USER INTERFACE MANAGEMENT SYSTEMS: A SURVEY AND A PROPOSED DESIGN

by

PHILIP D. GRAY

A Thesis Submitted for the Degree
of
Master of Science
in the
Department of Computing Science
at
The University of Glasgow

November 1986

ACKNOWLEDGEMENTS

My most sincere thanks must go first and foremost to my supervisor, Dr. Alistair Kilgour. Our conversations kindled my interest in human-computer interaction and his continual encouragement and advice have been crucial in my continuing with this work. I also wish to express my gratitude to St. Andrew's College of Education for its support for my research over the past three years. In addition, thanks must go to those staff and students in the Computing Science Department at the University of Glasgow who have provided information and suggestions, or even just listened sympathetically, often helping even when they weren't aware of it. Particular thanks in this regard go to Zdrav Podolski, Rob Sutherland, Nick Nei and Jack Campin. Finally, my gratitude and affection go to my wife, Beverly, for her patience and assistance during what must have seemed an endless task.

CONTENTS

Figures

ABSTRACT

The growth of interactive computing has resulted in increasingly more complex styles of interaction between user and computer. To facilitate the creation of highly interactive systems, the concept of the User Interface Management System (UIMS) has been developed. Following the definition of the term 'UIMS' and a consideration of the putative advantages of the UIMS approach, a number of User Interface Management Systems are examined. This examination focuses in turn on the run-time execution system, the specification notation and the design environment, with a view to establishing the features which an "ideal" UIMS should possess. On the basis of this examination, a proposal for the design of a new UIMS is presented, and progress reported towards the implementation of a prototype based on this design.

Chapter One

User Interface Management Systems - Definition & Rationale

1.1 User-Computer Dialogue

In order to be able to use computer systems to solve problems, people must converse with them. That is, they must issue commands which determine what tasks the system is to carry out and provide the data needed for the execution of the commands. And users must receive and understand the computer requests for information, its confirmation of messages received, help and error messages, as well as the data generated by the execution of commands. Where the communication occurs with only short intervals between human and computer system responses to one another, the system may be said to be (highly) interactive and the user and computer may be said to be engaged in a dialogue.

The purpose of user-computer dialogue is to carry out user-determined tasks. Success in carrying out the tasks depends in part on the actual applications to which the computer system can be put (can it do what the user wants?), but it also depends largely upon the user's ability to access the applications via dialogue (can the user make it do what he wants?). This ability is not only affected by the user's understanding of the appropriate ways to communicate with the system, but by all those aspects of the system of which the user is aware during the dialogue -- the **user interface**. As Moran states, "the user interface of a system consists of those aspects of the system that the user comes in contact with -- physically, perceptually and conceptually." (Moran, 1981, p. 4) For example, the use of a keyboard for command and data entry may cause problems for a user who cannot type well, a cluttered screen design may cause poor comprehension of displayed data, and a complex system of command abbreviations may result in a high level of errors in command entry due to overloading of the user's short-term memory.

In the past, when highly-trained computer personnel were the primary users of computer systems, inadequacies in user interfaces tended to be overcome, and hence hidden, by the expertise of the operators. However, the growth of computer applications has broadened the population of computer users such that many, if not most, do not have a technical background

nor are they able or willing to invest substantial time and effort in training. User interfaces for such users need to be easy to learn, equipped with comprehensible and easily accessible help, and capable of efficient and accurate use.

When user needs and capacities are treated as central to interactive system design, it can be seen that a multi-disciplinary collection of factors are relevant to the design of the user interface. These include general cognitive and psycho-motor aspects of users, as well as the particular design constraints of the attitudes, skill level and training requirements of the target user population (Thomas, 1983, pp. 9-10).

