selected (Δ) and condition P2 is true.
C is available for selection when condition P1 is true.

Existence dependencies
The existence of an object may depend upon the existence of another object in the same or different subdialogue, or upon the existence of a dialogue. The dependency may be mutual. Examples are shown below.



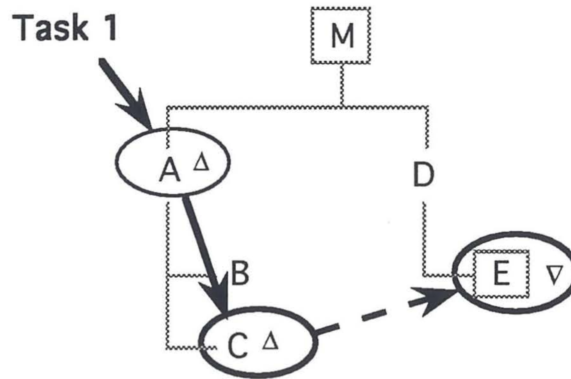
(u) Object M exists only if dialogue N exists.
Objects R and S are mutually dependent.

C3. The task layer

The *task layer* provides for the description of higher level user tasks in terms of primitive actions (and associated events), capturing any temporal relationships between them. Sequences or groups of task actions relating to a higher level task are overlayed on the base layer dialogue tree diagram. The task layer is orthogonal to the constraint-based behavioural interface model described in Sections 1 and 2 above.

The notation distinguishes user initiated and system initiated events by means of the arcs connecting selections. A plain arc between two successive selections denotes that the new selection is the result of a user action, whereas a dashed arc indicates that the new selection is a system response triggered by a user action. Selection and deselection actions are indicated as defined above for selection triggers.

The example below shows a task action *sequence* involving two user selections and one resultant system deselection. The link which triggers the system deselection is defined in Figure (r) above. The entry point is labelled with the task name. Where there is an absence of sequence, and it is only the *grouping* of selection actions which is significant, no entry point or links are shown.



(v) Task 1 consists of user selections A then C, followed by system deselection of E (triggered by the selection of C).

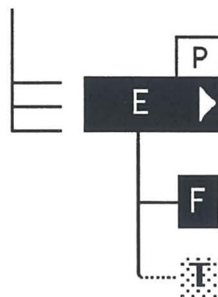
C4. Execution of a Lean Cuisine+ specification

A Lean Cuisine+ dialogue tree diagram can be 'marked' to show its state at a given point in a dialogue in terms of meneme states. Menemes can be shown as being either selected or unselected, and either available (for selection) or unavailable. This provides for the 'execution' of a Lean Cuisine+ specification - that is of simulating dynamically the behaviour of the dialogue as selections are made. This takes cognisance of all specified constraints, including selection triggers and option preconditions.

The four meneme states are delineated on the dialogue tree as follows. Examples are shown in Figure (w) below:

unselected and available:	normal presentation
unselected and unavailable:	greyed normal presentation
selected and available:	reverse video
selected and unavailable:	greyed reverse video.

Two further visual features are associated with generic objects during execution, as shown in Figure (w). A *tab* displaying the name of the current object instantiation appears above the object box, and a *right arrow* appears in the box to provide access to current instantiations (the exact nature of the mechanism is implementation dependent).



(w) Instance P of generic object E, and option F, are selected and available.
Option T is unselected and is also unavailable for selection.
Other instances of E are available for selection via the right arrow.

Appendix D

A Support Environment Prototype

This appendix presents a Hypercard prototype of part of a support environment for Lean Cuisine+. The displayed diagrams relate to the Finder document folder system. The simulation provides for:

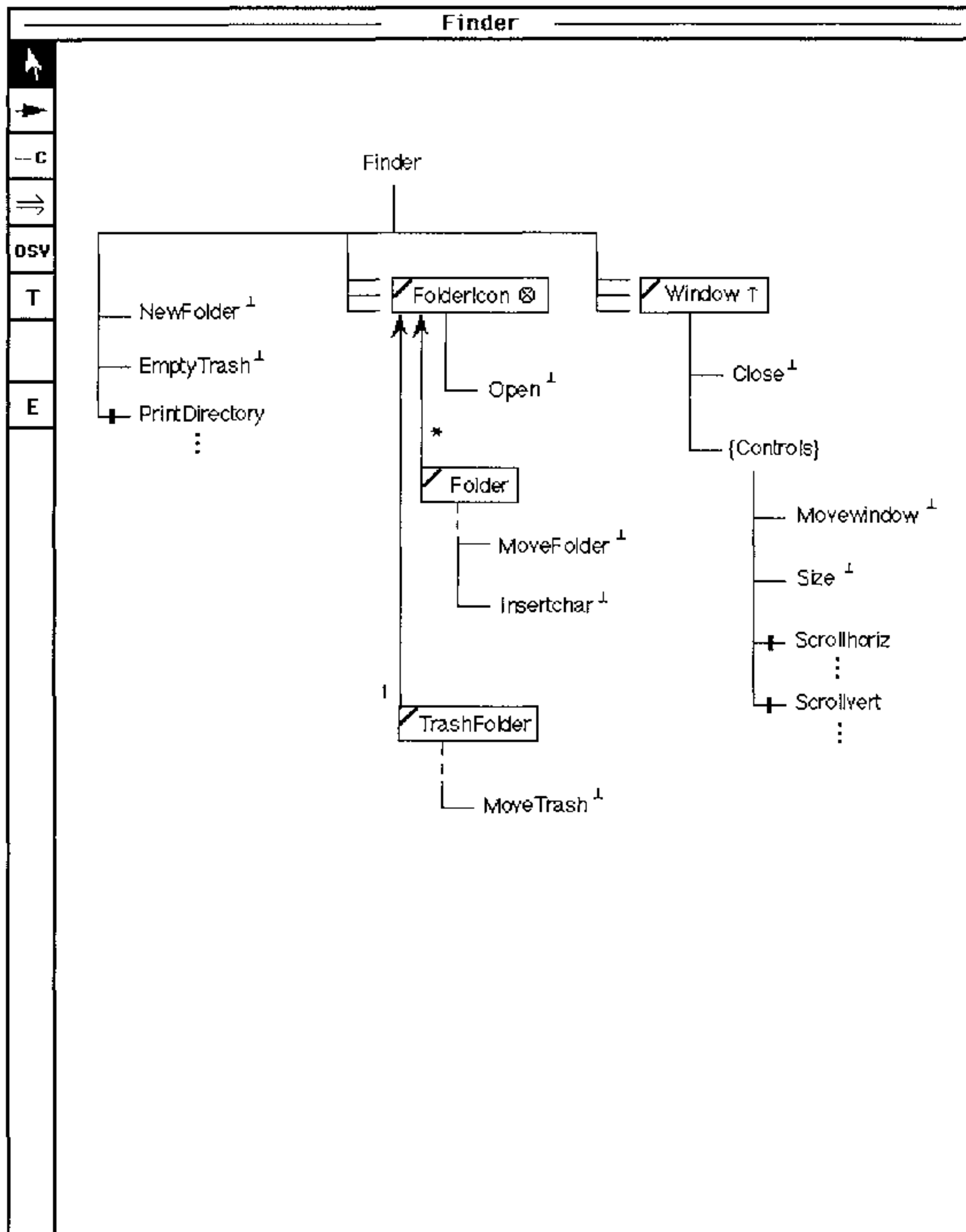
- the browsing of parts of the Lean Cuisine+ description of this system, including dialogue and subdialogue tree diagrams and constraint overlays, and
- an execution of the specification, based on a specific sequence of actions.

The support environment is displayed in a window offering a palette of seven selectable options, each relating to a particular mode within the environment. A mode is selected by 'pointing and clicking' on the corresponding icon, which remains highlighted until another mode is selected. The functionality of the prototype environment is described in Section 8.2, and the window format is shown in Figure 8.1. The simulation is presented in seven parts, each relating to a particular mode:

D1	The browsing of Lean Cuisine+ dialogue and subdialogue trees.
D2 - D4	The switching in and out of further constraints in the form respectively of selection triggers, option preconditions, and existence dependencies.
D5	The display of the state variables relating to specified objects or operations.
D6	The display of task action sequences.
D7	The execution of the specification.

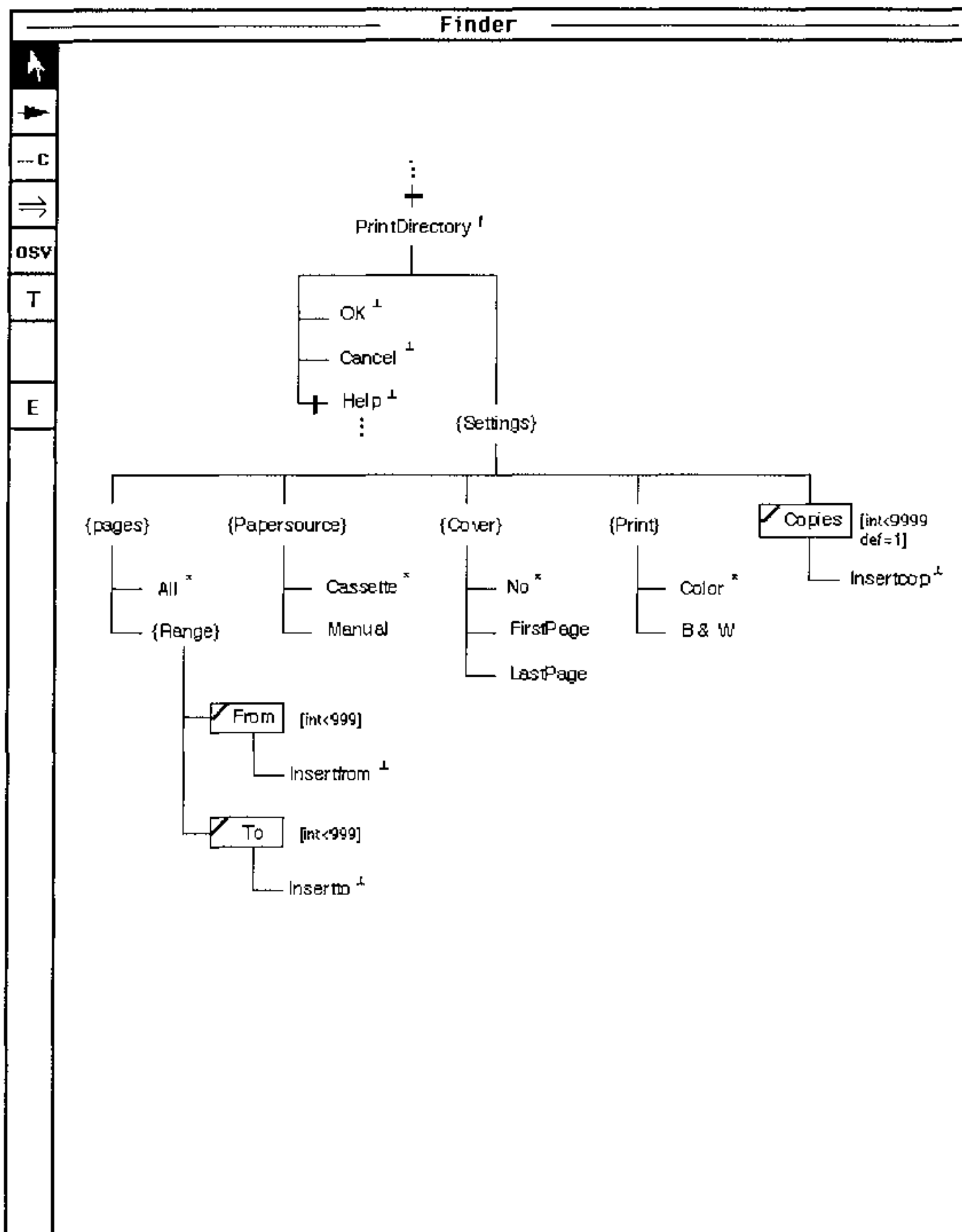
D1. Basic browsing mode

This mode provides for exploration of the hierarchy of dialogue and subdialogue trees which form the base layer of a Lean Cuisine+ specification. Pointing and clicking on a meneme carrying an ellipsis causes the corresponding subdialogue tree to be displayed in the window. The window below displays the first level tree diagram for the Finder document folder system.



Basic browsing mode: Finder base layer: first level tree

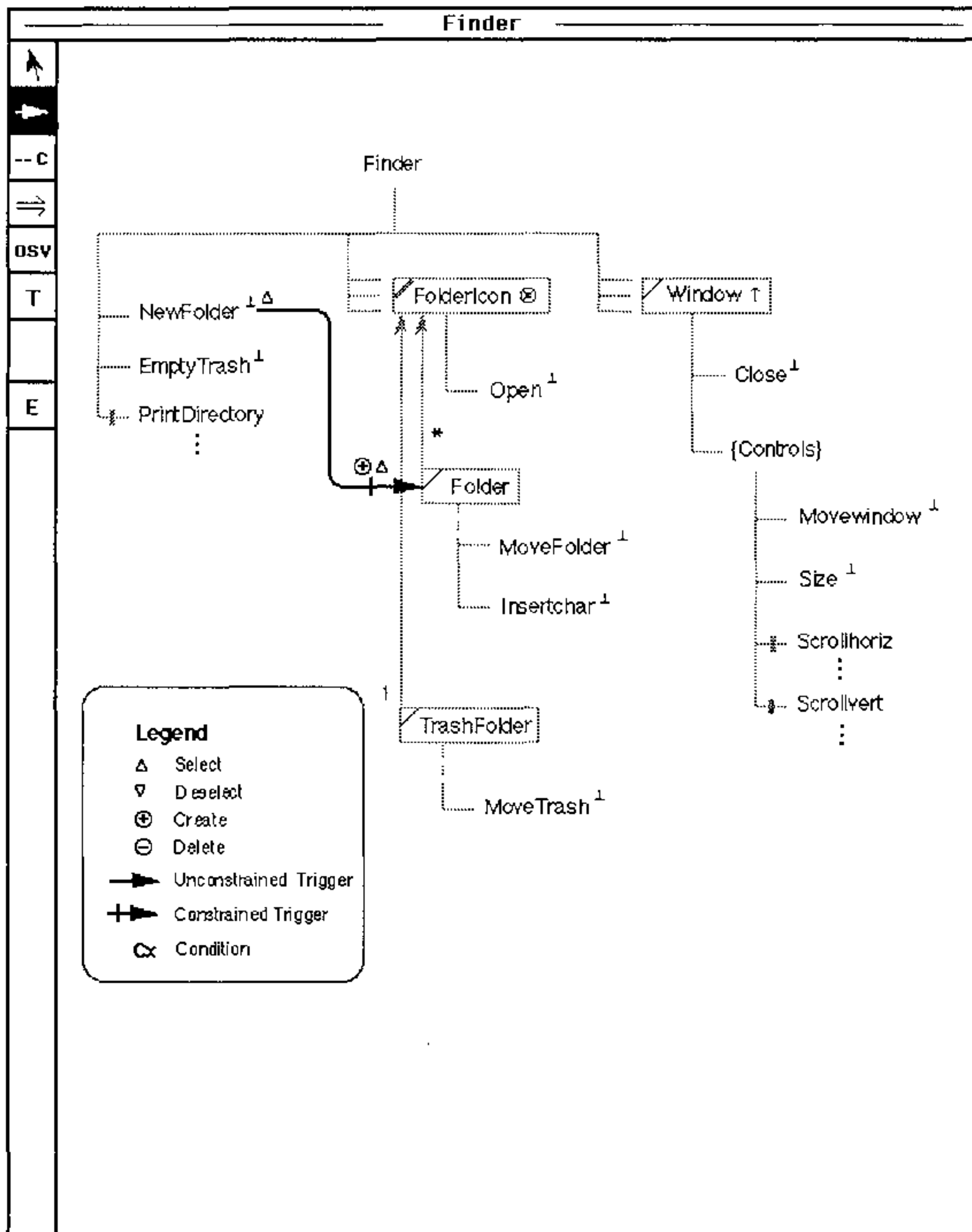
In the window below, the tree diagram for the *PrintDirectory* subdialogue has been displayed following 'selection' of the subdialogue header in the first level diagram. Return to the first level diagram is achieved by pointing and clicking on the subdialogue header meneme.



