DIALOGUE ACTIVATION: AN APPROACH TO USER CENTRED CONSTRUCTIONAL MODELLING OF DIRECT MANIPULATION INTERFACES

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

Paul Stuart Anderson

1995

Acknowledgements

I would like to thank my supervisors, Mark Apperley and John Hudson for many helpful discussions and suggestions over the course of this research.

I should also like to thank my family (Kathy, Christopher, Miles, Philip and Rosemary) without whose support this project would not have been possible.

Finally I should like to thank Chris Phillips for reading a draft of this thesis and giving me a great deal of useful feedback, and encouragement to finish it.

This work was funded in part by a grant from the Massey University Research Fund.

Abstract

Early stages in the development of interfaces involve the construction of models that aid interface analysis prior to construction. These behavioural models generally take a user-centred perspective. In contrast, subsequent implementation models tend to take a system-centred view of the interface. As a consequence of this change in viewpoint, the task of translating an analysis model into its implementation equivalent is extremely difficult.

This thesis proposes a constructional modelling approach for direct manipulation user interfaces (DMUI) that takes a user action viewpoint. Based on a hierarchy of dialogue groups and the notion of dialogue activation, sequence and concurrency within the interface can be described. Dialogues can be in one of two possible states, active or inactive. An active dialogue is one with which the user is able to interact. A dialogue becomes active only if its parent 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 may be carried out. Dialogue Activation Language (DAL), a language for describing such models is developed, and shown to be applicable to a range of interaction styles. An architecture capable of implementing the dialogue activation model is proposed, and a user interface development system (PIPS), based on this architecture and using DAL is described.

It is argued that DAL takes the same view of an interface as would be used in its initial analysis, and as a consequence, facilitates the translation of these early interface models into working prototypes. In addition, it is proposed that taking the DAL approach to modelling DMUI allows great flexibility in describing interaction and encourages experimentation with entirely new interaction styles.

Contents

Chapter 1
Motivation and Scope

1.1 INTRODUCTION

The Achilles heel of any interactive software system is its interface. The user interface is responsible for giving the user access to the functionality of the underlying application. It follows that a deficient or poor interface will inhibit this access, resulting in a feeling of frustration by the users, and lead to a number of possible performance related problems. For example the user may find the program very difficult to learn and use, or they may constantly make errors. The problem may be so extreme that the user will simply refuse to use part or all of the system [Thimbleby91]. Such difficulties are commonplace, and with the increasing use of computers in our society are becoming increasingly well known to the public at large.

As the software market place has become larger and increasingly more competitive, pressure has increased on software developers to increase the functionality of their products. This trend towards comprehensive more sophisticated software has been assisted by hardware improvements and cost reductions. For example, it is now not uncommon for microcomputer applications to be 2 to 3 Mb in size (1994). Over the past five years average memory sizes on desktop computers have increased tenfold.

It can no longer be guaranteed that users of software will be computer literate. It is increasingly the case that the computer, as the most flexible tool, is being used by people from all walks of life. Complex interactive software needs to be easy to use and very supportive of such users. Even expert users will be unfamiliar with some of the features of a complex software package. Consequently the interface needs to be easy to learn and use, and unobtrusively supportive of both novice users, and expert users when using infrequently used

features. Traditional interactive software used a command language interface. This had the disadvantage that the user had to learn the language of the application before they could use it. As the complexity of the interface increased, this was reflected in increasingly complex interaction grammars, that in turn were more difficult to learn and use. The user interface in this case presented a major barrier between the user and the application.

These problems have led to the development of model based graphical user interfaces in which the user is presented with graphical objects representing semantic entities [Shneiderman83]. The user is able to directly manipulate these objects and in so doing interact with the underlying application. These ***direct manipulation*** interfaces present particular difficulties in their implementation. To quote [Myers89;page 15]

> *"As interfaces become easier to use, they become harder to create. The easy-to-use, direct-manipulation interfaces popular on many modern systems are among the most difficult to implement."*

The implementation of an interface is just one stage in the development process. As is the case in other areas of software engineering, there is an emphasis on analysing the system prior to construction in order to identify problems before they are incorporated into the application and become prohibitively expensive to correct. The analysis involves the construction of one or more models of the system which can subsequently be used to answer questions about, and quantify characteristics of, the prospective interface.

1.2 THE BEHAVIOURAL AND CONSTRUCTIONAL DOMAINS

The development of user interfaces falls into two distinct phases (Figure 1.1). In the first phase the *interaction* between the user and the computer is designed. In the second phase the *software* required to support the interaction is developed. These two activities are very different. Hartson and Hix refer to the domains of these two stages as *behavioural* and *constructional* respectively [Hix93a].

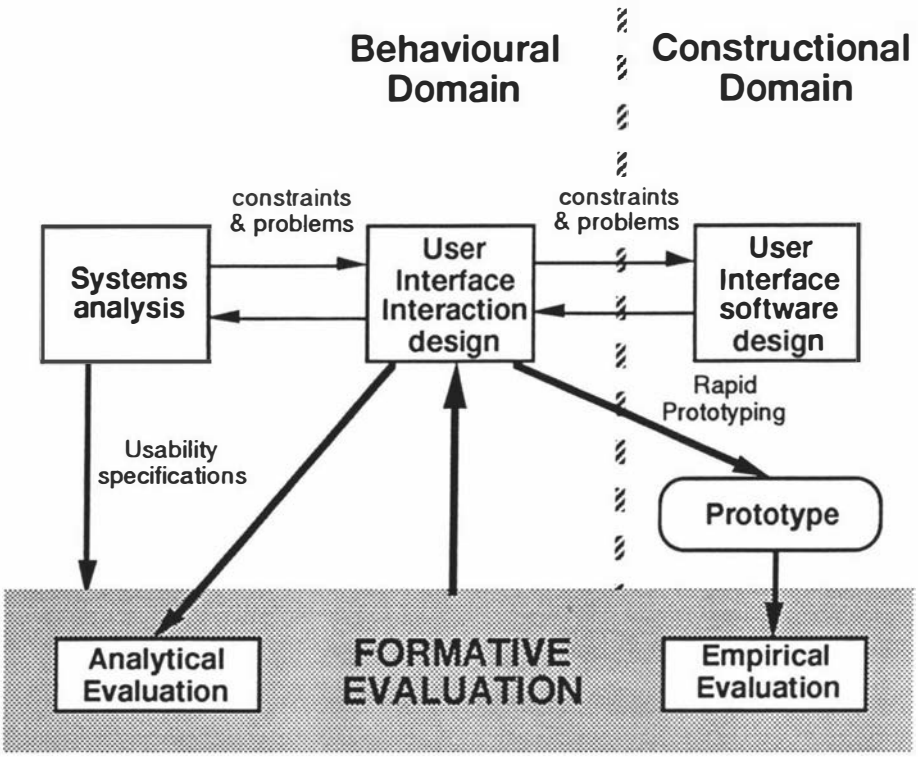


Figure 1.1: Formative Evaluation and its relationship to other user interface development activities. (Adapted from [Hix93a])

Modelling activities in the behavioural domain take a user perspective, indeed in many respects such models can be considered as models of both the user and the interface. Behavioural modelling is generally the responsibility of human factors (HF) personnel. By way of contrast, modelling activities in the constructional domain take a system centred perspective and are geared towards the development of software that responds to the actions carried out by the user and defined within the behavioural model. Constructional modelling is the responsibility of software engineers.

Prototyping is well established as a necessary approach to the development of interface software [Wasserman85]. It is also an essential activity within the behavioural domain. In the prototyping approach an interaction design is developed and evaluated. Based on the results on this evaluation the design may be refined and the evaluation-refinement cycle repeated. The problem with the prototyping approach is defining when it is complete. One solution to this problem is to produce a usability specification as part of the requirements definition of the system. Prototype interfaces can then be evaluated against the

usability specification and the prototyping cycle terminated when they match the desired usability.

This rapid, incremental refinement of prototype interaction designs is referred to as ***formative evaluation*** [Hix93b]. An interaction design can be evaluated either by *analysis* of a description or by an *assessment* of its implementation. The former is referred to as analytic, the later as empirical evaluation. Underpinning empirical formative evaluation is the need to rapidly prototype an interface given an interaction design. A number of tools are available to assist with the rapid implementation of user interfaces, but as is the case with normal construction modelling approaches these tools require the developer to take a system centred perspective. Since the interaction design takes a user centred perspective it follows that a translation between viewpoints is necessary in order to implement a prototype from an interaction design. The same problem is encountered when, on completion of the interaction design, the corresponding (final) user interface software is developed.

It has been recognised that this change of perspective in passing from the behavioural to the constructional domain is [Hartson90;page 201]

> *"not a trivial problem, because it is not a line-by-line translation between languages. It is translation from one "coordinate system" to another, requiring transformations in structure that can reach deeply into a design."*

This results in a *barrier of perspectives* that makes it difficult for the full advantage of early analysis being fully realised. Contemporary developments are frequently carried out by multi-disciplinary teams. Human factors (HF) personnel are involved from the earliest stages of interface design right through to the final evaluation [Grudin87]. HF personnel consider systems in terms of the user task viewpoint, and not the implementation viewpoint normally considered by software personnel [Siochi89].

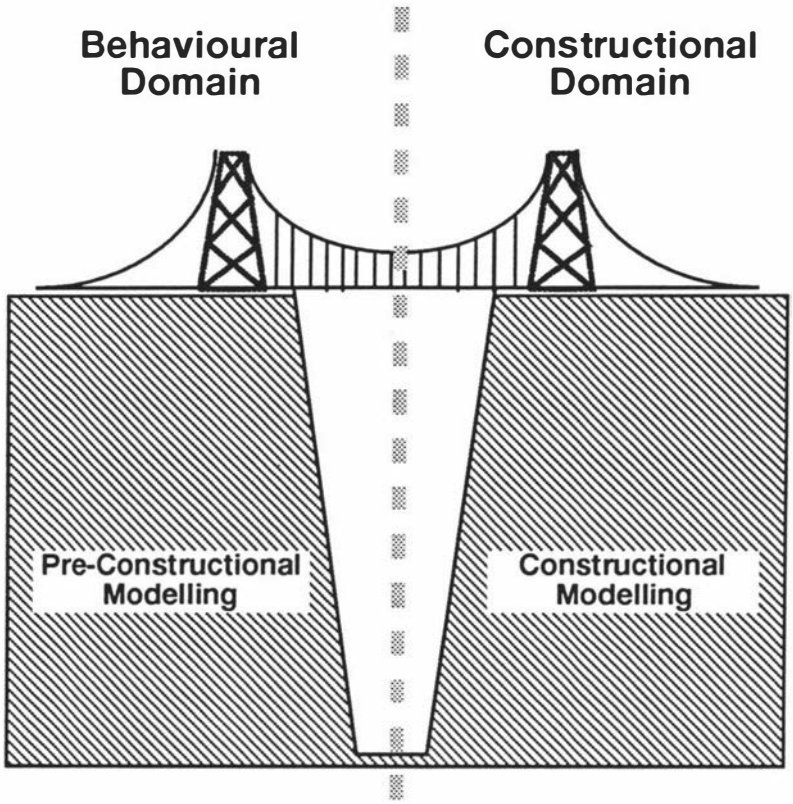


Figure 1.2: The *barrier of perspective's* between behavioural and constructional modelling domains.

This in turn can lead to communication difficulties between the two disciplines. If this problem is not addressed, HF involvement in user interface development will tend to be restricted to late rather than early stages of the development. This is obviously undesirable since early input is essential to ensure the appropriateness of the design and the early detection of usability problems [Jeffries91]. It follows that a technique is needed to bridge this *barrier between disciplines* [Figure 1.2].

In this thesis a new approach to constructional modelling (Dialogue Activation) is examined in which a user-action perspective is used. It is surmised that since a similar viewpoint is taken to that used in behavioural modelling, translation between the two is significantly eased (Figure 1.3).

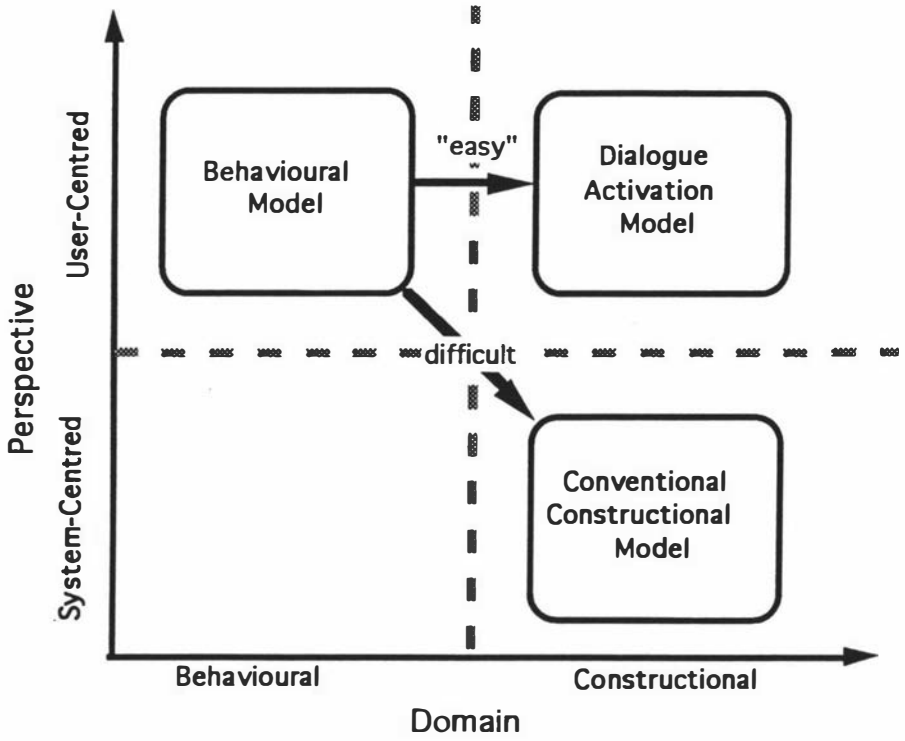


Figure 1.3: Translation to the Constructional Domain.

Dialogue Activation modelling is restricted in scope to what will be referred to as Direct Manipulation User Interfaces (DMUI). Although a commonly-used term, there has been much discussion within the literature as to its exact meaning. In the following section a definition of direct manipulation as used in this thesis is presented.

1.3 DIRECT MANIPULATION

Direct manipulation is a term first coined by Ben Schneiderman [Shneiderman83]. He describes such interfaces in terms of a set of characteristics, namely:

- Continuous representation of the objects and actions of interest.
- Physical actions or labelled button presses instead of a complex syntax.
- Rapid incremental reversible operations whose impact on the object of interest is immediately visible.

Applications with this type of interface have a number of advantages, including:

- Novice users can learn basic functionality quickly.
- There is no such thing as a syntax error, any action that the user is able to carry out is syntactically correct. Error messages are only rarely necessary.
- Users can immediately see if their actions are taking them in the direction of their desired goals. The expression "What You See Is What You Get" (WYSIWYG), has become associated with this style of interface when used within editor or similar applications [Shneiderman87].

In summary, users of DMUI have described a "feeling of involvement directly with a world of objects rather than communicating with an intermediary" [Hudson87].

Counterbalancing this positive side of DMUIs is a very negative one, namely implementation. DMUIs are very difficult to implement [Shneiderman88, Myers89]. A major reason for this is that they are extremely complex [MacLean87] and may constitute a considerable proportion, in terms of size, of the overall software package. Studies have shown that in some cases as much as 80% of the code in an application may deal with user interface issues [Lee90]. These size and complexity problems can make prototyping of such interfaces extremely difficult. Without appropriate tools to accelerate their production and modification, time and money considerations will severely restrict the use of prototyping.

1.3.1 A Definition of Direct Manipulation

The concept of direct manipulation was introduced above. Shneiderman used the term for interfaces that presented the user with graphical representations of objects of interest that could be manipulated using some form of pointing device. Summarising direct manipulation [Shneiderman88] Shneiderman describes such interfaces as those that "...create a visual representation of the 'world of action' that includes selectable displays of objects and actions of interest".

Other workers have tried to pin the definition of direct manipulation down to more concrete foundations other than subjective feelings on the part of the user.

Hutchins et al [Hutchins86] attempts to categorise interfaces in terms of semantic objectives and the degree to which the user feels that they are directly manipulating the objects of interest. They divide interfaces into two groups, those based on the conversational world metaphor and those based on a model world metaphor. Traditional command language interfaces are of the former type. They are characterised by the user and system having "a conversation about an assumed, but not explicitly represented world". direct manipulation interfaces are an example of the latter. In these systems "the interface is itself a world where the user can act, and that changes state in response to user actions". Hutchins et al attempt to further classify interfaces on the basis of the relationship of the users actions and system responses to the users semantic objectives. The concept of distance is proposed as a measure of the cognitive effort required by the user to relate the task that they wish to accomplish to the way that they are able to achieve that task via the interface. The users feeling of directly manipulating the objects of interest is referred to as engagement. Based on these notions of cognitive distance and engagement a classification of interface types was proposed as shown in Figure 1.4.

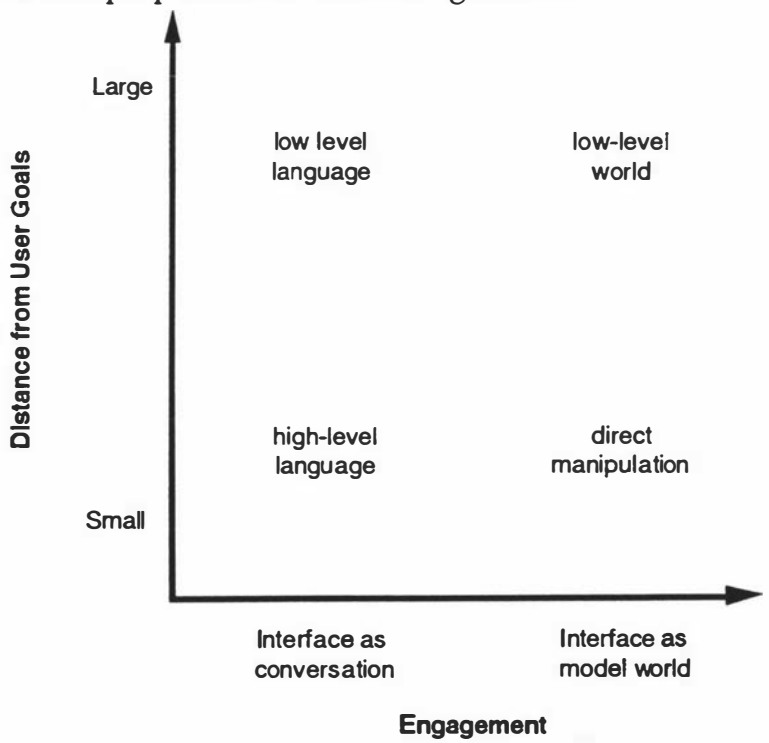


Figure 1.4: A classification of interaction styles [Hutchins86].

The interpretation of this classification is as follows:

- Conversational world interfaces use a linguistic interface. Low level interfaces of this type are domain independent and require verbose dialogues in order to undertake even simple tasks. Conversely high level language interfaces are domain specific and support suitable constructs to specify tasks in their particular domain in a short, concise fashion.

- Model world interfaces use graphical entities that the user is able to manipulate in order to specify an action to be carried out in the task domain. Similar to its conversational counterpart, a low level model world may require many involved and non-intuitive manipulative interactions in order to achieve a desired semantic goal. Where a model world has been mapped onto a specific problem domain, tasks within that domain can be carried out in short, simple, intuitive operations on the graphic entities within the interface. It is these domain specific model world interfaces that are classified by Hutchins as direct manipulation.

Others have also identified that model world type user interface need semantic engagement in order for them to be classified as direct manipulation [Edmondson91]. Indeed Edmondson suggests that the Macintosh Finder® interface which many have suggested as a classic example of a direct manipulation interface, is in fact not a direct manipulation interface at all. He observed that the operation of dragging an icon onto the trash can have different meanings depending on whether the icon represented a file or a disk. This context sensitive interpretation was said to be typical of a linguistic rather than a model based interaction.

This classification of model world interfaces into low-level at one end and direct manipulation at the other is a simplification. One could envisage a whole spectrum of model based interfaces with varying degrees of semantic distance. It is interesting to note that interface style guidelines like those of the Macintosh® [Apple-Computer87] or those defined for OpenLook® [Sun-Microsystems89] are domain independent. They define the "look and feel" of the interface and say very little about the interpretation that the application is to make of the users actions, or the (semantic) feedback it should produce in response. In this thesis Shneiderman's original definition of direct manipulation is taken rather than a subjective definition based on the degree of semantic linkage.

DMUIs also show other characteristics not seen in interfaces using a conversational world metaphor. Within a DMUI a user is presented with a number of objects with which he or she is able to interact. The interface has no control over which object and associated dialogue the user chooses; and hence is non-deterministic with respect to the users actions. The user may undertake an interaction with one group of dialogue objects, interrupt this to interact with another group, subsequently to return to the first group and carry on from where they left off. This is referred to as ***interleaving***. In order to support this behaviour the interface must be capable of maintaining multiple threads of execution. In addition the actions that the user carries out can occur at any point in time, that is actions by the user are asynchronous. These characteristics are the opposite to that observed in conversational world type interfaces in which only a single task path is maintained, and the conversation follows a fixed sequence and is synchronous in nature.

1.4 TOOLS FOR USER INTERFACE DEVELOPMENT

The need for tools to support the rapid development of direct manipulation user interfaces was identified in section 1.2 . Broadly speaking, such tools fall into two categories:

- Toolkits; libraries of interaction routines that can be called up from the application code as and when required. The use of these libraries has the advantage of enforcing a degree of consistency both within a single application and across multiple applications running on the same platform. These interaction routines usually have a corresponding display object (*widget*) associated with them, the toolkit routines defining the behaviour of the widget in response to user actions.

- User Interface Management Systems (UIMS); these are sometimes called User Interface Development Systems (UIDS). These usually consist of a collection of related tools that together permit particular aspects of an interface to be defined and automatically implemented. They often make use of a toolkit to support the basic interaction components of the interface. Part of the motivation for UIMSs has been the desire to separate the interface from the application, separation being very

desirable since it allows user interface issues to be dealt with in isolation from the application.

Toolkits define a great deal of the look and feel of an interface. A UIMS usually supplies a run time kernel that makes use of toolkit routines for the basic user interaction. Figure 1.5 depicts this relationship (from [Bass88]).

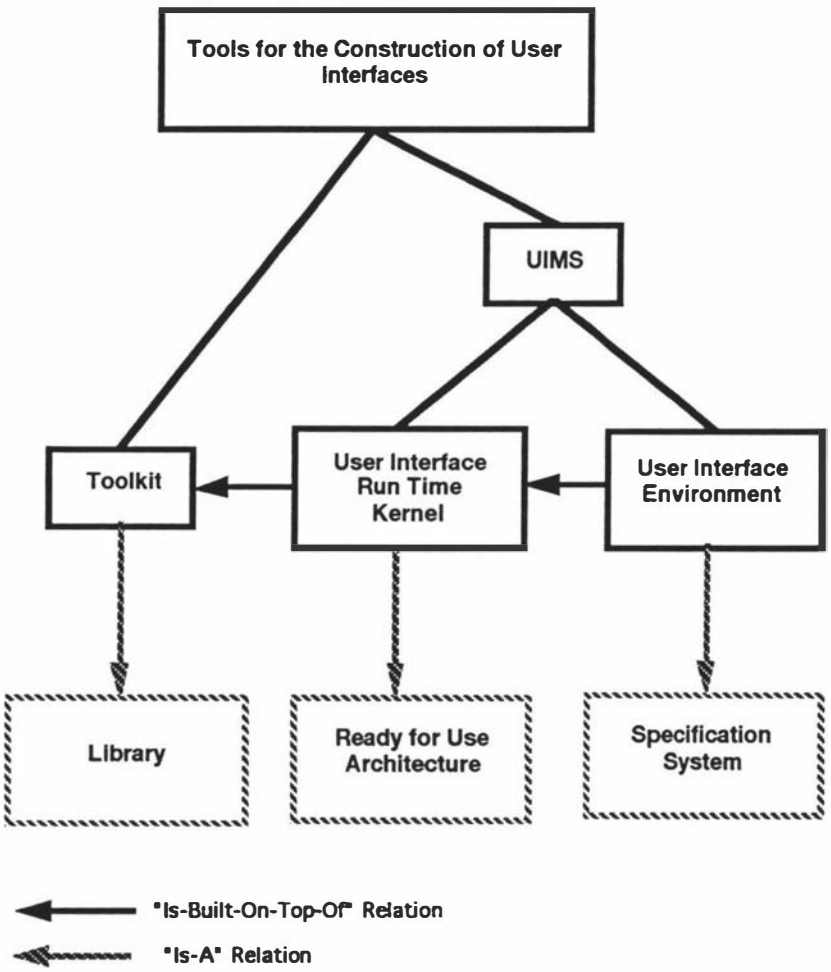


Figure 1.5: A taxonomy of user interface development tools [Bass88].

It follows that a developer making use of toolkits either directly, or indirectly through a UIMS is restricted to those interaction techniques supported by the toolkit. This results in experimentation to find the best interaction technique being severely curtailed. Some designers are unwilling to accept this restriction because they want products to have a unique look and feel [Myers89]. This problem can be addressed by using object-oriented toolkits, permitting the user to develop new interaction components from existing ones. However, this

strategy is critically dependent on the quality of the set of basic components [Took90].

Another concern with these tools is that they are very much geared towards interface development by programmers. Although this may be alright for the development of the final user interface software, the need for programming ability may be a major impediment to the use of such tools by HF personnel when producing prototypes. This problem has been recognised and address to some extent by tools such as Peridot [Myers87] and Garnet [Myers90], in which programming by demonstration approaches are used. However the use of such techniques is very limited, for example the Garnet environment consists of several tools, programming by demonstration being simply a minor component. It is clear that users of such a system would still have to carry out a substantial amount of programming in one form or another.

1.5 MOTIVATION

User interface interaction designs take a user-centred view. Any translation of such a design to either a prototype for evaluation or a (final) user interface software design requires a change of model perspective, from user-centred to system-centred. Current tools take a system centred perspective of the interface, hence the translation between the user perspective in the interaction design and the system perspective in the implementation is left to the developers. As discussed above, such a translation is very difficult. Without the assistance of such tools, the value of interaction modelling is likely to be considerably reduced due to the corresponding reduction of:

- refinement of interaction models through prototyping,

and

- the degree to which the developers of the interface software accept the interaction design without modifying it for 'ease of implementation'.

If the improvements in user-interface development due to behavioural modelling are to be fully realised, it is essential that tools are developed to address this translation issue.

1.6 RESEARCH OBJECTIVES

The primary objective of this thesis is to show that user-centred constructional modelling is a viable approach to addressing the problem of translating interaction designs into their corresponding implementation models. A secondary objective of this work is to demonstrate that a dialogue selection interaction paradigm is capable of modelling most direct manipulation dialogue styles. These objectives are to be met by the development of a dialogue specification language based around the concept of dialogue selection, and tested via the implementation of a supporting environment.

1.7 THESIS OUTLINE

This Chapter has identified that there are problems in translating a behavioural interface model into its constructional equivalent and has proposed that the use of a user-centred constructional model could alleviate these translation difficulties. A review of behavioural and constructional interface models is presented in Chapters 2 and 3 respectively. Chapter 4 examines the translation of a specific behavioural interface model into its constructional counterpart and then attempts to draw some general conclusions from this example. The concept of dialogue activation is introduced in Chapter 5 and Dialogue Activation Language (DAL), a language for user-centred constructional modelling of direct manipulation user interfaces introduced. The syntax of DAL is defined in appendix A. A formal analysis of DAL is presented in Chapter 6 with particular emphasis on techniques for statically identifying run-time problems in systems defined in DAL. Chapter 7 describes PIPS, a prototype user interface development environment based on DAL. DALTRAN, a tool in the PIPS environment, includes some static analysis facilities based on the theory presented in Chapter 6. Appendix B presents analyses produced by DALTRAN of a number of example sources which contain event cycles, a problem of DAL identified in Chapters 5 and 6. In order to demonstrate the flexibility of DAL, a range of dialogue design examples are developed in Chapter 8. Implementations of these dialogues together with others introduced in Chapter 5 are contained in Appendix C. The thesis concludes in Chapter 9 with a summary of the findings from this research and suggestions for further work.

Chapter 2
Dialogue Modelling in the Behavioural Domain

2.1. INTRODUCTION

As presented in Chapter 1, interface design falls into behavioural and constructional domains. In this chapter the modelling approaches that have been used to describe dialogues in the behavioural domain are examined. Dialogue modelling approaches used in the constructional domain are described in Chapter 3.

A large variety of techniques have been used for dialogue modelling. Before embarking on a review of dialogue models, it is necessary to define the terms model and dialogue.

In the Oxford English Dictionary a ***model*** is defined as "a simplified description of a system for purpose of calculation" [Allen87]. Hence a model is a representation of a system in which those aspects of the system that are of importance are emphasised over those aspects that conversely are seen as unimportant. The meaning of important in this context obviously depends on the purpose to which the model is to be put. For example a model of a car might include the number of seats and the type of carburettor used. The former would probably be of importance to a salesman, the latter only of interest to a mechanic maintaining the car.

The topic area of human-computer interaction abounds with models. This profusion of models has given rise to the need for clarification as to their precise nature. This need has been further emphasised by Long [Long86], who observed that the term "user model" could, and indeed had, been interpreted in a number of different ways within the literature. To address this need various taxonomies have been proposed. For a comprehensive discussion of these the

reader should read [Nielsen90]. Nielson made the important observation that in order to characterise a model, it is important to ascertain the subject of the model, who it is held by, and for what purpose.

A dictionary definition of the term ***dialogue*** is "a conversation" or "a discussion between representatives of two groups" [Allen87]. In the topic area of human-computer interaction a dialogue usually defined as the "...observable two-way exchange of symbols and actions between human and computer" [Hartson89]. In this work a slightly different interpretation is used, the term dialogue being used to refer to the ***structure*** of human-computer interaction.

As mentioned in Chapter 1, direct manipulation interfaces require support for ***interleaving*** and preferably ***concurrency***. A number of terms related to these concepts are used within the HCI topic area and need to be clearly defined before proceeding.

A ***Thread*** is a task based concept, a thread of a dialogue being "a coherent subset of that dialogue" [Dix93;page 138]. Similarly, the term ***multi-threaded*** is frequently used to refer to "the multiplicity of task paths available to the end-user at any given instant during the dialogue" [Hartson89;page 9].

Interleaving, or ***interleaved multi-threading*** is also a task based concept. For example, Bass and Coutaz talk about interleaving in terms of the users perception; "Interleaving inside the presentation component allows the operator to act on a presentation object (for example, specify several fields in a form), then manipulate other presentation objects, and later to come back to the original one" [Bass91;page 155]. Dix et al describe interleaved multi-threading as permitting "temporal overlap between separate tasks but stipulates that at any given instant, the dialogue is restricted to a given task." [Dix93;page 138]. ***Concurrency*** is similar to interleaving, but differs in that "more than one thread can be executed simultaneously" [Hartson89;page 11].

The above task based concepts require particular support within the interface software. In order to support multi-threading the interface software must maintain the state information for each thread separate from all others. Dix et al define ***multi-threaded*** as "interactive support for more than one task at a time" [Dix93;page 138]. Tanner defines ***multi-threaded input*** as "where each

user input stream is processed in a manner that is lexically and syntactically separate from other streams" [Tanner87;page 142] emphasising the need for thread separation. If the interface is to support concurrency at the task level, then the underlying interface software must also be concurrent, with each thread in the task domain mapping onto a concurrent entity within the constructional domain. This concurrent entity is essentially a process. Supporting concurrency in itself is not enough. Sharing of information between tasks is an essential facility within an interface supporting multi-threaded interaction. It follows that the underlying software must support data sharing and synchronisation between the interface processes. Dix et al defines the term ***concurrent multi-threading*** as the "simultaneous communication of information pertaining to separate tasks" [Dix93;page 138]. This is similar to the normal interpretation of the terms ***threads*** or ***threads of control*** within computing referring to "several parallel execution sequences" [Deitel90;page 74].

It follows from the above that the terms ***thread, multi-threaded, interleaving*** and ***concurrency*** have definitions within both behavioural and constructional domains. Within this chapter the behavioural interpretation of these terms will be used. Within subsequent chapters the constructional interpretation will be assumed unless indicated otherwise.

Behavioural models are usually referred to as *user-centred*. Conversely, constructional models are usually considered *system-centred.* This raises the question as to what exactly is meant by user-centred and system-centred.

User-centred means presenting the user interface from the perspective of the user. Such models are primarily concerned with modelling what the user thinks, does and sees. System centred means modelling the user interface from the perspective of the system. Such models are concerned with defining the language of communication between the user and the underlying application; the language being defined in terms of its alphabet (lexemes), structure (syntax) and interpretation by the application (semantics). It follows that the distinction between user-centred and system-centred can be made in terms of the *composition* of the corresponding models.

In the case of behavioural models (as discussed above), a wide range of components are addressed, with cognitive modelling being addressed in some

cases and user actions in others. However, it is not usually the case that a behavioural model addresses just one single aspect of the interaction. Indeed in some cases a behavioural model may well include some aspects that would normally be considered as falling within the system perspective. A dialogue model, by definition is concerned with both user and system sides of communication. It follows that a dialogue model in either the behavioural or constructional domains must contain both user and system view elements.

Models in the behavioural domain have a number of purposes. These include [Johnson92]:

- Predicting user behaviour.
- Communicating designs.
- Providing consistency between levels of design.
- Evaluating aspects of a design.

Hartson [Hartson89] recognised three distinct categories of behavioural models. These are:

- Task Oriented Models; for the purpose of task analysis.
- Structural Decomposition Models; define a model of interaction for the general process of human-computer interaction.
- Interface Representation; for representing particular instances of human-computer interaction.

The third category might be considered dialogue models. However the picture is not as simple as might first appear. Most modelling techniques assist both synthesis (ie. construction) and analysis activities [Hix93]. In some cases a modelling technique "cuts across several types of models" [Hartson89;page 18]. The Command Language Grammar [Moran81] is just such a technique with a wide ranging application. In other cases a modelling technique that was originally designed for one purpose has been extended to include additional uses. An example is Task Action Grammar (TAG) [Payne86], which was initially developed for the purpose of interface evaluation, subsequently being extended in Extended Task Action Grammar (ETAG) [Veer90], to make it a tool "for the analysis and design of user interfaces" [Veer90;page 169].

Because of this difficulty of model classification on the basis of their primary purpose, a number of behavioural models are considered in the next section without any attempt to restrict the discussion to particular category of model.

2.3. DIALOGUE MODELS

2.3.1. User Action Notation

User Action Notation (UAN) [Siochi89,Hartson90] is a simple, task oriented notation for describing user actions in asynchronous user interfaces. The primary abstraction used in this modelling approach is the user's actions. These actions are augmented with the resulting feedback and any resulting change in the state of the underlying system.

User actions are defined using a simple set of mnemonics (Figure 2.1) for describing the actions in terms of key presses, mouse movements, and named contexts for these actions.

Action	Meaning
~	move the cursor
[X]	the context of object X.
~[X]	move the cursor into the context of object X
~[x,y]	move the cursor to (arbitrary) point x,y outside any object
[X]~	move the cursor out of context of object X
Xv	depress button, key or switch called X
X^	release button, key or switch X
Xv^	click (depress then release) button, key or switch X
( )	grouping mechanism
*	iterative closure, task is performed zero or more times
+	task is performed one or more times
{ }	enclosed task is optional (preformed zero or one time)
A B	sequence; perform A then B
OR	disjunction, choice of tasks (used to show alternative ways to perform a task.
&	order independence; connected tasks must all be performed, but the relative order is immaterial.
;	task interrupt symbol; used to indicate that the user may interrupt the current task at this point.
:	separator between condition and action or feedback.
Feedback	**Meaning**
!	highlight object
-!	dehighlight object
!!	same as !, but use an alternative highlight
!-!	blink highlight
(!-!)n	blink highlight n times

Figure 2.1: Summary of UAN symbols.

As discussed above, the temporal aspects of the interaction are of great importance for direct manipulation user interfaces. An important feature of UAN is its facilities for specifying the temporal ordering of tasks. Given two

tasks, no less than six different temporal orderings are possible. These are summarised in Figure 2.2.

Action	Meaning
(A B)	Sequence. Task A followed by task B.
(A & B)	Order independence. Both A and B must be completed before the next step, but the sequence of carrying out the tasks can be (A B) or (B A).
(A -> B)	Interruptibility. A can interrupt B.
(A <=> B)	Interleavability. A can interrupt B and B can interrupt A.
(A || B)	Concurrency. A and B can be carried out concurrently.
A (t<n) B	Time interval. Carry out task A, and then task B in an interval of less than *n* seconds.

Figure 2.2: UAN temporal operators.

As an example of UAN, consider the action of moving the trash icon on the Macintosh desktop can be defined as shown in Figure 2.3

Action	Meaning
~[trash]Mv	Move the cursor into the context of the trash can icon and depress the mouse key.
(~[x,y])*	Move the cursor to any number of arbitrary screen locations.
M^	Release the mouse key.

Figure 2.3: Moving a Macintosh trash icon (from [Siochi89]).

In order to describe the behaviour of the system in terms of the feedback that the user perceives (by virtue of carrying out the actions) the user action trace is annotated with a feedback column. In addition, user actions may give rise to changes in the state of the interface and/or underlying applications. Columns are added to recorded these changes as and when they are required. Finally, pre-conditions to restrict the scope of actions and feedback can be added by prefixing an action or feedback with the corresponding condition.

Specifications in UAN would be very verbose if user tasks could only be defined at the detailed, user action level. To overcome this problem, UAN uses the concept of task macros, which allow a task to be defined, and then referenced from another task instead of having to re-specify the detail again. These macros can take arguments and return values, allowing them to capture generic behaviour. Consider for example the task of selecting menu items from a pull-down menu. This task can be viewed as being composed of two distinct sub-tasks:

i) The user needs to chose the appropriate pull-down menu from the menu bar.

ii) They then need to make the desired selection from the pull-down menu.

The specification for this task is given in Figure 2.4, with the two tasks defined in two separate macros, the select-menu macro referencing the select-pull-down-choice macro to describe the menu item selection sub-task.

TASK: select-menu(x, choice')		RETURNS: choice'
User Actions	**Feedback**	**Interface State**
~[x-menu-bar-choice] Mv	x-menu-bar-choice! show x-menu	
select-pull-down-choice(x,choice')	hide x-menu	Return choice'

TASK: select-pull-down-choice(x, choice') RETURNS: choice'		
User Actions	**Feedback**	**Interface State**
(~[x.choice]	x.choice!	
[x.choice]~)*;	x.choice-!	
~[x.choice']	x.choice'!	
M^	x.choice'!!	Return choice'

Figure 2.4: UAN specification for the task of selecting from a pull-down menu (from [Siochi89]).

The main strength of UAN lies in its simplicity and resulting power in aiding communication. This is reflected in its rapid up take within the interface development community [Hartson90], within which it has been used for defining a range of different interfaces, and to document human-computer interface guidelines. The UAN description is far shorter, and easier to read than the corresponding textural definition.

To further demonstrate the use of UAN, a simple dialogue will now be considered. This dialogue will subsequently be specified using a number of other notations in order to aid the comparison of the strengths and weaknesses of each.

A simple paint tool, similar to MacPaint® is to be constructed. In this tool, the user can configure the shape of the paint brush by selecting from thin and thick brush thicknesses in horizontal and vertical orientations respectively. This dialogue will henceforth be referred to as the ***brush shape dialogue****.*

On selecting the change brush shape option, the user is to be presented with a dialogue box within which they are able to select the desired shape (Figure 2.5). As with normal dialogue boxes, the user will be unable to interact with the rest of the application until the interaction with the dialogue box has been completed.

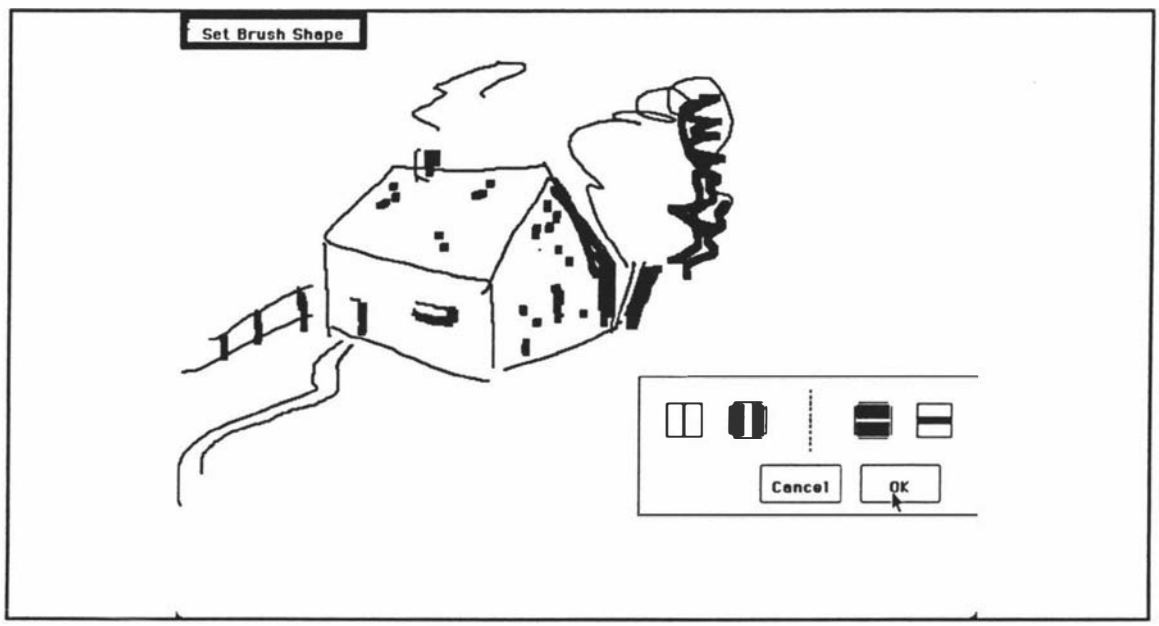


Figure 2.5: Screen of paint tool when selecting brush shape.

Within the dialogue box, the user is to be presented with two, two button groupings for selecting horizontal and vertical brush thicknesses respectively. Within these groupings, clicking on a button should result in the corresponding option being selected, and the button should be left highlighted to reflect this. The other button within the grouping should likewise be left unhighlighted following such a selection. The dialogue box is to contain two exit buttons labelled Cancel and OK respectively. Clicking OK will result in the brush thicknesses being set to those selected and the dialogue box being hidden. Clicking on cancel will result in the dialogue box being hidden, but no change being made to the brush shape. After the dialogue box is hidden, the user is to be able to resume the painting task. Finally, on entering the dialogue box for setting brush shape, the thin options for both horizontal and vertical thicknesses are to be selected by default.

Buttons in this system are to behave in the following way. Depressing the mouse key with the cursor within the context of an un-highlighted button should cause the button to highlight. Moving the cursor outside of the button, whilst keeping the mouse key depressed should cause the button to de-highlight. Dragging back into the context of the

button whilst still keeping the mouse key depressed will cause the button to re-highlight. Finally releasing the mouse key with the cursor within the button context will cause the button to become selected, and remain highlighted.

This dialogue is presented in UAN in Figures 2.6-2.9. The *Select* macro presented in Figure 2.6 defines the behaviour of a single button. The *SelectME* macro presented in Figure 2.7, builds on the Select macro, and describes the selection of one choice (button) object within a mutually exclusive group. The *SelectBrushShape* macro in Figure 2.8 describes the behaviour of the dialogue box, allowing zero or more selections from the horizontal and vertical brush thicknesses, and terminating with selection of cancel or OK options. Finally, the *Paint* macro in Figure 2.9 describes the behaviour of this system at the top level. The users main task is to paint, but this painting task can be interrupted at any point by the SelectBrushShape task.

TASK: Select(button)		
User Action		Feedback
(~[button]Mv		button!
(		
	[button]~	button-!
	~[button]	button!
)*		
[button]M^		button!

Figure 2.6: The button dialogue.

TASK: SelectME(choice1, choice2, choice') RETURNS: choice'		
User Actions	Feedback	Interface State
choice1-!: Select(choice1)	choice1!	
	choice2-!	Return choice1
choice2-!: Select(choice2)	choice2!	
	choice1-!	Return choice2

Figure 2.7: Selection from a mutually exclusive group.

TASK: SelectBrushShape			
User Actions	**Feedback**	**Interface State**	**System State**
Select(SetBrushShape)	dialogue box made visible Horizontal-Thin! Vertical-Thin!	vertical=thin horizontal=thin	
(			
SelectME(HorizontalThin, HorizontalThick, hchoice)		horizontal =hchoice	
| SelectME(VerticalThin, VerticalThick,hchoice)		vertical = vchoice	
)*			
(			
Select(Cancel)			
| Select(OK)			set-brush-shape (horizontal, vertical)
)+	dialogue box hidden		

Figure 2.8: The behaviour of the dialogue box.

TASK: Paint	
User Action	**Feedback**
(	
SelectBrushShape->Painting	
)*	

Figure 2.9: Invocation of the brush shape dialogue.

The above specification will now be considered in terms of how successfully UAN has been able to capture different aspects of the system. The *success of capture* is divided into three categories:

1. **clearly specified;** the aspect explicitly included, and is clearly apparent within the specification.

2. **poorly specified;** the aspect is implicitly included, and can be deduced from the specification.

3. **not specified**; the aspect is not captured in any way within the specification. The specification would need to be augmented to address this shortcoming.

In the case of the UAN specification of the brush shape dialogue, the following observations can be made:

clearly specified
User actions and the low level feedback in response to those actions, being the core of the notation, are clearly brought out in the specification. The same can also be said with regards to the relationship between user actions and interface and/or system state.

Through the use of macros, generic behaviour such as the way in which buttons operate are well defined. The ordering of tasks, and the (optional) repetition of tasks is also explicit within the specification.

poorly specified
Grouping behaviour as shown in the *SelectME* task in the example is somewhat hidden within the syntactic foliage of the description. Only the action pre-conditions and resulting feedback really show that these choices are mutually exclusive.

The selection of default choices is not handled very well. Default vertical and horizontal line thickness are set by introducing interface state variables and pre-setting them to the appropriate values at the start of the *SelectBrushShape* dialogue.

not specified
The UAN does not address the appearance of the screen or individual interaction objects. To address these aspects the specification needs to be supplemented with suitable pictures.

2.3.2. Lean Cuisine

Lean Cuisine [Apperley89] is a graphical, tree based notation developed for describing the behaviour of hierarchical menu systems. A menu is considered

to be composed of a set of selectable, bistable elements referred to as *menemes*[1]. The developers of Lean Cuisine observed that menu systems fall into two distinct categories; those in which only one choice could be selected at any one time, a 1 from N or mutually exclusive grouping, and those in which any number of the choices may be selected, an N from M or mutually compatible grouping. These two different group behaviours were then mapped onto a tree based representation in which mutually exclusive and mutually compatible groups were represented as vertical and horizontal branches respectively (Figure 2.10).

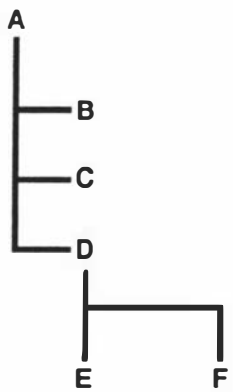


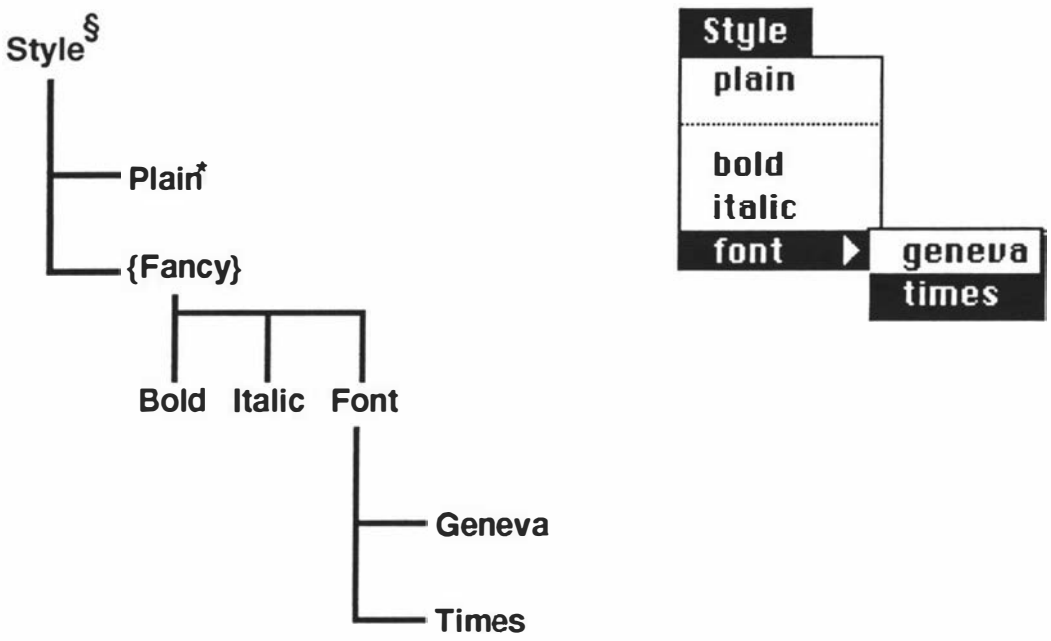
Figure 2.10: A simple Lean Cuisine Tree showing a mutually exclusive group B,C and D, and a mutually compatible group E and F.

It was further observed that although the group behaviour of all the menemes in a single displayable menu was often the same, this was not always the case. Sometimes mutually exclusive and mutually compatible groupings would be combined within the same menu[2]. To accommodate this within the notation, the concept of a virtual meneme was introduced. A virtual meneme is shown by enclosing the meneme name in braces (eg. {X}). Virtual menemes have no displayable counterpart, and hence cannot *directly* be selected or de-selected by the user. Selecting a child of a virtual parent causes the parent to become selected. Conversely, de-selecting all the child menemes of a virtual parent causes the parent to de-select. If the virtual parent is a member of a mutually exclusive group, then it may be de-selected in response to the selection of one

[1]Menemes are considered to be the basic unit of the menu input language. The term is considered as to have the same derivation as lexeme, phoneme etc.

[2]On a Macintosh the user's attention is drawn to such mixed groupings by separating them with a dotted line running across the menu.

of its siblings, which in turn would result in de-selection of all the virtual menemes children. Hence virtual menemes permit the propagation of state changes both up and down the tree (Figure 2.11).



(a) (b)

Figure 2.11: Lean Cuisine specification of a style menu (a) and its corresponding menu (b) .

Finally it was observed that it was sometimes necessary to have a default selection that would *automatically* become selected if all other choices became de-selected. This menu feature was subsequently modelled by marking the parent of such default selections with a '§' to signify a *required choice* grouping and marking the default meneme with a '*'. This feature is shown in Figure 2.11 in which the *Style* menu has a required choice of *Plain* if all other choices are de-selected.

Although Lean Cuisine was defined only for describing the behaviour of hierarchical menus, its extension to describing the behaviour of the brush shape dialogue is fairly straight forward. If we view the system as a set of task choices, then this permits the system to be modelled using the Lean Cuisine notation (Figure 2.12).

The important thing to recognise is that the state of the system is the set of currently selected menemes. Also certain states (eg; vthin and vthick) are

mutually exclusive. Selection of *OK* or *cancel* can be done at any time, hence these options are mutually compatible with respect to the others. The select brush shape dialogue effectively prevents access to all other dialogues within the system. This is represented in the example by making *Do_painting* and *Select_brush_shape* mutually exclusive to one another.

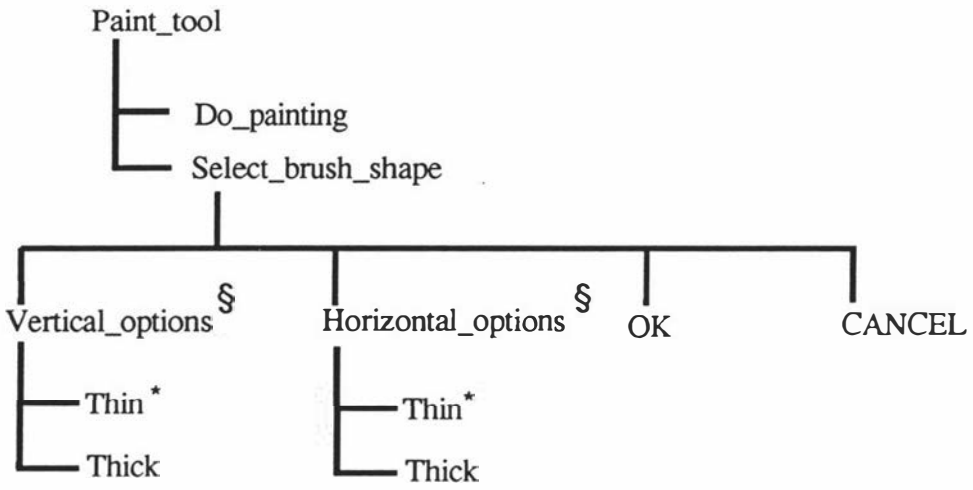


Figure 2.12: Using Lean Cuisine to model the brush shape dialogue.

The above specification can now be assessed in terms of the clarity and completeness of the specification:

clearly specified

This representation captures the basic behaviour of the system very well, in many respects this dialogue can be thought of as being a menu with a different layout. The emphasis of the Lean Cuisine notation is group behaviour and default selections, consequently the mutually exclusive behaviour of the brush width options and the definition of default options is very clear within the specification.

not specified

Lean Cuisine was developed for the specification of hierarchical menu systems. The low level details of user actions and resulting interface feedback are assumed to be pre-defined by the environment. Consequently the notation does not address the specification of these details.

The semantic response resulting from making a selection are not addressed, although as pointed out by Apperley and Spence [Apperley89;page 65], the

extension of the notation to include the linkage between changes in meneme state and the invocation of a specified application function is straight forward.

Putting the painting and brush shape dialogues in a mutually exclusive group is not quite correct from the Lean Cuisine viewpoint, because the model effectively only says that the user can be painting or selecting the brush shape at any given point in time, selecting one option having the effect of deselecting the other. A dialogue box however prevents the user the interacting with any other dialogue until the task associated with the dialogue box is completed.

2.3.3. GOMS

The GOMS (Goals, Operators, Methods and Selection rules) formalism [Card83] is based on a model of human information processing referred to as the *model human processor* (MHP). Underpinning the GOMS methodology is the assumption that the user's behaviour is dependent on the goals they wish to accomplish and the methods and operators available to them to achieve their goals. The user is modelled in terms of a goal (task) hierarchy and a set of operators available to carry out each sub-task. The methods to achieve each task are specified in a schema, for example, to achieve goal *X* carry out operation *a* followed by *b*. Finally the task breakdown is re-written annotated with the methods to achieve each sub-goal and conditions associated with each method to help decide which of a series of alternatives methods the user would use. GOMS is very much a method for modelling the users cognitive model of the interaction, but has been included in this overview for completeness. The ideas of GOMS have been further developed in Cognitive Complexity Theory (CCT) [Kieras85], a formalism based around the use of production systems.

2.3.4. Command Language Grammar

Command Language Grammar (CLG) [Moran81] was developed in the early 1980's as an approach to designing interactive systems. The psychological basis behind CLG is that it "describes the user's conceptual model of the system" [Moran81;page 5]. However the level of detail in the CLG model is such that it can be thought of as being a task-oriented model, a structural model and a dialogue model [Hartson89;page 33].

CLG is structured into three components:

- **Conceptual component**; defines the abstract concepts around which the system is organised.
- **Communication component**; defines the command language and conversation dialogue.
- **Physical component**; defines the physical devices that the user sees and comes into contact with.

Each of these components is described using a series of levels addressing increasingly fine detail. In the case of the conceptual component a task level describes the task domain addressed by the system and the semantic level describes the concepts represented by the system. Consider for example the specification of an interface for a clock design tool[3]. The task level could be described using a goal structure as shown in Figure 2.13. The semantic level could then be described using a programming language type of notation as shown in Figure 2.14. In this description the semantic entities, operations and methods of the system would be defined.

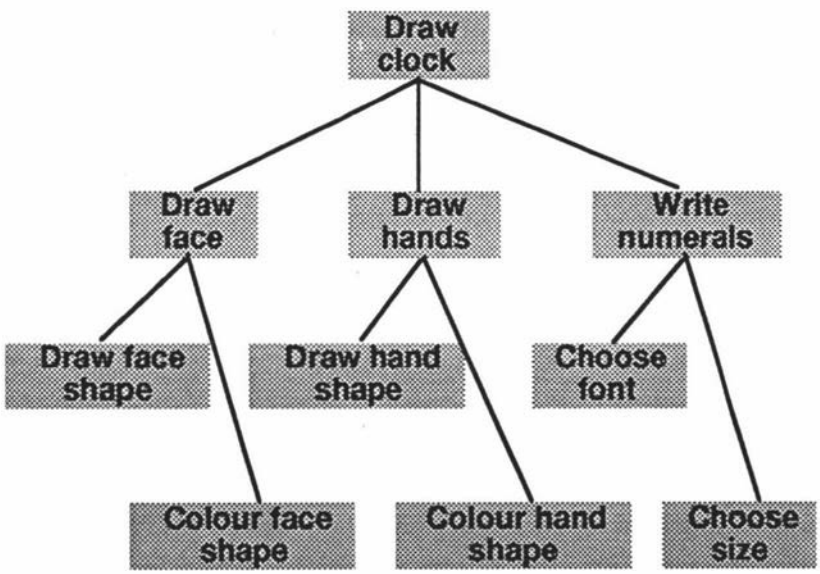


Figure 2.13: CLG task breakdown

[3]This example is taken from Johnson, P. (1992), *Human Computer Interaction: Psychology, Task Analysis and Software Engineering*, McGraw Hill.

```
DOODLE   =   (A SYSTEM
             NAME="DOODLE"
             ENTITIES=(SET:SHAPE TEXT LINES POINTS COLOURS)
             OPERATIONS=(SET:DRAW PAINT WRITE POSITION DELETE)
SHAPE    =   (AN ENTITY
             REPRESENTS (DRAW_SHAPE)
             NAME="shape")
TEXT     =   (AN ENTITY
             NAME="text"
             VALUE=(ONE-OF: NUMBERS LETTERS SPECIFIED_CHARACTERS))
                                  (a)

DRAW     =   (A SYSTEM OPERATION
             OBJECT = (A PARAMETER
                   VALUE = (AN ENTITY))
             IN (A PARAMETER
                   VALUE = (A PLACE ON SCREEN)
                   DEFAULT-VALUE=(UNKNOWN)))

PAINT    =   (A SYSTEM OPERATION
             OBJECT = (A PARAMETER
                   VALUE = (A SHAPE)))
                                  (b)
SEM_M1   =   (A SEMANTIC_METHOD
             FOR DRAW_FACE_SHAPE
             DO (SEQ: (START DOODLE_SYSTEM)
                   (DRAW SHAPE)
                   (STOP DOODLE_SYSTEM)))
                                  (c)
```

Figure 2.14: A CLG semantic level description, defining conceptual entities (a), operations (b) and a conceptual method (c).

In a similar fashion the communication component would be described in terms of a syntactic level that describes the command argument structure and an interaction level that describes the dialogue structure. The physical component was not fully defined by Moran, except to say that it would be specified in terms of spatial layout and device levels.

The CLG model is able to described an interactive system at a very high level of detail. Unfortunately this is its undoing. In attempting to provide linguistic, psychological and design views of the system the CLG description becomes extremely long, even the description of a simple system running to many pages. CLG was defined specifically for the purpose of specifying command language interfaces. As such it does not address issues such as temporal ordering or details of the users actions (the physical component being undefined). Consequently it is generally unsuitable for the specification of direct manipulation interfaces.

2.3.5. Notations for Comparison of Interactions

The analytical comparison of alternative interaction designs is an important activity in the design process. Given two alternatives the question arises as to how the designer can ascertain which is the better alternative. This in turn raises the question as to the basis for such a comparison. Two approaches have been used for this purpose, action analysis and usability comparison based on consistency.

Action analysis is based on the notion that the quicker the user is able to carry out a task, the better. This approach requires a technique for estimating the duration of a task from the description of the interaction. The Keystroke Level Model (KLM) [Card80] is based on the idea of breaking a task down into primitive steps and associating an empirically derived time duration with each step. Summation of the step duration's then gives an estimate of the total duration of the task. This form of model also needs to include a cognitive model of the user since mental processes take a significant amount of the overall time in carrying out tasks.

As an example, consider comparing the alternatives of setting the vertical brush thickness to thick in the brush shape dialogue using a function key or the dialogue box using a mouse respectively. Figure 2.15 gives a comparison of the two methods using KLM.

Function Key Method

Step	Operator	Time (sec)
Move hand to the keyboard	H[keyboard]	0.4
Mentally prepare	M	1.35
Press function key	K[function key]	0.28
	TOTAL	2.03

Menu Method

Step	Operator	Time (sec)
Point with mouse (to vertical thick option)	P[vthick]	1.1
Depress mouse button	B[down]	0.1
Mentally prepare	M	1.35
Release mouse button	B[up]	0.1
	TOTAL	2.65

Figure 2.15: Comparison of function key and menu methods using KLM.

This approach is very simple, but is surprisingly accurate. The negative aspect of the approach being that the description of each task is very detailed, rendering the comparison of two interfaces a very long and tedious undertaking.

One important point that can be seen in this example is that the model takes account of cognitive reasoning as an important process that takes a significant amount of time, mental preparation accounting for over 50% of the overall time in both of the above task methods.

Consistency based comparisons are based around the idea that consistency in an interface is very important. A user makes sense of what they see within an interface by trying to relate it to what they already know. If an interface is consistent then the user is able to extrapolate from their current knowledge of the interface and deduce the behaviour of commands that they have not used before. If the user discovers that a part of the interface does not behave in the same way as the rest of the interface then they have to remember this as an exception, a process known as accommodation. The fewer exceptions, then the lighter the memory load involved in using the system.

Having identified that consistency may indeed be related to usability, it is then necessary to identify a strategy via which a measure of consistency can be obtained. Early work on this was carried out by Reisner with her Task Action Language (TAL) [Reisner81]. TAL is a description of the interaction using BNF [Naur63]. To obtain a measure of consistency, Reisner used a series of metrics obtained by counting attributes of the grammar based description such as the number of different terminal symbols and the number of rules necessary to describe the structure of the interaction. Comparison of predictions based on such metrics against those obtained by empirical analyses showed some correlation between the two [Reisner81].

Task Action Grammar (TAG) [Payne86] uses a different technique to assess consistency. The interactions supported by the interface are described as a series of rule based schemas. Consistent behaviour is capture by means of generic schemas that effectively capture consistency within the interface. An example of this is given in Figure 2.16.

In BNF:

```
copy      ::=    'cp' + 'filename' + 'filename'
             |   'cp' + 'filename' + 'directory'
move      ::=    'cp' + 'filename' + 'filename'
             |   'cp' + 'filename' + 'directory'
link      ::=    'cp' + 'filename' + 'filename'
             |   'cp' + 'filename' + 'directory'
```

In TAG:

```
file-op[Op]  ::=   'cp' + 'filename' + 'filename'
                   'cp' + 'filename' + 'directory'

command[Op=copy]      :=    'cp'
command[Op=move]      :=    'mv'
command[Op=link]      :=    'ln'
```

Figure 2.16: Consistency of the Unix commands for copy, move and link as captured by TAG (example from [Dix93]).

Attempts have been made to extend TAG, so as to make it a tool useful for more than just interface evaluation. Early versions of TAG were simply for command language interfaces. A later version, Display Task Action Grammar (DTAG) [Payne91] includes facilities for graphical user interfaces.

2.4. SUMMARY

In this chapter a number of dialogue specification techniques used in the behavioural domain have been examined. Models in this domain place particular emphasis on what the user thinks, sees and does.

A range of modelling paradigms have been used. The importance of recognising the *purpose* of a given modelling approach was identified, and the observation made that models in this domain address a range of different purposes.

The only notations examined in this chapter that are really concerned with communicating designs are UAN, Lean Cuisine and CLG, and of these only

UAN is designed for describing direct manipulation interfaces. Lean Cuisine can be used for describing aspects of direct manipulation interfaces outside of its designated scope of hierarchical menu systems. A subject evaluation of the descriptive powers of UAN and Lean Cuisine suggests that weak aspects of UAN (defining group behaviour and default choices) are clearly represented in Lean Cuisine, suggesting that a composite modelling approach based on the two notations may have some merit.

