

University of Bath



PHD

Graphical interaction management

Barn, Balbir Singh

Award date:
1988

Awarding institution:
University of Bath

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. May. 2019

Graphical Interaction Management

submitted by

Balbir Singh Barn

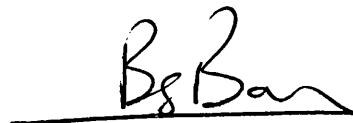
for the degree of Ph.D. of the

University of Bath

1988

Attention is drawn to the fact that the copyright of this thesis rests with its author. This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

A handwritten signature in black ink, appearing to read 'Bs Barn', is written above a horizontal line.

Balbir Barn

UMI Number: U018893

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI U018893

Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

UNIVERSITY OF BATH LIBRARY		
23	3 - AUG 1989	
Ph.D		

5031967

Summary

We present a survey of existing research on user interface design and interactive graphics programming. Next we argue that existing software methodologies are not suitable for developing interactive graphical applications. Using this as a basis of our research, we have identified control constructs which are suited to developing such applications. In addition, a new interaction model has been designed. This new model is a refined version of the traditional *repeat..until* loop commonly found in interactive applications. The combined model and control constructs provide a methodology for constructing well structured applications.

The new model has been implemented by a preprocessor and a specification language. The language is used to specify the interaction requirements of the components that make up an interactive graphical application. The specification incorporates the new control constructs. The preprocessor, uses such an interaction specification to generate the interaction 'C' code in the form of the new model.

Applications designed and prototyped in this manner have been used to identify typical problems in interactive graphics design. Finally, the model has been critically compared with a number of other interaction models.

To Ravi

Contents

Chapter 0. Introduction	1
Chapter 1. User Interface Design	4
1.1. Interactive graphics programming	4
1.2. Building user interfaces	9
1.3. Specification aspects of user interfaces	20
1.4. Modelling the user design process	29
1.5. Window systems and user interface toolkits	30
1.6. This work	31
Chapter 2. The New Interaction Model	34
2.1. Existing models	34

2.2. The New Interaction Model	38
2.3. Formal and informal approaches to user interaction	47
2.4. Summary	53
Chapter 3. Implementing the New Model	54
3.1. The need for a design language	55
3.2. System design	57
3.3. An example application	60
3.4. The SIDL syntax	62
3.5. SIDL program structure	62
3.6. Code generation	68
3.7. Summary	70
Chapter 4. Using the New Interaction Model	71
4.1. Task synchronisation	71
4.2. Parameter collection	75
4.3. Task management	83
4.4. Multi-button pucks	87
4.5. Window management	88
4.6. Summary	89
Chapter 5. Comparison of SIDL with Other Models	90
5.1. The state diagram method	90
5.2. The object oriented approach	97
5.3. Summary	105
Chapter 6. Conclusion	107

References	111
Appendix A. The SIDL Syntax	119
Appendix B. A SIDL Program	125
Appendix C. Publication	132

Acknowledgements

I would like to take this opportunity to thank my supervisor Phil Willis for his considerable support and guidance during the research. My thanks also to Geoff Watters for the many useful discussions, Russell Bradford for solving some of the typesetting problems and to GEC Software for providing the document production facilities.

I wish to record a special note of thanks for my parents who have consistently encouraged me during these last few years. I owe them more than I could ever express in words.

0. Introduction

The advent of the bitmapped high resolution graphics workstation has led to a trend of developing graphics interfaces for existing applications. More importantly, interactive graphical applications have become widespread. Unfortunately, the attendant problems of scheduling mechanisms; interpreting mouse/tablet input; screen layout and interrupt handling associated with such applications have all been dealt with in a very unsatisfactory manner, each programmer solving the problem in an individually stylised fashion.

Research in graphical interaction has concentrated on the study and development of user interface management systems (UIMS), ranging from theoretical studies to practical systems. Automatic generation of user interfaces has relieved the programmer of some of the problems mentioned above, but still fails to ease the complexity of programming the basic building blocks of these applications. These

building blocks include the fundamental interaction techniques such as picking, dragging, and rubberbanding. The asynchronous, multiple-processes nature of graphical programs continues to be a major hurdle that consumes a programmer's effort.

Software development in most other computing fields follows some sort of methodology. There is however, no equivalent technique for specifying the building blocks of graphical programs.

We present a methodology for programming the interaction techniques used in graphical programs. The methodology was derived by examining a number of interactive graphics applications and it was tested by developing a software tool to produce improved versions of those applications. The results of this research have subsequently been published[7].

Thesis structure

In chapter one we introduce the notion of interactive graphical programming. We indicate the nature of the problems associated with such applications and show how in an attempt to solve such problems a new genre of software tools has arisen. The architecture of these tools is described together with examples. We touch on the problem of modelling the user thought process and and we conclude with a section on how the research in this thesis fits in with work done earlier.

In chapter two we describe the basic scheduling strategy used to implement interactive systems. This scheduling strategy is termed the **Interaction Model**. We then go on to describe the shortcomings of existing models. The second half of the chapter is a detailed description of a model which more closely fits the characteristics of this type of software. This model draws freely from both operating systems design and compilation theory. Finally the model is compared with a formal description of an

earlier model proposed in some closely related research.

Chapter three describes the practical realisation of the interaction model. The Language SIDL and its pre-processor GRIP are described in detail.

In chapter four, we discuss some of the problems that are typically found with interactive graphical programs. These problems are discussed from the baseline of our interaction model proposed in chapter 2. The solutions for these problems are presented. In addition, we look at some additional problems which arise because of our model. These can be regarded as the inherent disadvantages of using our model. The discussion is illustrated with a mixture of example code generation and pseudo-code.

Chapter five is a detailed study of two methods mentioned in chapter one. Emphasis will be given to a specification language for interface design based on transition diagrams and secondly to object-orientated techniques for user interface design. These methods will be compared with the approach taken by the Interaction Model described in chapters three and four.

Finally, in chapter six we outline the measure of success of this work and present some possible future directions.

1. User Interface Design

In this chapter we introduce the notion of interactive graphical programming. We indicate the nature of the problems associated with such applications and show how in an attempt to solve such problems a new genre of software tools has arisen. The architecture of these tools is described together with examples. We touch on the problem of modelling the user thought process and and we conclude with a section on how the research in this thesis fits in with work done earlier.

1.1. Interactive graphics programming

Interactivity is a mode of execution of an application. For teletype terminals an interactive program is characterised by the dialogue (typically prompts and responses) between the user and the application. We are concerned with interactive graphical programs. In these applications the application communicates via graphical entities on the screen whilst the user directs the control of the application via a number of

physical input devices.

1.1.1. Characteristics of interactive graphical programs

A graphical application can be described as interactive if we can observe the following characteristics.

1. User interaction is composed of input/output. The user inputs data to the program which then proceeds to use this data to execute some action. The program communicates with the user by displaying objects on the screen. Data flow is bi-directional and the communication between user and program is closely coupled.
2. Interaction is composed of a number of input techniques. These are typically determined by the available hardware and they are described in section 1.1.3 of this chapter.
3. The user controlling the input devices and the computer process are independent of each other except when the program needs data from the user or the user needs a response from the computer. Thus there is both logical and physical concurrency. This necessitates the need for synchronisation to effect information transfer.
4. The passage of time is of significance. Many interactive input techniques are based on sampling some device for a particular parameter. Some common examples are rubberbanding and dragging objects. It is important to note that the actions of the user affect the program in real time and hence synchronisation is needed.

1.1.2. Input devices & interaction techniques

An application's "interactivity" is constrained by the number of physical input devices it has available for the user and also the way they are used. Physical input devices

include: joysticks, lightpens, puck and tablet, keyboards and so on. These devices need to be used in a structured manner so that the design of an application is straightforward both to implement and to change. Both the European Graphical Kernel System (GKS) standard[27] and ACM Siggraph Core standard have proposed the following structure for implementing the interactive components of graphics programs. Terminology has been adapted from the GKS standard because the GKS proposal is now the accepted ISO and ANSI standard.

Graphical input uses the concept of **logical input classes**. Physical input devices operate under these classes.

Locator provides a position in cartesian coordinate system.

Stroke provides a sequence of positions.

Valuator provides a real number.

Choice provides a non-negative number indicating a particular list item.

Pick provides a pick status, a graphical object name.

String provides a character string.

Each logical input device can be operated in three modes. At any one time, an input device is in exactly one mode. The modes supported are given below. The relationship between logical and physical input devices is explained using the concepts of measure and trigger[51]. See Figure 1.1.

Each logical input device contains a measure, a trigger and an initial value. A measure is the state of an active process which is available as a logical input value. The measure process is in existence while an interaction with the logical input device is taking place.

The trigger of a logical input device is a physical input device, with which the operator can indicate the significant moment of time to take over the measure value. At these moments the trigger is said to fire. A trigger can be seen as an active process

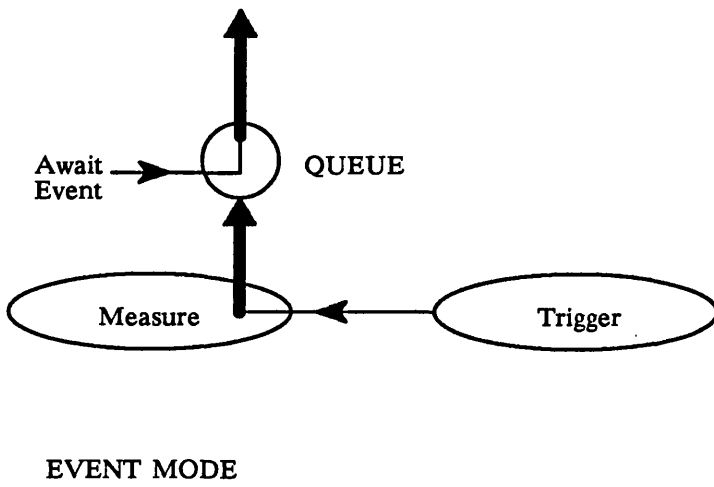
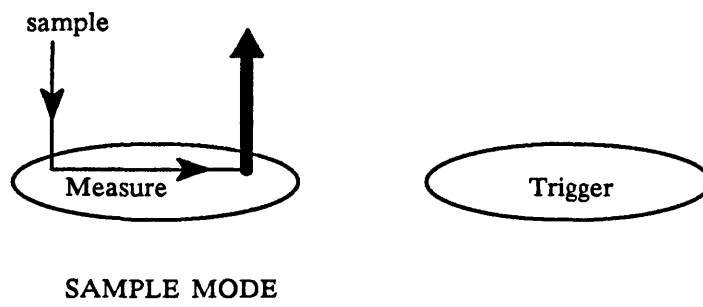
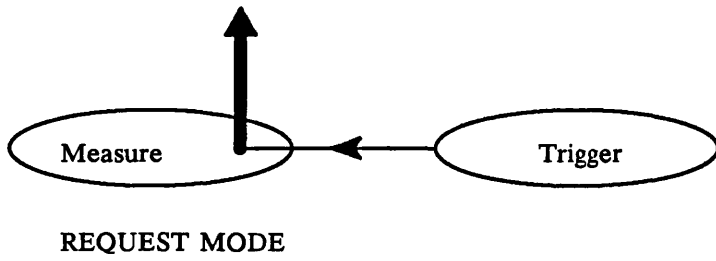


Figure 1.1. Logical Device Input Modes (From [51])

sending a message to one or more logical input devices when it fires.

Request Mode

The device only supplies data when requested. The application ceases computation until the data has been supplied.

Sample Mode

The device is continually supplying data at set intervals. There is no trigger action.

Event Mode

When a device is triggered, an event report is issued, and if there is data it is either used or queued. The program does not stop as in request mode.

Physical devices fall into one or more of these classes. Thus examples of locators are :- puck and tablet, mouse. For most applications the interactivity can be extended quite dramatically by simulating different logical classes using devices which do not belong to that logical class. Thus the locator can be simulated by arrows on the keyboard. Choice can be implemented by using light buttons on the screen and a pick device.

1.1.3. Interaction techniques

A good user interface usually means that the appropriate interaction technique must be used for the job in hand. A brief overview of some the common techniques used is given below. These tasks directly modify the graphical image. They are classified as controlling techniques[18] because their purpose is to form and transform visible objects, usually by continuous modification.

Stretching

A target object (such as a line or circle) has its shape distorted by forcing one of its points to coincide with some specified position. Positioning is a key component and locator simulation is used. Generally this technique is more usefully employed using continuous rather than discrete feedback.

Some examples are:- *Stretched lines*. The stretched or "rubberbanded" line is a task that maintains a line from some fixed reference point to point specified by a locator position. As the latter point is moved the line is modified to follow. The resulting effect is similar to a rubberband being stretched. Some refinements include constraining the rubberbanding effect to horizontal and/or vertical lines.

Other variations of stretching include *rubber rectangles*, *rubber circles* and *rubber pyramids*.

Sketching

This task involves the specification of a curved line. The significant characteristic of this task is sampling, since this task depends entirely on continuous feedback. The application determines either time sampling or space sampling. A line can be sketched using a lightpen or mouse, or it may be *shaped* between fixed points by adjusting the curvature using splining techniques, such as B-splines.

Manipulating

Operations are performed on a visible object such that the appearance of the object remains the same but the position and orientation are altered.

Dragging occurs when a user picks or locates a graphical object and causes it to coincide with some position on the screen determined by him. For example a circle on

the left hand side of the screen may be picked and moved across to the right hand side.

Twisting occurs when an object is caused to rotate along a pre-determined axis. The degree of twist is specified by a valuator device.

Scaling occurs when a valuator scale is manipulated causing the object selected to alter its size.

Shaping

The task moulds an object until it reaches a desired shape. In interactive graphics systems shaping is dependent on information held internally. For example lines may be internally represented using control points as in the case of B-splines. Thus the shaping relies on manipulation of these control points. Control points may be dragged to new positions and smoothing functions applied.

Other techniques

Some other techniques which do not fit so well in the above schema include:

Gravity fields. Here lines or other objects have an area around them which is sensitive to selection. Thus selection of the endpoints of a line is made easier, because the area of error is that much greater. This illustrated in figure 1.2.

Gridding. Here a grid is drawn over a region. Subsequent actions such as selections and sketching are constrained to lie either on the points where the grid lines cross or the regions between the grid lines. Gridding is useful for diagram editing for example.

1.2. Building user interfaces

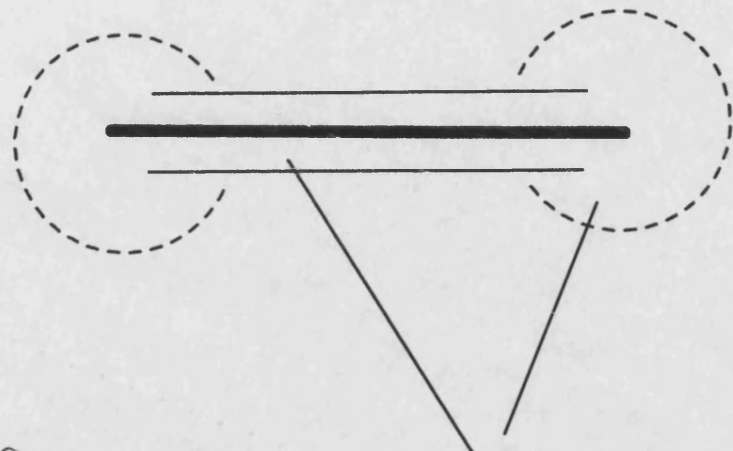


Figure 1.2. Gravity Fields

Gravity Field

The increasingly widespread use of personal workstations by non-computer trained professionals has led to the need to develop user interfaces for advanced applications. User Interfaces are one example of interactive graphical applications. They show the characteristics discussed earlier and their internal structure is composed of the interaction techniques described earlier in section 1.1. User interface design is a well researched area and the need for some classification and definitions of the needs and issues arising from user interface study must be met. The following report on existing work attempts to meet this requirement.

1.2.1. User interface definitions

The User Interface (UI) is the component between the user and the rest of the system. The rest of the system may be just an application or include the overall system hardware. Figure 1.3 highlights the main features of a User Interface.

Software Engineering principles dictate that the user interface should be a separate module which handles all the interaction between the user and the application. This separate module has been termed the **User Interface Management System (UIMS)** in 1982[36].

There are several advantages to defining a UI as a separate entity from the application and the graphics package. Firstly, development, maintenance and future extensions are made easier if the systems are defined in a modular layered fashion. Secondly, device and application independence is useful in that the same interface may be used across a variety of different hardware and applications. Finally, both the application and the user interface can be developed independently of each other.

Writing interactive graphical programs using conventional programming languages is both awkward and time consuming. A UIMS is designed to overcome these shortcomings. Its purpose is to support the design/specification, implementation and

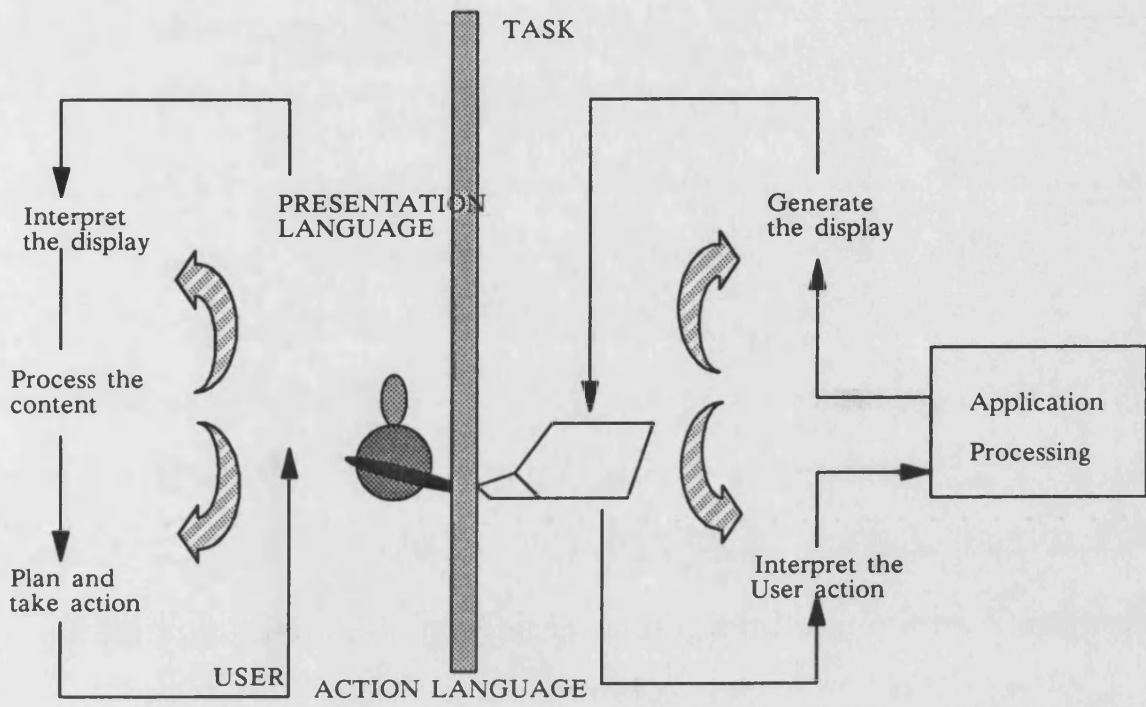


Figure 1.3. The Basic User Interface

evaluation of human-computer dialogues. Adapted from Olsen et al[46] the goals of UIMS designers are typically:-

1. reduce the duplicity of code across applications;
2. use a common module to make a uniform interface both within and across applications.

The design of existing UIMS have largely been based on the following principles[54] in addition to those mentioned above.

1. All variations of dialogue styles should be supported.
2. UI design is an iterative process based on design-implementation-evaluation.
3. Design should be ideally carried out by experts in human factors rather than programmers[19].
4. The effect on the application for which the interface is being developed should be minimal.
5. The Tools should render complex interfaces maintainable, extendable and easy to use.

Tools that are typically provided by the UIMS include:-

- a graphics package;
- standardised graphics communication protocols;
- a runtime support environment for the user dialogue;
- dialogue creation tools.

1.2.2. An abstract model for UIMS

A UIMS is essentially composed of two modules, a preprocessor to design and build the user interface and a run-time support package providing the framework within which the user interface will execute. The UI definition file will contain the state

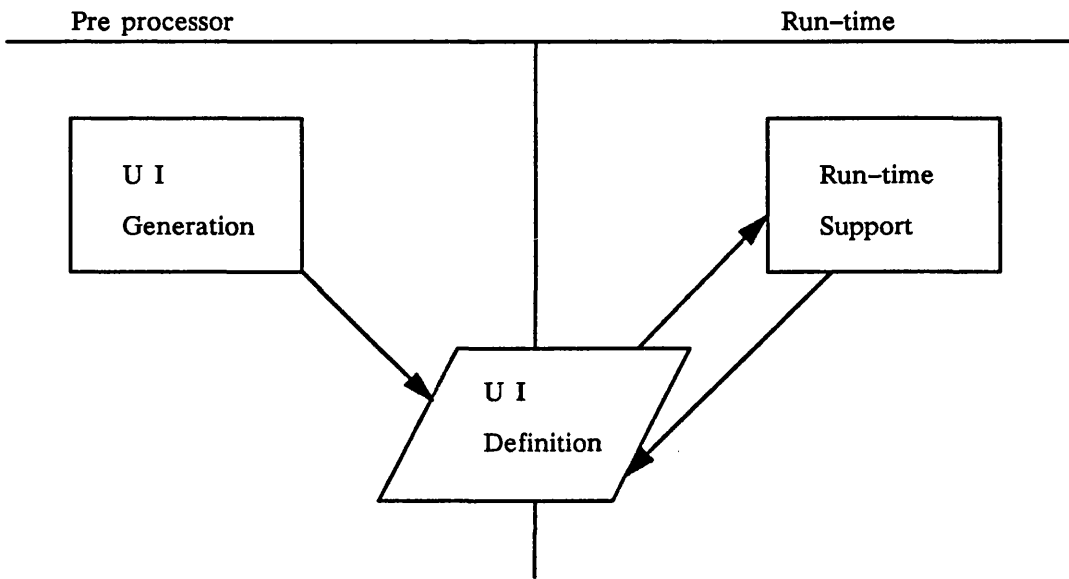


Figure 1.4. Requirements of a User Interface

transition information[54]. Figure 1.4 illustrates the basic requirements of a user interface.

A UIMS goal is that it should be separate from the application but the framework within which the UIMS will operate is largely affected by the relationship between the user and the application. Three frameworks have been identified. They are the External, Internal and Concurrent[23] frameworks. They correspond to the event, sample and event/sample modes of graphics standards such as GKS.

External

Application procedures are invoked in response to user inputs. Thus the user is in control.

Internal

The application has control, it requests various abstract devices when required by the application.

Concurrent

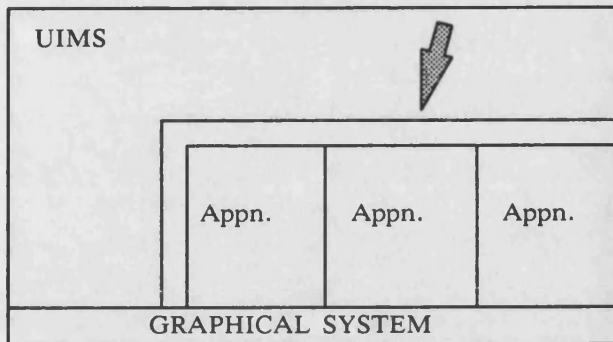
This has a mixture of the internal and external frameworks.

Figure 1.5 illustrates the various frameworks for UIMS design.

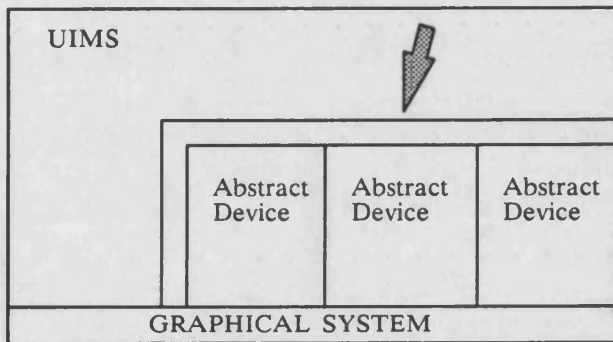
We have already outlined the basic structure of a UIMS. Workshops have since identified a more detailed structure of a UIMS. From a study of existing UIMS, Tanner & Buxton present a notional model based on 'Glue Systems' and 'Module Builders'.

1.2.3. Glue systems

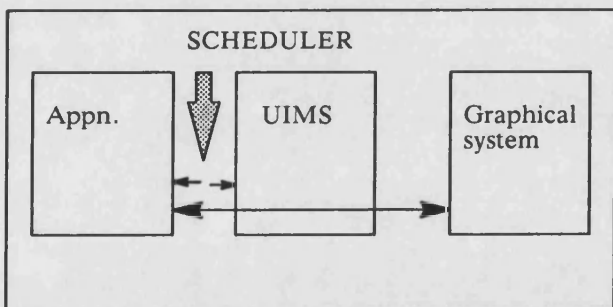
In glue systems interactive dialogues are created by prepackaged tools; suitable dialogues are selected by the user/programmer from a library. The UIMS provide access to the library and a general support environment to bind the modules for creation of the UI. The power of these systems lie in the size of the library. An



External Control



Internal Control



Concurrent

Figure 1.5. Frameworks for UIMS Design

example of such systems is MENULAY[9] and Dialogue Cell[26].