Basic browsing mode: Finder base layer: *PrintDirectory* subdialogue tree

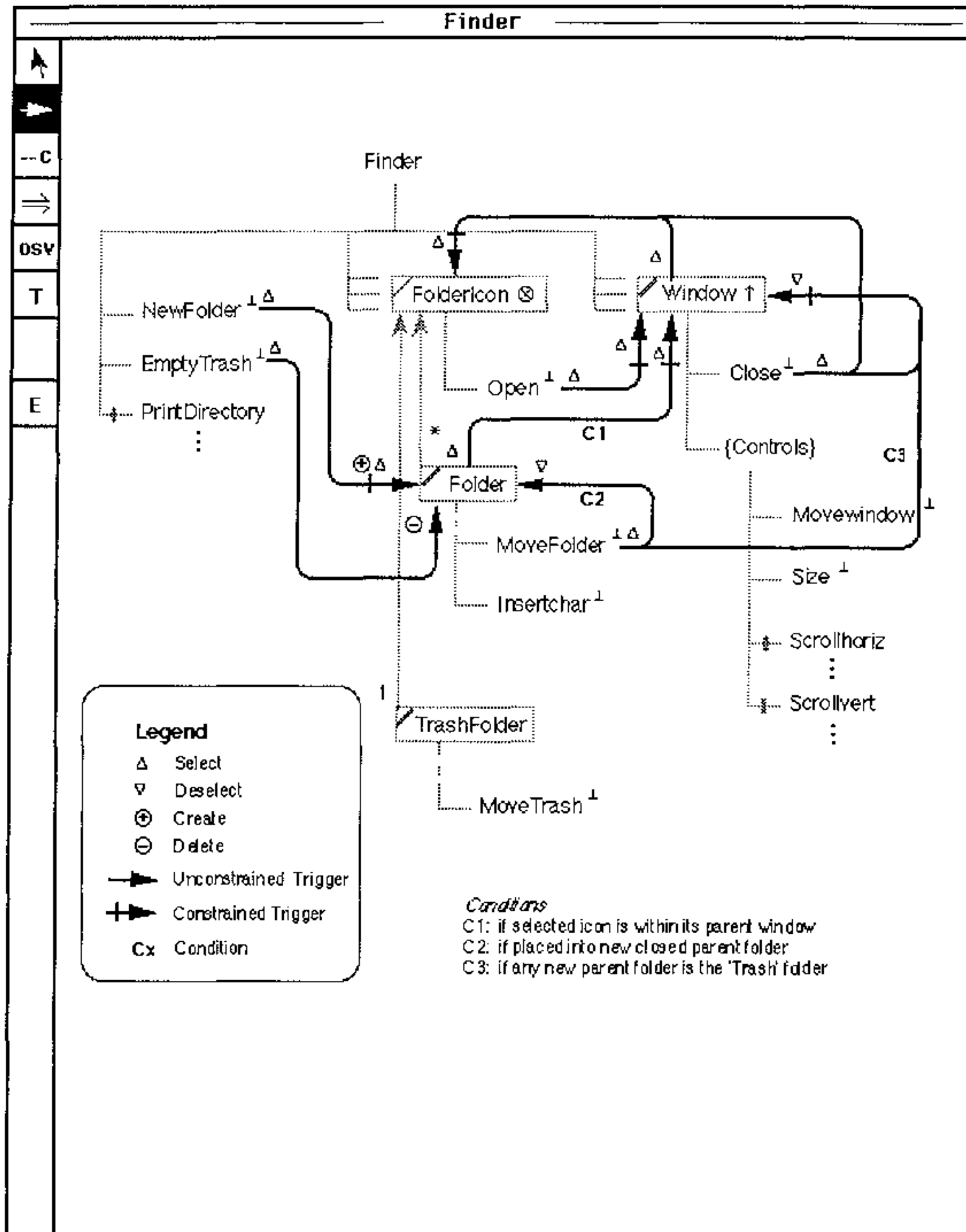
D2. Selection trigger mode

In the window below, the selection trigger relating to the *NewFolder* command has been displayed over a greyed base layer tree diagram, by selecting the mode and then pointing and clicking on the *NewFolder* meneme. A legend is displayed with the diagram.



Selection trigger mode: display of trigger for *NewFolder*

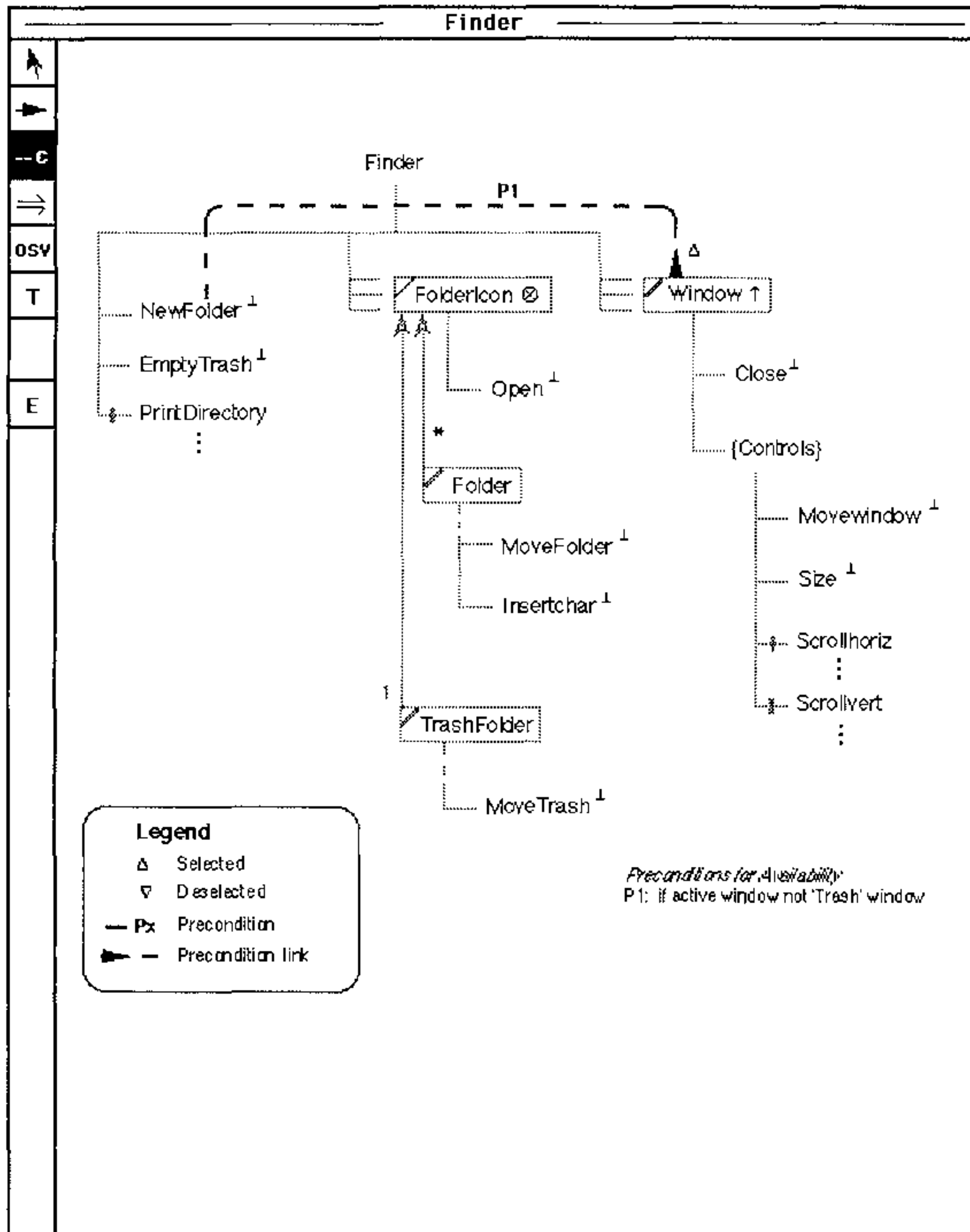
In the window below, *all* triggers and associated conditions relating to the Finder dialogue have been displayed, by pointing and clicking on the already selected modal icon in the palette.



Selection trigger mode: display of *all* triggers

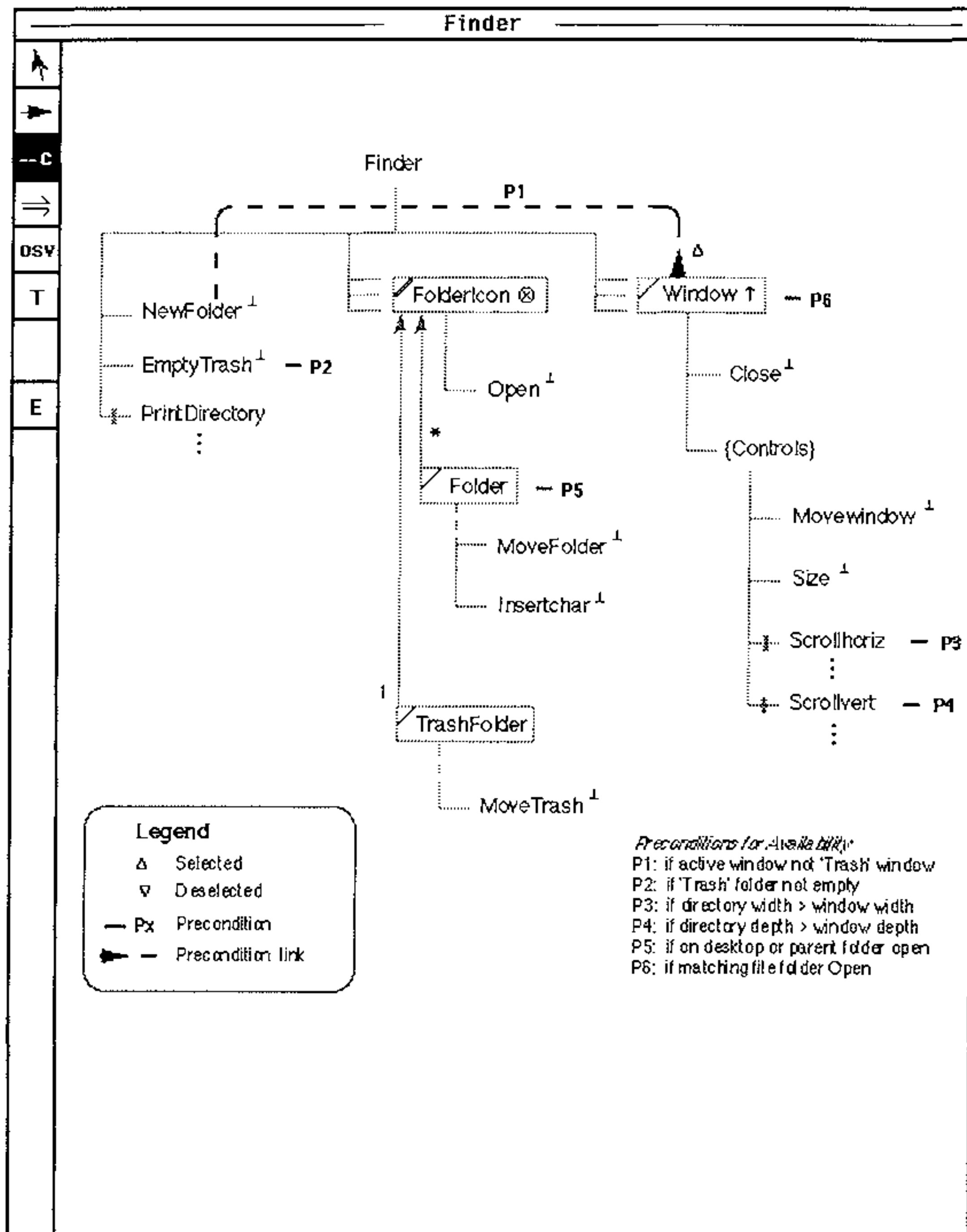
D3: Option precondition mode

In the window below, the option precondition relating to the *NewFolder* command has been displayed over a greyed base layer tree diagram, by selecting the mode and then pointing and clicking on the *NewFolder* meneme. A legend is displayed with the diagram.



Option precondition mode: display of precondition for *NewFolder*

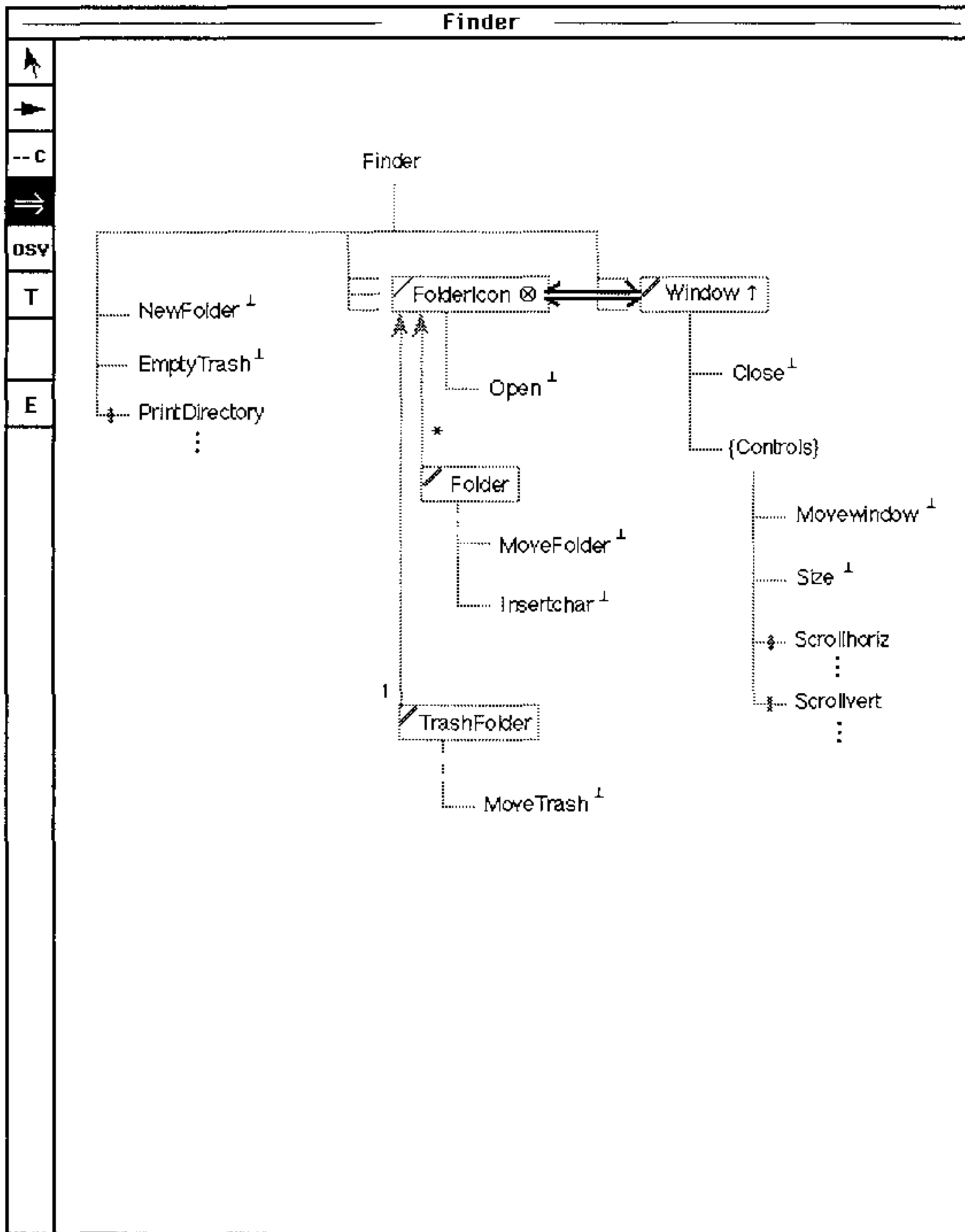
In the window below, *all* option availability preconditions relating to the Finder dialogue have been displayed, by pointing and clicking on the already selected modal icon in the palette.