Chapter 3
Dialogue Modelling in the Constructional Domain

3.1. INTRODUCTION

In this chapter dialogue models used in the construction of user interfaces are examined. Constructional dialogue models are defined for the purpose of implementing user interfaces. They are concerned with defining the structure of the user-computer interaction in a way that lends itself to the implementation of a software system capable of facilitating the specified interaction. Because of the difficulties of implementing such software, construction dialogue models are normally used in environments that support the automatic implementation of a user interface from the abstract dialogue model.

Two distinct forms of model can be identified within the constructional domain, ***structural models*** and ***dialogue models***. Structural models define the separation of user interface and the application as well as the components from which the user interface is constructed. Dialogue models define the structure of the communication between the user and the application.

3.2. STRUCTURAL MODELS

Structural models attempt to define an interface between the application and the interface and identify generic communication components. Such models can then be used as a basis for defining a run-time architecture for a UIMS. If a structural model is used as the basis for implementation it is called an ***architectural model***. The distinction between a conceptual architectural model and a physical architectural model has frequently been confused in the literature. Johnson [Johnson92] refers to conceptual models as ***frameworks*** and physical models as ***architectures***. Shevlin and Neelamkavil [Shevlin90] refer instead to logical and physical UIMS models. They identify that the primary

task of such models is to define the separation between the interface and the application. Bass and Coutaz [Bass93] identify two distinct types of architectural models, ***monolithic sequential architectures*** and ***multiagent architectures*** depending on the granularity of the architectural decomposition.

A number of different architectural models are possible depending on the underlying abstraction used [Took90]. These models include:

- Window managers; based on a data abstraction of windows and their operations.
- UIMS; based on a linguistic abstraction.
- Agents; based on the concept of an agent. An agent is an encapsulation of data, computation, input and output.
- Surface interaction; based on the factoring of the presentation of user interface components and operations on those components.

Window managers are far from being a complete model since they do not provide an abstraction for the content of windows. Of the other three architectural models, only linguistic and agent based models have been addressed to any great extent within the literature.

3.2.1. Linguistic based architectures

Early interfaces were generally of the command language type, and this resulted in a grammatical perspective of the user-computer interaction being used as the basis for separation and partitioning. Foley's language model [Foley80] partitioned the interface into lexical, syntactic and semantic layers similar to the structure of a compiler. The Seeheim model [Green85a] (3.1), takes a similar view of the interface with presentation, control and application layers roughly corresponding to lexical, syntactic and semantic roles respectively.

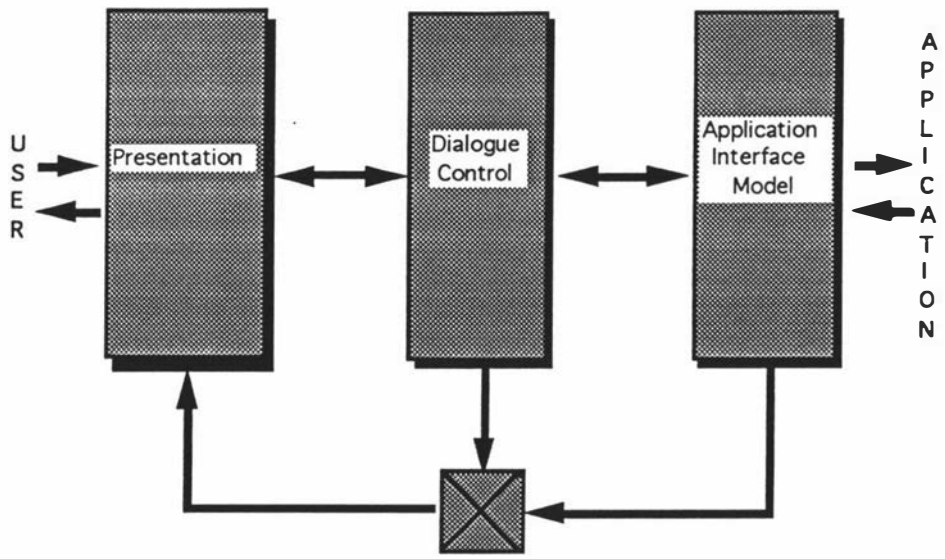


Figure 3.1: The Seeheim Model.

The layers present in the Seeheim model are:

The Presentation Layer. This is responsible for displaying the user interface and interpreting the users actions within the context of the display.

The Control Layer. This is responsible for modelling the dialogue between the user and the system.

The Application Interface Layer. This layer is responsible for linkage between the user interface and the underlying application functions.

The control layer is described by Green as being composed of "the main structuring mechanisms of the user interface, such as menus and command selection. The control model can be viewed as the user's model of the user interface." [Green85a]. In many UIMS based on the Seeheim architecture it is this component that has attracted the most attention [Green86]. A range of paradigms have been used to model the dialogue structure.

Such models have been referred to as monolithic architectural models [Bass91] since the whole of the interface is handled by each layer in turn. This type of architecture has been criticised for having an inappropriate granularity for graphical user interfaces [Coutaz90]. However Shevlin and Neelamkavil

[Shevlin90] identified that much of the criticism was due to the fact that UIMS logical architectural models (such as Seeheim) were usually mapped onto a corresponding physical architectural model at run-time. This in turn produced a physical separation of interface and application that gave rise to communication bottlenecks and other undesirable features. They pointed out that it was not necessary for a separation of use interface and application as defined in the logical model to be carried through to the physical model. The separation of user interface and application being defined in the logical model but not being carried through to the physical model they referred to as ***virtual separation.***

Hoffer [Hoffner89] has proposed viewing the interface as an incremental translator in which the language of the user is translated in a number of discrete steps into the language of the application. A corresponding feedback translation thread facilitates the communication of the application with the user. Unlike the Seeheim model, Hoffer's language model uses "as many steps as it takes" to achieve this translation.

3.2.2. Agent based architectures

Another approach that has been extensively used for interface abstraction is to view the interface as a number of interacting agents. An agent is a complete data processing system in its own right. It is able to maintain an internal state and responds to external stimuli by communicating its response to one or more other agents [Bass91]. The most common use of agents in interface design is in the form of toolkits.

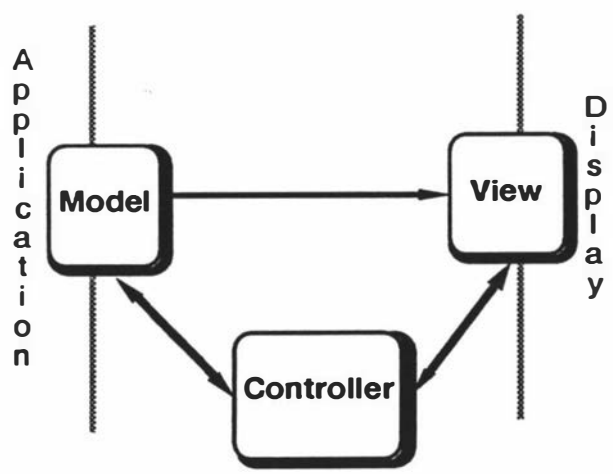


Figure 3.2: The MVC architecture.

Toolkits do not define an architecture. An agent based architecture however, structures the whole of the interface (and possibly the application itself) as a collection of interacting agents. Possibly the earliest example of a multiagent architecture is the Smalltalk Multi-View Controller (MVC) architecture [Goldberg84]. In MVC (Figure 3.2) three distinct types of agents with different responsibilities are identified:

- Model agents; these form the application interface.
- View agents; display a view of a model agent on the screen.
- Controller agents; interpret user input through the presentation of a view agent. Every view agent has a corresponding controller.

In response to user actions interpreted by a controller agent, the corresponding model agent may produce an appropriate change in the application. The model agent will then inform the view agent of this change and the view will be updated accordingly. In the case of complex interfaces, hierarchies of view-controller pairs can be defined.

Coutaz's Presentation, Abstraction and Control (PAC) architecture [Coutaz87] takes a different approach. In this architecture agents are subdivided on the basis of their function into:

1. **Presentation** (interaction) agents, defining the appearance of the interface, and handling user actions on the corresponding display objects (widgets). Hence a presentation agent in PAC corresponds to the combination of view and controller agents in MVC.
2. **Abstraction** (application) agents, defining the semantics.
3. **Control** agents that maintain the mapping between the presentation and abstraction agents.

In the example given in Figure 3.3 the application contains a set of values perhaps relating to temperature settings for an oven with minimum, maximum and current settings being 100, 300 and 150 degrees respectively. The presentation of these values is in the form of a dial. If the user changes the value by dragging the pointer on the dial then the control agent is responsible for informing the abstraction agent accordingly. Conversely, if the application changes the setting, then the control agent is responsible for informing the presentation agent to update the dial.

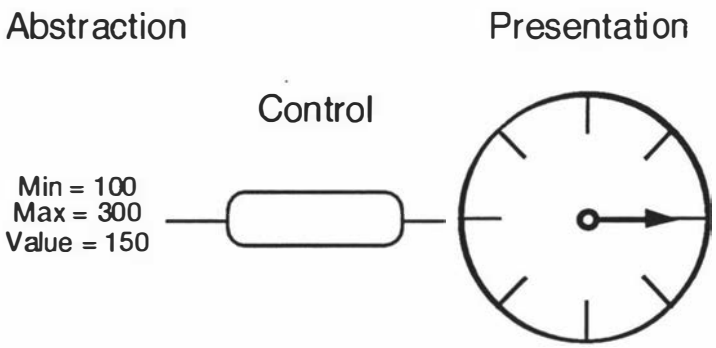


Figure 3.3: The PAC architecture (adapted from [Coutaz87]).

An application and its interface is normally composed of many agents. To handle this PAC hierarchically divides the application and its interface. A compound widget is viewed as being composed of several simpler widgets, each with their own presentation, abstraction and control agents (Figure 3.4), the hierarchy linkage being made through the control agents. In this way an interface can be hierarchically decomposed into simple components.

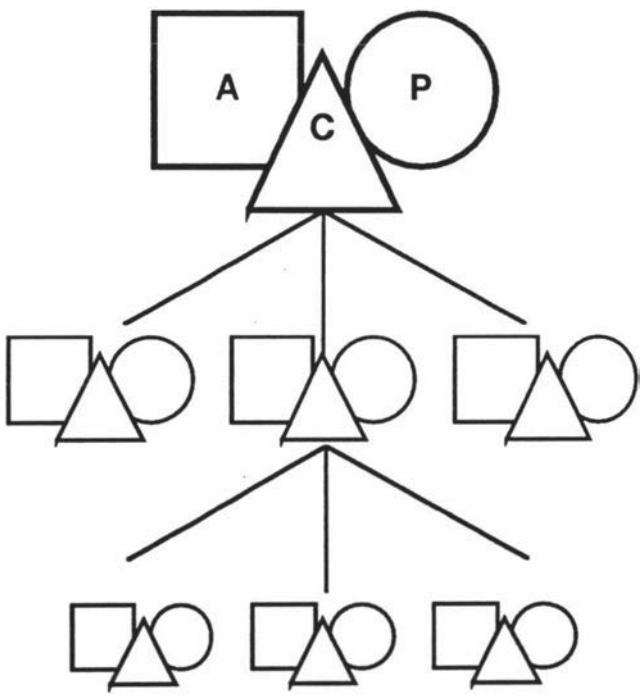


Figure 3.4: Hierarchical linkage of components through control agents in the PAC architecture (from [Bass91]).

3.2.3. Surface Interaction

Surface Interaction [Took90] is an architecture that achieves the separation of interface and application by factoring out the concepts of presentation and its manipulation. The presentation forms a *surface* shared between, and controlled by both the user and the application.

Presenter [Took90] is a UIDS that uses the surface interaction architecture. In Presenter, the shared surface is accessed through, and protected by, a server process accessible to both application and user. Presentation objects are based around regions, rectangular areas with a defined size, location. These regions are arranged into a tree, permitting arbitrary object groupings to be constructed. Regions occurring as terminal nodes can have an associated content (ie, appearance). A selection behaviour can be associated with a region, allowing a given region to be selected or to restrict selection access to other regions (for example, preventing the selection of one region lying underneath another). Finally, size and position constraints may be defined between regions, either through a child-parent relationship or through the use of linkage regions, introduced specifically for the purpose of imposing constraints.

3.3. DIALOGUE MODELS

Having defined an architecture, it is necessary to define the components within the architecture. The different approaches that can be used have been used as a basis for classifying UIDS [Myers89].

- Language based UIDS used a 'language' to describe the structure of the interaction. The language may be graphical.
- Graphical specification UIDS allow the user to define the components of the user interface graphically. The Peridot [Myers87] and Garnet [Myers90] UIDS fall into this category.
- Automatic generation UIDS: the user specifies the application semantics and the UIDS automatically generates the user interface. An example is Foley's IDL [Foley87;Foley89].

Considering just language based UIDS (which may be UIMS or agent based), the components may be specified using a conventional programming language. In the case of MVC and PAC, agents are simply instances of classes defined in an object-oriented language.

Having defined an architecture it is necessary to model the components identified. This could be done using a suitable programming language. Alternatively a more appropriate abstract notation can be used. Within a Seeheim architecture, the control layer, which models the structure of the dialogue with the user, has received particular attention, and a variety of Dialogue Specification Languages (DSLs) have been used for this. DSLs have been based on the following paradigms:

- Context free grammars
- State transition networks
- Event dispatch systems
- Process algebras

In the following sections these different categories of DSL are examined. Since they are primarily concerned with describing the syntactic level within the interface, user action and feedback issues tend to fall largely outside the scope of such models, lexical issues such as the being addressed by the incorporation of widgets from toolkits in the resulting software system.
Consequently, in contrast to the actual user actions referred to in the behavioural dialogue models, the constructional dialogue models refer instead to tokens generated within the lexical level in response to the users actions.
The brush shape dialogue described in Chapter 2 is used throughout this chapter to describe and compare the different types of DSL's. Common to all of these examples is the set of tokens originating from the lexical level of the interaction that 'drives' each model. The tokens are defined in Figure 3.5.

Token	Results from user clicking on...
SET_BRUSH_SHAPE	'set brush shape' button
VTHIN	vertical thin line button
VTHICK	vertical thick line button
HTHIN	horizontal thin line button
HTHICK	horizontal thick line button
OK	OK button
CANCEL	CANCEL button

Figure 3.5: The tokens defined for the brush shape dialogue.

3.3.1. Context Free Grammars

It is clear that for a sequential dialogue, the interaction between the user and the computer can be thought of as a conversation between the two, and hence can be modelled in terms of the structure of the language used by the two parties. This to some extent can even be applied to graphical user interfaces, in which the selection of icons etc, can be viewed as basic tokens from which the language of the interaction is composed.

A grammar is a set of rules which define valid constructions within a language. Grammars are well understood within the computing community due to work in the areas of programming languages, hence the use of grammars for defining the dialogue between human and computer was perhaps an understandable approach to take. A grammar is usually defined using a meta-language such as Backus-Naur Form (BNF) [Naur63]. In BNF a grammar is defined as a set of rules defining non-terminal symbols of a parse tree for the language.

Using this approach to define the dialogue between a user and the computer is very one sided in that it only defines valid input sequences from the user. Command language interfaces are characterised by the user entering a command in response to the system prompting the user. Hence a grammar based dialogue model has to be augmented to include the system side of the interaction. In the SYNGRAPH UIMS [Olsen83], the augmenting was done by associating Pascal code with each production. The need to define the system side of the interchange tends to make the dialogue specification far more difficult to read than the specification of a programming language in BNF. MIKE [Olsen86] and MIKEY [Olsen89] were later developments based on the same ideas.

Multi-party grammars [Shneiderman82] are also based on BNF with the following additions to the basic BNF structure:

- Non-terminals are labelled with a party; either human (H) or computer (C).
- Non-terminals can have a value associated with them.
- A default non-terminal can be defined, that will match any input string if no other parse match is possible.
- The inclusion of display related details.

An example of a multi-party grammar for an open file command is shown in Figure 3.6.

```
<CMD> ::= <H:OPEN><C:OPEN-ACK>
<H:OPEN> ::= OPEN <H:FILENAME>
<C:OPEN-ACK> ::= [<H:FILENAME>] IS OPEN
```

Figure 3.6: An example of the use of multi-party grammar.
(Non-terminals are identified by < >) (from [Shneiderman92])

The definition of languages using BNF is straight forward and a number of tools are available to support the implementation of parsers, for example Lex [Lesk75] and Yacc [Johnson75]. However the use of dialogue specification languages based on BNF is not without problems. One major problem is that representation based on meta-languages (ie. languages for describing other languages) such as BNF are "are almost unreadable to the average person" [Hartson89; page 25]. Comparisons of BNF and state based representations have shown that latter to be easier to use for dialogue specification [Guest82;Jacob83].

For direct manipulation interfaces the user of grammars suffers from major limitations. Interleaving and concurrency are important features of such interfaces. Unfortunately grammars are not able to cope with either. Consider interleaving, that is the ability for the user to switch between two or more dialogues as they wish, returning to an earlier dialogue at some future point, and then being able to carry on from where they had left off.

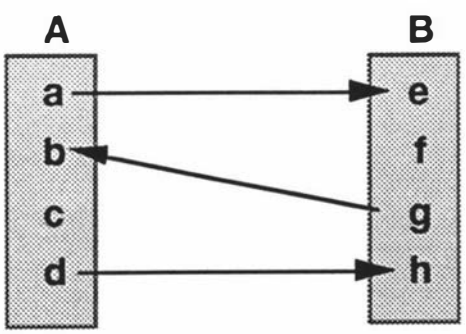


Figure 3.7: Interleaving.

Consider Figure 3.7 in which two separate dialogues, A and B are depicted. In order to carry out dialogue A the user must carry out actions a, b, c, and d (in order). Similarly dialogue B requires the user to carry out actions e, f, g and h. Support for interleaving would allow the user to carry out the two dialogues in the fashion indicated in the diagram, that is by carrying out actions a, e, f, g, b, c, d, h; or indeed any other combination of action sequences that preserve the ordering of actions for each dialogue. It follows that in order to model this with a grammar, a large number of rules are needed since there are a large number of valid action sequences (66 in the example). Although it might be possible to develop a tool that would construct all the rules need to recognise all the valid action sequences, it would be extremely difficult to develop a corresponding parser that could do so within acceptable time limits.

Grammars are unable to model concurrency. It follows that grammars are not a reasonable alternative for direct manipulation interfaces.

It should be pointed out, however, that for a command line style of interface the use of a grammar may well be the best suited. To quote Green "the user interface designer should be able to choose the type of notation that best suits the application at hand" [Green86].

Despite the shortcomings identified above, it is useful to consider the specification of the brush shape dialogue using a grammar representation. The terminal nodes within the specification represent lexemes within the system, a lexeme being produced by a lexical analyser that has parsed the user actions and generated the corresponding lexical tokens. Within the brush shape dialogue there are six terminal nodes corresponding to each of the buttons within the dialogue box.

```
Terminal_Nodes :
      { SELECT_VTHIN, SELECT_VTHICK, SELECT_HTHIN, SELECT_HTHICK, OK,
      CANCEL }
```

The specification is the brush shape dialogue is given within Figure 3.8. In this example annotation for addressing feedback and semantic linkage has been omitted.

```
PaintTask    ::=   Painting PaintTask
             |     SetBrushShape PaintTask

SetBrushShape ::= Select_Thicknesses OK
             |     Select_Thicknesses CANCEL

Select_Thicknesses      ::=   Select_VThickness Select_Thicknesses
      |     Select_HThickness Select_Thicknesses
      |     NULL

Select_VThickness ::=   SELECT_VTHIN
      |     SELECT_VTHICK

Select_HThickness ::=   SELECT_HTHIN
      |     SELECT_HTHICK
```

Figure 3.8: Specification of the brush shape dialogue using a context free grammar.

Reviewing the grammar based specification of the brush shape dialogue shown in Figure 3.8, the following observations can be made:

clearly specified

Repetition is modelled using recursive productions. For example, the rule Select_Thicknesses captures any sequence of selecting and re-selecting vertical and horizontal brush thicknesses, including the option of the user not doing anything. The behaviour of the dialogue box is captured in that the model is only able to define sequential behaviour, and consequently the definition of PaintTask implies that the SetBrushShape task has to run to completion before the next task is undertaken.

not specified

This specification does not address:

- the user actions. The terminal symbols (shown in upper case in the specification) would need to be defined within a lexical specification of the system.
- the mutually exclusive behaviour of the thickness selections within each orientation. The production for Select_VThickness for example states that the user can select thin or thick options, but does not state that selection of one causes the other to become deselected.
- feedback and semantic actions. As was seen in the SYNGRAPH example, grammars need to be augmented with this information.

- default selections. In order to define default selections it would be necessary to add a special initialisation rule to the specification, for example:

```
init ::= SELECT_VTHIN SELECT_HTHIN
```

and record the corresponding effects within the augmentation for that rule.

3.3.2. State Transition Networks

State Transition Networks (STN) in various forms have been used extensively as DSLs. The basic STN consists of a set of nodes representing states with arcs between them representing transitions. These nodes are generally marked with input tokens that will cause the transition to occur. Semantic routines can be associated with transitions or nodes.

	Operation	Action	State
	Start		1
1.	press the button to start pen tracking		2
2.	track pen to starting point of line		
3.	press button to fix starting point	store starting point	3
4.	track pen to end point	display line	
5.	press button to fix end point and stop tracking		1

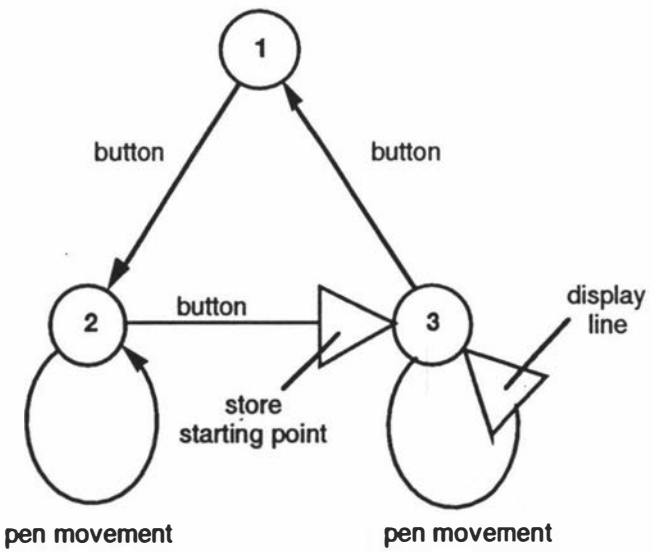


Figure 3.9: A state diagram representing rubber-band line-drawing using Newman's notation (taken from [Newman68]).

The earliest such use was by Newman with his Reaction Handler [Newman68]. Newman linked the dialogue to application functions through the states. Consider the drawing of a rubber band line being created with a pen and push-button. The sequence of operations to create such a line and the corresponding STN using Newman's notation are shown in Figure 3.9.

The large, labelled arrow heads in Figure 3.9 represent linkage to application routines that are called on entering a state. For more complex dialogues Newman identified the need for a conditional test to control branching to states. His notation supported the association of a test routine with an arc, with two or more outgoing arcs, the arc actually taken depending on the result of the test routine.

Simple STN tend to rapidly increase in complexity with minor additions to the system being modelled. Simply adding one more state can produce a substantial increase in the number of possible transitions. Interleaving of two dialogue with N and M states respectively requires a single STN with NxM states. One solution to this problem to reduce the number of states by introducing state variables. Augmented Transition Networks (ATM) [Foley90] associate conditional statements, defined in terms of such state variables, with transitions. This is similar to Newman's condition tests mentioned above.

Parnas [Parnas69] also employed simple STN to model dialogues. Unlike Newman, Parnas associated application functions with the transition arcs rather than the states. Arcs were labelled with the user input that would cause the transition and the resulting output.

Simple transition network models lack any approach to partitioning them into smaller sub-units. This lack of decomposition is a major limitation for describing all but the simplest dialogues. Denert [Denert77] introduced the idea of handling complexity within a STN dialogue model by breaking the STN down into a number of smaller networks, with one being able to call another in much the same way as a procedure is called in a conventional programming language. One STN was able to call itself, hence this notation is referred to as a Recursive Transition Network (RTN). The semantic linkage was achieved by the use of task nodes, which cause application routines to be called. The result

from calling these routines could control the next state to be entered. Denert's notation was subsequently extended by Edmonds [Edmonds81] by merging the RTN model with grammars to allow recognition of complex input objects.

Wasserman's RAPID/USE system [Wasserman85a,Wasserman85b] used a similar approach (Figure 3.10). One transition net was able to call another like a sub-routine, for example <add> in the figure refers to a sub-diagram called 'add'. Transitions could be labelled with the tokens that cause them (eg; '1', '2', 'help') and calls to application routines (the boxes labelled 'ca'). The calls to application routines were numbered to indicate which routine to call. Each STN would have associated annotation relating such things as numbers to application routines and prompts. As with Newman's notation, the values returned from the application routines could be used to decide on which branch to take. The token '+' is used to denote a transition without any input. Hence, in the example the application routine 2 is called immediately from the initial state 'setup'. RAPID/USE was designed for use with alphanumeric terminals.

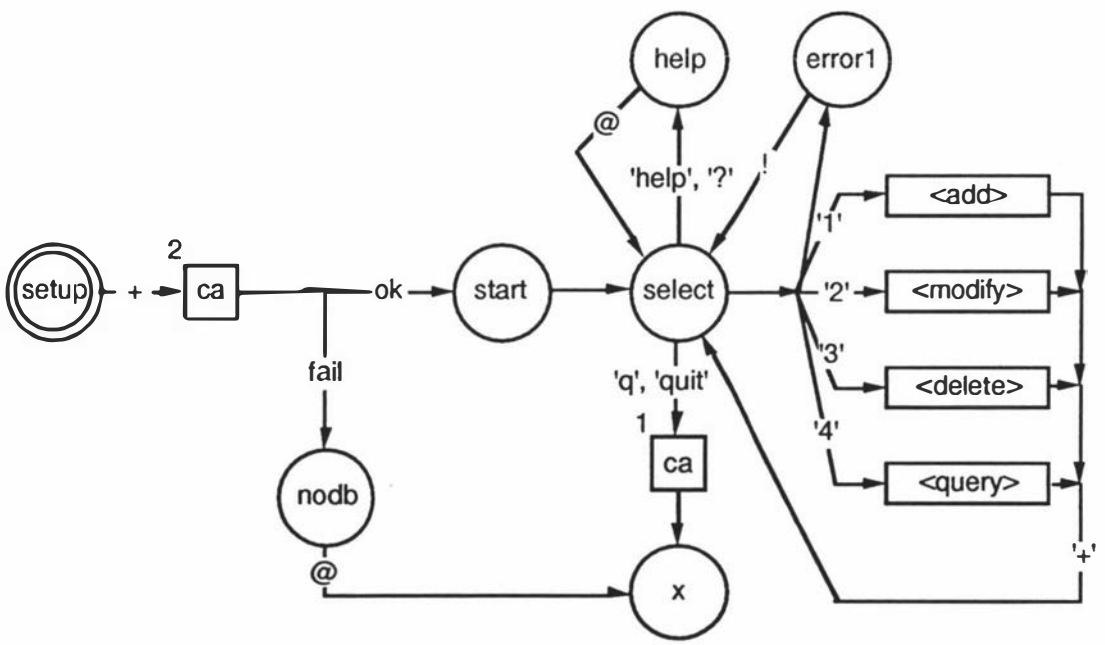


Figure 3.10: Example of a RAPID/USE STN (from [Wasserman85a]).

Jacob's UIDS [Jacob85] was designed specifically for graphical user interfaces. The interface was defined in terms of a number of interaction objects, the dialogue for each being defined using a STN (Figure 3.11). A top level executive was responsible for calling the appropriate STN in response to user input. The

STN was very similar to the others seen above in that transitions could be labelled with user inputs, condition routines and application routines. Additionally a prompt could be associated with a state, the prompt being output on entering the state. States within the dialogue specification could be labelled with '+' to indicate to the system that the dialogue could be suspended at this point, the state of the dialogue being saved.

Jabob's system also supported the definition of the lexical level using the same STN notation, although LISP could also be used for this purpose.

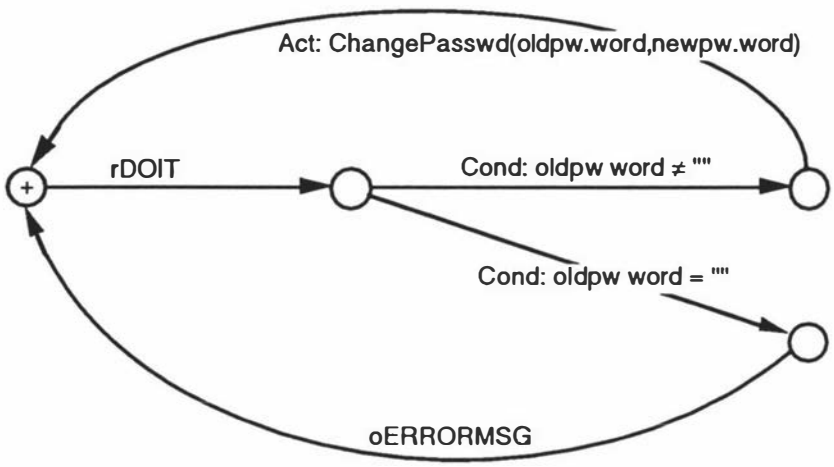


Figure 3.11: A dialogue for changing a password using Jacob's notation (from[Jacob86]). Of particular note are the conditions associated with transitions and the link to the application routine *ChangePasswd*.

This approach of defining a user interface as a collection of objects with their associated dialogues and remembered states was able to cope with interleaving, however it was not possible to model true concurrency using this system. A similar approach was subsequently used in HutWindows, a UIMS from the Helsinki University of Technology [Koivunen88].

The specification of the brush shape dialogue using a STN is given in Figure 3.12. In a STN the dialogue is modelled in terms of states, events, transitions and actions.

States = { {vthin+hthin},
{vthin+hthick},
{vthick+hthin},
{vthick+hthick}
}

actions = { hilite_vthin, hilite_vthick, hilite_hthin, hilite_hthick,
dehilite_vthin, dehilite_vthick, dehilite_hthin,
dehilite_hthick, show_dialogue_box, hide_dialogue_box }

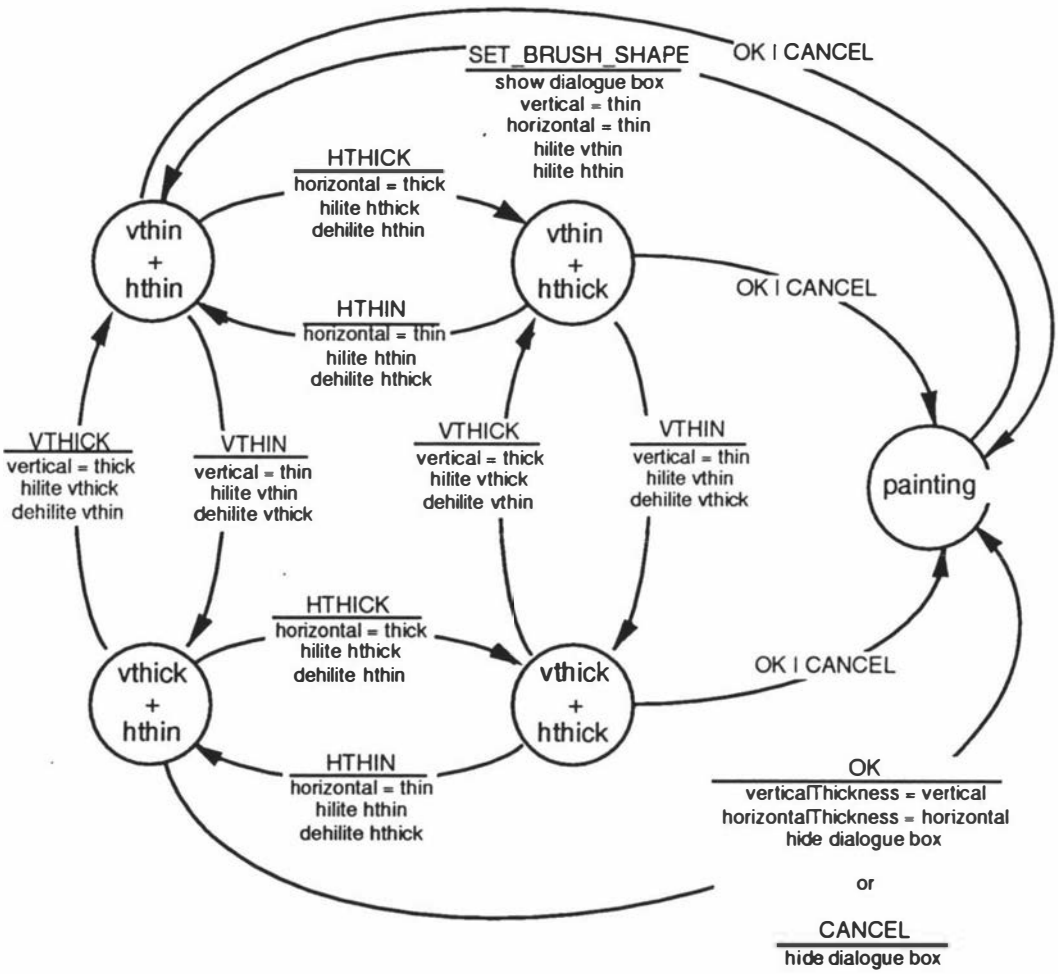


Figure 3.12: The brush shape dialogue modelled using an STN.

Reviewing the STN based specification of the brush shape dialogue shown in Figure 3.12, the following observations can be made:

clearly specified

As was the case with the grammar based specification, because of the sequential nature of the notation the user is only able to re-enter the paint state

after selecting OK or CANCEL from the dialogue box, this being the desired behaviour.

poorly specified

The mutually exclusive behaviour of brush thickness options is captured in terms of the valid states of the system (ie. state vthin + vthick is invalid). This tends to hide the group relationship.

not specified

The STN only defines the control layer, input and output aspects of the presentation layer are effectively ignored. With respect to input, the tokens that cause the transitions within the STN are assumed to be produced using a suitable lexical analyser (just as was the case with the grammar based model). Output can be partially addressed by labelling the transitions with actions that can produce changes in the presentation (eg; hide-dialogue-box and hilite-hthin), but aspects such as layout are not addressed.

3.3.3. Statecharts

Each of the above state based notations lack suitability for direct manipulation user interfaces since they are unable to model concurrent dialogues. Harel developed a new state based notation known as statecharts [Harel84,Harel87,Harel88] to overcome this important restriction in conventional STN models.

Statecharts add three key concepts to the conventional STN model:

- Grouping; several states with the same transitions, triggered by the same event can be grouped together and a single transition specified for the group as a whole. For example, the simple STN in Figure 3.13(a) can be replaced by the equivalent statechart in Figure 3.13(b), in which the states *A* and *B* are grouped together and the transition to state *C* in response to receiving event *x* is collapsed into a single transition.
- Concurrency; groups of states can be combined using XOR or AND groupings. In an XOR grouping, only one of the group of states is active if the parent is active. In an AND grouping all states within a given active group will be active. The AND groups represent concurrency. In Figure 3.13(c), states *A* and *B* form an XOR grouping, the group *A* and *B* forming a semi-autonomous state machine. The two state machines *{A,B}*

and *{C,D}* are separated by a dashed line. The identifies that the two state machines form an AND grouping and hence are concurrent with respect to one another, the super-state demo being composed of the combined states of the two state machines. The small arrows going into states *A* and *C* at the top of the diagram show that states *A* and *C* are the start states of the system, hence if the initial state of the overall machine is *{E}* and the system receives event *x*, then the resulting state will be *{A,C}*.

- Event broadcasting; transitions can be labelled with both an event that causes the transition and an associated *output* event. These output events are broadcast and can cause further transitions to occur. Referring again to Figure 3.13(c), the transition from state *A* to *B* is caused by event *b*. In response to this transition, the event *c* will be broadcast. This in turn can cause the transition *D* to *C*. It follows that if the system is in state *{A,D}* and receives event *b*, then the final state of the system will be *{B,C}*.

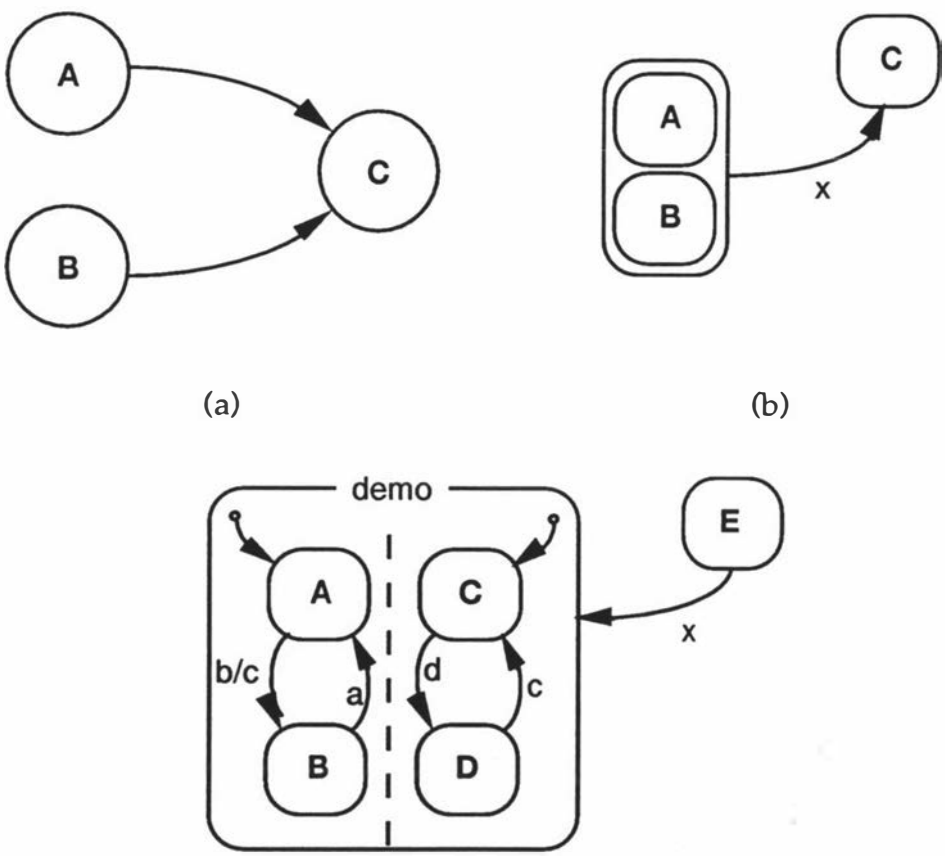


Figure 3.13: Statechart semantics.

Statecharts were originally developed for the design of real-time systems, however statemaster, a user interface development system based on the notation has been described [Wellner89]. The original statechart notation does not address the issue of linkage to application routines. To address this, the statechart formalism was extended for statemaster, each state having a set of actions associated with entering and leaving the state. The statechart for the brush shape dialogue is presented in Figure 3.14. In this diagram, on entering state *hthin* the actions *hilite_hthin* and setting the variable *horizontal* to *thin* would be executed. On leaving the state *hthin*, the action *dehilite_hthin* would be executed.

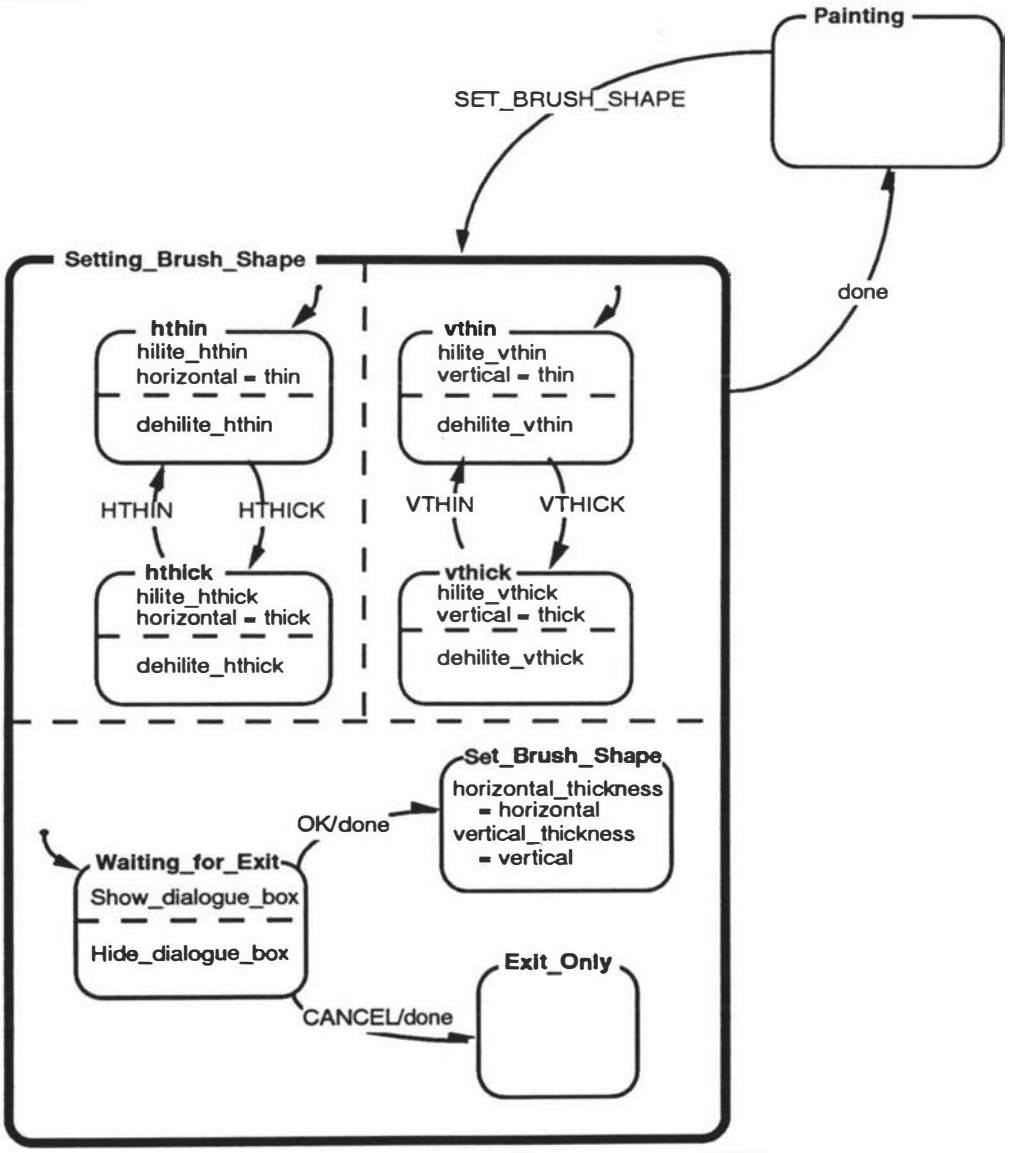


Figure 3.14: Statechart of brush shape dialogue.

Reviewing the statechart based specification of the brush shape dialogue shown in Figure 3.14, the following observations can be made:

clearly specified

As was the case with the grammar based specification, because of the sequential nature of the notation the user is only able to re-enter the paint state after selecting OK or CANCEL from the dialogue box, this being the desired behaviour. The resulting specification is considerably simpler than the equivalent STN specification shown in Figure 3.12, the 13 transitions in the STN reducing to 8 transitions within the statechart.

poorly specified

The mutually exclusive behaviour of brush thickness options is captured in terms of the valid states of the system (ie. state vthin + vthick is invalid). This tends to hide the group relationship.

not specified

The STN only defines the control layer, input and output aspects of the presentation layer are effectively ignored. With respect to input, the tokens that cause the transitions within the STN are assumed to be produced using a suitable lexical analyser (just as was the case with the grammar based model). Output can be partially addressed by labelling the transitions with actions that can produce changes in the presentation (eg; hide-dialogue-box and hilite-hthin), but aspects such as layout are not addressed.

3.3.4. Event Dispatch Models

Event Dispatch Models (EDM)[1] are another formalism capable of describing concurrency. The traditional EDM partitions the dialogue into a number of discrete event handlers. These autonomous agents receive stimuli in the form of events and may generate further events in response that are in turn received by other agents. Within the body of one of these event handlers various processing may be carried out, for example modifying state variables or invoking presentation or application functions.

---

[1]These have variously been referred to in the literature as Event Dispatchers, Event Models and Event Driven Models. I have used the term Event Dispatch Model to distinguish this type of model from STN which are also event driven.

One of the first such systems to be described was the Input-Output Tool (IOT) model of Van Den Bos et al. [Bos83]. The IOT model was based on the idea of defining a user interface in terms of a number of 'tools' which were capable of recognising given input characters or sequences thereof. On recognising an input, a tool would execute an internal procedure and produce some form of output. The tools of the IOT model were essentially input handlers, and as such only differed from the EDM model in that they dealt with both the lexical and syntactic levels within the interaction.

```
EVENT HANDLER line;
      TOKEN
            button Button;
            move Move;
      VAR
            int state;
            point first, last;

      EVENT Button Do {
            IF state == 0 THEN
                  first = current position;
                  state = 1;
            ELSE
                  last = current position;
                  deactivate( self );
            ENDIF;
      };

      EVENT Move Do {
            IF state == 1 THEN
                  draw line from first to current position;
            ENDIF;
      };

      INIT
            state = 0;

END EVENT HANDLER line;
```

Figure 3.15: Event handler for a rubber band (from [Green86]).

Green describes the more typical form of EDM in a survey of dialogue models [Green86]. A typical example is given in Figure 3.15. An event handler is characterised by having local variables (state, first and last in the example) and a number of handlers that are executed if the handler receives an event (token) matching that specified in the header of the handler. A special handler, INIT is executed in instantiating the handler and serves the purpose of initialising any internal variables. The important point to note about event handlers is that they

are autonomous, and could be executed in parallel. It follows that such a model is capable of describing concurrent behaviour.

A variation on this traditional form of EDM are Event Response Systems (ERS) [Hill87a]. Based on the concepts of production systems from the expert system arena [Hopgood80,Rich83], the ERS model has three components:

- a Boolean flag memory for maintaining system state.
- a set of rules in terms of current event and system state.
- an execution engine that for each event in turn scans the rules and marks those that can be fired.

An ERS has two distinct forms of rule, regular rules and ε-rules.

$\sigma : F_1 \rightarrow F_2$ (regular rule)
$F_1 \rightarrow F_2$ (ε-rule)

The interpretation of a regular rule is that if the current event matches σ and all the flags in set F_1 are set, then the rule can marked for firing. On executing the rule all the flags in set F_1 are cleared and all the flags in set F_2 are set. ε-rules are exactly like regular rules except that there is no event to be matched. An ERS specification is executed as shown in Figure 3.16.

```
REPEAT FOREVER
        mark all fireable ε-rules
        WHILE there are marked ε-rules DO
                fire the marked rules
                mark all fireable ε-rules
        END WHILE
        get the next event
        mark all fireable regular rules
        fire the marked rules
END REPEAT
```

Figure 3.16: Execution cycle for an ERS.

Event Response Language (ERL) [Hill87a,Hill87b] is a dialogue specification language based on the ERS model. Associated with the ERS rules are actions such as setting a flag (eg; $x\uparrow$ denotes setting flag x) and issuing an event (eg; $y!$ denotes issuing event y).

Rule	ERL	Comment
1	INITIALLY ->	Initialise
	getCmd ↑	set flag getCmd
	getArg ↑	set flag getArg
2	Command getCmd ->	(Event==Command and flag getCmd set) ... clear flag getCmd
	Process.command <- Command.name	Record the command
	cmd ↑	set flag cmd
3	Argument getArg ->	(Event==Argument and flag getArg set) ... clear flag getArg
	Process.argument <- Argument.name	Record the argument
	arg ↑	set flag arg
4	--- cmd arg ->	ε-rule if (cmd and arg are both set)
	Process !	Issue event Process
	waitProcessing ↑	set flag waitProcessing
5	DoneProcessing waitProcessing ->	(Event==DoneProcessing and flag waitProcessing set) ... clear flag waitProcessing
	getCmd ↑	set flag getCmd
	getArg ↑	set flag getArg

Figure 3.17: A simple command processor in ERL. (based on an example in [Hill87a])

Figure 3.17 shows and example in ERL of a dialogue in which the user needs to enter both a command and an associated argument. Only after both have been entered can the command be processed. The command and argument could be supplied in any order. When the system is first started the initialisation rule (1) is fired and flags *getCmd* and *getArg* are both set. This effectively enables rules (2) and (3) which are responsible for receiving commands and arguments respectively. If a command is received, rule (2) is marked and fired. This results

in flag *getCmd* being cleared and flag *cmd* being set. After both a command and an argument have been received, flags *cmd* and *arg* are both set, and the ε-rule (4) will be marked and fired. This results in the event Process being emitted (with fields for the command and argument set appropriately), and flag *waitProcessing* being set. On processing being completed, the event *DoneProcessing* will be issued by the processing agent, and rule (5) will be fired, set flags *getCmd* and *getArg* which enable rules (2) and (3) again. The process can then repeat.

A number of UIDS have been based around the EDM approach. Algae [Flecchia87], part of the University of Alberta UIMS [Green85b] uses event handlers specified in an extension of the C programming language [Kernigan78] to define the control layer within a Seeheim architecture. Sassafras [Hill87a and Hill87b] was a prototype UIDS based on ERL.

The brush shape dialogue can readily be modelled in ERL as shown in the Figure 3.18 (a) and (b).

.....the paint module.....

```
INITIALLY ->
      WaitPaintCmd^

...various rules for paint commands....

paintCmd WaitPaintCmd ->            // received a paint command
      doPaintCmd!                   // do it
      WaitProcessingCmd^            // ...and wait for it to complete

paintCmdDone WaitProcessingCmd ->   // completion of painting command
                                    // notified
      WaitPaintCmd^                 // wait for the next paint
                                    // command

setBrushShape WaitPaintCmd ->       // request to change brush shape
      doSetBrushShape!              // invoke set brush shape dialogue
box
      WaitSetBrushShape^            // wait for completion

doneSetBrushShape WaitSetBrushShape ->    // completion of set brush
                                          // shape notified
      WaitPaintCmd^                 // wait for next paint command
```

Figure 3.18 (a) : Specification of the brush shape dialogue in ERL
The paint module.

.....the dialogue box module.....

```
doSetBrushShape ->                  // request to change brush shape
      vthin^                        // set default vertical and
      hthin^                        // horizontal thicknesses
      showDialogueBox!              // make the dialogue box visible

selectVThin vthick ->               // deal with selection of line
      vthin^                        // thicknesses

selectVThick vthin ->
      vthick^

selectHThin hthick ->
      hthin^

selectHThick hthin ->
      hthick^

cancel ->                           // exit dialogue box with making
                                    // changes
      doneSetBrushShape!            // communicate completion to the
paint
      hideDialogueBox!              // module

OK vthin ->                         // exit dialogue box, making
      doSetVThin!                   // changes
      WaitSetVertical^

OK vthick ->
      doSetVThick!
      WaitSetVertical^

OK hthin ->
      doSetHThin!
      WaitSetHorizontal^

OK hthick ->
      doSetHthick!
      WaitSetHorizontal^

DoneSetVertical WaitSetVertical ->  // detect completion of process
      doneVertical^                 // for setting vertical brush
                                    // thickness

DoneSetHorizontal WaitSetHorizontal ->    // ditto for horizontal
      doneHorizontal^                     // thickness

--- doneVertical doneHorizontal ->  // E rule fired on completion of
                                    // setting brush shape
      doneSetBrushShape!            // Notify and reactivate
                                    // the paint module
      hideDialogueBox!              // hide the dialogue box
```

Figure 3.18 (b) : Specification of the brush shape dialogue in ERL.
The dialogue box module.

Reviewing the ERL based specification of the brush shape dialogue shown in Figure 3.18, the following observations can be made:

clearly specified

- It captures he separation of the paint and set brush shape dialogues, with the user only being able to interact with the paint interface after the dialogue box has been made to go away.
- The mutually exclusive behaviour of thin and thick line selections is modelled effectively using twinned rules with shared Boolean flags.
- Selecting of default brush thicknesses is handled well by the initialisation handler for the dialogue box.
- Feedback and semantic links are explicitly stated using internal events, for example *doSetVThick* and *hideDialogueBox*.
- The model is able to represent the concurrent update of vertical and horizontal brush thicknesses, the synchronisation of their completion being achieved by the use of an ε-rule.

poorly specified

In ERL this system would be looked at in terms of two separate modules, one for the paint environment, the other for the set brush shape dialogue box. The two would be made mutually exclusive by means of two condition flags.

not specified

What the model does not capture is the low level user actions that give rise to the user generated events such as *OK*. Another problem is that the mutually exclusive structure of buttons is largely hidden within the rule structure.

3.3.5. Propositional Production Systems

Propositional Production Systems (PPS) [Olsen90] are a superset of state machines and EDM. A PPS consists of a state space consisting of groups of mutually exclusive Boolean flags, and a set of rules for transforming the state space. Consider Figure 3.19.

```
Input Field Input( NullEvent, MouseDown, MouseUp, LineMenu,
      CircleMenu, RectMenu, DelMenu)
Field ActiveCommand( NoCmd, LineCmd, CircleCmd, RectCmd )
Semantic Field SelectedObject( NoSelection, LineSel, CircleSel,
      RectSel )
Field DragMode( NotDragging, DragSelected )
```

(a)

```
NoCmd, LineMenu ->
      LineCmd, NotDragging
NoCmd, CircleMenu ->
      CircleCmd, NotDragging
```

(b)

Figure 3.19: A Propositional Production System (part), with a state space (a) and some of its associated rules (b). (adapted from [Olsen90]).

Field Input identifies a set of Boolean flags corresponding to all input events for the system. If a *MouseDown* event is generated then the *MouseDown* flag would be (automatically) set. This can then be used as a condition on a rule. Consider the first rule in Figure 3.19(b). This states that if the flags *NoCmd* and *LineMenu* are both set then the *LineCmd* and *NotDragging* flags are to be set (which would automatically unset any other flags in the corresponding fields, since the groupings are mutually exclusive).

3.3.6. Process Algebras

The importance of concurrency in direct manipulation interfaces has already been discussed. In recognition of this, techniques for modelling systems as a set of concurrent processes have been used for dialogue modelling. These techniques include Communicating Sequential Processes (CSP) [Hoare95], Calculus of Communicating Systems (CCS) [Milner80] and Petri Nets [Peterson77,Reisig85]. A number of dialogue languages and user interface development systems have been developed based on these process algebras. These include SPI [Alexander87, Alexander90] based on CSP, Squeak [Cardelli85] based on CSP and CCS, Abowd's Agents based on CSP [Abowd90] and SCENARIOO [Roudaud90] which uses event graphs based on Petri nets.

Hoare's CSP is based on two classes of entities, events and processes. Process constructs supported by CSP are summarised in Figure 3.20. Note that since (e -> P) is a process, if e_1 and e_2 are events, (e_1 -> (e_2 -> P)) is also a process. The inner parentheses can be omitted to give (e_1 -> e_2 -> P).

expression	meaning	explanation
(e -> P)	prefix	do event e, then behave like P
P [] Q	choice	behave like process P or behave like process Q
P ; Q	sequence	behave like process P then (when P has completed) behave like process Q
P || Q	parallel	behave like P and behave like Q, synchronising on common events
SKIP	termination	a process which indicates successful completion

Figure 3.20: CSP process expressions.

Consider a simple example (from [Hoare95]), a vending machine that waits for a coin to be inserted, then dispenses a chocolate, and then repeats the process again, waiting for the next coin. In CSP such a vending machine could be described in the following way:

```
VEND = ( coin -> ( choc -> VEND ))
```

By convention, processes are in upper case and events in lower case. Repetition is handled in CSP by processes recursively calling themselves (either directly or indirectly via another process).

SPI, an interface prototyping environment based on CSP was developed by Alexander [Alexander87]. Dialogue design in this environment is a two stage process:

1. The structure of dialogues is defined in terms of primitive steps (referred to as events), using eventCSP, a subset of CSP.

2. The events are defined using eventISL.

Consider Figure 3.21 which contains part of the specification for a dialogue for an automatic teller machine. The structure of the dialogue is given in (a)

consists of a list of dialogue events such as put-in-card and read-card. The expression e -> P means deal with event e, then behave like process P. Process P may well be a further sequence of events to be processed. The [] symbol represents choice, hence the interpretation of the second clause of the ATM dialogue specification is read-card or cannot-read (card) or stop-atm. Non-terminal symbols (eg; ATM1 in the ATM dialogue specification represent further expression.

Figure 3.21(b) shows how an event is defined using eventISL. The purpose of this definition is to specify when the event is available and to associate any actions that go with the event. Each dialogue has an associate state and the availability of an event is defined as a condition in terms of a predicate on the dialogue state. The when clause in (b) defines that the event thief? is only available if the number of tries (at entering the PIN number) is >= 3. Inputs, outputs and state transformations may also be associated with an event definition.

```
ATM = ( put-in-card ->
            ( read-card -> ATM1
            [] cannot-read -> eject-card -> take-card ->
ATM
            [] stop-atm -> SKIP ))

ATM1 = ( enter-pin ->
            ( pin-ok? -> TRANSACTION
            [] pin-not-ok? ->
                  ( try-again? -> ATM1
                  [] thief? -> ATM )))
                      (a)
event thief? =
use tries in
      when tries ≥ 3
      out "Card retained; please see the manager"
                      (b)
```

Figure 3.21: Part of the SPI specification of a dialogue for an ATM. (a) Dialogue definitions in eventCSP and (b) Definition of an event in eventISL. (Adapted from [Alexander90]).

The dialogue example given in Figure 3.21 is purely sequential, the specification appearing very similar to a specification in BNF. However, the real strength of using CSP is that it is able to model concurrency.

```
Mouse = ( press -> get-posn -> send-posn -> release ->
Mouse )

Kbd = ( get-char ->
            ( newline? -> send-line -> Kbd
            [] text-ch? -> add-to-line -> Kbd ))

Text = ( send-posn -> save-position -> Text
      [] send-line -> write-line -> Text )

INPUT = ( Text || Mouse || Kbd )
```

Figure 3.22: Concurrent dialogues in eventCSP. (Adapted from [Alexander87])

Consider the example in Figure 3.22. Mouse, Kbd and Text are dialogue processes for handling Mouse inputs, Keyboard inputs and the accumulation of text characters in a buffer. The expression a || b means execute processes a and b in parallel. Hence the definition for INPUT specifies that each of these input processes is executed concurrently. An important consideration in executing processes concurrently is synchronisation. CSP achieves this by synchronising processes on common events, hence in the Mouse, Kbd and Text example, the Mouse and Text processes would synchronise on the event *send-posn*.

Using eventCSP, the brush shape dialogue can be modelled as shown in Figure 3.23. Reviewing the CSP based specification of the brush shape dialogue shown in Figure 3.23, the following observations can be made:

clearly specified

- CSP captures the separation of the paint and set shape dialogues well. The user is only able to resume interaction with the paint process after the set shape process has terminated.
- Sequences are captured well, with the specification appearing not dissimilar to BNF, particularly in the way that recursion is used for repetition.
- Concurrency is clear and straightforward to specify, for example defining the concurrent invocation of the application functions to set vertical and horizontal brush thicknesses.

```
PAINT = (   paint-cmd -> DO-PAINT-CMD; PAINT
          [] set-shape-cmd -> SET-DEFAULTS; SET-SHAPE; PAINT
          [] quit -> SKIP )

SET-SHAPE = (     select-vthin -> VTHIN-SELECT; SET-SHAPE
                [] select-vthick -> VTHICK-SELECT; SET-SHAPE
                [] select-cancel -> CANCEL; SKIP
                [] select-ok -> OK; SKIP )

SET-DEFAULTS = ( default-vthin -> default-hthin -> SKIP )

VTHIN-SELECT = (  select-vthin -> highlight-vthin ->
                        dehighlight-vthick -> SKIP )

VTHICK-SELECT = ( select-vthick -> highlight-vthick ->
                        dehighlight-vthin -> SKIP )

HTHIN-SELECT = (  select-hthin -> highlight-vthin ->
                        dehighlight-hthick -> SKIP )

HTHICK-SELECT = ( select-hthick -> highlight-vthick ->
                        dehighlight-hthin -> SKIP )

CANCEL = ( hide-dialogue-box -> SKIP )

OK = ((SET-VERTICAL-THICKNESS || SET-HORIZONTAL-THICKNESS);
      hide-dialogue-box -> SKIP )

event default-vthin =
      out highlight-vthin
      set vertical = thin

event default-hthin =
      out highlight-hthin
      set horizontal = thin

event select-vthin =
      set vertical = thin

event select-vthick =
      set vertical = thick

event select-hthin =
      set horizontal = thin

event select-hthick =
      set horizontal = thick
```

Figure 3.23: Specification of the brush shape dialogue in CSP.

poorly specified

- The mutually exclusive grouping of thick and thin options in each orientation is not clear, but is simply implied by the sequence of highlighting and dehighlighting events.

- The state changes in the system are recorded within the event specifications. This separation of sequence and associated state changes requires effort to relate the two together.

not specified

As was the case with grammars and ERL, user actions and display layout and appearance are not addressed.

A general criticism of CSP is that the clarity of the specification is very much dependent on the use of descriptive names.

3.3.7. Other methods

Apart from the dialogue modelling approaches described above, a number of composite approaches have been used. As discussed under structural models, agent based models have gained considerable acceptance in recent years. The dialogue modelling approaches discussed above can be used to specify the dialogues that a given object in the interface is to support.

Penguin [Yap90] uses a grammar based language to model agent dialogues. At run-time, individual agents can be scheduled asynchronously, a separate parser being used for concurrently interacting agents. DIWA [Six90] uses a similar approach with event handlers being defined within an object-oriented framework, and the behaviour of each event handler defined using STN. The TUBE UIDS [Hill89,Hill90,Kuntz90] uses a so-called Composite Object Architecture based on the concept of User Interface Objects, the behaviour of individual objects is defined using a variant of ERL.

3.4. SUMMARY

In this chapter a number of constructional dialogue models have been examined. It has been observed that in order to construct an interface it is necessary to consider two classes of model; structural models that define the architecture of the interface and dialogue models that define the structure of the interaction between the user and the computer. In contrast to the behavioural dialogue models examined in Chapter 2, dialogue models within the constructional domain primarily concentrate on a syntactic view of the

interaction, the recognition of user actions being subsumed into a lexical layer assumed to be constructed from appropriate toolbox routines.

Finally, it has been noted that only statecharts, event dispatch models and process algebras are really suitable for the specification of direct manipulation interfaces, since only these notations are able to describe systems supporting interleaving and concurrency.

Chapter 4
Behavioural to Constructional Model Translation

4.1. INTRODUCTION

In this chapter the problem of translating a dialogue model in the behavioural domain into an equivalent constructional model is examined. First a simple meta-model of the translation process is discussed. The translation of a Lean Cuisine specification to its constructional equivalent is then described. Finally some conclusions are drawn from this example.

4.2. MODEL TRANSLATION

Given two models, M_1 and M_2, and the task of translating M_1 into M_2, how in a general sense can this be achieved? The translation can be thought of as a process that takes as its input the ***information*** contained in M_1 and ***restructures*** it to produce the new representation M_2. If all the information required to construct M_2 is contained in M_1, it follows that the translation process is simply a re-write system that could perhaps be automated. Such a translation would probably be classed as "straight forward" (Figure 4.1).

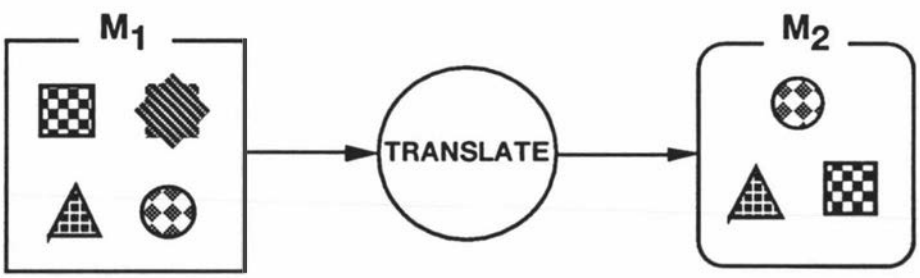


Figure 4.1: A Simple Translation.

It may however be the case that some of the information contained in M_2 is not present in M_1. As a consequence, such a translation requires some additional information to be entered into the translation process from some external

source (Figure 4.2). Such translations will be more difficult than the simple re-write. The greater the proportion of information that needs to be added, the more difficult the translation is likely to be. If the amount of information that needs to be added is only small, then the process of translation might be considered one of *refinement*. However, in the extreme case M_1 and M_2 would have no information in common, and M_1 would be of no value at all in constructing M_2. The translation of M_1 into M_2 would clearly be impossible in such a case.