1.2.4. Module builder

These UIMS are concerned with the specification and implementation of the low-level interaction primitives used in a dialogue (module). The modules are collected together to form a library. Typically they are based around some special purpose programming language and because of that they have greater expressive power but are that much harder to learn. Examples include SYNGRAPH[47] and TIGER[36]. In Systems such as the Mackintosh Toolbox and most Window Managers the routines are the library portion. See section 1.5.

A novel approach to building the interaction has been put forward in Squeak[11]. Squeak is a user interface implementation language that exploits the essential concurrency among multiple interaction devices. The language is based on concurrent programming constructs that are compiled into conventional C. Squeak programs are composed of processes executing in parallel. A process typically deals with a particular action or external device. Communication between processes is achieved by sending messages on *channels*. Channels are either *primitive* or *non-primitive*. Primitive channels are predefined and provide access to external devices. Non-primitive channels are for ordinary message based communication.

A Squeak program is compiled by analysing all the possible execution sequences of the program and expanding them into C code. There is no scheduling on the user channels: scheduling and communication is translated into sequential code interleaved with random choices and calls to the underlying event manager.

Parallel sequences are decomposed by advancing one of the processes by one step and considering all the possible continuations of that and all the other processes. The entire system state is then returned to the initial and the step repeated for another

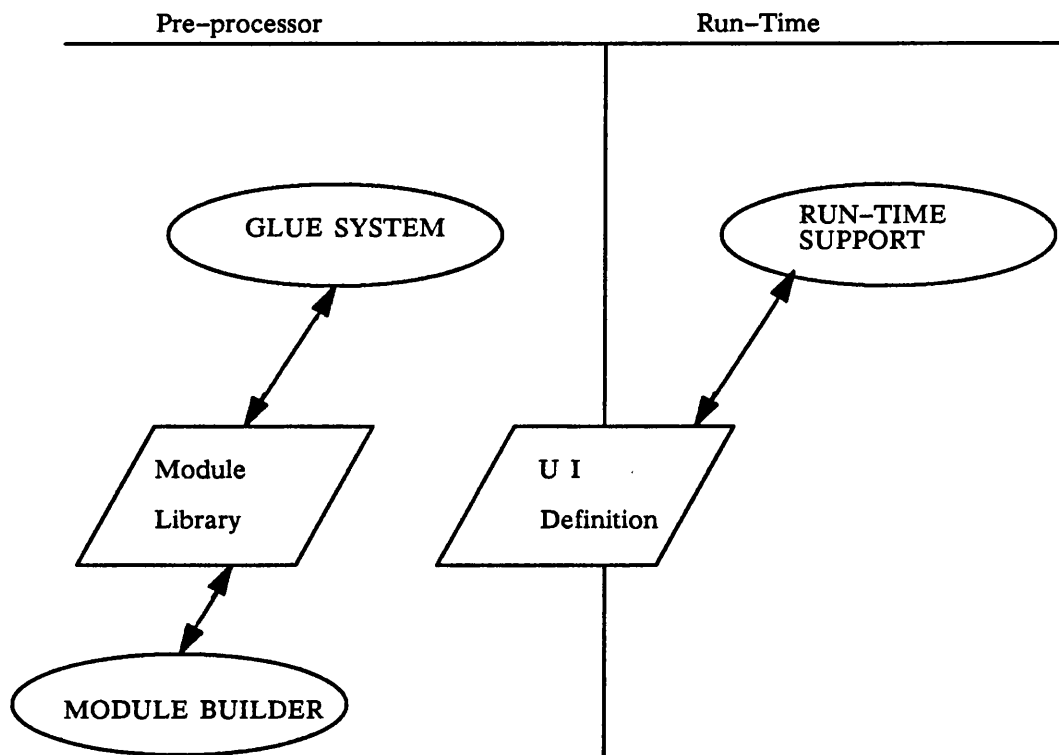


Figure 1.6. Combining Glue Systems and Module Builders

path. There is some 'pruning' and 'flattening' to avoid deadlocks and redundant paths. The remaining available paths are compiled as a dynamic random selection of which path to take. Young et al[59] rightly point out that Squeak

"...suffers potentially explosive expansion when compiled into C code, effectively limiting its usefulness to the lowest levels of input processing. "

Glue Systems and Module Builders can exist together. A UIMS that falls into that category is PERIDOT[43]. Figure 1.6 shows how glue systems and module builders can be combined together.

1.2.5. The Seeheim Model

This model for UIMS design was a result of working discussions held at the Seeheim Workshop on User Interfaces and is discussed in detail by Green[23]. Members of the group were Jan Derksen, Ernest Edmonds, Mark Green, Dan Olsen and Robert Spence. It is now the main model which has been put forward for future design of UIMS because of the advantages of modularity and the incorporation of design concepts arising from experts in human factors. A description of the model is given below.

The User Interface is divided into three components as shown in figure 1.7 below. They are the Presentational Component, Dialogue Control and the Application Interface Model.

The transformation of the communication between the user and the application across this model can be discussed in terms of the language model introduced by Foley[19]. The language model is a convenient representation of the levels of interpreting or transforming from the external to internal representation and vice versa. The model regards a transformation as a process which can be broken down into a sequence of steps: lexical analysis, syntactic analysis, semantic analysis, and conceptual analysis.

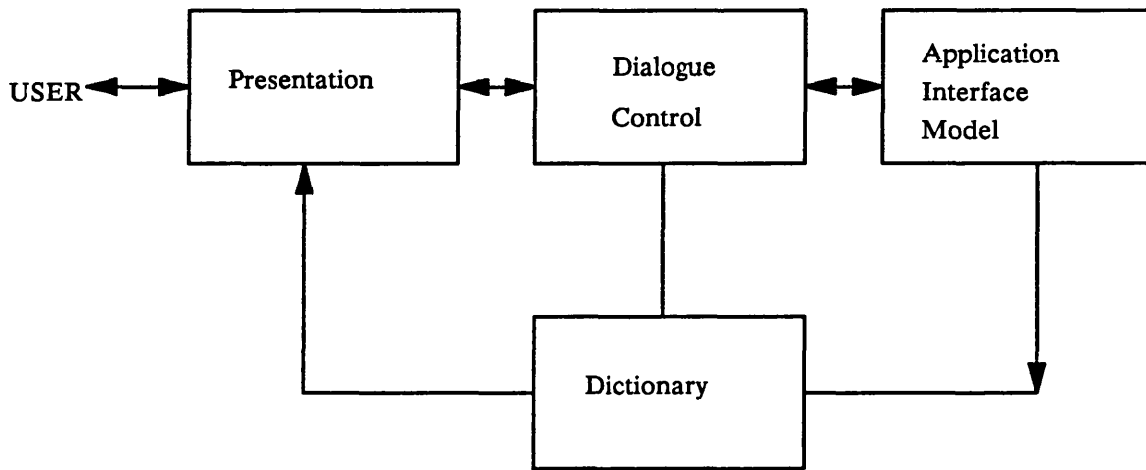


Figure 1.7. The Seeheim Model

It is useful to provide a brief description of these terms.

Lexical Analysis determines how input and output tokens are formed from the available hardware primitives (lexemes). Input and output lexemes are produced by the input and output primitives respectively. Lexemes are the smallest units of input which can be processed by the computer.

Syntactic Analysis deals with the **syntax** of the dialogue and defines the sequence of allowed inputs to and from the application. The **syntax** is the set of rules used to organise and order input and output lexemes. **Abstract Syntax** is a generalised abstraction which can be represented by state transition diagrams for example. **Concrete Syntax** is the set of concrete objects actually used in the syntax, for example menu buttons, etc.

Semantic Analysis is a description of the functionality of the application. For example it will define what information is needed for each operation on an object, how semantic errors are to be handled and other similar context sensitive interpretations.

Conceptual Analysis defines the more abstract aspects of the interaction. It defines the key concepts which must be mastered by the user. For example it will define which interactive graphical objects will exist and also their relationships to each other.

Presentational component

The Presentational Component describes the physical appearance of the interface. It reads the physical input devices and converts the raw data to a form usable by the other components. The menus are a special aspect of the presentation component. When a selection from a menu is made, the presentation component generates the appropriate token to be used by the other sections. Thus we can observe that the presentation component effectively carries out the lexical processing undertaken by the

UIMS. The notion of a separate module for presentation makes porting to different display devices simple. The component can be easily changed to allow for a variety of input devices. The screen layout can be altered to suit the user, for example switching from a lefthand screen layout to a righthand layout.

Dialogue control component

The Dialogue control contains information relating to the dialogue between user and the application. It converts tokens generated by the presentation component to execute commands or passes back tokens from the application interface model to the presentation component for visual feedback.

Only the dialogue component has access to what is essentially the dictionary component. The dictionary is used to convert the lexical tokens received from the presentational component to tokens which the application interface model can understand.

This area of UIMS design has been the recipient of considerable research effort. As a consequence there is considerable expertise in this area. A number of notations for representing the user-computer dialogue have been developed. They tend to fall into three basic categories:-

- Context Free Grammars;
- Recursive Transition Networks;
- Event Notations.

These notations will be described in more detail later on in this chapter and some examples given.

Application interface model

The application interface model describes the interface between the UIMS and the rest of the application. It holds the semantics of the application, the procedures of the application which are available to the UIMS. This component exists only implicitly in UIMS built before 1984. It is also in this module that the framework in which the UIMS is to operate is defined. The external or internal model of execution is normally proposed as an acceptable framework. Also at the Seeheim workshop a third framework **Mixed** was proposed. There are essentially two communicating processes for the user and the application. Some method of interleaving the execution of the interface or the application is required. It could be implemented by co-routines or multiple processes.

1.2.6. Examples of UIMS

A brief description of a number of key UIMS is given below. The list of UIMS is not exhaustive but is meant to give a flavour of what a UIMS entails beyond the theoretical structure discussed earlier.

Menulay

This is part of a UIMS[9] which aims to aid the design and implementation of menu driven programs. It consists of two modules.

The first module, Menulay and Makemenu, is essentially a preprocessor to design and specify the graphical layout and functionality of menu-driven programs. It allows the construction of networks/hierarchies of menus and provides hooks for application specific procedures. Menulay is itself menu-driven. It creates textual items such as light buttons and icons and allows their size and colour, for example, to be modified interactively. It generates a metacode which is input to an ancillary program

Makemenu which generates C code which will allow the incorporation of user supplied function procedures.

The second module, the runtime support package, handles events, hit detection, procedure invocation and management of the display. The code executed by this module is generated automatically from the preprocessor.

It can be observed that this UIMS falls neatly into the simple model of a UIMS. However it must be noted that there is no formal description of the input. The input language is specified by selecting interactive techniques from a library and incorporating them to form an interactive application. Thus Menulay is an example of a Glue system.

SYNGRAPH

The SYNGRAPH tool[45,47] supports graphical applications. The input language is a special form of BNF grammar. The menu and graphical valuator simulation is automatically formatted and all prompting, echoing and error reporting is also automatically generated. The modified BNF grammar has made possible a very powerful undo/rubbing out facility. Additionally, the system is designed within the framework of *Conceptual*, *Semantic*, *Syntactic* and *Lexical* levels as described in earlier sections. The conceptual level (i.e. the thought process of the designer leading to the initial design of the user interface) is not addressed. The semantic level is represented by the commands available in the interface. However the influence of the syntactic and lexical levels is greatest. At the lexical level there are the tokens which represent user input, for example button presses. At the syntactic level, a sequence of specific lexical tokens corresponds to a dialogue. A Pascal program is produced and there are no evaluation tools provided.

The University of Alberta UIMS

The goal of this research was to build a UIMS following the abstract model proposed at Seeheim. The UIMS[24] is divided into three components. The presentation component which is concerned with the lexical level of the user interface is implemented by a window based package WINDLIB. Screen layout, interaction techniques and displaying are supported by an interactive layout program. The Dialogue component supports all notations. However to achieve this flexibility the underlying format is event notation as it has greater descriptive power. There are tools to edit transition diagrams but at the moment there is no facility for going from BNF to event form. This is the subject for current research. The application interface model currently supports only one mode (User initiated).

Other UIMS

Other UIMS research efforts include the information display project at George Washington University. The Abstract Interaction Handler[35] (AIH) has a number of components:- an interaction language adapted from augmented transition networks; an interpreter for that language; a set of 'style' modules to handle interactions which are style dependent, for example levels of prompts; a library of user profiles and interaction techniques; and finally a logical screen handler.

User interfaces have been built using the 'programming by example' paradigm as in Tinker[39] and Peridot[43] In case of the latter, a variation on the theme is used - 'programming by demonstration'. The Higgens system[28] has concentrated on efficient recovery and reversal in user interfaces, a hitherto under-researched area. They have developed a special model which together with algorithms they have also developed makes undo very cost effective. An object oriented approach to user interface design has been developed by Lieberman[40] and in the GWUIMS[53].

Object oriented design will be considered in greater detail in later chapters. The concurrency aspects of User Interface design have been utilised in Squeak[11] and the formal semantics of these aspects described.

1.3. Specification aspects of user interfaces

Formal Specification is now widely used in the field of software engineering because it allows the designer to describe the external characteristics of system without going into its internal structure. Formal specification of a system provides a source of reference for both implementors and users. It is implementation independent and provides a basis for proofs of correctness. In addition it allows the precise formulation of queries and answers. In the traditional software cycle of :-

Requirements - Specification - Design - Implementation - Testing - Maintenance

formal specification covers requirements, specification and design.

Computer Graphics in general, has been the subject of a number of formal studies. GKS is an example of an attempt at formalising graphics using English. Mallgren[41] has formally specified hierarchical picture structures and user interaction using algebraic techniques. His interaction specification will be the subject of further discussion in chapter three. Duce[17] uses VDM to formally specify the user interface.

Formal specification methods have been applied to user interface design in a quite comprehensive manner. In terms of the UIMS models described earlier, the dialogue component has been the subject of most study. Broadly speaking most dialogue specifications have fallen within the categories mentioned earlier. These are :- BNF, state transition diagrams and event notations. Sections 1.3.1 to 1.3.3 describe the three techniques in detail.

1.3.1. Context free grammars (BNF)

The theory of these grammars will not be discussed as it can be found in any standard compiling theory text[2]. The underlying motivation for this model is that human-computer interaction is a dialogue as in human-human communication. For both cases there must be an agreed syntax if communication is to be effective. However, in the latter case only one common language is used whereas in the case of the former the human uses one language to describe the user's actions while the computer use another language to respond to the user's actions. Thus the model attempts to unify these two different approaches to communication.

The terminals in these grammars are the input tokens from the presentation component. These tokens represent the user's actions. The non-terminals and the productions represent structure or syntax of the dialogue. There could be a command associated with each non-terminal. Each production with the non-terminal on the left hand side defines the syntax of the command. For example a BNF grammar for a login command (shown in Figure 1.8a) could be:-

```
login      ::=  userid password
userid     ::=  <character string>
password   ::=  <character string>
```

This does not cover the response generated by the computer. As we shall see later, program responses for state machines are attached to the arcs so, correspondingly, program actions can be attached to each of the productions in the grammar.

One problem with this approach is the time when the production is used (R.H.S). Bottom up parsing uses the production when all the symbols on the right hand side have been used, top down parsing uses the production when the first symbol on the

right hand side is met. In the examples presented here we assume a top down parse. The second example describes the context-free grammar for a rubber band line. The productions have been augmented by program actions which take place at certain places during the dialogue. This modification has introduced three additional rules (d1, d2, d3).

```

line      ::=  button d1 end_point
end_point ::=  move d2 end_point
            |  button d3

d1        ::=
            { record first point }

d2        ::=
            { draw line to current position }

d3        ::=
            { record second point }

```

Reisner[49] provides an example of how BNF can be used to describe a user interface. One problem with this technique is still unresolved. Sometimes it is difficult to determine *when* something will occur, consequently it is not easy to handle how the output tokens are to be produced. Thus error messages, prompting etc. are only possible with the inclusion of largely unnecessary non-terminals and productions. Olsen and Dempsey[47] have used BNF grammars for the construction of the user interface for SYNGRAPH. They have allowed the designer of the dialogue to specify an error recovery mechanism allowing for more natural and graceful error recovery.

1.3.2. State transition diagrams

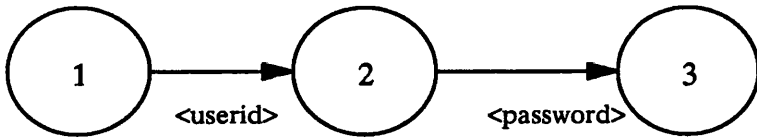
This is the oldest notation used and its history can be traced back to Conway[12]. He did not address the problem of interactive user interfaces. Parnas in 1969 proposes

the use of state diagrams for describing user interfaces[48]. Newman[44] in 1968 however, had already put this into practice in the seminal work 'The Reaction Handler'. Jacob[30] used this technique for a military message system in 1983 but in this case there was no interactive graphics requirement.

A State Transition Diagram is a collection of directed graphs. Each graph contains a set of nodes (states) and a set of arcs (state transitions) representing the actions a user can perform at a given stage of the interaction. A dialogue goes from one state to another if a user performs the appropriate action (the arc label). The computer's side of the dialogue is described by adding actions to the states. Figures 1.8a and 1.8b provide some examples of state transition diagrams. The number of arcs/states needed to represent all the states an interaction can go through can be quite large. This problem has led to a number of partitioning schemes, typically subnetworks. An extension of this is the use of recursion giving Recursive Transition Networks (RTN). Partitioning and recursion make the RTN equivalent in power to BNF but they also introduce non-determinism which should be treated with care in interactive systems design as usually it is not possible to retrace a path that has generated output for the user.

Another problem with this approach is handling unexpected user actions. The easiest course is to ignore the user input. This is obviously not suitable. A second method is to introduce a wild card. This matches all user actions not catered for. Such an action leads to a state where error recovery can be attempted.

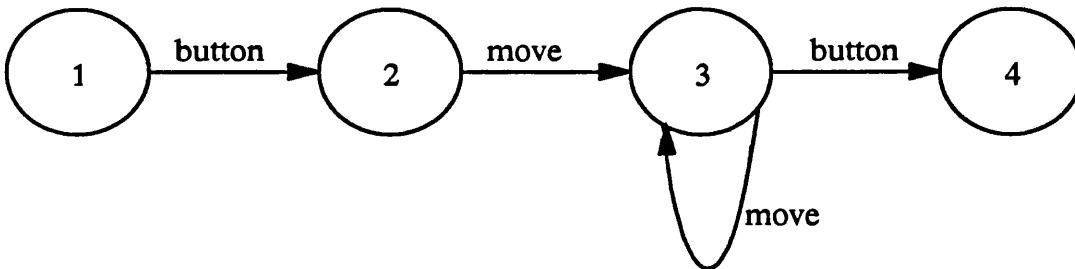
Kamran uses a variant of the RTN, the Augmented Transition Network (ATN) for the model on which he bases his dialogue control[35]. An ATN is a network with a global data structure attached to it. Functions can be assigned to arcs and their results stored in the data structure. An ATN has significantly more computational power than a normal transition network. It is important because it can implement context sensitive dialogue, for example the data structure could hold information concerning a database



Actions:

- 1) print 'login'
- 2) print 'password'
- 3) print login

Figure 1.8. a State Transition Diagram for the Login Command



Actions;

- 2) record first point
- 3) draw line to current position
- 4) record second point

Figure 1.8 b. Transition Diagram for Rubberbanded Line

and thus if a file is already open an appropriate message is given.

1.3.3. Event notations

This notation is not as well known as the notations described earlier. Each event has a name and a collection of values that characterize the event. An event originates from one or more input devices available to a graphics package or it can be generated by the application. These events are processed by a number of event handlers. An event handler is a special procedure which performs a set of actions based on the name of the event it receives. These actions could be calculations, output of tokens or newly generated events. Mallgren uses event algebra to describe user interfaces[41].

The procedural form of an event handler makes notations for events very similar to programming languages.

A major advantage of the event model is its ability to describe multithreaded dialogues. This is where a user is involved in more than one dialogue at the same time and is also free to switch between dialogues. The multiple dialogues are possible because all event handlers execute concurrently. Event models are commonly used to implement window management systems. In an abstract form, once an event handler has been created it is active until it is destroyed and only while an event handler is active can it process events.

The user interface is described by all the event handlers it uses and a special event handler serving as the main event handler in the user interface. The expressive power of event notations allows other dialogue control models to be expressed in some event notation. Green[25] takes advantage of this when providing multiple format dialogue component construction. The following example is drawn from Green[25].

Eventhandler login is

```

Token
  keyboardstring s;
Var
  int state = 0;
  string userid: password;

Event Init {
  print "login"; }

Event s: string {
  if (state == 0) {
    userid =s;
    state= 1;
    print "password";
  }
  else {
    password= s;
    state= 0;
    process_login_cmd(userid,password);
  }
}

```

In addition to specification methods for the dialogue component, a number of abstractions for both the user interface and interactive graphical applications have been developed. These abstractions attempt to model the user interface in terms of the user's view and behaviour towards the user interface. Sections 1.3.4 to 1.3.6 describe some examples.

1.3.4. BOX: A layout abstraction

Coutaz[13] identifies the need for dialogue independence and the expression of object-orientated I/O. He makes the assumption (based on current cognitive science theory) that applications reason in terms of specific abstractions (or objects) to perform a task. An object is made concrete by views of this object by specific agents. Thus an object on a screen is viewed by two agents, the user and the application which owns the object. Some views may be suitable for one agent and not the other, so an additional view, that of the **Box**, is introduced. This represents an intermediate view

that both agents are happy with. The box view contains a specification of the object which is used to format the object on the screen.

The box is only concerned with high level I/O. The application is only concerned with manipulating the object and not by the process by which the manipulation is done. A structural and spatial relationship is a second requirement. Thus objects may be composed of other objects. Dynamic modification of a box must also be possible. A box is specified by spatial and adornment attributes which determine the size and appearance of a box.

The method of specifying the box is left unclear, the examples given in the paper being insufficiently complex, and it seems that this particular tool is suitable only for the expression of relatively simple objects such as menus, slots and forms.

1.3.5. The dialogue cell

Another abstraction has been provided by ten Hagen and Derksen[26]. A new programming language construct called a **Dialogue Cell** is presented. It allows the separation of algorithms from the dialogue design and makes use of the extensive parallelism in user interface design. A dialogue cell is a unit which can completely specify one step in a dialogue. It has four components.

Prompt

Initialises sub-dialogues and informs the user it is ready to accept input.

Symbol

Specifies the syntax of the input sentence, and *how* the input monitor will treat input words.

Echo

The acceptance of an input by a dialogue cell. This is the method by which graphical objects are dynamically maintained on the screen. There are two types of echo. Local

Echo: Objects disappear once the dialogue cell is no longer active. Global Echo: Graphical objects persist on the screen.

Value

A value is associated with each symbol and thus attribute grammars can be accommodated.

Dialogue cells can either be activated via direct calling as in the Request mode in GKS, or as sub_cells by other dialogue cells. Input tokens can be collected for all active cells and thus there is a multi-stream parser which processes input against each syntax component for each dialogue cell. It does not matter in which order the inputs arrive because the parser can dynamically add and remove syntax rules from the currently active grammars. This allows for very flexible parameter collection.

The examples discussed so far have been mainly concerned with the dialogue component. The following examples again are related to this aspect, but they are also concerned with modelling graphical interaction at a more general level.

1.3.6. The device model of graphical interaction

The device model of interaction as discussed by Anson[4,5] describes an interactive system (or device) in terms of its component devices and data paths (channels). There are two views of a device. The outside of a device is composed of a visible state, events and actions, while the inside contains details of variables needed to implement the outside. Devices communicate with each other by reference to some aspect of the outside view. The binding of events to actions is a channel. A channel is an ordered pair with the source-event given first and the destination-action given last. For a simple user interface manipulating keyboards, lightbuttons, cursor position, the devices would be:-

1. tablet;
2. function keyboard;
3. cursor;
4. display.

Examples of Inputs and actions i.e. the channels are:-

(Tablet.PenPosition, Cursor.location)

(Tablet.PenDown, Cursor.Engage)

(Tablet.PenUp, Cursor.Disengage)

The extended notation described in the paper has been implemented as a functional equivalent in UCSD Pascal.

Gangopadhyay[20] has similarly provided a set of programming language constructs as an extension to Pascal. The constructs are based around events and event reports with associated actions. Physical devices are not *a priori* grouped into logical classes as in GKS. Instead an event of a certain type may be generated from any tool depending how the operator uses it. A set of event types is defined:

Selection Event;

Location Event;

Button Event;

Value Event;

Keyboard Event;

Clock Event.

The application program declares expected events and their expected interpretations. The procedural structure makes graphical programming straightforward. Gangopadhyay observes that a single operation on a tool can generate many invocations of several action processes. This can either be regarded as advantageous,

as he does, or it can introduce the notion of ambiguity. Since the application program declares expected events, dis-ambiguating facilities are available and hence the advantageous nature.

1.4. Modelling the user design process

Throughout this chapter human factors considerations have been implicit. It has been long recognised that research in modelling the interactive processes needed by the user will help the developer of a user interface make more informed specific design decisions regarding presentation and dialogue language. Coutaz[14] calls for advice from experts such as psychologists and graphics artists who have a better understanding of human behaviour than have computer scientists.

Card, Moran and Newell[10] have attempted to model the knowledge a user must have when carrying out tasks through the interface. The formal modelling of the user knowledge process seeks to develop a representation and method for constructing the representation so that early evaluation of user interfaces will allow economically feasible changes to interfaces. This type of study is currently at a very early stage[33]. However Green[22] made an early attempt and a brief description of this work is given below for completeness.