Option precondition mode: display of *all* preconditions

D4: Existence dependency mode

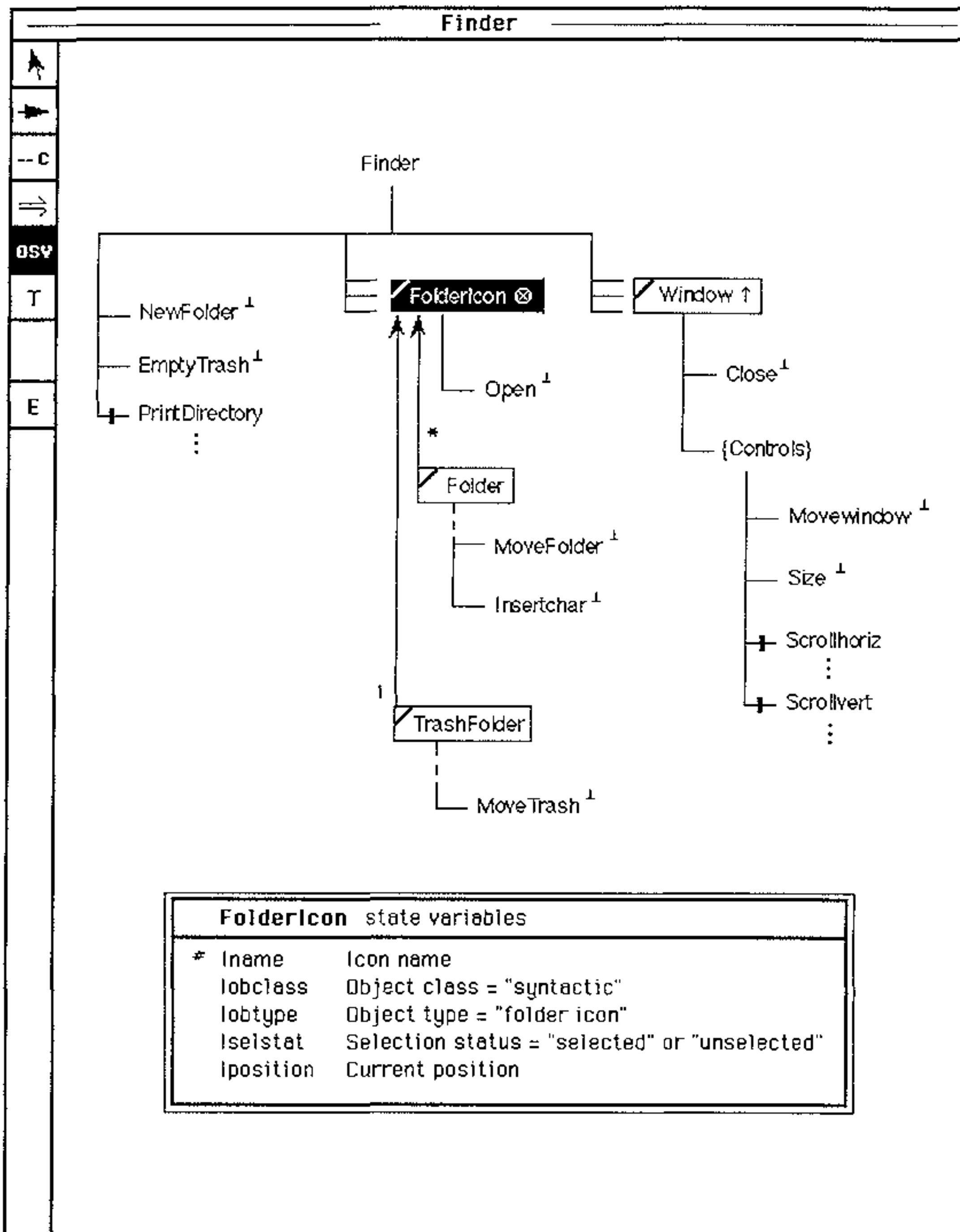
In the window below, the existence dependency relating to the *FolderIcon* object has been displayed over a greyed base layer tree diagram, by selecting the mode and then pointing and clicking on the *FolderIcon* meneme. Although this is the only example in the Finder dialogue, an *all* option, actioned in the manner described above for the other constraints, would be generally available.



Existence dependency mode: for the *FolderIcon* object

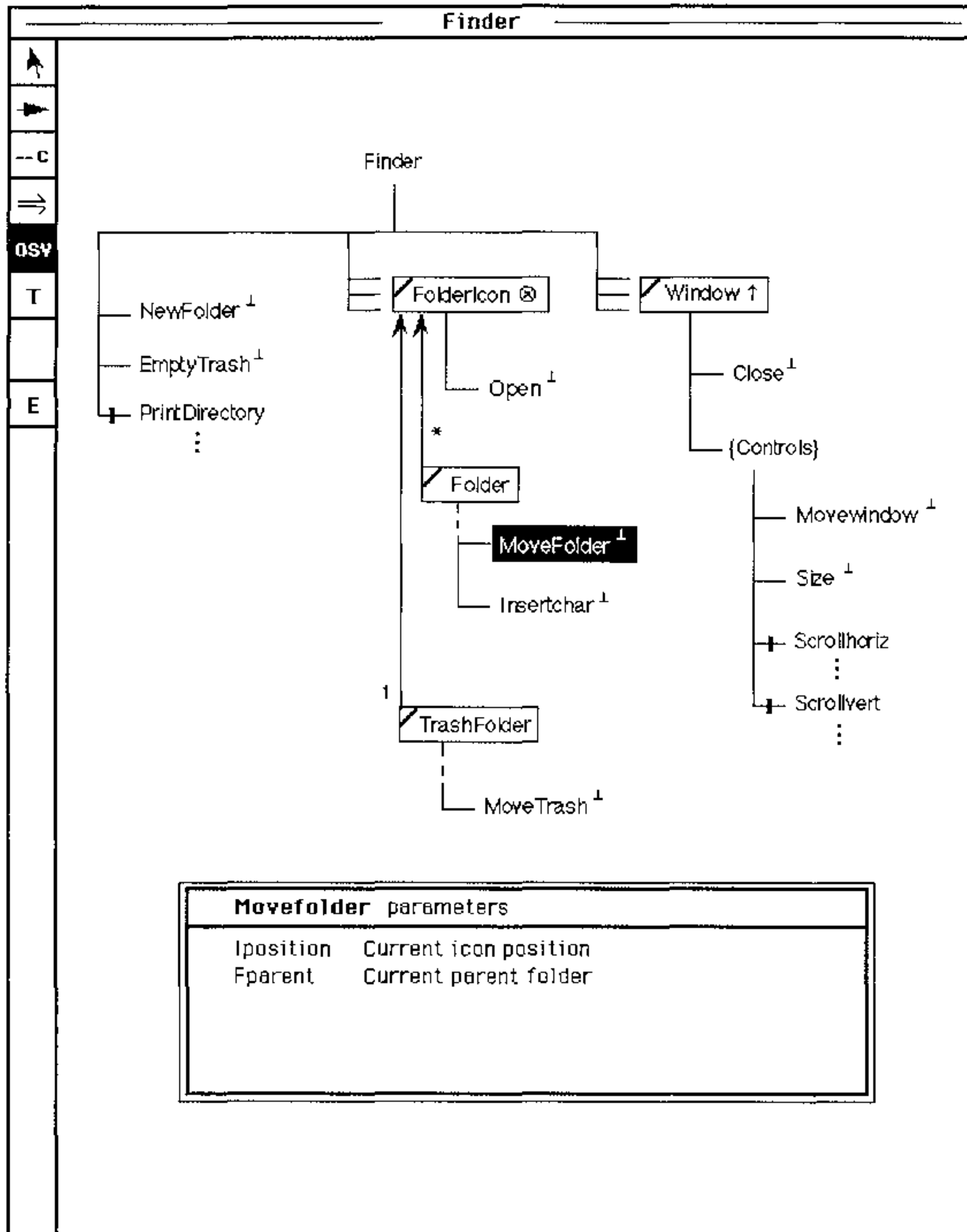
D5: Object state variable (OSV) mode

In the window below, the state variables relating to the *FolderIcon* object have been displayed in a separate overlaid object window, by selecting the mode and then pointing and clicking on the *FolderIcon* meneme. The window can be closed by pointing and clicking again on the highlighted object, or by selecting another object.



OSV mode: for the *FolderIcon* object

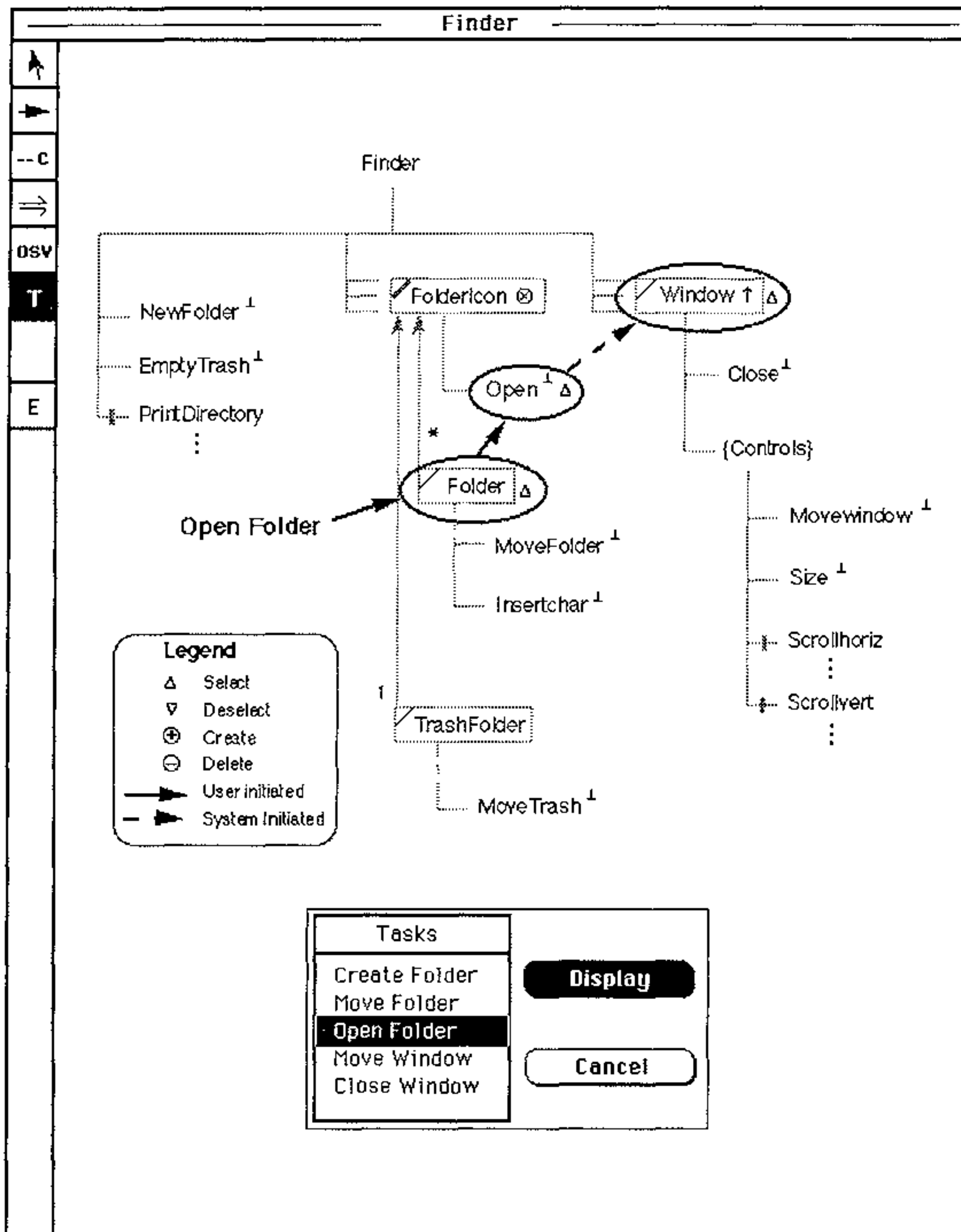
The state variables which may be modified as a result of selecting an operation, and which are parameters to the operation, can also be displayed by pointing and clicking on the meneme representing the operation. In the window below, the parameters relating to *MoveFolder* have been displayed.



OSV mode: for the *MoveFolder* operation

D6: Task mode

In the window below, the sequence of primitive actions relating to the *Open Folder* task has been displayed over a greyed base layer tree diagram, by selecting the mode, choosing the *Open Folder* task from the menu displayed in the overlaid task window, and then choosing the 'Display' option. The task action sequence can be hidden by selecting the 'Cancel' option.



Task mode: for the *Open Folder* task

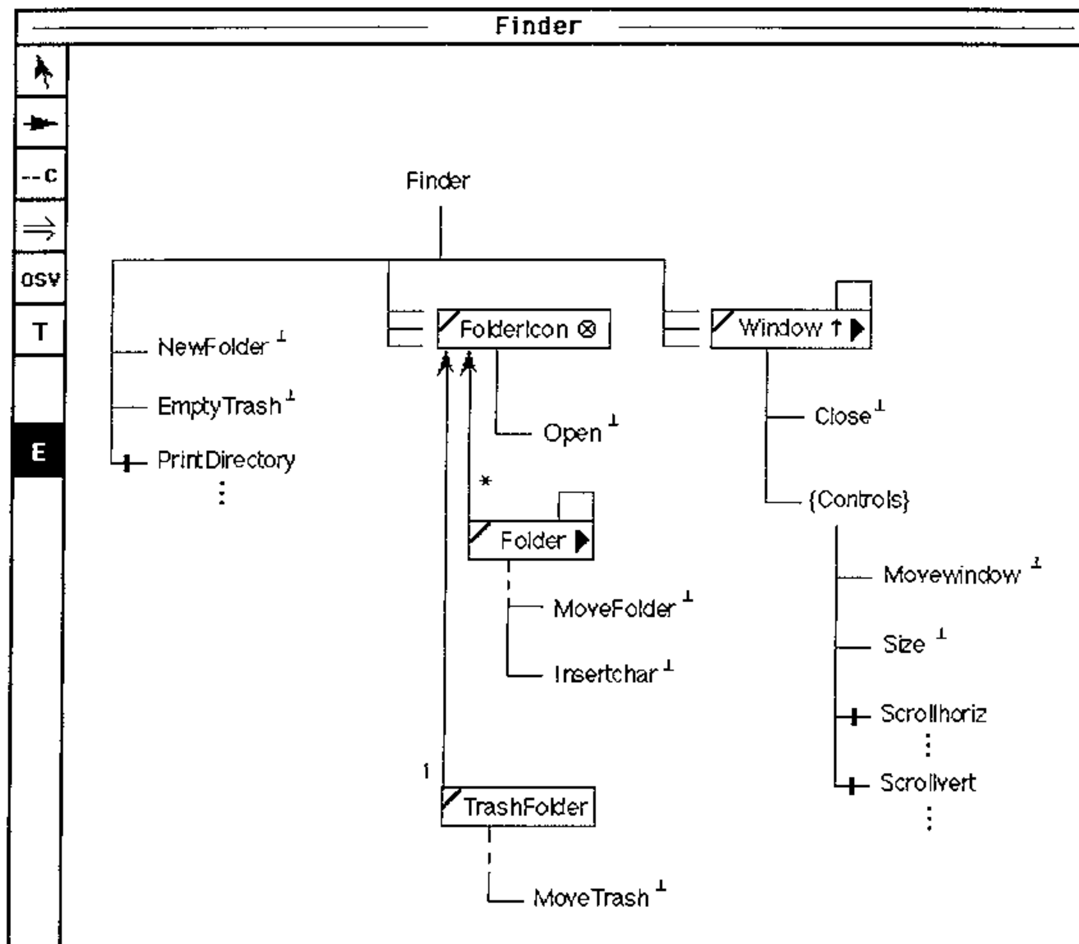
D7: Execution mode

The diagrams which follow relate to the execution sequence shown in Table D1, which describes the behaviour of the small system of icons and windows defined in Section 8.2. Given the repetitive nature of the sequence, diagrams are omitted for some stages. Brief explanatory notes are included with each diagram.