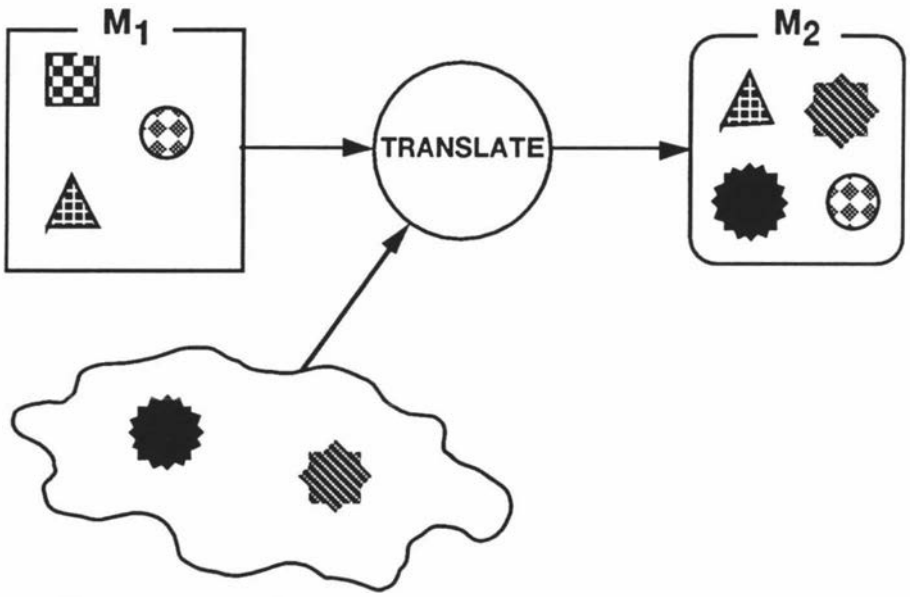


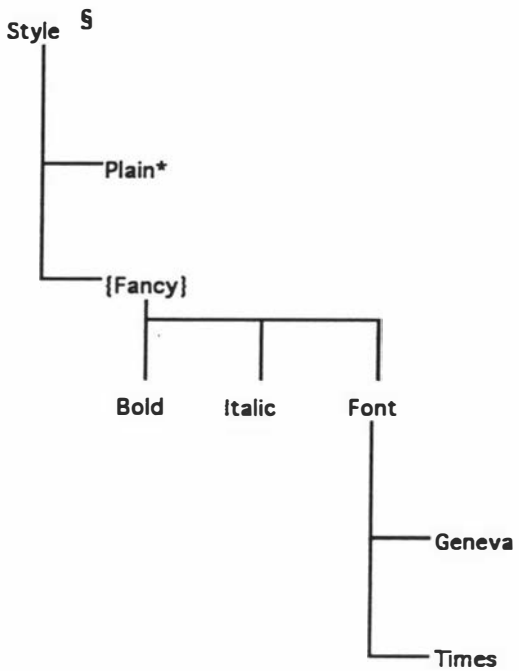
Figure 4.2: A more difficult translation than that depicted in Figure 4.1.

The information content of a model could thus form the basis for a taxonomy of user interface models that could give some indication as to the degree of difficulty of translating between them. Hartson and Hix referred to the *completeness of representation* of a model [Hartson89] which is essentially the same concept.

4.3. TRANSLATION OF LEAN CUISINE

Lean Cuisine differs from the other behavioural dialogue models in that it is restricted to hierarchical menu systems. Within this limited domain however it is a fairly complete formalism, addressing all system layers except the semantic linkage. A menu prototyping system based on Lean Cuisine was implemented early on in this research [Anderson90]. In this system a Lean Cuisine specification (in textual form) was translated into a presentation model (Figure

4.3) and a dialogue model based on ERS (Figure 4.4). At run time the hierarchical menus are constructed in accordance with the presentation model. In response to the user making selections the ERS is interpreted, the resulting system state being displayed (Figure 4.5).



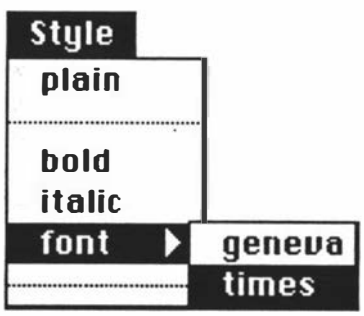
(a)

```
(Style$,ME,
(Plain*,
({Fancy},MC,
   (Bold,
   Italic,
   (Font,ME,

      (Geneva,
      Times)
   ))
)))
```

(b)

```
Style
Plain
(-
Bold
Italic
Font
<
Font
Geneva
Times
>
(-
```

(c)



(d)

Figure 4.3: (a) A Lean Cuisine specification, and (b) its translation from a textual description to: (c) equivalent presentation model and (d) the resultant menu.

```
###  ———————— LC SPEC ————————
###  (Style$,ME,(Plain*,({Fancy},MC,(Bold,Italic,(Font,ME,(Geneva,Times))))))
###  ------------------------------------
%%
Style
Plain
(-
Bold
Italic
Font
<
Font
Geneva
Times
>
(-
%%
###  Regular Rules:
STARTUP       : [Plain]  -> Plain
bold    : Bold  -> [Bold]
bold    : [Bold]  -> [Plain] Bold
font    : Font  -> [Font]
font    : [Font]  -> [Plain] Font
geneva          : Geneva  -> [Geneva]
geneva          : [Geneva]  -> [Times] Geneva
italic    : Italic  -> [Italic]
italic    : [Italic]  -> [Plain] Italic
plain   : Plain  -> [Plain]
plain   : [Plain]  -> [Font] [Italic] [Bold] Plain
times   : Times  -> [Times]
times   : [Times]  -> [Geneva] Times
###  E-rules:
        : [Font] [Italic] [Bold] [Plain]  -> Plain
```

Figure 4.4: Interface definition file corresponding to the menu of Figure 4.3 showing presentation and ERS models.

In order to accommodate the Lean Cuisine concepts, some modifications are made to Hill's ERS notation. As described in Chapter 3, an ERS specification consists of regular and ε-rules. Regular rules are of the form *e:F1->F2*, where *e* is an event and *F1* and *F2* are sets of Boolean flags. ε-rules are similar, except they do not have an associated event. A variant of the ERS notation has been devised, called a Modified Event Response System (MERS). Like ERSs, MERSs also consist of regular and ε-rules, however they differ in their method of rule interpretation and effect. The lists of flags within MERS rules not only identify the flags but also their state (ie, set or unset). A regular rule is said to match if its event matches the current event and all the F1 flags match their state specified in the rule; the same reasoning is applied to ε-

rules. When either type of rule is fired, the F2 flags are set to the states specified in the rule, but the F1 flags are not cleared as is the case in an ERS.

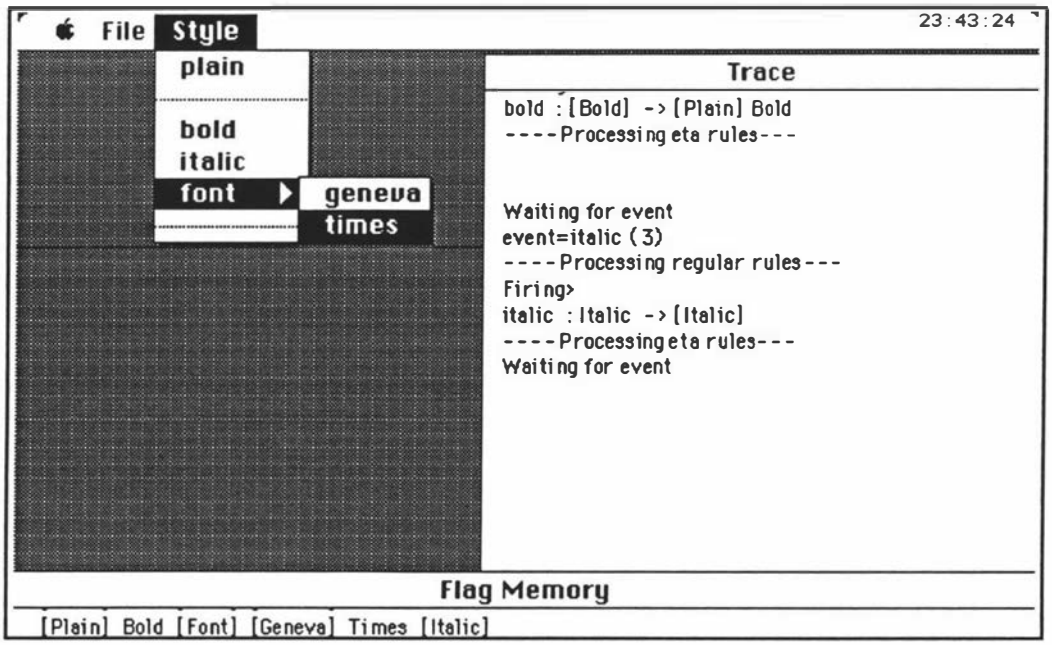


Figure 4.5: Screen dump of the Lean Cuisine interpreter in use.

The translation is based on the following method. An event corresponding to the user's selection is associated with each real meneme. As a consequence of the bistable nature of menemes, two rules are associated with each meneme, corresponding to the setting and clearing of the meneme respectively.

In setting a real meneme, side effects are possible and have to be considered. If the meneme is a member of a mutually exclusive set, then all the other members of the set need to be cleared. However, side effects can carry very much further than just the set within which the meneme is located. Setting a meneme within a menu with a virtual parent has the effect of setting the parent. Hence, effects are propagated up the meneme tree through virtual menemes.

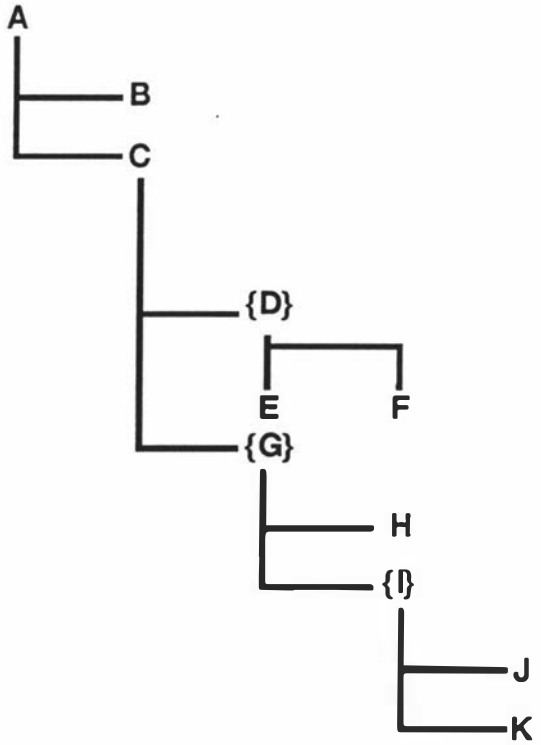


Figure 4.6: Lean Cuisine Tree.

Consider the Lean Cuisine tree represented in Figure 4.6. The effect of setting meneme **K** would be to :

- clear **J** (member of a mutually exclusive set)
- set **I** (virtual parent)
- clear **H** (member of a mutually exclusive set contain **I**)
- set **G** (virtual parent)
- clear **D** (member of a mutually exclusive set containing **G**)

Because it is a real meneme, **C** would be unaffected. Hence the effects would not propagate further up the tree and **B** would be unaffected.

Taking the above example, the following regular rule describes the outcome of selecting **K**.

$$k: \overline{K} \rightarrow K\ \overline{J}\ \overline{H}\ \overline{D}$$

The virtual menemes **I** and **G** are not included in this rule since they are on the direct line of ascent from **K**; their substitution as described below would simply result in duplication. The effect of clearing a virtual meneme (eg **D** in

Figure 4.6) is to generate further changes which propagate down through the associated sub-trees until stopped by real menemes. The virtual sub-tree D can be included by replacing D with real flags using the following substitution rule:

$$\overline{D} := \overline{E} \cdot \overline{F}$$

Carrying out this substitution the following regular rule is produced.

$$k: \overline{K} \rightarrow K\ \overline{J}\ \overline{H}\ \overline{E}\ \overline{F}$$

From the above analysis, the following general principle for generating the rule corresponding to the direct excitation of a real meneme was derived:

Direct excitation of a real meneme when it is in the cleared state gives rise to one regular rule of the form:

$$x: \overline{X} \rightarrow X\ \overline{Y}$$

where:

- **X** is the meneme directly excited by event **x.**
- **Y** is the set of menemes that are members of mutually exclusive sets occurring within the enclosing real set and excluding **X.**

Any virtual menemes occurring in **Y** will be substituted as described above, this process being repeated until no further virtual menemes remain.

Direct excitation of a meneme in the set state, the act of clearing it, produces no side effects (except for those associated with default menemes in required choice groups described below), hence a rule of the following form is sufficient:

$$x: X \rightarrow \overline{X}$$

Within a Lean Cuisine specification there are facilities to allow a meneme to be defined as a default choice, and the set of which it is a member to have a qualifier forcing a required choice. Menemes that are a default need to be set

at interface initialisation (in the implementation, meneme flags are initially cleared). To enable this to be done a dummy *STARTUP* event can be used that is only matched at initialisation, one regular rule being generated for each such meneme.

$$\text{STARTUP} : \overline{X} \rightarrow X$$

Finally, examining the behaviour of default menemes in required choice groups further, some means of setting the default menemes in response to all the flags within the group being cleared is necessary. In order to carry out this action, an ε-rule of the following form is needed:

$$: \overline{X}\ \overline{Y} \rightarrow X$$

where:

X is the default meneme

Y is the set of all menemes within the same group as **X** but excluding **X**.

Again, any virtual menemes appearing in the resultant rule would be replaced by substitution.

4.4. CONCLUSION

What observations can be made from the above example? It does demonstrate that the translation of a behavioural model to its implementation equivalent is possible, although given the limited scope of Lean Cuisine (hierarchical menu systems) this observation cannot be generalised to all dialogue models used in the behavioural domain.

In terms of the simplistic model of the translation process presented at the start of this chapter it is apparent that this translation is straight forward because within the limited domain of hierarchical menu systems Lean Cuisine defines both the appearance of the menus and the grouping behaviour of menemes. The user action and feedback perspectives of the interaction are assumed to be pre-defined by the environment within which a menu is implemented. By combining this environmentally defined information (by utilising a toolkit) with the information present in the Lean Cuisine specification, the menu prototyping system described above is able to produce a complete implementation model (Figure 4.7). The only difficulty in the translation is the

conversion of the group behaviour component of the Lean Cuisine model into an equivalent capable of execution.

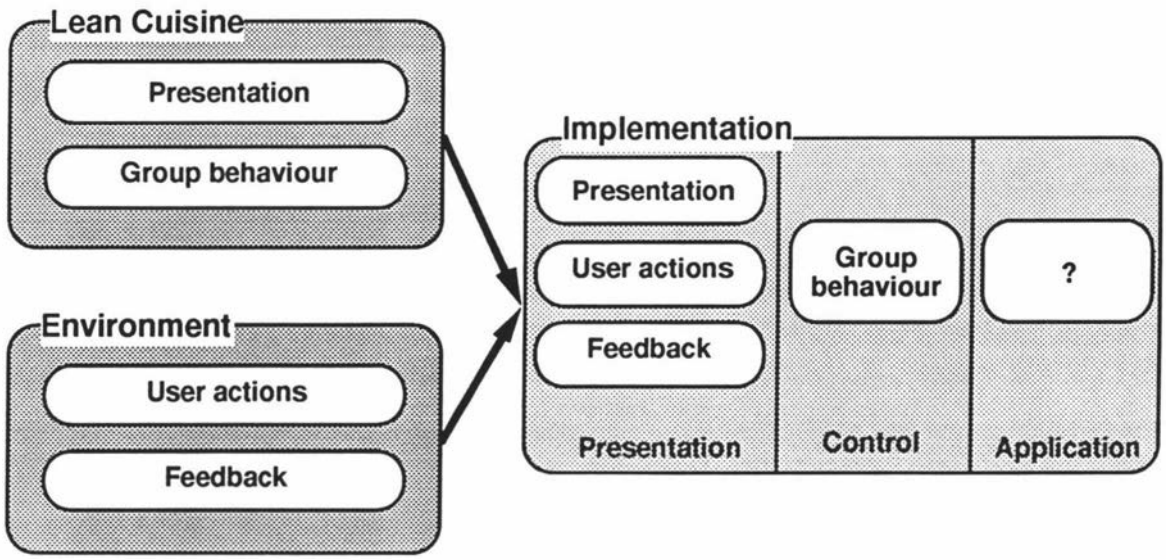


Figure 4.7: A compositional view of the translation of a Lean Cuisine model into its implementation equivalent. The implementation model is presented in terms of a Seeheim architecture.

It would appear from this analysis of the translation process that what is important in assessing the suitability of a behavioural dialogue model for translation into an implementation is the information content of the behavioural model. It follows that a taxonomy of behavioural dialogue models based on their information content is likely to be useful for such an assessment. Figure 4.8 attempts to classify behaviour models in terms of a layered view of the human computer interaction. The basis of this taxonomy is the division of the interaction domain into a number of distinct layers. These are:

1. **Cognition**: the model attempts to reason about the users mental processes.
2. **Actions**: the model describes the users actions.
3. **Presentation**: the model describes the layout and appearance of the interface.
4. **Feedback**: the model describes the systems feedback in response to user actions. The level from which this feedback originates is not considered.

5. **Lexical layer**: the basic actions or tokens that the interface is able to recognise. This is closely related to, but not quite the same as Actions above. In UAN the action ~[close_button]Mv may translate into the lexical input 'close'.
6. **Syntactic layer**: the model defines the *structure* of supported dialogues.
7. **Semantic layer**: the model defines the affect on the application of the user's actions.

Layers 2-5 represent the presentation layer within a Seeheim architecture. Layers 1-4 can be thought of as presenting the user's view of the interface, layers 5-7 as presenting a linguistic view of the system. The full specification of an interface needed for implementation needs layers 2-7 defined.

It should be pointed out that this classification of behavioural models is very simplistic since it does not show whether the information pertaining to a given layer is explicitly or implicitly addressed within the model, or the completeness of this information.

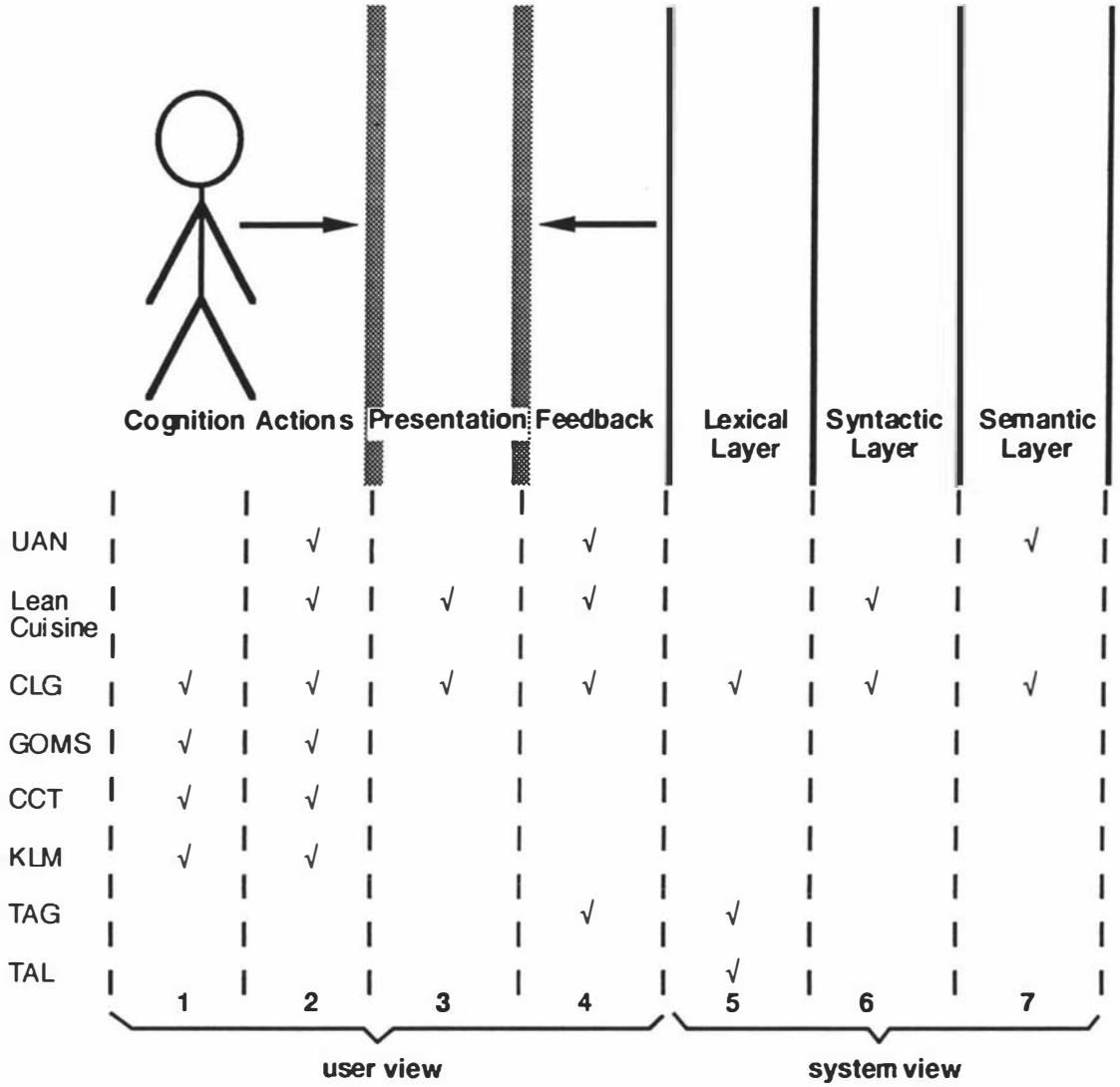


Figure 4.8: A taxonomy of behavioural models

It is apparent from Chapter 2 that behavioural dialogue models place particular emphasis on aspects such as user actions and feedback. Within a Seeheim architecture these are handled by the presentation layer of the architecture. Dialogue models in the constructional domain have been primarily concerned with the control layer within such an architecture, the presentation layer being constructed from standard toolkit routines. In terms of our information content view of models presented above, the dialogue models used in constructional modelling are incomplete. Within the limited scope of the Lean Cuisine example above, this was not a problem. However when dealing with DMUIs in general, low level interaction issues as addressed in the presentation layer would need to be explicitly addressed in the constructional model to provide a

target for the translation of user action and feedback information in the behavioural model.

Unlike Lean Cuisine, UAN and CLG address interfaces in general. CLG however, suffers from a number of shortcomings:

- It was originally developed for describing command language user interfaces, and as such does not address aspects such as interleaving and concurrency, which are very important for direct manipulation user interfaces.

- The physical component of the CLG model is undefined [Johnson92;page 118]. This level of the interaction is very important for direct manipulation user interfaces.

- CLG specifications tend to be extremely verbose, even simple interface specifications running to many pages in length [Hartson89;page 33].

In contrast, UAN is designed specifically for describing direct manipulation user interfaces and places great emphasis on user actions, feedback and temporal ordering. For these reasons, UAN is arguably the best candidate behavioural user interface model for translation into a constructional equivalent.

Chapter 5
Dialogue Activation Language

5.1. INTRODUCTION

In this chapter a new concept, dialogue activation, is introduced. DAL, a new dialogue description language based on the notion of dialogue activation is presented. The chapter concludes with a description of an architectural model necessary to support the execution of DAL.

5.2. DIALOGUE ACTIVATION

Recent work on the classification of interaction tasks within direct manipulation interfaces has suggested that all such tasks reduce to a sequence of selections [Phillips91]. What is unclear from this work is what object and/or attribute selections need to be specified in order to define an interface.

One observation that can be made about DMUI is that only one window is *active* at one time, but the question then arises as to what is meant by active. It is generally considered that the active window is the destination of any keyboard or mouse events generated by the user. However, closer examination suggests that the meaning of active is somewhat deeper. In direct manipulation dialogues the user's actions cause representations of the objects of interest to be manipulated on the screen. However these objects are accessible to the user only when the window in which they are contained is active, since only active windows receive user events. Hence to scroll a window, first it is activated by selecting the window, and secondly the scroll operation is carried out by interacting with the appropriate scroll bar.

Consider a simple dialogue encountered within the Macintosh environment, that of a window and its associated close box. How can such a system be described? The close box is 'activated' by depressing the mouse key whilst the cursor is within its context. In response to this action the close box highlights.

Dragging the cursor out of the close box context without releasing the mouse key results in the close box dehighlighting. However, this does not mean that the interaction has terminated since dragging the cursor back into the close box context results in the close box rehighlighting. Releasing the mouse key within the context of the close box results in the interaction terminating and the associated window closing. Releasing the mouse key in any other context results in the interaction with the close box terminating, but with no effect on the window. This behaviour is depicted using User Action Notation (UAN) [Siochi89] in Figure 5.1.

ACTION	FEEDBACK	SYSTEM STATE	COMMENT
~[close_box]Mv	closebox highlights		interaction with the closebox started
[close_box]~	closebox dehighlights		
~[(x,y)]*			dragging cursor to many arbitrary locations
~[close_box]	closebox highlights		
M^	closebox dehighlights associated window hidden	window is closed	termination of interaction with the closebox.

Figure 5.1: Macintosh closebox behaviour.

A number of observations can be made from this example:

There is a sequence associated with the interaction

~[closebox]Mv	~[(x,y)]*	M^
start	middle	end

i) The sequence is started by an event, that of the user depressing the mouse key within the context of the closebox. This action causes the closebox dialogue to activate. Only by carrying out this action is the user able to gain access to this dialogue.

ii) The sequence is terminated by an event, that of the user releasing the mouse key, no matter what the context. If this action is carried out within the context of the closebox the associated window will close. However, if

this action is carried out outside the context of the closebox, although this will have no effect on the window it will terminate the interaction with the closebox dialogue. After terminating the interaction, the user will only be able to resume the interaction by once again carrying out the start action identified above.

iii) The closebox dialogue is available to be activated by the user only if the associated window is currently active. This point may seem obvious, but this availability hierarchy extends much further than to two such closely related dialogue items.

To examine this last statement, consider a pull-down menu of the style normally encountered on a Macintosh. Figure 5.2 describes how selections are made using this system.

It can be seen in this example that the menu title dialogue can only be activated if the menu bar dialogue is already active (depressing the mouse key within the context of a menu title will cause both the menu bar dialogue to activate as well as the menu title since the context of the menu title is also within the context of the menu bar). Only by activating the menu title dialogue can the menu window dialogue be made available, and only then can the menu item dialogues be accessed.

ACTION	FEEDBACK	SYSTEM STATE	COMMENT
~[menubar]Mv			interaction with the menubar started.
~[menu_title]	menu title highlights. menu window displayed.		interaction with the menu title and menu window started.
~[menu_item]	menu_item highlights		interaction with a menu item started.
[menu_item]~	menu item dehighlights		interaction with menu item terminated.
~[menu_item]M^	Menu item flashes. Menu item dehighlights. Menu window hidden. Menu title dehighlights.	Item selected.	iteration with menu item, menu window, menu title and menu bar terminated.

Figure 5.2: Selection from a pull-down menu.

From the above observations the following generalisation can be made:

Dialogue cells are arranged hierarchically, a sub-dialogue cell being active only if its parent dialogue cell is also active.

What causes a dialogue cell to become active, or indeed an active dialogue cell to become inactive? As discussed above some form of *activation event* will cause the dialogue cell to become active, and conversely, a *deactivation event* will cause an active dialogue cell to become inactive. In practice a dialogue cell may have several activation and deactivation events associated with it. Some dialogue cells may have none, and simply become active or inactive in synchronisation with their parent. Generalising:

Associated with a dialogue cell are two sets of events, a set of activation events or ***activators****, the occurrence of any of which will cause the inactive dialogue cell to activate, and a set of deactivation events or* ***deactivators****, the occurrence of any of which will cause the dialogue cell, if active, to become inactive.*

This notion of dialogue cells with associated activators is referred to as ***Dialogue Activation*** (DA) [Anderson91].

Each dialogue cell has an associated behaviour apart from its role of restricting access to dialogue cells further down the hierarchy. For this reason henceforth they will be referred to as micro-dialogues (μdialogues).

In describing the interaction layer of a direct manipulation interface, five distinct perspectives have to be addressed:

i) User Actions and their context.
ii) User action sequences.
iii) Concurrency.
iv) The relationship to the interface display.
v) The relationship to the semantics of the application.

If an interface can be described in terms of the first three, the resulting model can then be related to the display and application. This approach essentially abstracts the characteristics of an interface that define its "look and feel". User actions and their context can be described using UAN. Action sequences can be defined using a hierarchical framework based on the notion of Dialogue Activation identified above. It will subsequently be shown that by incorporating the ideas of Lean Cuisine [Apperley89] in this framework, concurrency issues can also be addressed. A language based on these ideas has been developed, and is subsequently referred to as ***Dialogue Activation Language*** (DAL) [Anderson92] (Figure 5.3).

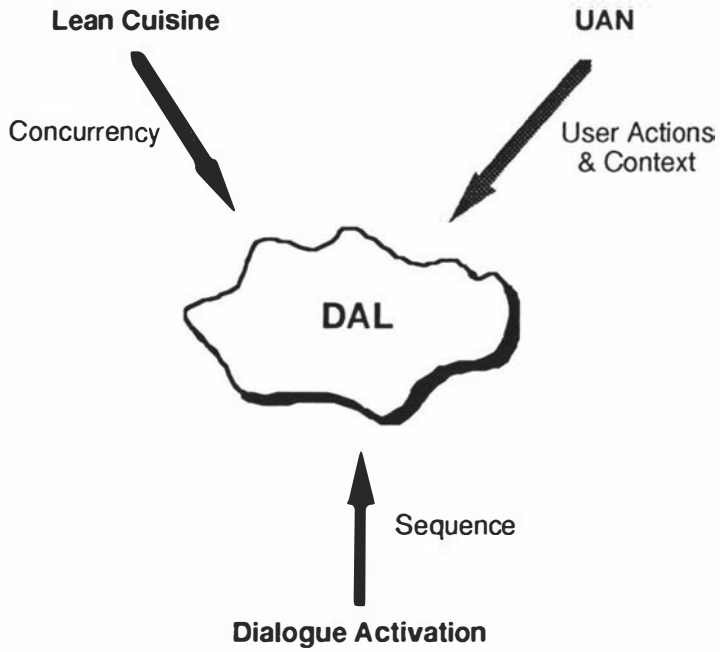


Figure 5.3: The components of Dialogue Activation Language.

5.3. SPECIFICATION OF EVENTS

Within the DAL model, user's actions and the system responses are represented by events. Events fall into one of two distinct categories:

Action Events: Those generated by the user, eg., moving the cursor into the context of a window or depressing the mouse key.

System Events: Those generated internally, for example a close event produced by a close box dialogue.

System events can be referred to by using a suitably expressive identifier. Action events are more difficult to describe. A variation of User Action Notation (UAN) [Siochi89] has been devised for this purpose. In this notation events such as depressing the mouse key are defined using a simple mnemonic. It is often the case that an action event must be considered within a specified context. Within UAN this is done by prefixing the event with a named context within square brackets. For example:

[MENU_BAR]Mv

is interpreted as the action of depressing the mouse key within the context of the MENU_BAR object.

In DAL two restrictions are introduced that significantly simplify the specification of user action events.

i) A µdialogue can have at most one (although sometimes zero) display objects (widgets) associated with it.
ii) A user action event can only specify a context that is local to the µdialogue within which it is specified.

As a consequence of these restrictions it is not necessary to explicitly name the context of an event. A context, if specified, will always refer to the context of 'this' object, 'this' being its associated display object, ie., a self referential context. Hence within a µdialogue associated with the MENU_BAR we would specify the above event simply as `[ ]Mv`. Figure 5.4 summarises the specification of action events.

Event type	Example	
	description	notation
Simple event	depress the mouse key	Mv
Simple event in a specified context	release the mouse key in the context of 'this' object.	[ ] M^
A change of context	move into the context of 'this' object	~[ ]
	move out of context of 'this' object	[ ]~

Figure 5.4: Notation for action events

It can be shown that restricting the context of a user action to the local context does not actually restrict the power of the notation. The ancestors of an active µdialogue must by definition be active. Hence a system event can always be generated in response to a user action occurring within the context of any of the µdialogue ancestors. This system event can subsequently be received by the µdialogue in question.

5.4. SPECIFICATION OF ACTIVATION & DEACTIVATION

A µdialogue is specified by a unique name followed by two bracketed lists of those events that will cause it to activate (activators), and those events that will cause it to deactivate (deactivators). Hence:

```
close_box( []Mv )( M^ )
```

defines a µdialogue called close_box, that is activated by depressing the mouse key within the context of it's associated close_box object, and is deactivated by releasing the mouse key irrespective of the context.

5.5. ACTION SEQUENCES

Using the above concept of dialogue activation and arranging µdialogues hierarchically, a valid user action sequence can be specified. Consider a dialogue composed of 3 µdialogues, A, B and C, where A is the parent of B which in turn is the parent of C. This dialogue hierarchy is presented in Figure 5.5.

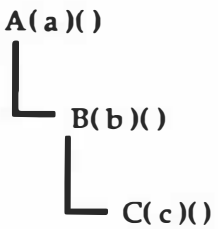


Figure 5.5: Specification of action sequences.

Consider that µdialogue A receives event 'a'. This will cause it to activate. Since B requires event 'b' in order to activate the event does not propagate any further and the system will behave as specified in the body of dialogue A, ie., the system behaves like {A}[1]. That is to say that the user is only able to interact with µdialogue A. If subsequently A receives event 'b' then it will process this event and then pass it on to B, causing B to activate. Since C requires event 'c' to activate the event 'b' will be prevented from propagating further. The system will now behave like {A,B}. If A then receives event 'c', then this will propagate down through A and B and eventually activate C, thus causing the system to behave like {A,B,C}. Put another way, in order for the user to gain access to µdialogue C, they must first carry out actions 'a' followed by 'b' followed by 'c'. Other actions could be carried out between these (provided A or B are not deactivated), but only by carrying out these actions in the order 'a','b','c' will µdialogue C become available to the user.

---

[1]The behaviour of the system is determined by the set of those µdialogues that are active.

5.6. CONCURRENCY

Concurrency has been recognised as a necessary feature of direct manipulation interfaces [Hill87a]. As was discussed in section 2.1, concurrency is a concept relating to both behavioural and constructional domains. DAL crosses the boundary between these two domains and needs to cope with the concept of concurrency relating to both task and system views of the interaction.

A direct manipulation style of interface must handle the asynchronous arrival of events. The user is normally in control and they will, in general, be unrestricted in what actions they can carry out, and hence what events they can cause within the system. In these interfaces a number of separate dialogues are often available to the user at any point in time, and the user is able to choose by pointing or other gestures as to which one they wish to interact with. This 'choice' characteristic means that from a user action perspective these interfaces are non-deterministic. In order to support this switching between multiple dialogues, direct manipulation interfaces must be multi-threaded, that is multiple execution paths through the system are maintained with the ability to switch between them.

However, is it necessary for them to also be concurrent, that is multi-threaded with more than one thread executing in parallel? It has been argued that this is indeed the case on the grounds that user interaction through multiple devices concurrently is a desirable feature leading to more efficient human computer interaction [Buxton86]. In addition to this normal understanding for such a need, it is possible for concurrency to be involved when only a single input device is being used, a single event being broadcast to, and subsequently interacting with, multiple dialogue units. Consider for example a hierarchical pull-down menu. This can be thought of as being composed of a number of discrete yet interconnected dialogue units. Assuming that the action of releasing the mouse key is necessary to make a selection, carrying out this action will cause:

i) the selection to be recorded by the system (which may or may not include the application),

ii) other selections/deselections to be made in response to the choice made , (this may perhaps involve control and applications layers in a Seeheim architecture), and

iii) menu components to be hidden, a presentation layer response.

Obviously such responses do not have to occur in parallel, indeed it may be impossible for (i) and (ii) to execute concurrently. However rapid response is an essential characteristic of these interfaces [Bass88] and concurrency, where possible, can aid its achievement. In addition, considering such semi-autonomous dialogue components in isolation may be a great aid in simplifying the design, but only if the interfaces with the rest of the system are clearly defined.

5.7. VISUALISATION OF CONCURRENCY

DAL extends the sequence hierarchy defined above, by superimposing it on a visual tree notation that explicitly defines concurrent threads through the dialogue. This visual framework is based on Lean Cuisine [Apperley89], a notation derived from, and developed for the purpose of describing hierarchical menu systems. Within Lean Cuisine a hierarchy of menu items (menemes) is defined top down within a tree. Groups of menemes forming the same sub-menu behave in either a mutually exclusive (ie. one from many), or mutually compatible (ie. many from many) manner. Subtrees in turn are represented on the tree as vertical and horizontal branches[2] respectively.

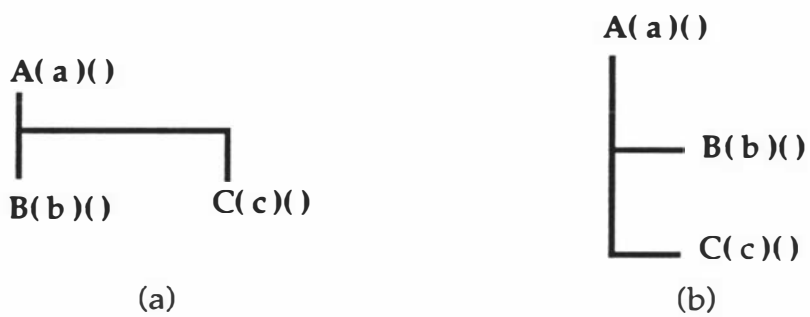
In DAL the nodes on the tree are μdialogues rather than menemes, and the concepts of mutually exclusive and mutually compatible behaviour are reinterpreted. A set of μdialogues with a common parent is referred to as a group. The behaviour interdependencies between members of this group is defined within a grouping attribute of the parent μdialogue. As is the case with Lean Cuisine, this grouping attribute can be either mutually compatible or mutually exclusive.

[2] A textual representation of the tree is used in DAL

µDialogues within a mutually compatible group[3] can all be active together. Execution threads running through each of these µdialogues can be concurrent. Conversely, µdialogues within a mutually exclusive group[4] are explicitly not concurrent. If one µdialogue in the group is active, and another receives an activating event, then the currently active one will deactivate, as will its associated sub-tree.

As discussed above, user action events are propagated down the µdialogue tree. On being presented to a µdialogue which is itself the parent of a mutually compatible group, the event is processed by the µdialogue and then broadcast to *all* of its child µdialogues. This action is similar to the local event broadcast method (LEBM) of Hill's Sassafras system [Hill87b]. However, unlike LEBM this broadcasting of action events is intrinsic to the Dialogue Activation Model. This contrasts with LEBM which is a feature of the Sassafras system used to glue parts of an implementation together.

In order to show how concurrency is modelled in DAL, consider the systems depicted in Figure 5.6.



(a) (b)

Figure 5.6: Specification of concurrency.

Figure 5.6 (a) describes a system in which concurrent threads are possible. Feeding the sequence of events $\langle a,b,c \rangle$ into the system results in threads {A,B} and {A,C} both being active.

---

[3]This is the default grouping in DAL and is implicit in any µdialogue groups without an explicitly declared behaviour. A mutually compatible group can however be declared using a 'grouping MC' construct.

[4]Declared using a 'grouping ME' construct in DAL

Figure 5.6 (b) describes a system in which the two possible threads through the system are explicitly not concurrent. What happens to this system if fed the same sequence of events as used in our previous example? After events $\langle a,b \rangle$ thread {A,B} is active. If event $\langle c \rangle$ is then submitted to the system this causes µdialogue 'C' to activate, but since µdialogues 'B' and 'C' are in a mutually exclusive group this results in 'B' becoming inactive. Hence only the thread {A,C} will now be active.

5.8. DIALOGUE SPECIFICATION

Within the body of a µdialogue new system events can be generated in response to the processing of events received. For example the statement:

```
X <- Y
```

denotes that if the µdialogue in which it is contained is active and it receives event Y, then it will generate system event X in response. Event Y may be either a system or user action event. Where the same event can be produced in response to two or more other events then a shorthand notation can be used. As an example consider a system that will generate an event X in response to either of events A, B or C. This could be specified as three separate event statements as in Figure 5.7(a) or by using the more compact notation depicted in Figure 5.7(b) in which each possible originating event is specified separated with a '|'.

```
X <- A
X <- B
X <- C
```

(a)

```
X <- A | B | C
```

(b)

Figure 5.7: Event statements.

There are occasions (see below) when a given system event is to be generated in response to every event that the µdialogue receives, irrespective of what that event is. This can specified by simply naming the event.

The body of a µdialogue is contained within curly brackets '{' '}'. Within this block each statement is terminated with a semi-colon ';'. The example of a

Macintosh style close box can now be described using the DAL as shown in Figure 5.8.

```
Closebox( []Mv )( M^ ){
        hilite <- []Mv | -[] ;
        dehilite <- []~ | M^ ;
        close <- []M^ ;
}
```

Figure 5.8: Closebox dialogue.

It can be seen that in this example the activator ([]Mv) gives rise to a *hilite* event. Similarly the deactivator (M^) produces a *dehilite* event. It is a common requirement in μdialogues to model a behaviour that is specific to the μdialogue either when it is activating or conversely, when it is deactivating. In order to accommodate this requirement DAL allows ON ACTIVATION and ON DEACTIVATION blocks to be defined. The statements contained within these blocks are processed only in the event of a corresponding state change in the μdialogue. Taking the close box example, use of these optional state change specific blocks allows the same μdialogue to be defined, as in Figure 5.9.

```
Closebox( []Mv )( M^ ){
        ON ACTIVATION { hilite; }
        ON DEACTIVATION { dehilite; }
        hilite <- -[] ;
        dehilite <- []-  ;
        close <- []M^ ;
}
```

Figure 5.9: Closebox dialogue - alternative form.

In this particular example there is no great advantage in using one representation over the other. However there are situations when the "on block" style is mandatory. For example consider a menu item in a mutually exclusive group. This system might be modelled as depicted in Figure 5.10. In this example a menu item deactivates as a consequence of either its parent (the menu itself) deactivating, or one of its siblings activating by virtue of the fact that the menu items form a mutually exclusive group. Since deactivation is not as a result of an event within the context of the item being deactivated it would

be awkward and unwieldy to specify an event statement for the generation of the *dehilite* event[5].

```
menu( open_menu )( M^ ){
      GROUPING ME;
      menu_item1( ~[] )( ){
            ON ACTIVATION { hilite; }
            ON DEACTIVATION { dehilite; }
            select <- []M^
      }
      menu_item2( ~[] )( ){
            ON ACTIVATION { hilite; }
            ON DEACTIVATION { dehilite; }
            select <- []M^
      }
}
```

Figure 5.10: Use of ***on activation*** and ***on deactivation*** statements

It is often the case that it is necessary to know not just that the user depressed the mouse key, but the location of the cursor at the time the event was generated. In order to support the passing of information like this around the system an event has four fields associated with it that can be assigned in the case of system events, and in the case of user action events are filled with the position of the curser when the event was generated (in global coordinates). Within an event statement the field specification is optional. In order to generate a new system event with one or more event fields specified, the event statement is modified as shown in Figure 5.11.

```
new_event[field_values] <- current_event;
```

Figure 5.11: Specification of field values in an event statement.

The contents of the four fields in the event that is currently being processed can be referenced using four pre-declared variables; Ex,Ey,Ea,Eb. Consider the example in Figure 5.12. In this example the µdialogue *Do_Popup* generates an event *MoveTo* whenever it is activated. The value of the first two fields in the event record will be the same as the corresponding field positions in the event

[5]The alternative would be to make the grouping mutually compatible and to explicitly generate a system event in each menu item µdialogue that would deactivate every other µdialogue within the group. Although this approach would work the clarity of the description would be severely reduced.

that caused the μdialogue to activate, a MouseDown event (Mv). It follows that these will be the coordinates of the MouseDown event.

```
Do_Popup( Mv )( M^ ){
    ON ACTIVATION {
        MoveTo[Ex,Ey];
        ShowWindow;
    }
    ....
}
```

Figure 5.12: Referencing event fields.

5.9. VARIABLES

The event fields are special examples of variables within the system. They are a clean way of passing information between μdialogues. The value of an event field can only be set when the event is first generated and cannot be *tampered with* thereafter. There are occasions when it is necessary for a μdialogue to maintain information either local to itself or within a restricted group of associated μdialogues. For these situations DAL supports variables.

These are declared in a similar fashion to events and follow the same scoping rules. All variables are integers and assignment expressions are similar to those used in C [Kernigan78]. For example consider the system specified in Figure 5.13.

```
Do_Pen( Mv )( M^ ){
    VAR pen_size;
    GROUPING ME;

     ON ACTIVATION{
          pen_size = 1;
          Move_To[Ex,Ey];
          Show_Pen;
    }
    ON DEACTIVATION{
          Hide_Pen;
    }

    Move_To[Ex,Ey] <- cursor_moved;

    Increase_Pen_Size( UP_ARROWv )( UP_ARROWv  ){
          pen_size = pen_size + 1;
          Set_Pen_Size[pen_size];
    }
    Decrease_Pen_Size( DOWN_ARROWv )( DOWN_ARROWv  ){
          pen_size = pen_size - 1;
          Set_Pen_Size[pen_size];
    }
}
```

Figure 5.13: Use of variables.

In this example the variable *pen_size* is in scope throughout the µdialogues *Do_Pen, Increase_Pen_Size* and *Decrease_Pen_Size*. Apart from the setting the initial *pen_size* value when *Do_Pen* activates, *pen_size* is only changed by the latter two µdialogues. These form a mutually exclusive grouping and hence there is no danger of multiple µdialogues attempting to access the same variable at the same time. If however the same variable was updated by two concurrent threads then this does present a problem. This possibility can be tested for within a specification, and is dealt with in Chapter 6.

5.10. GUARD CONDITIONS

In the example of a dialogue for varying the pen size in a paint program interface presented in Figure 5.13, the µdialogues that increase or decrease the pen size execute every time that they receive their respective activating events. This would of course mean that the pen size could be reduced to zero (or less, whatever that may mean), or increase to some unacceptably large number, perhaps too large for the associated pen widget to cope with. It would obviously be useful to be able to set limits on such parameters. This could be done by requiring the interface to 'ask' the application if the whether a µdialogue should execute or not. Although this semantic feedback would sometimes be necessary, more often than not such decisions can be delegated to the lower levels of the interface.

DAL supports this by introducing the concept of a ***guard condition*** , this notion of a guard being derived from Dijkstra's guarded command language [Dijkstra76]. This is a Boolean expression associated with a µdialogue. The µdialogue can be activated if, and only if the guard condition evaluates to true. A guard condition is specified by following the name of a µdialogue with a conditional expression in square brackets. Taking the example from Figure 5.13, limits can be set on the pen size by using a guard condition as shown in Figure 5.14.

```
Do_Pen( Mv )( M^ ){
  VAR pen_size;
  GROUPING ME;

   ON ACTIVATION{
        pen_size = 1;
        Move_To[Ex,Ey];
        Show_Pen;
  }
  ON DEACTIVATION{
        Hide_Pen;
  }

  Move_To[Ex,Ey] <- cursor_moved;

  Increase_Pen_Size[pen_size<MAX]( UP_ARROWv )( UP_ARROWv  ){
        pen_size = pen_size + 1;
        Set_Pen_Size[pen_size];
  }
  Decrease_Pen_Size[pen_size>MIN]( DOWN_ARROWv )( DOWN_ARROWv ){
        pen_size = pen_size - 1;
        Set_Pen_Size[pen_size];
  }
}
```

Figure 5.14: Guard Conditions.

In this example MIN and MAX could either be pre-defined constants, or variables (that could be set by the application).

5.11. SCOPE OF EVENTS

Unlike action events that are propagated top-down through the µdialogue framework, system events when generated do not propagate down the tree but are broadcast to other µdialogues, widgets and application interface modules. This event broadcasting is restricted in scope and is referred to as ***Localised System Event Broadcasting*** (LSEB)

Consider the menu item dialogues depicted in Figure 5.10. Each menu item µdialogue can give rise to *hilite* and *dehilite* events. These system events drive the feedback of the associated widget. Without any means of restricting the interaction scope of an event there would be no way of ensuring that the *hilite* event generated by *menu_item1* caused only its associated widget to highlight.

In order to specify scope, events are declared within the body of a µdialogue and follow the same scoping rules as variables in most block structured languages. That is, an event is in scope in the µdialogue in which it is declared

and throughout any of its descendants. If an event of the same name is declared at a lower level then this more local declaration *masks* out the earlier one, the declaration with the most local scope being the one taken.

Figure 5.15 shows how the scoping of events is extended to the corresponding presentation components in a menu specification.

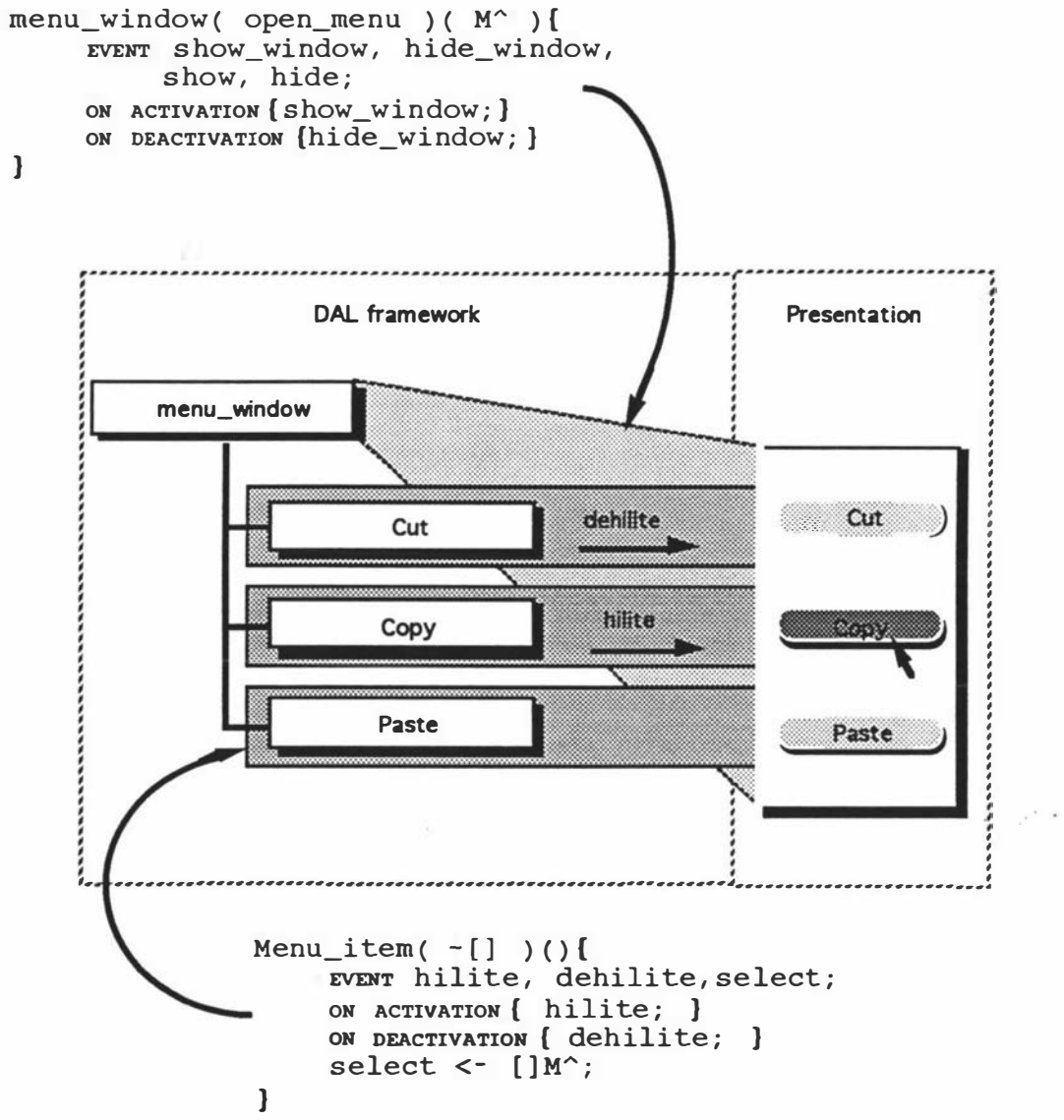


Figure 5.15: Extension of event scoping to the presentation layer.

In this example a window widget will respond to events *show_window* and *hide_window* and a button (menu item) will respond to *show, hide, hilite* and

dehilite events. The *select* event in this example would be sent to an associated application interface module.

In Figure 5.16 the earlier example of a closebox within a simple window is shown. In this case the window widget can respond to *show_window* and *hide_window* events, and the closebox widget is able to respond to *hilite* and *dehilite* events. The close event generated in the *Close_Box* µdialogue is declared within the *Simple_Window* µdialogue and hence is the same event as the deactivator for *Simple_Window*.

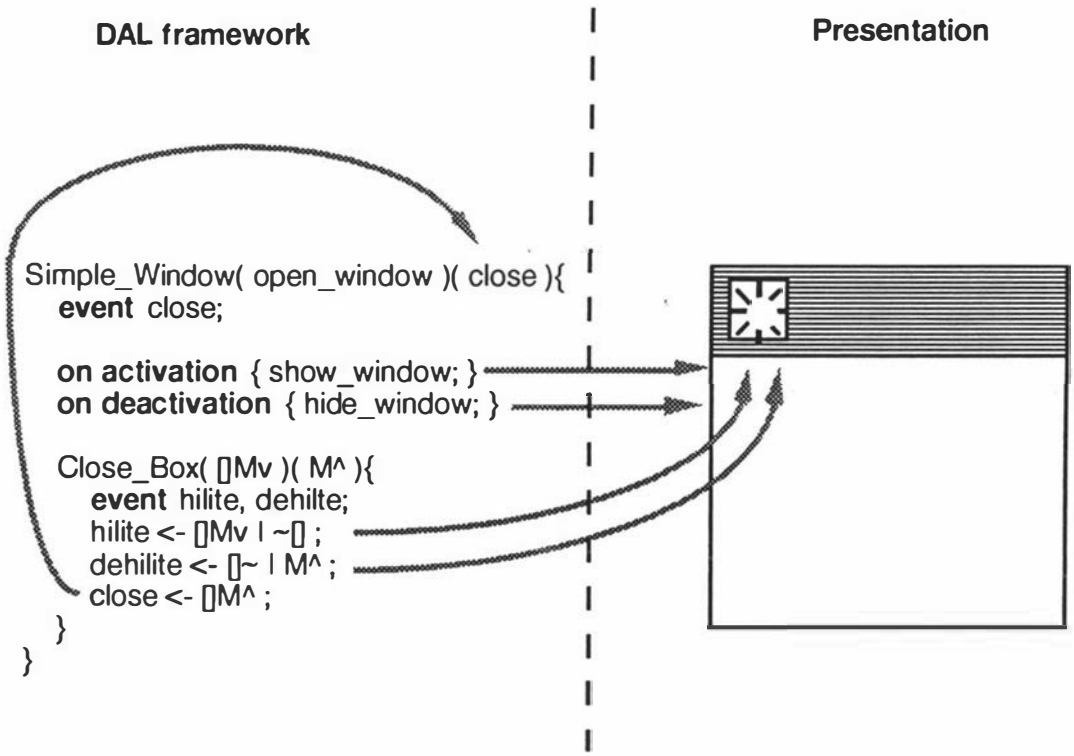


Figure 5.16: Scoping of events for closebox in window example.

5.12. PRESENTATION AND APPLICATION ATTACHMENT

The notation described so far permits the modelling of dialogues from a user action perspective. It has been assumed up to this point that the events generated by the µdialogues are consumed by the appropriate display entities or application interface module. In order to make this attachment DAL supports **DISPLAY** and **APPLICATION** statements.

5.12.1. DISPLAY STATEMENTS

In the examples considered so far, it has been assumed that there are presentation objects (widgets) associated with μdialogues, and which respond to events that can be sent to them from the μdialogue. Under the topic of event scoping it has been shown how events can be sent to a restricted set of such widgets. In the initial design stages the choice may be taken to ignore such presentation issues, simply assuming that there is some display entity associated with a given μdialogue, that will respond to such and such an event. However the design will be progressively refined, until eventually a sufficiently complete specification of the interface is arrived at such that it can be prototyped. In order to do this it is necessary to augment the design with the necessary information needed to link each μdialogue through to an instance of the appropriate widget type.

In building this facility into the language, the most important consideration was that it should be possible to add new widget types, or to change the characteristics of existing widgets without affecting the syntax of the language. To achieve this, the language provides an **ATTRIBUTE** statement. Consider the specification of a window widget. This would have attributes of size, location and visibility that could be set, and possibly subsequently changed at run time. We can define the attributes of this widget type as shown in Figure 5.17.

```
ATTRIBUTE
 window
        size        DD,
        location    DD,
        visible,
 ;
```

Figure 5.17: Defining widget attributes.

In this example a widget type called *window*, is defined that has attributes of *size*, *location* and *visibility*. In order to define the size of the window we need to specify two parameters, both of which are digits, and this is indicated within the attribute statement by the mnemonic 'DD'. In the case of visibility, the default is for the widget to be initially hidden. However, if an instance of a window widget is declared with the attribute *visible* specified, then the corresponding widget will be made visible as soon as it is created. Unlike the size attribute, the visible attribute does not need any parameters specified.

Some attributes require *string* values associated with them and in this case the mnemonic 'S' is used to signify this within the attribute statement.

To create an instance of a widget, its associated μdialogue is annotated with a **DISPLAY** statement. In this statement, any of the parameters identified in the **ATTRIBUTE** statement can be preset. In the presentation model used a window is taken as being a special object type. If a new window is declared, then any positional parameters are in terms of global coordinates. If a child μdialogue has an associated widget that is not a window, then it is assumed to be displayed within the parent window, and the coordinates are given in terms of the parent window coordinate system. Figure 5.18 shows how a simple window containing a button can be defined.

```
ATTRIBUTE
        window
                size            DD,
                location        DD,
                visible
        ;
        button
                size            DD,
                location        DD,
                visible,
                name            S
        ;
...
DialogueBox( do_dialog )( done ){
        EVENT done, show_window, hide_window;
        DISPLAY TYPE window,
                location 10 10,
                size 200 100;
        ON ACTIVATION{ show_window;  }
        ON DEACTIVATION{ hide_window; }

        DoneButton( []Mv )( M^ ){
                EVENT hilite, dehilite;
                DISPLAY TYPE button,
                        location 160 10,
                        size 30 20,
                        visible,
                        name "Done";
                hilite <- []Mv | -[];
                dehilite <- M^ | []-;
                done <- []M^;
        }
}
```

Figure 5.18: Display statements.

In this example the µdialogue *DialogueBox* has an associated *window* widget. This widget responds to events *show_window* and *hide_window.* Initially this *window* widget is invisible (the visible attribute not being preset). When the *DialogueBox* µdialogue activates, a *show_window* event is sent to its associated widget causing the window to be displayed at the preset (global) coordinates. The child µdialogue *DoneButton* has an associated button widget. This is initially visible, hence when the parent window is display, this button is displayed within it at the specified (window) coordinates.

There is no provision within the language to specify the events to which a widget will respond. This is hard coded into the widget definition, and recorded in an include file 'WidgetEvents.dal'. This consists of a set of define statements relating the name of events to their corresponding numeric event codes (Figure 5.19).

```
DEFINE
        PIPS_Stop               31,
        WE_ShowWindow           100,
        WE_HideWindow           101,
        WE_Show                 102,
        WE_Hide                 103,
        WE_MoveTo               104,
        WE_MoveBy               105,
        WE_Hilite               106,
        WE_Dehilite             107,
        WE_SetSize              108,
        WE_SetLoc               109;
```

Figure 5.19: Widget event definitions.

5.12.2. APPLICATION STATEMENTS

An interface is of little use without the associated functionality of the underlying application. In designing an interface it is important to resolve how the interface and application are to be linked. The DAL model assumes that the connection between the two is via a message or token based communication mechanism, with an intermediary, an Application Interface (API) between them. This communication can be two way, so that not only can an application function be invoked through the interface, but the application can *tell* the interface to respond or change in some way.

As already discussed, in order to simplify the specification of action event contexts a µdialogue has at most one presentation object associated with it.

Similarly it is also assumed that there is at most one application function (and sometimes none) that can be invoked through the µdialogue. The application function is assumed to respond to one or more events generated within the interface, and within the scope defined at the interaction level using the DAL scoping rules. Hence, taking the earlier example of a menu (Figure 5.15), we could add in an interface to application functions corresponding to each menu item µdialogue (Figure 5.20).

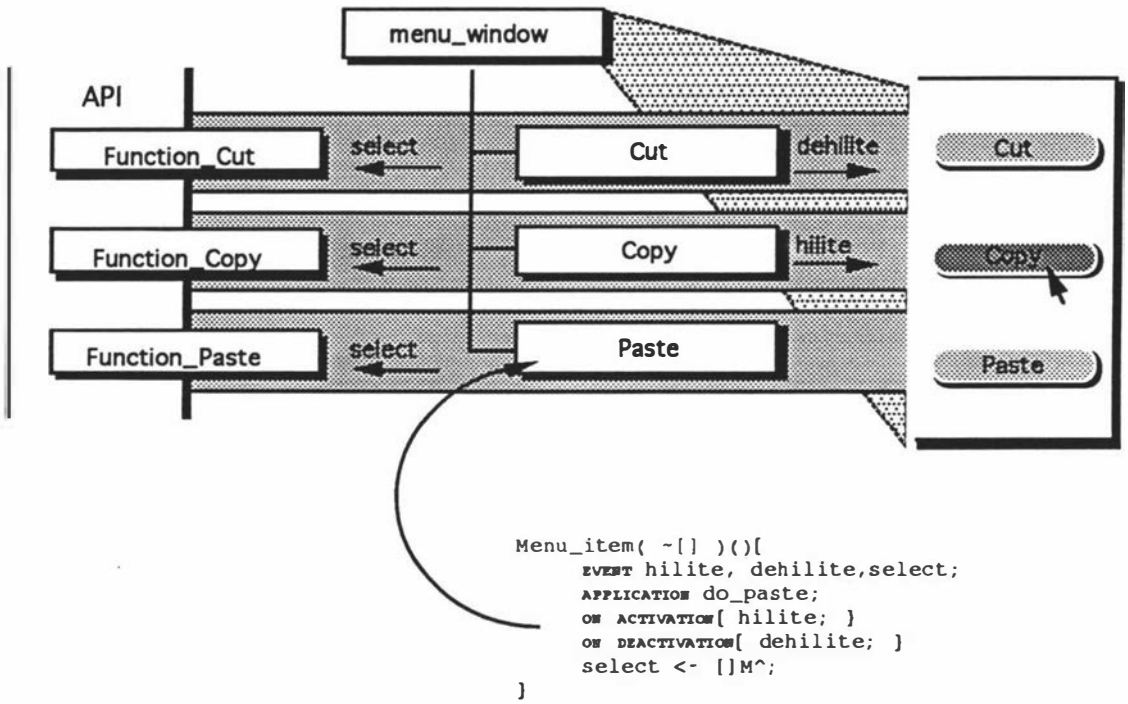


Figure 5.20: The Application Interface.

In order to specify the application linkage an **APPLICATION** statement is used. This states the name of the C++ application interface class, the interface through which the application will receive events from the interface. In exactly the same way as the events that a widget will respond to is hard coded into the widget class definition, the same applies to application interface classes. Again the numerical interpretation of events that the application is able to receive and generate is defined within a header file, that can subsequently be included into a DAL specification. It is up to the designer to specify what events they believe the application should receive, and the semantic interpretation of those events. The form of an application statement is as shown in Figure 5.20, with the name of the application function following the keyword **APPLICATION**.

5.13. OBJECT ORIENTED FEATURES OF DAL

Consistency in the 'look and feel' both within a given interface and across different application interfaces on the same platform has been recognised as a desirable feature[6] [Shneiderman87]. In order to support this requirement, DAL supports the definition of µdialogue templates or classes that define a generic behaviour and appearance. A new class can inherit the characteristics of another, that is stating that the new class of µdialogue behaves like some other, but has some additional behaviour. Hence DAL is an object oriented language supporting single inheritance. In addition it is often the case that an interface will have many widgets that behave the same but have a different appearance, and may, for each single instance have slightly different behaviour. DAL allows instances of a given class to be defined within a design with differences being specified for just that one instance. The notation used for defining and using classes is based on C++ [Stroustrup86].

A class is introduced by using the **CLASS** keyword. This identifies that the µdialogues that follow are, in fact, templates for µdialogues and not actual instances. The structure of a class definition is exactly the same as that of a normal µdialogue. Following the class definitions will be the specification of the actual interface. The start of this section is denoted by the keyword **SYSTEM**. An instance of a µdialogue based on a given class, can be introduced by giving the class name, followed by a colon and the name of the actual instance. For example **menu_item:paste** denotes that paste is a µdialogue based on the definition of class *menu_item*. If an instance of a class has additional activators, deactivators or statements defined, then these are taken as being additional to those defined for the class. Any additional statements are deemed to be executed *after* those defined within the class.