1.4.1. A methodology for user interface design

In Green's methodology there are two components, the first, the **User Model**, is a description of how the user views the task domain (the problem to be solved), and the second is a formal description of the user interface.

In the User Model, the informal task analysis (working out the requirements), is formalised using the concepts of *objects* (the entities in the problem to be solved), and *operators* (the operations that can be performed on the objects to manipulate them).

The formal description is called the *task model* and is the basis of the methodology. It is this aspect which is one of the earliest attempts to model the user thought process. Also part of the User Model is the *Control Model*. The Control Model uses the same notation but instead of describing the tasks to be performed it describes the commands that are available to perform them.

The control model provides a high level description of the user interface, but a detailed description is also needed. This is a specification language based on state machines. The more pertinent aspect of this work is the fact that the various levels of specification can be used to evaluate the interface. If there is no mapping between specifications then there is something missing and more detail needs to be added somewhere within the specification.

1.5. Window systems and user interface toolkits

Window Systems originated with Smalltalk at Xerox PARC. They have since been developing in two basic directions, for the UNIX environment (powerful, bitmapped graphics workstations) and for the lower end PC based systems.

PC-based systems include the Macintosh Environment and MS-Windows. These systems provide the illusion of multiple independent processes, when in fact there are not. Synchronisation is implicit in the single thread of control.

A window system is software that utilizes a graphics workstation by allowing a number of applications to run concurrently thus increasing the productivity of the user. It does this by dividing the screen into a number of regions or windows, where each window is controlled by one application. The system maintains the windows. The user is allowed to create and position windows at will.

The windows are laid out on the screen by one of two metaphors, desktop style or

tiled. The desktop metaphor more closely mirrors the working habits of a user as windows are allowed to overlap over each other, rather like pieces of paper on a desk. The tiled system does not allow overlap of windows, the windows are organised so that they make maximum use of the screen and thus windows will vary in size (but will still be rectangular).

Supplied with most window systems is a user interface toolkit. This is normally a library of pre-defined procedures to which a user can link to create interactive graphics applications. The facilities available with a toolkit vary with the type of window system. For example, a kernel based window system such as SunWindows has available with it the Suntools Toolkit[1]. the X window System has the Xtoolkit[50]. The essential feature of all the toolkits is that they contain interaction objects such as scrollbars, menus, button objects. Further, they contain facilities for the management of these objects. Thus they fall into the module builder category discussed earlier. However they do go a bit further in that some of the glue is also provided.

A common problem with such toolkits, is that they do not allow the sort of control and uniformity that is required when building complex applications. Further, there is lack of portability between toolkits even allowing for the common principles and thus an application developed under SunWindows will not work under X.

1.6. This work

Writing interactive graphical programs is notoriously difficult. Generally, this difficulty can be attributed to the asynchronous nature of user input; the lack of existing "structured" programming techniques; and not ensuring that the synchronisation of actions with user events is done within time constraints imposed by what is deemed "user friendly".

The research in UIMS and interactive language constructs has demonstrated that there is still as yet no ideal methodology. User toolkits commonly available with window systems allow easy construction of applications but introduce additional problems of portability and suitability of design procedures. Programming language constructs as exemplified by Gangopadhyay are not powerful enough, but at least he clearly identifies a need for programming techniques for these type of applications.

Recent years have finally seen the maturation of research in Integrated Project Support Environments (IPSEs). Typically, such an environment will include a number of software tools and it is essential that the tools have a standard user interface to reduce effort in learning new tools. A number of IPSEs embody this basic principle of consistency.

In the field of Software Engineering, the ideas of 'structured programming' have been with us for some time[15,16]. Various methodologies have been advocated, but they all rely on the premise that there exists a set of standard or conventional programming language control constructs which can be used to write a program. These constructs are:-

if ... then ... else;

while ... do;

repeat... until.

It has been shown that such restriction has resulted in a significant improvement in the quality and cost of code produced. Figure 1.9 gives an indication of the 'effort multiplier' factors for software utilising various levels of modern programming practice[8]. It is assumed that such practices are employed across the software cycle.

Methodologies which have proved their worth include Jackson Structured Programming[29] and Structured Design[61,42]. Identification of similar constructs

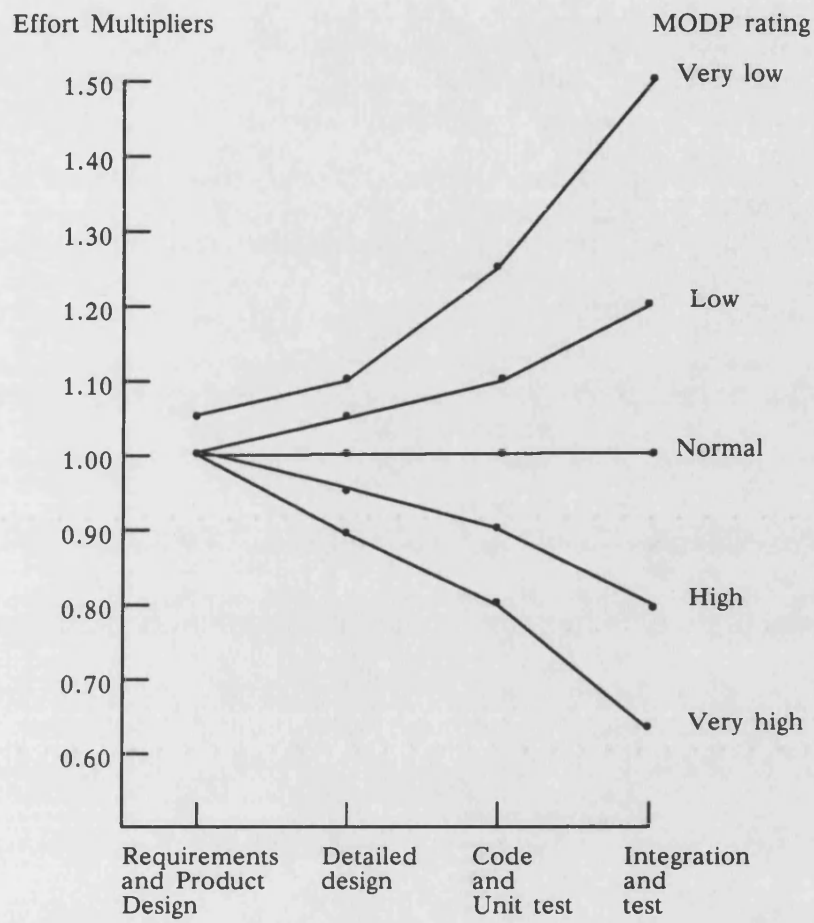


Figure 1.9. Effort multipliers by phase: Modern Programming Practices (From [8])

which embody the concurrent, and largely event driven nature of graphical programs is a prime consideration. This work identifies such constructs and it utilises and builds upon the experience gained in UIMS design. In addition it attempts to provide a prototype software tool which implements the constructs so derived.

2. The New Interaction Model

In this chapter we describe the basic control strategy used to implement interactive systems. This scheduling strategy is termed the **Interaction Model**. We then go on to describe the shortcomings of existing models. The second half of the chapter is a detailed description of a model which more closely fits the characteristics of this type of software. This model draws freely from both operating systems design and compilation theory. Finally the model is compared with a formal description of an earlier model proposed in some closely related research.

2.1. Existing models

Elsewhere we have stated that an interactive graphical program is essentially an event driven activity, where an event is a user action supplied via some physical input device. Typically the program responds to events and then waits for other events to occur. More often than not, the major portion of an application's execution time is

spent waiting for events to occur. This wait time corresponds to the user "Think Time" i.e. the user is thinking of some action to perform[19]. The typical schema for an interactive program reflects this wait time.

```
repeat
  Wait for an event;
  case event device of
    1: action 1;
    2: action 2;
    :
    n: action n;
  end;
until exit;
```

Await Event routines such as this allow several input devices to be in use at the same time. The program could be receiving input on some device and still be waiting for events on another device. Thus events will need to have their status preserved and so typically there is some queueing data structure. The result is a need for complex event-decoding logic. We will need to determine which event device generated the event and then search the queue for the event information.

Many applications however require input from only one device at a time. Also some time-sharing systems prohibit the simultaneous use of several devices so the generality of the event queue becomes a needless overhead. This leads us onto a slight variation of the above scheduling loop. Instead of waiting for an event on any device, there exists an event handler for each device attached to the system. Within each event handler the scheduler waits on the data rather than the device. The schema could be shown as:-

For a device A.

```
repeat
  wait for an event from device A
  case data from device A of
    1: action 1;
    2: action 2;
    :
    n: action n;
  end;
until exit;
```

This basic schema is implemented for all devices used by the application. An example illustrating an event handler for each device is now described.

Example

An interactive menu-driven program to draw boxes and lines at user supplied positions on the screen. The basic input/output hardware configuration is:- display device, tablet and mouse. The user's actions are composed of the following sequence of events: a button press in the menu area (corresponding to selection of a menu option) followed by button presses in the drawing area (defining the opposite corners of a rectangle or the vertices of a line).

For the program to carry out the required function, it needs to monitor the interactive element by:-

1. checking for events in the menu area;
2. checking for events from the point device, in this case the mouse.

Finally the raw input data needs to be processed into useful information. The user and program actions defining the interaction are summarised below.

User actions are:-

- a menu event;

- a point device event.

Program action are:-

- check for menu events;
- read menu event data;
- check for point device event;
- read point device event data.

An implementation of this is:-

```

program example;

var
  /* declarations */

begin

  Initialise picture area;
  Initialise the interface I_f.

  while true do
    begin
      while test_menu(I_f)= empty do;
        cmd= readmenu(I_f);
        clr_menu(I_f);

      while test_point_device(I_f)= empty do;
        position_1 = read_point_device(I_f);
        clr_point_device(I_f);

      while test_point_device(I_f)= empty do;
        position_2 = read_point_device(I_f);
        clr_point_device(I_f);

      case cmd of
        line: drawline(position_1, position_2);
        box: drawbox(position_1, position_2);
      end;

    end;
  end.

```

There are two basic design faults with this model. Firstly, basing the design of the scheduler around the data means that a separate function to clear data, unused or used, needs to be provided. Ideally the event handler should simply return the new

data, calling clear operations as required and update the interface at the same time. Secondly, all synchronisation to maintain user action fluidity is performed by the sample program. The result is unstructured looping of the While statement before each read to ensure that data is available.

Finally as there is concurrent access to the interface by both the application and the user, there is a need for additional synchronisation requirements. These requirements are sufficiently complex to indicate the provision of special purpose interaction primitives in which the synchronisation needs of the user and the application are embedded. The design of a suitable scheduler and its primitives is now described.

2.2. The New Interaction Model

To develop our new model we start with the premise that graphical interaction is essentially an input-event driven activity. Generally interaction offers a rich set of options, most of which will not be in use at any given time. A small number may be intensively used (e.g. reading the tablet; updating the cursor position). Some will occasionally start other actions. Also, most actions require a rapid response to maintain fluid interaction.

Some definitions and associated terminology will now be introduced.

Application

A menu-driven graphical program.

Interaction

An application defined as tasks.

Window

A user defined area set within the screen of the display device.

Task

Within the constraints of graphical applications, a task can be defined as an object that provides a means of performing an action specified by the user.

2.2.1. The scheduler

The first component of the new interaction model is a scheduler. We can develop the scheduler by examining the role of tasks. A task can be seen as the conceptual unit of all graphical applications in that all such applications can be defined in terms of tasks. If an application is composed of tasks, a study of the interaction in the application can be undertaken by a study of the interaction of its tasks. A typical structure of an interaction application is given in figure 2.1.

Here $f_1...f_n$ are functions which a user may wish to execute. They are made active by appropriate selection of the function from the menu-window. Further, interdependence of functions is indicated by directed arrows. For example a task which sketches lines is dependent upon a task which allows the user to select inks. In terms of tasks, there is a one to one correspondence between a function and a task. More significantly, boxes labelled cursor-tracking and menu are also considered tasks, but they have a higher priority as they occur earlier in the hierarchical structure.

These observations suggested that interaction can be handled by a fixed scheduler scan with associated tasks. A large pool of worker tasks will typically be used, but most tasks will lie dormant until needed and expire once used. Further, parameter gathering should be separated from invoking the worker task, to maintain flexibility of interaction and simplicity of workers. In outline, the Scheduler is:

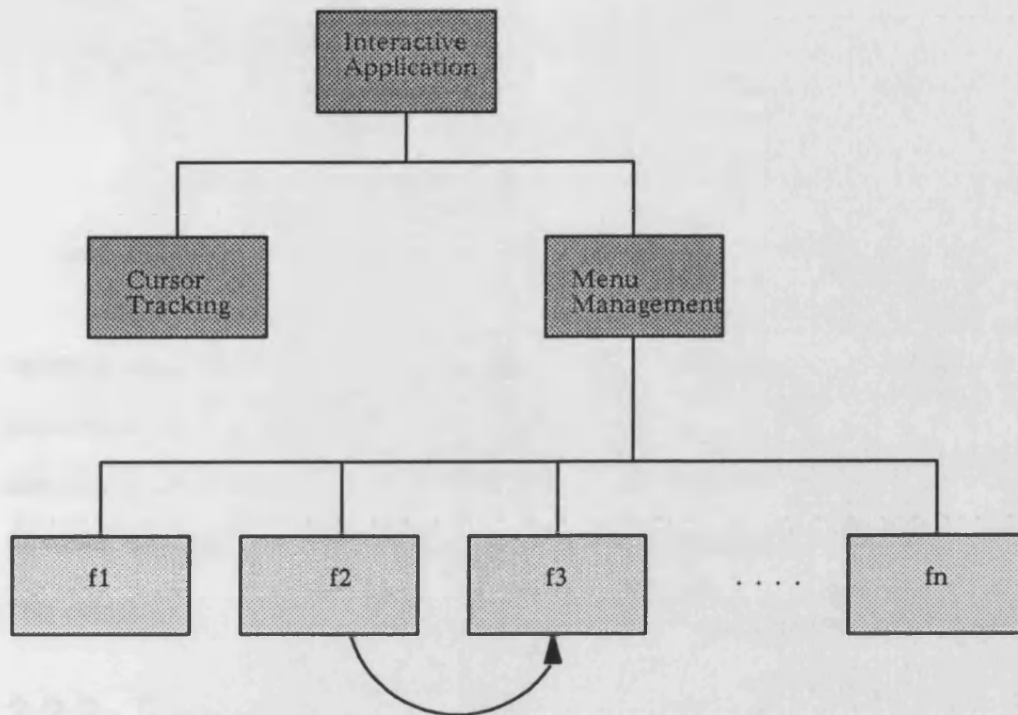


Figure 2.1. Structure of an Interaction Application

```
repeat
  for task:=1 to numtasks do
    if active[task]
      begin
        if not runnable[task] then Try(task,gather); /* Collect parameters */
        if runnable[task] then Try(task,perform); /* Perform the action */
      end;
  until exit;
```

During a single scan of the task pool, the scheduler attempts to collect the necessary parameters for any task t which is active but not runnable. If all the parameters for task t have been collected the task is made runnable. In the same scan, if task t is runnable, the task is performed.

This completes our introduction of the scheduler.

2.2.2. Tasks

The second component of the new interaction model is the Task. We require a Task to have two components. The first is a parameter-gathering section and the second is the code which performs the actions required of the task. It is immediately clear that the first component corresponds to the user interface, i.e. it contains the interaction requirements of the user and models the actions of the user. The second component is clearly the application interface. Correspondingly we need a mechanism for switching between these two components.

To implement this we use a procedure *Try* both to collect parameters and to execute the appropriate application-specific procedures. In outline *Try* is:-

```

procedure Try(t,op: Integer);
begin
  case t of
    1: case op of
      gather: ...
      perform: ...
    end;
    2: ...

    n:
  end;
end;

```

Each case on *t* introduces a section of code specific to the parameter gathering needs of the task. When called as *Try(t,gather)*, any parameters for the task are identified, but no interpretation is made of them. For example, a task to draw a straight line requires two pairs of (x,y) coordinates which will later be interpreted as the vertices of the line.

Try(t,perform) runs task *t* to completion by binding the parameters to a call of the appropriate worker task. We use this arrangement for convenience: the gather and perform sections of a given task are lexically adjacent, making for easier maintenance.

2.2.3. Task progression

The third component of the new interaction model is a pair of mechanisms to control the progress of tasks. The first of these controls the macro-behaviour of the task (the way the scheduler treats it) while the second controls its micro-behaviour (how the task executes).

For the macro-behaviour we note that, typically, most tasks are dormant until needed. Even when needed they usually pass through a parameter-gathering phase before completing and once more becoming dormant. We envisage a system (Figure 2.2) in which tasks progress from frozen (no need to do anything); to suspended (lacking enough parameters to run); to runnable (having a complete set of parameters and only

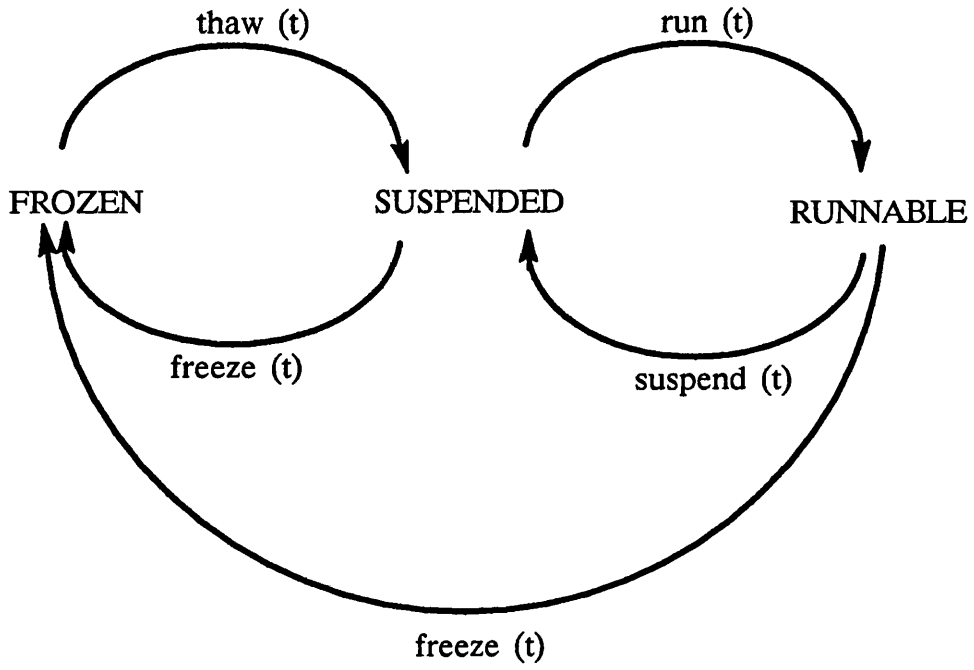


Figure 2.2 Transition of tasks between states

awaiting scheduling). We provide a task progression mechanism to reflect this. To implement this progression we provide appropriate control booleans and trusted primitives to manipulate them. For example, a task initiated from the menu progresses from frozen to suspended, remaining in this state until its parameter needs are met, when it subsequently becomes runnable. After being scheduled it might run to completion and then return to its former frozen state. Transition between macro-states is accomplished by the primitives used to label the arcs in the diagram in figure 2.2.

The primitives are of the form:-

```
procedure Run (task: Integer; progression: stage);
begin
  runnable[task]:= true;
  which[task]:= progression;
end;
```

```
procedure Suspend (task: Integer);
begin
  runnable[task]:= false;
end;
```

```
procedure Freeze (task: Integer);
begin
  active[task]:= false;
  runnable[task]:= false;
end;
```

```
procedure Thaw (task: Integer);
begin
  active[task]:= true;
end;
```

Now we consider the micro-behaviour of a task. Not every task which reaches *runnable* state will run to completion when it is scheduled. It may simply have reached a stage where a certain parameter may be collected. Progression to a later stage would then depend on that parameter being collected and any associated actions being performed. Hence we can impose a degree of sequentiality within a specific task. Thus we separate the *perform* section into three logical phases which we call *start*, *middle* and *finish*. The table *which* holds the current stage value for each task.

Typically this is useful because a new task has set-up actions which can be assigned to the start phase. The middle phase is used for the main part of the task and then the finish phase can be used to tidy up. To give an example, the outer phases can be used to change the cursor pattern back and forth to give feedback to the user, with the actual task being invoked in the middle phase. In the definition of **Run** there is an additional parameter *progression* which indicates the next stage of the current task to be processed.

Tasks do not have to freeze when completed. It is in the nature of some that they will gather one set of parameters, perform work and then repeatedly do the same thing. Such tasks can be suspended rather than frozen, as the diagram makes clear. In this way they continue to gather parameters as long as required.

This resemblance to finite state automata is fully documented in the case of user interface specification. This characteristic has been further used in systems where graphical programs are generated using interactive finite state machine editors.

Delineation into stages makes some housekeeping intricacies relatively easy to implement. If a task requires variables to be initialised prior to the task running, the necessary code can be inserted into the *start* stage. Also, menu window management problems such as menu highlight synchronisation (ensuring the correct menu box is highlighted according to the current task) can be set up as *start* and *finish* actions.

Examples of task progression

We can give a better idea of how a task will progress from a frozen state to execution of its primitives. As we stated earlier, some tasks are active all the time and are continually scheduled, some tasks run to completion on selection from the menu and some run to completion only when their parameters have been collected.

Following figure 2.3, all tasks reside in the task pool. Some of the tasks in the pool are

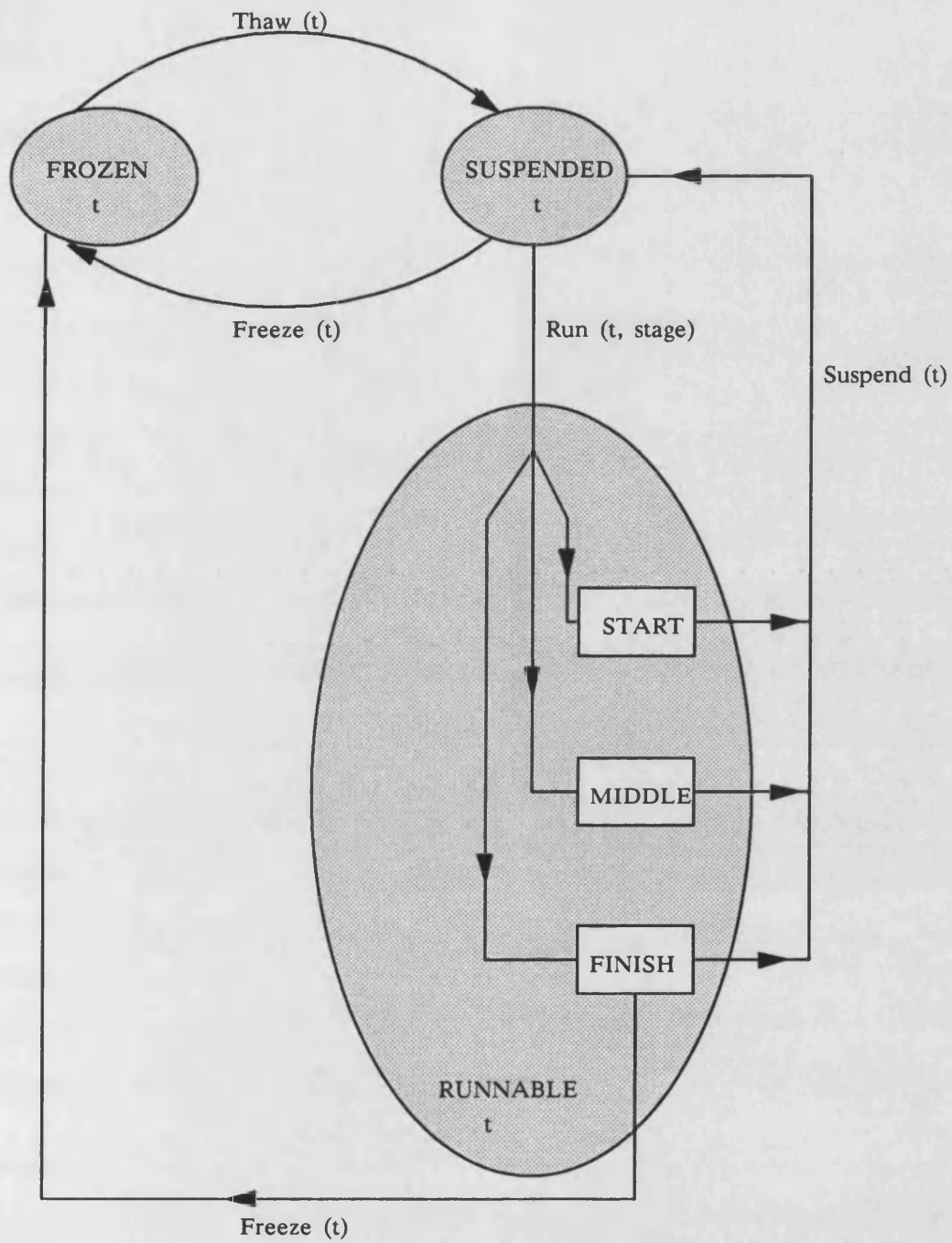


Figure 2.3. Task Progression Showing both Macro and Micro States

frozen, while others are suspended.

Example 1. Cursor Tracking.

A common function in interaction is that of updating the screen cursor position to that of the puck on the tablet. There is no user interaction required to do this and it takes place continually. This task is always needed and so it alternates between suspended and runnable. When the task is first scheduled it will be in the suspended state. The scheduler executes a Try(task,gather) but, as no user-input parameters are required, it immediately executes a Try(task,perform). The "perform" section of the task consists of the code to update the cursor position and the task is then suspended in readiness for its next call.