To these user-centred factors must be added the demands of the application. Highly interactive graphics systems (e.g., in engineering design, TV and cinema animation, CAL, video games) and systems which use graphical interaction as a means of dialogue (e.g., the Macintosh interface, spreadsheets, electronic publishing) require graphical modes of communication from and to the user. A repertoire of interaction techniques, e.g., pull-down menus, function keys, rubber-banding of lines, is available to the interface designer to apply as appropriate, along with complementary physical devices such as high-resolution raster displays, touch screens, graphics tablets, mice and systems support for concurrent asynchronous processes, multiple screen windows, speech input and synthesis.

Finally, the possible interactions must be sequenced in ways which make best use of the interaction techniques available, given the application(s) involved, the limitations of hardware and software support, and the needs and capacities of the user population. Figure 1.2 displays these factors influencing user interface design.

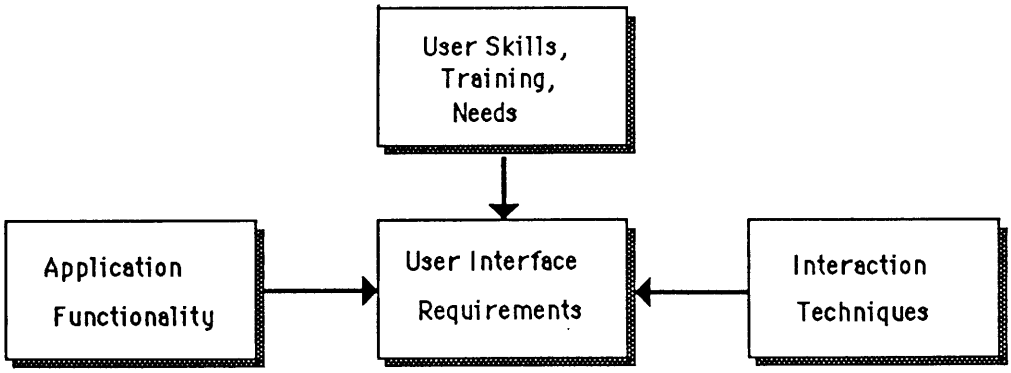

Figure 1.1 Factors affecting user interface design

Bringing all these factors together to produce a successful user interface suggests the need for:

i. A multi-disciplinary effort.

This may involve only a single individual, a user-interface designer with a wide range of knowledge. Indeed, it could be the same person who designs the application(s). However, in complex systems, it is more likely to include a number of individuals: human factors experts, dialogue designers, graphics designers, interface evaluators as well as application designers and programmers.

ii. A design methodology which enables practitioners from (i) to influence the design of the user interface.

Beyond intuition, how can this variety of factors be assembled to produce a user interface? Several methods have been proposed to support the systematic construction of user interfaces, based on software engineering principles (Ehrich, 1982a) and methods of formal language design (Moran, 1981; Wasserman, 1984). Unfortunately, there is as yet no generally agreed set of criteria by which to judge the resultant design. As Allen states:

"Practitioners of the art of constructing human interfaces have proposed quasi-cognitive criteria for the effective design of systems. Factors such as simplicity, naturalness, ease of use, consistency, feedback, and individualization are frequently and perhaps rightfully mentioned. However, there are potential problems with these informal prescriptions: the details for applying them are rarely specified and there is little independent validation of the assertions." (Allen, 1982, pp. 1-2)

iii. Tools to assist in the production of software which implements the designs from (ii).

Analogous to the connection between software engineering methodology and sets of tools to provide programming environments, the existence of a design methodology for user interfaces, however incomplete or inadequate, would appear to call for a user interface design environment. A User Interface Management System provides just such a set of tools.

1.2 User Interface Independence

In June 1982 a workshop on Graphical Input Interaction Techniques in Seattle described a software tool called a User Interface Management System, or UIMS (Thomas, 1983). Roughly contemporaneously, examples of systems called User Interface Management Systems appeared (Kasik, 1982; Buxton et al, 1983b). Although some similar systems already existed, e.g., SYNICS (Edmonds, 1981), and some features of UIMSs had been in use for a considerable time (Newman, 1968), the introduction of the term and the concept it represented helped to focus thinking about user interface structure and design.