Consider, for example the definition of a menu. Using classes a menu containing several (similar) menu items could be defined as show in Figure 5.21

[6]This is one of the strongest arguments for toolkits.

```
CLASS
  menu_item( ~[] )( []- ){
        EVENT hilite, dehilite, select;
        DISPLAY TYPE button;
        ON ACTIVATION { hilite; }
        ON DEACTIVATION { dehilite; }
  }

SYSTEM  /* The interface definition follows...*/
  desktop( )( ){
        EVENT do_cut, do_paste, do_copy;
              ...

        edit_menu( show_menu )( hide_menu ){
              cut:menu_item( )( ){ do_cut <- []M^; }
              paste:menu_item( )( ){ do_paste <- []M^; }
              copy:menu_item( )( ){ do_copy <- []M^; }
        }
  }
```

Figure 5.21: Use of a μdialogue class definition.

In this example cut, paste and copy are all menu items, and behave exactly as defined in the menu_item class. Additional behaviour has been added, in this case the generation of different events in response to releasing the mouse key within the context of the menu item.

The definition of derived classes is done in a similar fashion. Figure 5.22 shows how the behaviour of menu items within a hierarchical pull-down menu can be defined.

```
CLASS
      /*
       *    Assume a ME group
       */
      menu_item( ~[] )( ){
            DISPLAY TYPE button;
            EVENT hilite, dehilite;
            ON ACTIVATION { hilite; }
            ON DEACTIVATION { dehilite; }
      }
      menu_header:menu_item( )( ){
            EVENT show_sub, hide_sub;
            ON ACTIVATION{ show_sub; }
            ON DEACTIVATION{ hide_sub; }
      }
      selectable:menu_item( ) ( ){
            EVENT select;
            select <- []M^;
      }
```

Figure 5.22: Derived classes.

In this example, a generic button object *menu_item* is defined that is activated by dragging into its context. It is not defined how it can be deactivated. As the comment states it is assumed that a button belonging to this class would be deactivated by virtue of it being in a mutually exclusive group, another sibling activating causing it to deactivate. A *menu_header* is a *menu_item* that when activated causes the associated sub-menu to be shown. The corresponding menu window display object responding to *show_sub* and *hide_sub* events. A selectable object is a menu_item that the user is able to select by releasing the mouse key within its context, this action giving rise to a *select* event.

The use of these classes to defined a hierarchical pull-down menu is shown in Figure 5.23.

```
SYSTEM

menu_bar( []Mv )( M^ ){
      DISPLAY TYPE window;
      GROUPING ME;
            file_menu_header:menu_header()(){
                  DISPLAY name FILE;
                  file_menu()(){
                        DISPLAY TYPE window;
                        GROUPING ME;
                        new:selectable()(){
                              display name NEW;
                        }
                        open:selectable()(){
                              display name OPEN;
                        }
                        file_type_header:menu_header()(){
                              DISPLAY name TYPE;
                              file_type()(){
                                    DISPLAY TYPE window;
                                    GROUPING ME;
                                    type1:selectable()(){
                                          DISPLAY name TYPE1;
                                    }
                                    type2:selectable()(){
                                          DISPLAY name TYPE2;
                                    }
                              }
                        }
                  }
            }
            edit_menu_header:menu_header()(){
                  DISPLAY name EDIT;
                  edit_menu()(){
                        DISPLAY TYPE window;
                        GROUPING ME;
                        cut:selectable()(){
                              DISPLAY name CUT;
                        }
                        paste:selectable()(){
                              DISPLAY name PASTE;
                        }
                  }
            }
      }
}
```

Figure 5.23: Specification of a hierarchical pull-down menu.

The example might appear on the screen as depicted in Figure 5.24.

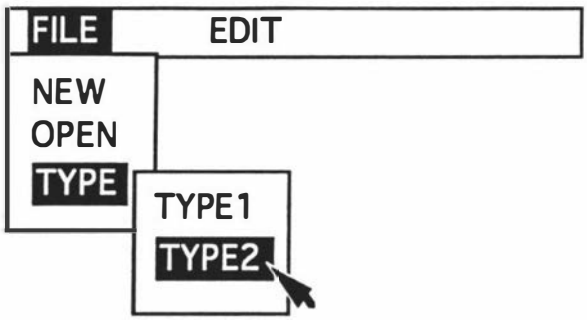


Figure 5.24: A possible appearance of the menu specified in Figure 5.23

This specification of this hierarchical pull-down menu may appear complex, however this textual specification could be produced with a graphical editor in which the user is simply presented with the Lean Cuisine framework. This is much simpler in appearance, allowing the sequence and concurrency of the system to be clearly seen (Figure 5.25).

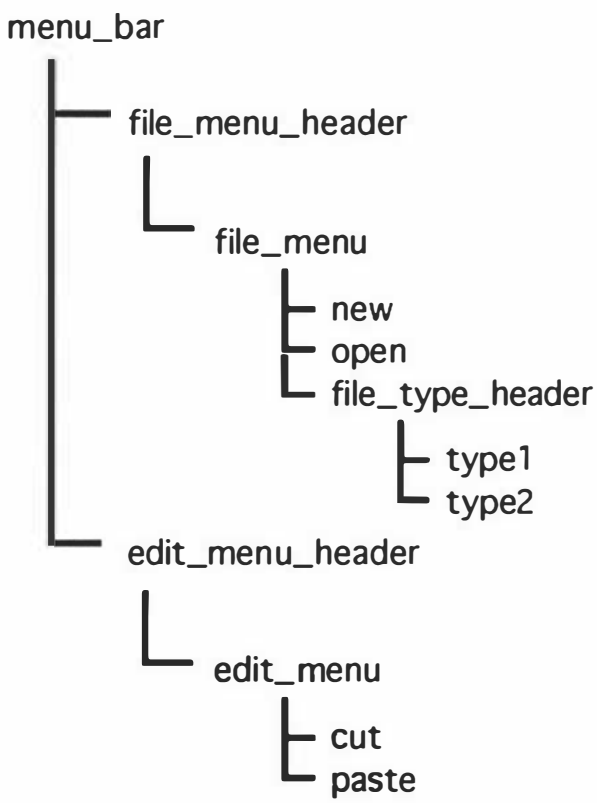


Figure 5.25: The Lean Cuisine framework of pull-down menu.

5.14. DYNAMIC CREATION OF µDIALOGUES

DAL as described allows static interfaces to be defined, and the appropriate widgets and application routines to be attached. However, many interfaces are not static. Consider for example an application like MacDraw®. A user can draw an object on the screen and subsequently manipulate that object. It follows that in creating such an object, the user must also be creating the associated interaction dialogue and application components. As another example, consider the Macintosh Finder®. If the user creates a new folder, then the corresponding widget will appear on the screen, and can be manipulated in the way that the user associates with such objects.

It follows that a language for defining such interfaces needs facilities to support the creation of new dialogue instances with a ***previously defined behaviour***. In DAL, generic dialogues are defined using dialogue class definitions. The question of how to extend the language to cover dynamic interfaces thus comes down to addressing the following issues:

i) If a new µdialogue is created, where does it fit into the existing DAL µdialogue framework?
ii) What will be the activation state for the new µdialogue (ie., will it immediately be active or inactive)?
iii) How can the default position and size of the widget defined in the class definition be overridden?

DAL supports dynamic interfaces by permitting the creation of new µdialogue instances in response to a µdialogue processing a given event. This is defined in a *new* statement as shown in Figure 5.26.

```
NEW A <- event;
```

Figure 5.26: New Statement.

The interpretation of this statement is that a new µdialogue, based on *class* A will be generated in response to *event*. The new µdialogue will become a child of the µdialogue containing the *new* statement that gave rise to it. Initially the new µdialogue will be inactive, although this default behaviour can easily be overridden. The new statement can optionally have up to four parameters associated with it (Figure 5.27). These correspond to the location of the new

widget (parameters 1 and 2) and its size (parameters 3 and 4). If any parameters are not specified, defaults are taken from the class definitions. Specified parameters can be constants, variables or field values form the current event.

```
NEW Z[x,y,a,b] <- C;
NEW Rect [StartX,StartY,Height,Width] <- M^;
NEW Folder [Ex,Ey] <- NewFolderHere;
```

Figure 5.27: New statements with parameters.

Figure 5.28 shows a simple example of the use of the new statement.

```
CLASS
    SimpleButton( []Mv )( ){
          DISPLAY TYPE Button,
                size 10 10,
                location 0 0,
                visible;
          ON ACTIVATION{ hilite; }
          ON DEACTIVATION{ dehilite; }
    };
SYSTEM
    sample( )( ){
          VAR x, y;
          EVENT NewButton;
    ...
          MyWindow( )( ){
                NEW SimpleButton[x,y] <- NewButton;
          }
    }
```

Figure 5.28: Use of the new statement.

The state of a new μdialogue is inactive by default. However, a design can easily include the facility for a new μdialogue to receive an activating event immediately after it is created, hence making it active. Indeed, there is no reason why the event that causes the new statement to be executed cannot be an activator for the μdialogue, in which case this event will be passed down to the new μdialogue immediately following the processing of the body of the current (parent) μdialogue (if it is an action event).

The above features allow one to specify an interface that can extend itself. However this facility would be incomplete without the ability to also delete μdialogues. This is achieved with a ***delete*** statement. This takes the form shown in Figure 5.29.

```
DELETE <- event;
```

Figure 5.29: Delete Statement.

The interpretation of this statement is that the μdialogue in which it is contained will be destroyed if the current event is *event*. As a consequence its associated widget and application interface will also be deleted. All μdialogues must be located within the single DAL tree. As a consequence, deleting a μdialogue also leads to any child μdialogues being deleted.

```
CLASS
      Rectangle( []Mv )( ){
            VAR Right, Bottom;
            EVENT show_handle, hide_handle;
            DISPLAY TYPE RectObject;
            ON NEW {
                  // Ex,Ey,Ea,Eb are parameters passed to new
                  // statement to place handles at each corner.
                  Right = Ex + Width;
                  Bottom = Ey + Height;
                  NEW Handle[Ex,Ey];
                  NEW Handle[Right,Ey];
                  NEW Handle[Right,Bottom];
                  NEW Handle[Ex,Bottom];
            }
            ON ACTIVATION{ show_handle; }
            ON DEACTIVATION{ hide_handle; }
            MoveBy[Ea,Eb] <- cursor_moved;
            Resize[Ex,Ey] <- ResizeRect;
            DELETE <- 'DELETE'v;
      },
      Handle( []Mv )( M^ ){
            DISPLAY TYPE icon,
                  resource BlackSquare,
                  size 5 5;
            VAR xinc, yinc;
            xinc = Ea / 2;  yinc = Eb / 2;
            ResizeRect[xinc,yinc] <- cursor_moved;
      };
```

Figure 5.30: New and delete in class definitions.

On creating a new μdialogue, or deleting an existing one, it is often desirable to be able to carry out some special set of actions. To facilitate this requirement, DAL has two special commands, *on new* and *on delete* that are exactly like their *on activation* and *on deactivation* counterparts. The *on new* statement allows μdialogue subtrees to be instantiated, since a *new* statement can be contained in a *on new* block.

Figures 5.30 and 5.31 show how these language constructs can be used to produce part of an interface for a drawing application similar to MacDraw®. In this example we have defined a Rectangle class. Both *on new* and *on delete* blocks do not have a current event associated with them. Within these blocks the special variables Ex, Ey, Ea and Eb refer to the parameters within the new or delete statements that caused them to be executed. In this example, new handle μdialogues are created, attached to each of the four corners of the rectangle when the rectangle is instantiated. It is unnecessary to specify the converse operation, since deleting a μdialogue deletes its widget, API and any child μdialogues.

```
SYSTEM
      ...
      DrawingBoard( )( ){
            GROUPING ME;
            Tools( )( ){
                  // ... various drawing tools ....
            }
            DrawingWindow( )( ){
                  DISPLAY TYPE window,
                        location 0 0,
                        size 200 200,
                        visible;
                  NEW Rectangle[Ex,Ey,Ea,Eb] <- NewRect;
                  GROUPING ME;
                  DoOutline( Mv )( M^ ){
                        ON ACTIVATION{
                              MoveTo[Ex,Ey];
                              StretchTo[Ex,Ey];
                              ShowOutline;
                              StartX = Ex; StartY = Ey;
                        }
                        ON DEACTIVATION{
                              Length = Ex - StartX;
                              Height = Ey - StartY;
                              HideOutline;
                        }
                        StretchTo[Ex,Ey] <- cursor_moved;
                  }
                        // ... Drawn widgets ...
            }
      }
```

Figure 5.31: MacDraw® style interface using classes as defined in Figure 5.30.

5.15. ARCHITECTURAL CONTEXT

As discussed in Chapter 3, it has been identified that two types of models have to be considered in the implementation of a user interface [Green86]. The first is an *architectural* model that defines the relationships between the system components. The second is a *dialogue* model the defines the structure of the human-computer interaction possible with the user interface.

DAL is primarily concerned with defining the latter. However DAL also implies a particular relationship between the system components. Hence a substantial component of an architectural model is inferred. A DAL model of a user interface is based on the concept of an ***agent***. An agent is an entity capable of maintaining an internal state, and able to receive events from its environment and respond by generating new events that it feeds back into its environment [Bass91]. It is apparent that a µdialogue is an example of an agent. The structure of the DAL tree imparts a hierarchy to the µdialogue agents. Hence a DAL model implies an overall structure of a user interface in the form of a multi-agent architecture. Within this system structure the association of µdialogues with their corresponding widgets and API implies a tripartite division of the interface that is similar in its separation of functionality to that found in the Seeheim architecture [Green85], although the latter is a monolithic architecture.

Traditionally most UIMS have been based on a notation for modelling the dialogue control component, with lexical and semantic levels attached as appropriate. The DAL approach starts at a lower level. It assumes that a very simple object-based toolkit is available that is able to respond to the events generated by the µdialogues of DAL. It assumes a similar interface with respect to the application. High level dialogue control has not been considered within this work, but it is envisaged that this aspect of the interface could also be modelled as a group of event handlers that could be inserted into the design where and when required.

The interaction layer, as defined in DAL, can be viewed as an object that converses with the user, with display on one side and the application (via an Application Interface (API)) on the other (Figure 5.32). This shared layer approach is similar to Took's surface interaction model [Took90] in which the display surface is considered a shared medium between the user and the application. In the DAL model the concept of access to dialogue elements is

viewed as being the most important feature of direct manipulation interfaces. The user by virtue of their actions is able to gain access to μdialogues by carrying out appropriate actions. This is clearly an example of external control. Conversely, the application (by means of events) is able to explicitly activate or deactivate dialogues within the interaction layer, effectively modifying the options available to the user. This is a form of internal control. Hence DAL supports what Koivunen and Mantyla call a mixed control model [Koivunen88].

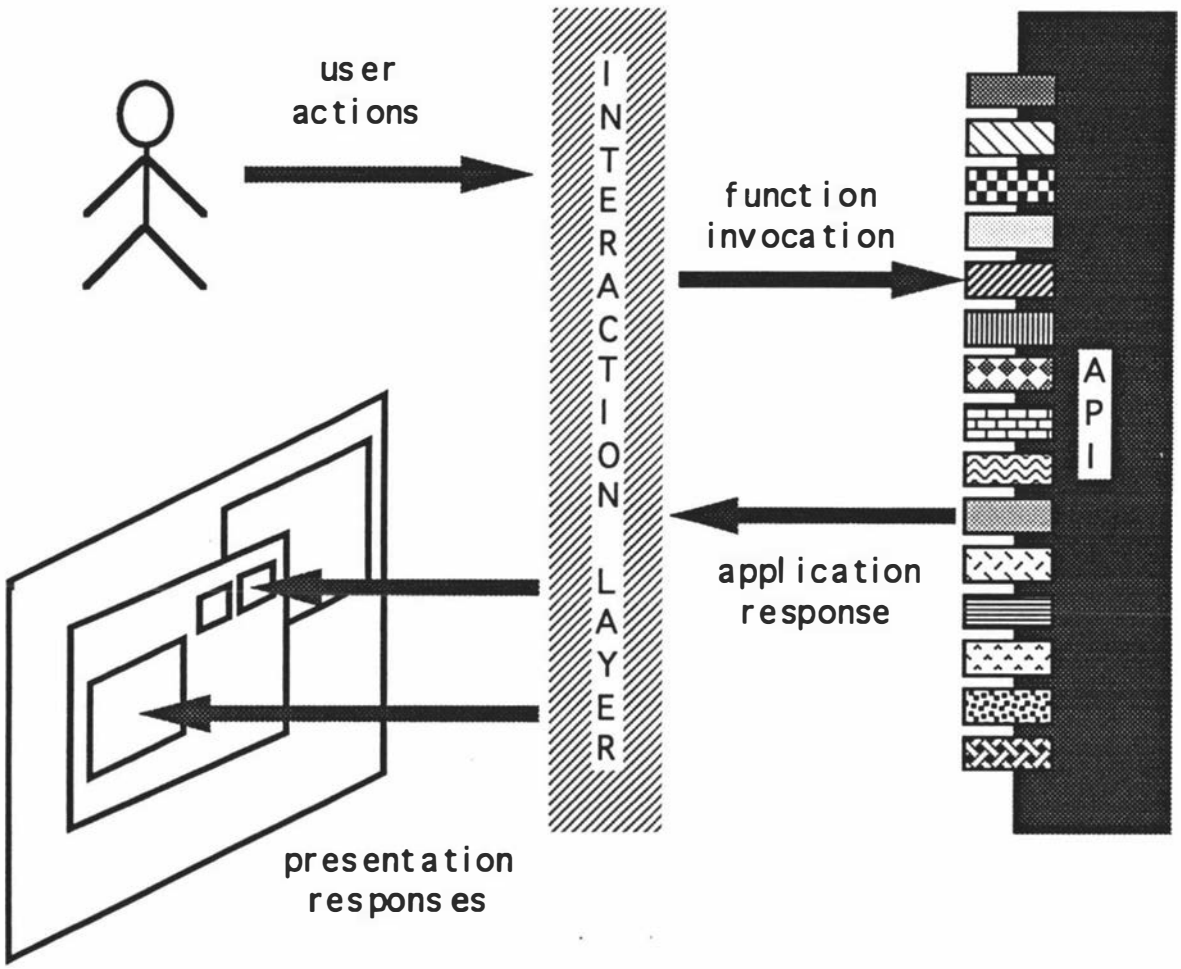


Figure 5.32: The interaction layer communication relationship to external entities.

Summarising, the DAL model implies a multiagent architecture within which a series of event handlers are hierarchically arranged. The architecture is divided (vertically) into display, dialogue and application layers. Events within this system fall into two categories, namely action and system events.

Action events are generated by the user and have the following characteristics:

i) Their ordering is important, they must be processed one at a time in the order in which they are generated.
ii) They are restricted to the dialogue layer of the interface.
iii) They are propagated down through the dialogue layer of the interface, starting at the root dialogue.

System events are generated within the dialogue layer as a consequence of event statements firing, or originate from the application layer. They have the following characteristics:

i) Their ordering is important, they must be processed in the same order as they are generated.
i) They have a scope defined in terms of the dialogue layer hierarchy.
ii) They are broadcast to all display, dialogue and application handlers that are able to consume them and fall within their context.

A single user action event may give rise to a large number of system events. It follows that these system events must be queued. System events correspond to the systems response to the user's actions. All responses to a user's action must be processed before the next user action. Hence all system events within the system must be processed prior to dealing with the next user action event. System events themselves may give rise to further system events. These system events produced in response to system events are indirect responses to the user action and hence must also be processed before the next user action.

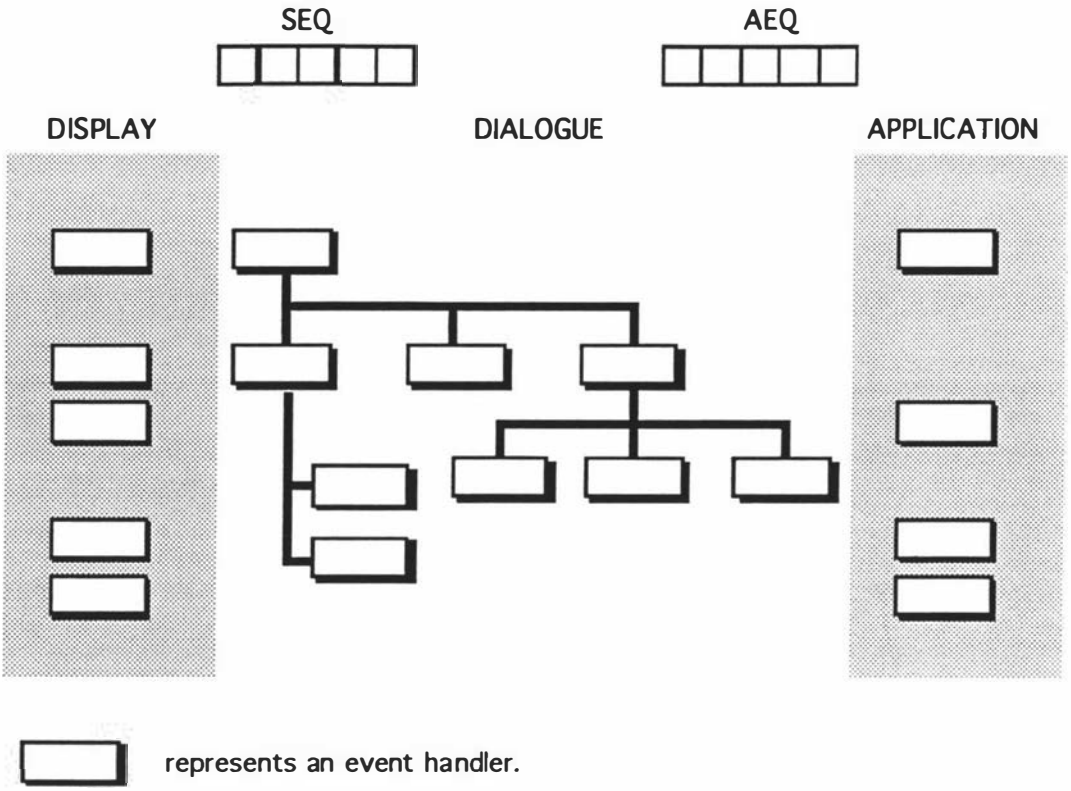


Figure 5.33: The components of the DAL system architecture.

The architecture of a DAL system is divided into three layers (Figure 5.33):

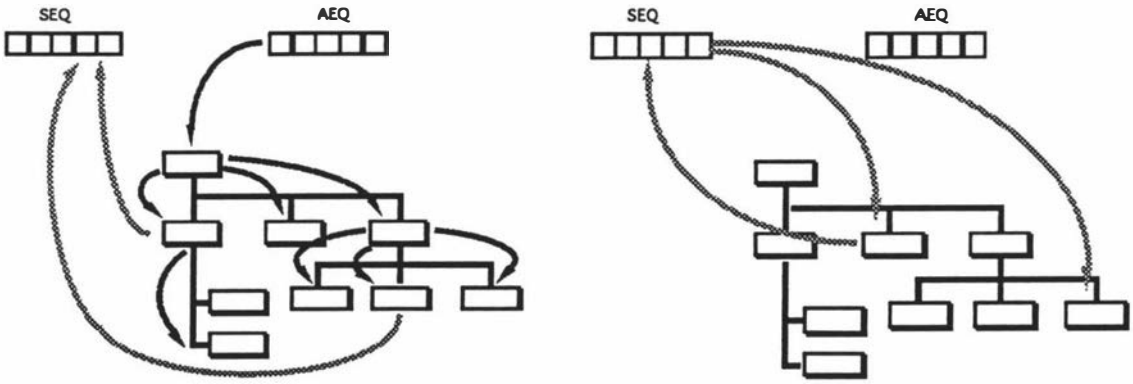
1. The ***display layer***; composed of display agents responsible for interpreting the context of user actions and displaying simple widgets.
2. The ***dialogue layer***; the μdialogue tree defined in DAL.
3. The ***application layer***; consisting of API agents responsible for linking the interface to the application.

In addition, there are two event queues:

AEQ Action Event Queue containing user events in the sequence in which they have been generated.

SEQ System Event Queue. A temporary holder for system events.

The execution cycle of this system has two distinct stages (Figure 5.34). In these diagrams the display and application layers have been omitted for brevity.



(a) (b)

Figure 5.34: DAL execution stages.

In the first stage (Figure 5.34(a)) the next action event is removed from the AEQ, and propagated top-down through the μdialogue tree. As the event is processed, system events may be generated in response and these are queued to the SEQ. When the action event has been completely processed, ie., it has propagated down as far as possible through the μdialogue tree, the second stage begins. In this stage (Figure 5.34(b)) the system event on the top of the SEQ is removed and broadcast to its recipient μdialogues, display and application handlers. These then execute to process the event. In so doing, further system events may be produced and are queued to the SEQ. This stage repeats until the SEQ is empty. The cycle then repeats (Figure 5.35).

```
REPEAT FOREVER
        Get action event for AEQ                                    Stage 1
    REPEAT
        Process event queuing any system events generated to the SEQ
        Propagate event
    UNTIL (no further propagation possible)
    WHILE (SEQ not empty)                                           Stage 2
        Get next system event from SEQ
        Broadcast to receptor μdialogue, widgets and api
        Process event, queuing any system events generated to the SEQ
    END WHILE
END REPEAT FOREVER
```

Figure 5.35: The DAL execution cycle.

This execution cycle is similar to that used by Hill for ERS [Hill87c ;Hill87b]. It is possible for endless cycles to result. Chapter 6 discusses an approach for detecting possible repeating cycles.

Chapter 6
Formal Semantics of DAL

6.1. INTRODUCTION

The purpose of this chapter is to produce an unambiguous definition of the structure and behaviour of a system using DAL. This is then used as a basis for identifying potential problems in the dynamic behaviour of such systems. Methods are proposed for statically analysing DAL systems for such problems.

The framework of DAL is based on Lean Cuisine [Apperley89], a formal analysis of which was carried out by Hudson [Hudson92]. Some use of the latter has been made in developing the formal description of the DAL skeleton.

6.2. THE DAL FRAMEWORK

A DAL interface specification is a tuple

(μD, Parent, Children, Type, Events, Variables, Guards, Activators, Deactivators, OnActStatements, OnDeactStatements, BodyStatements)

where:

(1)	*μD:*	a finite set of μdialogues
(2)	***Parent:***	a function returning the parent μdialogue $\mu D \rightarrow \mu D \cup \{\varnothing\}$
(3)	***Children:***	a function returning the set of child μdialogues for a given μdialogue μD->P(μD)
(4)	***Type:***	a function returning the type associated with a μdialogue μD->{MC, ME}
(5)	***Events:***	a finite set of events recognised by the system
(6)	***Variables:***	a finite set of variables defined within the system

(7)	***Guard:***	µdialogue -> guard condition
(8)	***Activators:***	µdialogue -> set of activating events
(9)	***Deactivators:***	µdialogue -> set of deactivating events
(10)	***OnActStatements***	µdialogue -> sequence of ***on activation*** statements
(11)	***OnDeactStatements***	µdialogue -> sequence of ***on deactivation*** statements
(12)	***BodyStatements***	µdialogue -> sequence of statements defined within the main body of the µdialogue.

satisfying:

(13) $\exists_1 x \in \mu D: Parent(x) = \varnothing$

There is only one µdialogue without a parent, this is the root of the system

(Note: $\exists_1$ means "there exists one and only one")

(14) $\forall x \in \mu D: \sim (\exists n > 0: Parent^n(x) = x)$

There are no cycles

(15) $\forall x, y \in \mu D: (x \in Children(y) \Leftrightarrow y = Parent(x))$

x is a child of y iff y is the parent of x

That is, the µdialogues form a tree, with one unique µdialogue forming the root.

6.3. SYSTEM STATES

A µdialogue has an associated ***activation state*** of ***active*** or ***inactive***. In addition, integer variables can be declared within a µdialogue. If follows that the overall state of the system can be defined in terms of both the activation state of each of the µdialogues forming the system and the value of all variables declared within the system.

The state of the system can be defined by a tuple:

(DialogueStates, VariableValues)

where:

DialogueStates is a function μD -> *{active,inactive}*

VariableValues is a function *Variables* -> *N* (where *N* is the set of all integers)

Before proceeding with the rest of this analysis, it is useful to define the following functions applicable to a µdialogue *x*:

(16) ***Default(x):*** $Activators(x) = \varnothing$

(17) ***MutuallyCompatible(x):*** $Type(x) = MC$

(18) ***MutuallyExclusive(x):*** $Type(x) = ME$

(19) ***Statements(x):***
$range(OnActStatements(x))$
$\cup\ range(OnDeactStatements(x))$
$\cup\ range(BodyStatements(x))$

The set of all statements contained within a given µdialogue x. In some of the following analyses, the order and repetition of statements is unimportant, hence a set of statements contains sufficient information to support the analyses.

The Boolean function ***IsEventStatement*** takes a given statement and returns *true* if that statement is an event statement, else *false* (if the statement is an arithmetic statement). Functions returning the set of event and arithmetic statements for a µdialogue, *x*, can now be defined:

The set of ***on activation*** event statements for µdialogue *x*.

(20) ***OnActEventStatements(x):***
$\{s \in range(OnActStatements) \mid IsEventStatement(s)\}$

The set of ***on deactivation*** event statements for µdialogue *x*.

(21) ***OnDeactEventStatements(x):***
$\{s \in range(OnDeactStatements) \mid IsEventStatement(s)\}$

The set of event statements contained within the main body of µdialogue *x*.

(22) ***BodyEventStatements(x):***
$\{s \in range(BodyStatements) \mid IsEventStatement(s)\}$

The set of all event statements defined for μdialogue x.

(23) ***EventStatements(x):***

$$\{s \in Statements(x) \mid \text{IsEventStatement(s)}\}$$

The set of all arithmetic statements defined for μdialogue x.

(24) ***ArithmeticStatements(x):***

$$\{s \in Statements(x) \mid \neg\text{IsEventStatement(s)}\}$$

On completion of processing an event the system must be left in a ***stable state*** satisfying the following conditions:

$\forall x, y, z \in \mu D$:

(25) $((s(x) = active \wedge Parent(x) \neq \varnothing) \Rightarrow s(Parent(x)) = active)$

A μdialogue can be active only if its parent is also active. It follows that deactivating a μdialogue will cause the μdialogue subtree beneath it to deactivate.

(26) $((MutuallyExclusive(x) \wedge x = Parent(y) = Parent(z)$
$\wedge s(y) = s(z) = active) \Rightarrow y = z)$

A mutually exclusive μdialogue can have only one active child.

(27) $(s(x) = active \wedge (\exists y \in Children(x) \mid Default(y))$
$\Rightarrow (\exists z \in Children(x) \mid s(z) = active)$

If a μdialogue has a default child, then if the μdialogue is active then one of its children will also be active. In the case of a mutually compatible μdialogue a default child will be active as long as the parent is active. For a mutually exclusive μdialogue a default child will be active if none of its siblings is active. Since in both cases the result of deactivating a default μdialogue directly would result in it immediately being reactivated, it follows that it does not make sense for a default μdialogue to have a deactivator.

(28) $Default(x) \Rightarrow (Deactivators(x) = \varnothing)$

Similarly a guard conditions associated with a default µdialogue would be an anachronism, since by rule (27), a default µdialogue should be able to activate whenever its parent activates. Hence:

(29) $Default(x) \Rightarrow (Guard(x) = \varnothing)$

6.4. STATE TRANSITIONS AND EVENT PROPAGATION

The state of the system is changed in response to events. Conversely, the propagation of events through the system is dependent on the state of the system. In considering changes in the system state it is necessary to take an event perspective.

Events in the system fall into two disjoint sets:

E_a: set of user **action events**

E_S: set of **system events**

(30) $Events = (E_a \cup E_s)$
$E_a \cap E_s = \varnothing$

Action events propagate down through the system as explained below in section 6.4.1, whereas system events have a defined scope, and are broadcast as described in section 6.4.2.

If an event is presented to an active µdialogue, it is processed, propagated down the tree (if the event is an action event) and then if the event is a deactivator for the µdialogue, the µdialogue will deactivated. Only one event is processed by a µdialogue at once. This event is referred to as the ***current event***.

If an inactive µdialogue is presented with an event which matches one of its activators, then that µdialogue is activated. It follows that in the case of a mutually exclusive group of µdialogues, for the system to be deterministic there must be no two µdialogues with the same activator.

(31) $(MutuallyExclusive(x) \Rightarrow$
$(\forall y,z \in Children(x) \mid ((Activators(y) \cap Activators(z)) \neq \varnothing) \Rightarrow y = z))$

This also applies to the case where the set of Activators = Ø; ie, you cannot have more than one default µdialogue in a mutually exclusive grouping.

6.4.1. Action Event Propagation

If an event is a deactivator for a μdialogue, then the event is first propagated, then the μdialogue deactivates, the effect of this deactivation being reflected in the underlying subtree being deactivated in accordance with rule 25. The algorithmic description of the deactivate function is:

(32)
```
deactivate(e,x):
        execute(e,OnDeactStatements(x))
        for y in Children(x) do
                if ( s(y) = active ) then
                        deactivate( nil,y )
                endif
        endfor
        set s(x) to inactive
```

The function ***execute*** takes an event and a sequence of statements. Statements have an associated type, the function ***EventStatement*** returning *true* if the supplied statement is an event statement and returning *false* for an arithmetic statement. Event statements have two parts, an *originating event* against which the current event is compared and a *resultant event* which is generated if the current event matches the originating event, or if the originating event is *nil*. The generation of an event is represented by the *emit* function in the following algorithmic description.

Arithmetic statements are composed of a variable identifier and an expression. The function ***lhs***[1] takes an arithmetic statement as its argument and returns the variable to be set. The function ***rhs***[2] takes an arithmetic statement and returns the expression to be evaluated. Execution of an arithmetic statement consists of evaluating the expression (signified by the ***eval*** function in the following algorithmic description) and setting the value of the *lhs* variable to the resulting value.

---

[1] lhs stands for "left hand side"

[2] rhs stands for "right hand side"

(33)
```
execute(e,s):
   for y in s do
      if (EventStatement(s) ∧
            (e = OriginatingEvent(s) ∨ OriginatingEvent(s) = nil)) then
         emit(ResultingEvent(s))
      else ... ArithmeticStatement(s)...
         set lhs(s) to eval(rhs(s))
      endif
   endfor
```

When a µdialogue changes from the inactive to the active state (it is said to ***activate***), any ***on activation*** statements defined for the µdialogue will be executed. Similarly a µdialogue changing from an active to an inactive states (deactivating) would result in any ***on deactivation*** statements defined for the µdialogue being executed. The execution of actions in response to entering or leaving a state is similar to that seen within the extended version of statecharts used in Statemaster [Wellner89].

In contrast to the execution of actions in response to state changes, the statements defined within the body of a µdialogue define its behaviour when it receives an event whilst in the active state. In this case the system is acting in a similar fashion to processes in the RT-SASD formalism [Hatley87], in which processes defined in a data flow diagram are switch on and off in response to state changes in an associated STN.

Guard conditions are evaluated according to the following rule. If there is no guard condition specified, or the result of evaluating the guard expression is not zero, then the guard condition is true. If the guard expression evaluates to zero, then the guard condition is false. This is expressed in the

(34)
```
GuardTrue(x):
   if (Guard(x) = ∅ ∨ eval(Guard(x)) ≠ 0) then
      true
   else
      false
   endif
```

Given an action event *e*, its propagation through a µdialogue subtree starting at µdialogue *x*, can be described recursively as shown in the following algorithmic description of the propagate procedure:

propagate(e,x):

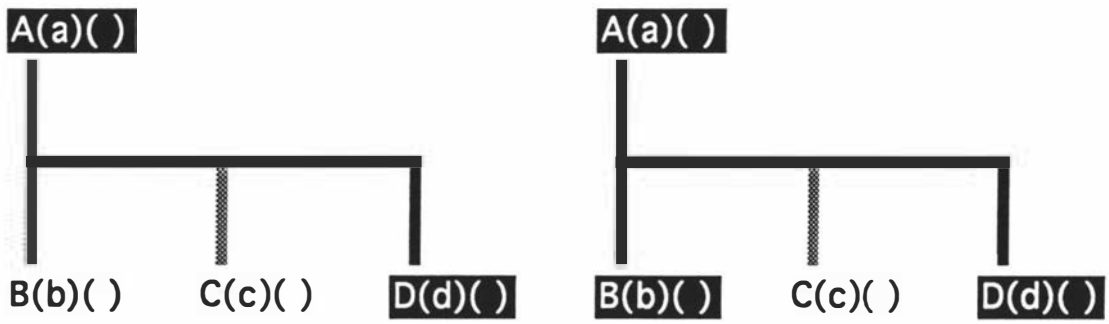
```
        // If this µdialogue is inactive and can be activated, activate and
        // execute any on activation statements.
(35)
        if ((s(x) = inactive) ∧ (Default(x) ∨
                (e ∈ Activators(x) ∧ GuardTrue(x)))) then
                set s(x) to active
                execute(e,OnActStatements(x))
        endif

        // If the µdialogue is was already active, or activated by the previous
        // block execute any statements contained in the body of the µdialogue
(36)
        if (s(x)=active) then
                execute(e,BlockStatements(x))

                // For MC µdialogues, propagate the event down through any
                // active µdialogues or any capable of activating.
(37)
                if (MutuallyCompatible(x)) then
                        for y in Children(x) do
                                if (s(y) = active ∨ Default(y) ∨
                                        (e ∈ Activators(y) ∧ GuardTrue(y))) then
                                        propagate(e,y)
                                endif
                        endfor

                else    ... MutuallyExclusive(x) ...
                        Boolean done
                        set done to FALSE

                        // Start by looking for an inactive (but not default) child
                        // µdialogue capable of activating. If we find one deactivate
                        // any active children, then propagate the event down
                        // through the child.
(38)
                        for y in Children(x) do
                                if (s(y) = inactive ∧
                                        (e ∈ Activators(y) ∧ GuardTrue(y))) then
                                        for z in Children(x) do
                                                if (s(z)=active) then
                                                        deactivate(nil,z)
                                                endif
                                        endfor
                                        propagate(e,y)
                                        set done to TRUE
                                endif
                        endfor
```

```
                    // No µdialogues can be activated with this event so if
                    // there is an active µdialogue then propagate the event
                    // down through it.
(39)
                    if ( NOT done ) then
                            for y in Children(x) do
                                    if (s(y)=active) then
                                            propagate(e,y)
                                            set done to TRUE
                                    endif
                            endfor
                    endif


                    // Locate a default child, if one exists and propagate
                    // the event down through it.
(40)
                    if ( NOT done ) then
                            for y in Children(x) do
                                    if (Default(y)) then
                                            propagate(e,y)
                                    endif
                            endfor
                    endif

            endif   ...MutuallyExclusive...

            // Finally check if this event is a deactivator for this µdialogue
            // and if so deactivate it (which executes any on deactivation
            // statements and deactivates any µdialogue sub-trees beneath).
            // The parent µdialogue needs to be informed, so that any default
            // children are activated.
(41)
            if (e ∈ Deactivators( x)) then
                    deactivate(e,x)
                            if  (Parent(x) ≠ nil ∧
                                    MutuallyExclusive(Parent(x)) then
                                    NotifyDeactivated(Parent(x));
                            endif
            endif
    endif   ...active...
```

Taking each section of the algorithm in turn:

Section 35 is responsible for activating inactive µdialogues. Any ***on activation*** statements defined for the µdialogue will be executed as a result. If the

Section 36 is only executed for mutually compatible groups. The propagation of the event for a mutually compatible group is down through each child that is active, default, or for which the current event is an activator. Consider the system in Figure 6.1[3]. If the initial state of the system is {A,D} (Figure 6.1(a)) and the current event for A is 'b', then the final state of the system will be {A,B,D} (Figure 6.1(b)), and the event 'b' will be propagated by μdialogues B and D.



(a) (b)

Figure 6.1: Propagation through a MC grouping.

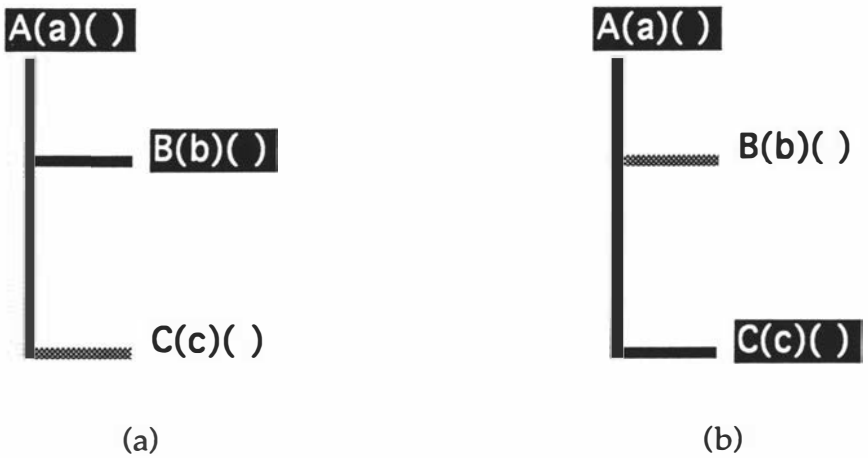
If the event is a deactivator for any active μdialogues then provided that they are not defaults (ie. they have one or more activators) then the μdialogues will be activated.

It is important to note that propagation down from a mutually compatible μdialogue is (a least conceptually) done in parallel. That is the ***for y in*** *Children(x)* ***do*** in the algorithm is really a parallel operation.

Section 38 is executed for a mutually exclusive group, if it contains a child that has the current event as one of its activators. From rule 31, this can only apply to a single μdialogue within a mutually exclusive grouping. In this case the child containing the matching activator becomes active and all of its siblings will be left in an inactive state in keeping with rule 26 above.

[3] In these diagrams high lighted nodes correspond to active μdialogues.

Consider the system in Figure 6.2. If the initial state of the system is {A,B} (Figure 6.2(a)) and the current event for A is 'c', then the final state of the system will be {A,C} (Figure 6.2(b)), and the event 'c' will be propagated by µdialogue C.



(a) (b)

Figure 6.2: Propagation through a ME grouping (1)

Section 39 is executed for a mutually exclusive group, if parent µdialogue contains an active member and the current event is not an activator for any members of the group. In this case there is no change of state, the event simply propagating through the currently active member.

Considering the system depicted in Figure 6.2 (a) again, but this time with the current event for A being 'x'. There would be no change of state for the system and the event x would be propagated by B.

Section 40 ensures that the rule 27 is adhered to, with the event being propagated down through a default child if one exists.

Section 41 is executed for all µdialogues if the event they are processing is a deactivator for the µdialogue. Such events cause the µdialogue to become deactivated, resulting in the underlying µdialogue tree to become inactive, and any ***on deactivation*** statements to be executed. If the parent µdialogue is mutually exclusive, then it needs to be notified that this µdialogue is deactivating so that any default children are activated in accordance with rule 27. This is achieved through the ***NotifyDeactivated*** function defined below:

(42)
```
NotifyDeactivating(x):
        for y in Children(x) do
                if ( Default(y) ) then
                        activate(y)
                endif
        endfor
```

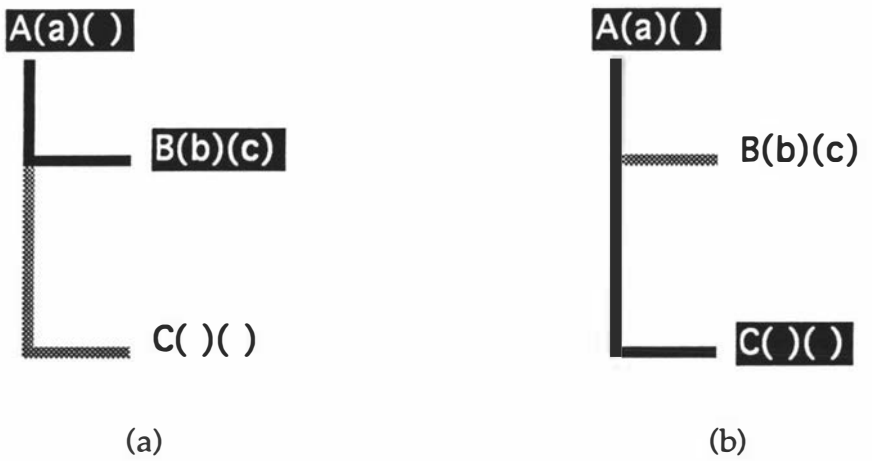
Mutually compatible parent µdialogues do not need to be notified in this way, since default µdialogues of a mutually compatible parent remain active, as long as the parent is active. The ***activate*** function referenced within the definition of NotifyDeactived causes any *on activation* statements to be executed, and any underlying default µdialogues to be activated:

(43)
```
activate(x):
        execute(nil, OnActStatements(x) )
        for y in Children(x) do
                if ( Default(y) ) then
                        activate(y)
                endif
        endfor
```

Consider the system in Figure 6.3. If the initial system state is {A,B}, and the current event for A is 'c', then the following actions will occur:

- The event 'c' will be propagated by B.
- B will deactivate.
- A will be notified of B's deactivation
- C will be activated by A.

The final system state will now be {A,C}.



(a) (b)

Figure 6.3: Activation of a default in a mutually exclusive group.

6.4.2. System Event Broadcasting

If a system event is generated, then it is broadcast to those recipient μdialogues that are in scope and are currently active, or for which the event is an activator. To describe this, it is necessary to define the ***scope*** of an event and defining the meaning of ***receptor***.

Scope of System Events

A system event is uniquely identified by its name and the μdialogue within which it is declared. That is an ordered pair:

(44) $e = (\mu, i)$

where

- μ is the μdialogue in which e is declared.
- i is the identifier (name) of the event.

Notation:
In what follows, given an event x, $x.\mu$ refers to the μdialogue within which it is declared and similarly $x.i$ refers to its identifier.

The ***scope*** of a system event is the set of μdialogues that will receive a copy of the event in response to its production.

For a system event, e, the scope is defined as the set of μdialogues forming the subtree under, and including $e.\mu$. and excluding the scope of any system events with the same identifier with an intersecting scope.

We can define Subtree as a recursive function in the following way:

(45) $$\textit{Subtree}(\mu) = \{\mu\} \cup \bigcup_{x \in children(\mu)} \textit{Subtree}(x)$$

System events with the same identifier and falling in the same scope as event e are:

(46) $$\textit{MatchingEvents}(e) = \{x \in E_s \mid e \neq x \wedge e.i = x.i \wedge x.\mu \in \textit{Subtree}(e.\mu)\}$$

The scope of an event e is obtained by subtracting the scope of the matching events from the subtree under the μdialogue within which e is declared.

(47) $$\textit{EventScope}(e) = \textit{Subtree}(e.\mu) - \bigcup_{x \in MatchingEvents(e)} \textit{Subtree}(x.\mu)$$

Receptors of system events

A ***receptor*** of an event e is a μdialogue that is within the scope of e, and that is able to use e in one way or another, either as:

i) An activator.
ii) A deactivator.
or iii) An originating event in an event statement within the μdialogue.

An **event statement**, e_s, can be modelled as an event pair

(48) $$e_s = (e_r, e_o)$$

where

e_r is the ***resulting system event.***

e_o is the ***originating event***, ie., the event that causes the statement to fire, generating e_r. This originating event may by *nil.*

Consider for example the event statement:

```
X <- A;
```

X is the resulting system event, e_r

A is the originating event, e_o

We can define a Boolean function, ***Accepts***, that for an event, e, and μdialogue, x, will return true if x is a receptor for e.

(49) $$\textbf{\textit{Accepts}}(x,e): (e \in Activators(x) \vee e \in Deactivators(x) \vee e \in range(EventStatements(x)))$$

Then the set of μdialogues that are receptors for an event, e, is:

(50) $$\textbf{\textit{Receptors}}(e) = \{\forall x \in EventScope(e) \mid Accepts(x,e)\}$$

State Transitions due to System Events

On receiving a system event, a μdialogue undergoes exactly that same state transitions as it would for those associated with action events, as defined in the function *propagate*. The only difference is that unlike action events, system events are not propagated, their sphere of influence being restricted to each of their receptor μdialogues. If a system event activates a receptor, any default μdialogue children will be activated in accordance with rule 27 above. Similarly, if a system event deactivates a receptor, then any parts of the underlying subtree that are active will be deactivated in accordance with rule 25.

The state changes and processing associated with the receipt of an event, e, by a μdialogue, x, is defined within the procedure ***receive***:

(51)
```
receive(e,x):
    if ( s(x) = inactive ) then
        if (Parent(x) ≠ nil ∧ s(Parent(x)) = active ∧
            (e ∈ Activators(x) ∧ GuardTrue(x))) then
            if (Parent(x) ≠ nil ∧
                MutuallyExclusive(Parent(x))) then
                NotifyActivating(Parent(x),x)
            endif
            set s(x) to active
            execute( e, OnActStatements(x) )
        endif
    endif
    if ( s(x) = active ) then
        execute( e, BodyStatements(x))
        if (e ∈ Deactivators(x)) then
            deactivate( e, x )
            if (Parent(x) ≠ nil ∧
                MutuallyExclusive(Parent(x))) then
                NotifyDeactivated(Parent(x))
            endif
        endif
    endif
```

The *receive* procedure is in many respects similar to the *propagate* procedure defined for action events, the main difference being that system events are not propagated down the tree.

One minor complication is that if an event causes a µdialogue to activate, and if that µdialogue is a member of a mutually exclusive set, then any active siblings need to be deactivated. This is achieved with the procedure *NotifyActivating*:

(52)
```
NotifyActivating(x,z):
    for y in Children(x) do
        if ( s(y) = active) then
            deactivate(nil,y)
        endif
    endfor
```

Similarly, if the a system event causes a µdialogue to deactivate, and if that µdialogue is a member of a mutually exclusive set, then if a default sibling exists, then it needs to be activated. This is achieved with the procedure *NotifyDeactivated* defined above.

6.4.3. Anonymous Transitions

There are three circumstances when an event may indirectly change the state of a μdialogue.

i) When a μdialogue is active and a member of a mutually exclusive group, and the event causes one of its siblings to activate. In response the μdialogue in question will deactivate.

ii) When a default μdialogue in a mutually exclusive set is inactive, and any of its siblings become inactive but the parent is still active. In this situation the μdialogue in question must activate to satisfy rule 27.

iii) When a μdialogue is active and the parent is deactivated.

These will be referred to as ***anonymous transitions***. In these situations any *on activation* or *on deactivation* statements are executed. However, if any events statements reference an originating event, then they will never 'fire' during these anonymous transitions.

Anonymous transitions are deemed to be atomic operations with respect to the rest of the system. To understand what this means, consider the following example. If a μdialogue, x, in a mutually exclusive group is activated, and one of its siblings, y, is active, then this sibling will be anonymously deactivated in response. This deactivation of y executes before x is activated. If y has any active children, then these will also be deactivated before x is activated. Within a subtree that is undergoing an anonymous transition, the μdialogues undergo their state changes top-down within the tree.

6.5. VARIABLES

Variables can be uniquely identified and scoped in the same way as system events.

(53) $v = (\mu, i)$

where

μ is the μdialogue in which v is declared.

i is the identifier (name) of the variable.

Following the same conventions used for system events, the set of variables with an intersecting scope can be defined:

(54) $$\textbf{\textit{MatchingVars}}(v) = \{x \in Variables \mid v \neq x \wedge v.i = x.i \wedge x.\mu \in Subtree(v.\mu)\}$$

and using this, we can define a function, ***VarScope***, that returns the scope of a given variable.

(55) $$\textbf{\textit{VarScope}}(v) = Subtree(v.\mu) - \bigcup_{x \in MatchingVars(v)} Subtree(x.\mu)$$

An ***arithmetic statement***, a_s, can be modelled as an ordered pair:

(56) $$a_s = (v, Expr)$$

where:

v	corresponds to the *lhs* of the statement, the target of the assignment.
$Expr$	corresponds to the *rhs* of the statement. The set of variables whose value is used in computing the value of the expression.

In the following discussion, if s is an arithmetic statement then $s.v$ is the variable on the lhs of the statement and $s.Expr$ is the set of variables on the rhs of the statement.

The functions *rhsVars* and *lhsVars* return for a given µdialogue the set of variables on the rhs and lhs of its arithmetic statements respectively:

(57) $$\textbf{\textit{rhsVars}}(x) = \bigcup_{s \in ArithmeticStatements(x)} s.Expr$$

(58) $$\textbf{\textit{lhsVars}}(x) = \bigcup_{s \in ArithmeticStatements(x)} s.v$$

Guard was defined at the beginning of this chapter as a function returning the guard condition associated with a given µdialogue. The guard condition will be simply recognised within this analysis by the set of variables contained within it.

We can now define two functions that return, for a given variable, the set of μdialogues for which the variable has read access (*Readers*) and write access (*Writers*) respectively:

(59) $\mathbf{Readers}(v) = \{x \in VarScope(v) \mid v.i \in rhsVars(x) \vee v.i \in Guard(x)\}$

(60) $\mathbf{Writers}(v) = \{x \in VarScope(v) \mid v.i \in lhsVars(x)\}$

6.6. CONCURRENCY

As already described, a μdialogue executes its *on activation* and *on deactivation* statements in response to state changes. In addition, an active μdialogue will execute its body statements on receiving an event. The statements defined in a μdialogue are generally independent of statements defined in another. Consequently, it is possible for two μdialogues to be executed concurrently.

A μdialogue can only execute if it is in the active state. It follows that provided that two μdialogues can be active at the same time, then the two μdialogues can be executed in parallel. Consider the system depicted in Figure 6.4. The μdialogues *B*, *C* and *D* are within a mutually compatible group. Because of this, it is possible for them all to be in the active state at the same time, and hence could be executed at the same time. Since from rule (25) a μdialogue can only be active if its parent is active, a μdialogue will always be active at the same time as any μdialogue ancestrially related to it, for example *E* and *A* in Figure 6.4.

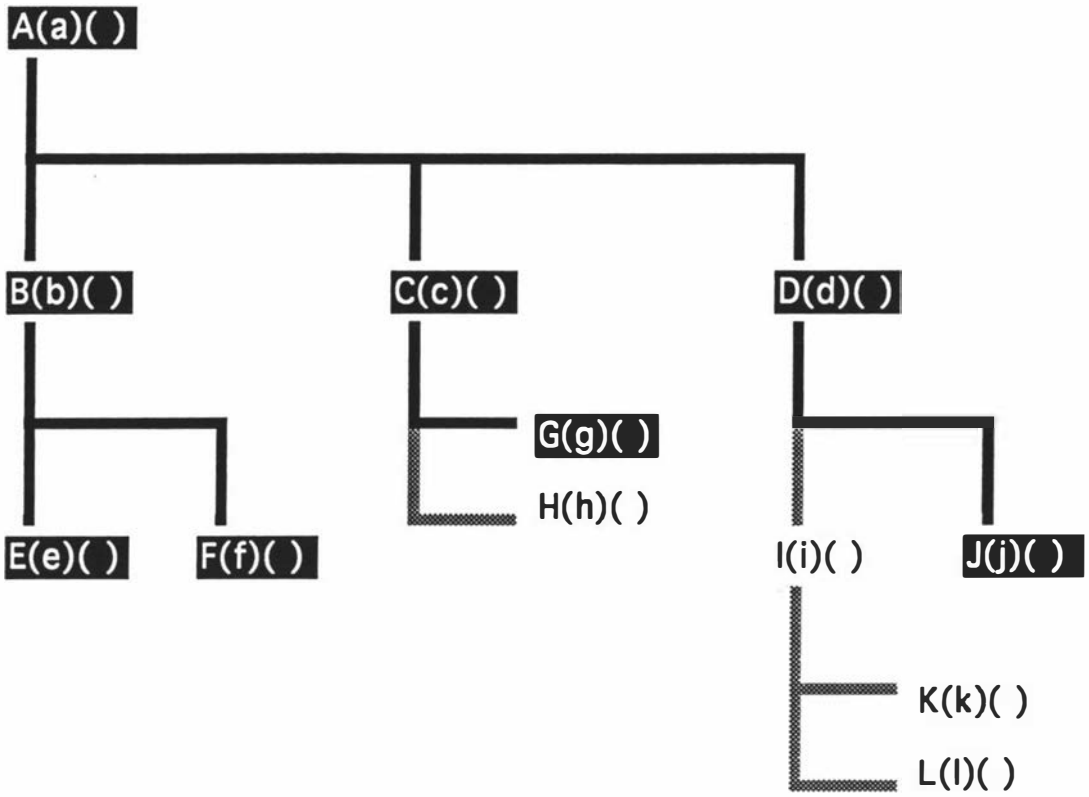


Figure 6.4: Concurrently executing spines.

A *Spine* is a sequence of µdialogues, each related with the previous by way of the parent/child relationship. A spine starts at a given µdialogue and terminates at the root. We can define the function Spine(d), that returns the sequence of µdialogues constituting the spine for µdialogue *d* in the following way:

(61) *Spine(d): if Parent(d) = nil then (d)*
else (d)++Spine(Parent(d))

(where *a ++b* is the concatenation of the sequences *a* and *b*)

Spines may overlap other spines. All spines overlap at the root. In the above example (A,B,E) and (A,B,F) are spines. Two spines meet at their Nearest Common Ancester (NCA). Given two spines, x and y, we can define a function returning their NCA in the following way:

(62) $NCA(x,y)$: ***if*** $x = (nil) \vee y = (nil)$ ***then***
nil
else if $head(x) \in Range(y)$ ***then***
$head(x)$
else $NCA(tail(x),y)$

if the NCA of two μdialogues is mutually compatible, then it is possible for the μdialogues to be concurrent during the first stage of the execution cycle.

In the first phase of the execution cycle, events are passed down the tree from parent to child. For this reason two μdialogues can only execute concurrently if they do not belong to the same spine. Given two μdialogues, x and y, the Boolean function ***Separate*** identifies if the they are located on separate spines.

(63) $Separate(x,y)$: $x \notin Range(Spine(y)) \wedge y \notin Range(Spine(x))$

We can now determine if two μdialogues can be concurrent in this first stage by ascertaining if they are on separate spines and if their NCA is mutually compatible:

(64) $Concurrent1(x,y)$: $Separate(x,y) \wedge MutuallyCompatible(NCA(x,y))$

In the second stage of the execution cycle, system events are processed by broadcasting them to receptor μdialogues. Receptors could be on the same spine, hence concurrency at this stage is global, ie. both across and down the tree. Given two μdialogues x and y, they can be concurrent in this second stage if they both accept the same event and can both be active, or have active parents at the same time.

(65) $Concurrent2(x,y)$: $(e \in E_s \mid (Accepts(x,e) \wedge Accepts(y,e))$
$\Rightarrow MutuallyCompatible(NCA(x,y))$

Two μdialogues can be potentially concurrent if they could be concurrent in either stage 1 or stage 2.

(66) $Concurrent(x,y)$: $(Concurrent1(x,y) \vee Concurrent2(x,y))$

6.7. POTENTIAL SYSTEM PROBLEMS

Potential problems of a DAL system fall into two main categories:

i) Faults associated with concurrent systems in general, for example race conditions.
ii) Faults particular to DAL, for example non-reachability and redundancy.

The approach taken in this section is to show how faults may be identified so that within an implementation of a development environment, static analysis of a design can be used to draw the attention of the developer to the possibility of a problem. The ability of static analysis to identify such problems is limited. For example consider the system depicted in Figure 6.5.

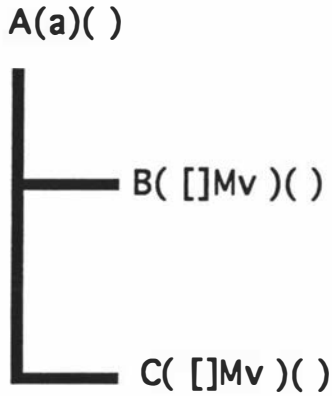


Figure 6.5: Possible non-determinism

In this example we have a mutually exclusive group, with two μdialogues with matching activators. The activators in question have a specified context and hence it is only possible to ascertain if this specification contravenes rule 26 if we know if the corresponding widgets intersect. Indeed the problem may be more complex than this since the widgets may be dynamic and hence moved at run time. Hence their spatial relationships may change, in which case it may be impossible to determine if this system is non-deterministic.

6.7.1. Concurrency Related Faults

A *Race Condition* is non-deterministic system behaviour due to unsynchronised, time dependent actions by parts of the system. They are well known in shared memory parallel programs where they occur as a consequence of unsynchronised access to shared data [Tanenbaum87; Netzer92]. It is possible to define a system in DAL that allows multiple access to shared data,

and hence will result in a race. These race conditions will be referred to as ***General Races*** in keeping with the categorisation of races presented in [Netzer92]. There are other potential sources of non-deterministic time dependent behaviour within a DAL system. Since a system event originating from a µdialogue in one spine can change the activation state of a µdialogue in another, it follows that if two or more such interactions occur within a single execution cycle the behaviour can be non-deterministic, and can cause a race. These races will be referred to as ***Activation Races***. Finally, non-determinism of a guard condition owing to it containing a variable shared between concurrent µdialogues can also lead to a race condition with respect to system state. This form of race will be referred to as ***Guard Race***.

General Races

Shared variables are a problem if their value can be changed by more than one potentially concurrent µdialogue. Consider for example the system in Figure 6.6. In this example µdialogues *B* and *C* are concurrent. If *B* executes first, then the variable y will contain the initial value of x, that is 1. If *C* executes first, then the value of x is updated prior to the assignment to y, and y will be left containing the value 2.

```
A( )( ){
        VAR x, y;
        ON ACTIVATION{ x = 1; }
        B( )( ){
                ON ACTIVATION{ y = x; }
        }
        C( )( ){
                ON ACTIVATION{ x = x + 1; }
        }
}
```

Figure 6.6: A General Race.

For a general race to happen the following conditions would have to apply:

i) The µdialogues would need to be truly concurrent (as defined in function 66).

ii) Two or more µdialogues are able to read the variable concurrently, and at least one of the µdialogues is able to change the variable.

We can define a set ***SharedConflict*** containing the variables for which there is the potential for an access conflict. For such a conflict to exist, one or more μdialogues must have read access and one or more μdialogues must have write access, the μdialogues in question being concurrent with respect to one another.

(67) ***SharedConflict:***
$\{v \in Variables \mid$
$(\exists x,y \in VarScope(v) \mid x \neq y \wedge$
$x \in Readers(v) \wedge y \in Writers(v) \wedge Concurrent(x,y))\}$

Activation Races

These occur when a particular sequence of system events is specified, but their order is undefined since they originate from concurrent, unsynchronised sources. Consider the system in Figure 6.7. If initially μdialogue *A* is inactive, and then receives event *'a'*. It activates, as do its children *B* and *C*. *B* and *C* are concurrent and generate events *x* and *y* respectively on activating. These system events will be placed in the system event queue for subsequent broadcasting in the next stage of the execution cycle. Since these events originate from concurrent sources, it follows that the SEQ may contain the sequence (x,y) or (y,x) on completion of this stage. After the subsequent broadcasting and processing of the events in the SEQ the final system state would end up as {A,B,C,D,E} in the former case or {A,B,C,D} in the latter.

```
A( a )( ){
        EVENT x, y;
        B( )( ){
                ON ACTIVATION{ x; }
        }
        C( )( ){
                ON ACTIVATION{ y; }
        }
        D( x )( ){
                E( y )( ){ }
        }
}
```

Figure 6.7: An Activation Race.

There are two distinct situations that can give rise to this problem. The first is where the system events are generated in the action event propagation stage of the execution cycle, in which concurrent sources are as defined by function ***Concurrent1*** above. The second is in the second stage of the execution cycle, in

which the concurrent sources are as defined in the function ***Concurrent2*** above[4].

Given two μdialogues, x and y, an activation race can occur iff

i) x and y are directly related.
ii) an activator for x, x_a is a system event.
iii) an activator for y, y_a is a system event.
iv) x_a and y_a can originate from concurrent sources.

(68) ***FeasibleActivationRace***(x, y):

$$(x \in Spine(y) \vee y \in Spine(x)) \wedge$$
$$(\exists e_1, e_2 \in E_s \mid (e_1 \in Activators(x) \wedge e_2 \in Activators(y)) \wedge$$
$$(\exists s_1 \in EventScope(e_1), \exists s_2 \in EventScope(e_2) \mid Concurrent2(s_1, s_2))$$

Guard Races

If a guard condition references one or more variables that are capable of being updated by concurrent μdialogues, then this can produce non-deterministic behaviour with regards to the system state. The concurrently executing μdialogues capable of updating the variable do not have to be concurrent with respect to the execution of the μdialogue containing the guard condition. Consider the example given in Figure 6.8.

[4]It should be noted that system events generated by executing μdialogues located on the same spine during the first stage of the execution cycle are generated in the order specified, and hence cannot give rise to activation races.

```
A( )( ){
        VAR x;
        GROUPING ME;

        ON ACTIVATION{ x=1; }

        B( x==2 )( b )( ){
        }
        C( )( ){
              D( )( ){
                    ON ACTIVATION{ x = x + 1; }
              }
              E( )( ){
                    ON ACTIVATION{ x = x * x; }
              }
        }
}
```

Figure 6.8: A Guard Race.