Stage	Operation or State	Folder Icon Set	Trash Icon	Window Stack
#0	Start-up Finder			
#1	At start-up	A B C	T	
#2	Select folder B	A B C	T	
	Folder B selected	A B C	T	
#3	Open folder B	A B C	T	
	Window B open	A B C F G	T	B
#4	Select folder C	A B C F G	T	B
	Folder C selected	A B C F G	T	B
#5	Open folder C	A B C F G	T	B
	Window C open	A B C F G M N	T	C B
#6	Select folder F	A B C F G M N	T	C B
	Folder F selected	A B C F G M N	T	B C
#7	Open folder F	A B C F G M N	T	B C
	Window F open	A B C F G K M N	T	F B C
#8	Folder F deselected	A B C F G K M N	T	F B C
#9	Select window B	A B C F G K M N	T	F B C
	Window B active	A B C F G K M N	T	B F C
#10	Create new folder in B	A B C F G K M N	T	B F C
	New folder # created	A B C F G K M N #	T	B F C
#11	Rename new folder as D	A B C F G K M N #	T	B F C
	Folder renamed D	A B C D F G K M N	T	B F C
#12	Move folder D to Trash	A B C D F G K M N	T	B F C
	Folder D in Trash	A B C D F G K M N	T	B F C
#13	Trash folder selected	A B C D F G K M N	T	B F C
#14	Open Trash folder	A B C D F G K M N	T	B F C
	Trash window open	A B C D F G K M N	T	T B F C
#15	Empty Trash (of D)	A B C D F G K M N	T	T B F C
	Trash folder emptied	A B C F G K M N	T	T B F C
#16	Close Trash window	A B C F G K M N	T	T B F C
	Trash window closed	A B C F G K M N	T	B F C
#17	Close window B	A B C F G K M N	T	B F C
	Window B closed	A B C K M N	T	F C
#18	Close window F	A B C K M N	T	F C
	Window F closed	A B C M N	T	C
#19	Close window C	A B C M N	T	C
	All windows closed	A B C	T	
#20	Shut down Finder	A B C	T	

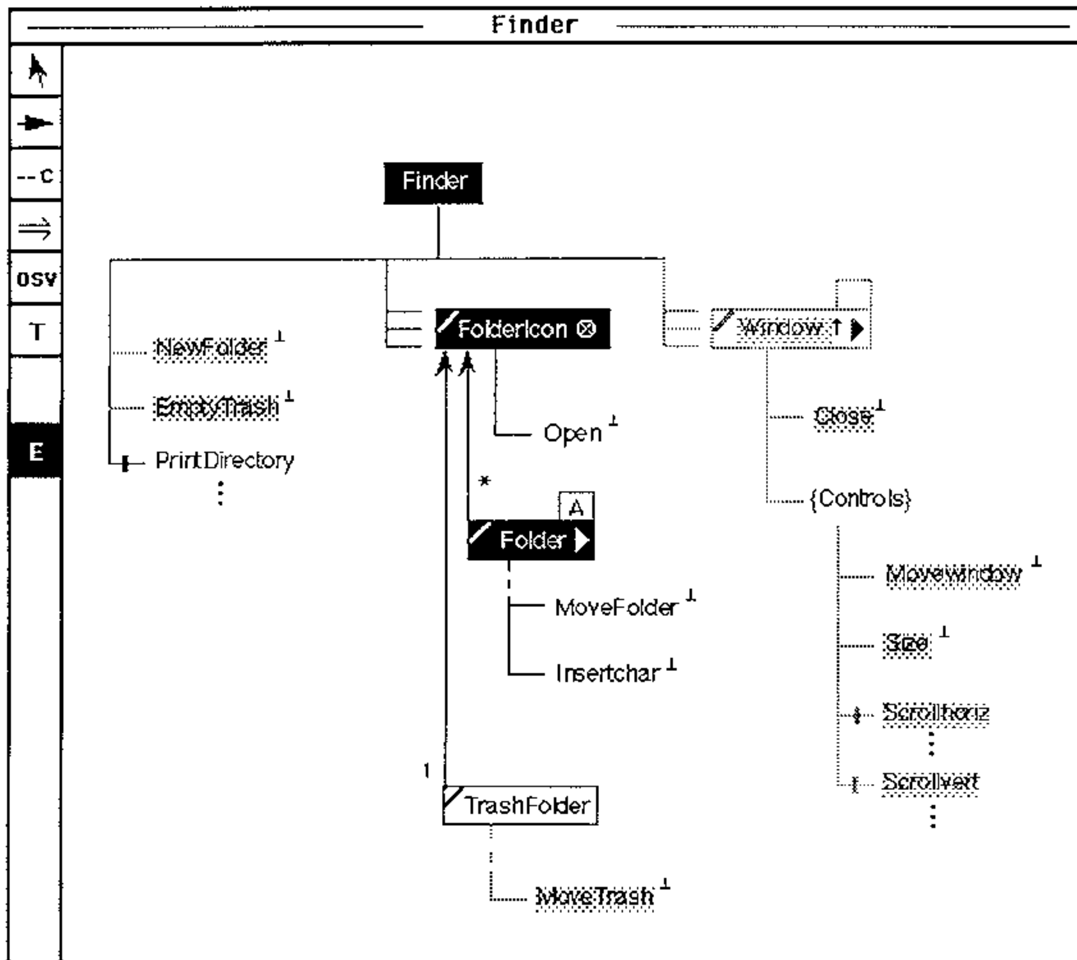
Table D1: Icon set and window stack at each stage of the simulation

Notes: 1. Currently selected instances are shown bold underlined at each stage.
2. # is the default new folder name.



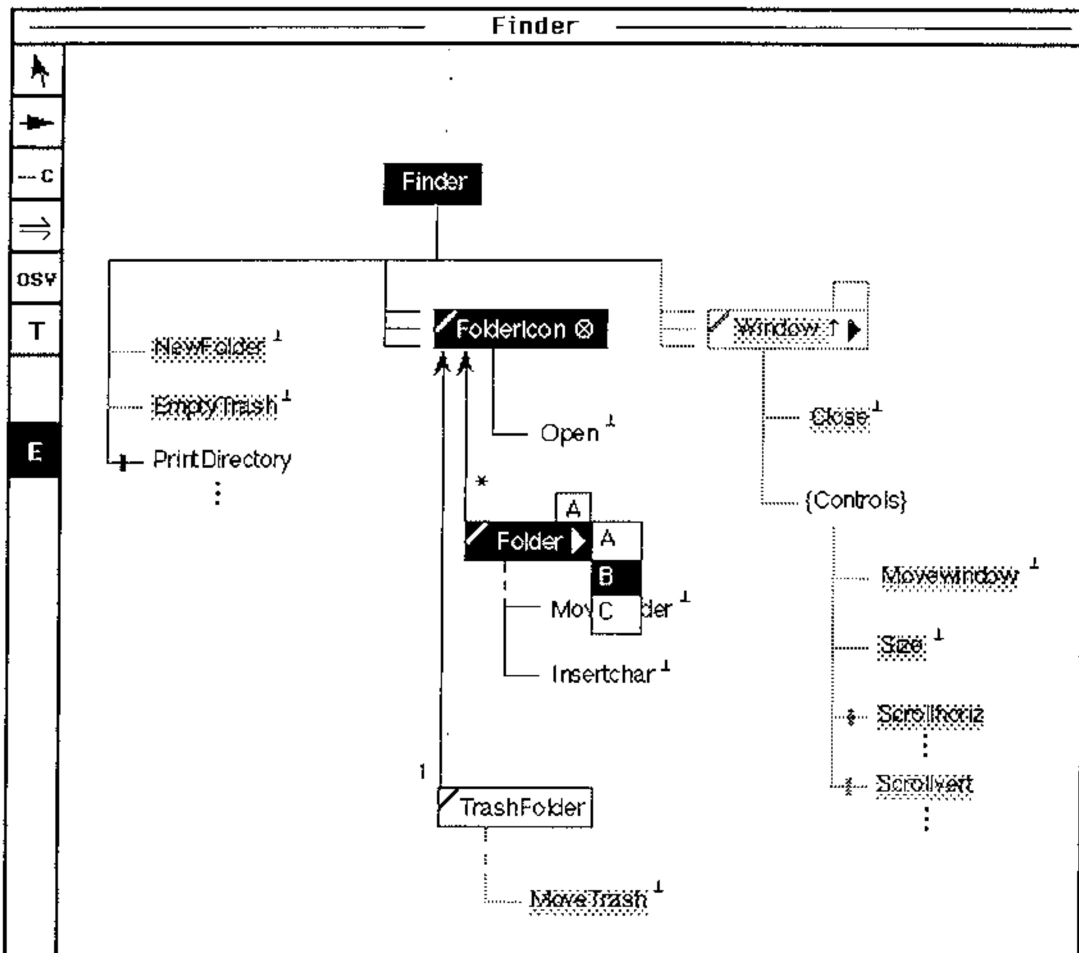
Stage #0: Execution window prior to start-up

- This diagram shows the execution mode window prior to starting the session.
- In execution mode, 'tabs' appear above generic objects. They are used to display at any time the name of the currently selected object instance, if any. The arrows to the right of generic object boxes provide at any juncture a means of accessing, and selecting from, the current instance set.
- Throughout the execution, available menemes are displayed normally, unavailable menemes are greyed, and selected menemes are shown in reverse video. Monostable menemes such as *Open* remain selected only until the operation they represent is completed, when they revert to an unselected state.



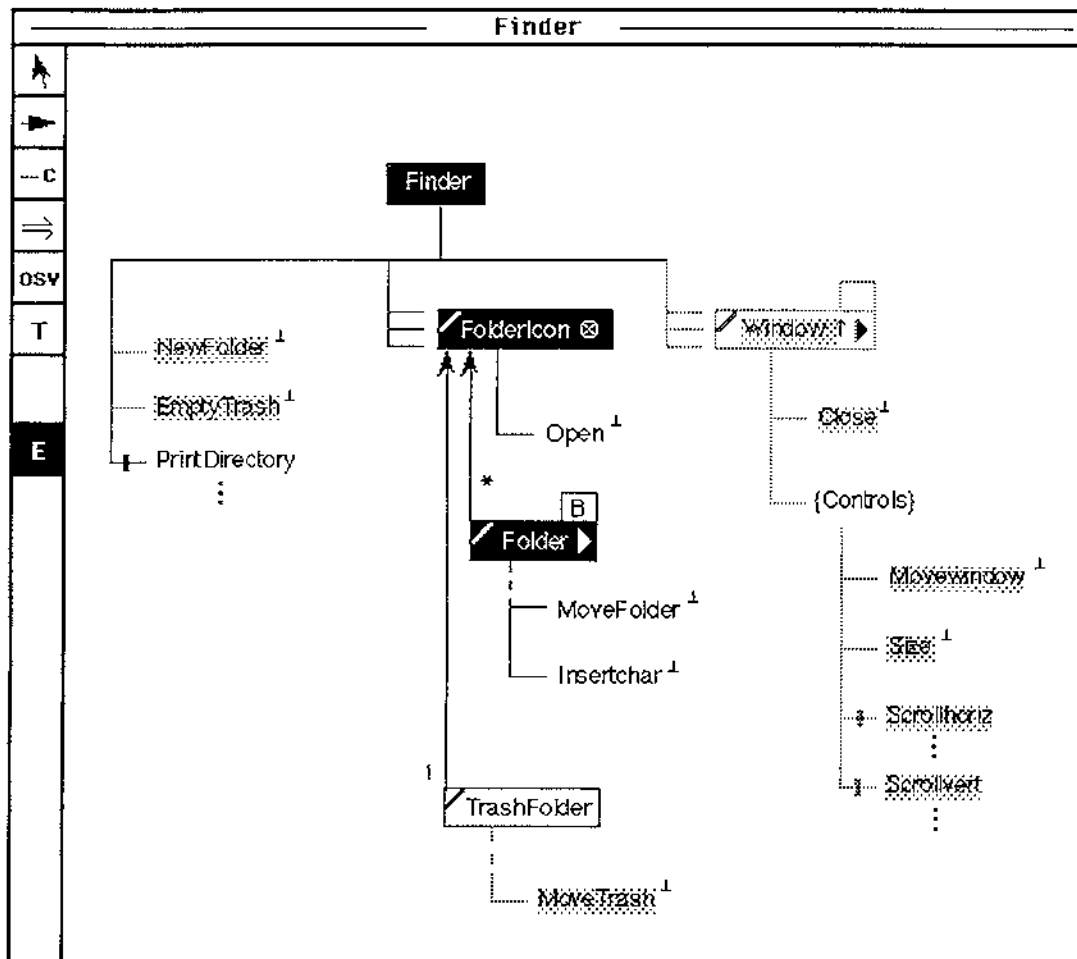
Stage #1: Start-up state

- Selecting the *Finder* meneme has produced the state shown, with three menemes initially selected. The *Finder* dialogue header remains selected throughout.
- The system (disk) icon, *A*, is shown as the initial default folder choice, and is represented by the *FolderIcon/Folder* meneme pair, both of which are in a selected state.
- At this stage, *PrintDirectory* is available in the root dialogue, and *Open*, *MoveFolder*, or *Insertchar* could be selected for application to folder *A*.
- The entire *Window* subtree is unavailable and therefore greyed (it has been assumed that no windows were left open at the last shutdown), as are the *NewFolder* and *EmptyTrash* options at the root level, and *MoveTrash* in the *TrashFolder* subdialogue.



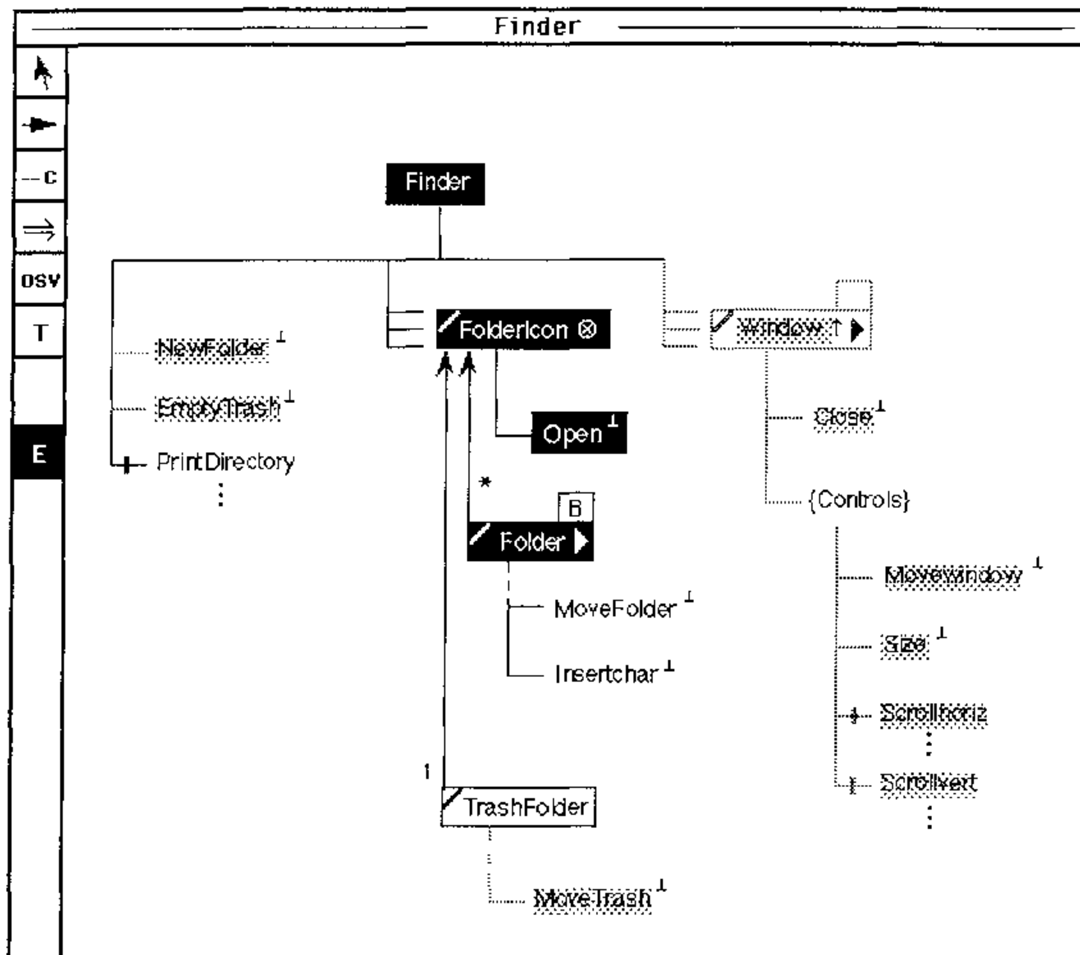
Stage #2: Selection of Folder *B*

- Folder *B* is about to be selected from the pull-down menu accessed via the right arrow in the *Folder* object box. The icon subset is displayed in alphabetical order at each level in the folder hierarchy.