However, before looking in more detail at the notion of a UIMS, the user interface itself must be examined more closely. As stated in Sec. 1.1, any aspect of the system which affects the user's behaviour in using it may be considered part of the user interface. This includes the physical design of I/O devices (displays, keyoards, mice) and the overall work environment, both subject to ergonomic factors (Olsen, 1984b). At the other extreme, features of the application code can cause user frustration and disorientation due to his having to remember the current state of the dialogue longer than is comfortable. Good interface design can overcome such difficulties, by the use of "wait" messages and the display of the dialogue state, but UI design cannot ignore the effects of application design and implementation on the quality of dialogues.

Nevertheless, it is useful to isolate that part of the software which supports communicative transactions between the user and system, i.e., the dialogue, leaving aside environmental, physical and application factors. Such transactions include all input accepted by the system from the user and all output presented to the user by the system. Unfortunately, this component of the system is often also referred to as the user interface. To avoid confusion, the broader definition of the term will hereafter be termed the 'global user interface' while the terms 'system user interface' or just 'user interface' will be used for the more restrictive definition. The relationship among these components is illustrated by figure 1.2, where the solid lines indicate communicative transactions and the dotted lines indicate behavioural influence on the user.

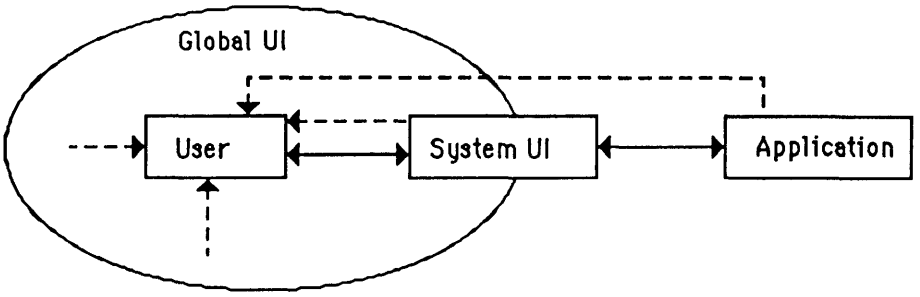

Figure 1.2 System UI Controlling all I/O

An alternative, and even more restrictive view of the system user interface limits it to the software supporting the acceptance of input from the user, along with that output required to support input actions, such as prompts and feedback generated without reference to the application. Excluded is output generated by the computational activity of the application. The term 'user interface' will also be used to refer to this more restricted system component; where the difference between the two is significant, context will usually be sufficient to distinguish between them. Figure 1.3 shows this alternative version of the UI.

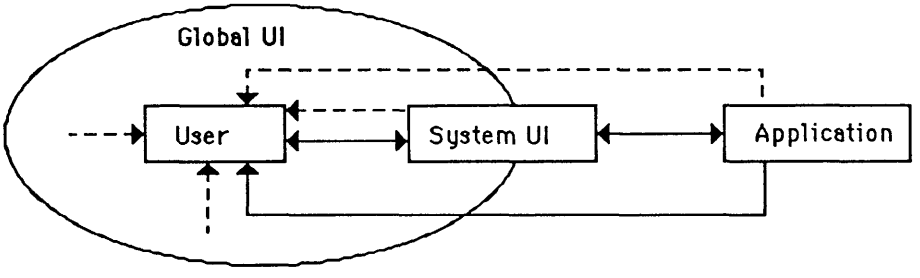

Figure 1.3 System UI with Application Generated Output

So far, the characterisation of the UI is only a logical separation, referring to the functions of components of the software. The implementation of the UI, so construed, might well be embedded in and distributed throughout the application code. However, a functional separation of the UI leads naturally to a physical separation from the application. Green states that "good software engineering practice suggests that the user interface should be a separate program module." (Green, 1985b, p. 205) Similarly , Edmonds declares:

"...the software component of an interface should be treated as a separate module within the computer system as a whole and not simply embedded at a range of points throughout it." (Edmonds, 1981, p. 389)

Edmonds goes on to suggest that the provision of an independent system as an interface between user and computer system is not new:

"It is common to separate the interface between the user and the main system from the main system. Perhaps the most well-known example of such a separated unit is a compiler , which in effect interfaces the user to the machine. The role of the interface is to take from the user strings which are meaningful to him and to transform them into strings which are meaningful to the machine or main system." (Edmonds, 1981, pp. 391-2)

A compiler may be construed as an interface between a user (a programmer) and a machine and there is a useful analogy between a compiler's translation function and a user interface "translating" sequences of user-generated physical events into commands and data meaningful to applications. Nevertheless, the analogy may be misleading. A compiler, although an interface of sorts, can also be described as an application which takes a program description in a high-level language and produces code in a machine-executable form. This may be termed a user task. Its status as an application is further implied by the fact that a compiler may itself have a user interface module mediating interaction between it and its users.

Unlike a compiler, a UI does not carry out user tasks itself; it translates user actions into commands which are then passed to other (application) modules or processes to be executed. In principle, the jobs done by an independent UI module could be done by user-interface routines built into applications. Thus, the separation of the interface is not directly related to increasing the functionality of a system, except in terms of improving the accessibility of system functions.

The principal motivation for separating the UI from the applications is related to improving their design and implementation both in terms of the efficiency of the design process and the quality

of the resulting products (Olson et al, 1985b, p. 191). A number of benefits accrue once the UI is viewed independently. Most importantly, perhaps, the UI design can be undertaken independently of application design. As Edmonds argues:

"In a batch processing environment it is often thought desirable to determine the inputs and outputs of a program first and then design it so as to achieve those ends. In an interactive system the equivalent, but much more complex, procedure would be to design the interface first. The specification of background tasks would then provide first level description of the background processor [i.e., the applications]. It is suggested that the isolation of the dynamics processor [the UI] leads to a clarification of the design." (Edmonds, 1982, p. 234)

A detailed examination of this and other advantages of user interface independence can be found in Section 1.4.

Where the UI is treated as a separate module or modules, the specification of the interface can be integrated into an effective design and implementation environment not possible where the UI implementation is embedded in the application. In particular, tools can be envisaged which will assist:

i. the designer in the production of a UI specification,

ii. the implementor in creating the UI which satisfies the specification,

iii. the evaluator in collecting data for assessing the performance of the UI. (Buxton et al, 1983b, p. 35)

In fact, once the UI is conceived as modularised, it becomes possible to generate the actual UI from the high-level description of the interface, either via a compilation process or by providing a run-time execution environment which is driven by the UI specification. Thus, arising out of the logical and physical separation of the UI is the opportunity to provide a set of software tools which were noted in Section 1.1 as desirable for UI production. This set of software tools forms the basis for the concept of a User Interface Management System.

1.3 UIMS: A Definition

Unfortunately, there is in the relevant literature no single clear and adequate definition of the term 'User Interface Management System'. Also, similar systems have gone under the names Dialogue Management System (Ehrich, 1982b) and Abstract Interaction Handler (Feldman et al, 1982a), among others. What, then, is a UIMS?

Thomas states that:

"The term UIMS has been coined to represent a set of software tools for the construction and control of the interaction dialogue between the user and the computation resource." (Thomas, 1985, p. 81)

This description is accurate when applied to all current UIMS and seems to capture at least part of what UIMS designers have in mind. However, it would appear that the facilities for supporting graphical input in, say, GKS, or even the I/O support within a language like Basic would qualify by this definition as user interface management systems. Although a UIMS might well use the input and graphical output facilities of GKS, the UIMS itself needs to operate at a higher level of abstraction of the dialogue. GKS provides for the definition of logical input devices bound to abstract workstations (Hopgood et al, 1983). This is already a level of abstraction beyond physical devices. However, only six types of logical device are available: locator, pick, choice, valuator, string and stoke (Hopgood et al, 1983, p. 68) The legal sequences of input from these devices and any transformation or validation of input for an application of input for an application are not the responsibility of GKS, yet they are surely issues which belong to a user interface rather than to the application. Futhermore, GKS only provides facilities for low-level echoing as feedback; again feedback related to the syntactic and semantic aspects of the interaction are not supported and would have to be constructed using either ad hoc methods or other tools.