Example 2. Menu Management.

This is a task that detects button presses in the menu region. It is also always active. The task is first scheduled to gather its parameter, a button press. If there has been no button press it is not scheduled to perform, instead it is returned to the task pool until the button press event occurs. At that point, it has gathered enough parameters, it becomes runnable and it is scheduled to perform the primitive associated with the menu task. This is usually to thaw the task associated with the menu button pressed. On completion it is suspended, awaiting a further button press.

Example 3. A Menu Initiated task.

This task is always frozen until it has been initiated by the menu management task. Once the task has been made runnable it follows the same route as the menu management task with one significant difference. Once the application-specific procedure has been performed the task is returned to the task pool in a frozen state.

2.2.4. Event detection

We have already mentioned that we expect interactive programs to be event driven, so the fourth component of the new interaction model is a mechanism for detecting events. It is commonly the conjunction of a button press (or release, or holding down) with a specific area of screen/tablet which needs to be distinguished. We therefore adopt:

procedure Event(butstate: button; region: area): **boolean**;

as an enquiry function. The function Event is required to test the puck state to see if it corresponds to that named (e.g. button 1 just pressed) and also the region (e.g. command menu). Event has to be sufficiently general to allow for pop-up menus, overlapping menus etc. *Butstate* is defined to cover *new*, *pressed* and *released* where *new* defines a button press, *pressed* represents a button held down and finally *released* indicates a button has been released. It thus defines both level (held down or remaining released) and transitory (just pressed or just released) states, an important distinction. *Region* is used to identify the area on the screen/tablet where the button action occurred.

2.2.5. Well-formed constructs

For the fifth component of our new interaction model we identify a number of well-formed constructs. Structured programming uses a modest number of well-formed constructs, where each construct displays some simple property. These are repeat...until, while, for loop and if... then... else. Such constructs are satisfactory for sequential programs but they are of less value for interactive programs as they do not adequately reflect the asynchronous, fluid behaviour of such applications. This is because interaction is typically used to direct the flow of control along a multitude of

paths.

This section contains details of some of the more suitable control constructs we have identified up to the present time. These constructs are not sufficient in themselves for complex interaction requirements, but when used in conjunction with additional type declarations they are adequate. (Some of the type declarations are described in section 4.4.)

Continual update

There are many applications in which part of the interaction requires a continual update of program variables. Typically such variables are associated with control of the displayed image. For example, a screen cursor may be continually updated under program control to indicate the puck position. This is expressed as:

```
gather_n :   Readpuck(x,y); Run(n);           { Puck monitor }
perform_n:   MoveCursor(x,y); Suspend(n);    { Cursor update }
```

Continual update is also needed in such cases as thermometer scales, rubberbanded lines and boxes and the dragging of windows.

Point and do

A simple form of interaction is the "point and do" action such as selecting "clear screen" by pointing to a labelled menu box and pressing a puck button. There is then an immediate effect which proceeds, out of further control, to completion. Such a task will be permanently active and will run whenever a particular puck button is pressed within a certain area on the screen. Other examples are deleting graphical objects, pulling down a menu, scrolling a window and selecting an icon. There is also a degenerate case in which the position of the puck is not relevant, corresponding to using a puck button for a dedicated action such as menu pop-up. All such permanent

tasks can be coded in the form:

```
gather_n :    if Event(button,viewport) then Run(n);
perform_n:    execute(n);
```

Event recogniser

In principle menu-selection could be implemented as a number of point-and-do tasks, treating each menu entry as a separate area of the screen. In practice this can be cumbersome for all but very short menus. A conventional top-down approach to manage this interaction has been adopted. At the top level, it is sufficient to identify that a relevant event has occurred, namely that a new button press has just happened in the menu. We thus get:

```
gather_n :    If (Event(new,menu)) then Run(n);    { Menu monitor }
perform_n:    Thaw(Comm); Suspend(n);            { Action the command }
```

The monitor component is already familiar. The action component uses procedure *Comm* to decode the puck position and return the task number needed for that command. This task is then enabled by *Thaw*. The event recogniser task takes no further action until the next new press of a button in this menu. It has, however caused a non-permanent task to spring to life (the one associated with the menu item) and this will have its own Gather/Perform entry.

This particular construct is important as it can be used as the basis for a number of interactions. Thus we can have an event recogniser task for detecting events in a number of different windows.

2.3. Formal and informal approaches to user interaction

Our new model has been derived pragmatically with the emphasis on the needs of the practicing programmer. This approach can usefully be compared with formal specification, for example as described by Mallgren in his thesis[41].

In this work he too has identified the need for synchronisation primitives. However his work has concentrated on their algebraic specification rather than their utilisation. For the example of section 2.1, the program can be simplified by defining interaction primitives which combine the test, read and clear operations for each device. The operations wait until the device has data and then return the data and clear the device. This method will result in more readable programs (contrast the program below with that in section 2.1). The waiting performed by the while statement still exists but now at a lower level. This problem will exist while the scheduler is of this style and is based around the sequential data type.

Program operations:

```
getmenu;  
getpen.
```

User actions:

```
menuevent;  
penevent.
```

The new sample program thus becomes:-

```
program example2;

var
  /* declarations */
begin

  Initialise picture area;
  Initialise the interface I.

  while true do
    begin
      cmd <- getmenu;
      pos <- getpen;

      case cmd of
        line: drawline(position);
        box: drawbox(position);
      end;
    end;
  end.
end.
```

A sequential datatype is a collection of unchangeable objects that are accessed from a sequential process via some well defined operations. These objects can be manipulated by creating new objects or by assigning them to variables.

Our own model is again a sequential datatype model, but the nature of the data structures used, i.e. the booleans representing the states of the task, is such that each task is uniquely identifiable. Therefore the sharing of data and state is not a problem. The only data that is shared is that data input by the user and that particular problem is described in detail later in this section.

At this point it is useful to discuss those aspects of our model which are related to the issues raised in Mallgren's thesis. As we mentioned earlier, unstructured looping is still unresolved in the 'beautified' program given above. The actions *getmenu* and *getpen* are essentially event handlers which loop until an event occurs. In our model those particular actions and similar ones are given a permanently active state and the single structured loop, the **while** statement, ensures that these actions get scheduled.

Mallgren also assumes, once a command from the menu has been selected, that it then enters its own controlling structures. Again this is undesirable because of its

unstructured nature. In the Interaction Model we propose, all control of the tasks remain external.

Mallgren argues the concept of the *shared* datatype with internal states and restrictions which can be imposed on those states is more suitable for the interfaces we are concerned with. The interface is itself treated as a shared object. The problem of ensuring data objects are available to one or more operations is overcome by using a special event algebra to identify the states in which shared data variables can exist.

The event algebra is sufficient for describing the transfer of data through an interactive interface, the synchronisation between the user and the application, but it does not adequately model the user actions by which data is entered. User behaviour is modelled by defining the set of user actions, where each action corresponds to a specific user activity. A sequence of user actions is represented by a special program called **user description**. For our earlier example, a user description is given below.

```
user example2;  
  
begin  
  while true do  
    begin  
      menuevent;  
      penevent;  
    end;  
  end.
```

As well as primitive synchronisation operations, there also needs to be a set of low-level operations for controlling user input. Mallgren identifies the following low-level operations:-

Wait (device1,device2,device3...device_n);

Test (device);

Read (device).

The Wait operation waits until any of the devices in its parameter list has data and then returns the device name. The Test operation returns true if a particular device has data available. The read operation returns the data available at a particular device.

The formal specification of low-level input primitives raises some questions, these problems are identical to the ones that are also addressed by the informal Interaction Model we propose. These problems are given below.

1. The Testdev operation makes it clear when data is available but is not clear when data should be made unavailable. When should it return False?.

In the Interaction Model, when the **Event(button,viewport)** primitive is used it contains button state information which is subsequently available to all tasks lower down the scheduling loop. For example, some active task may have as part of its **gather** section:-

```

:
:
If event(new>window) then
  run(t,start);
:
:

```

Assume the function returns True. The task becomes runnable and executes the appropriate portion of its **perform** section. However, tasks lower down the scheduling loop, which are also active may require collection of the same parameter, so should the same data/parameter be bound to the second task? Clearly, in some cases the nullifying of data is desirable, but there are instances where we would wish data to be available for all active tasks. The current (x,y) coordinates of the puck position is one such case.

2. The Waitdev operation has the ambiguity of (1) and an additional one. If there are more than one device ready which one should report back to the application?

3. The Readdev has its own problem. If additional data has arrived at a device, should the data be ignored or queued?

Mallgren makes the following assumptions and it is pertinent that these assumptions are not dissimilar to the ones made by the author during the design of the Interaction Model when the author was unaware of this work.

Assumptions

1. Testdev returns False only after data has been read.
2. Conflicts in Waitdev are resolved according to a pre-determined priority amongst devices.
3. Input is not queued; newly arriving data is discarded.

The scheduler design in the Interaction Model largely resolves the related problems associated with test and wait. The scheduler is essentially composed of a cyclical scan through a pool of active tasks. The nature of the data structures ensures there is a natural ordering of these tasks. Thus parameters/data become available to active tasks in a specific order. It should be mentioned that in our case the Waitdev problem is much simpler because there is only one physical input device which we are continually sampling.

We can conclude that the Interaction Model described in this chapter is very similar to that independently derived by Mallgren. While the convergence is not a proof that our approach is correct, the fact that Mallgren and the author developed similar models from very different starting points, using dissimilar techniques is striking. Mallgren has formally specified an essentially complex problem which suggests that our model is a viable one for programming the user interface for graphical problems.

It is also apparent that although Mallgren has concerned himself with the design of a suitable scheduling mechanism and the appropriate synchronisation constructs arising therefrom, he has not defined any ideal programming language constructs. Thus our model improves on Mallgren's model and in addition identifies some important programming language primitives.

2.4. Summary

Starting from simple forms of interaction we have developed a new model based on the pragmatics of programming user interfaces. This model consists of five major components:

1. a scheduler;
2. a task model;
3. mechanisms for task progression;
4. an event detection primitive;
5. three well-formed constructs.

We have compared our own model with that formally developed by Mallgren and have found several satisfactory parallels which reinforce our confidence.

In the following chapter the new model is used as the basis for an implementation of a tool to assist the interface designer.

3. Implementing the New Model

For the new model to be of use to the interface designer it needs to be embedded in a programming language. Such a language has been implemented by the author and given the name SIDL (Simple Interaction Design Language).

In order to create the language in a reasonably efficient way it was decided to build it on C. This was done by writing a pre-processor to convert SIDL programs into C. The pre-processor is called GRIP (Graphical Interaction Pre-processor).

This chapter first describes the reasons for developing the new specification language and then continues with a section on how the language is implemented. A detailed description of the syntax and some of the semantics of the language is provided.

3.1. The need for a design language

Natural language mirrors the thinking habits of people using the language. People are constrained to think and express themselves in that language. If there is difficulty in this, the language must be further developed by the invention of new idioms and the like. This characteristic also applies to programming languages. A programming language will typically influence the user in at least the following manner.

1. The language will determine how a particular problem will be solved and also the range of problems that can be solved;
2. The basic programming constructs will affect the style of programming;
3. Portability and efficiency issues will vary according to the language used.

Traditional high level language design reflects these points and also the change in computing technology. Thus we have had the development of static imperative programming languages such as FORTRAN and COBOL. These languages reflect their usage, FORTRAN in its suitability for numerical computation and COBOL with its orientation to the data processing requirements of the business community.

Systems implementation languages evolved from assembly languages and their most significant characteristic is that they allow the programmer direct access to machine operations and addresses. Examples of these languages include BCPL and C.

Block structured languages are derived from static languages. They typically contain a selection of control constructs and possess the ability to classify program objects to be of a particular type. There is a limited amount of dynamic storage allocation called the block structure. Languages of this class include Algol, Pascal and Ada.

Dynamic high level languages such as Lisp and Prolog have all storage management performed dynamically and tend to be tailored for specific applications. Thus Lisp is used for list processing applications.

3.1.1. Fourth generation languages

The 1970s and early 1980s saw the advent of a new generation of languages, the so-called application generators and fourth generation languages (4GL). Their development has arisen from the need to overcome the problems associated with traditional software development.

The advantages of 4GL can be simply stated as :-

1. developers need not be professional programmers;
2. the tools are easy to use;
3. there is a high level of productivity.

Application generators are normally interactive and either generate third generation high level language code or tables which drive the application directly. The applications are very specific and there is virtually no flexibility.

4GL are mainly non-procedural, thus they do not require complex programming skills. Applications created by 4GL take longer to develop than those by application generators but the target problem environment is unrestricted.

So what are our criteria for a design language and where will our language reside in the brief classification described above?

Our primary reason for designing a suitable language is to utilize the Interaction Model defined in the preceding chapter. It is clearly inadequate to expect software designers to follow some methodology and then fail to provide the tool environment to support the methodology. Constraining the designer to use a language such as SIDL will ensure that the subsequent design is consistent (in this case, with our methodology).

Our criteria for our language will now be briefly described and details amplified in later sections as the need arises.

1. The language should reflect the Interaction Model such that the functions and the objective of the program should be obvious from a static study of that program. Thus the language may serve as an informal specification of the application being developed allowing the design of an application to be discussed in detail.
2. The language should incorporate non-procedural elements so that the designer can describe what is required without having to list the detailed steps on how it should be achieved.
3. Interactive graphics programs are complex so some procedural elements will have to be included so that flexibility is not lost. Thus at least some standard programming constructs will have to be provided.
4. The language is primarily intended for use by programmers, thus it would be helpful and it would ease the transition to use of this language if the language bore resemblance to existing programming languages.
5. Interface design is an area which has considerable scope for discussion and variation. Research is still trying to identify the factors that constitute an ideal interface. In an attempt to define such an interface considerable effort has been put into the post evaluation of interfaces and a number of tools are currently being developed. SIDL, as well as being an informal specification of an application should also form the basis of tools for late evaluation of interfaces.
6. The language should be expressed by a context free grammar so that construction of the parser is simplified.

3.2. System design

Modern programming's key concept for controlling complexity is abstraction - the notion of emphasising particular detail. To this end, the design of SIDL is no different. We have identified the requirements of interactive graphical applications as exemplified

by the Interaction Model and we allow the SIDL language to express only those abstractions which are concerned with the interaction element.

In this section we are concerned with the philosophy behind the design and the implementation of the language and its associated interpreter GRIP.

SIDL and GRIP are in essence a front end to the UNIX C compiler. A program written in SIDL is used to generate a C program which contains the complete interaction needs of an interactive graphical application. Thus we can immediately note that SIDL falls into the category of 4GL in the simple classification outlined in section 3.1. Further, because the language is targeted onto a very specific application area we can associate it more closely with application generators.

3.2.1. Implementation approach

In this section we will first discuss the choice of development language, then we will then go on to consider the first implementation and conclude by describing the final implementation.

The development system was an HLH Orion 32 bit superminicomputer running UNIX 4.2 BSD. Suitable languages were essentially C and Pascal and, of these, C was preferred. This was because, firstly, C forms a more coherent aspect of UNIX than does Pascal. Secondly there are a large number of software tools available. The availability of the following software tools affected the decision.

yacc and Lex : compiler construction tools.

adb, dbx: interactive debugging tools.

make and sccs: configuration management tools.

cb: general purpose tools.

These two factors were of considerable importance when choosing the development language.

Early attempts at implementation are now described for historical reasons. An experimental syntax for SIDL was defined. The syntax was formally expressed using BNF. The grammar rules were then used to construct a recursive descent syntax analyser. As we were attempting to generate approximate C code quickly, the simplest means of generating the code appropriate to the SIDL specification was by including the code generation within the analyser.

Very briefly, syntax trees of the SIDL program were constructed, these data structures were then used to generate C code. The SIDL syntax was however going through a rapid metamorphosis: each change meant a change in the analyser which made alteration to the syntax both difficult and unwelcome. Clearly a new approach was required.

On the Unix System, several tools are available which allow the non-specialist to define and process rich input languages. These tools were originally intended for the development of classical compilers, but have proved useful in a wide variety of applications as well.

One such tool is yacc[34] "Yet another compiler compiler". Yacc generates parsers from an input specification language that describes the desired syntax. Each rule in a yacc input is associated with a fragment of C code. As a rule is recognised the appropriate code is invoked. The parsers generated by yacc consist of a finite state machine and a stack. Yacc is based on the theory of LALR(1) parsers[2], and it permits controlled use of ambiguous grammars, with disambiguating rules making it much easier to handle traditionally difficult problems such as operator precedence and the dangling else.

A similar tool Lex is used to generate lexical analysers[38]. Figure 3.1 indicates the dependencies between various Unix tools and the interpreter derived from them.

As with the original attempt, it was decided to continue with the basic policy of incorporating the code generation within the analyser. Within yacc this meant that for

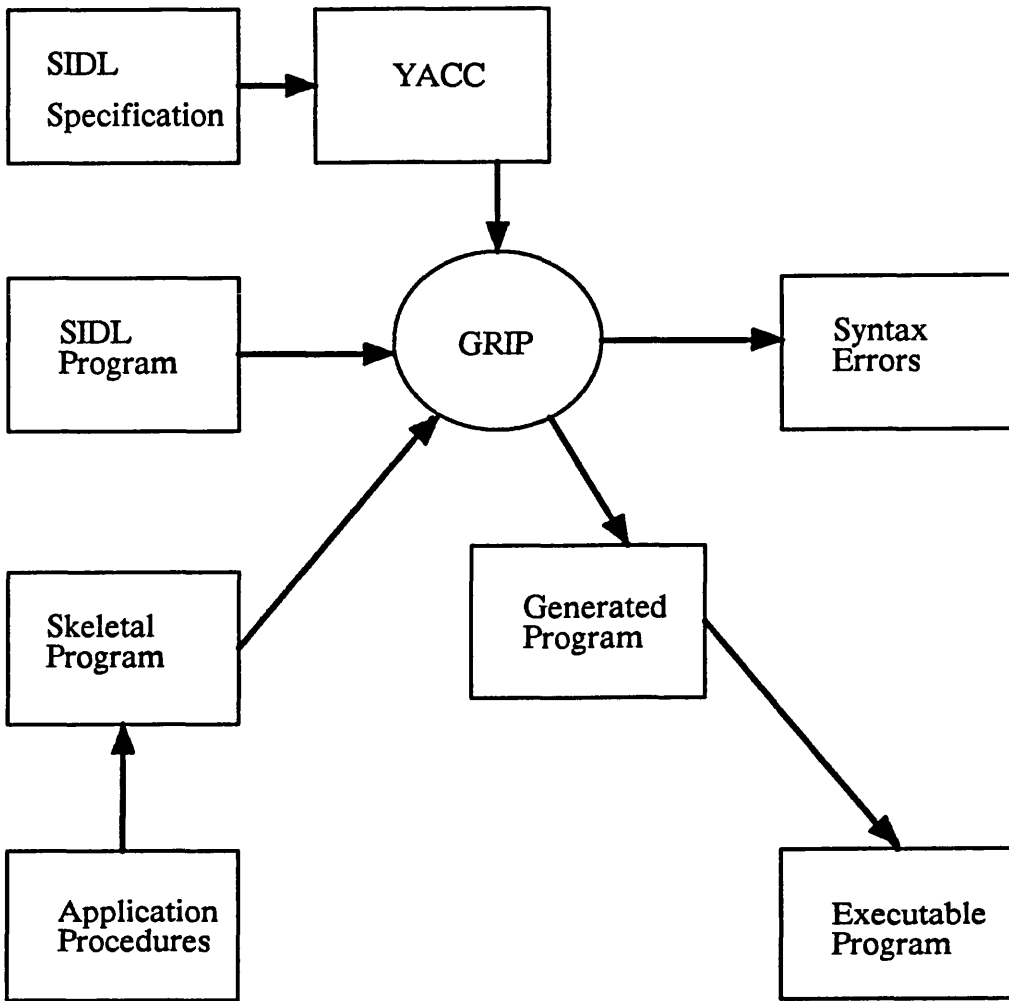


Figure 3.1. GRIP Design

each syntactic structure successfully parsed the action part associated with that rule invoked C code, which in turn generated the interaction C code.

3.3. An example application

To illustrate the syntactic structure of a SIDL program, a substantial example will now be described. The full SIDL code for this application appears in Appendix B. We will refer back to this example later in this chapter and also in Chapter 4. **Vecfnt[6]** is an interactive graphical bitmap font editor which has been programmed in SIDL. The application allows a user to create and edit bitmaps representing characters in a font. The characters can be saved and retrieved from font files.

Figure 3.2 describes the layout of the screen. There are four screen regions: a character edit region, a menu region, a font grid region, and the remainder of the screen can also be considered as a separate region. The character edit region is divided into a grid, each grid box representing a pixel. The font grid region is used for storing the current font on the screen. The menu region is composed of a series of buttons which the user selects to perform various editing functions. To aid the font designer in laying out the character baseline, height etc a number of graphical objects called "handles" are provided. These are movable rules superimposed on the character edit region, shown as thick lines in Figure 3.2.

The standard input devices used are a puck with four buttons (however, they behave as one logical button) and a keyboard for entering text strings.

The following functions are available.

There is a general purpose task which is independent of the menu, used for editing the character area with the current colour.

All other tasks are initiated from the menu. They are:-

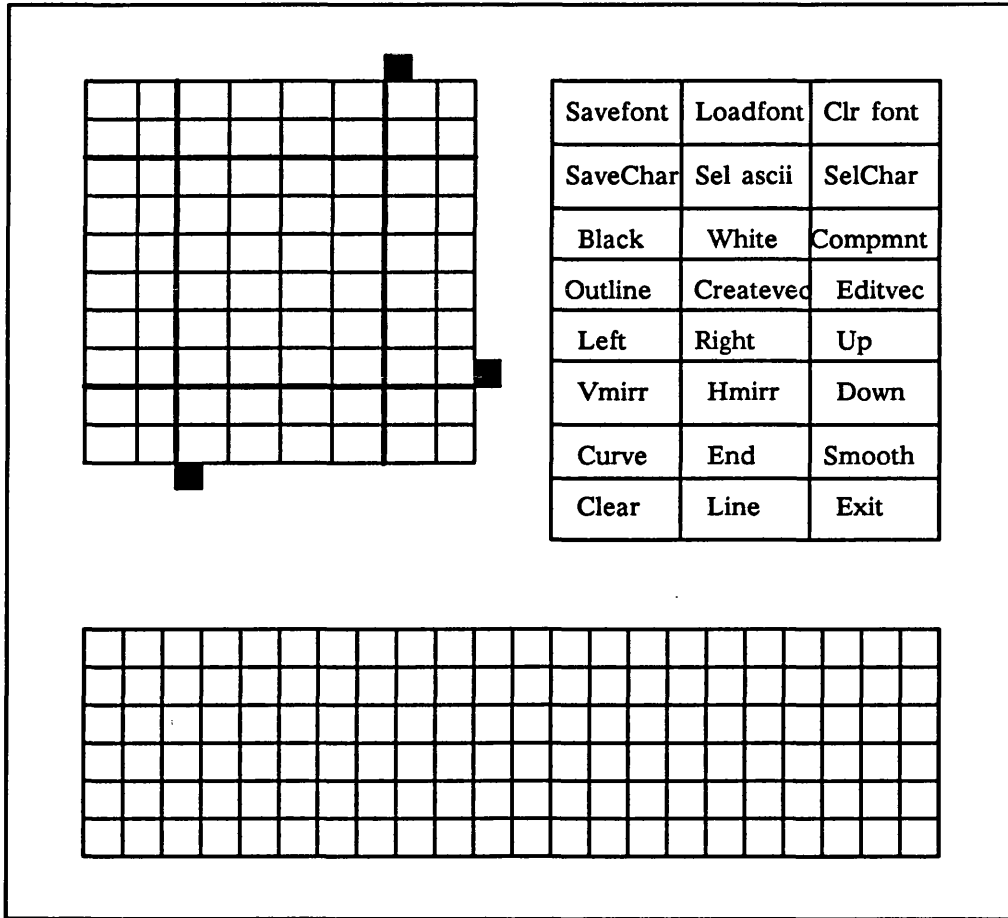


Figure 3.2. Vecfnt - Bitmap Font Editor

Black, White and Complement.

These tasks set the ink with which the character is edited. Thus each time an edit grid cell is selected using the puck, the cell is filled with the current ink. In the case of Complement, if there is a black cell, it is replaced by a white cell and vice versa.

SelChar and SaveChar.

These tasks are used to select characters from the font grid area for editing purposes or to save the current character in an appropriate font grid cell.

Left, Right, Up and Down.

These functions shift the current character in the edit area in the direction selected.

LIne.

This function draws a line in the edit area with the current ink, between two user selected points.

SaveFont and SelFont.

These functions either save the currently displayed font in a file or load a new font into the font grid area.

As an example of its use, consider the following sequence of actions. The user selects SelFont from the menu and enters a file name when prompted. The font is displayed in the font grid area. The user next selects SelChar from the menu and then selects a character from the Fontgrid area by a point and click. The character is displayed in the edit area, where it is subsequently edited. The character can finally be saved using the SaveChar function.

The purpose of this application was not to implement a fully blown font editor, although that was an offshoot from the work, but was to create a test vehicle for the Interaction Model. In the following sections we use this example to illustrate the SIDL syntax. In addition we provide examples of some of the C code that is generated.