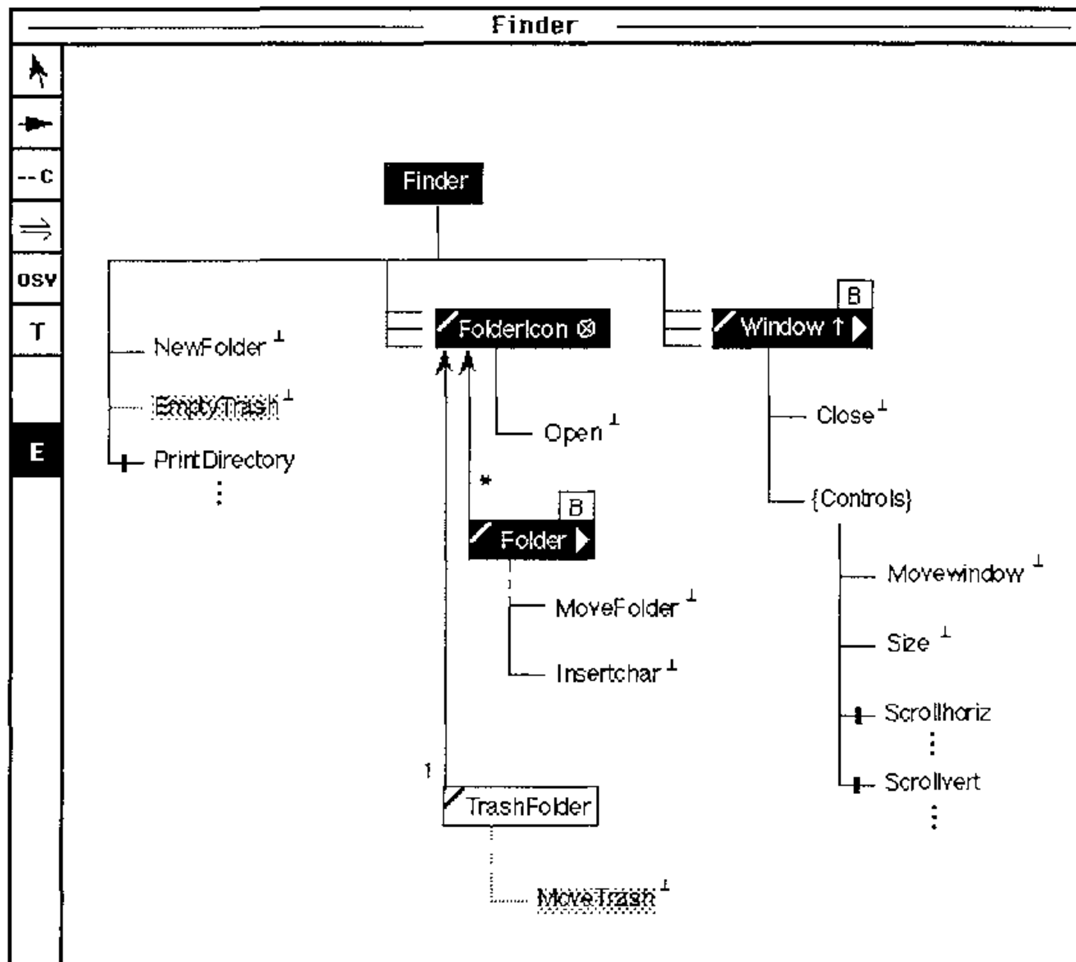
Stage #2: Folder *B* selected

- Folder *B* has been selected, and *B* has replaced *A* as the current folder instance in the tab above the *Folder* object box.



Stage #3: Opening of folder *B*

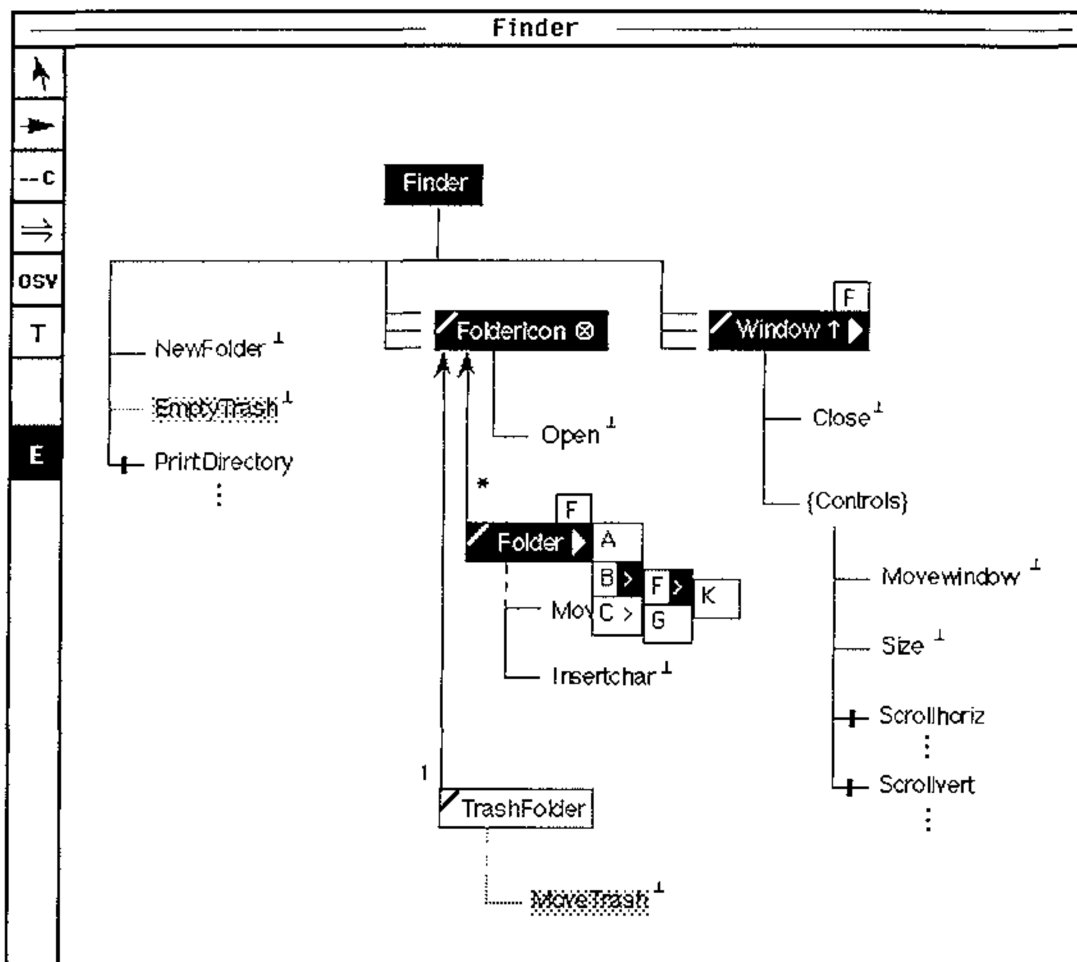
- *Open* has been selected for the current folder instance, *B*. This operation, which is common to all folders, is offered at the *FolderIcon* supertype level.
- *Open* will remain selected only until the operation is completed, when it will revert to an unselected state.



Stage #3: Window *B* open

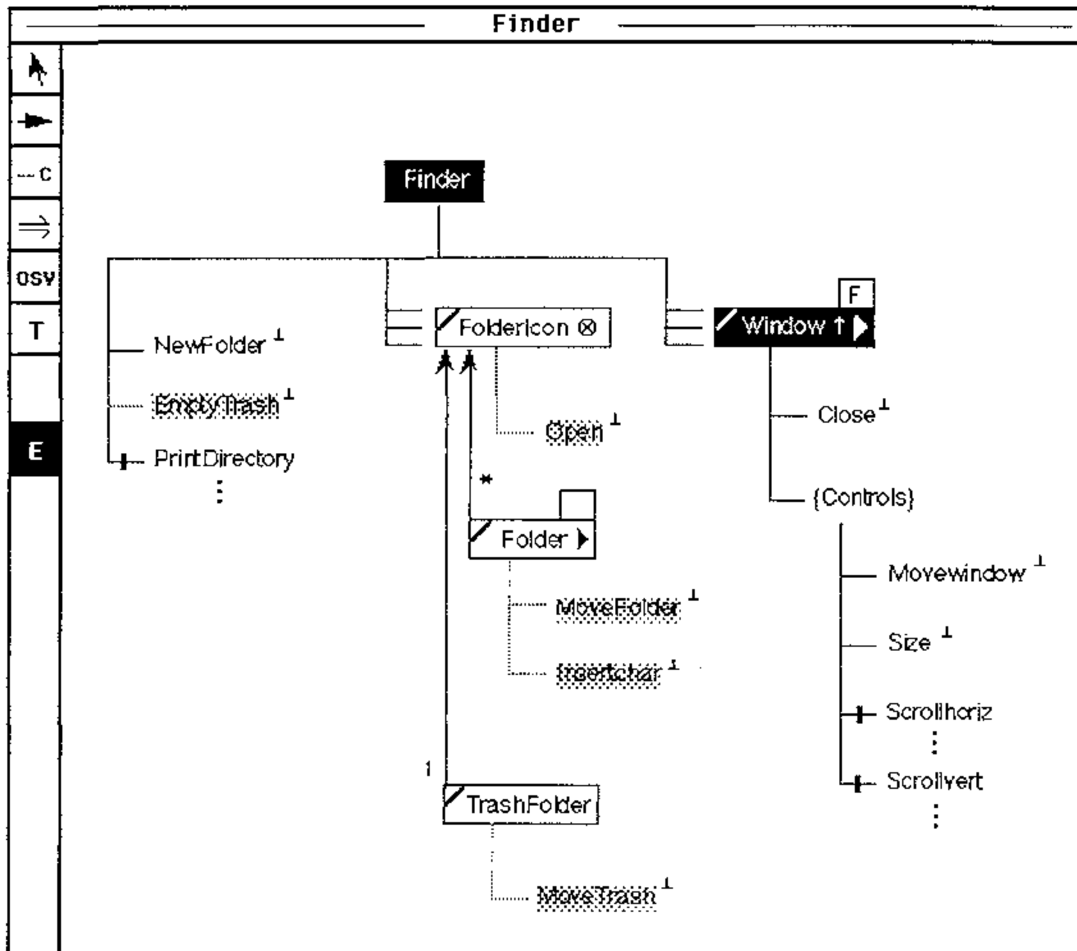
- Window *B* has become selected via a selection trigger as a result of opening folder *B*. *B* is displayed in the *Window* object tab, to denote the currently active instance.
- Following selection of a window instance, the *Window* subdialogue has become available, and is shown ungreyed. The precondition for availability of the *NewFolder* command at the root dialogue level has now been satisfied, and this option has also been ungreyed.

Stages #4 to #7 of the simulation involve similar operations for folders *C* and *F* to those described in stages #2 and #3 above for folder *B*, that is, they are selected and opened. The selection of folder *F* (Stage #6) causes its parent window *B* to become selected. The subsequent opening of folder *F* (Stage #7) causes window *B* to be deselected again.



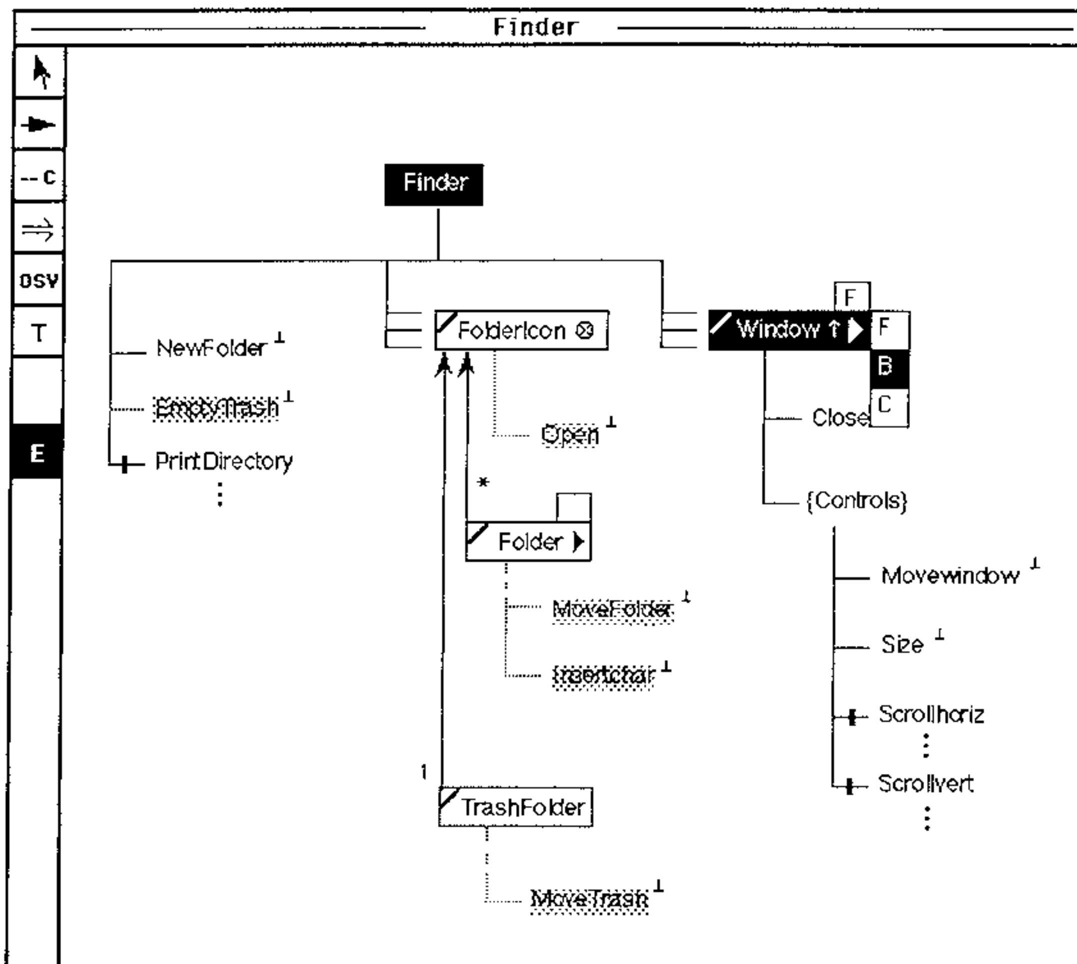
Stage #7: Window *F* open

- The state of the dialogue following the opening of window *F* is shown above.
- Additionally, the pull-down menus relating to the selection of folder instances are shown displayed. The hierarchical parent-child folder structure is clearly visible. Open folders are denoted by the presence of a right arrow providing for the display of their children.



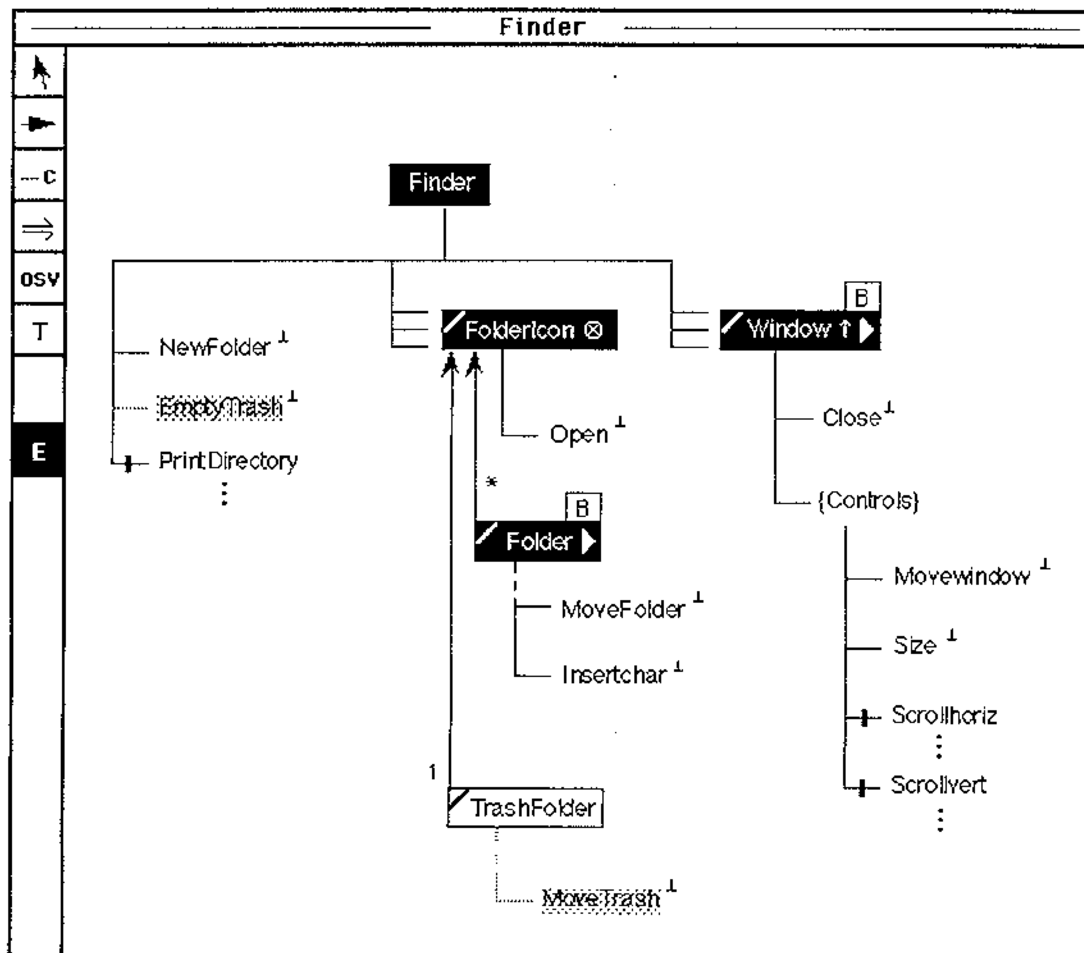
Stage #8: Folder *F* deselected

- Folder *F* has been deselected, also causing *FolderIcon* to be deselected through propagation up the type hierarchy. The subdialogues relating to the deselected object are now unavailable at both levels in the type hierarchy, and are shown greyed.