If the μdialogue *A* activates:

i) *C* activates by virtue of it being default.

ii) *D* and *E* both activate, being defaults in a mutually compatible group. In so doing they will execute their ***on activation*** statements unsynchronised. If *D* executes first, *x* will be left containing the value 4, conversely, if *E* executes first, *x* will be left containing the value 2.

If, subsequently *A* receives the event *b*, then if *x* contains the value 4 (due to *D* executing first), then *B* will not activate and the state of the system will remain {A,C,D,E}. If however *x* contains the value 2 (due to *E* executing first), then *B* will activate, and the final system state will be {A,B}.

In order for there to be a feasible guard race, there has to be a SharedConflict variable (as defined in 64) that is also present in a guard condition. The set of all variables present in guard conditions is:

(69) $\textbf{\textit{GuardVars}}: \bigcup_{d \in \mu D} Guard(d)$

it follows that the set of variables that can potentially produces a guard race is:

(70) $\textbf{\textit{GuardRaceVars}}: GuardVars \cap SharedConflict$

6.7.2. DAL Specific Faults

Apart from the concurrency related issues, two further problems are possible with systems defined in DAL.

Non-reachability

Consider the system in Figure 6.9, and assume events *a* and *b* to be action events. If *A* is inactive and receives event *a*, then the following actions occur:

i) *A* activates.
ii) Event *a* is processed within the body of *A*.
iii) An attempt is made to propagate event *a* to any children of *A*. Since *B* requires event *b* to activate, event a is not propagated.
iv) *A* deactivates, since event *a* is a deactivator for *A*.

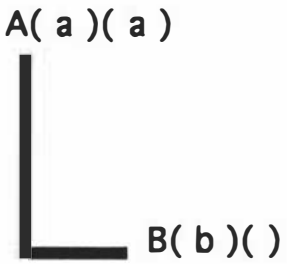


Figure 6.9: Non-reachability.

It follows from this that µdialogue B can never be presented with any event other than a, hence B is unreachable.

In trying to analyse this behaviour, it is worthwhile considering what we can deduce about the stream of events that can be presented to a µdialogue. Consider an inactive µdialogue *A*, with activators *a,b,c* as shown in Figure 6.10.

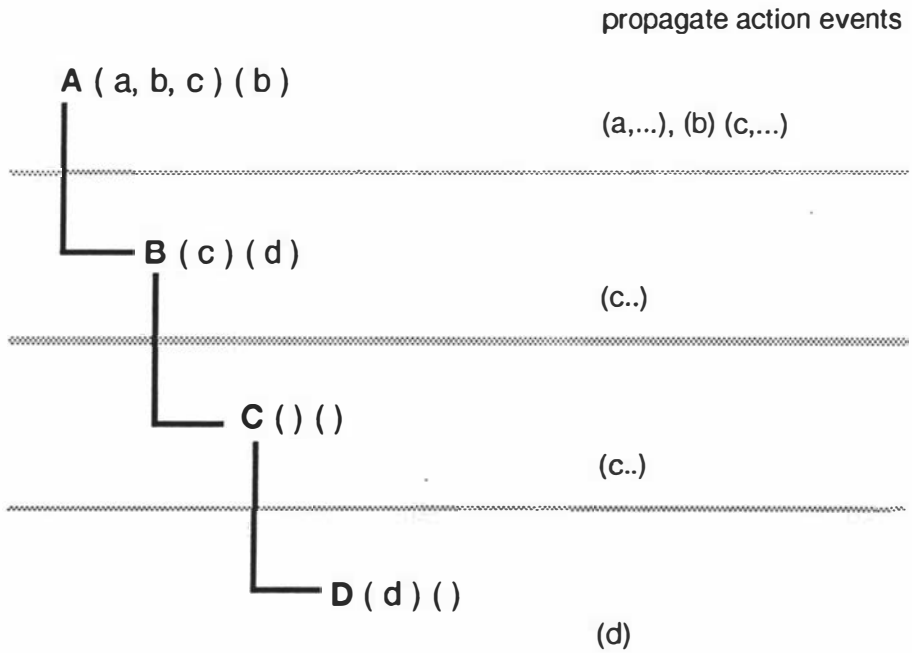


Figure 6.10: Restricted action event streams.

If *A* receives event *a*, it will activate, and proceed to propagate event *a* and all subsequent events until either it receives a deactivating event or is deactivated through some other means. Considering this stream of events as a sequence, we know that the first event has to be an activator, follow by a sequence of any type and number of other events available in the alphabet of events capable of being presented to the µdialogue. Hence in the case of this µdialogue *A* these possible event sequences would be *(a,...)*, *(b,...)* or *(c,...)*, where '...' signifies an undefinable sequence of events coming from those available to *A*. Since *A* has event *b* as a deactivator, we know that the stream *(b,...)*, will be terminated immediately, that is the possible event streams are *(a,...)*, *(b)* or *(c,...)*. Any children of *A* will have these event sequences presented to them. The streams *(a,...)* and *(b,...)* can contain all of the events present in the alphabet presented to *A*, hence provided that they have either no activator, or an activator present within this alphabet they can be reached. Imagine that *A* has a child *B*, with a single activator *c*, and deactivator *d*. It follows that the event stream passed on by *B* will start with *c* . If *B* has a default child *C*, then the event stream presented to *C* will be propagated to its children unchanged. Finally, consider that *C* has a child *D*, with a single activator, event *d*. If the event *d* occurred in the event stream, then it would be the last event in the event stream presented to *D*, since it is a deactivator for *B*. It follows that the only event *D* will propagate to its children will be *d*.

Generalising from these observations:

i) The action events available to a child µdialogue are those available to its parent if either the parent is a default, or all of its activators are not also its deactivators.

ii) The action event at the head of the event stream is one of the activators of the nearest, non-default ancestor, or undefined, being any one of those present in E_a, if there is no default ancestor.

iii) If there is a non-default ancestor, and its activators are deactivators for any of the µdialogues forming the sub-spine between and including it, and the µdialogue in question, then only those activating events are present in the alphabet of events propagated by the µdialogue.

In order to formalise these observations it is useful to define the following functions:

(71) ***NNDA(x): if*** *Parent(x) = nil* ***then*** *nil*
else if *(¬Default(Parent(x))* ***then*** *Parent(x)*
else *NNDA(Parent(x))*

The ***Nearest Non-Default Ancestor.*** Taking the example in Figure 6.10, NNDA(D) would yield B.

(72) ***NNDA_Spine(x): if*** *NNDA(x) = nil* ***then*** *nil*
else if *Parent(x) = NNDA(x)* ***then*** *Parent(x)*
else *NNDA_Spine(Parent(x)) + +(x)*

The sub-spine of µdialogues from and including *x* up to and including NNDA(x). Taking the example in Figure 6.10, NNDA_Spine(D) would yield (B,C,D).

The alphabet of events propagated by a µdialogue, x is:

(73) ***Alphabet(x): if*** $NNDA(x) = nil \vee$
$$\left(\bigcup_{d \in NNDA\_Thread(x)} Activators(d) \cap \bigcup_{d \in NNDA\_Thread(x)} Deactivators(d)\right) \neq \varnothing$$
then E_a
else $\bigcup_{d \in NNDA\_Thread(x)} Activators(d)$

A non-reachable μdialogue is one that:

i) has a non-reachable parent, or
ii) is not a default, and does not have any action event activators that are within the alphabet of events presented to it, and does not have any system event activators.

(74) ***NonReachable***(x):

$$NonReachable(Parent(x)) \vee$$
$$(\neg Default(x) \wedge (e \in Activators(x) \mid e \notin E_s \wedge e \notin Alphabet(x)))$$

Redundancy

It is possible to specify a system in DAL in which some arguments (activators or deactivators) and statements are superfluous since they reference action events that are unavailable to the μdialogue in question.

(75) ***Redundant***(x):

$$(e \in E_a \mid e \notin Alphabet(x) \wedge$$
$$(e \in Activators(x) \vee e \in Deactivators(x)$$
$$\vee e \in range(EventStatements(x)))$$

If an event exists that is unavailable to a μdialogue, but is referenced within the μdialogue, then those references are redundant.

Non-Termination

In the second stage of the execution cycle, system events are repeatedly broadcast and processed, and resulting system events are placed in the SEQ. This cycle repeats until the SEQ is empty. It follows that if one or more system events are always generated when the existing events in the SEQ are processed, then the cycle will never terminate. It is obviously important to be able to detect the possibility of non-termination.

This is an example of the Halting Problem [Turing36], and as a consequence no general solution is possible to determine if every case terminates. However there are situations when it is possible to show that the system terminates.

Each stage of the execution cycle can be modelled in terms of:

i) the sequence of events held in the SEQ, Q, and

ii) the state of the system, S.

By processing all the events Q, the system will reach a new state, S', and a new sequence of events, Q', will be left in the SEQ for the next execution cycle. We can define a history function μ, that for an initial sequence of events, Q_0, and system state, S_0, will return the event sequence, Q_n, and system state, S_n, after n execution cycles.

(76) $\mu(Q_0,S_0) = ((Q_0,S_0), (Q_1,S_1), (Q_2,S_2), \ldots (Q_n,Sn))$

If we assume that the behaviour of the system is deterministic, given values for Q and S will always produce the same resultant, Q' and S' . It follows that the system will not terminate if the pair (Q,S) repeats in the history sequence. That is:

(77) $(\exists n,m \mid m \neq n \wedge Q_n = Q_{(n+m)} \wedge S_n = S_{(n+m)})$

For a given system, the proof that this may occur could be a very difficult problem to solve. However, by simplifying the problem we can arrive at a method for determining whether there is a *possibility of non-termination*. If at least one system event is able to generate itself, either directly or indirectly, then the system will not terminate.

Consider a directed graph[5] with a node for each system event and a directed edge *a* -> *b* whenever system event *a* can (directly) generate system event *b*. If this graph has no cycles, then there can be no infinite directed paths through the graph. Every system event can be given a depth, which is the length of the longest directed path through this graph that ends at the event in question.

Now let *Q1* be the event queue after the action event is processed. *Q1* may contain system events of any depth. When all the events in *Q1* have been processed we get a queue, *Q2*, which contains events of depth 2 or more. When *Q2* has been processed we have an event queue containing events of depth 3 or more, and so on. Hence after m iterations, Q_m will contain events of depth m or

[5]This argument was suggest by Dr John Hudson.

more. If there are no cycles, there will be a maximum depth for the event graph. It follows that the maximum number of cycles possible is equal to this depth, and hence the system event processing must terminate. Hence, if we can show for a given system that there is an event cycle, then this is an indication that non-termination is possible.

Consider the system depicted in Figure 6.11. If this system receives event b, then the following event cycle will result:

$$(b \rightarrow \{a,c\} \rightarrow b \rightarrow \{a,c\} \rightarrow b \ldots)$$

This can be thought of as *b* giving rise to itself (indirectly) through the sequence *b* -> *c* -> *b*. In this example the event *a* also repeats, but this is incidental, the event *a* not giving rise to any other events.

```
A( )( ){
        B( )( ){
                a <- b;
                c <- b;
        }
        C( )( ){
                b <- c;
        }
}
```

Figure 6.11: An example of non-termination.

Consider the systems depicted in Figure 6.12. In the first of these (Figure 6.12(a)), provided that μdialogue *A* is able to activate and receives event *x*, then event *x* will continue to regenerate itself, and the system will clearly not terminate.

The second example (Figure 6.12(b)) is not quite so obvious. If μdialogue *A* activates, it will generate event *b*, that will cause *B* to activate. *B* will process the event *b*, and produce event *c*. This will then cause *C* to activate and process event *c* to produce event *b*. This sequence *(b,c,b,c,b,c,....)* will continue indefinitely.

```
A( )( ){
      x <- x;
}
```

(a)

```
A( )( ){
      EVENT b, c;
      GROUPING ME;
      ON ACTIVATION{ b; }

      B( b )( ){
            c <- b;
      }
      C( c )( ){
            b <- c;
      }
}
```

(b)

Figure 6.12: Possible non-terminating systems.

Figure 6.13 shows a system with an entirely equivalent behaviour to that shown in Figure 6.12(b), except the events that link the repeat cycle together are generated in response to ***on activation*** and ***on deactivation*** event statements in which originating events are absent.

```
A( )( ){
      EVENT c;
      GROUPING ME;
      B( b )( ){
            ON ACTIVATION{ c; }
      }
      C( c )( c ){
            ON DEACTIVATION{ b; }
      }
}
```

Figure 6.13: Event cycle linked by state changes.

Anonymous transitions are also able to take part in non-terminating event cycles. Consider the system depicted in Figure 6.14. Activating μdialogue *Z* gives rise to event *b*, which causes *Z* to deactivate. This in turn causes *Y* to activate as the default within a ME set. *Y* then gives rise to event *a*, which activates *Z* and continues the cycle.

```
X( )( ){
      GROUPING ME;
      Y( )( ){
            ON ACTIVATION{ a; }
      }
      Z( a )( b ){
            ON ACTIVATION{ b; }
      }
}
```

Figure 6.14: Event cycle linked by an anonymous transition.

Central to this analysis is the recognition of how one event can give rise to another. Anonymous transitions can cause the generation of new events far from the initial source of the transition. Consider the system depicted in Figure 6.15. If the initial state of the system is *{X,B,C,E}*, event *a* would cause µdialogue *A* to activate and *B* and its associated subtree to deactivate, the final state of the system being *{X,A}*. This would result in the generation of event z as a result of the deactivation of *E*.

```
X( x )( y ){
        GROUPING ME;
        A( a )( ){ }
        B( b )( ){
                GROUPING ME;
                D( d )( ){ }
                C( c )( ){
                        E( e )( g ){
                                ON DEACTIVATION{ z; }
                        }
                }
        }
}
```

Figure 6.15: Event generation in response to a deactivation caused by an anonymous transition.

Now consider the system in Figure 6.16. If the initial state of the system is *{X,A}*, event *y* will cause µdialogue *A* to deactivate. This in turn will result in *B* activating, along with its default child, *C*. The events *b* and *c* would result from this transition.

```
X( x )( ){
        GROUPING ME;
        A( a )( y ){ }
        B( )( ){
                ON ACTIVATION{ b; }
                C( )( ){
                        ON ACTIVATION{ c; }
                }
        }
}
```

Figure 6.16: Event generation in response to a activations caused by an anonymous transition.

From these observations we can identify five distinct ways in which one system event can give rise to another:

i) Explicitly, as an originating event in any event statement.

ii) As an activator of a µdialogue with one or more *on activation* event statements in which an originating event is absent. If a µdialogue is a default, then an activator of its nearest non-default ancestor can trigger an *on activation* event statement.

iii) As a deactivator of a µdialogue with one or more *on deactivation* event statements in which an originating event is absent. Deactivation of any ancestors of a µdialogue, *d*, will result in the deactivation of *d*, and produce the same result.

iv) As an activator of a µdialogue, *d*, in a mutually exclusive group. Activation of the µdialogue is then able to cause the anonymous deactivation of any siblings of *d*, and their associated subtree. If any of the deactivating µdialogues contain *on deactivation* event statements in which an originating event is absent, then the resultant event of such statements will be generated. The system depicted in Figure 6.15 would behave in this way.

v) As the deactivator of a µdialogue, *d*, in a mutually exclusive group. Deactivation of *d* would result in the anonymous activation of a default sibling of *d*, and any default descendants. If any of these default µdialogues contain *on activation* event statements in which an originating event is absent, then the resultant event of such statements will be generated. The system depicted in Figure 6.16 would behave in this way.

It is useful to define the following functions in order to support the subsequent analysis:

The function ***EffectiveActivators*** which for a given µdialogue, *d*, returns the set of events that can cause *d* to activate either directly (as one of its activators) or indirectly (as an activator of its NNDA).

(78) ***EffectiveActivators*** *(d):*

```
if Default(d) then
        if NNDA(d) ≠ nil then
                Activators(NNDA(d))
        else
                ∅
        endif
else
        Activators(d)
endif
```

The function ***EffectiveDeactivators*** which for a given μdialogue, *d*, returns the set of events that can cause *d* to deactivate either directly (as one of its deactivators) or indirectly by deactivation of any of its ancestors.

(79) ***EffectiveDeactivators****(d):*

$$\bigcup_{x \in Range(Spine(d))} \{e \in E_s \mid e \in Deactivators(x)\}$$

A function ***ME_Ancestors***, which for a given μdialogue, *d*, returns the set of mutually exclusive μdialogues that are ancestors of *d*. For the system depicted in Figure 6.15, ME_Ancestors(E) would return {B,X}.

(80) ***ME_Ancestors****(d):*

$$\{x \in Range(Spine(d)) \mid x \neq d \wedge MutuallyExclusive(x)\}$$

The function ***ME_Siblings***, that returns for a given μdialogue, *d*, the set of μdialogues that could cause *d* to anonymously deactivate. For the system depicted in Figure 6.15, ME_Siblings(E) would return {A,D}.

(81) ***ME_Siblings****(d):*

$$\bigcup_{x \in ME\_Ancestors(d)} \{y \in Children(x) \mid y \notin Range(Spine(d))\}$$

The function ***ME_SiblingActivators*** which returns for a given μdialogue, *d*, the set of events that can cause activation of any of the μdialogues that would result in the anonymous deactivation of *d*. For the system depicted in Figure 6.15, ME_SiblingActivators(E) would return {a,d}.

(82) ***ME_SiblingActivators****(d):*

$$\bigcup_{x \in ME\_Siblings(d)} \{e \in E_s \mid e \in Activators(x)\}$$

The function ***NNDA_Siblings*** which for a given μdialogue, *d*, returns a set of μdialogues, the deactivation of which would result in the anonymous activation of *d*. For the system depicted in Figure 6.16, NNDA_Siblings(C) would return {A}.

(83) ***NNDA_Siblings***(d):
if $\neg Default(d) \vee NNDA(d) = nil$ ***then***
$\emptyset$
else
$$\bigcup_{x \in Children(NNDA(d))} \{y \in x \mid y \notin Range(Spine(d))\}$$
endif

The function ***NNDA_SiblingDeactivators***, which returns for a given μdialogue, *d*, the set of events that would deactivate a μdialogue in NNDA_Siblings(d), and hence cause *d* to anonymously deactivate. For the system depicted in Figure 6.16, NNDA_SiblingDeactivators(C) would return {y}.

(84) ***ME_SiblingDeactivators***(d):
$$\bigcup_{x \in NNDA\_Siblings(d)} \{e \in E_s \mid e \in Deactivators(x)\}$$

In order for an event to regenerate itself, it must either do so directly, or indirectly by generating some other system event that will in turn eventually regenerate the initial event. Hence only those events that can give rise to further system events via one of the above mechanisms can possibly give rise to non-termination. We can define a function ***GeneratedEventPairs*** that contains the pairs of originating and resultant system events for a system:

(85)
$$GeneratedEventPairs = \bigcup_{e_o.e_r \in E_s,\ d \in \mu D} \{(e_o.e_r) \mid$$
$$(s_e \in EventStatements(d) \mid (e_o = s_e.e_o \wedge e_r = s_e.e_r)) \vee$$
$$(s_a \in OnActEventStatements(d) \mid$$
$$s_a.e_o = nil \wedge e_r = s_e.e_r \wedge$$
$$(e_o \in EffectiveActivators(d) \vee e_o \in NNDA\_SiblingDeactivators(d))) \vee$$
$$(s_d \in OnDeactEventStatements(d) \mid$$
$$s_d.e_o = nil \wedge e_r = s_d.e_r \wedge$$
$$(e_o \in EffectiveDeactivators(d) \vee e_o \in ME\_SiblingActivators(d)))$$
$$\}$$

For the system in Figure 6.13, this function would contain {(b,c),(c,b)}. We need to determine if the graph represented by this function contains any cycles, since such cycles represent an event regenerating itself. We can define a set ***Cycles***, containing all system events that can regenerated themselves:

Cycles:

(86) $\{e \in domain(GeneratedEventPairs) \mid$
$(\exists x \in T^*(GeneratedEventPairs) \mid x.e_o = e \wedge x.e_o = x.e_r)\}$

where $T^*(x)$ is the transitive closure of x.

It is only possible to be sure that a system terminates if there are no cycles:

(87) $(Cycles = \varnothing) \Rightarrow Termination$

6.8. Scope Of This Analysis

In this analysis, the behaviour of a system composed purely of μdialogues has been considered. The presence and possible effects of other associated event handlers such as widgets and API's has not been considered. Widgets do not create events, but are purely destinations for system events. Hence they are purely passive entities as far as the system is concerned, unable to cause any problems in terms of the overall system behaviour. API's on the other hand are able to both receive and generate events and could play an active role in problematic system behaviour. For example, an API could form a link in a chain of event pairs giving rise to non-termination of the system.

The list of events consumed and produced by an API is not defined by DAL, but by the code of the corresponding event handler. Although such information could be incorporated into the DAL specification for analysis purposes, there would need to be facilities within the development environment to force a correspondence between the model as defined by DAL and the implementation. Without such a facility to coerce the user interface designer in this way, there would be little point in utilising the additional information in system analysis.

Such issues are considered outside the scope of this thesis since they have little bearing on the question of whether user centred constructional modelling is a practical approach to user interface development. However, a commercial implementation of a user interface development environment based on DAL would need to consider this issue further.

Dynamic creation and deletion of µdialogues has not been considered within this analysis. However, possible µdialogue structures resulting from execution of *new* or *delete* statements could be analysed in precisely the way outlined above, and potential problems identified.

Chapter 7
The PIPS Development Environment

7.1. INTRODUCTION

In order for DAL to be a useful constructional modelling approach it is necessary to be able to automatically generate a working prototype of the interface from a specification written in DAL. In this chapter possible approaches to accomplishing this are examined and the method used in PIPS[1] [Anderson94], an interface development environment that uses DAL is described.

It should be pointed out that PIPS is only a prototype user interface prototyping environment; implemented to prove the suitability of DAL for interface implementation. For PIPS to be regarded as a full user interface prototyping environment it would need to have a number of additional support tools, such as a graphical screen layout editor and graphical tree editor for DAL. Although desirable additions, it was felt that they were unnecessary for proving the concepts, and were outside the scope of this research.

7.2. IMPLEMENTATION OPTIONS

The automatic implementation of a working prototype interface from a DAL specification could be accomplished in two distinct ways:

i) The specification could be compiled to a machine language equivalent system that could be executed directly by the hardware.

ii) The specification could be interpreted, either directly or after translation to some suitable intermediate form.

---

[1] PIPS stands for Paul's Interface Prototyping System.

The former suffers a major problem in that DAL is an inherently concurrent programming language. It is still the case that most computer systems are single processor, and any attempt to model concurrency on such an architecture has to use some form of time-slicing to simulate a multi-processor machine. A μdialogue is essentially a process. The number of processes in an interface is typically around 50-100. For complex interfaces this may increase to several hundred. Admittedly many of these processes will not be active at the same time, however the number that are could still be very high. This number of processes is greater than that usually supported in Unix and other multi-tasking operating systems. Synchronising the processes would also be difficult, particularly when considering the event (message) passing and shared data features of the language. The latter would prevent the use of a transputer based hardware platform, because of the need for shared memory . Even with multi-processor hardware, it is likely that the number of processes for many interfaces would be higher than the number of processors. In this circumstance some form of process scheduling would be necessary and context switching may become a significant overhead. However, in time there is little doubt that multi-processor based systems will become the norm, and when that happens it may well make good sense to investigate the implementation of a DAL specification directly on concurrent hardware.

The second option is to interpret the specification either directly, or after translating to some intermediate form. There are basically two main approaches that fall into this category. The first is to translate a DAL specification into an equivalent production based event response system, and then to interpret that in a similar way to Sassafrass [Hill87]. This approach has been successfully used for the implementation of a menu prototyping system based on Lean Cuisine [Anderson90]. In this system a Lean Cuisine description of the menu system was translated into a set of event driven production rules in a language based on ERL [Hill87]. This was subsequently interpreted. This approach is capable of handling concurrency, but suffers from a number of problems.

i) The number of production rules can become very large, resulting in a substantial and possibly prohibitive overhead in marking and firing rules.

ii) If the dynamic creation and destruction of µdialogues is to be supported, then this could be difficult to implement using a production rule approach. The addition or deletion of a single µdialogue may affect a large number of rules, and would incur a substantial overhead in locating the rules that have been affected by any changes. The dynamics of a direct manipulation style of interface are very important, and anything that is likely to cause the interaction to periodically stop is obviously undesirable.

The second approach in this category is to map a DAL specification onto an equivalent network of agents, in which each µdialogue has a corresponding agent in the implementation. This latter approach was the one chosen for PIPS.

7.3. THE PIPS 'AGENT' MODEL.

As discussed in Chapter 3, the use of agents in user interface implementation is well established. An agent has the following characteristics:

i) It responds to external stimuli (events).
ii) It maintains an internal state.
iii) It can generate events that in turn interact with further agents.

A DAL µdialogue behaves in precisely this way, but has the following additional behaviour:

i) Its ability to respond to external events is constrained by the state of its parent.
ii) A µdialogue may have both local and external variables that it is able to access. Hence its state can be thought of as having both internal and external components, the former being entirely under the control of the µdialogue, the latter being controlled by a pre-defined group of µdialogues.

As discussed in Chapter 5, the DAL model suggests an architecture for the corresponding interface. Each µdialogue is conceptually connected (by events) to at most one presentation agent (widget), and one application interface agent (API). The latter may in turn invoke one of a number of possible application functions dependent on the event received from the µdialogue.

The use of an object oriented approach to constructing multi-agent architectures is now well established. For example MVC [Goldberg84] used Smalltalk, and PAC [Coutaz87] used C++. PIPS is implemented using a subset of C++ on a Macintosh, by means of the Think C® compiler .

7.4. THE PIPS ARCHITECTURE

The components of the PIPS UIDS are shown in figure 7.1. A user interface specification is defined in DAL using any convenient text editor. This source file is then compiled using the DAL translation program, DALTRAN[2]. The purpose of this application is to translate the human readable description contained in the DAL source, into a form more easily assimilated by the PIPS run time environment. The machine readable output from DALTRAN is called an Interface Definition File (IDF). Finally the user interface defined in the IDF is read in and interpreted by PIPS. Any widgets or application interfaces (API) referenced in the DAL source have to be developed in C++ and linked into the PIPS source. Instances of these routines are then created by PIPS as and when required. In the case of widgets that need icons, the icons can first be created using a suitable resource editor (for example ResEdit®), and these resources loaded by PIPS. As a run-time option, PIPS will generate trace files that record the activation and deactivation of dialogue agents within the system, and the lifetime of events passing through the system. These can then be evaluated by the interface designer, and may lead to the developer modifying the design.

[2]DALTRAN - DAL TRANslator.

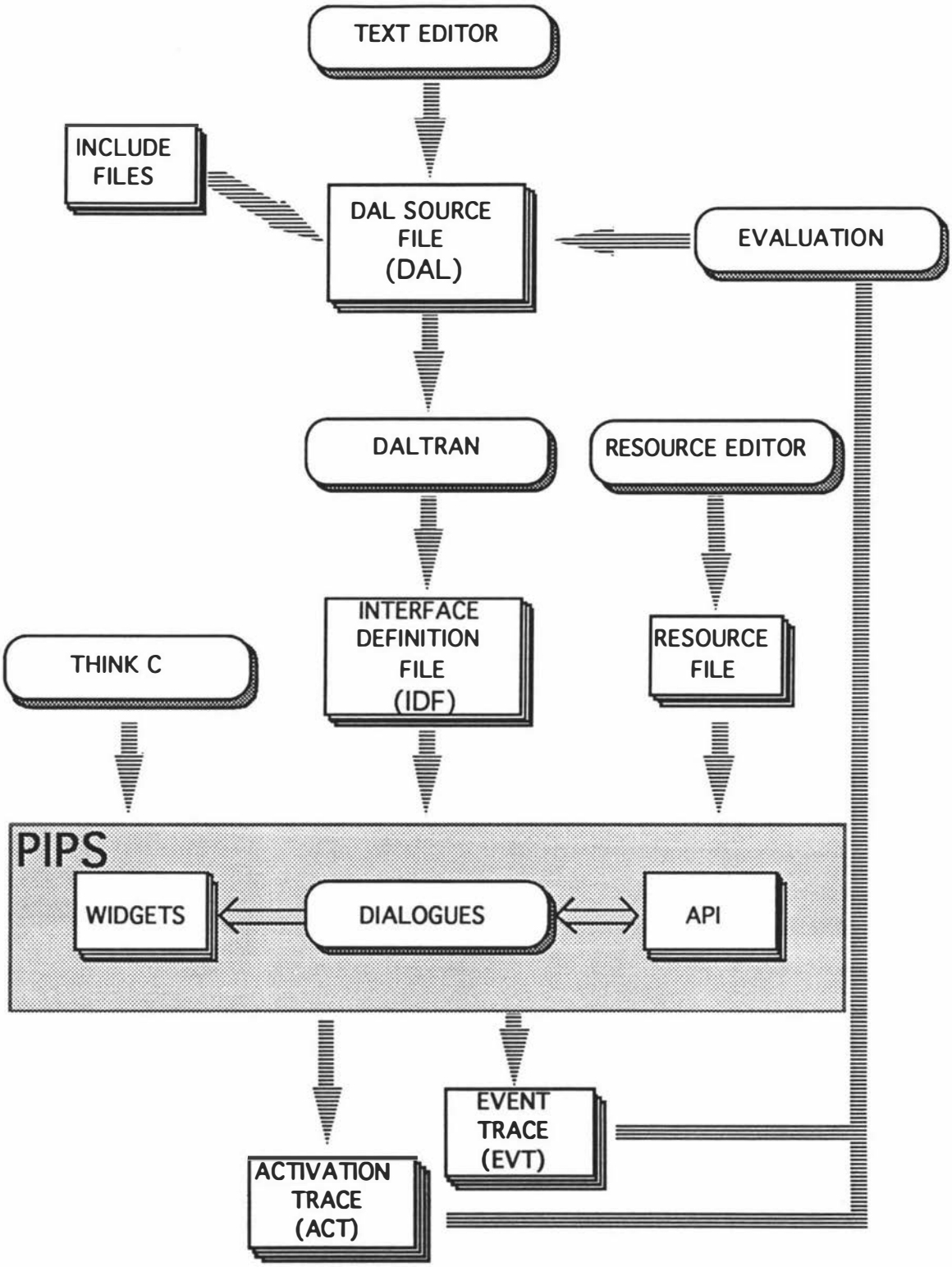


Figure 7.1: The PIPS Development Environment.

7.5. DAL COMPILATION

DAL source files are *compiled* by the DALTRAN program to a corresponding Interface Definition File (IDF) prior to being loaded into and translated by PIPS. This compilation involves the following:

i) The syntax of the DAL source file is checked.
ii) The system is examined for possible event cycles leading to non-termination in accordance with the analysis described in Chapter 6[3].
iii) DAL classes are fully expanded.
iv) Dialogues referencing class definitions are fully expanded.
v) Numeric identifiers are assigned to system events and variables.
vi) Statements are tokenised.
vii) Arithmetic expressions are converted to postfix format.
viii) The resulting translation is copied to a file with the same name as the source file, but with the extension '.idf'.
ix) A listing file is produced with the same name as the source file, but with the extension '.lis'. This file contains the results of the analyses and a symbol table in which event and variable identifiers assigned by DALTRAN are listed. Sample listings are given in Appendix C.

7.5.1. CLASS EXPANSION

DAL classes are fully expanded so as to minimise the overhead of generating new instances of derived classes at run time. The composition, C, of two classes, A and B, in which B is derived from A is obtained in the following way:

1. The guard condition of C is that of the most locally defined class, in this case B.
2. The activator list for C is the concatenation of the activators for A and B. The same applies to deactivators.
3. Statement lists, whether they be in the main body of the dialogue, or in an *on activation, on deactivation, on new* or *on delete* block are obtained by appending the most locally defined list onto the other.

Consider the two classes defined in Figure 7.2. The expansion of class B is as shown in Figure 7.3.

---

[3]Currently, DALTRAN only tests for a subset of the possible cases that can cause event loops as identified in Chapter 6.

```
CLASS

   A[ GuardA ]( a )( c ){
         ON ACTIVATION{ DoOnActA; }
         ON DEACTIVATION{ DoOnDeactA; }
         ON NEW{ DoNewA; }
         DoBodyA;
   }
   B:A[ GuardB ]( b )( d ){
         ON ACTIVATION{ DoOnActB; }
         ON DEACTIVATION{ DoOnDeactB; }
         ON DELETE{ DoDelB; }
         DoBodyB;
   }
```

Figure 7.2: Class expansion.

```
B[ GuardB ]( a, b )( c, d ){
      ON ACTIVATION{ DoOnActA, DoOnActB; }
      ON DEACTIVATION{ DoOnDeactA, DoOnDeactB; }
      ON NEW{ DoNewA; }
      ON DELETE{ DoDelB; }
      DoBodyA;
      DoBodyB;
}
```

Figure 7.3: Expansion of class B in figure 7.2.

In developing the method of this class expansion, the main question that had to be addressed was how to handle guard conditions. There were two possible approaches:

i) Take the most locally defined condition.
ii) Generate a new guard, a logical composition of the other two.

The first of these two options was taken since it was felt that the logical composition of guards could become difficult for developers to visualise, especially in the case of deep class hierarchies.

Class definitions can contain ***display*** and ***application*** statements. The approach taken in constructing the composition of two classes containing these statement types is to take the most locally defined statement. Consider the two classes shown in figure 7.4(a), their expansion is as shown in figure 7.4(b).

```
CLASS

  SimpleButton( []Mv )( ){
    EVENT Hilite, Dehilite;
    DISPLAY Icon,
      resource BoxIcon,
      size 10 10;
    ON ACTIVATION{ Hilite; }
    ON DEACTIVATION{ Dehilite;}
  }

  Button2:SimpleButton( )( ){
    EVENT Select;
    DISPLAY Icon2,
      resource Pic1, Pic2,
      size 15 20;
    APPLICATION DoButton;
    ON ACTIVATION{ Select; }
  }
```

(a)

```
CLASS

  Button2( []Mv )( ){
    EVENT Hilite, Dehilite,
          Select;
    DISPLAY Icon2,
      resource Pic1, Pic2,
      size 15 20;
    APPLICATION DoButton;
    ON ACTIVATION{      Hilite;
                        Select; }
    ON DEACTIVATION{ Dehilite; }
  }
```

(b)

Figure 7.4: A derived class containing display and application links (a) and the resulting expansion (b).

Dialogue instances can be based on a class definition but with additional behaviour included. The expansion of such dialogue definitions is carried out using exactly the same approach as that just described for class expansion.

7.5.2. VARIABLE AND EVENT IDENTIFIER ASSIGNMENT

Variables and system events are named entities within the DAL source. Within the PIP run-time environment they are referenced by suitable numeric identifiers (the event code for events), and unique instances of them are identified by a tuple of this identifier and the address of the declaring dialogue agent. The latter defines the scope of the entity (see Chapter 6). It follows that the translation of DAL to its run time equivalent must involve the assignment of suitable numeric identifiers to these entities.

Within a class definition, these entities do not map onto a specified instance of the variable or event. They are instance independent in this case. Within actual dialogue specifications the entities do map onto specific instances, however the address of the declaring dialogue agent is only apparent at run time, hence it can be left up to the PIPS dialogue instantiation methods to scope its variables and system events.

Because of this, the assignment of numeric identifiers to variables and system events within both classes and dialogues instances is trivial, simply involving the assignment of a unique integer identifier to each variable or event entity with the same name in the DAL source.

System events are used to control widgets and the application interface as well as to drive the interaction layer. The widgets and API events are hard coded into the system. Events that fall into this category are explicitly assigned values by the developer by using a **define** statement. For example, consider the system in figure 7.5.

```
DEFINE
   hilite       100,
   dehilite     101;

CLASS
   simple_icon( []Mv )( ){
         DISPLAY TYPE icon,
               size 10 10;
         ON ACTIVATION{ hilite; }
         ON DEACTIVATION{ dehilite; }
   }
```

Figure 7.5: Setting the event code for a display event.

In this example, the event codes for *hilite* and *dehilite* events will be 100 and 101 respectively. The convention followed for the assignment of event codes is:

Event Code Range	Event Category
0-99	Action Events - Predefined
100-199	Display Events. Hard coded into the corresponding widgets. Predefined in DAL sources using a DEFINE statement.
200-299	API events. Hard coded into the API. Predefined in DAL sources using a DEFINE statement.
300-32767	System Events not referenced by widgets or API's. Event code assigned at compilation time by DALTRAN.

The content of separate files can be included in a DAL source file by referencing them through an *include* statement. Hence, display and API event code definitions can be defined once within such a file, and then included into a source file as required.

A naming convention is followed for system events so that the type and hence sphere of influence of an event is clearly identified. The convention followed is that display events have a prefix if 'WE_' (Widget Event), and API events have the prefix 'API_'.

7.5.3. STATEMENT TRANSLATION

One of the main objectives of the DAL translation is to simplify the assimilation of the interface specification into PIPS. To achieve this, each dialogue component (whether it be within a dialogue class or a dialogue instance) is replaced by a single dialogue component record. For example, in the original DAL specification a µdialogue may have two activators, a and b. In the IDF representation these activators will be held in two separate dialogue component records starting with a token that clearly marks them as being activators, and an event specification following that gives the corresponding event code and context.

7.6. RUN-TIME COMPONENTS

The PIPS architecture is based around a small number of agents that carry out specific functions within the system. These agents fall into one of two groups:

i) Agents not specific to a particular interface. These will be referred to as *Framework Agents* since they constitute the main framework of the run-time kernel.

ii) Agents that are specific to a given interface in terms of their number, type and relationship to the rest of the system. These will be referred to as *Interface Agents* since they are the components that define an interface. Interface agents include dialogue, widget and API agents.

Figure 7.6 shows how these components are interrelated.

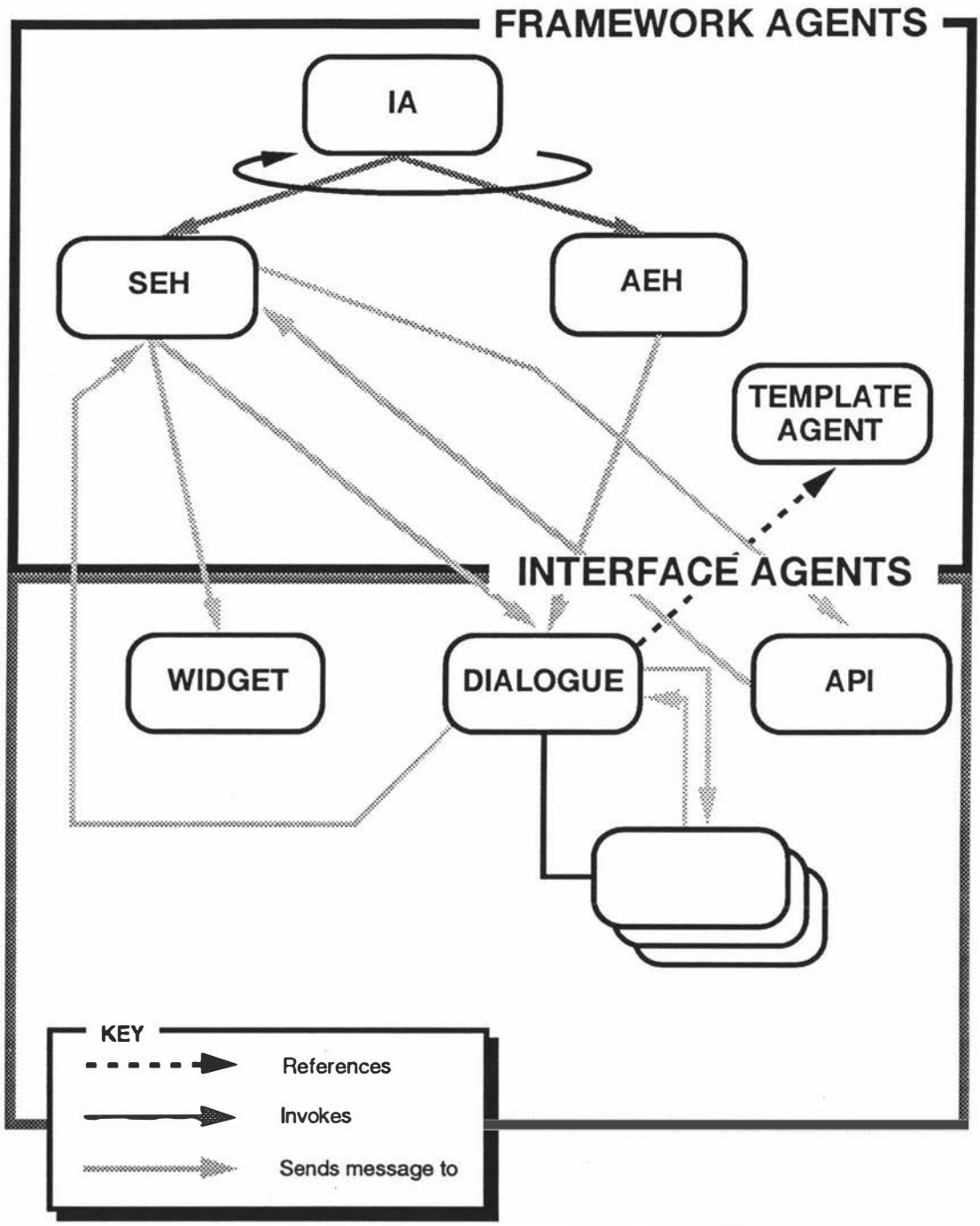


Figure 7.6: Run-time components of PIPS.

7.6.1. FRAMEWORK AGENTS

The following components fall into this category:

i) **Interface Agent** (IA). The purpose of this agent is to instantiate and link together all the agents within the user interface architecture. The IA is then responsible for carrying out the execution cycle.

ii) **Action Event Handler** (AEH). The purpose of this agent is to obtain DAL action events, and, when necessary, to buffer them in a queue for future processing.

iii) **System Event Handler** (SEH). The purpose of this agent is to receive and queue system events, and to subsequently broadcast them out to any receptor dialogue agents.

iv) **Template Agent**. The purpose of this agent is to maintain the class descriptions of an interface, such that they can be referenced by dialogue agents. As has been identified in the description of DAL, μdialogues often share the same behaviour. In addition, new dialogue agents are created by specifying that they will have a behaviour as defined in a given dialogue class. It follows that maintaining these class definitions separately at run time is both economical on memory and eases the implementation of the dynamic μdialogue creation.

There is only a single instance of each of these agents.

7.6.2. INTERFACE AGENTS

The following components fall into this category:

i) **Dialogue Agents**. These are the run-time embodiment of the μdialogues in the original DAL description.

ii) **Widgets**. These are the simple interface routines that behave as defined in the attributes statement of an interface specification. They are the DAL equivalent of a toolbox, although much simpler.

iii) **Application Interfaces** (API). These agents form the interface between run-time kernel and the application. They form the semantic link.

Many instances of these agents combined together to form a given interface. The dialogue agents form the basic framework to which the corresponding widgets and API are attached.

7.7. FRAMEWORK AGENTS

7.7.1. ACTION EVENT QUEUE

As seen in Chapter 5, action events in DAL fall into one of three categories:

i) Primitive events, eg; Mv.

ii) Primitive events in a specified context, eg; [ ]Mv.

iii) A change of curser context, eg; ~[ ].

In implementing a run time kernel for DAL on a Macintosh the events obtainable from the Macintosh toolbox event manager are mapped onto the corresponding DAL events. In the case of primitive events (i), there is a direct mapping from one to the other. In the case of primitive events in a defined context (ii), the mapping is the same, but the event requires further evaluation when presented to a µdialogue. The corresponding widget needs to assess if the event does in fact fall into its context.

In the case of the Macintosh, there is a small group of primitive DAL events that have to be derived from other events generated by the toolbox event manager. These are events associated with the depression or release of modifier keys (SHIFT, OPTION and COMMAND). In the case of these keys, separate toolbox events are not generated, but the state of these keys is recorded in a modifier mask associated with all other events. In order to generate these *extended* Macintosh events the Action Event Handler needs to maintain a simple finite state machine for each of these keys. A change in the state of a modifier flag between successive events causes a change in the corresponding FSM and the production of an appropriate DAL event.

The production of these extended Macintosh events can give rise to activation races. It is undefined as to whether they are produced before the event from which they were derived. It is also undefined as to the order in which such events are generated with respect to each other in the case where multiple extended Macintosh events are produced from a single toolbox event. Any circumstance that can cause non-deterministic event ordering can give rise to race conditions (Chapter 6). Although such races are possible, in practice they are unlikely and generally not serious.

Events in the final category (iii) are more difficult to obtain. All events produced by the Macintosh event manager contain the global coordinates of the curser at the time of the event. It follows that in order to generate a Change of Context (CoC) event, it is necessary to use this information from ***all*** events, so that the curser path can be traced. To establish a change of context the spatial context of every event to the specified widget must be established and the appropriate CoC event issued when the context is observed to change in the specified direction. Essentially the event coordinates are used to drive a finite state

machine (Figure 7.7), transitions in which cause the corresponding CoC events to be generated.

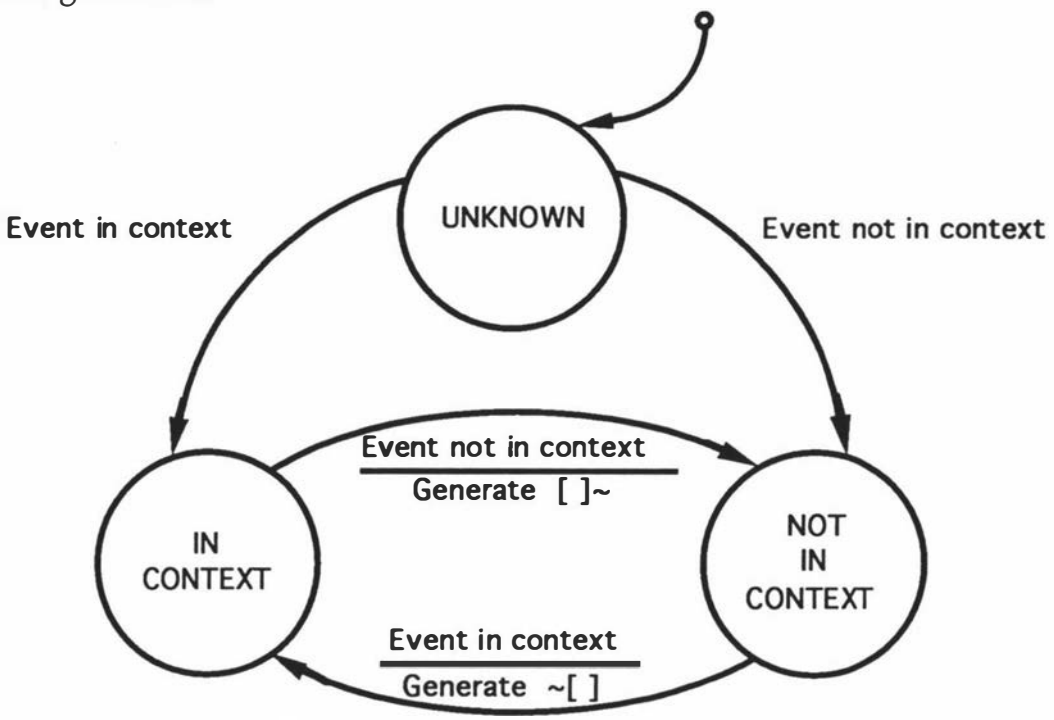


Figure 7.7: Generation of CoC events.

Since every primitive event has to be examined in order to generate CoC events, if the change of context was determined local to the μdialogue requiring the CoC, every event would need to be passed down the μdialogue, and the propagation and processing overhead would be considerable, and possibly unacceptable. In order to overcome this problem, CoC generation in PIPS is handled at the highest possible level. If a μdialogue is *interested* in a CoC event, then its interest is registered with the Action Event Handler (AEH). In registering a CoC, the type of CoC (move_into_context or move_out_of_context) and a pointer to the corresponding widget are recorded. A widget is a C++ object that has a Boolean 'Hit' method that for a specified global coordinate returns true if the coordinate falls inside the widget.

When a CoC event is initially registered the context of the cursor is unknown. As shown in figure 7.7, the approach taken in PIPS is to maintain three current context states, with the initial state being UNKNOWN. Only transitions between known context states cause CoC events to be generated. The corresponding handler is described in structured English in Figure 7.8.

```
FOREACH ( CoC interest record ) DO
    IF ( state is UNKNOWN ) THEN
        IF ( if event coordinates within widget ) THEN
            set state to IN_CONTEXT
        ELSE
            set state to NOT_IN_CONTEXT
        ENDIF
    ELSE                    ... we know the previous context ...
        IF ( interested in a MOVE_INTO_CONTEXT ) THEN
            IF ( state is NOT_IN_CONTEXT AND
                    event coordinates within widget ) THEN
                    set state to IN_CONTEXT
                    emit a MOVE_INTO_CONTEXT event
            ENDIF
        ELSE ... interested in a MOVE_OUT_OF_CONTEXT event ...
            IF ( state is IN_CONTEXT AND
                    event coordinates are not within widget ) THEN
                    set state to NOT_IN_CONTEXT
                    emit a MOVE_OUTOF_CONTEXT event
            ENDIF
        ENDIF
    ENDIF
ENDFOR
```

Figure 7.8: CoC Handler.

It is possible for a µdialogue to be interested in both a *move into context* and *move out of context* events. This is achieved by each event type being registered separately, and hence generated by the firing of a separate FSM.

Because a single primitive Macintosh event may give rise to several corresponding DAL events, this may in turn lead to non-determinism in the resultant system since the order in which these events are generated is undefined. This can cause activation races (see Chapter 6). This is a major implementation problem for DAL, although in practice it is perhaps not as severe as may first be imagined. In processing a single Macintosh event, the AEH can produce a **single** primitive event, and any number of CoC events. To get around the order problem the convention is followed that if a primitive event is issued, then this is done first, and only then are any CoC events issued. Any CoC events will be generated in a undefined order. Hence, a µdialogue hierarchy with CoC activators or deactivators ***may*** be non-deterministic. For example consider the system shown in figure 7.9.

```
A( ~[] }( ){
     B( ~[] )( ){
     }
}
```

Figure 7.9: CoC activation race.

In practice the need for such systems is rare and the spatial relationships of **A** and **B** may well be such that the actions of ~[**A**] and ~[**B**] are mutually exclusive, and cannot be generated in the same event cycle. Consider the widgets shown in figure 7.10. The curser has to be moved into the context of **A** in order to move into the context of **B**. Only after we have moved into the context of **B**, and that fact has been recorded by the AEH can a [**B**]~ event be generated. Hence ~[**A**] and ~[**B**] must be generated on separate event cycles. In this case the system is perfectly deterministic and there is no problem.

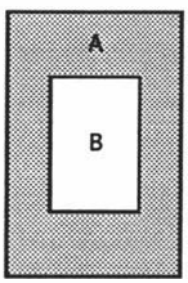


Figure 7.10: A spatial relationship between widgets A and B that would ensure that ~[A] would be generated before ~[B].

Only in the case of disjoint or spatially dynamic display object groupings can undefined CoC event sequences be generated.

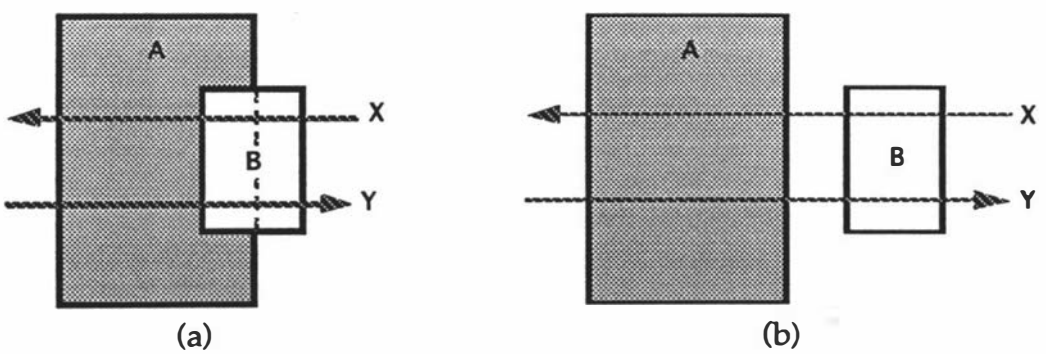


Figure 7.11: CoC non-determinism due to spatially disjoint object groupings.

Consider the widget groupings depicted in figure 7.11 (a) and (b). X and Y in these diagrams represent possible cursor tracks. In both (a) and (b), if the events

[A]~ and [B]~ were registered as being of interest, the order in which these events would be generated would be reversed in the case of track X compared with track Y. This may be of use to the designer, since it offers a means for determining the direction of a simple gesture command. However, if these events were used to activate a hierarchy of the type shown in figure 7.9 without regard for the fact that the outcome would be dependent on the direction of the users actions, then this would lead to a faulty design.

A potential problem with CoC events is that they are generated as a result of a spatial assessment of coordinates of the current primitive event at the time the CoC event is generated. Since this event may be queued for processing on some subsequent execution cycle, and that the corresponding widget may have been moved between the event being generated and its subsequent presentation to a μdialogue for processing, the CoC event could be out of synchronisation with the rest of the system. This is unlikely, but not impossible.

Another purpose of the AEH is to deal with display refreshes when windows become exposed. The widgets used by PIPS all make use of the normal Macintosh toolbox routines. When the content of a window needs updating due to the window being resized or a window in front of it being moved or resized, the toolbox event manager generates an update event that identifies the Macintosh window concerned. The AEH picks up this event in its normal event cycle and maintains a mapping of Macintosh window identifiers to the corresponding dialogues. On receiving an update event, the AEH tells the corresponding dialogue to update its display. The dialogue tree from this point down is then scanned, with each dialogue making its corresponding widget redraw itself. The recursion is terminated either at a terminal dialogue or at a dialogue with a widget which is itself a Macintosh window.

7.7.2. SYSTEM EVENT HANDLER

The System Event Handler (SEH) is responsible for:

i) Maintaining a queue (the SEQ) of system events waiting to be broadcast in the second stage of the execution cycle.
ii) Maintaining a broadcast table, identifying those agents that should receive a given event.
iii) Broadcasting events.

During the first stage of the execution cycle, system events are generated in response to the dialogue agents processing an action event. As they are generated they are queued on the SEQ. In the second stage of the cycle, the interface agent repeatedly requests the SEH agent to pop the top system event off its queue and to broadcast that event to any recipients of the event. These recipients can be dialogue, widget or API agents. Broadcasting a system event involves requesting each recipient in turn to process the event. In the case of dialogue and API agents this may give rise to further events that are then queued on the SEQ. This broadcast phase of the execution cycle repeats until the SEQ is empty. As discussed in Chapter 6, it is possible for this second stage of the execution cycle to enter a never ending loop. DALTRAN attempts to detect such cycles, but as a further safeguard PIPS assumes an upper limit to the number of broadcast cycles and terminates with a trace-back if this limit is exceeded.

In order for the SEH agent to broadcast events, it needs to know the identity of possible recipients of an event. This is achieved by widget, dialogue and API agents registering an interest in system events when they are instantiated. The SEH agent keeps a table with an entry for all system events (in terms of the address of the declaring dialogue agent and the event code) that relates them to a list of possible recipients of the event. In structured English, the broadcast method of the SEH agent is as given in figure 7.12.

```
METHOD Broadcast
        pop event off queue, making this the current event.
        find entry for event in recipient table.
        FOR ( each possible recipient ) DO
                request it to process the current event.
        ENDFOR
        delete the current event
```

Figure 7.12: SEH agent broadcast method.

A recipient dialogue will only process an event if either it is active, or its parent is active and the event is an activator for the dialogue. The list of recipients held by the SEH is ordered such that events are broadcast to dialogues in an order

depending on their position within the dialogue hierarchy. Dialogues at the top of the hierarchy will receive system events before any of their descendants. The order in which system events are presented to widget and API agents is determined by the position of their corresponding dialogue within the hierarchy.

7.7.3. TEMPLATE AGENT

The Template Agent maintains a table of dialogue class definitions that are referenced by dialogue agents. Classes are referenced by a unique integer, the class identifier. This is assigned by DALTRAN when compiling the DAL source. Class definitions are maintained by the class agent in exactly the same form as a dialogue agent. Hence to create a new dialogue agent, the template agent simply duplicates the corresponding class structure, and returns a pointer to it to the calling agent. Classes are referenced at run-time in two distinct situations:

i) By dialogues within the static interface description. If a dialogue is not based on a class, then a class id of 0 is specified within the IDF file. If a class is referenced, the dialogue is loaded by first obtaining a copy of the dialogue structure from the class agent, and then loading any additional behaviour directly from the IDF load channel. The rules for the addition of new statements, activators, widgets etc are the same as those used by DALTRAN for expanding derived classes.

ii) By new statements. When a new dialogue is instantiated, its behaviour is exactly as specified with the class definition. The parent dialogue agent requests a copy of the dialogue structure from the class agent. The widget size and location are set to those given in the new_field_spec of the new statement, and the resulting dialogue is linked into the parents child dialogue list.

Any variable or event references within a dialogue class are independent of any instance of that class. To uniquely identify such entities both their identifier and the nearest ancestor in which the entity is declared have to be specified. After a new dialogue agent has been created from a class definition, that agent must then scope any variables or system events. This is achieved by recursively scanning up the dialogue tree until a dialogue agent is found that declares the entity of interest.

7.7.4. INTERFACE AGENT

The interface agent is responsible for initialising the toolbox routines, instantiating the framework agents, initiating the preloading of interface agents and then controlling the execution cycle. The execution cycle is exactly as defined in Chapter 5.

7.8. INTERFACE AGENTS

7.8.1. DIALOGUE AGENTS

Dialogue Agents are the run-time instantiation of the μdialogues in the DAL specification. A dialogue agent interfaces with the following other agents (if they exist):

i) Its parent dialogue. There are four reasons for maintaining this link:
   a) So that it can inform the parent if it becomes deactivated. This ability is needed if the dialogue agent is in a ME group with a default member. In this case the default will be activated by the parent.
   b) So that it can scope its system events and variables by scanning up through the dialogue tree until a dialogue agent is found that declares the entity in question.
   c) So that it can reference shared variables. These are located within the dialogue agent that declares them. A dialogue agent has methods of GetVal(id), and SetVal(id, value) to control access to these shared variables.
   d) So that if the dialogue agent is deleted, the parent dialogue can be informed.

ii) Its children. Action events are propagated by dialogue agents sending the events to their children. A dialogue agent must maintain information about the state of each of its children, and be able to enquire as to whether or not the current event is an activator for any of its children.

iii) Its associated widget. In evaluating events with a specified context the dialogue must be able to find out if the coordinates in the current event fall within its widget. The widget has a method 'Hit' that returns the

corresponding Boolean response. Although conceptually a dialogue agent can send system events to its widget, this is in fact done indirectly via the SEQ. The widget will register an interest in a given event, the dialogue agent will send that event to the SEH which subsequently broadcasts the event to every agent with an interest in it. If the dialogue agent is deleted, then it informs its widget to delete itself.

iv) Its associated API. As in the case of widgets, if the dialogue agent is deleted, it tells an associated API to delete itself.

Dialogues handle events exactly as defined in Chapters 5 and 6. That is action events are processed and propagated down the tree and system events are simply processed. The difference in behaviour between the two execution cycles is handled by separate processing methods, *Run1* for the first execution cycle and *Run2* for the second.

Each µdialogue instance has a text string identifier that can be used when tracing the execution of an interface. In the case of µdialogue instances that have been statically created (ie, they are declared in the specification), they have the name given within the specification. In the case of dynamically created µdialogues, a unique name is generated by concatenating the name of the class from which the µdialogue has been created with an integer that is incremented for each new µdialogue instance.

7.8.2. WIDGETS

A widget is an agent responsible for creating the appearance of the user interface. A widget has the following characteristics:

i) A finite, pre-defined set of properties (attributes). Examples would be size, location, name, and hilighted.
ii) A state as defined by the current value of its properties.
iii) An appearance, which in turn is dependent on its state.
iv) A pre-defined set of events to which it will respond.

A widget is only able to receive events. Processing a received event will the values of one or more properties to change which is then reflected in a change

in its rendition. As soon as a property is changed then the widget is re-drawn to immediately reflect the change in appearance.

The properties of a widget may be preset to appropriate values within the display statement associated with a μdialogue as described in Chapter 5.

A widget is implemented as a C++ class within PIPS. A widget, by convention has the name DCxxx, where DC stands for Display Class, and xxx is any text appropriate for the particular widget. All widget classes are derived from the class CWidget or from another display class (hence allowing incremental refinement of widget classes). Associated with a widget are five key methods:

- *DoSetAttr* - that takes the name of an attribute and one or more values, and sets the value of the attribute accordingly. This is used by the IA to preset widget attributes.
- *SetSEInterest* - that registers the widget with the SEH as being interested in the system events to which it will respond.
- *Hit* - a Boolean function that is to return true if the specified global coordinates fall within the confines of the widget.
- *Draw* - that causes the widget to draw itself in accordance with its current state.
- *Run2* - that is responsible to processing system events in the second half of the execution cycle. Calling this method causes the current event to be referenced. If the current event is among those to which the widget will respond, an appropriate method will be invoked to update the properties that are to be altered in response to such an event.

A number of methods may be defined for updating the values of properties.

A widget will only respond to a small number of pre-defined events. These events are defined in a static integer array, *EventList*. Although, in theory, a widget could receive, and hence respond to any system event, if a given system event was able to interact with widgets, dialogues and APIs, then this could cause some confusion for the developer. Because of this it was decided to keep widget events entirely separate from other system events. The event codes for widget events are defined in the file "WidgetEvents.h". This file is the C

equivalent of the "WidgetEvents.dal" file referenced within the DAL source. The latter can be automatically created from the former using the utility *CtoDAL*.

The properties of a given widget are referenced by a string, and are defined in a local string array *AttribList*. This is referenced by the method *DoSetAttr* that takes a string referencing a given property within the original DAL source (and simply copied over to the corresponding IDF file), and causes the given property to be set according to the arguments given. The number and type of the arguments is dependent on the type of the property. Checking that the values given are of the right type and number is checked by DALTRAN using the information contained within an attribute statement.

7.8.3. APPLICATION INTERFACE

An Application Interface Agent (API) forms the boundary between the user interface and the underlying application. An API is able to receive events, which in turn cause corresponding methods associated with the API to be invoked. The method will be an application function (APF), and hence contains the semantics associated with the event. An API needs to be able to register an interest in events that it is able to consume in the same way as a widget, hence it also maintains a EventList. However, an API differs from a widget in that it is able to produce as well as consume events. A list of events that it is able to produce is maintained in a list called *EmitList*. This is necessary so that the events can be scoped with respect to the dialogue hierarchy.

Values contained in the event fields of an event that it is consuming can be read by the APF, allowing additional information to be transferred if needed. However, the type and number of such values is severely limited, there being only 4 data fields in an event record, and that the value of each is just an integer. In order to extend this, a pointer array is maintained by PIPS, and the fields in an event record can be used to pass indexes into the array to allow indirect access to much larger data structures. In this way, text for example can readily and efficiently be passed around the system.

7.9. OPTIMISATION OF THE PROTOTYPE SYSTEM

Direct manipulation interfaces can be thought of as a form of real-time system. The definition of the latter is a system that must respond to externally-generated input stimuli within a finite and specified period [Burns90]. It has

been observed that DMUI need to produce feedback at a rate equal to or faster than the human visual processor cycle time of around 100msec [Bass88]. If this is not achieved, then the user perceives the system as not being synchronised with their actions, and they are left clearly aware of the interface between them and the system with which they are conversing.

It follows that in order for the system to achieve these temporal constraints, optimisation may be an important consideration. Examination of the DAL architectural model shows a major optimisation that can be readily implemented. The basis of this optimisation is that a µdialogue is only *interested* in events if:

i) its parent is active and it is currently not active. In this case the µdialogue is interested in possible activating events.

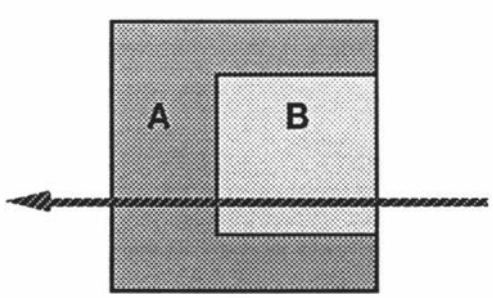
ii) it is active. In this case the µdialogue is interested in any events that it is able to process, whether this be as an originating event in an event statement or a deactivator.