3.4. The SIDL syntax

The syntactic structure of SIDL is similar to that of most programming languages in that it can be described by context free grammars. A grammar is context free if the left hand side of every production consists of a single non-terminal and the right hand side consists of any non-empty sequence of terminals and non-terminals. For example.

$$A \rightarrow x B y ;$$
$$B \rightarrow z ;$$

The context free elements of SIDL are described using an extended BNF notation. A full BNF definition is given in Appendix A. However to describe a language adequately, the semantics and those aspects which are context dependent (such as type checking) also need to be described. We discuss these secondary aspects informally in the following sections.

3.5. SIDL program structure

There are three sections to a basic SIDL program: a type section; window definitions; task definitions. There can be none or any number of window or task definitions.

The syntactic form of SIDL closely follows that of Pascal. Thus, commas and semicolons appear in similar positions as in Pascal. There were two reasons for adopting this approach. Firstly, a more precise BNF definition could be obtained by making use of the Pascal Language definition. Secondly, similarity to a high level language was bound to be useful in the early experimentation with this language. All SIDL keywords are in bold font.

```
INTERACT vecfnt;  
TYPE           { Type and Variable declarations }  
ENDTYPE  
WINDOW  
    .  
END  
TASK selchar;  
TYPE  
ENDTYPE  
  
DO  
END
```

3.5.1. Types

This section is loosely based on the Pascal Type and Var sections. Its purpose is to declare and specify some of the global variables which will then be used to produce tables for code generation. It is also used to specify the environmental/presentational aspects of the application.

SIDL declarations are used to specify the number and names of windows (screen regions) which will be required. An additional SIDL declaration is used to identify the window to be used as the menu. The total number of tasks that will be operating in the interaction is also specified.

```
NUMTASK 13;  
WINDOWS = (menu,grid,fontgrid,hand1,hand2,hand3);  
MENU menu;
```

Colour maps can be defined in a variety of ways. Colours can be declared as specific entries in the look up table with the required red, green and blue values, or they can be specified as a range of colours occupying a number of entries in the look up table. The start and end colours can either be default system supplied colours such as the

primary colours, or they can be user supplied r,g,b values, or a mixture of the two. A few examples are shown below.

In example 1, the *offcol* is at entry 13 in the colour table and its red, green and blue values are given by 0, 120 and 0 respectively. Example 2 uses a SIDL keyword definition of RED to enter default system r,g,b values at entry 20. Examples 3 and 4 are used to specify a range of colours at certain positions in the look up table. Thus in example 3, a range from colour (0,100,0) to white is specified. The range begins at position 100 and is interpolated over 40 look up table entries. Example 4 specifies a sunset between RED and YELLOW over 100 entries in the look up table.

```
COLOUR offcol(13,0,120,0);           /* 1 */
COLOUR red_offcol (20, RED);        /* 2 */
COLOUR red_range (100, 0,100,0) 40, WHITE; /* 3 */
COLOUR sunset (50, RED) 100, YELLOW; /* 4 */
```

The most important type declaration is that used to specify the mutual exclusion tables. The function of the tables are described in detail in Chapter 4 but here we are only concerned with the syntactic structure of the specification.

Each group of tasks is specified as a row together with its task type. The levels for tasks describing Vecfnt are shown below.

```
MUTEXL= (backgrnd,foregrnd,compment) OF TYPE 3;
MUTEXL= (selchar,savechar) OF TYPE 1;
MUTEXL= (line) OF TYPE 1;
MUTEXL= (left,right,up,down) OF TYPE 2;
MUTEXL= (clear,clearfnt,vmirror,hmirror) OF TYPE 2;
MUTEXL= (exit) OF TYPE 2;
```

The tasks are enclosed in parentheses and the types are indicated.

In addition to a general type section at the head of a SIDL program, there are also type sections in window block and task block declarations. In the case of windows they are used to describe the type of window being defined. Windows can be of the following types.

STATIC windows are those which remain fixed in their position, **MOTILE** windows are those which can be picked and dragged to a new location on the screen. **VARIABLE** windows are those which can shrink or increase in size. There is an additional window of type **BITMAP** which has bitmapped properties, thus scrolling can be implemented.

All window types are declared in the following manner. If there is no window type declaration, it defaults to **STATIC**.

handle_1: OF MOTILE;

This is a type definition for one of the handles in Vecfnt. The handles can be selected and moved to a new position and thus it is of type **MOTILE**. Logically, the menu and font grid area are fixed in their locations and are declared accordingly.

Type declarations for Tasks are currently limited to one type, at least in terms of semantics. In section 4.2 we discuss aspects of parameter collection in detail, here we are concerned with how to specify the parameters that a task has to collect before it will run to completion.

In Chapter 2 we defined a parameter to be a combination of a button press and a screen region, where button types are *new*, *pressed* and *released*. In complex task specifications, a number of parameters are typically collected. In the example shown below, the task Line requires two button presses in a window region to mark the two ends of a line.

gridop: [NEW,grid,2] OF PARAMS;

The variable *gridop* is used to store the value of the current parameter being counted. The information enclosed in the square brackets tells us that two **NEW** button presses are required in the *grid* area.

3.5.2. Windows

When designing an interactive application using SIDL, the screen region is considered as a composition of distinct regions where events can occur. These regions have to be given certain attributes which will decide their presentation and their functionality. The type definitions described in the earlier section give the reader an idea of some of the functionality that is currently implemented.

SIDL differs from a UIMS in the emphasis placed on appearance. In that respect it does not provide the expressive power that a typical UIMS would give. SIDL is more concerned with specifying the behaviour of interactive applications and also of easing the task of programming such systems. Thus windows are defined in a very simple manner. A window definition is simply the bottom left (x,y) coordinates and the width and height of the window. Initially, the user coordinate system is identical to that of the screen coordinate system. If the rectangular region (for we are only concerned with rectangular windows) is going to be composed of sub-regions (also rectangular) then an automatic grid facility is available. The window definition for the font grid is given below.

```
WINDOW fontgrid;  
DO  
  HT: 331;  
  WIDTH: 539;  
  XBOT: 661;  
  YBOT: 110;  
  GRID: NUMH = 8;  
  GRID: NUMV =16  
END
```

Note that the window is defined within a DO END block in a manner that is similar to Pascal.

3.5.3. Tasks

As we have stated earlier, tasks are the basic building blocks of an application. They are specified so that they implement both the user interface and the application interface (See section 2.2.1). The SIDL language constructs closely model the Interaction Model. Consider the following task specification examples taken from Vecfnt.

In the first example there is a type declaration indicating that this task is to be initially active, it is also not activated from the menu. Both examples display the two sided nature of a task definition. The Gather section shows how the user's behaviour is communicated to the application and what action should take place. For the *savechar* example, we can read the task as follows. If there is a button press in the menu button associated with *savechar* then carry out the first stage, in this case it would be task synchronisation activities (section 4.1). When the user finally presses button 2 while

```

TASK edit;
TYPE
  INIT= ACTIVE;
ENDTYPE
DO
  GATHER;
  IF NEW(2) BUTTON IN grid THEN
    run(t,start)
  ELSE
    IF PRESSED(2) BUTTON IN grid THEN
      run(t,middle);
    PERFORM;
    DO
      START:
      DO
        fillcell(xcoord,ycoord);
        suspend(t)
      END;
      MIDDLE,FINISH:
      DO
        fillcell(xcoord,ycoord);
        suspend(t)
      END
    END;
  END
END

```

```

TASK savechar;
DO
  GATHER;
    IF NEW(2) BUTTON IN menu THEN
      run(t,start)
    ELSE
      IF NEW(2) BUTTON IN fontgrid THEN
        run(t,middle);
  PERFORM;
  START:
    DO
      bsync(t);
      suspend(t)
    END;
  MIDDLE,FINISH:
    DO
      findentry(fontcomm);
      savechar();
      esync(t)
    END;
END

```

over the font grid area, then perform the application primitive which saves the character in that position. Note that the **PERFORM** section is the application interface. When carrying out prototype activities, procedure stubs will usually be placed here. Also note that once the task has completed it freezes until it is initiated again, whereas the edit task simply suspends and is always waiting to collect its parameters.

3.6. Code generation

Code generation was accomplished by use of tables and pre-written files. As a SIDL program is parsed, the type definitions are used to generate constants. All the window definition information is collected into a table, whose structural definition closely matches the structure definition in the final C program. The task definitions are likewise collected into a table structure. Specific data declarations such as the mutual exclusion groups and parameter information are also collected into tables.

Part of the overall system includes a skeletal C program. This contains the basic primitives required by the interaction model. These are the event detection procedures, procedures for identifying the menu box selected, all the primitives used to maintain task progression and finally skeletal table structures.

As well as producing tables during the parse, a merge of the skeletal program and the code tables is performed. The merge is done in the following manner. Throughout the skeletal program a number of markers are embedded in the text. The basic operation of the merge is to copy the skeletal file to a new file until a marker is met. Each marker is uniquely identifiable and, according to the marker reached, the appropriate code generation is carried out and subsequently inserted into the file. Thus for example, when generating all the task *#defines* the task table is scanned and the code generated at the position of the marker. Once the code has been generated, the program continues to copy the skeletal file until it reaches another marker.

The first copy occurs when the SIDL program name is parsed. Other copies of the skeletal file occur once all the window definitions have been processed.

During a parse of a Task definition, all the syntax encountered is translated into C. The nature of the SIDL language ensures that the code generation is straight forward. Only when certain control constructs peculiar to SIDL are met is there recourse to the use of tables. Once all the task definitions have had their appropriate code generated, that code is copied to the output C program and the merge continues.

The end result of the parse is a SIDL program listing which will contain any syntax errors reported. If there are no errors then a C program containing the prototype interaction is produced. Application specific procedures can then be added and the resultant program is ready for compilation.

3.7. Summary

In this chapter, we have presented the reader with the reasons for the design of a language for graphical interaction programming. The language and its implementation have been presented. Examples showing its syntactic structure have been described.

In the following chapter we look at some of the issues which arise from interaction management. We show how these are resolved by using the new interaction model and its implementation.

4. Using the New Interaction Model

In this chapter, we discuss some of the problems that are typically found with interactive graphical programs. These problems are discussed from the baseline of our new interaction model proposed in chapter 2. The solutions for these problems are presented. In addition, we look at some problems which arise because of our model. These can be regarded as the inherent disadvantages of using our model. The discussion is illustrated with a mixture of example code generation and pseudo-code.

4.1. Task synchronisation

All good interfaces must allow unrestricted asynchronous activity from the user. This freedom, however, makes the menu management a less than trivial problem. It is easy

to envisage the case where the user makes ad hoc selections from the menu buttons thereby initiating various functions and so losing any idea of what is happening. Where initiated tasks require additional parameters (such as selections from another menu), the problem is even more acute. The user must be able to select another task while the current task is waiting for a parameter, and still be able to return to the earlier task with the current state of that task intact. Furthermore this facility must be provided in a manner that is user friendly.

These objectives have to a large extent been achieved in the new model by enforcing a sub-structure over the tasks initiated from the menu.

An examination of menu-driven applications at Bath indicates that tasks initiated from menus can be divided into three categories:

tasks which run to completion when selected;

tasks which require one or more parameters before running to completion;

tasks which, when selected, set parameters to be used by other tasks.

Furthermore, tasks can be assigned to Mutual Exclusion Groups, such that at most one task from a group can be active at any time. In addition, the mutual exclusivity for a group extends to any preserved tasks. We can infer three rules to control these groups.

Rule 1:

Each group can have an active task, so suppose task A is in a different group from active task B. If the user now selects task A, then task B has to be preserved. It can therefore be re-instated later.

Rule 2:

Suppose tasks B and C are in the same group. If B is active and C becomes active, then B has to be killed because the user has indicated a preference for C.

Rule 3:

Suppose B has earlier been preserved and that C is in the same group. If C now becomes active, then the preserved state of B has to be discarded. This is because this rule has essentially degenerated into Rule 2, thus the user has again indicated a preference for C.

4.1.1. Task management algorithm

The task management rules are implemented by a set of tables, a stack and two primitives which are accessible to the SIDL designer. Each task definition includes a call to these primitives. The *Bsync(t)* primitive is called as the first action in the *start* of the *Perform* section and the *Esync(t)* primitive is called as the first action of the *finish* stage.

The *Bsync* function identifies the type of the task and the group within which the task has been specified. Tasks can be of the types described earlier in section 4.1. In our implementation the groups are simply a set of arrays. The function then performs the following task management (related to the menu) according to the task type.

If the task is one where a parameter has been set, then the function simply kills the task which is active in this group (there is implied mutual exclusion) and carries out menu management activities i.e. switching on and off the appropriate menu buttons.

If the task is a run to completion type then the function searches all the groups. If there is any active task its state is preserved in the stack and the menu management activities are executed. Usually, only tasks which require additional parameters will be affected.

Finally, when the task is one which requires additional parameters then the process is more complex. Firstly, the tables used to store the status of a task are searched for all active tasks. If an active task is a member of the same group as the current task then that task is killed. If the active task is a member of another group then the status of

that task is preserved on the stack. Secondly, the stack is searched for any tasks which also belong to the group of the current task. Such tasks are also killed. This is to maintain the mutual exclusivity of the group. The standard menu-management activities are again performed.

The *Esync(t)* primitive is much simpler in operation. Similarly it determines the group and task type of the current task, then according to the task type it performs some basic operations.

For tasks which set parameters to be used by other tasks, it simply freezes the task. The menu button light is not switched off, because it is used to indicate the current parameter that will be used by other tasks. Run to completion tasks and tasks which require additional parameters are processed in the same manner. Firstly, the menu button light corresponding to this task is switched off and the task frozen. Secondly, the top of the stack is popped and the task now available is made active, its menu button light is put on and to all intents and purposes this is the current task.

4.1.2. Grouping tasks

The exact manner in which the tasks are collected into groups is determined by the application designer. Typically the procedure is as follows. By and large, apart from some low-level details, the functionality of an application is determined by the number and type of menu options available. Thus menu options which perform similar tasks are put in the same group, menu options which require the same number of parameters or the same type of parameters may also be classified in one group.

If we look at an example the classification method will be clarified. Consider the Vecfnt Graphics editor developed at Bath (section 3.3). The menu options are:- White, Black, Complement, Selfont, Savefont, Left, Right, Up, Down, Horline, Verline, SelChar, SaveChar. White, Black and Complement are examples of tasks which, when selected, set a shared parameter to be used by other tasks. In this case they

set the ink or mode of action in which the editing task operates. The editing task simply fills in a box (pointed to by the user) in the edit grid area with the current ink. Because of their similar function they can be put in the same mutual exclusion group.

Tasks Left, Right, Up and Down shift the character currently being edited in one of four directions. They are examples of run to completion tasks. They can also be put in one group.

Selchar and Savechar are tasks which allow the user to select a character from the font area for editing or to save the current character in the font. They require a parameter (button click within the font grid) before running to completion. They are put in one mutual exclusion group.

We can now demonstrate how this method allows relatively unrestricted use of the menu.

Suppose the user has created a character in the edit grid area, and they are now ready to save the character in the font grid area prior to storing on disk. The user selects the task SaveChar. This task requires a button press over the font grid area before it runs to completion. The user has not yet supplied the button press and during the next few seconds decides to shift the newly created character towards the left. They do this by repeated selection of the Left task from the menu until a satisfactory position is reached. The user then supplies a button press in the font grid area and the character is saved in that grid accordingly.

Hidden to the user, task SaveChar was suspended and, because task Left was of a different type and therefore in a different group, Left executed and control was returned to SaveChar.

4.2. Parameter collection

The collection of parameters (detection and decoding of button activity) is a problem encountered in most graphical interaction problems. In this section we look at some of the problems and their solutions (if any).

4.2.1. Parameter counting

Typically, task definitions require the user to specify the events necessary for a task to run to completion. Where these events are unique there is no obvious difficulty. However serious problems arise when two or more identical events need to be detected. Events are considered identical if both the button type and the region are the same.

In the painting system developed at Bath[58,57] there is a command Vwipe to paint a rectangular region on the screen with a graded range of colours. The user specifies the rectangle by clicking two diagonally opposite vertices in the drawing region. The two extreme colours are specified by clicking two entries on the palette region. The operation completes by interpolating colours as it draws horizontal lines to fill the rectangle.

To avoid constraining the user, collection of the parameters can be done in any order. The action to be taken when a particular parameter is collected therefore depends on its sequence number within a particular region. For example, the first event in the drawing region marks the first corner of the rectangle. This results in a rectangle being rubberbanded on the screen from the marked position. The receipt of a second event in the same region results in the second corner of the rectangle being marked. The user selects the range of colours by two events in the palette region. Provided that a count is kept for each region the colour and the rectangle parameter collection can be interleaved. Thus the user can point to one corner, then to its colour, then to the second corner and finally to its colour. Alternatively the user could indicate both corners of the rectangle and then both colours, or even both colours and then both

corners.

A critical observation is that each event, although not unique, may have actions associated with it which depend upon the count. We can decide when to call worker task for this command by keeping a total count of the number of parameters collected.

Example

For the task Vwipe (from UltraPaint), the interaction requirements are two window regions Palette and Drawpad; two button presses to mark the region and two button presses to select the colour range.

The actions to be associated with the parameters as they are collected are as follows:

Menu:

1: Make the task active;

Palette:

1: select first colour;

2: select second colour;

Drawpad:

1: fix first corner

2: rubberbox rectangle from the first corner to current cursor position.

3: fix second corner and draw rectangle.

Thus for these requirements, the first new button press in the Drawpad area fixes the first corner of the wipe region. The next stage (2) is actually a dummy button press used for breaking up the interaction into convenient sections. The second button press (stage 3) is the second corner of the rectangle and results in the wipe region being marked out. The interaction for the Palette region operates in a similar fashion.

We can now describe the implementation of this example interaction. Firstly, the relevant window region counters are initialised. The Gather section attempts to collect the various events as the user supplies them. The task is initiated by a button press in the menu region. The *start* section carries out any necessary menu synchronisation

and the task suspends. When any of the other required events occur, the appropriate window counter is incremented and the *middle* section is executed. Also during the Gather, the current window variable is set. The *middle* section then performs the application procedures specific to the current event being processed. Thus if the first event in the palette region is being processed, the current window is palette and the first event is that associated with selecting the first colour. Again the task suspends. The Gather section also checks if all the parameters have been collected. If this is so, the *finish* stage of the perform section is executed. This contains the application procedure to perform the wipe.

initialise palette_counter to 0

initialise pad_counter to 0

Task VWIPE:

Gather:

```

begin
  If Event (new, menu) then Run (t, start);
else
  If Event (new, palette) then
    begin
      increment palette_counter;
      set current_window to palette;
      Run (t, middle);
    end
  else
    If Event (new, drawpad) then
      begin
        increment pad_counter;
        set current_window to drawpad;
        Run (t, middle);
      end
    else
      begin
        Run (t, middle);
        if complete set then Run (t, finish);
      end;
    end;
end;
Perform: case stage of
start: begin
      Bsync (t);
      Suspend (t);
    end;
middle: begin
      case current_window of
palette:case palette_counter of
          1: select first colour;
          2: select second colour;
        end;
drawpad:case pad_counter of
          1: fix first vertex,
          2: rubberband.
          3: draw rectangle;
        end;
      end
      Suspend (t);
    end;
finish: begin
      wipe_region (vwipe);
      Esync (t);
      Freeze (t);
    end;
  end;
end;

```

4.2.2. Parameter anullment

In section 4.1 we looked at how tasks initiated from menus are synchronised using similar tasks grouped together into tables and a set of rules which operate on these tables. This method presumes that, if two tasks are in different mutual exclusion groups and they require the same parameter, only one collects the parameter because the other task is inactive. However, this does not actually happen, consider the case below.

There are two tasks T1 and T2 in groups 1 and 2 respectively. Their order in the scheduling loop is:-

```
procedure try (t, op: Integer);
begin
  case t of
    T1: action;
    T2: action;
    :
    :
  end;
end;
```

- 1) Task T2 is made active, the system then cyclically executes the T2 gather section and attempts to collect its parameters.
- 2) T1 is selected (made active); this results in T2 being killed and its current state being stored for future reactivation according to Rule 1 as stated in section 4.1.
- 3) T1 is now active and periodically attempts to collect its parameters.
- 4) The user provides the parameter which T1 collects and it runs to completion. The primitives, operating on the tables, then re-initialise T2. These primitives are called after the T1 has run to completion, but since T2 is lower down the scheduling loop and the parameter just collected for T1 has not been cancelled , it is reused for T2.

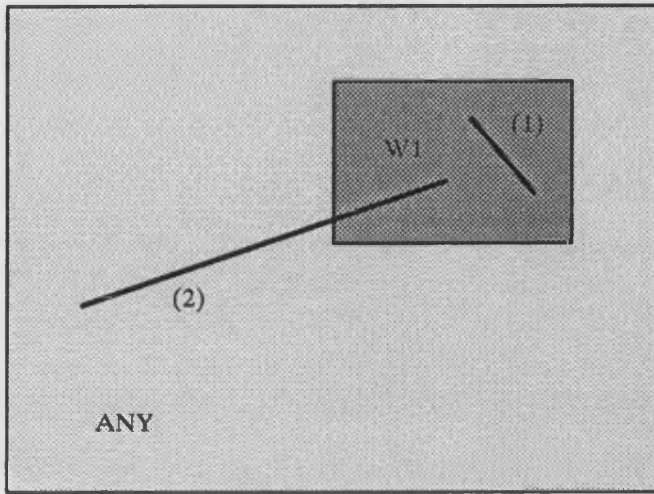
There is an obvious method by which we can solve this particular problem but it is also clear that the solution would remove advantages of the system we may wish to keep. The solution is to null all parameters when a task is re-initialised by the mutual-exclusion primitives. Thus in the example above, at step 4, prior to re-initialisation of T2 the parameter is null and so T2 would be forced to wait at least one scan of the task pool.

A disadvantage of this solution is that it prevents the sharing of parameters between tasks (section 4.2.4). This disadvantage is overcome by a simple extension to the proposed solution. We can either introduce an additional attribute for a task specification which indicates whether a task may or may not share parameters or we can introduce a similar attribute for a parameter declaration. This would indicate if the particular parameter is to be reusable. This second extension is preferable.

4.2.3. Binding parameters with a particular task

The method of choosing to which task a parameter should be bound needs attention. Consider two tasks, T1 and T2, which require two pairs of parameters (x,y coordinates). They have the same interaction requirements and perform the same function i.e. drawing a line between two user selected endpoints, with one subtle difference. T1 operates in Window W1 and T2 operates in Window ANY. See figure 4.1. The user interacts as follows:-

- (1) T1 is made active; The first (x,y) pair for T1 is collected by a button press in W1. This results in that point being fixed and rubberbanding ensuing. The rubberbanding is restricted to W1.
- (2) T2 is made active; the first (x,y) pair for T2 is collected, however this does not get decoded as the second (x,y) pair for T1 because the Event occurred outside W1. (Remember, an event/parameter is a combination of a buttontype and a window



- (1): Rubberbanded line in W1
- (2): Rubberbanded line in ANY

Figure 4.1. Binding Parameters with a Particular Task.

region.) Rubberbanding for T2 also commences.

We now have T1 and T2 both active, both having collected their first parameters and both are rubberbanding. We should note that T2 can rubberband over W1 because W1 is contained within window ANY. Therefore a button press in the W1 area will serve as the second (x,y) pair for both the tasks.

This scenario assumes that T1 and T2 exist in different mutual exclusion groups: it is this property that allows them to be active simultaneously.

Clearly, it is desirable in some cases to collect the second set of parameters separately for each task. This can be done at the application design stage. When the designer is allocating the groups to which tasks will belong, they can decide that in this case, T1 and T2 are essentially the same, so they can be assigned to the same group. Thus according to Rule 2 the above scenario is not possible, and the problem disappears.

4.2.4. Binding parameters with more than task

The converse of the above problem, for a parameter to be matched to more than one task, is often a requirement. This is easily done and again it is specified at the design level.

The designer puts the tasks with similar parameter requirements (where they have also decided that parameter sharing is an advantage) in two different groups. The natural hierarchy enforced by the scheduler ensures that the parameter gets used twice. Again with reference to the painting system UltraPaint we can illustrate this case with an example.

The Blend task is an editing operation on the colour palette in UltraPaint. Using the RGB colour model, the colour palette entries between two user-selected colours are blended to give a range of colours between them. The granularity of the blending

operation depends upon the number of entries between the two colours. Blend, like Vwipe (in Section 4.2.1), uses two button presses in the palette region.

Consider the following scenario. Task Vwipe is currently active and it has two of its four parameters collected, namely those which mark the rectangular region which will subsequently be painted. Task Blend is selected and so, following rule 1, Vwipe is saved. The user then supplies the Blend parameters, Blend is performed and subsequently freezes. Task Vwipe is initiated and since it has been specified so that it is executed after Blend, the parameters are used for this task as well. The fact that there is an inbuilt dependency between tasks could be construed as a disadvantage but in practice it reinforces the notion of a top-down structure. Thus in this particular example, Vwipe can reuse parameters originally collected for Blend but not vice versa.

4.3. Task management

This section emphasizes some of the ideas put forward in section 4.1 and presents discussion of some thoughts that arise from problems in task management.

4.3.1. Aborting tasks

There is often a requirement to abort a task. This may be for a number of reasons. The user may have inadvertently selected an undesirable parameter. For example a line may have had its first vertex positioned incorrectly. Or the user may simply no longer require that task. Whatever the reason, the actual mechanics of task abortion depends upon the type of the task and the problem environment. We are dealing with highly interactive applications, where the vast majority of the user's time is spent on waiting rather than doing things, thus there is no great overload on the user. It is reasonable to expect the user to carry out additional interactive activity and apply the existing facilities to return to the desired state.