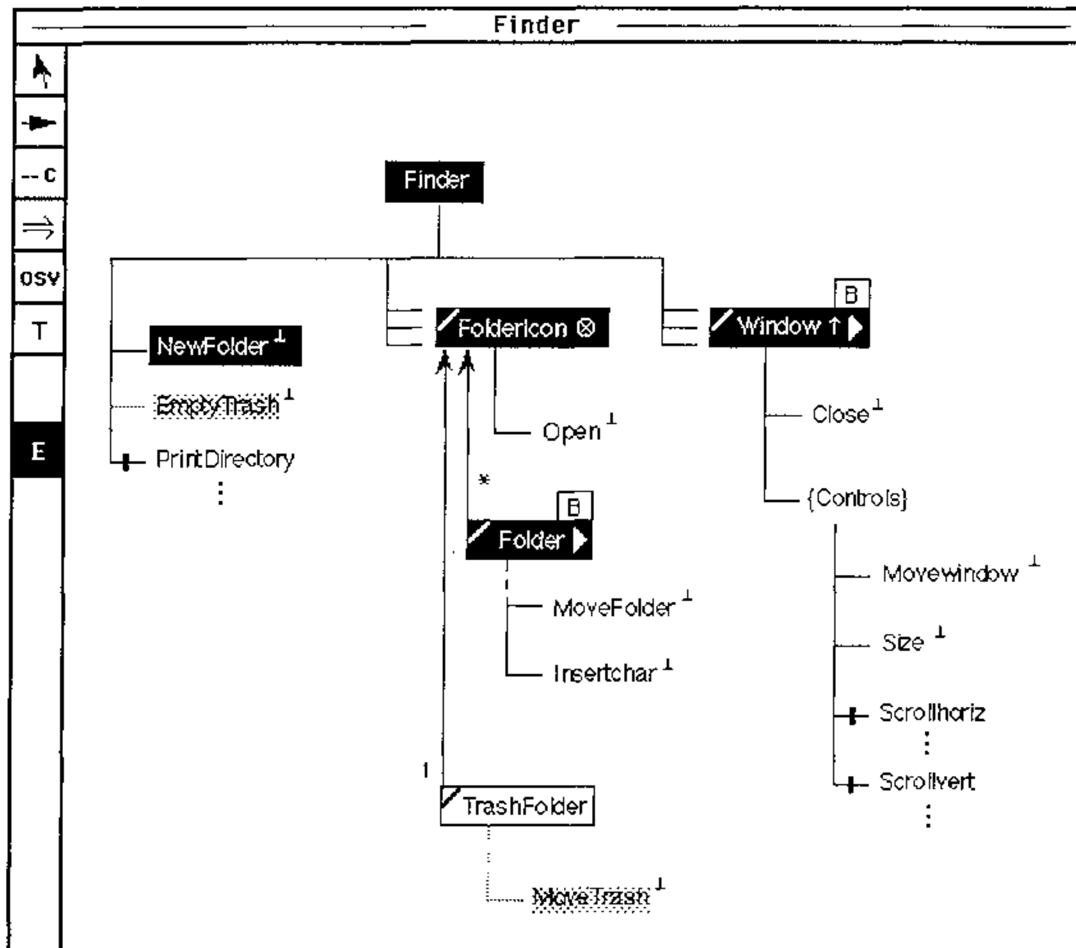
Stage #9: Selection of window *B*

- Window *B* is about to be selected from the pull-down menu accessed via the right arrow in the *Window* object box. The window names are displayed in stack order from the top down.



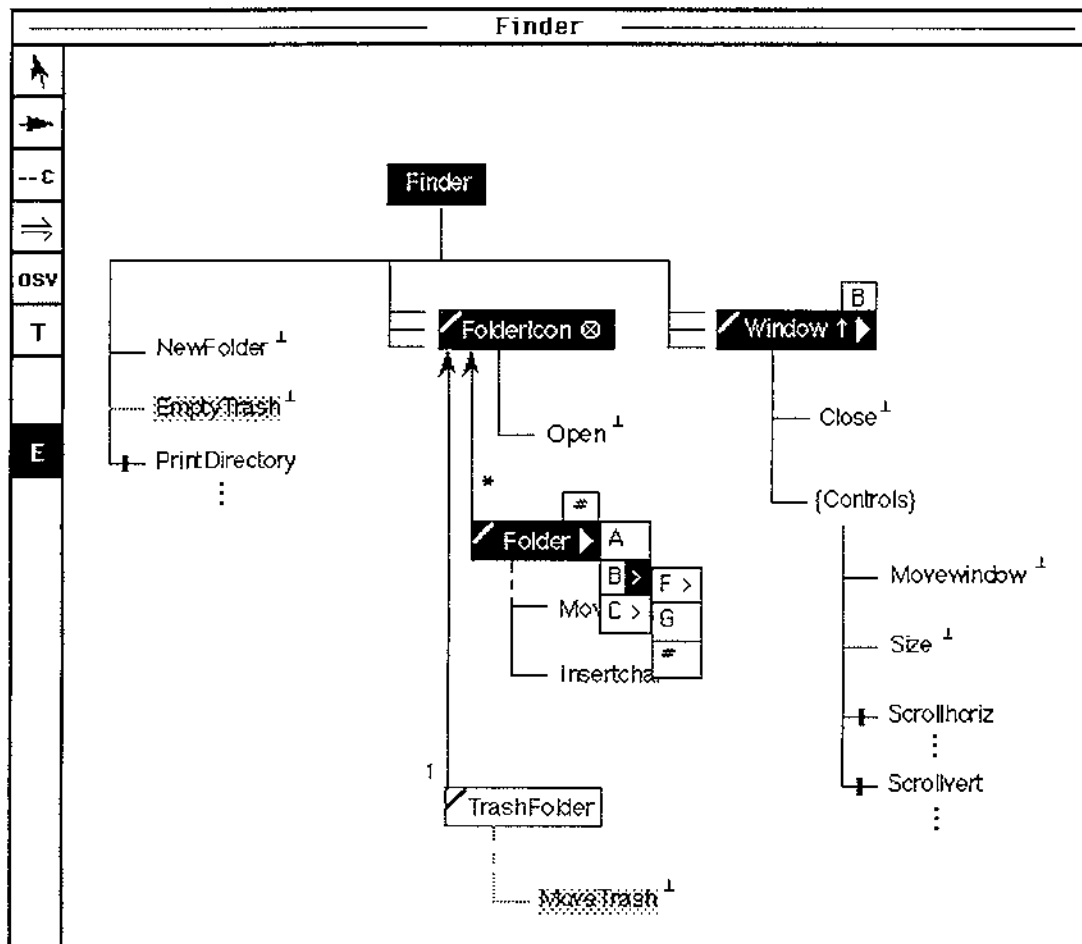
Stage #9: Window *B* selected (active)

- Window *B* is now the active window.
- The corresponding *FolderIcon/Folder* pair has also been selected via a selection trigger, with *B* now appearing in the *Folder* tab. This has also resulted in the ungreying of the *FolderIcon* and *Folder* subdialogue trees.



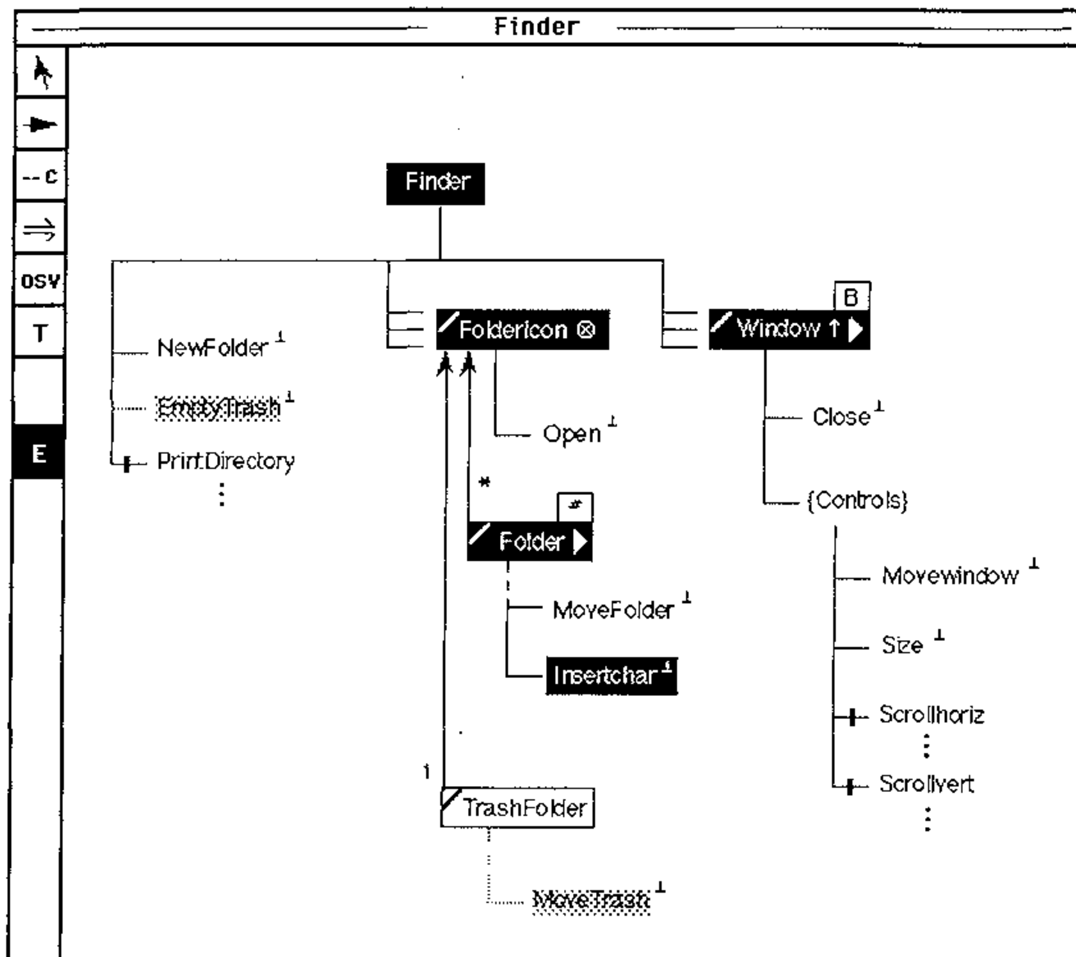
Stage #10: Creation of a new folder

- *NewFolder* has been selected at the root level, and will remain selected until the operation is completed. The new folder will be created within the current window - window *B*.



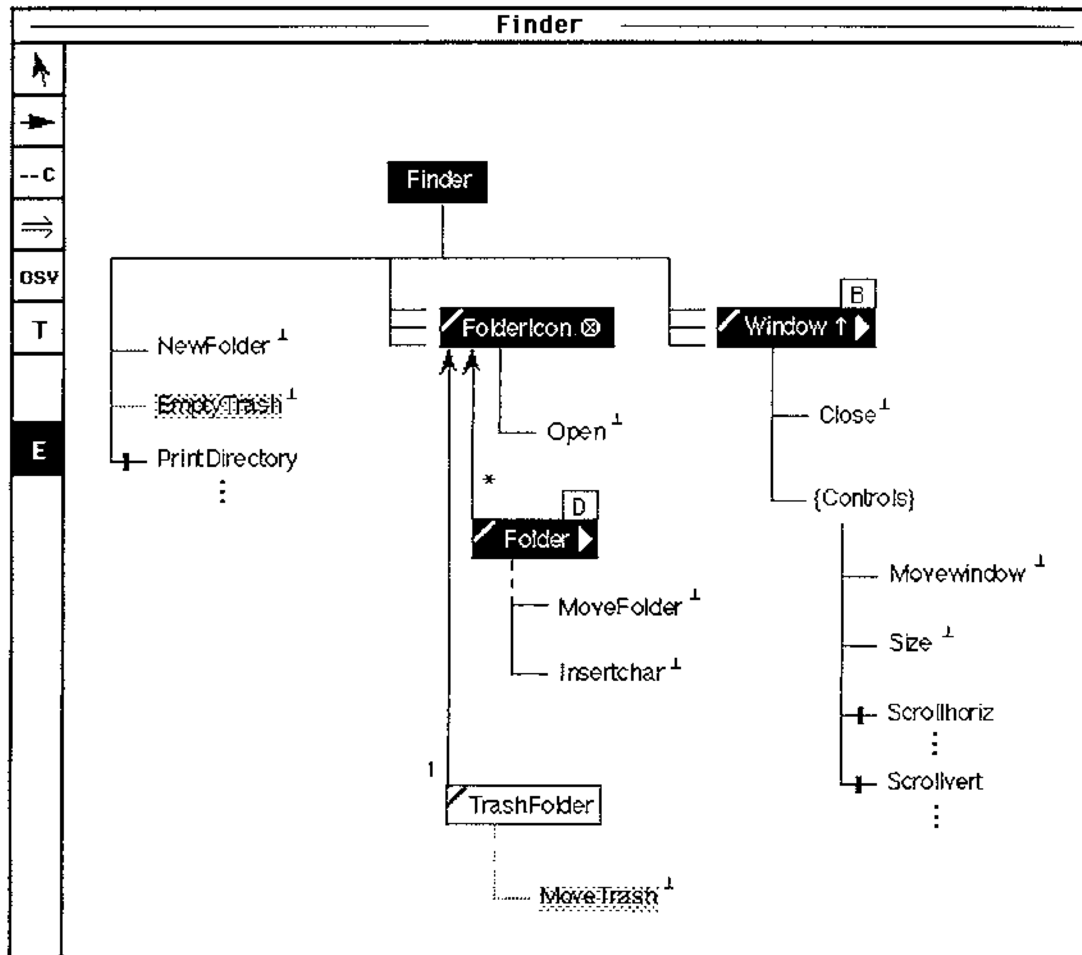
Stage #10: New folder created

- A new folder instance, with default name #, has been created with folder *B* as parent.
- Additionally, the pull-down folder menu has been displayed, showing the new folder under parent *B*.



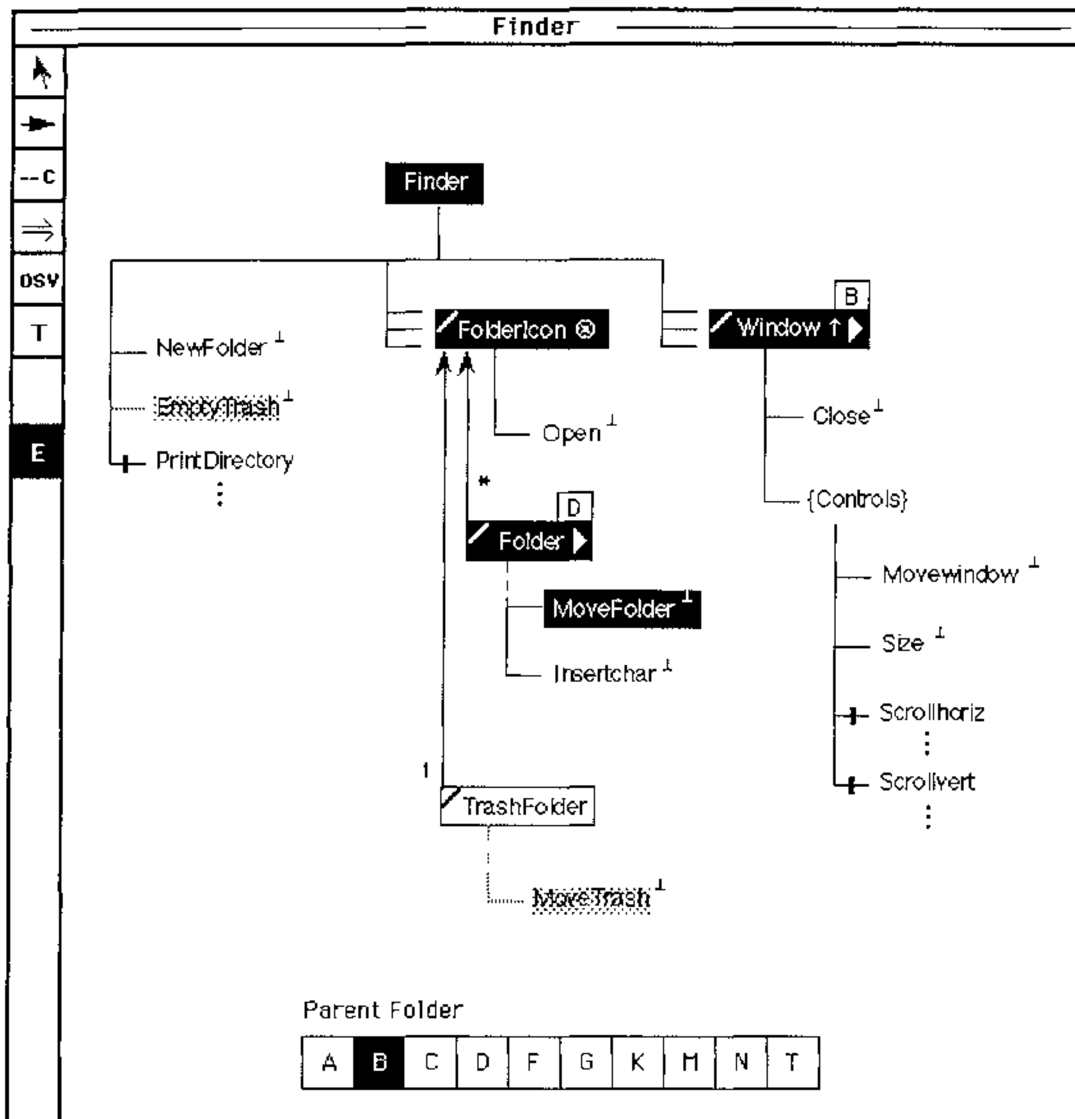
Stage #11: Renaming of the new folder

- *Insertchar* has been selected for the current folder instance, #. *Insertchar* will return the single character typed at the keyboard as the name of the folder, and will then revert to an unselected state.