Based on this observation it is possible to reduce the number of events submitted to the system and improve its response considerably. In order to implement this potential performance improvement the primary event sources have to have the functionality to both add and remove an event from an interest list. Only events in this interest list would subsequently be generated. Within the DAL architecture, there are two primary event sources, namely the action event handler (AEH) and the system event handler (SEH). However, from the perspective of the µdialogues this is unimportant. They are unaware of the origin of an event. In the following presentation of this scheme there are two key functions for registering and deregistering a set of events *E*.

```
Register((E):
        for e in E do
                if (e ∈ E_a) then
                        register an interest in e with the AEH
                else
                        register an interest in e with the SEH
                endif
        endfor

DeRegister(E):
        for e in E do
                if (e ∈ E_a) then
                        deregister an interest in e with the AEH
                else
                        deregister an interest in e with the SEH
                endif
        endfor
```

On the face of it, it would be sufficient for a μdialogue to register an interest in all of its event statement originating events and its deactivators when it is activated, and to request all of its children to register their activators. Conversely on deactivating the same events could be deregistered. However, as already seen, change of context (CoC) events complicate this simple view. Consider the system depicted in Figure 7.13.

```
SYSTEM
  ...
  A( ~[] )( ){
     B( ~[] )( ){
     }
  ]
  ...
```



(a) (b)

Figure 7.13: A system with potential optimisation problems. Its DAL specification (a) and appearance (b). The arrow represents the path of the cursor motion.

In this system two μdialogues *A* and *B* are both activated by dragging into their contexts. *A* is the parent of *B*. Consider the situation in which the cursor is dragged across the corresponding widgets as depicted by the arrow on the diagram. If the simple optimisation scheme described above was followed, then the following responses would result:

i) *A* would activate.

ii) *B* would register in interest in the (CoC) event ~[].

However, since the widget corresponding to *A* and *B* share their right hand boundaries the event ~[B] would never be generated, since at the point that the event would registered as being of interest the cursor would already be in the context of *B*. By contrast, if the cursor was dragged across the widgets from left to right this problem would not occur.

Such non-determinism is obviously unacceptable. A solution to this problem is to treat CoC events differently from other events, and to register an interest in all such events when:

i) the system initially starts up, and

ii) when a new μdialogue is created which is interested in CoC events.

Applying this approach, the control of event generation within the system can be defined in terms of three functions:

i) ***Initialise***, called as part of the system startup procedure.

ii) ***Activate***, called by a μdialogue whenever it activates.

iii) ***Deactivate***, called by a μdialogue whenever it deactivates.

Initialise:

for *x* ***in*** *Subtree(Root)* ***do***

Register $\{e \in Activators(x) \mid IsCoC(e)\}$

Register $\{e \in Deactivators(x) \mid IsCoC(e)\}$

Register $\{s.e_o \mid s \text{ in } EventStatements(x) \wedge IsCoC(s.e_o)\}$

endfor

if *Default(Root)* ***then***

Activate(Root)

else

Register $\{e \in Activators(Root) \mid not\ IsCoC(e)\}$

endif

Activate(x):

Register $\{e \in Deactivators(x) \mid not\ IsCoC(e)\}$
Register $\{s.e_o \mid s\ in\ EventStatements(x) \wedge not\ IsCoC(s.e_o)\}$
for *c* ***in*** *Children(x)* ***do***
Register $\{e \in Activators(c) \mid not\ IsCoC(e)\}$
endfor

DeActivate(x):

DeRegister $\{e \in Deactivators(x) \mid not\ IsCoC(e)\}$
DeRegister $\{s.e_o \mid s\ in\ EventStatements(x) \wedge not\ IsCoC(s.e_o)\}$
for *c* ***in*** *Children(x)* ***do***
DeRegister $\{e \in Activators(c) \mid not\ IsCoC(e)\}$
endfor

In this definition, the Boolean function *IsCoC* returns *true* if the argument is a CoC event:

IsCoC(e): $(e = MoveIntoContext \vee e = MoveOutofContext)$

7.10. PIPS DEBUGGING FACILITIES

PIPS supports two types of run time tracing accessible through the command line interface. These are ***activation*** and ***event record*** tracing.

7.10.1. ACTIVATION TRACING

Specifying the qualifier -t on the command line causes a trace file to be generated with the same name as the idf file, but with the extension '.trc'. With this option enabled, every time a dialogue agent activates or deactivates this information is recorded in the file in terms of a timestamp (in clock ticks since system startup) and the name of the dialogue. A prefix of A or D indicates if the record is for an activation or deactivation (this information is further shown by which column the name of the dialogue is placed). Figure 7.14 shows an example trace for the popup menu example, the listing of which is given in Appendix C. This method of monitoring the interface has proved to be very valuable since it shows the way that the user has navigated through the dialogue space, and the time it took them.

```
============ popup ============
A] 00219711 desktop
A] 00219711 monitor
A] 00219866 activate_popup
A] 00219866 undo_popup
D] 00219893                    undo_popup
A] 00219893 menu_window
A] 00219894 menu
A] 00219894 drag
A] 00219894 close_dialogues
A] 00219894 default
D] 00219961                    menu
D] 00219961                    default
A] 00219961 pinned
A] 00220080 menu
A] 00220080 Item_New
D] 00220084                    menu
D] 00220084                    Item_New
A] 00220205 menu
A] 00220206 Item_New
D] 00220225                    Item_New
A] 00220232 Item_Open
D] 00220251                    Item_Open
A] 00220267 Item_Open
D] 00220268                    Item_Open
A] 00220304 Item_Close
D] 00220336                    menu
D] 00220336                    Item_Close
A] 00220463 menu
A] 00220463 do_exit
```

Figure 7.14: An example activation trace.

7.10.2. EVENT RECORD TRACING.

Specifying the qualifier -e on the command line causes a trace file to be generated with the same name as the idf file, but with the extension '.evt'. When an event is generated in PIPS, it is assigned an event record, and a reference to this is passed around the system. When processing of an event is complete this event record is released. If event tracing is enabled, whenever an event record is released, its details are first recorded in the event trace file. The details recorded are:

- A timestamp; the number of clock ticks since system startup.
- The event code.
- The age of the event in milliseconds.
- The message field of the event. This contains the ASCII code for action events originating from the keyboard.
- The event fields Ex, Ey, Ea and Eb.

Figure 7.15 shows part of the event trace corresponding to the activation trace in Figure 7.14.

```
[00219722 0028 <000000>:00000:00123:00153:00123:00153]
[00219726 0028 <000000>:00000:00123:00152:00000:-0001]
[00219727 0028 <000000>:00000:00124:00151:00001:-0001]
[00219733 0028 <000000>:00000:00141:00126:00017:-0025]
[00219734 0028 <000000>:00000:00141:00125:00000:-0001]
[00219743 0028 <000000>:00000:00141:00126:00000:00001]
[00219750 0028 <000000>:00000:00142:00126:00001:00000]
[00219750 0028 <000000>:00000:00146:00124:00004:-0002]
[00219866 0019 <000000>:00000:00146:00124:00000:00000]
[00219866 0305 <000000>:00000:00000:00000:00000:00000]
[00219894 0001 <000001>:00000:00146:00124:00000:00000]
[00219895 0104 <000003>:00000:00146:00124:00000:00000]
[00219900 0100 <000011>:00000:00000:00000:00000:00000]
[00219900 0311 <000010>:00000:00000:00000:00000:00000]
[00219934 0028 <000000>:00000:00152:00130:00006:00006]
[00219935 0028 <000000>:00000:00153:00132:00001:00002]
[00219936 0028 <000000>:00000:00154:00133:00001:00001]
[00219937 0028 <000000>:00000:00156:00134:00002:00001]
[00219938 0028 <000000>:00000:00159:00137:00003:00003]
[00219939 0028 <000000>:00000:00160:00138:00001:00001]
[00219940 0028 <000000>:00000:00161:00139:00001:00001]
```

Figure 7.15: A sample event trace.

This information is valuable for two reasons. First it shows which events have been generated, which may be useful to the interface developer when the system appears to behave differently to what he was expecting. Secondly, the duration that the event was in the system shows how long it took to process the event. Bearing in mind that the temporal constraints on DMUI are important, this timing information is particularly useful.

Because the activation and event traces are both time-stamped, it is possible to relate the two together. Since activation tracing shows what the user is able to do, and what the user actually does, and the event trace shows how quickly the system responded to the users requests. Comparing the two may be of considerable use in analysing the effect of degraded system performance on user actions.

Chapter 8
Design Examples

8.1. INTRODUCTION

In this Chapter a number of different dialogue design examples are examined. The dialogues concerned have been chosen to demonstrate the flexibility of DAL in being able to cope with both well established dialogues such as drag and drop as well as some entirely new styles of interaction.

8.2. DIALOGUE BOX

The use of dialogue boxes is well established in current user interface. The key feature of such dialogues is that they are modal. That is, once the dialogue box is presented to the user, the user is unable to interact with the rest of the interface until they have completed their interaction with the dialogue box and have terminated that dialogue by selecting an appropriate exit option. This moded interaction style contrasts markedly with the unrestricted multi-threaded interaction seen in most other dialogues of direct manipulation interfaces. DAL was developed primarily to support the latter, and that of course raises the question as to whether DAL is capable of supporting these modal dialogues as well.

The 'first cut' at designing such a dialogue is to place the dialogue box in a mutually exclusive grouping with respect to the rest of the system. This will have the effect that if the user starts to interact with a dialogue box, then dialogues in the rest of the system will be deactivated. The shortcoming of grouping is that there is nothing to stop the user resuming interaction with the rest of the interface at any point in time without having to exit the dialogue box via one of its exits. This problem can be readily overcome in the following way:

1. Define a event local to the dialogue box, say *doneDB*, and make this a deactivator for the dialogue box.

2. Make the μdialogues associated with the exits from the dialogue box as being the only sources of this *doneDB* event.

This approach is presented in Figure 8.1.

```
MyInterface( )( ){
   EVENT startDB;
   GROUPING ME;
   DialogueBox( startDB )( doneDB ){
         EVENT doneDB;

         Exit : Button( )( ){
               // generate the doneDB event
               // to terminate the dialogue
               // box interaction.
               doneDB <- [ ]M^;
         }
   }
   Others( )( ){
         // ... generate startDB event in here
         // to start up the dialogue box.
   }
}
```

Figure 8.1: Implementation of a dialogue box in DAL.

The effect of this is to prevent the dialogue box being deactivated by any other route other than selection of one of its exit options. Hence the user will only be able to interact with the dialogue box until they have exited the dialogue box in the specified manner, this being the desired behaviour. The implementation of the *brush shape dialogue* example presented in Appendix C uses this approach.

8.3. A PIN-UP, POP-UP MENU

The use of popup menus that can be pinned up on the screen for further use has become popularised by their use in the OpenLook® environment [Sun-Microsystems89]. Consider the problem of specifying this type of interaction, but running on a Macintosh with its restriction of a single button mouse. The behaviour of this menu is to be as follows:

- On depressing the shift key, followed by the mouse key, the menu will be displayed. Dragging the cursor through the options will cause them to highlight and dehighlight. Releasing the mouse key within the context of one of the items will cause that option to be selected. Releasing the mouse key within any context other than the push pin in the top left hand corner will cause the menu to disappear.

- Releasing the mouse key within the context of the push pin will cause the push pin to move into the 'in' position, and for the menu to remain up on the screen. Whilst in this state selections can be repeatedly made. The menu disappears only if the mouse key is released within the context of the push pin.

Using DAL, this menu can be defined as shown in figure 8.2.

```
CLASS
   menu_item( ~[] )( []~ ){
         EVENT hilite, dehilite, select;
         DISPLAY TYPE button;
         ON ACTIVATION { hilite; }
         ON DEACTIVATION { dehilite; }
   }

SYSTEM
   desktop( )( ){
         activate_popup( 'SHIFT'v )( 'SHIFT'^ ){
               popup <- Mv;
         }
         menu_window( popup )(  close ){
               ON ACTIVATION{ show_menu; }
               ON DEACTIVATION{ hide_menu; }

               close_dialogues( )( ){
                     GROUPING ME;
                     close_it( )( ){
                           close <- M^;
                     }
                     push_pin( []M^ )( ){
                           ON ACTIVATION{ pin_in; }
                           ON DEACTIVATION{ pin_out; }
                           close <- []Mv;
                     }
               }
               menu( )( ){
                     copy:menu_item( )( ){
                           do_copy <- []M^;
                     }
                     cut:menu_item( )( ){
                           do_cut <- []M^;
                     }
                     paste:menu_item( )( ){
                           do_paste <- []M^;
                     }
                     clear:menu_item( )( ){
                           do_clear <- []M^;
                     }
               }
         }
   }
```

Figure 8.2: A pinup, popup menu.

In this example the display information and the event declarations have been left out for brevity. Of particular note in this example is the way that we change the way that the *close* event is generated depending on the state of the push pin. The default close dialogue is close_it, since it has no activators and hence will activate in unison with the parent. Within this µdialogue, releasing the mouse key, no matter what the context, will result in a close event which will in turn cause the menu_window to deactivate. If however we release the mouse key within the context of the push pin, then the push pin µdialogue will activate, deactivating the default close µdialogue. From this point on the *close* event can be produced only by depressing the mouse key within the context of the push pin.

8.4. DRAG AND DROP

Drag and drop has become a well recognised paradigm in current DMUI. Its use was well established in the earlier versions of the Mac Finder® [Apple-Computer87] as a means of moving folders into folders and files into the trash can, and with the introduction of System 7 extended to cover the invocation of applications by dropping data files onto them [Apple-Computer91]. Similar uses for the paradigm have appeared in OpenLook® [Sun-Microsystems89].

The basic sequence involved in a typical drag and drop operation is outline using UAN in Figure 8.3. In this paradigm a target icon (icon1 in our example) is selected and dragged onto a destination icon (icon2). Releasing the mouse key within the context of this destination icon represents a drop of the target icon onto this destination. The semantic interpretation of this operation would be defined within the application.

ACTION	FEEDBACK
[icon1]Mv	icon1 highlights
(~[x,y])*	outline of icon1 moves
~[icon2]	icon2 highlights
[icon2]M^	icon1 "dropped on" icon2

Figure 8.3: Drag and drop

Figure 8.4 describes the dialogue in UAN, (a), and shows how this description can be translated into the equivalent DAL, (d) and (e). In this example the source object is denoted as S, the destination object as D. In the UAN

description, the dialogue is presented from the users viewpoint, both S and D being present in the dialogue description. However, in order to translate this into DAL it is necessary to extract two distinct dialogue descriptions from the UAN, one dialogue from the perspective of the source object (b), the other from the perspective of the destination object (c). Hence, as presented in Figure 8.4, the first step is to 'split the view' between source and destination objects, the dialogues still being specified in UAN.

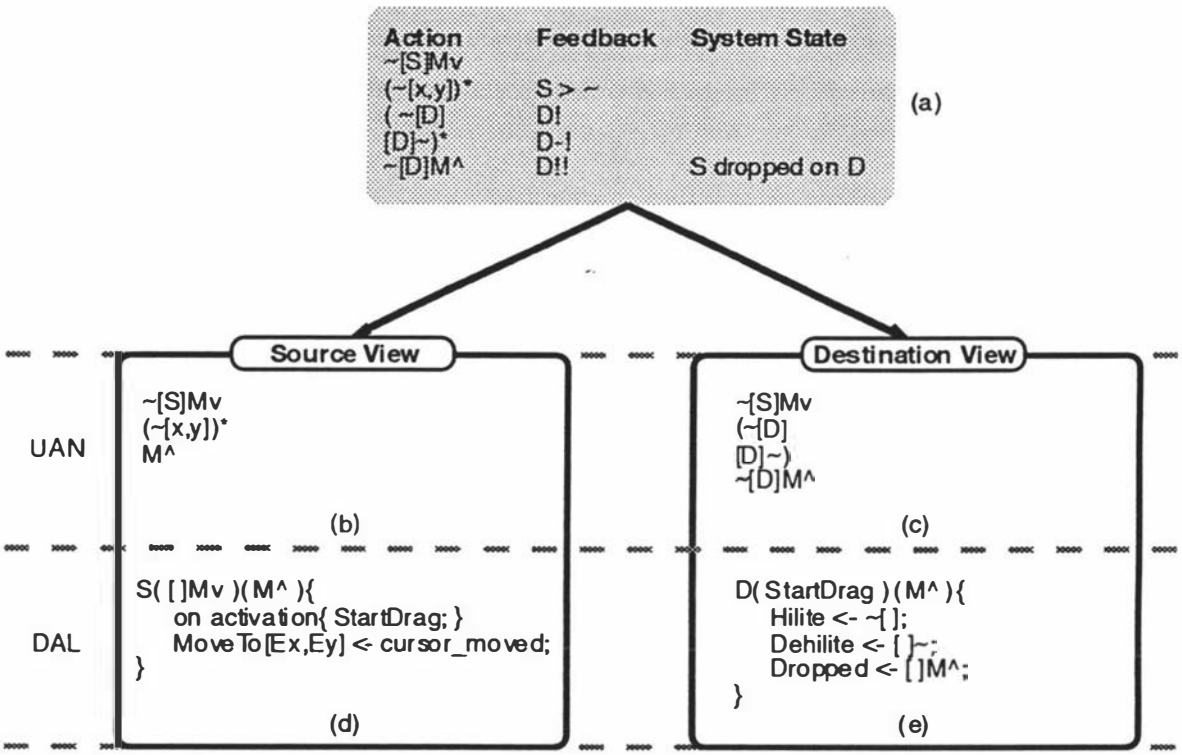


Figure 8.4: (a) A UAN description of a drop and drag dialogue, and its translation into source and destination views expressed in UAN, (b and c), and DAL (d and e).

A key point to note is that a user action within the context of the source occurs in the dialogue description for the destination. In translating this into the equivalent DAL it is necessary to translate this activating event into a dialogue between the two µdialogues. Hence on activating the S µdialogue, the event *StartDrag* is generated, which in turn activates the D µdialogue. In both cases, releasing the cursor, irrespective of the context, terminates the dialogue. The event *Dropped* is only generated if S is dropped on D. This event could be

accepted by either the application or another μdialogue, perhaps encapsulating the S μdialogue.

A DAL specification for FileIcon and Trash dialogue classes is depicted in Figure 8.5. In this example the event TrashIt will be passed by the FileIcon dialogue to its associated application function. This would be very much by way of a request, and the decision to accept or deny the request would lie entirely with the application.

```
CLASS

/*
 *    EnableDrag exported.
 *    A FileIcon is assumed to be contained within
 *    a ME group.
  */
      FileIcon( [ ]Mv )( M^ ){
            EVENT Hilite, Dehilite, MoveBy, TrashIt;
            ON ACTIVATION {
                  Hilite;
                  EnableDrag;
            }
            Dehilite <- M^;
            MoveBy[Ea,Eb] <- cursor_moved;
      }

      /*
       *    EnableDrag imported.
       *    DoTrash exported.
       */
      Trash( EnableDrag )( M^ ){
            EVENT Hilite, Dehilite;
            Hilite <- ~[ ];
            Dehilite <- [ ]~ | M^;
            DoTrash <- [ ]M^;
      }
```

Figure 8.5: A "drag and drop" trash dialogue specification.

Within the Macintosh Finder® it is possible to have many possible destinations for a file drop operation. Destinations could be the trash or one of many folder icons. Our previous example can easily be extended to coping with this as shown in Figure 8.6.

```
CLASS

  /*
   *     EnableDrag exported.
   *     A FileIcon is assumed to be contained within
   *     a ME group.
   */
  FileIcon( [ ]Mv )( M^ ){
        EVENT Hilite, Dehilite, MoveBy,
              TrashIt, MoveFile ;
        ON ACTIVATION {
              Hilite;
              EnableDrag;
        }
        Dehilite <- M^;
        MoveBy[Ea,Eb] <- cursor_moved;
  }

  /*
   *     EnableDrag imported.
   */
  DD_Destination( EnableDrag )( M^ ){
        EVENT Hilite, Dehilite;
        Hilite <- -[ ];
        Dehilite <- [ ]- | M^;
  }

  /*
   *     DoFileMove exported.
   */
  FolderIcon : DD_Destination( )( ){
        DoFileMove <- [ ]M^;
  }

  /*
   *     DoTrash exported.
   */
  Trash : DD_Destination( )( ){
        DoTrash <- [ ]M^;
  }
```

Figure 8.6: Multiple destination "drag and drop" interpretation.

The classes defined in Figure 8.6 suffer from the limitation that although icons representing application or data files (FileIcon) can be dragged and dropped, there is no provision for folder icons to be relocated or trashed. A folder icon has essentially two hats since it can be either a target for a "drag and drop" operation, or a destination. It can only "wear" one of these hats at once. It follows that we could have two mutually exclusive sub-dialogues associated with a folder icon corresponding to these different behaviours. Since the action that decides which of these two sub-dialogues should be in force is the initial `[]Mv`, we need to explicitly prevent the DD_Destination sub-dialogue for the target folder from activating. This is done in the example with a guard

condition using the variable *NotThisOne*. In Figure 8.7[1], this behaviour is captured by factoring out the behaviour of possible targets and destinations respectively.

```
CLASS
      //    EnableDrag exported. A FileIcon is assumed to
      //    be contained within a ME group.
      DD_Target( [ ]Mv )( M^ ){
            EVENT Hilite, Dehilite, MoveBy,
                  MoveFile , TrashIt;
            ON ACTIVATION {
                  Hilite;
                  EnableDrag;
            }
            Dehilite <- M^;
            MoveBy[Ea,Eb] <- cursor_moved;
      }

      //    EnableDrag imported.
      DD_Destination( EnableDrag )( M^ ){
            EVENT Hilite, Dehilite;
            Hilite <- -[ ];
            Dehilite <- [ ]- | M^;
      }

      //    DoFileMove imported.
      FileIcon : DD_Target( )( ){
      }

      //    DoFileMove exported.
      FolderIcon ( )( ){
            VAR NotThisOne;
            ON ACTIVATION{ NotThisOne = FALSE; }
            GROUPING ME;

            MoveFolder : DD_Target( )( ){
                  ON ACTIVATION{ NotThisOne = TRUE; }
                  ON DEACTIVATION{ NotThisOne = FALSE; }
            }
            ReceiveFile:DD_Destination[NotThisOne]( )( ){
                  DoFileMove <- [ ]M^;
            }
      }

      //    DoTrash exported.
      Trash : DD_Destination( )( ){
            DoTrash <- [ ]M^;
      }
```

Figure 8.7: Multiple "drag and drop" interpretation including folder targets.

[1]In this example we have assumed for clarity that classes containing sub-dialogues can be defined within DAL. In the current implementation this is not permitted, but can easily be "worked round" by specifying a new instance of a specified class in response to an instance of a given class being created (see Chapter 5).

8.5. GESTURE RECOGNITION

As seen in Chapter 7, by making use of spatially disjoint widget groupings activated by CoC events, it is possible to recognise the direction of a cursor movement. This can form the basis for a simple gesture recognition system.

In this example, the design of a *gesture tablet* will be examined. This is a small floating window with which the user can interact by dragging the cursor over it in a manner appropriate to the type of effect required. This gesture tablet could be used for any system in which there is a need to adjust two parameters simultaneously. Consider for example a drawing program similar to MacDraw®. In such a program the user could produce various draw objects and then may wish to adjust there height and width, perhaps independently, changing the aspect ratio of the objects of interest. Figure 8.8 shows the trace of the users actions and the desired effect on a rectangular draw object.

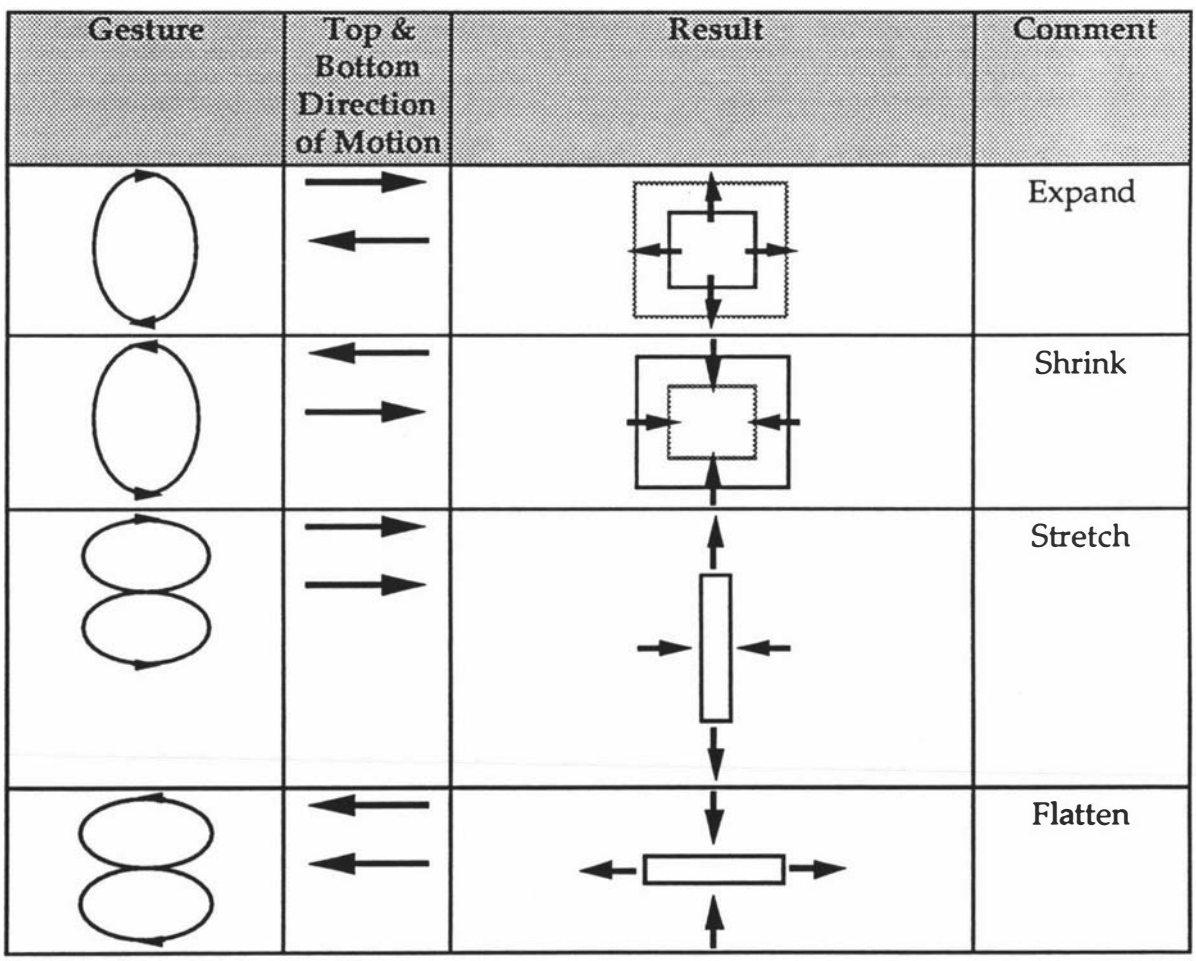


Figure 8.8: Gesture controls.

An important observation is that a gesture can be completely characterised by the direction of the cursor trace at the top and bottom of the gesture. Based on this, we can split the task of identifying the gesture into two sub-tasks or identifying the direction of motion at the top and bottom of the gesture respectively. These tasks are essentially the same, hence the problem for the designer reduces to one of being able to characterise the direction of motion across two, spatially separated objects.

Consider two icons, *A* and *B* as shown in Figure 8.9. If *A* and *B* had corresponding dialogues activated by dragging the cursor into their context, and forming a mutually exclusive group, the following gesture G1, the sub-system would be left in state {A}, and following gesture G2 state {B}.

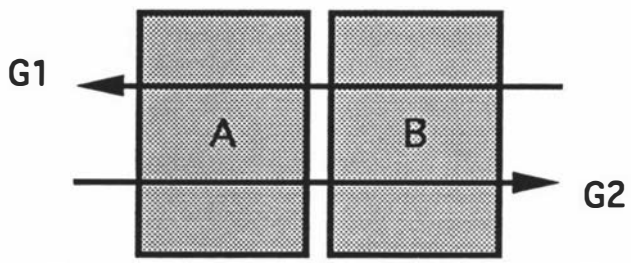


Figure 8.9. Determining the direction of motion.

Finally, in order to generate a single, categorising event we need to identify the end of the gesture. In a very simple system the user could identify this by depressing or releasing the mouse key. However this would clearly be impractical in our target system containing more than one gesture recognition unit. It follows that the best approach is to introduce a further object, **C**, an enclosing parent of **A** and **B**. The end of the gesture can then be denoted by observing when the cursor has left the context of **C** as shown in Figure 8.10.

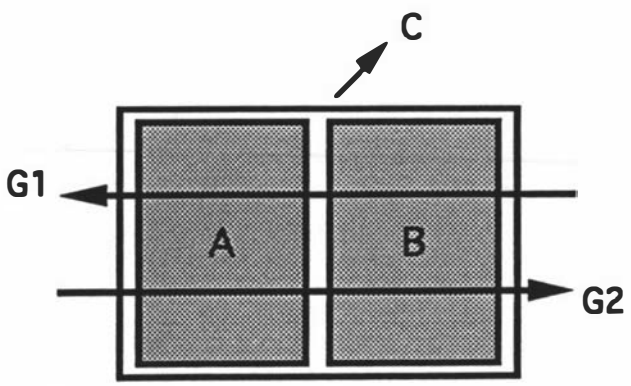


Figure 8.10: A Gesture Recognition Unit.

The DAL system corresponding to this gesture recognition unit is shown in Figure 8.11.

```
SYSTEM
   ...
   /*
    *     This subsystem exports events G1 and G2 in
    *     response to the corresponding gesture.
    */
   C( )( ){
         EVENT Done;
         GROUPING ME;

         Done <- [ ]~;

         A( ~[ ] )( ){
               G1 <- Done;
         }
         B( ~[ ] )( ){
               G2 <- Done;
         }
   }
```

Figure 8.11: DAL system corresponding to the gesture recognition unit in Figure 8.10.

Coming back to our original stated objective of designing a compound gesture tablet, capable of identifying one of four alternative actions, it can be seen that this can be constructed by incorporating two simple systems of the type just described. It is required that this gesture tablet sub-system exports one of four possible events signifying which action the user carried out. In the following example the gesture tablet generates events of Expand, Shrink, Stretch and Flatten as identified in Figure 8.8. The components of the tablet are shown in Figure 8.12. The top tablet components are kept somewhat separate from the bottom set so as to allow the user some leeway in generating their figure of eight actions.

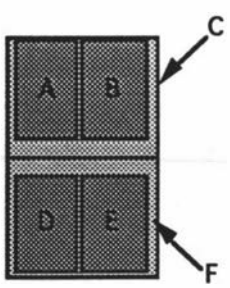


Figure 8.12: Compound Gesture Recognition Tablet Components.

The corresponding DAL source is shown in Figure 8.13. In this system the user's actions give rise to *LtoR* (Left to Right) and *RtoL* (Right to Left) events at the top and bottom respectively.

```
SYSTEM
  /*
   *    Exports events Expand, Shrink, Stretch and Flatten
   */
  GRT( )( ){
        EVENT TopLtoR, TopRtoL, BottomLtoR, BottomRtoL;
        C( )( ){
              EVENT CDone;
              GROUPING ME;
              CDone <- [ ]-;
              A( -[ ] )( ){
                    TopRtoL <- CDone;
              }
              B( -[ ] )( ){
                    TopLtoR <- CDone;
              }
        }
        F( )( ){
              EVENT FDone;
              GROUPING ME;
              FDone <- [ ]-;
              D( -[ ] )( ){
                    BottomRtoL <- CDone;
              }
              E( -[ ] )( ){
                    BottomLtoR <- CDone;
              }
        }
        Categorise( )( ){
              GROUPING ME;
              DidTopLtoR( TopLtoR )( ){
                    Stretch <- BottomLtoR;
                    Expand <- BottomRtoL;
              }
              DidTopRtoL( TopRtoL )( ){
                    Shrink <- BottomLtoR;
                    Flatten <- BottomRtoL;
              }
              ReSynchronise( Mv )( ){ }
        }
  }
```

Figure 8.13: DAL source for the Gesture Reconition Tablet shown in Figure 8.12.

Categorising of the gesture consists of determining which bottom action event (*BottomLtoR* or *BottomRtoL*) follows which top action event (*TopLtoR* or *TopRtoL*). The categorise dialogues are mutually exclusive and contain a ReSynchronise dialogue which is activated by depressing the mouse key. Hence, if the user wishes to change gesture at any point, depressing the mouse

key will effectively reset the categorising dialogues and prevent any spurious gesture recognitions.

Chapter 9
Conclusions and Further Work

9.1. INTRODUCTION

This chapter summarises the ideas developed in this research. A series of questions are stated that have been, or might well be asked about the work. In each case the questions are discussed and answered. Finally, a number of issues raised by the work, and that would warrant further investigation, are discussed.

9.2. CONTRIBUTIONS OF THIS WORK

In this work, the following new ideas have been developed:

1. A ***user-centred constructional modelling*** approach as a practical option for both rapid prototyping and implementation of direct manipulation user interfaces from a behavioural model specification.

2. The modelling of direct manipulation dialogues in terms of ***selection of dialogue handlers.***

3. The notion of ***dialogue activation*** as an approach to modelling the explicit selection and deselection of dialogue handlers.

4. The incorporation of dialogue handlers into a ***Lean Cuisine style tree*** as a semi-graphical approach to modelling both sequence and concurrency with in an interface.

5. The concept of a ***dialogue class*** as a mechanism for capturing generic dialogue behaviour in terms of user actions.

6. The concept of a ***platform independent*** approach for constructional modelling of direct manipulation interfaces.

9.3. QUESTIONS AND ANSWERS

In this section a series of questions that have been asked about this project are stated and answered.

Question 1: The initial hypothesis was that a solution to translating a behavioural model to its constructional equivalent was to use a user-centred constructional model in order to ease the translation process. Has this been demonstrated?

The analysis of behavioural models (Chapter 2) showed that they varied greatly in terms of the informational elements from which they are composed. As a consequence, only a small number of behavioural models could be considered dialogue models, and as such suitable for translation to a constructional equivalent (as discussed in Chapter 4). UAN was described as a dialogue description language designed for DMUI. A range of dialogues were examined in Chapters 5 and 8. In most cases the starting point was to describe the dialogues using UAN, and then to translate these into DAL, which in each case was fairly straight forward. This would appear to demonstrate that the above hypothesis is supported in the case of UAN and DAL. Whether or not the translation would be easy or indeed possible for other behavioural dialogue models has not been established. However other recent work has shown that Lean Cuisine+, a new, high level dialogue model for DMUI can be translated into DAL via UAN [Phillips93;Phillips94]. This suggests that the hypothesis may be more general.

Question 2: Who would be the users of DAL and PIPS?

In Chapter 1 it was shown that the translation between behavioural and constructional domains was important for both rapid prototyping and for the subsequent implementation. It is important that "both bridges are crossed" for the full benefit of behavioural modelling to be realised. The ease of translation of UAN to DAL shows that PIPS would be a suitable environment for rapid prototyping. In the case of each of the dialogues implemented in this research, PIPS was able to execute them well within the time constraints of DMUI. This

would tend to indicate the approach used by PIPS might be suitable for final implementation. It follows that the users could be both human factors personnel involved in prototyping new interfaces and software engineers implementing interfaces.

Question 3: Behavioural models are used to a large extent by non-programmers. DAL might be considered just another programming language requiring the skills of a programmer for it to be effectively used. Would this not be a major barrier to its use?

As the DAL/PIPS toolset currently stands this may well be the case. However, it may well be that the translation of UAN to DAL could, to a large extent, be automated. For example, entries in the feedback column of a UAN specification have a one-to-one mapping with corresponding event statements in the DAL translation. Activators and deactivators can also be recognised in a UAN specification, although their identification is not so straight forward. As an example, the end of a UAN action sequence may be `~[trash]M^`, the corresponding deactivator being `M^`. These observations suggest that an investigation into the development of tools to assist in the translation would be worthwhile. The same arguments hold with respect to tools for the translation of Lean Cuisine+ into DAL.

Question 4: How easy is a user-centred constructional design language to use?

An early version of DAL and PIPS was used by a group of ten honours students for an assignment to implement the interface for a paint tool. All of these students roughed out the design of the interface in the form of an activation tree (ie, the basic Lean Cuisine framework of DAL). Feedback from these students indicated that they had no problems modelling sequence and concurrency within this framework, and that they found the DAL notation easy to understand and use. The only problems reported concerned minor problems with the tools. The quality of the assignments from these students seemed to confirm the responses on the questionnaire.

Question 5: A default μdialogue is defined as one with no activators. In a mutually compatible group a default μdialogue is active whenever the parent

is active, so there is really no difference between the parent and child. Why have a separate child in this case?

Consider the system depicted in Figure 9.1.

```
A( start )( end ){
        B( )( ){
                EVENT dothis, done;
                C( dothis )( done ){
                }
        }
}
```

Figure 9.1: A default child in a mutually compatible group.

Whenever µdialogue A becomes active the µdialogue B will also activate by virtue of it being a default. Hence the activation states of A and B are intrinsically linked and A being active can be replaced by (A+B) being active. However this separation does have some advantages. The scope of variables and events is the µdialogue subtree beneath their point of declaration. Hence the events dothis and done defined in B are available to µdialogue C, but not to A. This scoping reduces complexity by reducing the number of events and variables that need to be considered at any given point within an interface specification.

Question 6: The behaviour of a default µdialogue is different in a mutually exclusive group from its behaviour in a mutually compatible group. Does this difference in behaviour between the two groupings cause any difficulties?

The interpretation of a µdialogue which does not have any associated activation events is that it is a µdialogue that does not need any events to cause it to be active. All that is necessary is for the parent to become active, and any constraints based on grouping must be met. Viewed in this way, there is no difference between the behaviour of a µdialogue without any activators between the two groupings. The difficulty (if any) seems to be the attachment of the term default to such µdialogues. In a mutually exclusive grouping, a default µdialogue is active if the parent is active and none of the µdialogues siblings are active. In this situation a µdialogue without any activators is truly

acting as a default. In a mutually compatible group a μdialogue becomes active as soon as the parent becomes active, and subsequently becomes inactive as soon as the parent becomes inactive. In this situation the μdialogue in question is becoming active by default. In both situations the behaviour of a default μdialogue is not dissimilar to the behaviour of default menemes within the Lean Cuisine model. No difficulties have been experienced related to any apparent "difference in behaviour".

9.4. FURTHER WORK

i) In this work the development of a DAL description from an initial UAN model has been examined. UAN was chosen because of its simplicity, and its being targeted towards direct manipulation interfaces. As discussed above, Lean Cuisine+ is a behavioural dialogue modelling notation for DMUI. Lean Cuisine+ is a layered notation, and as such would support incremental user interface development more effectively that UAN alone. Because of this, a UIDE using Lean Cuisine+ as the initial modelling approach, and its subsequent translation to DAL for implementation would have some advantages over the use of UAN alone.

ii) DAL and PIPS were developed to investigate the value of a user-centred constructional model. Because of this some aspects of the interface were not fully addressed. In particular this applies to the areas of high level dialogue control and application linkage. In order to construct a comprehensive user interface development environment based on DAL/PIPS, the language and the environment would need to be extended.

The majority of work undertaken on UIDSs has been concerned with the modelling of the control layer. Some aspects of sequence and control are modelled within DAL, other aspects are not.

Consider, for example a group of file icons within the Macintosh Finder®. Their default group behaviour is mutually exclusive, selecting one icon in the group causing any others that are selected to become deselected. However, if the SHIFT key is depressed, then the icons

exhibit mutually compatible behaviour. This dialogue is difficult to model in DAL since:

a) There is no facility to dynamically change the grouping attribute of a μdialogue.

b) Each widget can have only a single μdialogue associated with it.

As a consequence, in DAL this can only be modelled using approaches such as that shown in Figure 9.2. This is unsatisfactory.

```
// ...
Finder( )( ){
        Icons( )( ){
                I1( []Mv )( Desel1 ){
                        ON ACTIVATION{ Sel1; }
                }
                I2( []Mv )( Desel2 ){
                        ON ACTIVATION{ Sel2; }
                }
        }
        GroupBehaviour( )( ){
                GROUPING ME;
                MCGroup( 'SHIFT'v )( 'SHIFT'^ ){
                }
                MEGroup( )( ){
                        // Default
                        Desel1 <- Sel2;
                        Desel2 <- Sel1;
                }
        }
}
```

Figure 9.2: One approach to modelling a change in grouping behaviour.

A better solution to this problem is to use a suitable notation to model high level control separate from the low level control dealt with by DAL. A linkage between the DAL model and the high level control model would then need to be established. It was felt that this extension to DAL was a major project and fell outside the scope of the current research.

An improved mechanism for developing and attaching new widgets and application functions would need to be developed. In PIPS this is achieved by adding directly to the PIPS source code. It would obviously be desirable to be able to develop these features separately and be able to 'plug' them into the interface as required, perhaps as code resources. A graphical editor for sizing and positioning statically defined widgets within an interface would also be an essential requirement.

Apart from this inconvenience of the current scheme of adding application functions, the mechanism of application linkage has not been addressed to any great extent, the token passing used in PIPS being too restrictive for some applications. The extension of DAL and PIPS to use active values [Myers87] or constraint grammars [Zanden89] might be suitable approaches to addressing this problem.

iii) As examined in Chapter 1, one important function of behavioural modelling is analytical evaluation, that is the characterisation of the user interface in terms of one or more metrics obtained from the model.

DAL supports the abstraction of generic behaviour present within an interface. The degree to which such generic behaviour is referenced from within the interface as a whole is obviously a measure of the degree of consistency of interaction style within the interface. Metrics to quantify this consistency could be identified, and used to compare different designs in terms of useability and learnability, on the basis that consistency is directly related to these concepts.

The one difficulty with this approach is that dialogue classes were introduced into DAL as an aid for the 'programmer'. As with all such aids, how and to what degree they are used is dependent on the user. It follows that the class hierarchy produced by a developer would be unsuitable for metrication purposes. A solution to this problem is to develop a re-write system that takes a fully expanded specification (as produced by DALTRAN), and extracts a dialogue class hierarchy from it, producing a new, but equivalent specification that could then be used as the basis from which useful metrics could be obtained.

The hierarchical modelling present in DAL gives a measure of the number of actions and action sequence required in order to access a given μdialogue. It follows that the hierarchy depth is a measure of the user interface complexity. It may also be possible to relate measures such as numbers of activators, event statements etc. to user interface complexity. Complexity measured in this way could then be investigated in terms of its relationship to various performance criteria.

The identification of metrics that could be related to user related performance and their automatic extraction from a DAL description would greatly assist the development of good user interfaces. As discussed in Chapter 2, previous characterisation models have been developed purely for the purpose of obtaining such metrics and have not been suitable for translation to an implementation equivalent. It follows that the metrics obtained have only been suggested to be related to usability, but have not been shown to be related to usability. Because of its ability for direct execution it would be possible to extract candidate metrics from sample user interface and subsequently to evaluate the actual interfaces using human subjects. Correlation between performance inferred from the metrics and performance measured with the actual user interface would lend some strength to the argument as to the value of such metrics[1].

iv) PIPS is very much a prototype UIDE. The development of a production/commercial quality version would be a major, yet very worthwhile undertaking. Such a system would need to supply tools to assist with the development of a user interface definition using UAN and or Lean Cuisine+, and to assist with the translation of this model to its DAL equivalent. Lean Cuisine+ is a graphical notation for describing the behaviour of graphical user interfaces independently of their implementation [Phillips93].

v) In specifying an action event in DAL only point contexts can be specified with respect to the cursor. There are circumstances in which it would be necessary to specify other contexts. These would include:

   a) Specifying that the event does not occur in a given context. For example, deselecting an object by clicking the mouse outside its context. This behaviour can usually be modelled in an alternative fashion in DAL by defining a mutually exclusive relationship between some background object/dialogue and the one that you wish to deselect. For example:

---

[1] A grant to support an investigation into this has recently been awarded by Monash University.

```
...
GROUPING ME;
Object1( [ ]Mv  )( ){
}
Background( [ ]Mv )( ){
}
...
```

In this example, clicking on the background will cause Object1 to deselect.

b) Specifying a regional spatial relationship. For example, consider the technique for making multiple selections in the Macintosh environment. A region is identified by dragging the cursor, and any objects whose regions intersect this user defined region are selected. There is no way of defining such a spatial relationship between regions in DAL.

vi) The dialogue activation model in concerned with describing low level behaviour within the interface. In terms of a Seeheim architecture, this would probably be categorised as the presentation layer. No attempt has been made within this work to address the equivalent of the control or session management. It is now well recognised that practical user interface development environments need to support a variety of tools and techniques and the development of tools for adding in these higher level behavioural constraints within a user interface would be highly desirable.

Bibliography

[Abowd90] ABOWD, G.D. (1990): Agents: Communicating Interactive Processes. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*, pp 143-148

[Alexander87] ALEXANDER, H. (1987): Executable Specifications as an Aid to Dialogue Design. In *Human-Computer Interaction - INTERACT'87*, North Holland, pp 739-744

[Alexander90] ALEXANDER, H. (1990): Structuring Dialogues using CSP, In *Formal Methods in Human-Computer Interaction.*, Ch. 9, Harrison, M. and Thimbleby, H. (Eds), Cambridge University Press, Cambridge, UK., pp 273-295

[Allen87] ALLEN, R.E. (1987): *The Pocket Oxford Dictionary of Current English* , Oxford University Press

[Anderson90] ANDERSON, P.S. AND APPERLEY, M.D. (1990): An Interface Prototyping System Based on Lean Cuisine, *Interacting with Computers*, Vol 2, No 2, pp 217-226

[Anderson91] ANDERSON, P.S. AND APPERLEY, M.D. (1991): *Dialogue Activation: A Design Approach for Direct Manipulation Interfaces*, Massey University, Information Science Report, No 91/2

[Anderson92] ANDERSON, P.S. (1992): Dialogue Activation Language: A Structured Approach to Modelling Sequence and Concurrency in Direct Manipulation Interfaces. In *Proceedings of the New Zealand Computer Science Research Students Conference*, pp 15-22

[Anderson94] ANDERSON, P.S. (1994): PIPS: A User Centred Approach to Rapid Prototyping. In *Proceedings of OZCHI'94*, CHISIG, pp 47-52

[Apperley89] APPERLEY, M.D. AND SPENCE, R. (1989): Lean Cuisine: A Low-Fat Notation for Menus, *Interacting with Computers*, Vol 1, No 1, pp 43-68

[Apple-Computer87] APPLE COMPUTER, (1987): *Human-Interface Guidelines: The Apple Desktop Interface* , Addison-Wesley, Reading, MA.

[Apple-Computer91] APPLE COMPUTER, (1991): *Inside Macintosh* , Vol VI, Addison-Wesley, Reading, MA.

[Bass88] BASS, L. AND COUTAZ, J. (1988): Principles from cognitive psychology and human factors. In *Proceedings of IFIP WG 2.7 Seminar on Advanced User Interfaces at the University of Melbourne*, The University of Melbourne, pp 14-31

[Bass91] BASS, L. AND COUTAZ, J. (1991): *Developing Software for the User Interface* , Addison-Wesley Publishing Company

[Bass93] BASS, L. AND DEWAN, P. (1993): *User Interface Software* , Wiley

[Bos83] BOS, J.V.D., PLASMEIJER, M.J., AND HARTEL, P.H. (1983): Input-Output Tools: A Language Facility for Interactive and Real-Time Systems, *IEEE Transactions on Software Engineering*, Vol SE-9, No 3 [May 1983] , pp 247-259

[Burns90] BURNS, A. AND WELLINGS, A. (1990): *Real-Time Systems And Their Programming Languages* , Addison-Wesley

[Buxton86] BUXTON, W. AND MYERS, B.A. (1986): A Study in Two-Handed Input. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, pp 321-326

[Card80] CARD, S.K., MORAN, T.P., AND NEWELL, A. (1980): The keystroke- level model for use performance with interactive systems, *Communications of the ACM*, Vol 23, pp 396-410

[Card83] CARD, S.K., MORAN, T.P., AND NEWELL, A. (1983): *The Psychology of Human-Computer Interaction* , Lawrence Erlbaum Associates, Hillsdale, NJ.

[Cardelli85] CARDELLI, L. AND PIKE, R. (1985): Squeak: a Language for Communicating with Mice, *Computer Graphics*, Vol 19, No 3 [July 1985] , pp 199-204

[Coutaz87] COUTAZ, J. (1987): PAC, an Object Oriented Model for Dialog Design, *Proceedings of INTERACT'87*, pp 431-436

[Coutaz90] COUTAZ, J. (1990): Architectural Models for Interactive Software: Failures and Trends, In *Engineering for Human-Computer Interaction.*, Cockton, G. (Eds), North-Holland, pp 137-153

[Deitel90] DEITEL, H.M. (1990): *Operating Systems* , Addison Wesley, Reading, Massachusetts

[Denert77] DENERT, E. (1977): Specification and Design of Dialogue Systems with State Diagrams. In *International Computing Symposium*, North-Holland, Amsterdam, pp 417-423

[Dijkstra76] DIJKSTRA, E.W. (1976): *A Discipline of Programming* , Prentice-Hall, Englewood Cliffs, N.J.

[Dix93] DIX, A., FINDLAY, J., ABOWD, G., AND BEALE, R. (1993): *Human-Computer Interaction* , Prentice Hall

[Edmonds81] EDMONDS, E.A. (1981): Adaptive Man-Computer Interfaces, In *Computing Skills and the User Interface.*, Coombs, M.J. and Alty, J.L. (Eds), Academic Press, London, pp 389-426

[Edmondson91] EDMONDSON, W.H. (1991): Interaction Taxonomy, *Proceedings of the Second Venaco Workshop on the Structure of Multimodal Dialogue* [September 1991]

[Flecchia87] FLECCHIA, M.A. AND BERGERON, R.D. (1987): Specifying Complex Dialogs in ALGAE. In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, pp 229-234

[Foley80] FOLEY, J.D. (1980): The Structure of Interactive Command Languages, In *Methodology of Interaction.*, Geud, R.A. (Eds), North-Holland, pp 227-234

[Foley87] FOLEY, J.D., KIM, W.C., AND GIBBS, C.A. (1987): Algorithms to Transform the Formal Specification of a User-Computer Interface. In *Proceedings of IFIP INTERACT'87: Human-Computer Interaction*, pp 1001-1006

[Foley89] FOLEY, J., KIM, W.C., KOVACEVIC, S., AND MURRAY, K. (1989): Defining Interfaces at a High Level of Abstraction, *IEEE Software*, Vol 6, No 1 [January 1989] , pp 25-32

[Foley90] FOLEY, J., VAN DAM, A., FEINER, S., AND HUGHES, J. (1990): *Computer Graphics: principles and practice* , Addison-Wesley

[Goldberg84] GOLDBERG, A. AND ROBSON, D. (1984): *Smalltalk-80: The Language and its Implementation* , Addison-Wesley

[Green85a] GREEN, M. (1985): Report on Dialogue Specification Tools. In *User Interface Management Systems*, Pfaff, G.E. (Eds), Springer Verlag, Berlin., pp 9-20

[Green85b] GREEN, M. (1985): The University of Alberta User Interface Management System, *Computer Graphics*, Vol 19, No 3 [July 1985] , pp 205-213

[Green86] GREEN, M. (1986): A Survey of Three Dialogue Models, *ACM Transactions on Graphics*, Vol 5, No 3, pp 244-275

[Grudin87] GRUDIN, J., EHRLICH, S.F., AND SHRINER, R. (1987): Positioning Human Factors in the User Interface Development Chain. In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, pp 125-131

[Guest82] GUEST, S.P. (1982): The Use of Software Tools for Dialogue Design, *International Journal of Man-Machine Studies*, Vol 16, No 3, pp 263-285

[Harel84] HAREL, D. (1984): *Statecharts: A Visual Approach to Complex Systems*, Weitzman Institute of Science, No CS84-05, Rehovot, Israel

[Harel87] HAREL, D., PNUELI, A., SCHMIDT, J.P., AND SHERMAN, R. (1987): On the Formal Semantics of Statecharts. In *Proceedings of the 2nd Symp. on Logic in Computer Science*, pp 54-64

[Harel88] HAREL, D. (1988): On Visual Formalisms, *Communications of the ACM*, Vol 31, No 5, pp 514-530

[Hartson89] HARTSON, H.R. AND HIX, D. (1989): Human-Computer Interface Development:Concepts and Systems for its Management, *ACM Computing Surveys*, Vol 21, No 1, pp 5-92

[Hartson90] HARTSON, R., SIOCHI, A.C., AND HIX, D. (1990): The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs, *ACM Transactions on Information Systems*, Vol 8, No 3 [July 1990] , pp 181-203

[Hatley87] HATLEY, D.J. AND PIRBHAE, I.A. (1987): *Strategies for Real-Time System Specification* , Dorset House Publishing, New York

[Hill87a] HILL, R.D. (1987): Event-Response Systems - A Technique for Specifying Multi-Threaded Dialogues. In *Proceedings of ACM CHI+GI'87*, pp 241-248

[Hill87b] HILL, R.D. (1987): *Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction*, Ph.D. dissertation, University of Toronto

[Hill87c] HILL, R.D. (1987): Some Important Features and Issues in User Interface Management Systems, *Computer Graphics*, Vol 21, No 2 [April 1987] , pp 116-120

[Hill89] HILL, R.D. AND HERMMAN, M. (1989): The structure of TUBE - a tool for implementing advanced interfaces. In *Proceedings of Eurographics'89*, North-Holland, Amsterdam, pp 15-25

[Hill90] HILL, R. AND HERRMANN, M. (1990): The Composite Object User Interface Architecture. In *User Interface Management and Design*, Duce, D.A., Gomes, M.R., Hopgood, F.R.A., and Lee, J.R. (Eds), Springer-Verlag, Berlin, pp 257-271

[Hix93a] HIX, D. AND HARTSON, H.R. (1993): *Developing User Interfaces: Ensuring usability through product & process* , Wiley

[Hix93b] HIX, D. AND HARTSON, H.R. (1993): Chapter 1, *User Interface Software*,Bass, L. and Dewan, P., (Eds), John Wiley & Sons, pp 1-30

[Hoare95] HOARE, C.A.R. (1995): *Communicating Sequential Processes* , Prentice-Hall International, London

[Hoffner89] HOFFNER, Y., DOBSON, J., AND IGGULDEN, D. (1989): A New User Interface Architecture. In *Proceedings of the HCI'89 Conference on People and Computers V*, British Informatics Society Ltd, pp 169-189

[Hopgood80] HOPGOOD, F.R.A. AND DUCE, D.A. (1980): A Production System Approach to Interactive Graphic Program Design, In *Methodology of Interaction.*, Guedj, R. (Eds), North-Holland, Amsterdam, pp 247-263

[Hudson87] HUDSON, S.E. (1987): UIMS Support for Direct Manipulation Interfaces, *Computer Graphics*, Vol 21, No 2 [April 1987] , pp 120-124

[Hudson92] HUDSON, J. (1992): *Lean Cuisine Transition Functions*, [unpublished].

[Hutchins86] HUTCHINS, E.L., HOLLAN, J.D., AND NORMAN, D.A. (1986): Chapter 5, *USER CENTRED SYSTEM DESIGN*,Norman, D.A. and Draper, S.W., (Eds), Lawrence Erlbaum Associates: Hillsdale, NJ., pp 87-124

[Jacob83] JACOB, R.J.K. (1983): Using Formal Specifications in the Design of Human-Computer Interface, *Communications of the ACM*, Vol 26, No 4 [April 1983] , pp 259-264

[Jacob85] JACOB, R.J.K. (1985): A State Transition Diagram Language for Visual Programming, *IEEE Computer*, Vol 18, No 8 [August 1985] , pp 51-59

[Jacob86] JACOB, R.J.K. (1986): A Specification Language for Direct-Manipulation User Interfaces, *ACM Transactions on Graphics*, Vol 5, No 4, pp 283-317

[Jeffries91] JEFFRIES, R., MILLER, J.R., WHARTON, C., AND UYEDA, K.M. (1991): User Interface Evaluation in the Real World: A Comparison of Four Techniques. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, pp 119-124

[Johnson75] JOHNSON, S. (1975): *Yacc: yet another compiler compiler*, Bell Laboratories, Computing Science Technical Report, No 32, Murray Hill, NJ

[Johnson92] JOHNSON, P. (1992): *Human Computer Interaction: Psychology, Task Analysis and Software Engineering* , McGraw-Hill Book Company Europe

[Kernigan78] KERNIGAN, B.W. AND RITCHIE, D.M. (1978): *The C programming language* , Prentice Hall

[Kieras85] KIERAS, D. AND POLSON, P.G. (1985): An Approach to the Formal Analysis of User Complexity, *International Journal of Man-Machine Studies*, Vol 22, No 4, pp 365-394

[Koivunen88] KOIVUNEN, M.R. AND MANTYLA, M. (1988): HutWindows: An Improved Architecture for a User Interface Management System, *IEEE Computer Graphics and Applications*, Vol 8, No 1 [January 1988] , pp 43-52

[Kuntz90] KUNTZ, M. AND MELCHERT, R. (1990): Pasta-3's Requirements, Design and Implementation: A Case Study in Building a Large, Complex Direct Manipulation Interface, In *Engineering for Human-Computer Interaction.*, Cockton, G. (Eds), Elsevier Science Publishers B.V., pp 43-60

[Lee90] LEE, E. (1990): User-Interface Development Tools, *IEEE Software*, Vol 7, No 3 [May 1990] , pp 31-36

[Lesk75] LESK, M.E. (1975): *Lex: a Lexical Analyser Generator*, Bell Laboratories, Computing Science Technical Report, No 39, Murray Hill, NJ

[Long86] LONG, J.B. (1986): User and Abuse of Models: HI and IKBS perspectives. In *Conf. Record The Alvey Conf. (Sussex University, Brighton, UK)*, pp 40-41

[MacLean87] MACLEAN, A. (1987): Human factors and the design of user interface management systems: EASIE as a case study, *Inf. Softw. Technol.*, Vol 29, No 4, pp 192-201

[Milner80] MILNER, R. (1980): A Calculus of Communicating Processes, In *Lecture Notes in Computer Science.*, Springer-Verlag

[Moran81] MORAN, T.P. (1981): The command language grammar: A representation for the user interface of interactive computer systems, *International Journal of Man-Machine Studies*, Vol 15, pp 3-51

[Myers87] MYERS, B.A. (1987): Creating Dynamic Interaction Techniques by Demonstration. In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, pp 271-278

[Myers89] MYERS, B.A. (1989): User-Interface Tools: Introduction and Survey, *IEEE Software*, Vol 6, No 1 [January 1989] , pp 15-23

[Myers90] MYERS, B.A., GIUSE, D.A., DANNENBERG, R.B., ZANDEN, B.V., KOSBIE, D.S., PERVIN, E., MICKISH, A., AND MARCHAL, P. (1990): Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces, *IEEE Computer*, Vol 23, No 11 [November 1990] , pp 71-85

[Naur63] NAUR, P. (1963): Revised report on the algorithmic languages ALGOL60, *Communications of the ACM*, Vol 6 [January 1963]

[Netzer92] NETZER, R.H.B. AND MILLER, B.P. (1992): What Are Race Conditions? Some Issues and Formalizations, *ACM Letters on Programming Languages and Systems*, Vol 1, No 1 [March 1992] , pp 74-88

[Newman68] NEWMAN, W.M. (1968): A System for Interactive Graphical Programming. In *Proceedings of the AFIPS Spring Joint Computer Conference*, Thompson Books, Washington, D.C., pp 47-54

[Nielsen90] NIELSEN, J. (1990): A meta-model for interacting with computers, *Interacting with Computers*, Vol 2, No 2 [August 1990] , pp 147-160

[Olsen83] OLSEN, D.R. AND DEMPSEY, E.P. (1983): SYNGRAPH: A Graphical User Interface Generator, *Computer Graphics*, Vol 17, No 3 [July 1983] , pp 43-50

[Olsen86] OLSEN, D.R. (1986): MIKE: The Menu Interaction Kontrol Environment, *ACM Transactions on Graphics*, Vol 5, No 4 [October 1986] , pp 318-344

[Olsen89] OLSEN, D.R. (1989): A Programming Language Basis for User Interface Management. In *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems*, pp 171-176

[Olsen90] OLSEN, D.R. (1990): Propositional Production Systems for Dialog Description. In *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, pp 57-63

[Parnas69] PARNAS, D.L. (1969): On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System. In *Proceedings of the National ACM Conference*, ACM, New York, pp 379-385

[Payne86] PAYNE, S.J. AND GREEN, T.R.G. (1986): Task-action grammars: A model of the mental representation of task languages, *Human-Computer Interation*, Vol 2, No 2, pp 93-133

[Payne91] PAYNE, S.J. (1991): Interface Problems and Interface Resources, In *Designing Interaction: Psychology at the Human-Computer Interface.*, Ch. 8, Carroll, J.M. (Eds), Cambridge University Press, pp 128-153

[Peterson77] PETERSON, J.L. (1977): Petri Nets, *ACM Computing Surveys*, Vol 9, No 3 [September 1977] , pp 223-252

[Phillips91] PHILLIPS, C.H.E. AND APPERLEY, M.D. (1991): Direct Manipulation Interaction Tasks: A Macintosh-Based Analysis, *Interacting with Computers*, Vol 3, No 1, pp 9-26

[Phillips93] PHILLIPS, C.H.E. (1993): *The Development of an Executable Graphical Notation for Describing Direct Manipulation Interfaces*, Ph.D. dissertation, Massey University, New Zealand

[Phillips94] PHILLIPS, C. (1994): Serving Lean Cuisine+: Towards a Support Environment. In *Proceedings of OZCHI'94*, CHISIG, pp 41-56

[Reisig85] REISIG, W. (1985): *Petri Nets* , Springer-Verlag, Berlin

[Reisner81] REISNER, P. (1981): Formal Grammar and Human Factors Design of an Interactive Graphics System, *IEEE Transactions on Software Engineering*, Vol SE-7, No 2 [March 1981] , pp 229-240

[Rich83] RICH, E. (1983): *Artificial Intelligence* , McGraw-Hill

[Roudaud90] ROUDAUD, B., LAVIGNE, V., LAGNEAU, O., AND MINOR, E. (1990): SCENARIOO: A New Generation UIMS. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*, pp 607-612

[Shevlin90] SHEVLIN, F. AND NEELAMKAVIL, F. (1990): Designing the Next Generation of UIMSs. In *Proceedings of the Eurographics Lisbon Workshop on User Interface Management and Design*, Duce, D.A., Gomes, A., Hopgood, F.R.A., and Lee, E. (Eds), pp 123-133

[Shneiderman82] SHNEIDERMAN, B. (1982): Multi-party grammars and related features for defining interactive systems, *IEEE Systems, Man, and Cybernetics*, Vol SMC-12, No 2, pp 148-154

[Shneiderman83] SHNEIDERMAN, B. (1983): Direct Manipulation: A Step Beyond Programming Languages, *IEEE Computer*, Vol 16, No 8 [August 1983] , pp 57-69

[Shneiderman87] SHNEIDERMAN, B. (1987): *Designing the User Interface: Strategies for Effective Human-Computer Interaction* , Addison-Wesley Publishing Co., Reading, MA

[Shneiderman88] SHNEIDERMAN, B. (1988): We can design better user interfaces: A review of human-computer interaction styles, *Ergonomics*, Vol 31, No 5, pp 699-710

[Shneiderman92] SHNEIDERMAN, B. (1992): *Designing the User Interface: Strategies for Effective Human-Computer Interaction* , Addison-Wesley Publishing Co., Reading, MA

[Siochi89] SIOCHI, A.C. AND HARTSON, H.R. (1989): Task-Oriented Representation of Asynchronous User Interfaces. In *Proceedings of ACM CHI'89*, pp 183-188

[Six90] SIX, H.W. AND VOSS, J. (1990): DIWA: A Hierarchical Object-Oriented Model for Dialog Design, In *Engineering for Human-Computer Interaction.*, Cockton, G. (Eds), North-Holland, pp 383-402

[Stroustrup86] STROUSTRUP, B. (1986): *The C++ Programming Language* , Addison-Wesley, Reading, MA.

[Sun-Microsystems89] SUN MICROSYSTEMS, (1989): *OPEN LOOK Graphical User Interface* , Sun Microsystems, Moutain View, CA.