The simplest method of implementing task abortion is via the menu. We could have allocated a specific task to abort other tasks but this would mean adding a hierarchy to the menu structure. Remember, that our menu items are already in a group structure. Instead, the mutual exclusion groups are used to implement the task abortion facility. Note that our meaning of task abortion is restricted to aborting those tasks which have not yet completed collection of their parameter set. Thus run to completion tasks cannot be aborted. Task abortion is quite distinct from 'Undo', in that 'Undo' is something that the designer would include in his or her design if it was necessary.

Tasks can be killed via the following methods. Selecting an alternative task will abort the first task if both tasks exist in the same mutual exclusion group (From Rule 2). Re-selecting the same task can have one of two effects. If part of the parameter set of the active task has been collected then the task remains active, but it returns to a stage where no parameters are available (The *start* stage of the task). If no parameters have been collected then the task is killed.

4.3.2. Run to completion tasks that are slow

An interactive application will often contain a run to completion task which takes more than thirty seconds in completing its purpose. Thirty seconds or more can be considered a long time in a highly interactive application because feedback about the progress of the task is desirable. Moreover, the user will not wish to be tied to that task, they may want to perform some other interactive task.

In the painting system UltraPaint, the Fetch command gets a picture from disk and puts it in the painting region. Disk access can be slow so a long delay is possible. We can show how to implement this function with the interaction model as follows.

Firstly, the Fetch procedure is designed so that it delivers portions of a picture from the disk. Each invocation of the Fetch procedure brings in a new portion of the picture. Thus the data structures holding the picture need to be of such a design to allow that.

Secondly, the Fetch procedure has to send some information back to the interaction model about its status. Thus when the final portion of the picture has been sent it returns a value via its function parameter.

```

:
case fetch:
  switch ( op ) {
  case Gather: If ( event (new, menu) ) run (t, start);
    else
      If ( Signal ) run (t, finish)
      else run (t, middle);

    break;
  case Perform:
    switch ( stage ) {
    case start: Bsync (t);
      readfile(); /* 1 */
      Suspend (t);
      break;
    case middle:
      If ( fetch () ) Signal (ON);
      Suspend (t);
      break;

    case finish:
      Freeze (t);
      Esync (t);
      break;

    }
    break;
  }
  break;
:
:

```

Thus this task behaves in the following manner. The task is first initiated by a button press in the menu. The procedure readfile is invoked to carry out the basic housekeeping for the Fetch procedure, such as opening files and setting the Signal flag.

On the second and subsequent cycles of the scheduler, Fetch is executed, this brings in portions of the picture. A check is made to see if all the picture has been fetched

and if so, a flag, is set. This flag is used to perform final task synchronisation.

By making it easier to break up a task into smaller sections it is possible to interleave long running tasks with other tasks the user may wish to perform. With this scheduler, other tasks can easily be set up and parameters collected easily.

4.3.3. Task hierarchies

There are essential two types of task hierarchy. There is a hierarchy between tasks defined within the same interactive application and there is a hierarchy over applications embedded within one another. In the first instance some tasks are permanently scheduled because they are required all the time: cursor tracking and menu management are good examples. Secondly, there may be a requirement to have nested graphical programs. For example, during the development phase of an application there will be a need to perform debugging operations. These operations may require the use of an interactive graphics debugging aid. Typically the debugging aid will have to be invoked by a menu option within the application currently being developed.

During the development of UltraPaint an interactive graphics debugging tool DEBUG[55] was produced. DEBUG is itself an interactive program, it has its own tasks, menus, and its own control of the puck. The tool was subsequently redesigned using our new interaction model.

This simple inclusion of an interactive program within another maintains the inherent topdown nature of the scheduler design but it does mean that there are some duplicate tasks. For example, the cursor will be read twice, once by the main application and secondly by the DEBUG program. There will also be two scheduling loops. Obviously, control will return to the outer scheduler once DEBUG has been terminated.

4.4. Multi-button pucks

Discussion in this and earlier chapters has centered around pucks which have only one button. Pucks with more than one button are modelled to behave as if they have only one. This restriction was chosen because it was felt that most interactive programs can have a satisfactory user interface with only one button. Only occasionally is it desirable to have more than one button.

Even so the interaction model must allow for this, and the model implements multi-button pucks in the following manner.

We can consider the press of a specific button as a special task, a primitive task in the same group as that of moving the cursor or detecting button presses in the menu. The following sample of C code is an example of the typical code that is generated when multi-button pucks are specified.

The task is specified in SIDL but it is given special attributes so that it is initially active, and is independent of being initialised from the menu. It is also an example of a task that does not freeze on completion of its associated primitive.

```
case leftbut:
    switch (op) {
    case Gather:
        If ( event (button (new, 2) ) run (t, start);
        break;

    case Perform:
        cursorsize (up);
        Suspend (t);
        break;

    }
    break;

:
:
```

Figure 4.2 Monitoring a specific button.

In the above example derived from the graphics debugger, a specific button press is detected and in this case the result is an increase in the size of the rectangular cursor. There is a corresponding task for decreasing the cursor size and this is associated with another button.

4.5. Window management

Typically, windows provide an environment for task interaction. Many applications however, also include interactions where the window itself is an active member. The interaction needs of such windows vary and we have broadly classified the following requirements.

1. Picking and dragging of objects/windows.
2. Resizeable windows.
3. Invisible Windows.
4. Overlapping Windows
5. Static or movable windows.

All these cases are required when implementing the user interface of window management systems. The user interface to such systems can be built using SIDL definitions, constructs and special primitives previously defined.

In Chapter 3 we described some the window types available. The general approach taken is as follows. Windows which can be resized or re-positioned (cases 2 and 5) have special regions generated for them. The interaction model has been extended to detect events in these regions and there are pre-defined tasks which carry out the required functionality. The Bitmap primitives used for scrolling purposes have been developed at Bath[56]. These tasks and regions are only generated if such windows are declared. Windows which are invisible can not have events associated with them.

4.6. Summary

In this chapter we have discussed some of the issues that arise in interactive graphical applications.

We have shown how our new interaction model overcomes menu synchronisation problems by enforcing a sub-structure over the elements in the menu.

We have described a number of interaction problems associated with parameter (user input event) collection. In particular, problems concerning multiple identical events, parameter reuse and binding of parameters have been highlighted.

The natural hierarchy of tasks within an interactive application has been described. In addition we have shown how multi-button pucks and their interaction requirements can be utilised by our new interaction model.

This chapter has indicated how the abstractions presented in our new interaction model effectively illustrate the problems and issues that surround graphical interaction management.

In the next chapter we will look at a number of alternative models for interaction management. The models will be presented and a comparison with our own model made.

5. Comparison of SIDL with Other Models

We need to place the interaction model and the specification language SIDL described in chapters two and three in the context of related models and techniques. This chapter is a detailed study of two methods mentioned in chapter one. Emphasis will be given to a specification language for interface design based on transition diagrams and secondly to object-orientated techniques for user interface design. These methods will be compared with our own approach.

5.1. The State diagram method

The use of state transition diagrams has a long history stretching back to Newman's "reaction handler" and they have also been used to provide a visual programming

metaphor[31]. However, for the purpose of this chapter we will concentrate on the recent work of Jacob[32].

In this work, direct manipulation user interfaces are discussed. A direct manipulation user interface is characterised by an interacting collection of active and/or responsive objects. These objects are graphical in nature. Once an object has been selected by the user, a dialogue associated with that object begins. Thus the user sees a multitude of small dialogues each of which may be interrupted or resumed under the control of a master dialogue. Note that this is similar to the behaviour of co-routines.

Traditionally, user interfaces are highly moded, which has made them eminently suitable to be represented by state transition diagrams in which the various modes are described by a particular state. Direct Manipulation or interactive interfaces however appear modeless. Many objects appear on the screen and the user can apply a standard set of commands to any object. Thus the system always appears to be in some "universal" or "top-level" mode. However a closer study reveals that there is in fact a number of distinct modes. For example, moving a cursor over a pixel object results in a mode change, because the set of possible user actions has altered. The user may or may not select the object, but at least the choice is there. While the cursor was not on the object, that choice did not exist.

If direct-manipulation interfaces are not really modeless why do they appear to possess the advantages of modeless ones? The main stumbling block to moded user interfaces normally occurs at the mode change boundary. If the mode change can be made transparent to the user then the interface will appear modeless.

The specification language can now be described by focussing on a number of points.

1. It is generally accepted that a direct-manipulation interface is comprised of a collection of objects[52] and thus the specification language is centered around a collection of individual objects called *interaction objects*. Each such object has its own dialogue specification. Such an object is also the smallest unit with which the user

conducts a meaningful dialogue.

2. The individual dialogues relate to each other as a set of co-routines, where each interaction object can suspend and resume from a retained state. There is also a master executive co-routine.

3. Despite the surface appearance, there is a definite set of modes or states and thus state transition diagrams are a suitable notation for describing the dialogue for individual objects. Each individual object conducts a single-threaded dialogue with serialised input and retained states whenever the dialogue is interrupted by that of another object.

The specifications of individual objects are combined into an *outer loop* by the use of a standard executive which operates by collecting the state diagrams of all the interaction objects and executing them as a collection of co-routines, assigning events to them and arbitrating between them as they proceed.

5.1.1. Tokens and component objects

A collection of low-level inputs and outputs which can be invoked by state diagrams is defined by Jacob. Examples for input are button clicks and moving into specific regions. For output they include highlighting, rubberbanding or other continuous feedback. The internal details of these tokens are specified in some other distinct manner.

Interaction objects may also be defined as a combination of other objects. They are automatically instantiated whenever the enclosing object is created. They are essentially instance variables.

Before we proceed with a discussion on the two models, it is helpful to give an example. The example below is derived from Jacob[32].

5.1.2. An Example specification

A common interaction found in graphical interfaces is that of the *ScrollBar*. The specification of *ScrollBar* is shown in figure 5.1. To use it the user points to it and presses a mouse button; then as the mouse is moved, the bar on the screen drags to follow it. When the button is released, the display is scrolled in proportion to the new position of the bar. **From** contains a list of other interaction objects from which this one inherits elements. **Ivar** is a list of instance variables and their initial values. **Methods** are procedures unique to this object which are essentially the semantic component of the interface. **Tokens** are definitions of each input and output token used in the syntax diagram. **Syntax** is the input handler for this object. The diagram specifies the sequence of the dialogue. States where the dialogue can be suspended are shown with "+".

In this particular interaction object, the diagram explicitly handles the unusual case where the user depresses the button and, while holding it down, exits the scroll bar, possibly performs other interactions, reenters the scroll bar with the button still pressed down. This object will resume dragging when the cursor enters the scroll bar. For comparison the scroll bar specification in SIDL is presented below.

Jacob's specification has particular states where the interaction can be suspended. In the scrollbar specification, these states are: prior to the scrollbar interaction being initiated; the cursor entering the scrollbar region; during the update of the scrollbar position and finally when the interaction has been completed. Similarly, the SIDL specification also has states: prior to the task being initiated; during a scrollbar update; and finally on completion of the task.

A clear difference between the two scrollbar specifications is that the SIDL specification does not allow for the unusual case described above. In the paper, Jacob has not made it clear if the user performs other actions with the button pressed

INTERACTION_OBJECT Scrollbar is

FROM: GenericItem;

IVARS:

position;

legend; := "Scroll";

METHODS;

Draw () { DrawBar (position, legend, scrollOffset); }

TOKENS;

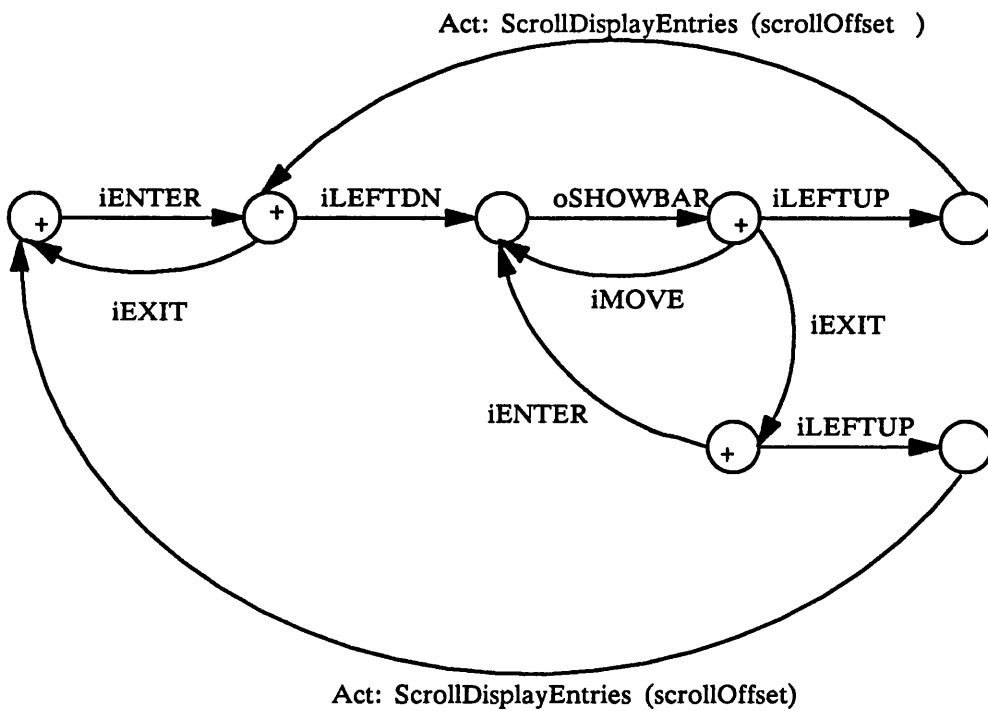
iMove { --Any mouse motion within boundaries of position,
--return scaled X coordinate of mouse--}

iLEFTDN { -- Overloads standard definition of iLEFTDN with one that accepts
-- same click then sets scrollOffset:= scaled X coord of mouse-- }

oSHOWBAR { -- Fill or erase bar up to location corresponding to scrollOffset --}

SYNTAX:

main



end **INTERACTION_OBJECT**

Figure 5.1. Specification of a Scroll Bar. (From [Jacob 86])

down. Other user interaction may require the button in a released state, thus handling this unusual case is unnecessary complication. For example, the user may need to manipulate another object. On completion of the second interaction, they return to the scrollbar interaction. The suspended state of the scrollbar interaction requires the user to enter the scrollbar region with the button pressed. If the scrollbar is entered with the button in an up state then the scrollbar interaction is re-initiated from the first state. The provision of the additional state is unhelpful to the user and is not provided by SIDL.

A useful advantage of the SIDL specification is that the scrollbar can be adjusted by keeping the button pressed with the cursor anywhere on the screen. Thus it is not necessary to maintain the cursor over a small region and perform intricate interactions. Jacob's specification restricts the interaction to the scrollbar region.


```

Interaction test;

Type
    :
    :
Endtype

Window scrollbar_region;
Do
    /* scrol bar size and current position */
End

Task scrollbar;
Do
    Gather;
    If New Button(1) In scrollbar_region Then
        run(t,start);
    Else
        If Pressed Button(1) In any Then
            run(t,middle);
        Else
            If Released Button(1) In any Then
                run(t,finish);
        End
    End
    Perform;
    Start: grab_scrollbar;
        suspend(t);
    Middle: move_scrollbar();
        suspend(t);
    Finish: position_scrollbar();
        scroll_region();
        suspend(t);
End

```

Fig 2. SIDL Specification of a scroll bar.

5.1.3. Discussion of the state diagram method

The State Diagram model describes the main interaction in the graphics area as a single thread dialogue with a main command loop. The command loop is implemented as a "super co-routine". Our interaction model also has the same basic structure. As described in chapter three, it is composed of cyclical scan of a task pool. Each active task enters its own dialogue and suspends according to its state and the number of events it has processed. In the interaction model the suspension of tasks is controlled by primitives embedded within task bodies whereas in the state model all such control is determined by the executive.

A direct manipulation interface is described as a collection of interaction objects, each object being implemented as a co-routine. In the interaction model, an interactive graphical application is also decomposed into individual tasks through which the user conducts a dialogue. Here the principles are the same with just alternative terminology.

A co-routine goes through a number of states depending on the complexity of the dialogue. It only suspends on those states where it expects input. Similarly Tasks go through a number of states. However in the interaction model there are initially only three states: start, middle and finish. If a dialogue is sufficiently complex to require further states then the middle stage is sub-divided. This will only occur when there is a large number of events to be processed, in which case the criterion for sub-division is determined by the parameter count mechanism as described in chapter four. The state diagram model simply includes additional states as necessary, but in a less structured way.

When a co-routine does suspend, the executive retains the state of that co-routine. As additional input tokens arrive, they are arbitrarily supplied to other co-routines currently suspended and waiting for that token. If there is more than one co-routine waiting for that same token a random choice is made. Jacob claims that in typical designs there will only be one co-routine waiting. Similarly in the interaction model, events/tokens are only supplied to active tasks (this is analogous to a suspended co-routine). In the case of menu-driven applications the problem of more than one task waiting for the same event is abstracted out by use of an enforced structure over a menu. This is explained in detail in section 4.1. In simpler cases the cyclical scan of the scheduler determines which active task receives the token.

The single threaded nature of the dialogue is inherent in both models: in one it is represented by state transition diagrams; in the other it is described by procedural statements.

In both models, cases of invalid events/tokens can easily be handled. In the state model, this normally requires an additional state and arc within the diagram. In our model the invalid events are removed at the procedural level. In both cases input/output primitives are specified separately typically by the use of an orthodox programming language. In the state model the presentational component is handled separately whereas in the interaction model the presentational component is specified within the Interaction program albeit in a separate section.

There is no distinction made between the user and application interface in the state model. Both interfaces are embedded within the syntax section. In the interaction model the dichotomy is clearly visible and manifests itself by the Gather and Perform sections.

Finally, Jacob has only specified simple interaction dialogues are. No attempt has been made to indicate how tasks requiring multiple identical events can be catered for.

Concluding, there are basic similarities between the two models. Differences appear to be cosmetic apart from the decision to implement as a set of co- routines or as a fixed scan.

5.2. The object oriented approach

In this section we present a brief overview of the history and underlying concepts of object oriented programming (OOP), we show how it has been used to design user interface management systems and more significantly we show how OOP has been used to study the interactions in graphical programs .

5.2.1. History and basic concepts

The history of object oriented programming has its ancestry in the programming language Simula. However the term became first associated with Smalltalk[21].

Smalltalk is a sufficiently general instance of an object oriented programming language that a treatment of it will suffice for both.

The Smalltalk programming environment was one of earliest products from the Learning Research group at the Xerox Palo Alto Research Center. Smalltalk is comprised of four pieces, a programming language kernel, a programming paradigm, a programming system and a user interface model. There is not however a set of clean cut boundaries.

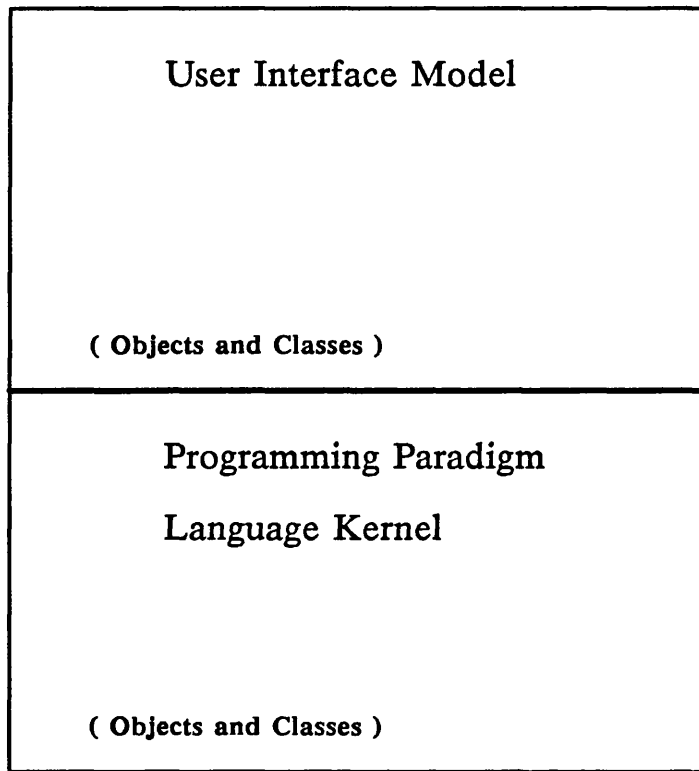
The syntax and semantics of the Smalltalk compiler are provided by the *programming language kernel*. The *programming paradigm* is the style of use of the kernel. The *programming system* is the set of system objects and classes that provide the framework for using the kernel and the paradigm. The *user interface model* is a combination of the given user interface and the tailored user interface. It is the use and usage of the systems building materials.

5.2.2. Basic concepts

Objects

The Smalltalk world (figure 5.2) is populated by items seen uniformly to be "objects". These objects are the sole inhabitants of a universe. In the diagram, the objects and classes referring to the programming paradigm and the language kernel are the object oriented aspects of Smalltalk.

Objects are uniform in that they all have the following properties:- inherent processing ability, message communication and a common appearance, status and reference. Further, no object is given any particular status, thus a "primitive" object such as integer has the same class and properties as a user defined object. An object is also referenced as a whole. An object cannot act as if it has been opened unless it has been given the means to display that type of behaviour. Thus an object can behave



Programming System Piece

Figure 5.2. The Smalltalk World

like a Pascal record type if it has been given the methods to do so.

The individually accessible components of an object are called the *instance variables*.

They can be both named and indexed. For example.

1. An object of class Point has named instance variables x and y which identify the coordinates of a point.
2. An object of class Array contains only indexed instance variables. These are identified by the integers 1 to the number of instance variables of the array.

Objects are self-describing, they include sizing information (number of instance variables) and the class to which they belong.

Classes are the program modules of Smalltalk. Like the abstract data types provided by the modules of Modula-2 and the packages of Ada[60], a class specifies the instance variables contained in the objects of that class and the methods (functions) that operate on the objects.

Smalltalk Classes are organised into a hierarchy with the class Object at the top. Superclasses are more generic; sub-classes are more specialised. A class inherits the named instance variables and methods of its superclasses.

Consider part of the Smalltalk hierarchy for the class Magnitude.

```
Magnitude
  Character
  Date
  Time
  Number
    Float
    Fraction
    Integer
```

The class Magnitude contains methods for calculating the maximum and minimum of numbers using comparison operators. The class Date will compare two dates, the class Float will compare two floating point numbers.

A Smalltalk terminology translation adapted from Anderson[3] is given below:-

Method	:	a function definition
Message	:	the invocation of a method i.e. a function call
Protocol	:	the specification of how a message is sent to a method including the method name and parameters
Object	:	a record of fields
instance variable	:	a field of a record
Class	:	a record type and all the functions that may be applied to record type.

Processing and Communication

In Smalltalk all processing activity takes place inside an object since an object is responsible for providing its own computational behaviour. While an object is carrying out some processing it may be independent of other objects but at other times it must have a means of communicating with them. This is achieved by the mechanism of message passing.

If a user wants an object to carry out some computation, they send that object a message. If that object requires data or some further sub-computation to be performed by another object it sends that second object a message.

Message sending is uniform in that the same mechanism is used for both a simple addition and for a complex file service operation.

Although a message is very similar to that of a function call, there is a subtle difference in that the caller of the function is not in control as in orthodox programming systems. In OOP the sender relinquishes control both philosophically and actually, so the interpretation of the message is left entirely up to its recipient.

5.2.3. OOP in user interfaces

In this and subsequent sections we will describe some applications of using OOP in the design and implementation of user interfaces. Note that OOP has digressed from its original meaning (described above with respect to Smalltalk) and it has now come to mean abstract data types, data encapsulation, modularisation. This section will utilize some of the newer meanings.

5.2.4. OOP for studying interaction

OOP techniques have been used to prototype video game design [37]. Designing video games is an extremely expensive and complex process. The user interaction functions are particularly complex because of the real-time components of a video game. The low-level visual effects are also a difficult element. In the first instance, however, the designer must conceptualize his design and in a ideal situation prototype this design. To this end, Larrabee and Mitchell have designed and implemented a special purpose language which has made the designers' job easier and therefore more creative. In the following section an outline of their approach is described and a comparison with the author's work is also presented.

A game in the special language (Gambit) is different from games in other languages. In most languages the objects on the screen are considered as part of the game's global state (possibly as entries in large table) with the games controller maintaining the table and performing any necessary calculations. In Gambit however, an object oriented model is employed. Objects on the screen are not simply entries in a global table, instead, they are described as objects in their own right, with each object possessing its own local state and the necessary intelligence to maintain that state.

In most object-oriented languages a rigid chain of control is defined, with one object controlling the game and sending messages to subordinate objects as necessary. In

Gambit this hierarchy is hidden from the user, all programmer defined objects are equal, they all interact with the system and with other objects. A Gambit object is an instance of user-defined classes that are in a class definition module. Each class definition contains a specification of what messages an object of that class can receive and send and what actions to perform on receipt of a particular message.

At runtime, the objects are instantiated and they interact with the system facilities and communicate with other objects via messages.

Space

Objects usually have a display of type picture and a location of type point. Together these definitions describe the appearance and position of the object. Objects overlap by being layered (similar in analogy to the desktop metaphor in window management systems). Objects can also have an interaction boundary and when the interaction boundaries of objects overlap the system detects this and notifies the objects if so required. Note not all objects will be interested in this condition. For example, an interaction may occur between a ball and a wall, but only the ball needs to be notified of the collision.