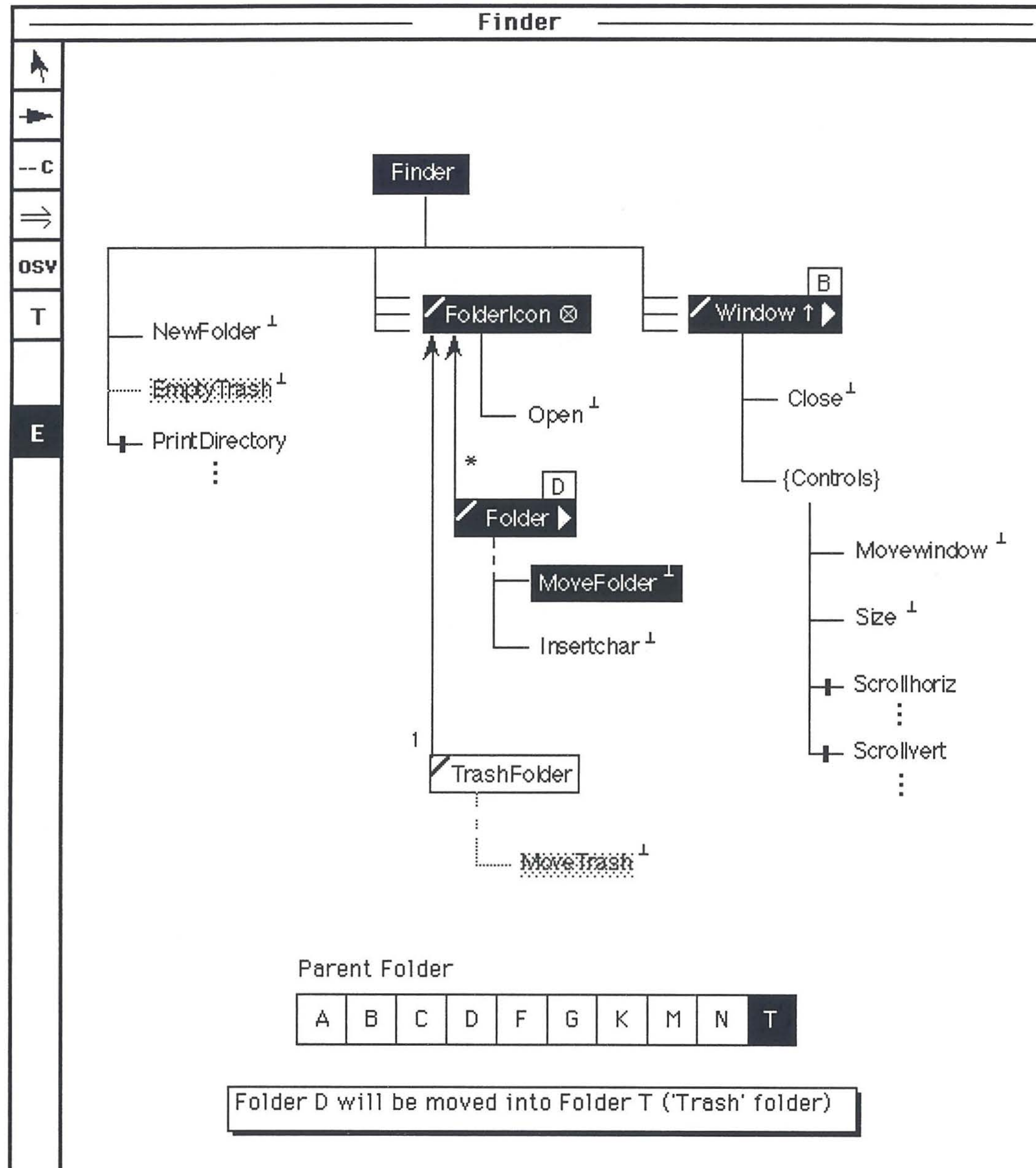
Stage #11: Folder renamed *D*

- The current folder has been renamed *D*, which name now appears in the *Folder* tab.



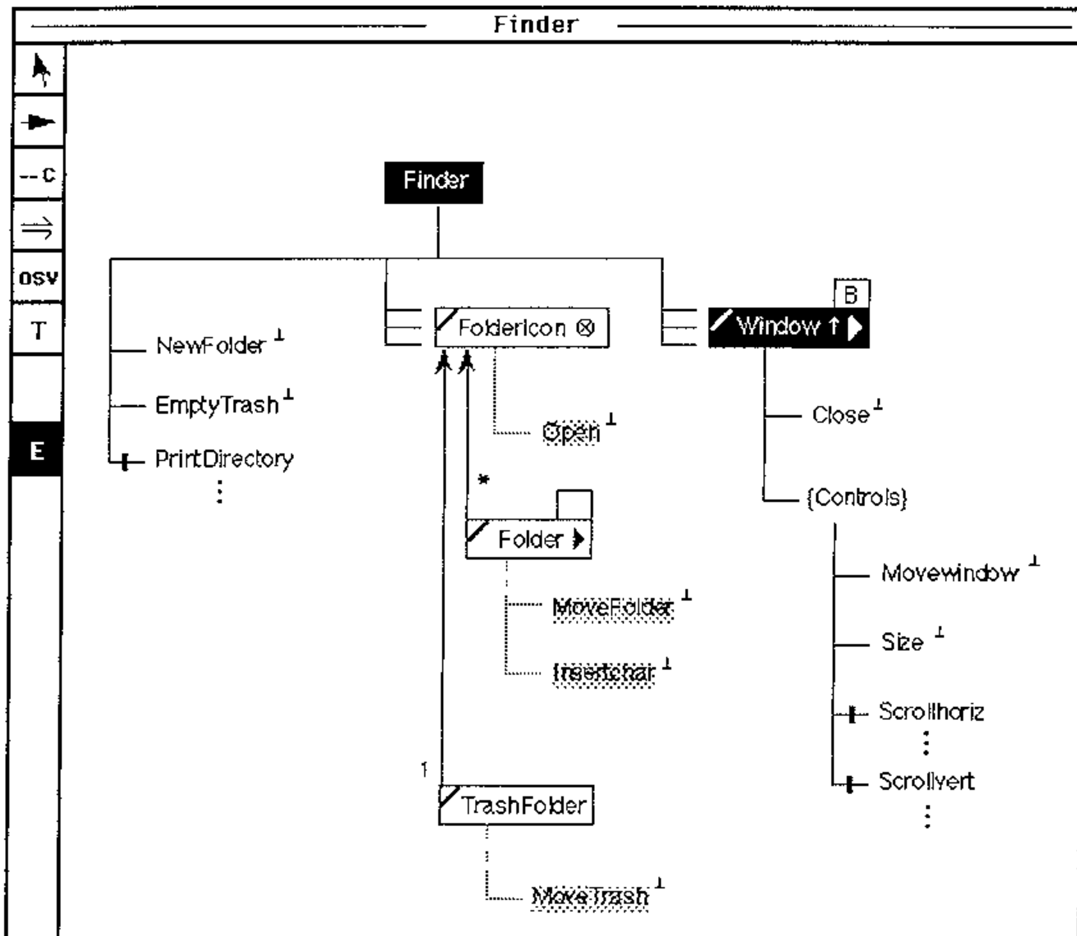
Stage #12: Moving folder *D* into the Trash folder - step 1

- *MoveFolder* has been selected for folder *D*. As moving a folder may result in the selection of a new parent folder, an array showing the names of potential parent folders (with the current parent *B* highlighted) has been displayed.



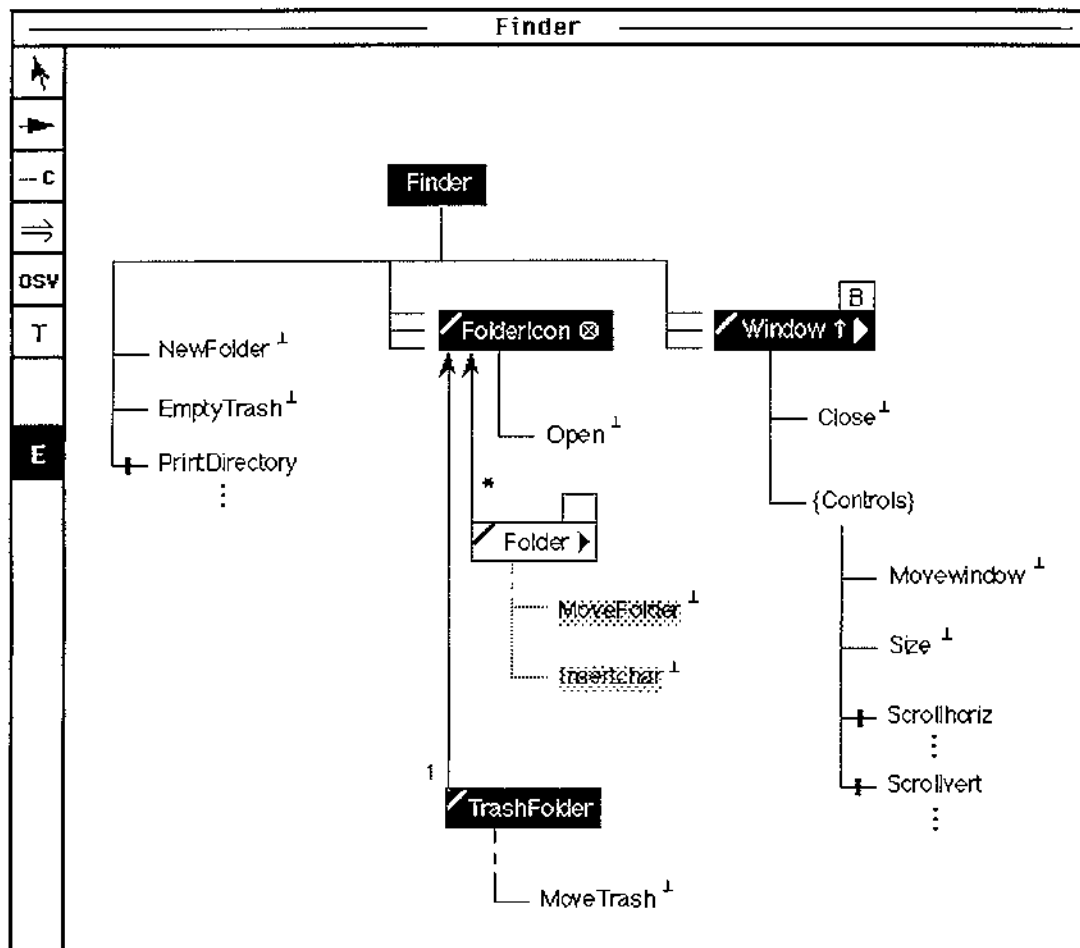
Stage #12: Moving folder *D* into the Trash folder - step 2

- *T* has been selected as the potential new parent folder. A confirmatory message is displayed.



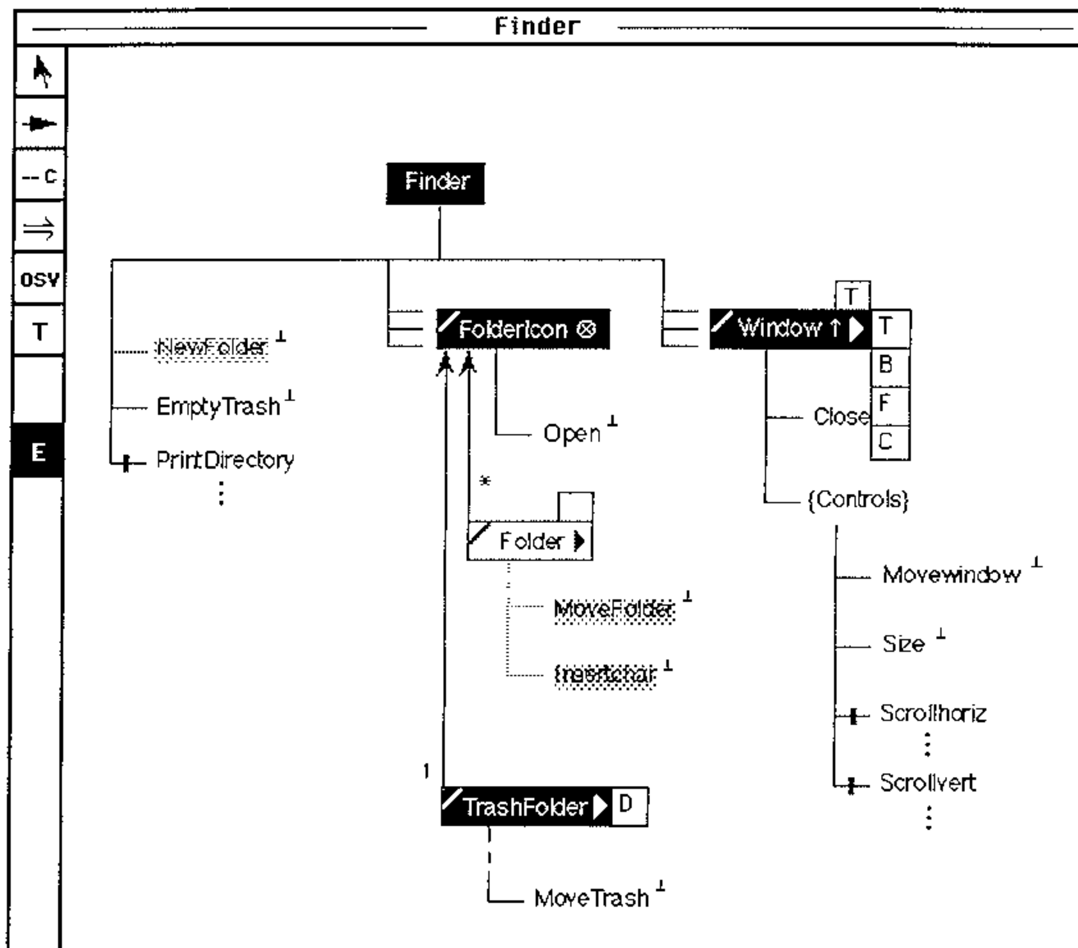
Stage #12: Folder *D* in the Trash folder

- Folder *D* has been moved into the Trash folder.
- As a result, the *EmptyTrash* option at the root level is now ungreyed and available, and the *FolderIcon/Folder* pair representing folder *D* has been deselected and the respective subdialogue trees greyed.