[Tanenbaum87] TANENBAUM, A.S. (1987): *Operating Systems: Design and Implementation* , Prentice-Hall

[Tanner87] TANNER, P.P. (1987): Multi-Thread Input, *Computer Graphics*, Vol 21, No 2 [April 1987] , pp 142-145

[Thimbleby91] THIMBLEBY, H. (1991): Can anyone work the video?, *New Scientist* [23rd February 1991] , pp 40-43

[Took90] TOOK, R. (1990): Surface Interaction: A Paradigm and Model for Separating Application and Interface. In *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, pp 35-42

[Turing36] TURING, A. (1936): On Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematics Society*, Vol 42, pp 230-265

[Veer90] VAN DER VEER, G.C., BROOS, D., DONAU, K., FOKKE, M.J., AND YAP, F. (1990): ETAG – Some Applications of a Formal Representation of the User Interface. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*, pp 169-174

[Wasserman85a] WASSERMAN, A.I. (1985): Extending State Transition Diagrams for the Specification of Human-Computer Interaction, *IEEE Transactions on Software Engineering*, Vol SE-11, No 8 [August 1985] , pp 699-713

[Wasserman85b] WASSERMAN, A.I. AND SHEWMAKE, D.T. (1985): The Role of Prototypes in the User Software Engineering (USE) Methodology, In *Advances in Human-Computer Interaction.*, Vol 1, Hartson, H.R. (Eds), Ablex, Norwood, N.J., pp 191-210

[Wellner89] WELLNER, P.D. (1989): Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. In *Proceedings of CHI'89*, ACM, pp 177-182

[Yap90] YAP, S.K. AND SCOTT, M.L. (1990): PENGUIN: A Language for Reactive Graphical User Interface Programming. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*, pp 619-624

[Zanden89] ZANDEN, B.T.V. (1989): Constraint Grammars - A new model for specifying graphical applications. In *Proceedings of CHI'89*, ACM, pp 325-330

Appendix A
BNF Definition of DAL

The table that follows is a definition of DAL in BNF. The description does not include 'include' statements. These are processed prior to the parser examining the source. They take one of the following forms:

```
INCLUDE <file>      'file' is read from the directory 'SYS' located in
                    the same directory as the source file, or

INCLUDE "file"      'file' from the same directory as the source file.
```

Include statements can occur anywhere within the source text.

```
DAL_system              ::=   header_spec
                            | system_spec
                            | header_spec system_spec

header_spec             ::=   attribute_statement
                            | define_statement
                            | template_statement
                            | header_spec attribute_statement
                            | header_spec define_statement
                            | header_spec template_statement

attribute_statement     ::=   'ATTRIBUTE' attribute_list ';'

attribute_list          ::=   IDENTIFIER
                            | IDENTIFIER type_spec
                            | attribute_list IDENTIFIER
                            | attribute_list IDENTIFIER type_spec
```

```
type_spec              ::=   'D'
                           | 'S'
                           | 'DD'
                           | 'DS'
                           | 'SS'
                           | 'SD'

define_statement       ::=   'DEFINE' define_list ';'

define_list            ::=   define_entry
                           | define_list ',' define_entry

define_entry           ::=   IDENTIFIER NUMBER
                           | IDENTIFIER IDENTIFIER

template_statement     ::=   'CLASS' template_list ';'

template_list          ::=   template_spec
                           | template_list ',' template_spec

template_spec          ::=   IDENTIFIER guard_condition
                                    '(' activator_list ')'
                                    '(' deactivator_list ')'
                             '{'
                                  header_statements
                                  body_statements
                             '}'
                           | IDENTIFIER ':' IDENTIFIER guard_condition
                                    '(' activator_list ')'
                                    '(' deactivator_list ')'
                             '{'
                                  header_statements
                                  body_statements
                             '}'

guard_condition        ::=   [ boolean_expression ]
                           | NULL
```

```
activator_list          ::=   event_list

deactivator_list        ::=   event_list

header_statements       ::=   event_declaration header_statements
                            | var_declaration header_statements
                            | on_block header_statements
                            | grouping_statement header_statements
                            | NULL

body_statements         ::=   event_statement body_statements
                            | arithmetic_statement body_statements
                            | new_statement body_statements
                            | delete_statement body_statements
                            | NULL

event_declaration       ::=   'EVENT' system_event_list ';'

variable_declaration    ::=   'VAR' variable_list ';'

on_block                ::=   'ON' on_type '{'
                                    body_statements
                              '}'

on_type                 ::=   'ACTIVATION'
                            | 'DEACTIVATION'
                            | 'NEW'
                            | 'DELETE'

grouping_statement      ::=   'GROUPING' group_type ';'

group_type              ::=   'MC'
                            | 'ME'

system_event_list       ::=   identifier_list
```

```
variable_list          ::=   identifier_list

identifier_list        ::=   IDENTIFIER
                         |   IDENTIFIER ',' identifier_list

event_statement        ::=   IDENTIFIER '<-' originating_event_list ';'
                         |   IDENTIFIER '[' field_list ']'
                                    <-' originating_event_list ';'

field_list             ::=   field_item
                         |   field_item ',' field_item
                         |   field_item ',' field_item ',' field_item
                         |   field_item ',' field_item ',' field_item
                             ',' field_item

field_item             ::=   INTEGER
                         |   IDENTIFIER
                         |   'EX'
                         |   'EY'
                         |   'EA'
                         |   'EB'

originating_event_list ::=   system_event
                         |   action_event
                         |   system_event '|' originating_event_list
                         |   action_event '|' originating_event_list

system_event           ::=   IDENTIFIER
                         |   in_context IDENTIFIER

in_context             ::=   '[]'

action_event           ::=   move_into_context
                         |   move_outof_context
                         |   primitive_event
                         |   in_context primitive_event
```

```
move_into_context       ::=    '~[]'

move_outof_context      ::=    '[]~'

primitive_event         ::=    'Mv'
                           |   'M^'
                           |   key_id 'v'
                           |   key_id '^'

key_id                  ::=    'SHIFT'
                           |   'OPTION'
                           |   'COMMAND'
                           |   A_PRINTING_CHARACTER

arithmetic_statement    ::=    IDENTIFIER '=' expression

expression              ::=    INTEGER
                           |   variable
                           |   event_field
                           |   expression '+' expression
                           |   expression '-' expression
                           |   expression '* expression
                           |   expression '/' expression
                           |   '(' expression ')'

boolean_expression             expression '!=' expression
                           |   expression '<' expression
                           |   expression '<= expression
                           |   expression '==' expression
                           |   expression '>=' expression
                           |   expression '>' expression

new_statement                  'NEW' IDENTIFIER '<-'
                               originating_event_list
                           |   'NEW' IDENTIFIER '[' field_list ']'
                                     '<-' originating_event_list
```

```
delete_statement          'DELETE' '<-' originating_event_list

system_spec        ::=    'SYSTEM' dialogue_spec

dialogue_spec      ::=    IDENTIFIER guard_condition
                                    '(' activator_list ')'
                                    '(' deactivator_list ')'
                          '{'
                               header_statements
                               body_statements
                               dialogue_list
                          '}'
                        | IDENTIFIER ':' IDENTIFIER guard_condition
                                    '(' activator_list ')'
                                    '(' deactivator_list ')'
                          '{'
                               header_statements
                               body_statements
                               dialogue_list
                          '}'

dialogue_list      ::=    dialogue_spec
                        | dialogue_list dialogue_spec
                        | NULL
```

Appendix B
Event Cycle Detection

In this appendix, a number of example DAL sources are presented that contain event cycles. In each case the DALTRAN listing is given, showing the successful identification of the event cycle in accordance with the analysis presented in Chapter 6. Following each listing is a run-time trace showing the detection of the event cycle.

LISTING

```
Compiled by: Daltran2.06          Thu May 18 09:52:33 1995
Source File: cycle1.dal

  /*
   *   Demonstration of a DAL system containing
   *  a simple event cycle.
   */
  INCLUDE   <Widgets.dal>
  INCLUDE   <WidgetEvents.dal>
  INCLUDE   <APIEvents.dal>

  CLASS

       Button( []Mv )( M^ ){
         DISPLAY TYPE DCButton,
              size 50 100,
              location 100 100,
              name "Start Cycle",
           visible;
         EVENT WE_Hilite, WE_Dehilite;
         WE_Hilite <- []Mv | ~[];
         WE_Dehilite <- []M^ | []-;
       };

  SYSTEM

       Root()(){
         EVENT PIPS_Stop, a, b;
         DISPLAY TYPE DCWindow,
              size 300 500,
              location 10 50,
              visible;

         Terminate( 'COMMAND'v )( 'COMMAND'^ ){
           PIPS_Stop <- 'q'v;
         }

         StartCycle:Button()(){
           a <- []M^;
         }

         A( )( ){
           b <- a;
         }

         B( )( ){
           a <- b;
         }
       }


NO ERRORS

Possible event cycles:

  [ a -> b -> a ]
```

```
No of potential cycles found = 1

=============== SYMBOL TABLE ===============
A                          DIALOGUE
B                          DIALOGUE
Button                     CLASS
DCButton                   WIDGET
DCButton.location          WIDGET_ATTR     DD
DCButton.name              WIDGET_ATTR     S
DCButton.size              WIDGET_ATTR     DD
DCButton.visible           WIDGET_ATTR
DCIcon1                    WIDGET
DCWindow                   WIDGET
DCWindow.location          WIDGET_ATTR     DD
DCWindow.size              WIDGET_ATTR     DD
DCWindow.visible           WIDGET_ATTR
PIPS_Stop                  EVENT           31
Root                       DIALOGUE
StartCycle                 DIALOGUE
Terminate                  DIALOGUE
WE_Dehilite                EVENT           107
WE_Hilite                  EVENT           106
a                          EVENT           303
b                          EVENT           304
```

TRACE

```
========== cycle1 ==========
A] 00537531 Root
A] 00537531 B
A] 00537531 A
A] 00537572 StartCycle
D] 00537577                    StartCycle

POSSIBLE EVENT CYCLE:
  107->303->304->303->304->303->304->303->304->303
  ->304->303->304->303->304->303->304->303->304->303

ERROR - CInterface::Event cycle detected
```

LISTING

```
Compiled by: Daltran2.06          Thu May 18 09:32:18 1995
Source File: cycle2.dal

  /*
   *   Demonstration of a DAL system containing a cycle.
   */
  INCLUDE   <Widgets.dal>
  INCLUDE   <WidgetEvents.dal>
  INCLUDE   <APIEvents.dal>

  CLASS

       Button( []Mv )( M^ ){
         DISPLAY TYPE DCButton,
              size 50 100,
              location 100 100,
              name "Start Cycle",
           visible;
         EVENT WE_Hilite, WE_Dehilite;
         WE_Hilite <- []Mv | ~[];
         WE_Dehilite <- []M^ | []~;
       };

  SYSTEM

       Root()(){
         EVENT PIPS_Stop, a, b, c;
         DISPLAY TYPE DCWindow,
              size 300 500,
              location 10 50,
              visible;

         Terminate( 'COMMAND'v )( 'COMMAND'^ ){
           PIPS_Stop <- 'q'v;
         }

         StartCycle:Button()(){
           b <- []M^;
         }

         A( )( ){
           B( )( ){
              a <- b;
              c <- b;
           }
           C( )( ){
              b <- c;
           }
         }
       }


NO ERRORS

Possible event cycles:

  [ b -> c -> b ]
```

```
No of potential cycles found = 1

=============== SYMBOL TABLE ===============
A                          DIALOGUE
B                          DIALOGUE
Button                     CLASS
C                          DIALOGUE
DCButton                   WIDGET
DCButton.location          WIDGET_ATTR      DD
DCButton.name              WIDGET_ATTR      S
DCButton.size              WIDGET_ATTR      DD
DCButton.visible           WIDGET_ATTR
DCIcon1                    WIDGET
DCWindow                   WIDGET
DCWindow.location          WIDGET_ATTR      DD
DCWindow.size              WIDGET_ATTR      DD
DCWindow.visible           WIDGET_ATTR
PIPS_Stop                  EVENT            31
Root                       DIALOGUE
StartCycle                 DIALOGUE
Terminate                  DIALOGUE
WE_Dehilite                EVENT            107
WE_Hilite                  EVENT            106
a                          EVENT            303
b                          EVENT            304
c                          EVENT            305
```

TRACE

```
============ cycle2 ============
A] 00538344 Root
A] 00538345 A
A] 00538345 C
A] 00538345 B
A] 00538404 StartCycle
D] 00538408                   StartCycle

POSSIBLE EVENT CYCLE:
  107->304->303->305->304->303->305->304->303->305
  ->304->303->305->304->303->305->304->303->305->304

ERROR - CInterface::Event cycle detected
```

LISTING

```
Compiled by: Daltran2.06          Thu May 18 09:42:25 1995
Source File: cycle3.dal

  /*
   *   Demonstration of a DAL system containing a cycle.
   */
  INCLUDE   <Widgets.dal>
  INCLUDE   <WidgetEvents.dal>
  INCLUDE   <APIEvents.dal>

  CLASS

       Button( []Mv )( M^ ){
         DISPLAY TYPE DCButton,
              size 50 100,
              location 100 100,
              name "Start Cycle",
           visible;
         EVENT WE_Hilite, WE_Dehilite;
         WE_Hilite <- []Mv | ~[];
         WE_Dehilite <- []M^ | []~;
       };

  SYSTEM

       Root()(){
         EVENT PIPS_Stop, a;
         DISPLAY TYPE DCWindow,
              size 300 500,
              location 10 50,
              visible;

         Terminate( 'COMMAND'v )( 'COMMAND'^ ){
           PIPS_Stop <- 'q'v;
         }

         StartCycle:Button()(){
           a <- []M^;
         }

         A( a )( ){
           EVENT b, c;
           GROUPING ME;
           ON ACTIVATION{ b; }

           B( b )( ){
              c <- b;
           }
           C( c )( ){
              b <- c;
           }
         }
       }


NO ERRORS
```

```
Possible event cycles:

  [ b -> c -> b ]

No of potential cycles found = 1

=============== SYMBOL TABLE ===============
A                          DIALOGUE
B                          DIALOGUE
Button                     CLASS
C                          DIALOGUE
DCButton                   WIDGET
DCButton.location          WIDGET_ATTR     DD
DCButton.name              WIDGET_ATTR     S
DCButton.size              WIDGET_ATTR     DD
DCButton.visible           WIDGET_ATTR
DCIcon1                    WIDGET
DCWindow                   WIDGET
DCWindow.location          WIDGET_ATTR     DD
DCWindow.size              WIDGET_ATTR     DD
DCWindow.visible           WIDGET_ATTR
PIPS_Stop                  EVENT           31
Root                       DIALOGUE
StartCycle                 DIALOGUE
Terminate                  DIALOGUE
WE_Dehilite                EVENT           107
WE_Hilite                  EVENT           106
a                          EVENT           303
b                          EVENT           305
c                          EVENT           306
```

TRACE

```
============ cycle3 ============
A] 00539481 Root
A] 00539535 StartCycle
D] 00539540                    StartCycle
A] 00539540 A
A] 00539541 B
D] 00539541                    B
A] 00539541 C
D] 00539541                    C
A] 00539541 B
D] 00539541                    B
A] 00539541 C
D] 00539541                    C
A] 00539541 B
D] 00539542                    B
A] 00539542 C
D] 00539542                    C
A] 00539542 B
D] 00539542                    B
A] 00539542 C
D] 00539542                    C
A] 00539542 B
D] 00539543                    B
A] 00539543 C
D] 00539543                    C
A] 00539543 B
D] 00539543                    B
A] 00539543 C
D] 00539543                    C
A] 00539543 B
D] 00539544                    B
A] 00539544 C
D] 00539544                    C
A] 00539544 B
D] 00539544                    B
A] 00539544 C
D] 00539544                    C
A] 00539544 B
D] 00539544                    B
A] 00539544 C

POSSIBLE EVENT CYCLE:
  107->303->305->306->305->306->305->306->305->306
  ->305->306->305->306->305->306->305->306->305->306

ERROR - CInterface::Event cycle detected
```

Appendix C
Example Dialogues

In this appendix, the example dialogues developed in the body of the thesis are presented. For each example a brief description is given together with a screen dump. This is followed by the source listing and symbol table produced by DALTRAN and a representative activation trace.

1. Widget and Application Classes

The version of PIPS used for these demonstration dialogues had the following widget types defined[1]:

DCIcon1:

A single icon. This widget responds to highlight, dehighlight, show, hide and move events. A highlight event causes the icon to be shown in reverse video.

DCIcon2:

Similar to DCIcon1. However this widget has two icons associated with it, a highlight event causing the widget to switch between icons.

DCButton:

A button widget containing a single, centred text string. It responds to the same events as DCIcon1 in the same way.

DCWindow:

A simple window, with no decoration. It responds to events of show window, hide window and move.

---

[1]The name of each widget class has the prefix 'DC' as an abbrieviation for *display class*.

The location of window widgets is given in global coordinates. The location of all other widgets is given within the coordinate system of their nearest window ancestor.

The definition of all widget class names, and their valid set of attributes is given in the file *Widgets.dal*, (Figure C.1) and is included into all DAL source files.

```
/*
 *                Widgets.dal
 *                ===========
 *
 *   Definition of the standard widgets supported by PIPS2.0
 *
 */

ATTRIBUTE
  DCIcon1
  resource  D,
  size    DD,
  location  DD,
  visible;

  DCIcon2
  resource  DD,
  size    DD,
  location  DD,
  visible;

  DCButton
  size    DD,
  location  DD,
  name    S,
  visible;

  DCWindow
  size    DD,
  location  DD,
  name    S,
  visible;
```

Figure C.1: Widgets.dal

Figure C.2 lists all the events that widgets respond to within PIPS[2]. The special event *PIPS_Stop* causes PIPS to terminate its execution. These definitions are held in the file *WidgetEvents.dal* and can be included into any DAL source file.

```
DEFINE
  PIPS_Stop              31,
  WE_ShowWindow          100,
  WE_HideWindow          101,
  WE_Show                102,
  WE_Hide                103,
  WE_MoveTo              104,
  WE_MoveBy              105,
  WE_Hilite              106,
  WE_Dehilite            107,
  WE_SetSize             108,
  WE_SetLoc              109,
  WE_SetHeight           110,
  WE_SetWidth            111;
```

Figure C.2: WidgetEvents.dal

The current version of PIPS has just two application interface types:

ACSelector:

Responds to the event *API_Select*. On receiving this event, it copies the name of the µdialogue that issued the event into a buffer shared by all application interface objects. It then issues the event *API_DoWrite*.

ACScribe:

Responds to the event *API_DoWrite* by writing the text contained in application interface buffer into the nearest window widget identified by tracing up the µdialogue tree from the µdialogue with which the *ACScribe* API object is associated.

The combination of these two API objects permits easy tracing of dialogues at run-time.

The events reserved for use by API objects are defined in file APIEvents.dal (Figure C.3).

---

[2]The prefix WE stands for *Widget Event*.

```
DEFINE
  API_Select              200,
  API_DoWrite             201,
  API_Value               202,
  API_Clear               203,
  API_On                  204,
  API_Off                 205;
```

Figure C.3: APIEvents.dal

2. CLOSEBOX

This example is the Macintosh style closebox as described in Chapter 5. For demonstration purposes the closebox has been made very large with respect to its parent window (Figure C.4). Clicking on the closebox causes the window the become hidden. Clicking on the button *show*, causes the window to be redisplayed.

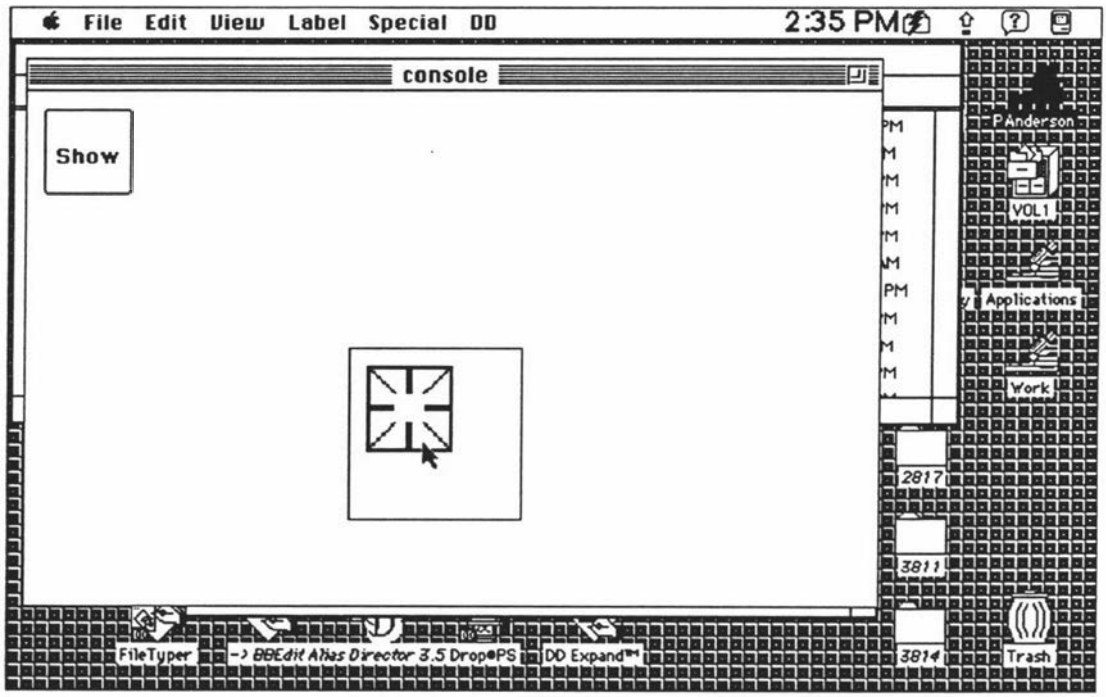


Figure C.4: Screen dump of closebox example, with the closebox selected.

Listing

```
Compiled by: Daltran2.06                    Wed Feb  1 14:03:03 1995
Source File: Closebox.dal

  /*
   *   Simple window with a closebox example.
   */

  INCLUDE   <Widgets.dal>
  INCLUDE   <WidgetEvents.dal>
  INCLUDE   <APIEvents.dal>

  DEFINE // Resource definitions for the closebox icons
     Hilite_cb   141,
     Plain_cb    142;

  CLASS

       Selectable( []Mv )( M^ ){
          EVENT WE_Hilite, WE_Dehilite;
          WE_Hilite <- []Mv | ~[];
          WE_Dehilite <- M^ | []~;
       },

       CloseBox : Selectable( )( ){
          // Note: this object exports event 'Close'
          DISPLAY TYPE DCIcon2,
               resource Plain_cb Hilite_cb,
               size 50 50,
               location 10 10,
               visible;
          Close <- []M^;
       };

  SYSTEM

       Root()(){
          EVENT ShowIt, PIPS_Stop, Close;
          DISPLAY TYPE DCWindow,
               size 300 500,
               location 10 50,
               visible;

          ShowButton : Selectable( )( ){
            DISPLAY TYPE DCButton,
                 size 50 50,
                 location 10 10,
                 visible,
                 name "Show";
            ShowIt <- []M^;
          }

          Terminate( 'COMMAND'v )( 'COMMAND'^ ){
            PIPS_Stop <- 'q'v;
          }
```

```
        DemoWindow( )( Close ){
          EVENT Close, WE_ShowWindow, WE_HideWindow;
          DISPLAY TYPE DCWindow,
               size 100 100,
               location 200 200,
               visible;
          ON DEACTIVATION{ WE_HideWindow; }
          WE_ShowWindow <- ShowIt;
          DemoCloseBox : CloseBox()(){ }
        }
      }


NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
Close                    EVENT          302
CloseBox                 CLASS
DCButton                 WIDGET
DCButton.location        WIDGET_ATTR    DD
DCButton.name            WIDGET_ATTR    S
DCButton.size            WIDGET_ATTR    DD
DCButton.visible         WIDGET_ATTR
DCIcon2                  WIDGET
DCIcon2.location         WIDGET_ATTR    DD
DCIcon2.resource         WIDGET_ATTR    DD
DCIcon2.size             WIDGET_ATTR    DD
DCIcon2.visible          WIDGET_ATTR
DCWindow                 WIDGET
DCWindow.location        WIDGET_ATTR    DD
DCWindow.size            WIDGET_ATTR    DD
DCWindow.visible         WIDGET_ATTR
DemoCloseBox             DIALOGUE
DemoWindow               DIALOGUE
Hilite_cb                MACRO          141
PIPS_Stop                EVENT          31
Plain_cb                 MACRO          142
Root                     DIALOGUE
Selectable               CLASS
ShowButton               DIALOGUE
ShowIt                   EVENT          303
Terminate                DIALOGUE
WE_Dehilite              EVENT          107
WE_HideWindow            EVENT          101
WE_Hilite                EVENT          106
WE_ShowWindow            EVENT          100
```

Trace

```
============ Closebox ============
A] 00721534 Root
A] 00721534 DemoWindow
A] 00721586 DemoCloseBox
D] 00721717                  DemoCloseBox
A] 00721759 DemoCloseBox
D] 00721772                  DemoCloseBox
D] 00721773                  DemoWindow
A] 00721795 DemoWindow
A] 00721829 ShowButton
D] 00721832                  ShowButton
A] 00721901 DemoCloseBox
D] 00721905                  DemoCloseBox
D] 00721906                  DemoWindow
A] 00721940 DemoWindow
A] 00721940 Terminate
```

3. SIMPLE PAINT DIALOGUE

This example is based on that described in section 5.9, and is to demonstrate:

i) The use of guard conditions.

ii) The support of concurrent dialogues offered by DAL.

In this example, the user is able to drag a paint brush around the screen. The trail of the paint brush is only visible when the mouse key is depressed. By depressing the cursor keys the thickness of the paint brush in both vertical and horizontal directions can be changed within the limits set by the guard conditions. This changing of the brush thickness can be carried out at the same time as moving the brush about the screen as is demonstrated by the screen shot displayed in Figure C.5.

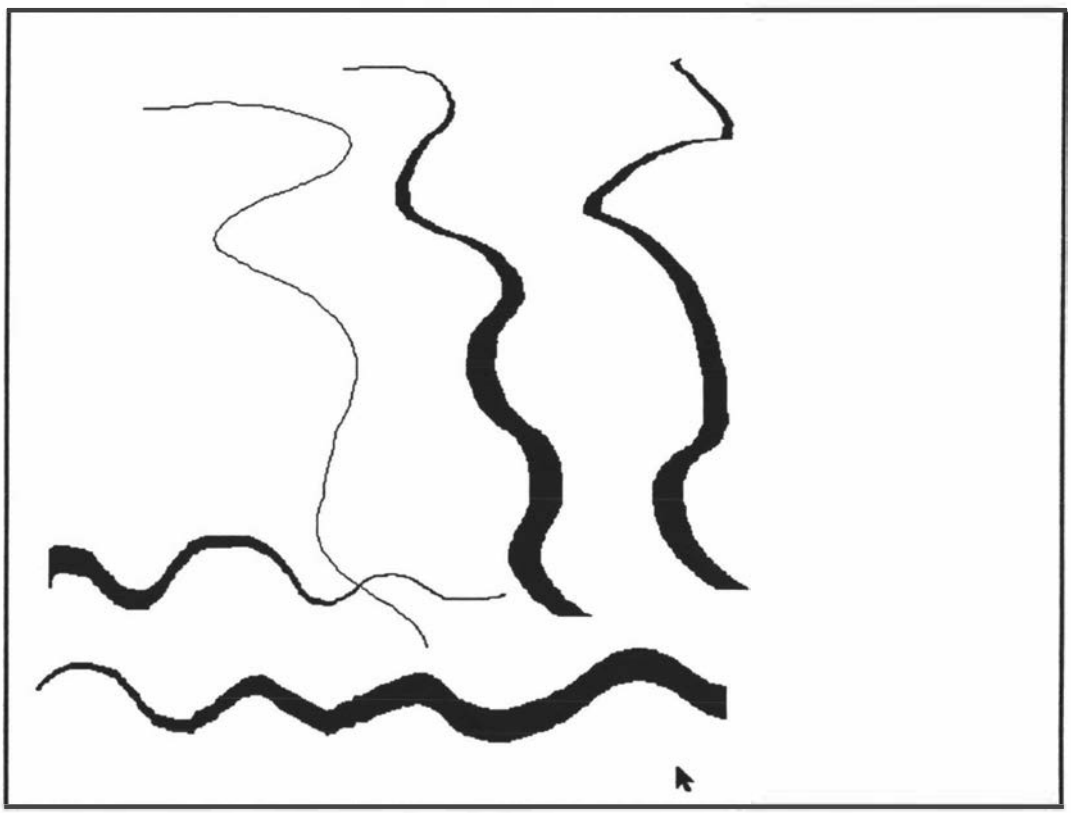


Figure C.5: Adjusting paint brush thickness concurrently with painting in the simple paint example.

Listing

```
Compiled by: Daltran2.06          Mon Jun 12 16:09:46 1995
Source File: Paint.dal

/*********************************************************\
*                        Paint.dal                        *
*                        =========                        *
* A very simple paint program in DAL. The user is able    *
* to move a paint brush around on the screen to 'paint'   *
* a line. The paint brush is only visible if the mouse    *
* key is depressed. The thickness of the paint brush      *
* can be varied in both vertical and horizontal           *
* directions by depressing the cursor keys.               *
*                                                         *
* This example demonstrates:                              *
* i)  The use of guard conditions. These restrict the     *
*     range through which the brush thickness can be      *
*     varied.                                             *
* ii) Concurrency. The paint brush thickness can be       *
*     varied at the same time as the paint brush is       *
*     moved.                                              *
*                                                         *
\*********************************************************/


INCLUDE   <Widgets.dal>
INCLUDE   <WidgetEvents.dal>

DEFINE          // Event codes for the cursor keys
   LeftArrow   28,
   RightArrow  29,
   UpArrow     30,
   DownArrow   31;

DEFINE          // Limits for brush thicknesses
   MinWidth    1,
   MaxWidth    20,
   MinHeight   1,
   MaxHeight   20;


SYSTEM

   // The window within which the user paints.
   PaintWindow()(){
      EVENT  DoPaint, brushMoved;
      VAR brushWidth, brushHeight;

      ON ACTIVATION{
        brushWidth=1;
        brushHeight=1;
      }

      DISPLAY   TYPE DCWindow,
           size 480 640,
           location 0 0,
           visible;
```

```
// Initiate the paint process by depressing the
// mouse key within the context of the painting
// window.
DoPaint[Ex,Ey] <- []Mv;

// Monitor brush movement from this level to ensure
// that it only happens if the cursor is within
// the context of the painting window.
brushMoved[Ex,Ey] <- []cursor_moved;

// command Q to exit
DoExit( 'COMMAND'v )( 'COMMAND'^ ){
  PIPS_Stop <- 'q'v;
}

// Control the paint brush. Start by depressing the
// mouse key with the cursor within the context of
// context of the paint window, finish by releasing
// the mouse key.
Mybrush( DoPaint )( M^ ){
  EVENT  WE_MoveTo, WE_Show, WE_Hide,
       WE_SetWidth, WE_SetHeight;

  DISPLAY TYPE DCPen;

  ON ACTIVATION{
    WE_MoveTo[Ex,Ey];
    WE_Show;
  }
  ON DEACTIVATION{
    WE_Hide;
  }

  // Move the paintbrush in response to the cursor
  // being moved within the context of the paint
  // window.
  WE_MoveTo[Ex,Ey] <- brushMoved;

  // Vary the thickness of the paint brush (within
  // limits), possibly concurrently with painting.
  Varybrush()(){

    IncreaseWidth [brushWidth<MaxWidth]
          ('RightArrow'v)('RightArrow'v){
      ON ACTIVATION{
        brushWidth = brushWidth + 1;
        WE_SetWidth[brushWidth];
      }
    }
    DecreaseWidth [brushWidth>MinWidth]
          ('LeftArrow'v)('LeftArrow'v){
      ON ACTIVATION{
        brushWidth = brushWidth - 1;
        WE_SetWidth[brushWidth];
      }
    }
```

```
            IncreaseHeight [brushHeight<MaxHeight]
                    ('UpArrow'v)('UpArrow'v){
               ON ACTIVATION{
                  brushHeight = brushHeight + 1;
                  WE_SetHeight[brushHeight];
               }
            }
            DecreaseHeight [brushHeight>MinHeight]
                    ('DownArrow'v)('DownArrow'v){
               ON ACTIVATION{
                  brushHeight = brushHeight - 1;
                  WE_SetHeight[brushHeight];
               }
            }
         }
      }
   }
```

```
NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
DCPen                       WIDGET
DCWindow                    WIDGET
DCWindow.location           WIDGET_ATTR     DD
DCWindow.size               WIDGET_ATTR     DD
DCWindow.visible            WIDGET_ATTR
DecreaseHeight              DIALOGUE
DecreaseWidth               DIALOGUE
DoExit                      DIALOGUE
DoPaint                     EVENT           300
DownArrow                   MACRO           31
IncreaseHeight              DIALOGUE
IncreaseWidth               DIALOGUE
LeftArrow                   MACRO           28
MaxHeight                   MACRO           20
MaxWidth                    MACRO           20
MinHeight                   MACRO           1
MinWidth                    MACRO           1
Mybrush                     DIALOGUE
PIPS_Stop                   MACRO           31
PaintWindow                 DIALOGUE
RightArrow                  MACRO           29
UpArrow                     MACRO           30
Varybrush                   DIALOGUE
WE_Hide                     EVENT           103
WE_MoveTo                   EVENT           104
WE_SetHeight                EVENT           110
WE_SetWidth                 EVENT           111
WE_Show                     EVENT           102
brushHeight                 VAR             303
brushMoved                  EVENT           301
brushWidth                  VAR             302
```

Trace

```
============ Paint ============
A] 00015957 PaintWindow
A] 00016254 Mybrush
A] 00016254 Varybrush
A] 00016269 IncreaseHeight
D] 00016269                    IncreaseHeight
A] 00016274 IncreaseHeight
D] 00016274                    IncreaseHeight
A] 00016280 IncreaseHeight
D] 00016280                    IncreaseHeight
A] 00016286 IncreaseHeight
D] 00016286                    IncreaseHeight
A] 00016292 IncreaseHeight
D] 00016292                    IncreaseHeight
A] 00016299 IncreaseHeight
D] 00016299                    IncreaseHeight
A] 00016304 IncreaseHeight
D] 00016304                    IncreaseHeight
A] 00016310 IncreaseHeight
D] 00016310                    IncreaseHeight
A] 00016316 IncreaseHeight
D] 00016316                    IncreaseHeight
A] 00016323 IncreaseHeight
D] 00016323                    IncreaseHeight
A] 00016328 IncreaseHeight
D] 00016328                    IncreaseHeight
A] 00016334 IncreaseHeight
D] 00016334                    IncreaseHeight
D] 00016340                    Mybrush
D] 00016340                    Varybrush
A] 00016429 Mybrush
A] 00016429 Varybrush
A] 00016431 DecreaseHeight
D] 00016431                    DecreaseHeight
A] 00016437 DecreaseHeight
D] 00016437                    DecreaseHeight
A] 00016443 DecreaseHeight
D] 00016443                    DecreaseHeight
A] 00016449 DecreaseHeight
D] 00016449                    DecreaseHeight
A] 00016455 DecreaseHeight
D] 00016455                    DecreaseHeight
A] 00016461 DecreaseHeight
D] 00016461                    DecreaseHeight
A] 00016467 DecreaseHeight
D] 00016467                    DecreaseHeight
A] 00016473 DecreaseHeight
D] 00016473                    DecreaseHeight
A] 00016479 DecreaseHeight
D] 00016479                    DecreaseHeight
A] 00016485 DecreaseHeight
D] 00016485                    DecreaseHeight
A] 00016491 DecreaseHeight
D] 00016491                    DecreaseHeight
A] 00016497 DecreaseHeight
D] 00016497                    DecreaseHeight
D] 00016501                    Mybrush
D] 00016501                    Varybrush
A] 00016616 Mybrush
```

```
A] 00016617 Varybrush
A] 00016643 IncreaseWidth
D] 00016643                 IncreaseWidth
A] 00016649 IncreaseWidth
D] 00016649                 IncreaseWidth
A] 00016655 IncreaseWidth
D] 00016655                 IncreaseWidth
A] 00016661 IncreaseWidth
D] 00016661                 IncreaseWidth
A] 00016668 IncreaseWidth
D] 00016668                 IncreaseWidth
A] 00016673 IncreaseWidth
D] 00016673                 IncreaseWidth
A] 00016679 IncreaseWidth
D] 00016679                 IncreaseWidth
A] 00016686 IncreaseWidth
D] 00016686                 IncreaseWidth
A] 00016691 IncreaseWidth
D] 00016692                 IncreaseWidth
A] 00016698 IncreaseWidth
D] 00016698                 IncreaseWidth
A] 00016703 IncreaseWidth
D] 00016703                 IncreaseWidth
A] 00016709 IncreaseWidth
D] 00016711                 IncreaseWidth
A] 00016715 IncreaseWidth
D] 00016715                 IncreaseWidth
A] 00016722 IncreaseWidth
D] 00016722                 IncreaseWidth
A] 00016727 IncreaseWidth
D] 00016727                 IncreaseWidth
A] 00016733 IncreaseWidth
D] 00016733                 IncreaseWidth
A] 00016739 IncreaseWidth
D] 00016739                 IncreaseWidth
A] 00016745 IncreaseWidth
D] 00016745                 IncreaseWidth
A] 00016751 IncreaseWidth
D] 00016751                 IncreaseWidth
D] 00016758                 Mybrush
D] 00016758                 Varybrush
A] 00016976 Mybrush
A] 00016978 Varybrush
A] 00016992 DecreaseWidth
D] 00016992                 DecreaseWidth
A] 00016997 DecreaseWidth
D] 00016997                 DecreaseWidth
A] 00017003 DecreaseWidth
D] 00017003                 DecreaseWidth
A] 00017009 DecreaseWidth
D] 00017009                 DecreaseWidth
A] 00017015 DecreaseWidth
D] 00017015                 DecreaseWidth
A] 00017021 DecreaseWidth
D] 00017022                 DecreaseWidth
A] 00017027 DecreaseWidth
D] 00017028                 DecreaseWidth
A] 00017034 DecreaseWidth
D] 00017034                 DecreaseWidth
A] 00017040 DecreaseWidth
```

```
D] 00017040                  DecreaseWidth
A] 00017045 DecreaseWidth
D] 00017045                  DecreaseWidth
A] 00017052 DecreaseWidth
D] 00017052                  DecreaseWidth
A] 00017057 DecreaseWidth
D] 00017057                  DecreaseWidth
D] 00017059                  Mybrush
D] 00017059                  Varybrush
A] 00017131 DoExit
```

4. PULLDOWN MENU

The example demonstrates the use of inheritance in dialogue classes to define the behaviour of a hierarchical menu system, as described in Chapter 5. Selection of an item, causes the name of the item to be displayed in a small window at the top right of the screen. Figure C.6 shows this system in operation.

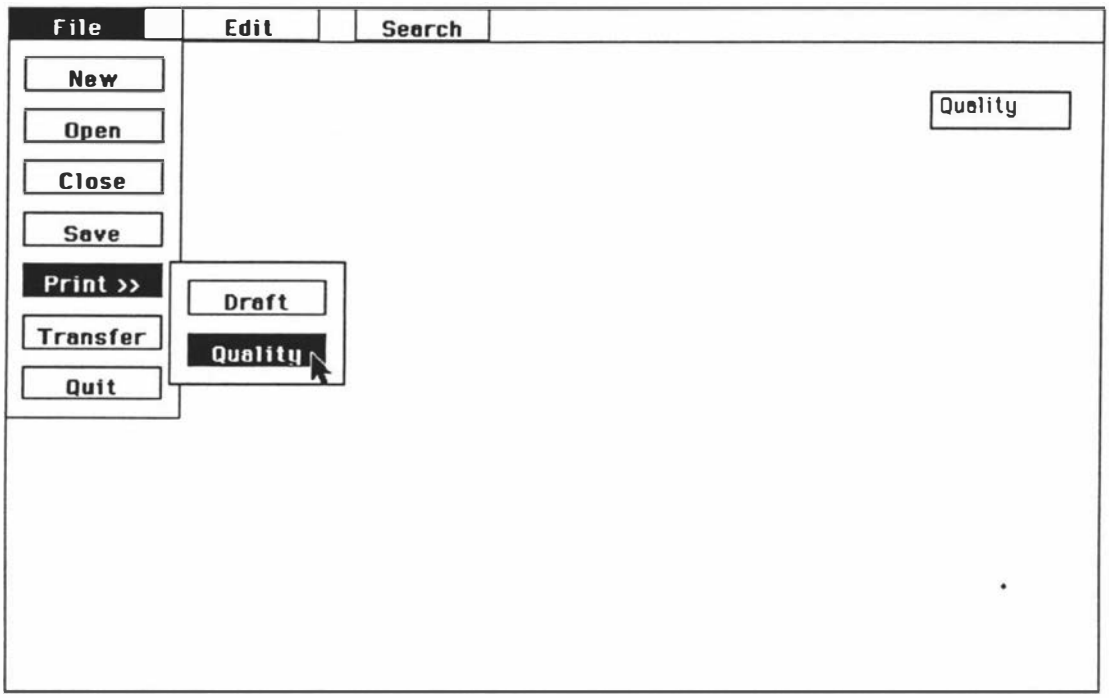


Figure C.6: Screen dump of Pulldown menu.

Listing

```
Compiled by: Daltran2.06                    Wed Feb  1 14:12:17 1995
Source File: PullDown.dal

  /*************************************************\
  *                   PullDown.dal                  *
  *                                                 *
  *   A sequence of pull-down menus from a single   *
  *   menu bar. One menu also has a hierarchical    *
  *   sub-menu. This example demonstrates the use   *
  *   of derived classes in defining the behaviour  *
  *   of both menu items and menu headers.          *
  *                                                 *
  \*************************************************/

  INCLUDE   <Widgets.dal>
  INCLUDE   <WidgetEvents.dal>
  INCLUDE   <APIEvents.dal>

  CLASS


    /*
     *   A generic menu item. It highlights on dragging
     *   the cursor into its context. Hilite and dehilite
     *   events are declared local to this object type.
     */
    menu_item( ~[] )( ){
      EVENT WE_Hilite, WE_Dehilite;
      ON ACTIVATION{ WE_Hilite; }
      ON DEACTIVATION{ WE_Dehilite; }
    },

    /*
     *   A menu header behaves exactly like a menu item. In
     *   addition depressing the mouse key in its context
     *   will also cause it to activate (and hence highlight).
     *   Events to activate/deactivate the sub-menu are generated
     *   at and are local to this level.
     */
    menu_header:menu_item( []Mv )(){
      EVENT ShowSub, HideSub;
      ON ACTIVATION{ ShowSub; }
      ON DEACTIVATION{ HideSub; }
    },

    /*
     *   A selectable menu item behaves exactly like a menu item.
     *   In addition dragging out of its context will also cause
     *   it to deactivate. Releasing the mouse key within its
     *   context will produces a selection event to be generated
     *   (local to this level).
     */
    selectable:menu_item()( []~ ){
      EVENT API_Select;
      APPLICATION ACSelector;
      API_Select[0] <- []M^;
    },
```

```
    /*
     *   A menu is activated by the show_sub and hide_sub events
     *   generated by menu_headers. On activation it generates a
     *   show event, causing the associated menu to be displayed.
     *   Conversely, on deactivation it generates a hide event in
     *   order to hide the associated menu.
     */
    menu( ShowSub )( HideSub ){
       EVENT WE_ShowWindow, WE_HideWindow;
       ON ACTIVATION{ WE_ShowWindow; }
       ON DEACTIVATION{ WE_HideWindow; }
    };

SYSTEM

   Desktop()(){
      EVENT popup, close_menu, API_DoWrite;
      DISPLAY type DCWindow,
           size 480 640,
           location 0 0,
           visible;

        // command Q to exit
      doExit( 'COMMAND'v )( 'COMMAND'^ ){
        PIPS_Stop <- 'q'v;
      }

      monitorAPI()(){
        DISPLAY type DCWindow,
             size 20 80,
             location 540 50,
             visible;
        APPLICATION ACScribe;
      }

      menu_Bar( []Mv )( M^){
        DISPLAY type DCWindow,
             location 0 0,
             size 20 640,
             visible;

        grouping ME;

        File_Menu:menu_header()(){
          DISPLAY type DCButton,
               location 0 0,
               size 20 80,
               name File,
               visible;

          FileMenu:menu()(){
             DISPLAY type DCWindow,
                  location 0 20,
                  size 220 100;

             // This group has to be ME since the print item is
             // a header to a sub-menu, and if it was to deactivate
             // on moving out of its context then we wouldn't be
             // able to select from its sub-menu.
```

```
grouping ME;

New:selectable()(){
  DISPLAY type DCButton,
       location 10 10,
       size 20 80,
       name New,
       visible;
}
Open:selectable()(){
  DISPLAY type DCButton,
       location 10 40,
       size 20 80,
       name Open,
       visible;
}
Close:selectable()(){
  DISPLAY type DCButton,
       location 10 70,
       size 20 80,
       name Close,
       visible;
}
Save:selectable()(){
  DISPLAY type DCButton,
       location 10 100,
       size 20 80,
       name Save,
       visible;
}
Print:menu_header()(){
  DISPLAY type DCButton,
       location 10 130,
       size 20 80,
       name "Print >>",
       visible;

  Print_menu:menu()(){
    DISPLAY type DCWindow,
         location 95 150,
         size 70 100;

    Draft:selectable()(){
      DISPLAY type DCButton,
           location 10 10,
           size 20 80,
           name Draft,
           visible;
    }
    Quality:selectable()(){
      DISPLAY type DCButton,
           location 10 40,
           size 20 80,
           name Quality,
           visible;
    }
  }

}
```

```
      Transfer:selectable()(){
        DISPLAY type DCButton,
             location 10 160,
             size 20 80,
             name Transfer,
             visible;
      }
      Quit:selectable()(){
        DISPLAY type DCButton,
             location 10 190,
             size 20 80,
             name Quit,
             visible;
        PIPS_Stop <- []M^;
      }
    }
  }

  Edit_Menu:menu_header()(){
    DISPLAY type DCButton,
         location 100 0,
         size 20 80,
         name Edit,
         visible;

    EditMenu:menu()(){
      DISPLAY type DCWindow,
           location 100 20,
           size 190 100;

      Undo:selectable()(){
        DISPLAY type DCButton,
             location 10 10,
             size 20 80,
             name Undo,
             visible;
      }
      Paste:selectable()(){
        DISPLAY type DCButton,
             location 10 40,
             size 20 80,
             name Paste,
             visible;
      }
      Select:selectable()(){
        DISPLAY type DCButton,
             location 10 70,
             size 20 80,
             name Select,
             visible;
      }
      Set_font:selectable()(){
        DISPLAY type DCButton,
             location 10 100,
             size 20 80,
             name Set_Font,
             visible;
      }
```

```
            Balance:selectable()(){
              DISPLAY type DCButton,
                   location 10 130,
                   size 20 80,
                   name Balance,
                   visible;
            }
            Options:selectable()(){
              DISPLAY type DCButton,
                   location 10 160,
                   size 20 80,
                   name Options,
                   visible;
            }
          }
        }
        Search_menu:menu_header()(){
          DISPLAY type DCButton,
               location 200 0,
               size 20 80,
               name Search,
               visible;

          SearchMenu:menu()(){
            DISPLAY type DCWindow,
                 location 200 20,
                 size 140 100;

            Find:selectable()(){
              DISPLAY type DCButton,
                   location 10 10,
                   size 20 80,
                   name Find,
                   visible;
            }
            Replace:selectable()(){
              DISPLAY type DCButton,
                   location 10 40,
                   size 20 80,
                   name Replace,
                   visible;
            }
            Goto:selectable()(){
              DISPLAY type DCButton,
                   location 10 70,
                   size 20 80,
                   name Goto,
                   visible;
            }
            Mark:selectable()(){
              DISPLAY type DCButton,
                   location 10 100,
                   size 20 80,
                   name Mark,
                   visible;
            }
          }
        }
      }
    }
```

```
NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
API_DoWrite                EVENT          201
API_Select                 EVENT          200
Balance                    DIALOGUE
Close                      DIALOGUE
DCButton                   WIDGET
DCButton.location          WIDGET_ATTR    DD
DCButton.name              WIDGET_ATTR    S
DCButton.size              WIDGET_ATTR    DD
DCButton.visible           WIDGET_ATTR
DCWindow                   WIDGET
DCWindow.location          WIDGET_ATTR    DD
DCWindow.size              WIDGET_ATTR    DD
DCWindow.visible           WIDGET_ATTR
Desktop                    DIALOGUE
Draft                      DIALOGUE
EditMenu                   DIALOGUE
Edit_Menu                  DIALOGUE
FileMenu                   DIALOGUE
File_Menu                  DIALOGUE
Find                       DIALOGUE
Goto                       DIALOGUE
HideSub                    EVENT          303
Mark                       DIALOGUE
New                        DIALOGUE
Open                       DIALOGUE
Options                    DIALOGUE
PIPS_Stop                  MACRO          31
Paste                      DIALOGUE
Print                      DIALOGUE
Print_menu                 DIALOGUE
Quality                    DIALOGUE
Quit                       DIALOGUE
Replace                    DIALOGUE
Save                       DIALOGUE
SearchMenu                 DIALOGUE
Search_menu                DIALOGUE
Select                     DIALOGUE
Set_font                   DIALOGUE
ShowSub                    EVENT          302
Transfer                   DIALOGUE
Undo                       DIALOGUE
WE_Dehilite                EVENT          107
WE_HideWindow              EVENT          101
WE_Hilite                  EVENT          106
WE_ShowWindow              EVENT          100
close_menu                 EVENT          310
doExit                     DIALOGUE
menu                       CLASS
menu_Bar                   DIALOGUE
menu_header                CLASS
menu_item                  CLASS
monitorAPI                 DIALOGUE
```

popup	EVENT	309
selectable	CLASS	

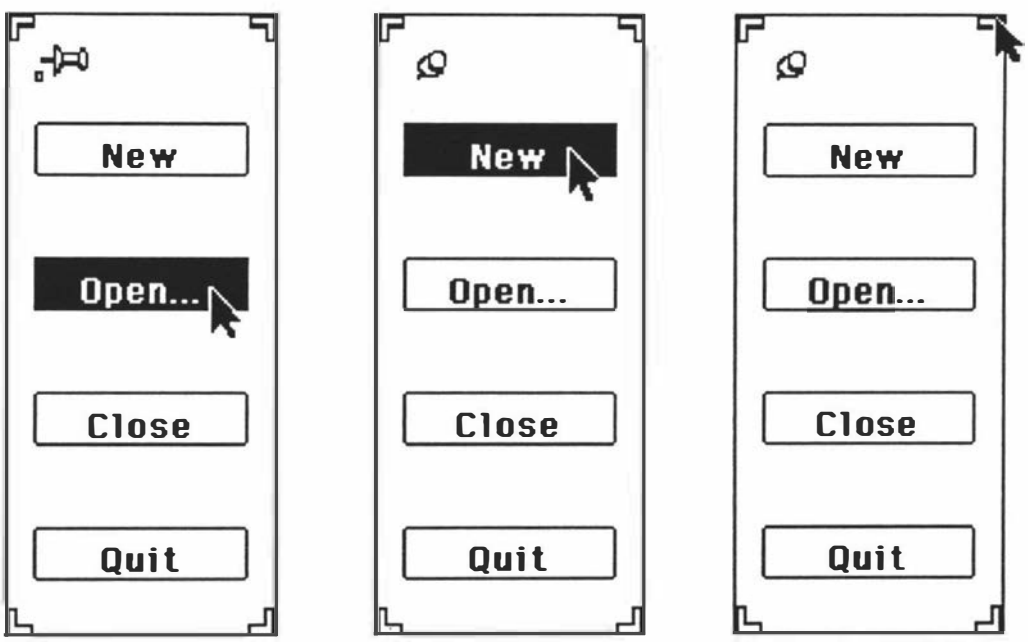
Trace

```
============ PullDown ============
A] 00754647 Desktop
A] 00754647 monitorAPI
A] 00754719 menu_Bar
A] 00754719 File_Menu
A] 00754720 FileMenu
A] 00754760 New
D] 00754765                  New
A] 00754769 Open
D] 00754831                  Open
A] 00754851 Close
D] 00754883                  Close
D] 00754928                  menu_Bar
D] 00754928                  File_Menu
D] 00754928                  FileMenu
A] 00755010 menu_Bar
A] 00755010 Edit_Menu
A] 00755011 EditMenu
A] 00755051 Undo
D] 00755055                  Undo
A] 00755070 Paste
D] 00755094                  menu_Bar
D] 00755094                  Edit_Menu
D] 00755094                  EditMenu
D] 00755094                  Paste
A] 00755158 menu_Bar
A] 00755158 File_Menu
A] 00755159 FileMenu
A] 00755189 New
D] 00755192                  New
A] 00755194 Open
D] 00755198                  Open
A] 00755213 Close
D] 00755216                  Close
A] 00755220 Save
D] 00755247                  Save
D] 00755284                  File_Menu
D] 00755285                  FileMenu
A] 00755285 Edit_Menu
A] 00755289 EditMenu
D] 00755339                  Edit_Menu
D] 00755339                  EditMenu
A] 00755339 Search_menu
A] 00755344 SearchMenu
A] 00755382 Find
D] 00755399                  Find
A] 00755401 Replace
D] 00755429                  Replace
A] 00755519 Mark
D] 00755536                  Mark
A] 00755537 Goto
D] 00755576                  menu_Bar
D] 00755576                  Search_menu
D] 00755576                  SearchMenu
D] 00755576                  Goto
A] 00755640 menu_Bar
A] 00755641 File_Menu
A] 00755642 FileMenu
A] 00755657 New
```

```
D] 00755659                    New
A] 00755659 Close
D] 00755663                    Close
A] 00755663 Print
A] 00755665 Print_menu
A] 00755697 Draft
D] 00755704                    Draft
A] 00755724 Quality
D] 00755735                    Quality
A] 00755759 Quality
D] 00755789                    menu_Bar
D] 00755789                    File_Menu
D] 00755790                    FileMenu
D] 00755790                    Print
D] 00755790                    Print_menu
D] 00755790                    Quality
A] 00755843 menu_Bar
A] 00755843 File_Menu
A] 00755844 FileMenu
A] 00755863 New
D] 00755865                    New
A] 00755870 Print
A] 00755871 Print_menu
D] 00755881                    Print
D] 00755881                    Print_menu
A] 00755881 Transfer
D] 00755900                    Transfer
A] 00755902 Quit
D] 00755931                    menu_Bar
D] 00755931                    File_Menu
D] 00755932                    FileMenu
D] 00755932                    Quit
```

5. PIN-UP POP-UP MENU

This example is a pin-up, pop-up menu, exactly as described in Chapter 8. Depressing the shift key, and depressing the mouse key causes the menu window to be displayed. Dragging into the context of a menu option causes the option to become highlighted (Figure C.7(a)), releasing the mouse key causes the option to be selected, and displayed in a small window in the top right of the screen. Making a selection in this way causes the menu window to become hidden following the selection. Releasing the mouse key with the cursor over the pin (at the top left of the menu), causes the pin icon to change, showing the menu is pinned up, and the menu to stay up on the screen (Figure C.7(b)). With the menu pinned to the screen, the menu window can be moved around by dragging it using the drag handles located at each corner (Figure C.7(c)). Finally, options on the menu can be selected through the keyboard by depressing the command key and keys 1, 2, 3 or 4.



(a) (b) (c)

Figure C.7: The pin-up pop-up menu: (a) making a selection, (b) making a selection from the pinned up menu and (c) dragging the menu by one of its drag handles.

Listing

```
Compiled by: Daltran2.06                    Wed Feb  1 14:19:48 1995
Source File: Popup.dal

 /************************************************\
 *                    Popup.dal                   *
 *                                                *
 *   A pinnable popup menu, that can be dragged   *
 *   around on the screen. Selection of menu items *
 *   can be done by:                              *
 *                                                *
 *   1) Popping up the menu and dragging to and   *
 *      releasing the mouse key over an item.     *
 *   2) Depressing the mouse key within the       *
 *      context of the menu and releasing it with *
 *      the cursor over an item.                  *
 *   3) With the menu pinned up, pressing 'COMMAND'*
 *      followed by keys 1..4 to select item 1 to *
 *      4 respectively.                           *
 *                                                *
 \************************************************/

 INCLUDE  <Widgets.dal>
 INCLUDE  <WidgetEvents.dal>
 INCLUDE  <APIEvents.dal>

 DEFINE
   pin_in    WE_Hilite,
   pin_out     WE_Dehilite;

 DEFINE                  // icon resources
   pin_in_icon    135,      // ....the pin
   pin_out_icon   136,

   tl_dh        137,     // ....drag handles
   tr_dh        138,
   bl_dh        140,
   br_dh        139;


 CLASS

   menu_item( []Mv, ~[] )( []~ ){
     EVENT WE_Hilite, WE_Dehilite, API_Select;
     APPLICATION ACSelector;
     ON ACTIVATION{ WE_Hilite; }
     ON DEACTIVATION{ WE_Dehilite; }
     API_Select <- []M^;
   },

   drag_handle( []Mv )( M^ ){
     // Mask the drag handles from move events, so
     // that those sent to the menu window do not
     // move the handles as well.
     EVENT WE_MoveTo, WE_MoveBy;

     drag_menu[Ea,Eb] <- cursor_moved;
   };
```

```
SYSTEM

   desktop()(){
      EVENT API_DoWrite;
      DISPLAY type DCWindow,
           size 480 640,
           location 0 0,
           visible;

        // command Q to exit
      do_exit( 'COMMAND'v )( 'COMMAND'^ ){
        PIPS_Stop <- 'q'v;
      }

      /*
       *   Record the text of the selection in a separate window.
       */
      monitor()(){
        DISPLAY type DCWindow,
             size 20 80,
             location 540 25,
             visible;
        APPLICATION ACScribe;
      }

      /*
       *   We need to depress the shift key in order to be able
       *   to activate the menu by depressing the mouse key. The
       *   menu will 'popup' at the cursor location.
       */
      activate_popup( 'SHIFT'v )( close_menu ){
        EVENT popup, close_menu;
        ON ACTIVATION{ popup; }
        GROUPING ME;

        undo_popup()( 'SHIFT'^ ){
          close_menu <- 'SHIFT'^;
        }

        menu_window( Mv )(){
          EVENT   WE_ShowWindow, WE_HideWindow,
                WE_MoveTo, WE_MoveBy;
          EVENT   drag_menu, do_select;
          DISPLAY type DCWindow,
                location 150 100,
                size 230 100;

          ON ACTIVATION {  WE_MoveTo[Ex,Ey];
                      WE_ShowWindow;
                    }
          ON DEACTIVATION{ WE_HideWindow; }

          do_select <- Mv;

          WE_MoveBy[Ex,Ey] <- drag_menu;
```

```
/*
 *   The default close action is to release the mouse
 *   key.If however we release the mouse key within the
 *   context of the pushpin, then the close action from
 *   then on is to depress the mouse key within the
 *   context of the pin.
 */
close_dialogues()(){
   GROUPING ME;

   default()(){
      close_menu <- M^;
   }
   pinned( []M^ )( []Mv ){
      // Protect the pin from move events
      EVENT  WE_MoveTo, WE_MoveBy,
           pin_in, pin_out;
      DISPLAY type DCIcon2,
           resource 136 135,
           visible,
           size 20 20,
           location 10 10;
      ON ACTIVATION{ pin_in; }
      ON DEACTIVATION{ pin_out; close_menu; }
   }
}

/*
 *   A drag handle is located at each corner of the menu.
 *   Depressing the mouse key within the context of a
 *   drag handle, and dragging causes the menu window to
 *   move by the same amount as the cursor.
 */
drag()(){
   tl:drag_handle()(){
      DISPLAY type DCIcon1,
           resource 137,
           visible,
           size 10 10,
           location 0 0;
   }
   tr:drag_handle()(){
      DISPLAY type DCIcon1,
           resource 138,
           visible,
           size 10 10,
           location 90 0;
   }
   bl:drag_handle()(){
      DISPLAY type DCIcon1,
           resource 140,
           visible,
           size 10 10,
           location 0 220;
   }
```

```
    br:drag_handle()(){
      DISPLAY type DCIcon1,
           resource 139,
           visible,
           size 10 10,
           location 90 220;
    }
  }

  /*
   *   The menu is active when 'popup up', or the mouse
   *   key is depressed with the cursor within the context
   *   of the window, or the COMMAND key is depressed.
   */
  menu( Mv, do_select, 'COMMAND'v )( M^, 'COMMAND'^ ){

    /*
     *   Menu items hilite on dragging the cursor into
     *   their context, and dehilite on dragging out of
     *   their context. Releasing the mouse key within the
     *   context of a menu item causes a select event to be
     *   generated. In additionto selection by mouse, each
     *   menu item can be selected using the keyboard
     *   keys 1..4
     */
    Item_New:menu_item('1'v)('1'v){
      DISPLAY type DCButton,
           visible,
           location 10 40,
           size 20 80,
           name "New";
      API_Select <- '1'v;
    }
    Item_Open:menu_item('2'v)('2'v){
      DISPLAY type DCButton,
           visible,
           location 10 90,
           size 20 80,
           name "Open...";
      API_Select <- '2'v;
    }
    Item_Close:menu_item('3'v)('3'v){
      DISPLAY type DCButton,
           visible,
           location 10 140,
           size 20 80,
           name "Close";
      API_Select <- '3'v;
    }
    Item_Quit:menu_item('4'v)('4'v){
      DISPLAY type DCButton,
           visible,
           location 10 190,
           size 20 80,
           name "Quit";
      API_Select <- '4'v;
      PIPS_Stop <- []M^|'4'v;
    }
  }
}
```

```
        }
    }


NO ERRORS

WARNING: Event (close_menu) not declared - assumed global
No potential event cycles found

=============== SYMBOL TABLE ===============
API_DoWrite                 EVENT           201
API_Select                  EVENT           200
DCButton                    WIDGET
DCButton.location           WIDGET_ATTR     DD
DCButton.name               WIDGET_ATTR     S
DCButton.size               WIDGET_ATTR     DD
DCButton.visible            WIDGET_ATTR
DCIcon1                     WIDGET
DCIcon1.location            WIDGET_ATTR     DD
DCIcon1.resource            WIDGET_ATTR     D
DCIcon1.size                WIDGET_ATTR     DD
DCIcon1.visible             WIDGET_ATTR
DCIcon2                     WIDGET
DCIcon2.location            WIDGET_ATTR     DD
DCIcon2.resource            WIDGET_ATTR     DD
DCIcon2.size                WIDGET_ATTR     DD
DCIcon2.visible             WIDGET_ATTR
DCWindow                    WIDGET
DCWindow.location           WIDGET_ATTR     DD
DCWindow.size               WIDGET_ATTR     DD
DCWindow.visible            WIDGET_ATTR
Item_Close                  DIALOGUE
Item_New                    DIALOGUE
Item_Open                   DIALOGUE
Item_Quit                   DIALOGUE
PIPS_Stop                   MACRO           31
WE_Dehilite                 EVENT           107
WE_HideWindow               EVENT           101
WE_Hilite                   EVENT           106
WE_MoveBy                   EVENT           105
WE_MoveTo                   EVENT           104
WE_ShowWindow               EVENT           100
activate_popup              DIALOGUE
bl                          DIALOGUE
bl_dh                       MACRO           140
br                          DIALOGUE
br_dh                       MACRO           139
close_dialogues             DIALOGUE
close_menu                  EVENT           307
default                     DIALOGUE
desktop                     DIALOGUE
do_exit                     DIALOGUE
do_select                   EVENT           315
drag                        DIALOGUE
drag_handle                 CLASS
drag_menu                   EVENT           305
menu                        DIALOGUE
menu_item                   CLASS
```

menu_window	DIALOGUE	
monitor	DIALOGUE	
pin_in	EVENT	106
pin_in_icon	MACRO	135
pin_out	EVENT	107
pin_out_icon	MACRO	136
pinned	DIALOGUE	
popup	EVENT	308
tl	DIALOGUE	
tl_dh	MACRO	137
tr	DIALOGUE	
tr_dh	MACRO	138
undo_popup	DIALOGUE	

Trace

```
============ Popup ============
A] 00780350 desktop
A] 00780350 monitor
A] 00780413 activate_popup
A] 00780413 undo_popup
D] 00780425                    undo_popup
A] 00780425 menu_window
A] 00780426 menu
A] 00780426 drag
A] 00780426 close_dialogues
A] 00780426 default
A] 00780468 Item_New
D] 00780490                    Item_New
A] 00780502 Item_Open
D] 00780531                    menu
D] 00780531                    Item_Open
D] 00780532                    activate_popup
D] 00780532                    menu_window
D] 00780532                    drag
D] 00780532                    close_dialogues
D] 00780533                    default
A] 00780604 activate_popup
A] 00780604 undo_popup
D] 00780613                    undo_popup
A] 00780613 menu_window
A] 00780613 menu
A] 00780613 drag
A] 00780613 close_dialogues
A] 00780613 default
D] 00780669                    menu
D] 00780669                    default
A] 00780669 pinned
A] 00780753 menu
A] 00780753 Item_New
D] 00780759                    menu
D] 00780759                    Item_New
A] 00780828 menu
A] 00780828 Item_Close
D] 00780833                    menu
D] 00780833                    Item_Close
A] 00780978 menu
A] 00780979 tr
D] 00781047                    menu
D] 00781048                    tr
A] 00781140 menu
A] 00781140 bl
D] 00781213                    menu
D] 00781213                    bl
A] 00781292 menu
D] 00781293                    pinned
A] 00781293 default
D] 00781293                    activate_popup
D] 00781294                    menu_window
D] 00781294                    menu
D] 00781294                    drag
D] 00781294                    close_dialogues
D] 00781294                    default
A] 00781360 activate_popup
A] 00781360 undo_popup
```

```
D] 00781374                    undo_popup
A] 00781374 menu_window
A] 00781374 menu
A] 00781374 drag
A] 00781374 close_dialogues
A] 00781374 default
D] 00781425                    menu
D] 00781425                    default
A] 00781425 pinned
A] 00781479 menu
A] 00781479 Item_Quit
D] 00781482                    menu
D] 00781482                    Item_Quit
```

6. BRUSH SHAPE DIALOGUE BOX

This example is an implementation of the brush shape dialogue used in Chapters 2 and 3, and further discussed in Chapter 8. Figure C.8 shows this system in operation.