Time

Time is modeled as a master clock that ticks at preset intervals, thus time is relative rather than actual. This forms the basis of a scheduler. At each tick a fixed cycle of activities is performed.

1. All new user input is detected and the appropriate objects notified.
2. Objects that have requested 'wake up' times are sent messages after the appropriate number of ticks.
3. Collisions are detected.
4. Objects requiring collision information are notified.
5. The display is updated.

Events

Asynchronous activity such as a button press is defined as a constant. Event constants belong to classes defined by **EventNames**. Any object can declare an interest in an event by the simple inclusion of the event constant in its definition.

The syntax of Gambit is designed in such a way that comparisons with Pascal and Smalltalk are meaningful. Pascal-like syntax is used for constructs having the same semantics as Pascal. Inter-object communication is borrowed from Smalltalk.

5.2.5. Discussion of the object oriented approach

The particular relevance of this implementation is that object oriented techniques have been used to model a highly interactive environment.

Both Gambit and SIDL use Pascal-like syntax for operations which have the same semantics. For semantics which can not be represented by existing language forms, Gambit uses a Smalltalk adaptation whereas SIDL introduces its own.

Both languages introduce discrete objects into the interactive model. The operations that each object can perform and the conditions under which they are performed are explicitly declared within the object.

Gambit's representation of objects is definitely OOP, whereas SIDL tasks are token representations of the concepts. Also the design of the SIDL language is closer to the procedural block structured model. This essential difference needs closer examination and is discussed below.

The OOP model does not require a scheduler in the form used in SIDL. But the operation of the clock provides the same semantics. A specific number of activities are performed at each tick. We can consider a tick to be equivalent to a single cyclical span of the tasks defined in a SIDL program.

One of the activities in Gambit monitors all user events. In SIDL a number of tasks will monitor events that can be considered global; i.e. events that are of interest to all tasks. Events which are specific to particular tasks are monitored by those tasks themselves. In Gambit, a separate activity informs objects of events just received. This bears comparison to a task monitoring events in the menu region of a menu-driven program. The task detects a menu button task and then activates (informs) the task associated with that button. Further, Events are user defined as constants. Objects which will use these events are defined so that they see these event constants. This is advantageous as duplication is avoided, however some clarity is lost.

The 'detonate' command normally sends a message to an object when either the correct number of ticks has passed or when a specific event which the object is interested in has happened. This is similar to the behaviour of a task which has been suspended until it receives the event information it needs to proceed. The number of ticks could be counted by an additional task which is always active and is therefore always scheduled.

The advantages of a proper OOP model become obvious in the case where collisions between objects are detected and the objects are subsequently informed. This is a lot simpler and cleaner with the OOP model when we consider how we would implement something similar using SIDL. We need tasks which simply update the screen with the current position of the object. This can either be implemented as separate tasks for each object, or one task for all the objects. The tasks will be permanently active and thus be scheduled with every cycle of the scheduling loop. An additional task is necessary to detect collisions using global information.

Interactions between objects are not easily implemented in SIDL because there is usually a need for additional task definitions and the use of complex data structures. Gambit achieves this by the message passing paradigm indigenous to OOP.

If we consider Gambit a bona fide example of Object-Oriented design then the preceding discussion has indicated the concepts which SIDL can and cannot easily implement, thus the discussion has served as a barometer informing us to what extent SIDL is object oriented. In particular all the scheduling and asynchronous user event monitoring can be implemented and it is only inter-object communication which poses problems.

A general conclusion is that object-oriented design can be mimicked by use of large global tables which represent the status of objects.

5.3. Summary

In this chapter two contrasting methods for interactive systems design have been described. A comparison between these models and our own interaction model has also been presented.

Firstly, an established dialogue design method (state machine) was described. We have shown how this dialogue method, when extended for visual programming purposes, bears a number of similarities to our own model. A typical interaction dialogue has been illustrated using both Jacob's method and ours. Some of the weaknesses in Jacobs's method have also been indicated.

Secondly, an overview of the fundamental concepts of OOP using Smalltalk-80 have been provided. This is important because an increasing number of user interface design methodologies display object oriented characteristics. We have described an object oriented system for Video Game design (Gambit). This system has been compared to our interaction model and the comparison has provided us with a measure of the extent of the 'object oriented' nature of our interaction model.

In the final chapter we give an indication of the success of this research and we provide some pointers for future research in this field. We place particular emphasis

on the software engineering aspects of user interface design.

6. Conclusion

The automation of the production of user interfaces has been the main focus of research in this field. To this end, a number of models for user interface specification have been developed. The automation of user interface production, however, has successfully evaded the real and more important issue of constructing user interfaces which are based on sound software engineering principles. Even more importantly, the ubiquitous nature of interactive graphical software, like that of concurrent software, requires the *extension* of basic software engineering principles.

It was the author's aim to identify the interaction requirements of interactive applications. To be able to perform this task successfully, a number of such applications were evaluated. The exercise was undertaken by considering a number of significant issues. Firstly, the evaluation had to place the typical design structure of interactive applications in relation to existing software engineering methodologies. Thus, software could make use of structured programming techniques but the resulting

programs were not necessarily well structured.

At a lower level, the construction of individual components of typical applications was also studied. In this case it was very clear that the existing methodologies were not sufficient. In a wider context, user interface research has not adequately addressed the issue of providing mechanisms of constructing those low-level interaction blocks which give interactive graphics programs their unique characteristics. The general approach has been to provide these blocks as libraries and the developer has no idea as to how these blocks have been engineered. Where such details are available, the over-riding impression is a lack of visible software engineering methodology.

Having identified the interaction requirements the next step was to provide a new set of control constructs which were capable of being used alongside more traditional constructs. Further, the fundamental *'repeat ... until'* loop commonly found in interactive applications was one of the principle sources of problems when considering the issues of maintainability and extendibility. Thus a new basic interaction model was designed. This new model allows the designer to place both the user and the application code in small object-like units. The control constructs can be applied consistently so that maintenance and extension of applications constructed with the new model are not major problems.

The new model was tested by implementing some interactive graphical applications to examine the feasibility of the model as a new design tool. Although the results were successful, some additional tool support was still necessary: the final result was a specification language SIDL and its preprocessor GRIP.

Future Work

There are a number of points where improvements can be made. We can continue to identify additional higher level constructs which are geared towards particular types of

graphical interaction. However, the author believes that these additional constructs will essentially be comprised of the constructs identified in this thesis.

The early evaluation of an interface design is an important area of research and there is potential for a number of software tools for this area. Evaluation tools operating on SIDL specifications could conceivably perform the following operations.

- Detection of Task Deadlock. This is the case where a task remains in an active state because the event it is waiting for cannot be collected.
- Detection of Windows defined outside the screen area.
- Reporting of all overlapped windows, so the user has at least the option of deciding whether the specification meets its requirements.
- Comparing specified events with events that are actually possible. For example, an event may have been specified but it may not be possible for it to occur (perhaps because the window with which the event is associated is outside the screen area).
- Human factors. Here we could look at the colours which have been specified, the position of the menu and the typical puck movement across the tablet, for example. This particular type of evaluation is currently subject to much research and some of the anticipated results are suitable for inclusion in such software tools.

Research into integrated project support environments (IPSEs) has now finally come to fruition. IPSEs provide a total software environment for all phases of the software cycle. In addition the environment is highly structured and provides both security and control over all objects within the environment.

Typically, an IPSE will include a number of software tools: a design methodology tool; editors; language compilers; configuration management tools and project management tools. It is now a basic requirement of all IPSEs that the user interface across all tools

and indeed across the entire IPSE should be consistent. SIDL and GRIP could both provide a consistent user interface and also be considered as a suitable tool for integration into an IPSE. SIDL specifications could form part of design document deliverables and the generated programs could be code deliverables.

The science of Software Metrics is now successfully used on conventional software. Tools are available which perform complexity analyses on source code. This same approach could now be applied to graphical software designs based on the New Interaction Model. The analysis would be performed at the design specification level rather than the source code level.

In this thesis, the author has identified the shortcomings in existing interactive software design. Standard programming constructs are not powerful enough to fully express the complexities of an interactive graphics application. This basic inadequacy has been successfully translated into a number of interactive programming constructs which, combined with the New Interaction Model, provide a means of constructing user interfaces which embody sound design technique.

The New Interaction Model and the programming constructs have also provided us with a vehicle to discuss the complex issues that surround graphical interaction. In chapter 4 we provided a detailed discussion of such issues. Further, this work also advances the possibility of a more quantitative approach to examining the needs of graphical application.

References

1. Sun Microsystems Inc., *Sunview Programmer's Guide*, Sun Microsystems Inc., Mountain View, Ca. (1986).
2. A.V Aho and J.D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, Mass. (1977).
3. J. Anderson and B. Fishman, "An Introduction to Object-Oriented Programming," *BYTE*, pp. 160-165 (May 1985).
4. Ed Anson, "The Semantics of Graphical Input," pp. 115-126 in *Methodology Of Interaction*, ed. Guedj, North Holland Co. (1980).
5. Ed Anson, "The Device Model of Interaction," *ACM SIGGRAPH Computer Graphics* 16(3) pp. 107-114 (1982).
6. B.S Barn, "Graphical Interaction for Font Editors," Internal Report, University of Bath (1985).

7. B.S Barn and P.J Willis, "Graphical Interaction Management," *Computer Graphics Forum*, (6) pp. 119-124 (1987).
8. B. W Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ. (1980).
9. W. Buxton, M.R Lamb, D. Sherman, and K.C. Smith, "Towards A Comprehensive User Interface Management System," *ACM SIGGRAPH Computer Graphics* 17(3) pp. 35-42 (1983).
10. S. Card, P.Moran, and B.Newell, *The Psychology of Human Computer Interaction*, Lawrence Erlbaum Associates, Hilldale, NJ. (1981).
11. L. Cardelli and R. Pike, "Squeak: A Language for Communicating With Mice," *ACM Siggraph Computer Graphics* 19(3) pp. 199-204 (1985).
12. M.E Conway, "Design of a Separable Transition-Diagram Compiler," *Comm. ACM* 6(7)(1963).
13. J. Coutaz, "The Box, A Layout Abstraction For User Interface Toolkits," Technical Report, CMU-CS-84-167, Carnegie-Mellon University (1984).
14. J. Coutaz, "Abstractions for User Interface Design," *IEEE. Computer* 18(9)(1985).
15. E.W Dijkstra, "Goto Statement Considered Harmful," *Comm. ACM* 11(3) pp. 147-148 (1968).
16. E.W Dijkstra, *A Discipline Of Programming*, Prentice-Hall, Englewood Cliffs, NJ. (1976).
17. D. A Duce, "Concerning the Specification of User Interfaces," *Computer Graphics Forum* 4 pp. 251-258 (1985).
18. J.D Foley, V.L Wallace, and P. Chan, "The Human Factors of Computer Graphics Interaction Techniques," *IEEE Comp. Graphics and Applications*, pp. 13-48 (November 1984).

19. J.D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison Wesley Inc. USA (1982).
20. D. Gangopadhyay, "A Framework for Modelling Graphical Interactions," *Software Practice and Experience* 12 pp. 141-151 (1981).
21. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass. (1983).
22. M. Green, "A Methodology For The Specification Of Graphical Interaction," *ACM SIGGRAPH Computer Graphics* 15(3) pp. 99-108 (1981).
23. M. Green, "Report on Dialogue Specification Tools," *Computer Graphics Forum* 3 pp. 305-313 (1984).
24. M. Green, "The University of Alberta UIMS," *ACM Siggraph Computer Graphics* 19(3) pp. 205-213 (1985).
25. M. Green, "Design notations and User Interface Management Systems," pp. 99-108 in *Seeheim workshop on User Interface Management Systems*, North Holland Co. (1985).
26. P.J.W ten Hagen and J. Derksen, "Parallel Input and Feedback in Dialogue Cells," pp. 109-124 in *Seeheim workshop on User Interface Management Systems*, North Holland Co. (1985).
27. F.R.A Hopgood, D.A. Duce, and D.C. Sutcliffe, *Introduction to the Graphical Kernel System (GKS)*, Academic Press, London (1983).
28. S.E Hudson and R. King, "Efficient Recovery and Reversal in Graphical Interfaces Generated by the Higgs System," *Graphics Interface '85*, pp. 151-158 (1985).
29. M. A Jackson, *Principles Of Program Design*, Academic Press, London (1975).
30. R.J.K Jacob, "Using Formal Specifications in the Design of a Human Computer Interface," *Comm. ACM* 26(4) pp. 259-264 (1983).

31. R.J.K Jacob, "A State Transition Diagram Language For Visual Programming," *IEEE. Computer*, pp. 51-59 (August 1985).
32. R.J.K Jacob, "A Specification Language for Direct-Manipulation User Interfaces," *ACM Trans. On Graphics* 5(4) pp. 283-318 (October 1986).
33. P. Johnson and M. Keane, "Preliminary Analysis for Design," Draft Document, Queen Mary College, University of London (1987).
34. S.C Johnson, "YACC: Yet Another Compiler-Compiler," *Computer Science Technical Report 39*, Bell Labs, (1975).
35. A. Kamran and M.B Feldman, "Graphics Programming Independent Of Interaction Techniques and Styles," *ACM Siggraph Computer Graphics*, pp. 58-66 (January 1983).
36. D. Kasik, "A User Interface Management System," *ACM SIGGRAPH Computer Graphics* 16(3)(1982).
37. T. Larrabee and C.L. Mitchell, "Gambit: A Prototyping Approach to Video Game Design," *IEEE Software*, pp. 28-36 (1984).
38. M.E Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator," *Computer Science Technical Report 32*, Bell Labs, (1975).
39. H. Lieberman, "Constructing Graphical User Interfaces by Example," *Proc. Graphics Interface Conf.*, pp. 295-302 (1982).
40. H. Lieberman, "There's More to Menu Systems Than Meets the Screen," *ACM SIGGRAPH Computer Graphics* 19(3) pp. 181-189 (1985).
41. W. R Mallgren, *Formal Specification of Interactive Graphics*, MIT Press (1982).
42. Tom De Marco, *Structured Analysis and System Specification*, Yourdon Inc., New York (1978).

43. B. A Myers and W. Buxton, "Creating Highly Interactive and Graphical User Interfaces by Demonstration," *ACM Siggraph Computer Graphics* 20(4) pp. 249-258 (1986).
44. W. Newman, "A System For Interactive Graphical Programming," *Spring Joint Comp. Conf.*, pp. 47-54 (1968).
45. D.R Olsen, "Automatic Generation Of Interactive Systems," *ACM SIGGRAPH Computer Graphics* 17(1) pp. 53-57 (1983).
46. D.R Olsen, W. Buxton, D. Kasik, R. Ehrich, J. Rhyne, and J. Sibert, "A Context for User Interface Management," *IEEE Computer Graphics and App.*, pp. 33-42 (December 1984).
47. D.R Olsen and E.P Dempsey, "SYNGRAPH: A Graphical User Interface Generator," *ACM SIGGRAPH Computer Graphics* 17(3) pp. 43-50 (1983).
48. D.L Parnas, "On The Use Of Transition Diagrams In The Design Of A User Interface For An Interactive Computer System," *Proc. 24th Natl. ACM Conf.*, pp. 379-385 (1969).
49. P. Reisner, "Formal Grammar and Human Factors Design of an Interactive Graphics System," *IEEE Trans. Soft. Eng.* SE-7(2) pp. 229-240 (1981).
50. R.W Scheiffler and J. Getty, "The X Window System," *ACM Trans. Graphics* 5 pp. 79-89 (April 1986).
51. J. Schoenhut, "Tutorial on the Graphical Kernel System (GKS)," Technical Document, Friedrich-Alexander-Universitat, Erlangen-Nurnberg, Federal Republic of Germany (1984).
52. B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer*, pp. 57-70 (August 1983).
53. J.I Sibert, W.D Hurley, and T.W Bleser, "An Object Oriented User Interface Management System," *ACM Siggraph Computer Graphics* 20 (4) pp. 259-268

- (1986).
54. P.P Tanner and W. Buxton, "Some Issues In Future UIMSs," pp. 67-79 in *Seeheim workshop on User Interface Management Systems*, North Holland Co. (1985).
 55. G.W Watters, "Debug," Internal Document, University of Bath (1986).
 56. G.W Watters, "The Bath fs Graphics Library," Internal Document, University of Bath (1986).
 57. G.W Watters and P.J Willis, "UltraPaint: A New Approach to a Painting System," *Computer Graphics Forum* 6(2) pp. 125-132 (1987).
 58. P.J Willis, "A Paint Program For The Graphic Arts," *Proc. of EuroGraphics Conf.*, pp. 109-120 (1984).
 59. M. Young, R. Taylor, and D. Troup, "Software Environment Architectures and User Interface Facilities," *IEEE. Trans on Soft. Eng.* SE-14(8) pp. 697-708 (1988).
 60. S. Young, *An Introduction to Ada*, Wiley & Sons (1982).
 61. E. Yourdon and L.L. Constantine, *Structured Design*, Prentice-Hall, Englewood Cliffs, NJ. (1978).

Additional Bibliography

The following references were consulted but were not cited in the thesis. They are included here to provide a more complete picture of this field of research.

1. J.L Bennett, "Tools for Building Advanced User Interfaces," *IBM Systems Journal* 25(3)(1986).

2. G. Carson, "The Specification of Computer Graphics systems," *IEEE Comp. Graphics and App.*, pp. 27-41 (September 1983).
3. C. Frasson and M. Erradi, "Graphics Interaction in Databases," *Graphics Interface '85*, pp. 75-81 (1985).
4. J. Gait, "An Aspect of Aesthetics in Human-Computer Communications," *IEEE Trans. Soft. Eng.* SE-11(8) pp. 714-717 (1985).
5. M.J Goodfellow, "WHIM, the Window Handler and Input Manager," *IEEE Comp. Graphics & App.*, pp. 46-52 (1986).
6. P. J Hayes, "Executable Interface Definitions using Form-Based Interface Abstractions," Computer Science Technical Report, CMU-CS-84-110, Carnegie-Mellon University (1984).
7. H. Lieberman, "Seeing what your programs are doing," *Int. J. Man-Machine Studie* 21 pp. 311-331 (1984).
8. T.E Lindquist, "Assessing the Usability of Human-Computer Interfaces," *IEEE Software*, pp. 74-82 (January 1985).
9. R.L London and R.A Duisberg, "Animating Programs Using Smalltalk," *IEEE Computer*, pp. 61-71 (August 1985).
10. J.A Pitcairn-Hill, "Menus and Menu Systems, An Approach to the User Interface," Computing Laboratory Report No 24, University Of Kent (1984).
11. M.H Richer and W.J. Clancey, "Guidon-Watch: A Graphic Interface for Viewing a Knowledge-Based System," *IEEE Comp. Graphics & App.*, pp. 51-64 (November 1985).
12. M. Shaw, E. Borrison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch, "Descartes: A Programming Language Approach to Interactive Display Surfaces," *ACM Sigplan Notices*, pp. 100-111 (1983).

13. P.P Tanner, D.A Mackay, D.A Stewart, and M.Wein, "A Multitasking Switchboard Approach to User Interface Management," *ACM Siggraph Computer Graphics* **20(4)**(1986).

Appendix A. The SIDL Syntax

YACC Grammar

Note:

The following symbols are meta-symbols belonging to the extended BNF formalism and not symbols of the language SIDL.

`::=` | `{ }`

The curly brackets denote possible repetition of the enclosed symbols zero or more times.

IDENT and INTCONST are higher level representations for identifiers and integer constants respectively.

SIDL keywords are in bold upper-case.

```

program      ::=  restofprogram
                ;

restofprogram ::=  programheading block END ':'
                ;

programheading ::=  INTERACT IDENT ';'
                ;

block        ::=  optdefnpt optwindandtaskpt optstatpt
                ;

optdefnpt    ::=  empty | TYPE doblk ENDTYPE
                ;

definitions  ::=  typekeywd IDENT
                | MENU IDENT
                | typekeywd INTCONST | typekeywd '=' types
                | partype OF PARAMS
                | IDENT ':' OF VARIABLE
                | IDENT ':' OF MOTILE
                | MUTEXL '=' mutype OF TYPE 3INTCONST
                | INIT '=' ACTIVE
                | COLOUR colconst rgbvalue lutentries limitcol
                | constraintdec
                | constraintuse
                | IDENT ':' OF bmaptype
                | parstatustype
                ;

parstatustype ::=  PARSTATUS pspars
                ;

pspars       ::=  '(' psblk ')'
                ;

psblk        ::=  { ';' IDENT }
                ;

bmaptype     ::=  POPUP | STATIC
                ;

colconst     ::=  IDENT ;

rgbvalue     ::=  '(' lut_entry ';' redv ';' greenv ';' bluev ')'
                ;

lut_entry    ::=  INTCONST

```

```

;
redv ::= INTCONST
;
greenv ::= INTCONST
;
bluev ::= INTCONST
;
lutentries ::= empty | INTCONST
;
limitcol ::= empty | colours
;
colours ::= BLACK
           | WHITE
           | RED
           | GREEN
           | BLUE
           | MAGENTA
           | CYAN
           | YELLOW
;
mutype ::= '(' mutypeblk ')'
;
mutypeblk ::= { ',' IDENT }
;
types ::= '(' typeblk ')'
;
typeblk ::= { ',' IDENT }
;
typekeywd ::= WINDOWSSY | NUMTASKSY
;
constraintuse ::= CONSTRAINT ':' IDENT
;
constraintdec ::= CONSTRAINT IDENT regionorline
;
regionorline ::= REGION ':' IDENT
                | LINE ':' horvertline conid
;
conid ::= /* null */ | '(' conidblk ')'
;

```

```

conidblk ::= xylevel | xylevel ',' leftvert ',' rightvert
          ;
xylevel  ::= INTCONST
          ;
leftvert ::= INTCONST
          ;
rightvert ::= INTCONST
          ;
horverline ::= HORIZ | VERTICAL
           ;
optwindandtaskpt ::= empty | windtaskdecpt
                  ;
optstatpt ::= empty | statementpt
           ;
windtaskdecpt ::= windortaskdec | windtaskdecpt windortaskdec
               ;
windortaskdec ::= windowdec | taskdec
               ;
windowdec ::= windowheading block
           ;
windowheading ::= WINDOW IDENT ';'
              ;
taskdec ::= taskheading block
        ;
taskheading ::= TASK IDENT ';'
            ;
statementpt ::= compoundstment;
            ;
compoundstment ::= DO doblk END
                ;
doblk ::= | statement | doblk ';' statement
       ;
statement ::= empty
           | simpstatement
           | structstment
           | winassnstment
           | c_statment
           | definitions
           ;

```

```

c_statement ::= '$' c_body '$'
              ;

c_body ::= empty
        ;

winassnstmt ::= winkeywd ':' INTCONST
              | GRID ':' horvert '=' INTCONST ;

horvert ::= NUMH | NUMV
         ;

winkeywd ::= HT
          | WIDTH
          | YBOT
          | XBOT
          ;

simpstatement ::= GATHER
                | PERFORM
                | VARIANT
                | procstatement
                | STUB
                | COLLECT
                | INTCONST ':' statement
                ;

procstatement ::= procidentifier | IDENT '(' parblock ')'
              ;

procidentifier ::= IDENT
               ;

parblock ::= { ',' identconst }
          ;

identconst ::= IDENT | INTCONST
           ;

labelstment ::= startstment | middlestment | finishstment
            ;

startstment ::= START
            ;

middlestment ::= MIDDLE
            ;

finishstment ::= FINISH
            ;

structstment ::= compoundstment
              | condstment
              | stagestment
            ;
    
```

```

| usecondstment
| parsdatablk ;

parsdatablk ::= parsdatastment | parsdatablk parsdatastment
;

parsdatastment ::= PARSDATA IDENT statement
;
stagestment ::= labelstmentlst ':' statement
;

labelstmentlst ::= labelstment | labelstmentlst ',' labelstment
;

condstment ::= IF expression THEN statement
| IF expression THEN statement
ELSE statement
| withpart IF expression THEN statement
| withpart IF expression THEN statement
ELSE statement
;

withpart ::= WITH IDENT
;

usecondstment ::= USE '(' IDENT '=' INTCONST ')' statement
;

partype ::= IDENT ':' '[' buttonpart ',' IDENT ',' INTCONST ']'
;

expression ::= buttonpart optbutid BUTTON IN windowpart
;

optbutid ::= empty | '(' INTCONST ')'
;

buttonpart ::= NEW | PRESSED | RELEASED
;

windowpart ::= ANY | IDENT
;

empty ::=
;