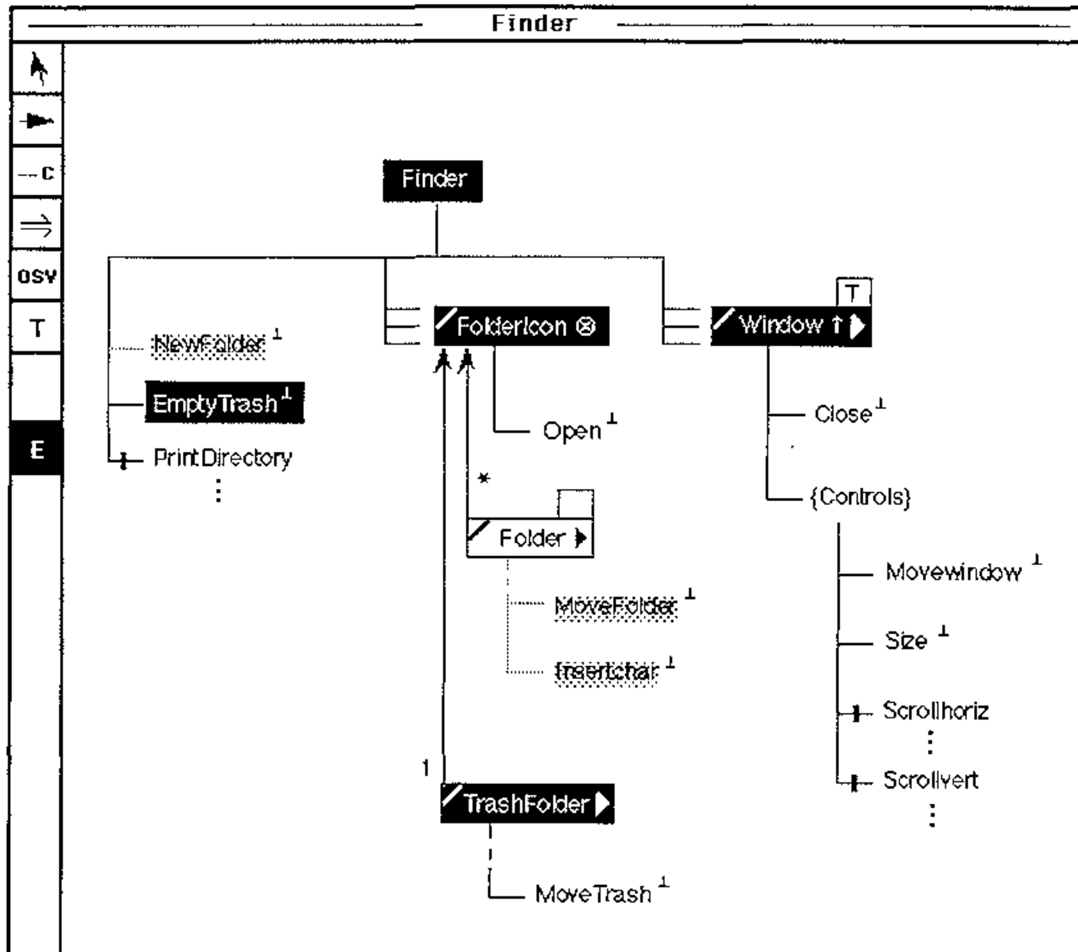
Stage #13: Trash folder selected

- *TrashFolder* has been selected, causing selection of the parent type *FolderIcon*. Options in the respective subdialogues have become ungreyed.



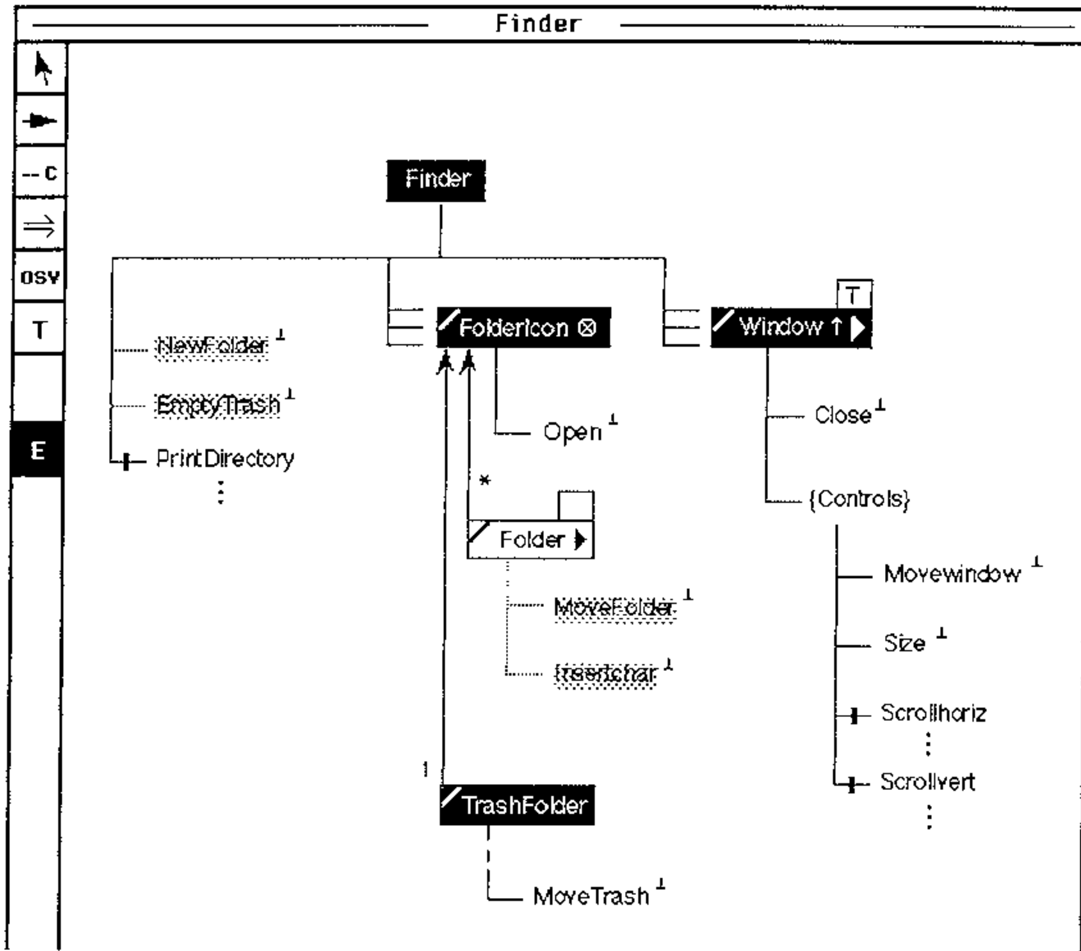
Stage #14: Trash window open

- The Trash window has been opened, causing the *NewFolder* option in the root dialogue to become unavailable and greyed.
- Additionally, both the window stack and the contents of *TrashFolder* have been displayed in pull-down menus.



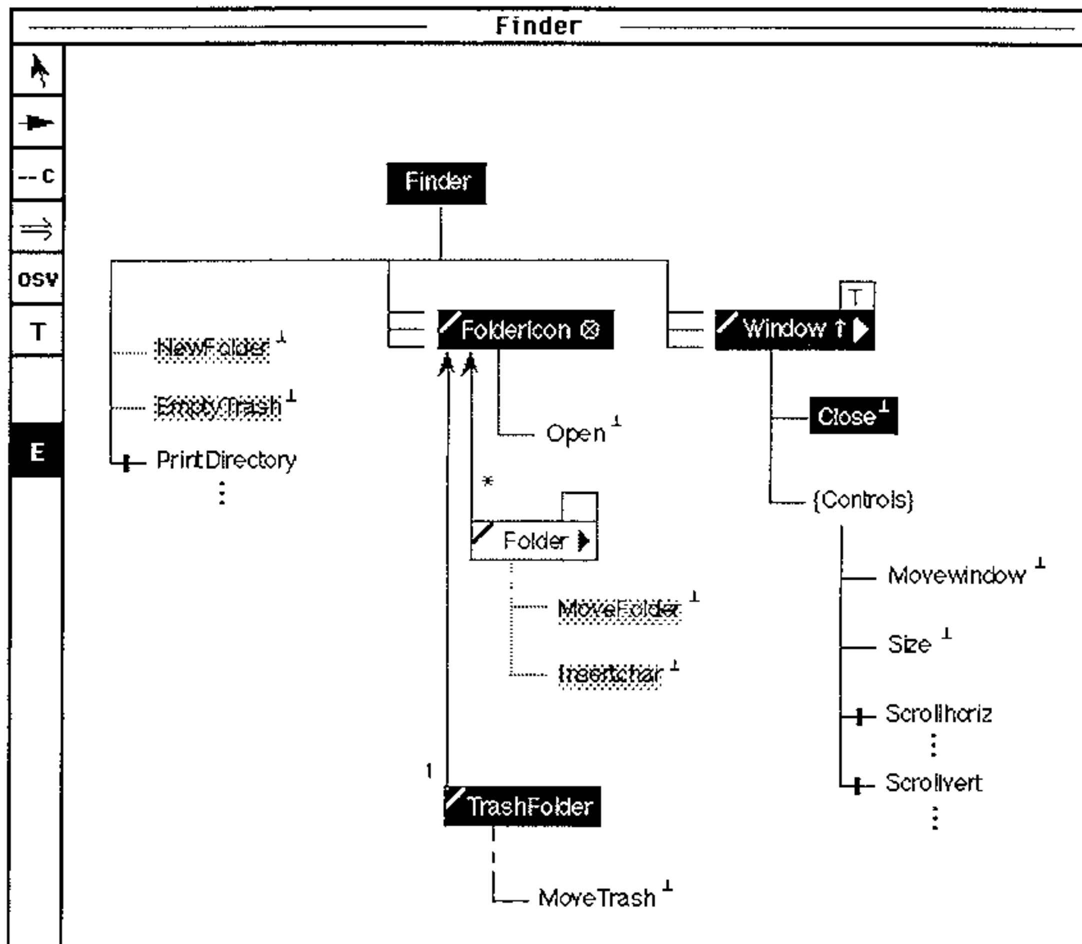
Stage #15: Emptying the Trash folder

- *EmptyTrash* has been selected at the root level, and will remain selected until the operation is completed.



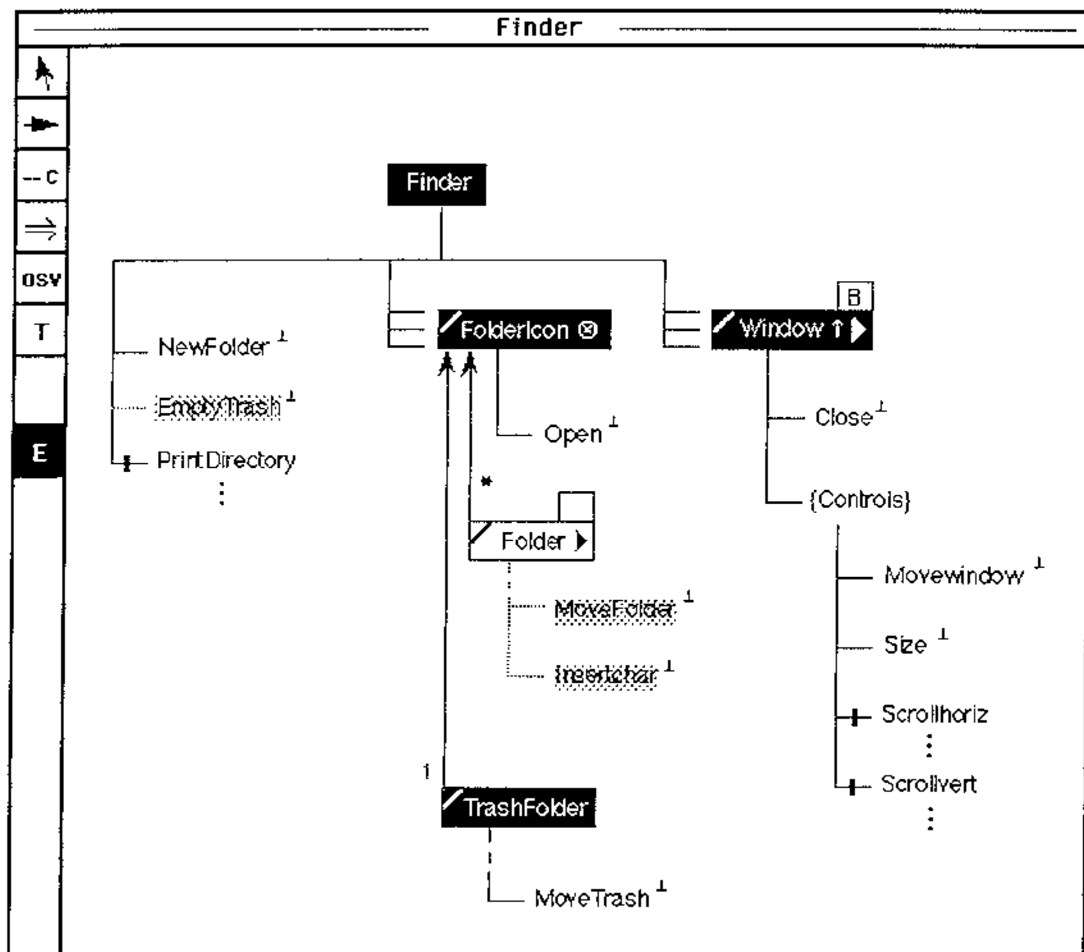
Stage #15: Trash folder emptied

- The Trash folder has been emptied.
- *EmptyTrash* is no longer available and has been greyed.



Stage #16: Closing the Trash window

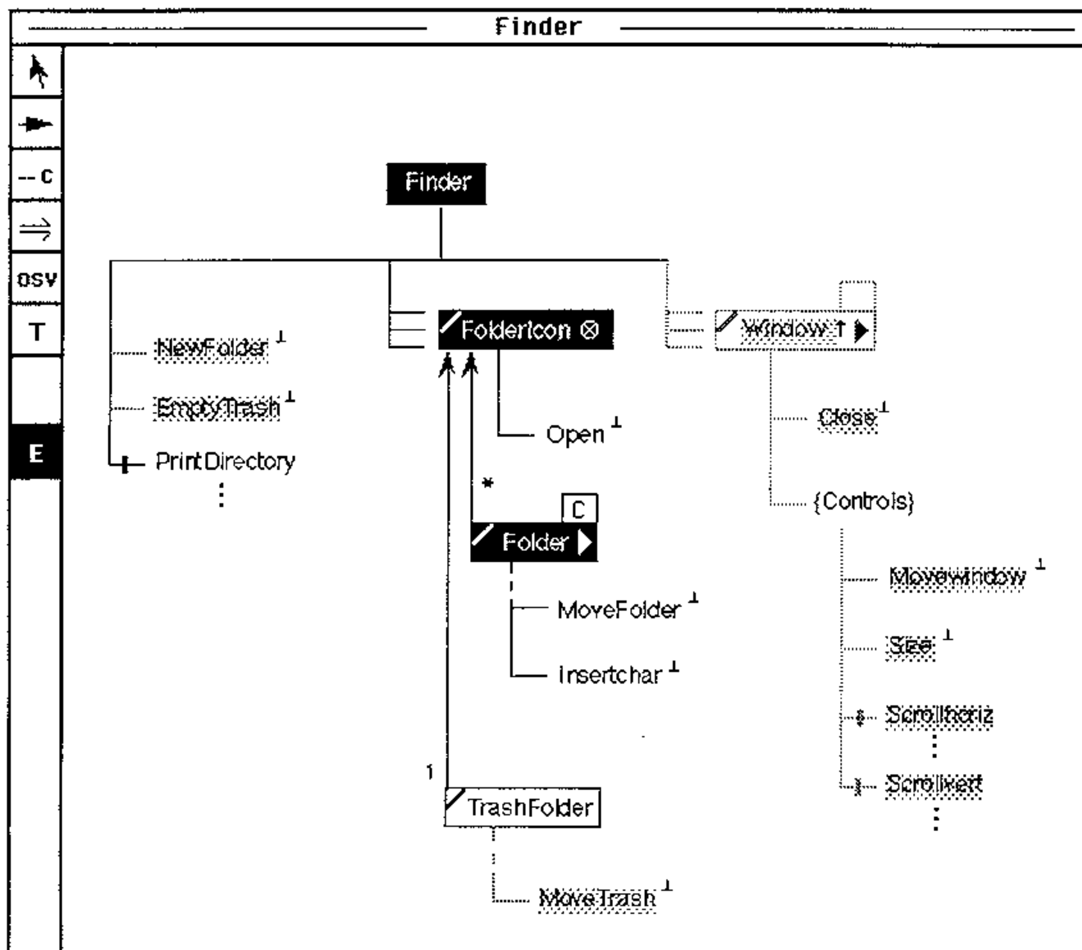
- *Close* has been selected for the active window, *T*, and will remain selected until completion of the operation.



Stage #16: Trash window closed

- The Trash window has been closed, and the next window on the stack, *B*, has become the active window.

Stages #17 to #19 of the simulation repeat Stage#16 for windows *B*, *F*, and *C*, that is these windows are successively closed.



Stage #19: All windows closed

- The above diagram shows the state of the dialogue at the end of Stage #19, following closure of all windows. It is similar to Stage#2 with folder *C* selected rather than *A*, with both the *Window* dialogue tree and the *NewFolder* option greyed.
- Finally the dialogue is terminated by deselecting the *Finder* meneme, which returns the dialogue to State #1.