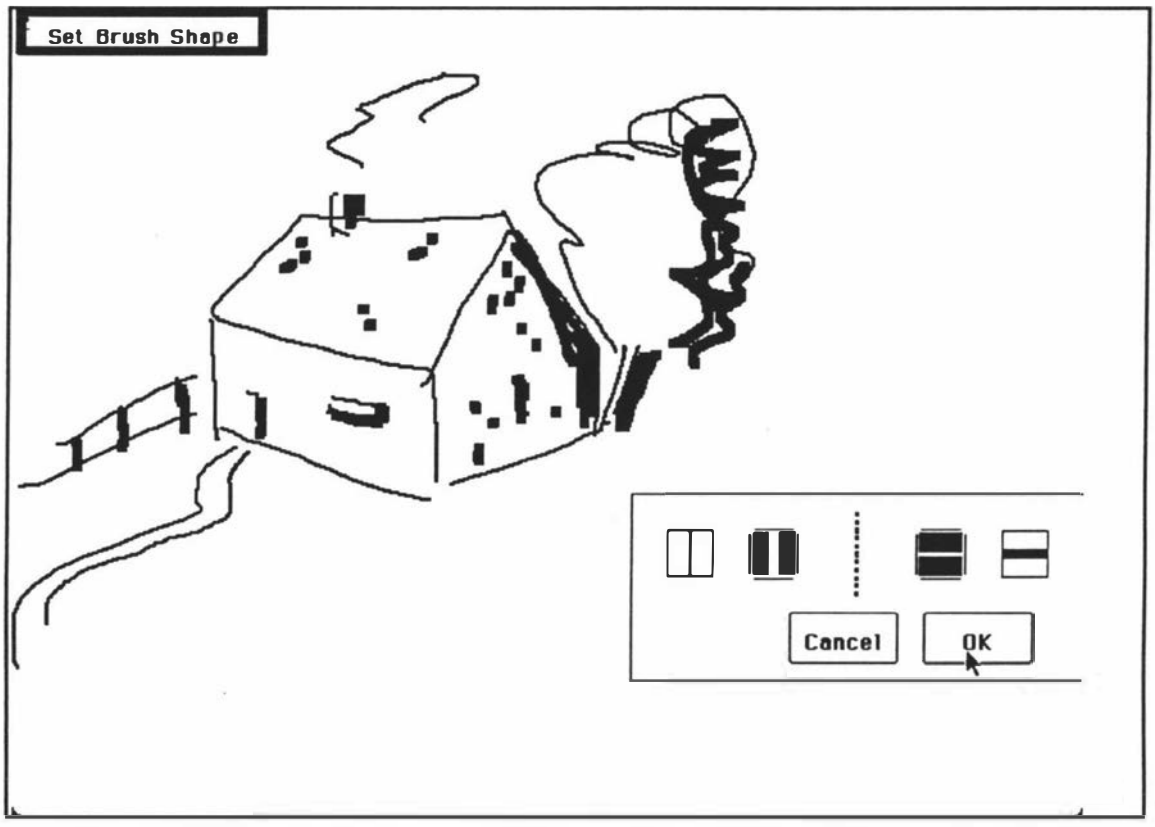


Figure C.8: The brush shape dialogue in operation. The user has just selected vertical thick and horizontal thin options.

Listing

```
Compiled by: Daltran2.06          Wed May 10 18:05:58 1995
Source File: BrushShape.dal

  /************************************************\
  *                 BrushShape.dal                 *
  *                 ==============                 *
  * A simple paint program in which the vertical and *
  * horizontal thicknesses of the paint brush can be *
  * set within a dialogue box. This example is the   *
  * same as that discussed in Chapter 2.             *
  \************************************************/
  INCLUDE  <Widgets.dal>
  INCLUDE  <WidgetEvents.dal>

  DEFINE
     VThinIcon   155,   // Icon resource numbers
     VThickIcon  156,
     HThinIcon   157,
     HThickIcon  158,

     ThinLine  2,     // Supported line thicknesses
     ThickLine   6,

     VDashIcon   159,   // Resource number for a dashed icon
                 // used to separate the ME groups

     IconSize  30;    // Size for icon buttons

  CLASS

     Selectable( []Mv )(){
       EVENT WE_Hilite, WE_Dehilite;
       ON ACTIVATION{ WE_Hilite; }
       ON DEACTIVATION{ WE_Dehilite; }
     },

     Button:Selectable()( M^ ){
       WE_Hilite <- -[];
       WE_Dehilite <- []-;
     };

  SYSTEM

     PaintSystem()(){
       EVENT  DoDB,
            DoneDB,
            SelectShape,
            WE_SetHeight,
            WE_SetWidth;

       ON ACTIVATION{
         WE_SetHeight[ThinLine];
         WE_SetWidth[ThinLine];
       }
```

```
/*   The dialogue box and the drawing window
 *   form a mutually exclusive set. Whilst
 *   interacting with a dialogue box, the user
 *   is prevented for interacting with any
 *   other dialogue.
 */
GROUPING ME;

DialogueBoxes(DoDB)(DoneDB){
  /*
   *   The dialogue box, Composed of 2 groups of
   *   icon buttons for setting horizontal and
   *   vertical brush line thicknesses.
   *   The event 'SetDefault' causes the dialogues
   *   associated with the default values to
   *   activate and become highlighted.
   */
  SelectionBox(SelectShape)(){
    DISPLAY   TYPE DCWindow,
         size 110 270,
         location 370 290;

    EVENT   WE_ShowWindow,
         WE_HideWindow,
         SetDefault;

    VAR VThickness, HThickness;

    ON ACTIVATION{
       WE_ShowWindow;
       VThickness=ThinLine;
       HThickness=ThinLine;
       SetDefault;
    }
    ON DEACTIVATION{ WE_HideWindow; }

    VSelect()(){
       GROUPING ME;

       ON ACTIVATION{
         SetDefault;
       }

       ThinSelect:Selectable(SetDefault)(){
         DISPLAY TYPE DCIcon1,
              resource VThinIcon,
              size IconSize IconSize,
              location 20 20,
              visible;
         ON ACTIVATION{ VThickness=ThinLine; }
       }
       ThickSelect:Selectable()(){
         DISPLAY TYPE DCIcon1,
              resource VThickIcon,
              size IconSize IconSize,
              location 70 20,
              visible;
         ON ACTIVATION{ VThickness=ThickLine; }
       }
    }
```

```
/*
 *   A dummy divider to separate the dialog box
 *   into the two ME groups.
 */
Divider()(){
   DISPLAY TYPE DCIcon1,
        resource VDashIcon,
        size 50 30,
        location 120 10,
        visible;
}

HSelect()(){
   GROUPING ME;

   ON ACTIVATION{
      SetDefault;
   }

   ThinSelect:Selectable(SetDefault)(){
      DISPLAY TYPE DCIcon1,
           resource HThinIcon,
           size IconSize IconSize,
           location 170 20,
           visible;
      ON ACTIVATION{ HThickness=ThinLine; }
   }
   ThickSelect:Selectable()(){
      DISPLAY TYPE DCIcon1,
           resource HThickIcon,
           size IconSize IconSize,
           location 220 20,
           visible;
      ON ACTIVATION{ HThickness=ThickLine; }
   }
}

/*
 *   Cancel and OK buttons.
 */
Done()(){
     Cancel:Button()(){
       DISPLAY TYPE DCButton,
            size 30 65,
            location 95 70,
            visible,
            name "Cancel";
       DoneDB <- []M^;
     }
     OK:Button()(){
       DISPLAY TYPE DCButton,
            size 30 65,
            location 175 70,
            visible,
            name "OK";

       WE_SetHeight[HThickness] <- []M^;
       WE_SetWidth[VThickness] <- []M^;
       DoneDB <- []M^;
   }
```

```
    }
  }
}

DrawWindow()(){
  EVENT  DoDraw,
         PenMoved;

  DISPLAY  TYPE DCWindow,
         size 480 640,
         location 0 0,
         visible;

  DoDraw[Ex,Ey] <- []Mv;
  PenMoved[Ex,Ey] <- []cursor_moved;

  /*
   *   Command Q to exit
   */
  DoExit( 'COMMAND'v )( 'COMMAND'^ ){
    PIPS_Stop <- 'q'v;
  }

  /*
   *   Depressing this button causes the dialog box
   *   to become visible.
   */
  ChangePen:Button()(){
    DISPLAY TYPE DCButton,
          size 30 150,
          location 0 0,
          visible,
          name "Set Brush Shape";
    DoDB <- []M^;
    SelectShape <- []M^;
  }

  /*
   *   The pen widget. It is activated by
   *   depressing the mouse key with the
   *   cursor in the context of the
   *   drawing window.
   */
  MyPen( DoDraw )( M^ ){
    EVENT  WE_MoveTo,
          WE_Show,
          WE_Hide;

    DISPLAY TYPE DCPen;

    ON ACTIVATION{
       WE_MoveTo[Ex,Ey];
       WE_Show;
    }
    ON DEACTIVATION{
       WE_Hide;
    }

    WE_MoveTo[Ex,Ey] <- PenMoved;
  }
```

```
    }
  }
```

```
NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
Button                      CLASS
Cancel                      DIALOGUE
ChangePen                   DIALOGUE
DCButton                    WIDGET
DCButton.location           WIDGET_ATTR     DD
DCButton.name               WIDGET_ATTR     S
DCButton.size               WIDGET_ATTR     DD
DCButton.visible            WIDGET_ATTR
DCIcon1                     WIDGET
DCIcon1.location            WIDGET_ATTR     DD
DCIcon1.resource            WIDGET_ATTR     D
DCIcon1.size                WIDGET_ATTR     DD
DCIcon1.visible             WIDGET_ATTR
DCPen                       WIDGET
DCWindow                    WIDGET
DCWindow.location           WIDGET_ATTR     DD
DCWindow.size               WIDGET_ATTR     DD
DCWindow.visible            WIDGET_ATTR
DialogueBoxes               DIALOGUE
Divider                     DIALOGUE
DoDB                        EVENT           302
DoDraw                      EVENT           317
DoExit                      DIALOGUE
Done                        DIALOGUE
DoneDB                      EVENT           303
DrawWindow                  DIALOGUE
HSelect                     DIALOGUE
HThickIcon                  MACRO           158
HThickness                  VAR             314
HThinIcon                   MACRO           157
IconSize                    MACRO           30
MyPen                       DIALOGUE
OK                          DIALOGUE
PIPS_Stop                   MACRO           31
PaintSystem                 DIALOGUE
PenMoved                    EVENT           318
SelectShape                 EVENT           304
Selectable                  CLASS
SelectionBox                DIALOGUE
SetDefault                  EVENT           312
ThickLine                   MACRO           6
ThickSelect                 DIALOGUE
ThinLine                    MACRO           2
ThinSelect                  DIALOGUE
VDashIcon                   MACRO           159
VSelect                     DIALOGUE
VThickIcon                  MACRO           156
VThickness                  VAR             313
VThinIcon                   MACRO           155
WE_Dehilite                 EVENT           107
```

```
WE_Hide                 EVENT           103
WE_HideWindow           EVENT           101
WE_Hilite               EVENT           106
WE_MoveTo               EVENT           104
WE_SetHeight            EVENT           110
WE_SetWidth             EVENT           111
WE_Show                 EVENT           102
WE_ShowWindow           EVENT           100
```

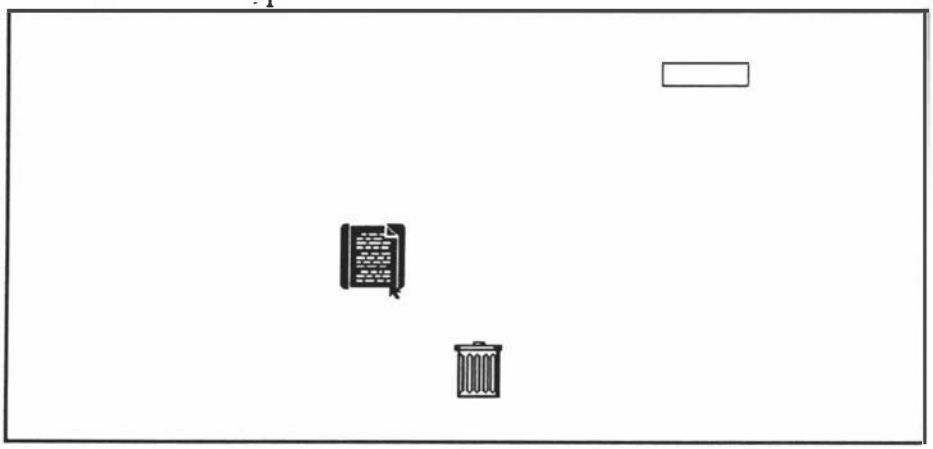
Trace

```
============ BrushShape ============
A] 00145974 PaintSystem
A] 00145974 DrawWindow
A] 00146043 ChangePen
A] 00146043 MyPen
D] 00146046                    MyPen
D] 00146047                    ChangePen
D] 00146047                    DrawWindow
A] 00146047 DialogueBoxes
A] 00146047 SelectionBox
A] 00146052 Done
A] 00146052 HSelect
A] 00146052 Divider
A] 00146052 VSelect
A] 00146052 ThinSelect
A] 00146052 ThinSelect
D] 00146125                    ThinSelect
A] 00146125 ThickSelect
A] 00146236 OK
D] 00146241                    OK
D] 00146241                    DialogueBoxes
D] 00146241                    SelectionBox
D] 00146241                    Done
D] 00146241                    HSelect
D] 00146242                    ThinSelect
D] 00146242                    Divider
D] 00146242                    VSelect
D] 00146242                    ThickSelect
A] 00146242 DrawWindow
A] 00146325 MyPen
D] 00146353                    MyPen
A] 00146401 ChangePen
A] 00146401 MyPen
D] 00146405                    MyPen
D] 00146405                    ChangePen
D] 00146405                    DrawWindow
A] 00146405 DialogueBoxes
A] 00146406 SelectionBox
A] 00146411 Done
A] 00146411 HSelect
A] 00146411 Divider
A] 00146411 VSelect
A] 00146411 ThinSelect
A] 00146412 ThinSelect
A] 00146463 OK
D] 00146467                    OK
D] 00146467                    DialogueBoxes
D] 00146467                    SelectionBox
D] 00146468                    Done
D] 00146468                    HSelect
D] 00146468                    ThinSelect
D] 00146468                    Divider
D] 00146468                    VSelect
D] 00146468                    ThinSelect
A] 00146468 DrawWindow
A] 00146509 MyPen
D] 00146531                    MyPen
A] 00146590 ChangePen
A] 00146591 MyPen
```

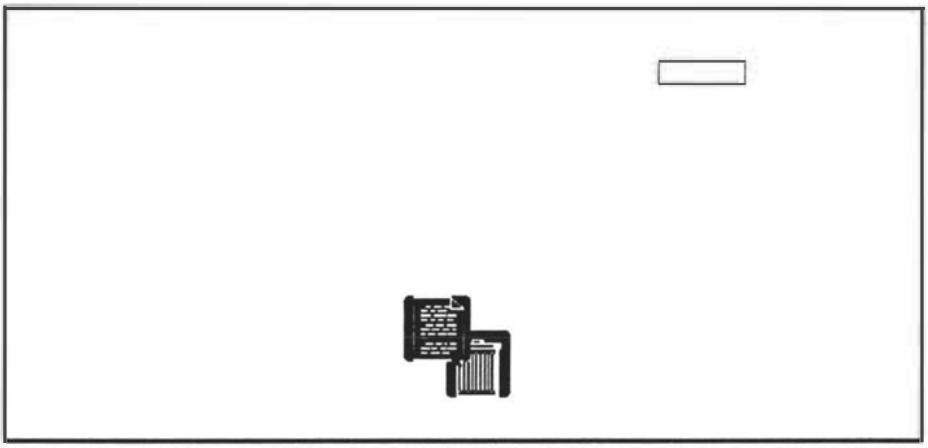
```
D] 00146594                    MyPen
D] 00146594                    ChangePen
D] 00146594                    DrawWindow
A] 00146594 DialogueBoxes
A] 00146595 SelectionBox
A] 00146599 Done
A] 00146599 HSelect
A] 00146599 Divider
A] 00146600 VSelect
A] 00146600 ThinSelect
A] 00146600 ThinSelect
D] 00146679                    ThinSelect
A] 00146679 ThickSelect
A] 00146769 Cancel
D] 00146774                    Cancel
D] 00146774                    DialogueBoxes
D] 00146774                    SelectionBox
D] 00146774                    Done
D] 00146774                    HSelect
D] 00146774                    ThinSelect
D] 00146774                    Divider
D] 00146775                    VSelect
D] 00146775                    ThickSelect
A] 00146775 DrawWindow
A] 00146815 MyPen
D] 00146839                    MyPen
A] 00146907 DoExit
```

7. SIMPLE DRAG AND DROP

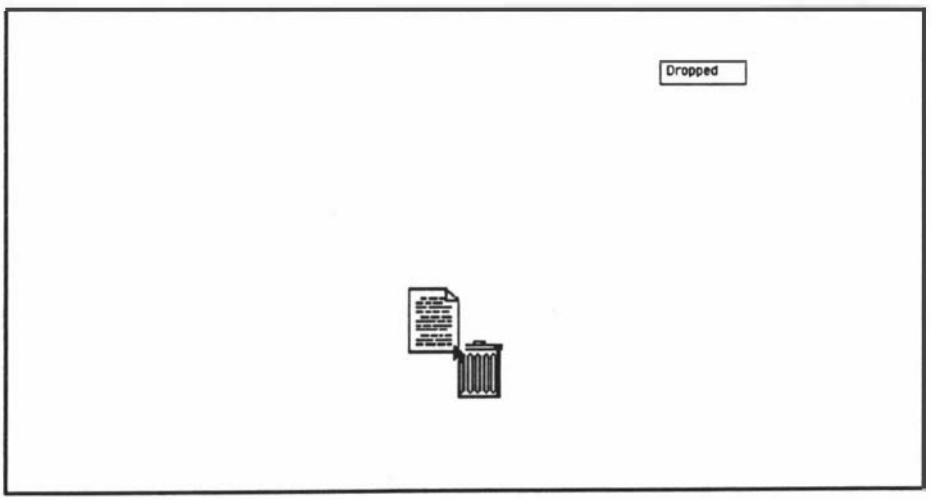
This example displays a file icon and trash can icon on the screen. Dragging and dropping the file icon onto the trash can causes the word dropped to be displayed in the small window in the top right of the screen as shown in Figure C.9(a-c). The drag and drop behaviour of this dialogue is exactly as described in Chapter 8.



(a)



(b)



(c)

Figure C.9: Simple drag and drop.

Listing

```
Compiled by: Daltran2.06                    Wed Feb  1 13:57:25 1995
Source File: DD1.dal

  /*
   *   Simple drag and drop example with a single file
   *   and trash can as described in Chapter 8.
   *   A monitor dialogue has been added that displays
   *   "Dropped" whenever the file is dropped on the
   *   trash icon.
   */
  INCLUDE   <Widgets.dal>
  INCLUDE   <WidgetEvents.dal>
  INCLUDE   <APIEvents.dal>

  DEFINE
     FileIcon    152,   // Resource numbers for icons
     TrashIcon   154,

     IconSize  60;

  CLASS

     /*
      *   EnableDrag exported.
      *   A FileIcon is assumed to be contained within
      *   a ME group.
      */
     File( []Mv )( M^ ){
        EVENT WE_Hilite, WE_Dehilite, WE_MoveBy;
        ON ACTIVATION {
           WE_Hilite;
           EnableDrag;
        }
        WE_Dehilite <- M^;
        WE_MoveBy[Ea,Eb] <- cursor_moved;
     },

     /*
      *   EnableDrag imported
      *   DoTrash exported
      */
     Trash( EnableDrag )( M^ ){
        EVENT WE_Hilite, WE_Dehilite;
        WE_Hilite <- ~[];
        WE_Dehilite <- []~ | M^;
        DoTrash <- []M^;
     };


  SYSTEM

     Desktop()(){
        DISPLAY TYPE DCWindow,
           size 480 640,
           location 0 0,
           visible;
        EVENT EnableDrag, DoTrash;
```

```
    /*
     *   command Q to exit
     */
    doExit( 'COMMAND'v )( 'COMMAND'^ ){
      PIPS_Stop <- 'q'v;
    }

    /*
     *   DoTrash imported.
     *   Report when a file is dropped on the trash.
     *   Event API_Select is picked up by ACSelector,
     *   which writes the name of the dialogue (Dropped)
     *   into an API data buffer. It then generates event
     *   API_DoWrite, which is picked up by the API agent
     *   ACScribe, which then writes this text into its
     *   window.
     */
    monitorAPI()(){
      EVENT API_Clear;
      DISPLAY type DCWindow,
           size 20 80,
           location 500 50,
           visible;
      APPLICATION ACScribe;
      API_Clear <- Mv;

      Dropped()(){
        EVENT API_Select;
        APPLICATION ACSelector;
        API_Select <- DoTrash;
      }
    }

    theFile : File()(){
      DISPLAY TYPE DCIcon1,
           resource FileIcon,
           size IconSize IconSize,
           location 200 200,
           visible;
    }

    theTrash : Trash()(){
      DISPLAY TYPE DCIcon1,
           resource TrashIcon,
           size IconSize IconSize,
           location 300 300,
           visible;
      }
  }


NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
API_Clear                EVENT          203
API_Select               EVENT          200
DCIcon1                  WIDGET
```

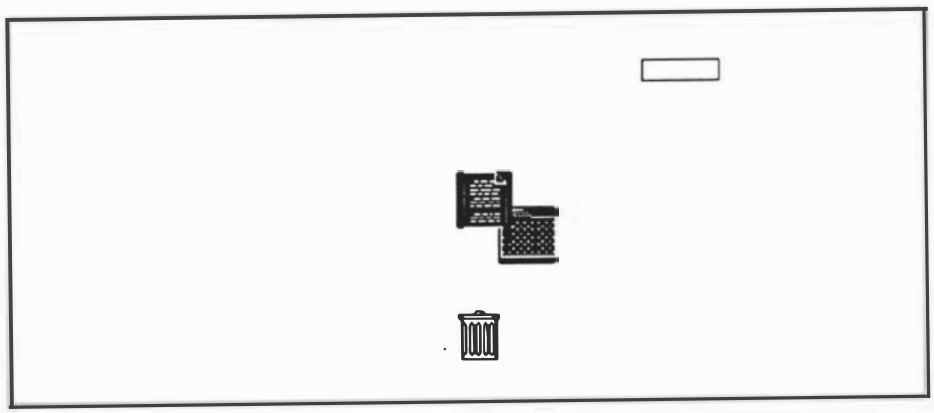
DCIcon1.location	WIDGET_ATTR	DD
DCIcon1.resource	WIDGET_ATTR	D
DCIcon1.size	WIDGET_ATTR	DD
DCIcon1.visible	WIDGET_ATTR	
DCWindow	WIDGET	
DCWindow.location	WIDGET_ATTR	DD
DCWindow.size	WIDGET_ATTR	DD
DCWindow.visible	WIDGET_ATTR	
Desktop	DIALOGUE	
DoTrash	EVENT	307
Dropped	DIALOGUE	
EnableDrag	EVENT	303
File	CLASS	
FileIcon	MACRO	152
IconSize	MACRO	60
PIPS_Stop	MACRO	31
Trash	CLASS	
TrashIcon	MACRO	154
WE_Dehilite	EVENT	107
WE_Hilite	EVENT	106
WE_MoveBy	EVENT	105
doExit	DIALOGUE	
monitorAPI	DIALOGUE	
theFile	DIALOGUE	
theTrash	DIALOGUE	

Trace

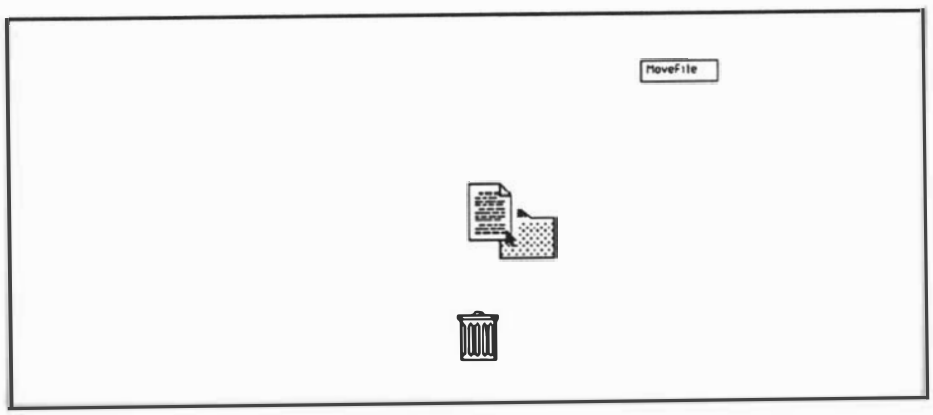
```
============ DD1 ============
A] 00705394 Desktop
A] 00705394 monitorAPI
A] 00705394 Dropped
A] 00705461 theFile
A] 00705462 theTrash
D] 00705567                    theTrash
D] 00705567                    theFile
A] 00705621 theFile
A] 00705622 theTrash
D] 00705651                    theTrash
D] 00705652                    theFile
A] 00705751 doExit
```

8. Drag and Drop with Multiple Destinations

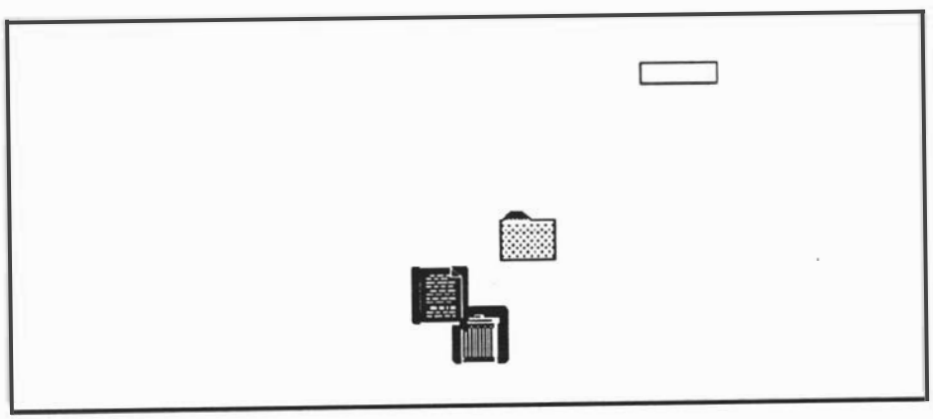
This example is to demonstrate the drag and drop dialogue with multiple destinations (a folder and a trash can) as described in Chapter 8. Figure C.10 is a sequence of screen dumps showing a file being dropped on a folder (a-b) and on the trash can (c-d). The window in the top right of the screen presents *MoveFile* or *TrashFile* as appropriate.



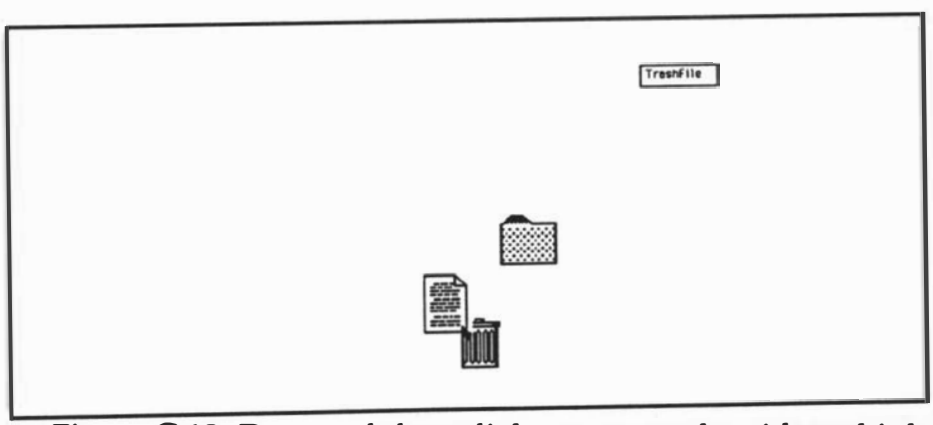
(a)



(b)



(c)



(d)

Figure C.10: Drag and drop dialogue example with multiple destinations.

Listing

```
Compiled by: Daltran2.06                    Wed Feb  1 13:57:50 1995
Source File: DD2.dal

  /*
   *   Drag and drop example with a single file
   *   and both folder and trash destinations as
   *   described in Chapter 8. A monitor dialogue
   *   has been added that displays "MoveFile" or
   *   "TrashFile" as appropriate.
   */
  INCLUDE   <Widgets.dal>
  INCLUDE   <WidgetEvents.dal>
  INCLUDE   <APIEvents.dal>

  DEFINE
     FileIcon    152,   // Resource numbers for icons
     FolderIcon  153,
     TrashIcon   154,

     IconSize  60;

  CLASS

     /*
      *   EnableDrag exported.
      *   A FileIcon is assumed to be contained within
      *   a ME group.
      */
     File( []Mv )( M^ ){
       EVENT WE_Hilite, WE_Dehilite, WE_MoveBy;
       ON ACTIVATION {
         WE_Hilite;
         EnableDrag;
       }
       WE_Dehilite <- M^;
       WE_MoveBy[Ea,Eb] <- cursor_moved;
     },

     /*
      *   EnableDrag imported
      */
     DD_Destination( EnableDrag )( M^ ){
       EVENT WE_Hilite, WE_Dehilite;
       WE_Hilite <- ~[];
       WE_Dehilite <- []~ | M^;
     },

     /*
      *   DoFileMove exported
      */
     Folder : DD_Destination()(){
       DoFileMove <- []M^;
     },
```

```
    /*
     *   DoTrash exported
     */
    Trash : DD_Destination()(){
      DoTrash <- []M^;
    };


SYSTEM

    Desktop()(){
      DISPLAY TYPE DCWindow,
         size 480 640,
         location 0 0,
         visible;
      EVENT EnableDrag, DoFileMove, DoTrash;

      /*
       *   command Q to exit
       */
      doExit( 'COMMAND'v )( 'COMMAND'^ ){
        PIPS_Stop <- 'q'v;
      }

      /*
       *   DoFileMove and DoTrash imported.
       *   Reporting these events in a monitor window.
       */
      monitorAPI()(){
        EVENT API_Clear;
        DISPLAY type DCWindow,
             size 20 80,
             location 500 50,
             visible;
        APPLICATION ACScribe;
        API_Clear <- Mv;

        MoveFile()(){
          EVENT API_Select;
          APPLICATION ACSelector;
          API_Select <- DoFileMove;
        }
        TrashFile()(){
          EVENT API_Select;
          APPLICATION ACSelector;
          API_Select <- DoTrash;
        }
      }

      theFile : File()(){
        DISPLAY TYPE DCIcon1,
             resource FileIcon,
             size IconSize IconSize,
             location 200 200,
             visible;
      }
```

```
        theFolder : Folder()(){
          DISPLAY TYPE DCIcon1,
                resource FolderIcon,
                size IconSize IconSize,
                location 350 200,
                visible;
        }

        theTrash : Trash()(){
          DISPLAY TYPE DCIcon1,
                resource TrashIcon,
                size IconSize IconSize,
                location 300 300,
                visible;
        }
    }


NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
API_Clear                   EVENT           203
API_Select                  EVENT           200
DCIcon1                     WIDGET
DCIcon1.location            WIDGET_ATTR     DD
DCIcon1.resource            WIDGET_ATTR     D
DCIcon1.size                WIDGET_ATTR     DD
DCIcon1.visible             WIDGET_ATTR
DCWindow                    WIDGET
DCWindow.location           WIDGET_ATTR     DD
DCWindow.size               WIDGET_ATTR     DD
DCWindow.visible            WIDGET_ATTR
DD_Destination              CLASS
Desktop                     DIALOGUE
DoFileMove                  EVENT           307
DoTrash                     EVENT           308
EnableDrag                  EVENT           303
File                        CLASS
FileIcon                    MACRO           152
Folder                      CLASS
FolderIcon                  MACRO           153
IconSize                    MACRO           60
MoveFile                    DIALOGUE
PIPS_Stop                   MACRO           31
Trash                       CLASS
TrashFile                   DIALOGUE
TrashIcon                   MACRO           154
WE_Dehilite                 EVENT           107
WE_Hilite                   EVENT           106
WE_MoveBy                   EVENT           105
doExit                      DIALOGUE
monitorAPI                  DIALOGUE
theFile                     DIALOGUE
theFolder                   DIALOGUE
theTrash                    DIALOGUE
```

Trace

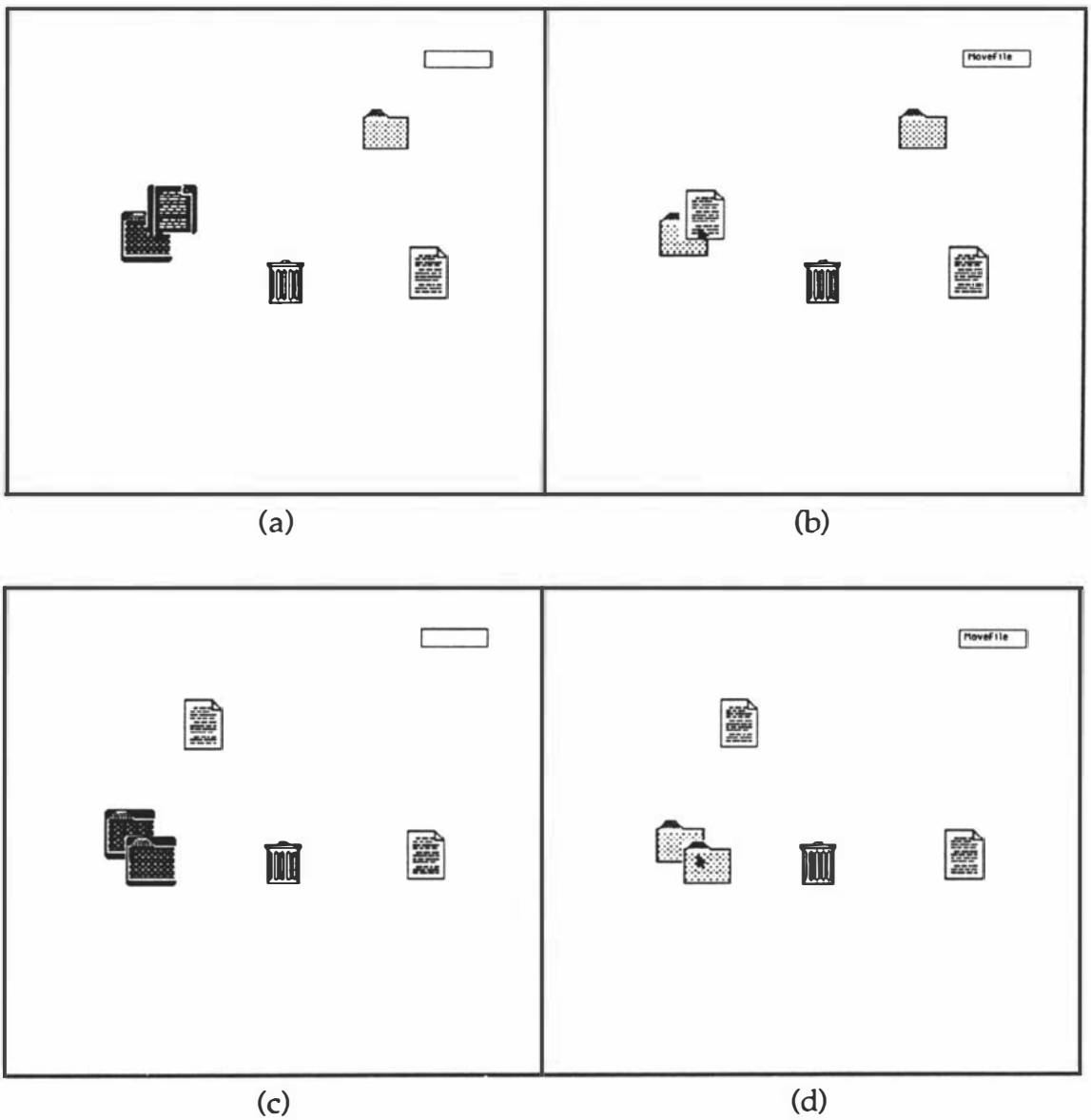
```
============ DD2 ============
A] 00704206 Desktop
A] 00704206 monitorAPI
A] 00704206 TrashFile
A] 00704207 MoveFile
A] 00704285 theFile
A] 00704286 theTrash
A] 00704286 theFolder
D] 00704377                 theTrash
D] 00704377                 theFolder
D] 00704377                 theFile
A] 00704442 theFile
A] 00704444 theTrash
A] 00704444 theFolder
D] 00704467                 theTrash
D] 00704467                 theFolder
D] 00704467                 theFile
A] 00704506 theFile
A] 00704507 theTrash
A] 00704507 theFolder
D] 00704659                 theTrash
D] 00704659                 theFolder
D] 00704659                 theFile
A] 00704717 theFile
A] 00704717 theTrash
A] 00704718 theFolder
D] 00704760                 theTrash
D] 00704760                 theFolder
D] 00704760                 theFile
A] 00704815 doExit
```

9. DYNAMIC "DRAG AND DROP"

This example is to demonstrate the drag and drop dialogue with dialogue objects (folders) that can be both destinations or targets within the drag and drop dialogue, as described in Chapter 8. This example also demonstrates the dynamic creation of μdialogues. New folder icons can be created by depressing the option key and clicking on the main window, new file icons are created by depressing the shift key and clicking on the main window. Within these μdialogues, further sub-μdialogues are created within the *on new* blocks.

The sequence of screen shots shown in Figure C.11 show the use of folder icons as destinations for file icons (a and b), and for other folder icons (c and d). As was the case with the previous drag and drop examples, on successfully dropping the target, an appropriate message is displayed in the small window in the top right hand corner of the screen.

The sequence of screen shots shown in Figure C.12 show the use of the trash icon as a drop destination for both file icons (a and b) and folder icons (c and d).



(a) (b)

(c) (d)

Figure C.11: Dropping a file icon on a folder to move it (a-b), and dropping a folder icon on another folder icon to move it (c-d).

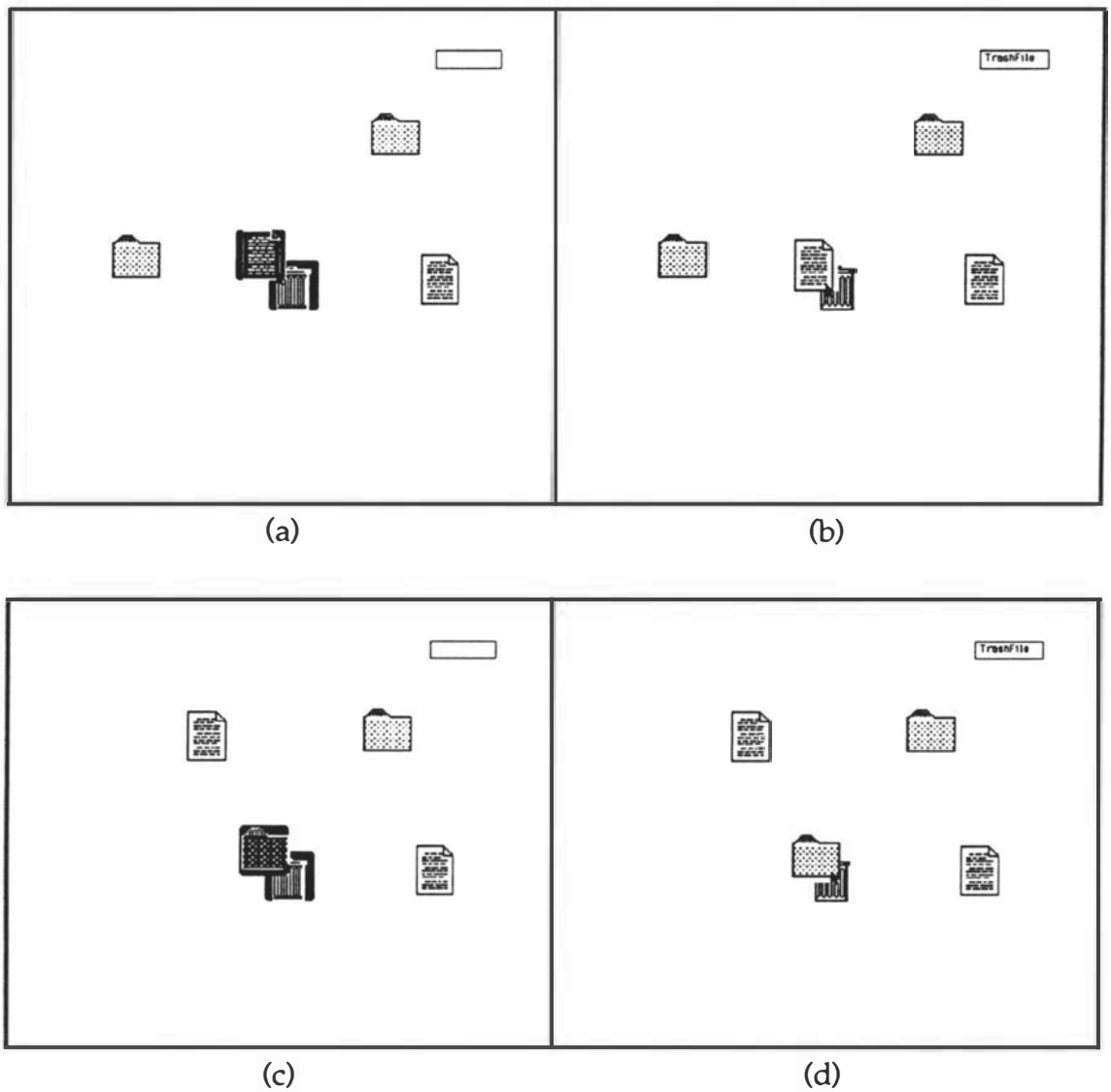


Figure C.12: Dropping a file icon on a trash icon to remove it (a-b), and dropping a folder icon onto a trash icon to remove it (c-d).

Listing

```
Compiled by: Daltran2.06          Mon Jun 12 16:31:35 1995
Source File: DD3.dal

  /*
   *   Drag and drop example with both files and
   *   folders. Files and folders can both be dropped
   *   on a trash icon, producing the response "TrashFile".
   *   In addition, both files and folders can be dropped
   *   on folders producing the response "Movefile". Hence
   *   folders can be both targets and destinations in
   *   the drop dialogue. This example is a described in
   *   Chapter 8.
   *   The file and folder µdialogues in this example are
   *   created dynamically.
   */
  INCLUDE   <Widgets.dal>
  INCLUDE   <WidgetEvents.dal>
  INCLUDE   <APIEvents.dal>

  DEFINE
    FALSE     0,
    TRUE      1;

  DEFINE
    FileIcon    152,   // Resource numbers for icons
    FolderIcon  153,
    TrashIcon   154,

    IconSize  60;

  CLASS

    /*
     *   EnableDrag exported.
     *
     *   Select imported from parent.
     *   WE_Hilite, WE_Dehilite, WE_MoveBy exported
     *  up one level.
     *
     *   A FileIcon is assumed to be contained within
     *   a ME group.
     */
    DD_Target( []Mv )( M^ ){
      ON ACTIVATION {
        WE_Hilite;
        EnableDrag;
      }
      WE_Dehilite <- M^;
      WE_MoveBy[Ea,Eb] <- cursor_moved;
    },

    /*
     *   EnableDrag imported
     */
    DD_Destination( EnableDrag )( M^ ){
      WE_Hilite <- -[];
      WE_Dehilite <- []- | M^;
    },
```

```
/*
 *   DoFileMove imported
 */
File : DD_Target()(){
   DISPLAY TYPE DCIcon1,
        resource FileIcon,
        size IconSize IconSize,
        visible;
   EVENT WE_Hilite, WE_Dehilite, WE_MoveBy;
},

/*
 *   Variable IsTarget imported from parent
 */
MoveFolder : DD_Target()(){
   ON ACTIVATION{ IsTarget=TRUE; }
   ON DEACTIVATION{ IsTarget=FALSE; }
},

/*
 *   DoFileMove exported
 *   Variable IsTarget imported from parent
 */
ReceiveFile : DD_Destination[IsTarget==FALSE]()(){
   DoFileMove <- []M^;
},

Folder()(){
   DISPLAY TYPE DCIcon1,
        resource FolderIcon,
        size IconSize IconSize,
        visible;
   VAR IsTarget;
   EVENT WE_Hilite, WE_Dehilite, WE_MoveBy;

   ON ACTIVATION{ IsTarget=FALSE; }

   GROUPING ME;

   ON NEW {
      NEW MoveFolder;
      NEW ReceiveFile;
   }
},

/*
 *   DoTrash exported
 */
Trash : DD_Destination()(){
   EVENT WE_Hilite, WE_Dehilite;
   DoTrash <- []M^;
};
```

```
SYSTEM

   Desktop()(){
      DISPLAY TYPE DCWindow,
         size 480 640,
         location 0 0,
         visible;
      EVENT EnableDrag, DoFileMove, DoTrash;

      /*
       *   command Q to exit
       */
      do_exit( 'COMMAND'v )( 'COMMAND'^ ){
         PIPS_Stop <- 'q'v;
      }

      /*
       *   DoFileMove and DoTrash imported.
       *   Reporting these events in a monitor window.
       */
      monitorAPI()(){
         EVENT API_Clear;
         DISPLAY type DCWindow,
              size 20 80,
              location 500 50,
              visible;
         APPLICATION ACScribe;
         API_Clear <- Mv;

         MoveFile()(){
            EVENT API_Select;
            APPLICATION ACSelector;
            API_Select <- DoFileMove;
         }
         TrashFile()(){
            EVENT API_Select;
            APPLICATION ACSelector;
            API_Select <- DoTrash;
         }
      }

      Finder()(){
         EVENT NewFile, NewFolder;

         NEW File[Ex,Ey,IconSize,IconSize] <- NewFile;
         NEW Folder[Ex,Ey,IconSize,IconSize] <- NewFolder;


         /*
          * Create new file icons by depressing the shift
          * key and clicking on the finder window.
          */
         MakeFiles( 'SHIFT'v )( 'SHIFT'^ ){
            NewFile[Ex,Ey] <- Mv;
         }
```

```
        /*
         * Create new folder icons by depressing the option
         * key and clicking on the finder window.
         */
        MakeFolders( 'OPTION'v )( 'OPTION'^ ){
          NewFolder[Ex,Ey] <- Mv;
        }
      }

      theTrash : Trash()(){
        DISPLAY TYPE DCIcon1,
             resource TrashIcon,
             size IconSize IconSize,
             location 300 300,
             visible;
      }
    }
```

```
NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
API_Clear                 EVENT           203
API_Select                EVENT           200
DCIcon1                   WIDGET
DCIcon1.location          WIDGET_ATTR     DD
DCIcon1.resource          WIDGET_ATTR     D
DCIcon1.size              WIDGET_ATTR     DD
DCIcon1.visible           WIDGET_ATTR
DCWindow                  WIDGET
DCWindow.location         WIDGET_ATTR     DD
DCWindow.size             WIDGET_ATTR     DD
DCWindow.visible          WIDGET_ATTR
DD_Destination            CLASS
DD_Target                 CLASS
Desktop                   DIALOGUE
DoFileMove                EVENT           306
DoTrash                   EVENT           313
EnableDrag                EVENT           300
FALSE                     MACRO           0
File                      CLASS           2
FileIcon                  MACRO           152
Finder                    DIALOGUE
Folder                    CLASS           3
FolderIcon                MACRO           153
IconSize                  MACRO           60
IsTarget                  VAR             305
MakeFiles                 DIALOGUE
MakeFolders               DIALOGUE
MoveFile                  DIALOGUE
MoveFolder                CLASS           0
NewFile                   EVENT           322
NewFolder                 EVENT           323
PIPS_Stop                 MACRO           31
ReceiveFile               CLASS           1
TRUE                      MACRO           1
Trash                     CLASS
```

TrashFile	DIALOGUE	
TrashIcon	MACRO	154
WE_Dehilite	EVENT	107
WE_Hilite	EVENT	106
WE_MoveBy	EVENT	105
do_exit	DIALOGUE	
monitorAPI	DIALOGUE	
theTrash	DIALOGUE	

Trace

```
============ DD3 ============
A] 00012991 Desktop
A] 00012991 Finder
A] 00012991 monitorAPI
A] 00012991 TrashFile
A] 00012991 MoveFile
A] 00013100 MakeFolders
A] 00013178 Folder00
A] 00013268 Folder03
D] 00013297                 MakeFolders
A] 00013329 MakeFiles
D] 00013474                 MakeFiles
A] 00013665 File06
A] 00013665 theTrash
A] 00013665 ReceiveFile02
A] 00013665 ReceiveFile05
D] 00013742                 theTrash
D] 00013742                 ReceiveFile02
D] 00013742                 ReceiveFile05
D] 00013742                 File06
A] 00013818 File06
A] 00013819 theTrash
A] 00013819 ReceiveFile02
A] 00013819 ReceiveFile05
D] 00013864                 theTrash
D] 00013864                 ReceiveFile02
D] 00013864                 ReceiveFile05
D] 00013864                 File06
A] 00013969 File06
A] 00013970 theTrash
A] 00013970 ReceiveFile02
A] 00013970 ReceiveFile05
D] 00014012                 theTrash
D] 00014012                 ReceiveFile02
D] 00014012                 ReceiveFile05
D] 00014012                 File06
A] 00014100 File06
A] 00014100 theTrash
A] 00014100 ReceiveFile02
A] 00014100 ReceiveFile05
D] 00014144                 theTrash
D] 00014144                 ReceiveFile02
D] 00014144                 ReceiveFile05
D] 00014145                 File06
A] 00014316 MoveFolder01
A] 00014317 theTrash
A] 00014317 ReceiveFile05
D] 00014378                 theTrash
D] 00014378                 MoveFolder01
D] 00014378                 ReceiveFile05
A] 00014450 MoveFolder01
A] 00014450 theTrash
A] 00014450 ReceiveFile05
D] 00014490                 theTrash
D] 00014490                 MoveFolder01
D] 00014490                 ReceiveFile05
A] 00014583 MoveFolder01
A] 00014584 theTrash
A] 00014584 ReceiveFile05
```

```
D] 00014648                    theTrash
D] 00014648                    MoveFolder01
D] 00014648                    ReceiveFile05
A] 00014740 MoveFolder01
A] 00014741 theTrash
A] 00014741 ReceiveFile05
D] 00014793                    theTrash
D] 00014793                    MoveFolder01
D] 00014793                    ReceiveFile05
A] 00014869 do_exit
```

10. Simple Gesture Recognition

The sample demonstrates the basic approach to recognising gestures using DAL, as described in Chapter 8. A simple gesture tablet is displayed on the screen, and dragging the cursor over the tablet causes a button to highlight, which button being dependent on the direction the cursor has moved (left to right, or right to left). The screen dump in Figure C.13 shows the screen immediately after completing a drag from right to left.

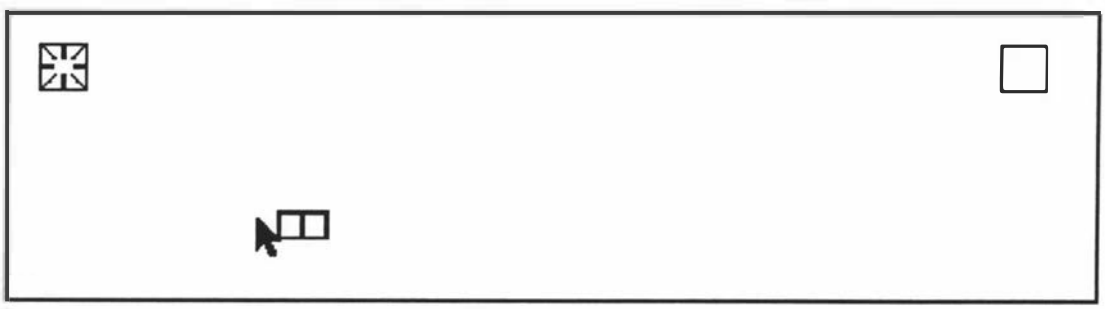


Figure C.13: Simple gesture recognition.

Listing

```
Compiled by: Daltran2.06                    Wed Feb  1 14:50:38 1995
Source File: SimpleG.dal

  /************************************************\
  *                                                *
  *   A simple gesture tablet, able to recognise   *
  *   left to right, and right to left gestures.   *
  *                                                *
  \************************************************/


  INCLUDE <Widgets.dal>
  INCLUDE <WidgetEvents.dal>
  INCLUDE <APIEvents.dal>

  DEFINE
     dark   141,
     light  142,
     box    142;


  SYSTEM

     Desktop()(){
        EVENT G1, G2;

        DISPLAY type DCWindow,
             size 480 640,
             location 0 0,
             visible;

          // Exit the interface

        Terminate( 'COMMAND'v )( 'COMMAND'^ ){
          PIPS_Stop <- 'q'v;
        }

        Indicator()(){
          EVENT WE_Show;
          ON ACTIVATION{ WE_Show; }

          GROUPING ME;

          LtoR(G1)(){
            EVENT WE_Hilite, WE_Dehilite;
            DISPLAY type DCIcon2,
               size 20 20,
               location 0 30,
               visible,
               resource light dark;
            ON ACTIVATION{ WE_Hilite; }
            ON DEACTIVATION{ WE_Dehilite; }
          }
```

```
        RtoL(G2)(){
          EVENT WE_Hilite, WE_Dehilite;
          display type DCIcon2,
             size 20 20,
             location 400 30,
             visible,
             resource light dark;
          ON ACTIVATION{ WE_Hilite; }
          ON DEACTIVATION{ WE_Dehilite; }
        }
      }

      Tablet(  )(  ){
        DISPLAY   type DCWindow,
             size 10 20,
             location 100 100,
             visible;

        Holder()(){
          EVENT Done;
          DISPLAY type DCIcon1,
             size 10 20,
             location 0 0,
             visible,
             resource box;

          Done <- []~;

          GROUPING ME;

          A( ~[] )( ){
             DISPLAY type DCIcon1,
               size 10 10,
               location 0 0,
               visible,
               resource box;
             G1 <- Done;
          }
          B( ~[] )( ){
             DISPLAY type DCIcon1,
               size 10 10,
               location 10 0,
               visible,
               resource box;
             G2 <- Done;
          }
        }
      }
    }
```

```
NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
A                           DIALOGUE
B                           DIALOGUE
DCIcon1                     WIDGET
```

```
DCIcon1.location          WIDGET_ATTR     DD
DCIcon1.resource          WIDGET_ATTR     D
DCIcon1.size              WIDGET_ATTR     DD
DCIcon1.visible           WIDGET_ATTR
DCIcon2                   WIDGET
DCIcon2.location          WIDGET_ATTR     DD
DCIcon2.resource          WIDGET_ATTR     DD
DCIcon2.size              WIDGET_ATTR     DD
DCIcon2.visible           WIDGET_ATTR
DCWindow                  WIDGET
DCWindow.location         WIDGET_ATTR     DD
DCWindow.size             WIDGET_ATTR     DD
DCWindow.visible          WIDGET_ATTR
Desktop                   DIALOGUE
Done                      EVENT           309
G1                        EVENT           300
G2                        EVENT           301
Holder                    DIALOGUE
Indicator                 DIALOGUE
LtoR                      DIALOGUE
PIPS_Stop                 MACRO           31
RtoL                      DIALOGUE
Tablet                    DIALOGUE
Terminate                 DIALOGUE
WE_Dehilite               EVENT           107
WE_Hilite                 EVENT           106
WE_Show                   EVENT           102
box                       MACRO           142
dark                      MACRO           141
light                     MACRO           142
```

Trace

```
============ SimpleG ============
A] 00894128 Desktop
A] 00894129 Tablet
A] 00894129 Holder
A] 00894129 Indicator
A] 00894208 A
D] 00894213                  A
A] 00894213 B
A] 00894219 RtoL
D] 00894283                  B
A] 00894283 A
D] 00894290                  RtoL
A] 00894291 LtoR
D] 00894342                  A
A] 00894342 B
D] 00894348                  LtoR
A] 00894349 RtoL
D] 00894400                  B
A] 00894400 A
D] 00894407                  RtoL
A] 00894407 LtoR
A] 00894481 Terminate
```

11. COMPOUND GESTURE RECOGNITION TABLET

This example is exactly like the compound gesture recognition example presented at the end of Chapter 8. Four labelled buttons are included, that highlight on a gesture being recognised. Figure C.14 shows the screen immediately after completing an anti-clockwise circular gesture.

Figure C.14: Demonstration of compound gesture recognition.

Listing

```
Compiled by: Daltran2.06                    Wed Feb  1 14:44:08 1995
Source File: GRT.dal

  /************************************************\
  *                      GRT.DAL                   *
  *                      =======                   *
  *    A Simple gesture recognition tablet.        *
  *                                                *
  *    Expand - circle clockwise.                  *
  *    Shrink - circle anti-clockwise.             *
  *    Stretch - Figure 8, clockwise at top.       *
  *    Flatten - Figure 8, anti-clockwise at top.  *
  *                                                *
  \************************************************/

  INCLUDE <Widgets.dal>
  INCLUDE <WidgetEvents.dal>
  INCLUDE <APIEvents.dal>

  DEFINE
    box       142,
    tl        137,
    tr        138,
    bl        140,
    br        139;


  SYSTEM

    Desktop()(){
      EVENT Expand, Shrink, Stretch, Flatten;

      DISPLAY type DCWindow,
           size 480 640,
           location 0 0,
           visible;

        // Exit the interface

      terminate( 'COMMAND'v )( 'COMMAND'^ ){
        PIPS_Stop <- 'q'v;
      }

      Indicator()(){
        EVENT WE_Show;
        ON ACTIVATION{ WE_Show; }

        GROUPING ME;

        DoExpand(Expand)(){
          EVENT WE_Hilite, WE_Dehilite;
          DISPLAY   type DCButton,
                size 30 100,
                location 50 30,
                visible,
                name EXPAND;
          ON ACTIVATION{ WE_Hilite; }
          ON DEACTIVATION{ WE_Dehilite; }
        }
```

```
    DoShrink(Shrink)(){
      EVENT WE_Hilite, WE_Dehilite;
      DISPLAY   type DCButton,
           size 30 100,
           location 200 30,
           visible,
           name SHRINK;
      ON ACTIVATION{ WE_Hilite; }
      ON DEACTIVATION{ WE_Dehilite; }
    }
    DoStretch(Stretch)(){
      EVENT WE_Hilite, WE_Dehilite;
      DISPLAY   type DCButton,
           size 30 100,
           location 350 30,
           visible,
           name STRETCH;
      ON ACTIVATION{ WE_Hilite; }
      ON DEACTIVATION{ WE_Dehilite; }
    }

    DoFlatten(Flatten)(){
      EVENT WE_Hilite, WE_Dehilite;
      DISPLAY   type DCButton,
           size 30 100,
           location 500 30,
           visible,
           name FLATTEN;
      ON ACTIVATION{ WE_Hilite; }
      ON DEACTIVATION{ WE_Dehilite; }
    }
}

/*
 *   Exports events Expand, Shrink, Stretch and Flatten
 */
GRT(  )(  ){
  DISPLAY   type DCWindow,
       size 20 10,
       location 100 100,
       visible;
  EVENT TopLtoR, TopRtoL, BottomLtoR, BottomRtoL;

  C()(){
    DISPLAY type DCIcon1,
       size 5 10,
       location 0 0,
       visible,
       resource box;
    EVENT CDone;

    GROUPING ME;

    CDone <- []-;
```

```
    A( ~[] )(){
       DISPLAY type DCIcon1,
         size 5 5,
         location 0 0,
         visible,
         resource tl;
       TopRtoL <- CDone;
    }
    B( ~[] )(){
       DISPLAY type DCIcon1,
         size 5 5,
         location 5 0,
         visible,
         resource tr;
       TopLtoR <- CDone;
    }
  }
  F()(){
    DISPLAY type DCIcon1,
       size 5 10,
       location 0 15,
       visible,
       resource box;
    EVENT FDone;
    GROUPING ME;

    FDone <- []~;

    D( ~[] )(){
       DISPLAY type DCIcon1,
         size 5 5,
         location 0 15,
         visible,
         resource bl;
       BottomRtoL <- FDone;
    }
    E( ~[] )(){
       DISPLAY type DCIcon1,
         size 5 5,
         location 5 15,
         visible,
         resource br;
       BottomLtoR <- FDone;
    }
  }
  Categorize()(){
    GROUPING ME;

    Resynch( Mv )(){
    }
    DidTopLtoR( TopLtoR )(){
       Stretch <- BottomLtoR;
       Expand <- BottomRtoL;
    }
    DidTopRtoL( TopRtoL )(){
       Shrink <- BottomLtoR;
       Flatten <- BottomRtoL;
    }
  }
```

```
        }
    }


NO ERRORS

No potential event cycles found

=============== SYMBOL TABLE ===============
A                           DIALOGUE
B                           DIALOGUE
BottomLtoR                  EVENT           319
BottomRtoL                  EVENT           320
C                           DIALOGUE
CDone                       EVENT           321
Categorize                  DIALOGUE
D                           DIALOGUE
DCButton                    WIDGET
DCButton.location           WIDGET_ATTR     DD
DCButton.name               WIDGET_ATTR     S
DCButton.size               WIDGET_ATTR     DD
DCButton.visible            WIDGET_ATTR
DCIcon1                     WIDGET
DCIcon1.location            WIDGET_ATTR     DD
DCIcon1.resource            WIDGET_ATTR     D
DCIcon1.size                WIDGET_ATTR     DD
DCIcon1.visible             WIDGET_ATTR
DCWindow                    WIDGET
DCWindow.location           WIDGET_ATTR     DD
DCWindow.size               WIDGET_ATTR     DD
DCWindow.visible            WIDGET_ATTR
Desktop                     DIALOGUE
DidTopLtoR                  DIALOGUE
DidTopRtoL                  DIALOGUE
DoExpand                    DIALOGUE
DoFlatten                   DIALOGUE
DoShrink                    DIALOGUE
DoStretch                   DIALOGUE
E                           DIALOGUE
Expand                      EVENT           300
F                           DIALOGUE
FDone                       EVENT           324
Flatten                     EVENT           303
GRT                         DIALOGUE
Indicator                   DIALOGUE
PIPS_Stop                   MACRO           31
Resynch                     DIALOGUE
Shrink                      EVENT           301
Stretch                     EVENT           302
TopLtoR                     EVENT           317
TopRtoL                     EVENT           318
WE_Dehilite                 EVENT           107
WE_Hilite                   EVENT           106
WE_Show                     EVENT           102
bl                          MACRO           140
box                         MACRO           142
br                          MACRO           139
terminate                   DIALOGUE
tl                          MACRO           137
```

```
tr                      MACRO           138
```

Trace

```
========== GRT ============
 00874763 Desktop
 00874763 GRT
 00874764 Categorize
 00874764 F
 00874764 C
 00874764 Indicator
 00874938 A
 00874963                    A
 00874963 B
 00874984 DidTopLtoR
 00875035 E
 00875048                    E
 00875048 D
 00875051                    DidTopLtoR
 00875051 Resynch
 00875133                    B
 00875133 A
 00875138                    Resynch
 00875138 DidTopRtoL
 00875153                    A
 00875153 B
 00875159                    DidTopRtoL
 00875160 DidTopLtoR
 00875195                    D
 00875195 E
 00875203                    E
 00875203 D
 00875206 DoExpand
 00875369                    DidTopLtoR
 00875369 Resynch
 00875457                    D
 00875457 E
 00875548                    B
 00875548 A
 00875562                    Resynch
 00875562 DidTopRtoL
 00875613                    E
 00875613 D
 00875621                    D
 00875621 E
 00875629                    DoExpand
 00875629 DoShrink
 00875705                    DidTopRtoL
 00875705 Resynch
 00876051                    A
 00876051 B
 00876070                    Resynch
 00876071 DidTopLtoR
 00876188                    E
 00876188 D
 00876198                    D
 00876198 E
 00876209                    DoShrink
 00876209 DoStretch
 00876352                    DidTopLtoR
 00876352 Resynch
 00876962                    B
 00876962 A
```

```
D] 00876968                    Resynch
A] 00876968 DidTopRtoL
D] 00877060                    E
A] 00877060 D
D] 00877069                    DoStretch
A] 00877069 DoFlatten
A] 00877198 terminate
```