```

Appendix B. A SIDL Program

The following is a SIDL program for the font editor Vecfnt. The functionality of the editor has been described in Chapter 3. The first type section contains the presentational definitions and also the specification of the mutual exclusion tables. The subsequent blocks define the various screen regions that will be required. The final section contains the task specifications implementing the functionality of the program.

```
INTERACT vec;

TYPE
  MENU menu;
  NUMTASK 13;
  WINDOWS = (menu,grid,fontgrid,hand1,hand2,hand3);

  COLOUR bg (11,200,80,70);
  COLOUR dl (12,225,221,225);
  COLOUR offcol (13,0,120,0);
  COLOUR oncol (14,150,150,0);
  COLOUR offcol (13,0,120,0);
  COLOUR mtext (15,200,200,200);
  COLOUR ccol (16,255,255,255);
  COLOUR redink (17,200,0,0);
  COLOUR blueink (18,0,200,0);
  COLOUR greenink (18,0,200,0);

  MUTEXL= (backgrnd,foregrnd,compment) OF TYPE 3;
```



```
MUTEXL= (selchar,savechar) OF TYPE 1;  
MUTEXL= (line) OF TYPE 1;  
MUTEXL= (left,right,up,down,clear,clearfnt,vmirror,hmirror) OF TYPE 2;  
MUTEXL= (exit) OF TYPE 2;
```

```
ENDTYPE
```

```
WINDOW grid;  
DO  
  HT: 500;  
  WIDTH: 450;  
  XBOT: 100;  
  YBOT: 400;  
  GRID: NUMH=32;  
  GRID: NUMV=32  
END
```

```
WINDOW fontgrid;  
DO  
  HT: 331;  
  WIDTH: 539;  
  XBOT: 661;  
  YBOT: 110;  
  GRID: NUMH=8;  
  GRID: NUMV=16  
END
```

```
WINDOW scratch;  
DO  
  HT: 300;  
  WIDTH: 300;  
  XBOT: 200;  
  YBOT: 100  
END
```

```
WINDOW menu;  
DO  
  HT: 489;  
  WIDTH: 350;  
  XBOT: 750;  
  YBOT: 400;  
  GRID: NUMH=8;  
  GRID: NUMV=3  
END
```

```
WINDOW hand1;  
TYPE  
  hand1: OF MOTILE;  
ENDTYPE  
DO  
  HT: 20;  
  WIDTH: 20;  
  XBOT: 400;  
  YBOT: 400  
END
```

```
WINDOW hand2;
TYPE
  hand2: OF MOTILE;
ENDTYPE
DO
  HT: 20;
  WIDTH: 20;
  XBOT: 500;
  YBOT: 400
END
```

```
WINDOW hand3;
TYPE
  hand3: OF MOTILE;
ENDTYPE
DO
  HT: 20;
  WIDTH: 20;
  XBOT: 600;
  YBOT: 400
END
```

```
TASK backgrnd;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t);
    selectop(bground);
    esync(t)
  END;
END
```

```
TASK foregrnd;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t);
    selectop(fground);
    esync(t)
  END;
END
```

```
TASK compment;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t);
    selectop(cment);
    esync(t)
```

```

    END;
END

TASK selchar;
DO
  GATHER;
  IF NEW(2) BUTTON IN menu THEN run(t,start)
  ELSE
    IF NEW(2) BUTTON IN fontgrid THEN
      run(t,middle);
    PERFORM;
    START: DO bsync(t); suspend(t) END;
    MIDDLE,FINISH: DO findentry(fontcomm);
      selchar(); esync(t)
    END;
END

TASK savechar;
DO
  GATHER;
  IF NEW(2) BUTTON IN menu THEN run(t,start)
  ELSE
    IF NEW(2) BUTTON IN fontgrid THEN
      run(t,middle);
    PERFORM;
    START: DO bsync(t); suspend(t) END;
    MIDDLE,FINISH: DO findentry(fontcomm);
      savechar(); esync(t)
    END;
END

TASK line;
TYPE
  gridop : [NEW,grid,2] OF PARAMS;
ENDTYPE
DO
  GATHER;
  IF NEW(2) BUTTON IN menu THEN run(t,start)
  ELSE
    WITH gridop
      IF NEW(2) BUTTON IN grid THEN STUB
      ELSE
        DO
          $ { $;
            run(t,middle);
            USE (gridop = 2) run(t,finish);
          $ } $
        END;
      PERFORM;
      START: DO on(); $ gridop = -1; $; suspend(t) END;
      MIDDLE: DO
        $ switch (gridop) {
          $ case 0: bsync(t); savcod; gridop++; suspend(t); break; $;
          $ case 1: rubberit(xcoord,ycoord); suspend(t); break; $;
          $ case 2: suspend(t); break;
        }
      END;

```

```

        $ case default: break;
        $ } /* switch */
    END;
    FINISH: DO line(xcoord,ycoord); esync(t) END;
END

TASK left;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t); procleft(); esync(t)
  END;
END

TASK right;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t); right(); esync(t)
  END;
END

TASK up;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t); procup(); esync(t)
  END;
END

TASK down;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t); procdown(); esync(t)
  END;
END

TASK vmirror;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t); vermirror(); esync(t)
  END;
END

```

```

TASK hmirror;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t); mirror(); esync(t)
  END;
END

TASK clear;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t); cleargrid(); esync(t)
  END;
END

TASK clearfnt;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t); clrfnt(); esync(t)
  END;
END

TASK exit;
DO
  GATHER;
  run(t,start);
  PERFORM;
  DO
    bsync(t);
    finishit();
    esync(t)
  END;
END

TASK handmon;
TYPE
  hand1: OF MOTILE;
ENDTYPE
DO
  GATHER;
  IF NEW BUTTON IN any THEN run(t,start)
  ELSE
    IF PRESSED BUTTON IN any THEN run(t,middle)
  ELSE
    IF RELEASED BUTTON IN any THEN run(t,finish);
  PERFORM;
  START: DO bsync(t); suspend(t) END;

```

```
    MIDDLE:DO suspend(t) END;
    FINISH:DO esync(t) END;
END

TASK edit;
TYPE
  INIT= ACTIVE;
ENDTYPE
DO
  GATHER;
  IF NEW(2) BUTTON IN grid THEN
    run(t,start)
  ELSE
    IF PRESSED(2) BUTTON IN grid THEN
      run(t,middle);
    PERFORM;
    DO
      START: DO fillcell(xcoord,ycoord); suspend(t) END;
      MIDDLE,FINISH: DO fillcell(xcoord,ycoord); suspend(t) END
    END;
  END
END

END.
```

Appendix C. Publication

The following paper "*Graphical Interaction Management*" co-authored by P.J. Willis was presented at Eurographics(UK) in March 1987. It was published in Computer Graphics Forum,6, 119-124. 1987.

Graphical Interaction Management

Balbir S. Barn and Philip Willis †

Abstract

Graphical interfaces and interactive graphical programmes are awkward to write because of a lack of top-down structure. A methodology for constructing graphical programs will be described, together with a system that generates the basic interaction requirements for such applications.

1. Introduction

The advent of the bitmapped high resolution graphics workstation has led to a proliferation of graphics interfaces to existing application programs and, more importantly, has resulted in an increasing number of interactive graphical applications. Unfortunately, the attendant problems of scheduling mechanisms; interpreting mouse/tablet input; screen layout and interrupt handling associated with such applications have all been dealt with in a highly unsatisfactory manner, each programmer solving the problem in an individually stylised fashion.

Research in graphical interaction has concentrated on the study and development of user interface management systems (UIMS), ranging from theoretical studies¹ to practical systems such as TIGER², MENULAY³ and SYNGRAPH⁴. Automatic generation of user interfaces has relieved the programmer of some of the problems mentioned earlier, but still fails to ease the complexity of programming the basic building blocks of these applications. These building blocks include the fundamental interaction techniques such as picking, dragging and rubberbanding. The asynchronous, multiple-processes nature of graphical programs continues to be a major hurdle that consumes a programmer's effort.

Software development in most other computing fields follows some sort of methodology. Typically we

have JSD and HIPO charts for commercial applications, and data flow diagrams for systems/scientific applications. There is however, no equivalent technique for specifying the building blocks of graphical programs. Instead each application is re-designed, with implementations varying only slightly. The methods used are ad hoc leading to future problems of maintenance and portability.

We present a methodology for programming the interaction techniques used in graphical programs. In addition, we provide details of a tool we are currently developing which uses this methodology to generate the basic interaction code for such applications. We first describe the interaction model on which the methodology is based. Then we discuss some well-formed constructs which help the designer of interactive systems. Next we discuss our approach to implementation using a preprocessor. Finally we describe in detail the resulting language.

2. Interaction Model

In this section we describe the basics of our management system. We have conservatively assumed that no special kernel support is available. In particular we make no special assumptions about interrupts. The resulting software should thus be easier to port across systems supporting our chosen base language, C.

2.1. The Scheduler

We start with the premise that graphical interaction is essentially an input-event driven activity. Generally interaction offers a rich set of options, most of which will not be in use at any given time. A small number may be intensively used (e.g reading the tablet; updating the cursor position). Some will occasionally start other actions. Also, most actions require a rapid response to maintain fluid interaction.

These observations suggested that interaction can be handled by a fixed scheduler scan with associated tasks. A large pool of worker tasks will typically be used, but most tasks will lie dormant until needed and expire once used. Further, parameter gathering should be separated from invoking the worker task, to maintain flexibility of interaction and simplicity of workers. In outline, the Scheduler is:

This paper was presented at the EUROGRAPHICS (UK) Conference, Norwich, April 13-15 1987

† School of Mathematical Sciences
University of Bath
Claverton Down
Bath, Avon,
BA2 7AY
England


```

repeat
  for task: = 1 to numtasks do
    if active[task]
      begin
        if not runnable[task] then Try(task,gather);
          /* Collect parameters */
        if runnable[task] then Try(task,perform);
          /* Perform the action */
        end;
      until exit;

```

During a single scan of the task pool, the scheduler attempts to collect the necessary parameters for any task t which is active but not runnable. If all the parameters for task t have been collected the task is made runnable. In the same scan, if task t is runnable, the task is performed.

2.2. Tasks

We require a Task to have two components. The first is a parameter-gathering section and the second is the code which performs the actions required of the task. Correspondingly we need a mechanism for switching between these two components.

To implement this we use a procedure *Try* both to collect parameters and to execute the appropriate application-specific procedures. In outline *Try* is:-

```

procedure Try(integer: t,op);
begin
  case t of
    1: case op of
        gather: ...
        perform: ...
      end;
    2: ...
    n:
  end;
end;

```

Each case on t introduces a section of code specific to the parameter gathering needs of the task. When called as *Try(t,gather)*, any parameters for the task are identified, but no interpretation is made of them. For example, a task to draw a straight line requires two pairs of (x,y) coordinates which will later be interpreted as the vertices of the line.

Try(t,perform) runs task t to completion by binding the parameters to a call of the appropriate worker task.

2.3. Task Progression

Typically, most tasks are dormant until needed. Even when needed they usually pass through a parameter-

gathering phase before completing and once more becoming dormant. We envisage a system in which tasks progress from dormant (no need to do anything); to active (lacking enough parameters to run); to runnable (having a complete set of parameters and only awaiting scheduling). We provide a task progression mechanism to reflect this. Tasks correspondingly progress from frozen to thawed to runnable and we provide the appropriate control booleans with trusted primitives to manipulate them. For example, a task initiated from the menu is thawed, it remains in the thawed state until its parameter needs are met, when it subsequently becomes runnable. After being scheduled it might run to completion and then return to its former frozen state. Transition between states is accomplished by the primitives used to label the arcs in the diagram below.

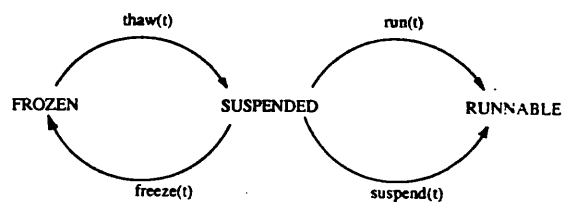


Figure 1. Transition of tasks between states

Not every task which reaches *runnable* state will run to completion when it is scheduled. It may simply have reached a stage where a certain parameter may be collected. Progression to a later stage would then depend on that parameter being collected and any associated actions being performed. Hence we can impose a degree of sequentiality within a specific task. Thus we have the *perform* section logically separated into three phases *start*, *middle* and *finish*. Typically this is useful because a new task has set-up actions which can be assigned to the start phase. The middle phase is used for the main part of the task and then the finish phase can be used to tidy up. To give an example, the outer phases can be used to change the cursor pattern back and forth to give feedback to the user, with the actual task being invoked in the middle phase.

Tasks do not have to freeze when completed. It is in the nature of some that they will gather one set of parameters, perform work and then repeatedly do the same thing. Such tasks can be suspended rather than frozen, as the diagram makes clear. In this way they continue to gather parameters as long as required.

This resemblance to finite state automata is fully documented in the case of user interface specification^{5,6}. This characteristic has been further used in systems where graphical programs are generated using interactive finite state machine editors^{6,7}.

Delineation into stages makes some housekeeping intricacies relatively easy to implement. If a task requires variables to be initialised prior to the task running, the necessary code can be inserted into the *start* stage. Also, menu window management problems such as menu highlighting synchronisation (ensuring the correct menu box is highlighted according to the current task) can be set up as *start* and *finish* actions.

2.4. Event Detection

We have already mentioned that we expect this style of interactive program to be event driven, so we also need a means of identifying such events. It is commonly the conjunction of a button press (or release, or holding down) with a specific area of screen/tablet which needs to be distinguished. We therefore adopt:

```
Event(butstate: button; region: area): Boolean:
```

as an enquiry function. The function *Event* is required to test the puck state to see if it corresponds to that named (eg. button 1 just pressed) and also the region (eg. command menu). *Event* has to be sufficiently general to allow for pop-up menus, overlapping menus etc. *Butstate* is defined to cover *new*, *pressed* and *released* where *new* defines a button press, *pressed* represents a button held down and finally *released* indicates a button has been released. It thus defines both level and transitory states, an important distinction. *Region* is used to identify the area on the screen/tablet where the button action occurred.

3. Well-Formed Constructs

Structured programming uses a modest number of well-formed constructs, where each construct displays some simple property. These constructs are satisfactory for sequential applications but they are of less value for interactive programs as they do not adequately reflect the behaviour of such applications. This section contains details of some of the more suitable control constructs we have identified up to the present time. These constructs are not sufficient in themselves for complex interaction requirements, but when used in conjunction with additional type declarations they are adequate. Some of the type declarations are described in section 4.4.

For each construct we will typically need to describe the code needed at both the *Gather* and *Perform* labels (see sections 2 and 5.1) for a task *n* entry.

3.1. Continual update

Commonly, a screen cursor is continually updated under program control to indicate the puck position. This is expressed as:

```
gather_n : Readpuck(x,y); Run(n);      { Puck monitor }
perform_n : MoveCursor(x,y); Suspend(n); { Cursor update }
```

3.2. Point and do

A simple form of interaction is the "point and do" action, an example is selecting "clear screen" by pointing to a labelled box and pressing a puck button. There is an immediate effect which proceeds, out of further control, to completion. Such a task will be permanently active and will run whenever a particular puck button is pressed within a certain area on the screen. There are also degenerate cases corresponding to using a puck button for a dedicated action (independent of screen position). All such permanent tasks can be coded in the form such as:

```
gather_n : if Event(button,viewport) then Run(n);
perform_n : execute(n);
```

3.3. Event recogniser

In principle menu_selection could be implemented as a number of point-and-do tasks. In practice this can be cumbersome for all but very short menus. A conventional top-down approach to manage this interaction has been adopted. At the top level, it is sufficient to identify that a relevant event has occurred, namely that a new button press has just happened in the menu. We thus get:

```
gather_n : if (Event(new,menu)) then Run(n); {Menu monitor}
perform_n : Thaw(Comm); Suspend(n); {Action the command}
```

The monitor component is already familiar. The action component uses procedure *Comm* to decode the puck position, returning the the task number needed for that command. This task is then enabled by *Thaw*. The event recogniser task takes no further action until the next new press of a button in this menu. It has, however caused a non-permanent task to spring to life and this will have its own *Gather/Perform* entry.

This particular construct is important as it can be used as the basis for a number of interactions. Thus we can have an event recogniser task for detecting events in a number of different windows.

4. The SIDL Language

In the implementation, descriptions of the interaction model are encoded in the abstract high-level specification language SIDL (Simple Interaction Design Language). The salient features of typical graphical interaction methods have been incorporated as structures of this language. Some of the features have been provided as a result of studies of existing window

management systems. Typical component parts of SIDL are described below.

4.1. Tasks and Windows

An application is considered as a set of tasks, with the user generating events via the puck in specified regions on the screen/tablet (window). We can observe that our language must provide means of specifying the interactions of tasks in terms of events in windows.

A SIDL program is composed of program blocks, each of which is either a window or a task definition. Both definitions vary in complexity according to the interaction requirements. Window definitions typically specify size, position on the screen, borders and other external features. There are various types of windows but, for the purpose of this discussion, we can simply say a window is a uniquely identifiable area on the screen where an event can occur. The command menu is a special case of window which is pre-defined in the skeletal program as it is integral to the applications we are concerned with.

Task definitions are more complex. Their SIDL structure closely follows that of the interaction model. This similarity aids code generation and the structure is relatively straightforward without being ambiguous. A task definition is broadly divided into three sections:- Type definitions, Gather and Perform. Type definitions will be ignored for the moment.

The Gather section is primarily concerned with the collection of parameters (events). The Perform section executes application-specific code on the collection of a particular event. Together these sections display the control constructs described earlier. A number of tasks are predefined and are largely concerned with either low-level details or monitoring other tasks.

Task definitions are directly translated to C code and form case entries in the main scheduling procedure *Try* as described in the IM (see section 2).

4.2. Task Synchronisation

All good interfaces must allow unrestricted asynchronous activity from the user. This freedom, however, makes the menu management a less than trivial problem. It is easy to envisage the case where the user makes ad hoc selections of menu buttons thereby initiating various functions and so losing any idea of what is happening. Where initiated tasks require additional parameters (such as selections from another menu), the problem is even more acute. The user must be able to select another task while the current task is waiting for a parameter, and still be able to return to the earlier task with the current state of that task intact. Furthermore, this facility must be provided in a manner that is user friendly.

These objectives have to a large extent been achieved by enforcing a sub-structure over tasks initiated from the menu. A study of two menu driven applications developed at Bath indicates that tasks initiated from menus can be divided into three categories: *tasks which run to completion when selected; tasks which require one or more parameters before running to completion; and finally tasks which when selected set parameters to be used by other tasks.* Furthermore, tasks can be assigned to *Mutual Exclusion Groups*, that at most one task in a group can be active at any time. We can infer three rules to control these groups.

Rule 1: Each group can have an active task so suppose task A is in a different group to active task B. If A now becomes active, then the state of B has to be preserved. Task B can therefore be reinstated later.

Rule 2: Suppose tasks B and C are in the same group. If B is active and C becomes active then B has to be killed.

Rule 3: Suppose B has earlier been preserved and that C is in the same group. If C now becomes active, then the preserved state of B has to be discarded.

The groups are represented as rows in a table and primitives are provided which carry out the rules described above.

4.3. Window Management

Typically, windows provide an environment for task interaction. Many applications however, also include interactions where the window itself is an active member. The interaction needs of such windows vary and we have broadly classified the following requirements:-

1. Picking and dragging of objects/windows
2. Stretching Windows
3. Invisible Windows
4. Overlapping Windows
5. Static or Movable Windows

Case 1 is an example of a common interaction; we have extended the definition of windows to include the specification of pixel objects. Cases 2, 3 and 4 are typical of facilities found in window management systems. Although we are not directly concerned with the development of such applications, their interaction is of interest, and providing some of their facilities within SIDL is justifiable.

For 5, the following window types are currently implemented:- *static*, *motile* and *variable*. *Static* windows remain fixed in their originally specified position.

Motile windows are those which can be selected and moved to some new location. They are commonly used to represent screen objects. For example: a paddle in a simple ball/paddle game. *Variable* windows can be moved to a new location and have their size altered. Windows default to *static* unless otherwise indicated.

Most interactions are based on the detection of events in a region. The region varies with the type of window and also with time. For example, to alter the size of a window requires the detection of an event in one of four regions, where each region specifies a corner of the window to be stretched.

Using control constructs similar to that used for menu selection (see section 3.3), pre-defined window monitor tasks (one for each window type) are used to identify a button press in a window. As there may be more than one window of the same type, the appropriate window is identified and the task containing the interaction code for operations on that window is initiated. This task will have been specified by the user.

4.4. Parameter Collection

The collection of parameters (detection and decoding of button activity) is a problem encountered in most graphical interaction applications.

Typically, task definitions require the user to specify the events necessary for a task to run to completion. Where these events are unique, there is no obvious difficulty. However, serious problems arise where two or more identical events need to be detected. Events are considered identical if both the button type and the region are the same.

In the painting system at Bath^{8,6}, there is a command to paint a rectangular region on the screen with a range of colours. The user specifies the rectangle with two diagonally opposite vertices in the drawing region. The two colours are specified by indicating two entries in the palette region. The elements in each pair are identical. To avoid constraining the user, collection of the four parameters can be done in any order. The action to be taken when a particular parameter is collected therefore depends on its sequence number within a particular region. For example, the first event in the drawing region marks the first corner of the rectangle. This results in a rectangle being rubberbanded from that marked position. The receipt of a second event in the same region results in the second corner of the rectangle being marked. The user selects the range of colours by two events in the palette region. Provided that a count is kept for each region the colour and rectangle parameter collection can be interleaved. A critical observation is that each event, although not unique, may have actions associated with it which depend on the count. We can decide when to call the

worker task for this command by keeping a total count of the number of parameters collected.

5. The GRIP Preprocessor

In this section we describe the basic structure of the preprocessor and its use of UNIX† software tools.

The basic scenario of the preprocessor is as follows. The user describes the interaction model for an application in a specification language. This description is then used to generate a program containing the interactions in our target high-level language C. The generated program will contain facilities to include application-specific procedures. This approach is orthodox and displays many of the characteristics of current UIMS.

In the implementation, the abstract high-level specification language SIDL is used to describe the interaction needs of the application. The GRaphical Interaction Preprocessor (GRIP) generates the C code contained within a skeletal program which also holds the necessary information to set up the display.

5.1. System Outline

The development of GRIP and SIDL is UNIX dependent, because of our desire to utilize the rich library of software tools available. Yacc is a program that generates a parser from a grammatical description of a language^{9,6}. The class of specifications accepted is very general: LALR(1) grammars with disambiguating rules. We use it both to parse a SIDL program and to generate code. Cb is a program that pretty prints a C program. We use it to beautify the code generated from the preprocessor.

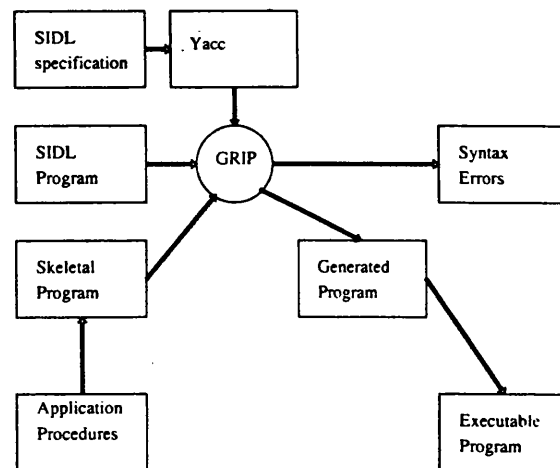


Figure 2. System Structure

† UNIX is a trademark of Bell Laboratories.

The bulk of the preprocessor is contained within the file used to drive yacc. The file contains both the lexical analyser module and the code generator module: experimental requirements have dictated this structure. Once a satisfactory language design has been implemented, the production of a special purpose syntax analyser and code generator will remove the dependence on yacc. Figure 2 shows the system.

6. Concluding Remarks

In this paper, we have discussed the basis of the design of our hybrid UIMS. We have presented a methodology which we think aids the design of multi-process event driven graphical applications. We have also outlined the practical aspects of some of the integral components of the language which we use to represent our model. We are still currently in the development phase of GRIP: our language is still undergoing significant design changes and we are trying to identify more well-defined control constructs.

We envisage fundamental design changes in the preprocessor, as at the moment error reporting is restricted to syntactical errors in SIDL source programs: there is only minimal error recovery (performed by "yacc"). The semantics of interaction specification need to be statically analysed in the preprocessing stage and errors reported. However, given that no language definition can be complete, postfix semantic error reporting seems viable, with interaction errors being reported with reference to both the SIDL program and the generated C program.

Even at this prototype stage, the system has shown that it is possible to specify complex interactions and automatically generate C code that is both easily maintainable and extensible, hence reducing the development cost of such applications. Additionally, rapid prototyping facilities will produce better quality user interfaces.

7. Acknowledgments

The work described here was supported by a grant from the SERC. We would also like to thank Geoff Watters for his many useful comments on this work.

References

1. M Green, "A Methodology For The Specification Of Graphical Interaction," *ACM SIGGRAPH Computer Graphics* 15(3) (1981).
2. D Kasik, "A User Interface Management System," *ACM SIGGRAPH Computer Graphics* 16(3) (1982).
3. W Buxton, M R Lamb, D Sherman, and K C Smith, "Towards A Comprehensive User Interface Management System," *ACM SIGGRAPH Computer Graphics* 17(3) (1983).
4. D R Olsen and E P Dempsey, "SYNGRAPH A Graphical User Interface Generator," *ACM SIGGRAPH Computer Graphics* 17(3) (1983).
5. D L Parnas, "On The Use Of Transition Diagrams In The Design Of A User Interface For An Interactive Computer system," *Proc. 24th Natl. ACM Conf* (1969).
6. R J K Jacob, "Using Formal Specifications in the Design of a Human Computer Interface," *Comm. ACM* 26(4) (1983).
7. R J K Jacob, "A State Transition Diagram Language For Visual Programming," *IEEE Computer* (August 1985).
8. P J Willis, "A Paint Program For The Graphic Arts," *Proc. of EuroGraphics* (1984).
9. S C Johnson, "YACC: Yet Another Compiler-Compiler," in *UNIX Programmers Manual*, Bell Labs (1978).