While Thomas' definition is too broad, at least one other which has been offered is too narrow. Sibert et al provide what they call a "working definition" of a UIMS (Sibert et al, 1985, p. 183). It is particularly interesting in that this is one of the only explicit definitions of a UIMS in the literature.

Their definition actually consists of five requirements. They are:

i. a UIMS "represents a higher level of abstraction than, for example, a graphics subroutine package" (Sibert et al, 1985, p. 183),

ii. a UIMS manages all the input and output between user and system,

iii. the UIMS should allow changes to the interface without changing application code,

iv. the UIMS should make possible alterations to the UI without programming,

v. the UIMS is an integrated software tool; that is, dialogue specification tools and the run-time environment are integrated. (Sibert et al, 1985, p. 183)

Requirement (i) is unobjectionable and simply encapsulates what was said about GKS being viewed as a UIMS. Requirement (iii) is based on the notion of separating the UI from the application. Together, they imply that the UIMS stands between graphics I/O packages on the one hand and the applications on the other. This view is supported by the GIIT workshop description of a UIMS as "a module which is separate from the application program and the graphics output package used." (Thomas, 1983, p.7)

Requirements (ii) and (iv) are surely desirable features of a UIMS. Systems incorporating (ii) enable the UI designer's view of the interface to match that of the ultimate user. If some output is generated by the application and passed by the application directly to the user, then the interface and consequently the quality of the dialogue can be affected by the design of or changes to application code, which should not be the concern of UI designers or implementors. "Semantic output [generated by the application, is]...inseparable, from the user's point of view, from the interface." (Sibert et al, 1985, p. 183) However, no currently implemented UIMS has full control over all output and many do not handle application output at all.

Similarly, requirement (iv) may be construed as a desirable objective, particularly since those with an interest and expertise in the user interface (dialogue designers and even users themselves) are often not skilled programmers. The degree to which actual UIMS allow adaptation without the use of programming is variable and many UIMSs do not possess this feature to any degree.

The integration of requirement (v) implies some element of UI construction around which all the UIMS components are organised. In fact there is such an element, viz., the UI, or dialogue,

specification which serves as the object which integrates UI design and run-time execution tools. Kasik states that:

"the user interface [management system] contains its own programming language to define interactive dialogue sequences distinct from the application and a run-time module that specifically handles an end-user's physical interaction." (Kasik, 1982, p.99)

Green describes the relationship more tightly, stating that a UIMS is "a program that automatically constructs a user interface given a description of it." (Green, 1985a, p. 89)

The UI so "constructed" may be viewed as the process supporting dialogue which is brought into being by a run-time module interpreting the UI specification. There need not, although there may, be a separate compilation phase which transforms the UI specification into a UI executable image. The interpretative version of a UIMS is described by Hayes:

"The interface system finds the details of the interface required by each application in an external, declarative database, called an interface definition, - one definition for each application. These interface definitions are executable in the sense that they can be interpreted by the interface system to produce the required interface behaviour for each different application." (Hayes, 1985, p. 162)

The University of Toronto UIMS, which includes the MENULAY and MAKEMENU programs, takes the compilation approach as described thus:

"Specifications made using the package [MENULAY] are converted into the C programming language and compiled through the use of companion program MAKEMENU. The resulting code can be linked with application specific routines." (Buxton et al, 1983b, p. 37)

The GIIT workshop appears to remain neutral on the compilation/interpretation issue, simply saying that the UIMS "accepts as input a dialogue specification, describing the detailed structure of the interaction." (Thomas, 1983, p. 16) In a similar vein, any definition of a UIMS while

accepting that the dialogue description should serve to generate the actual UI, must not specify the method by which this is carried out.

Returning to Sibert et al's five requirements, (ii) and (iv) are perhaps better seen as desirable goals for UIMS research. Requirement (i), (iii) and (v), however, can provide the basis for a workable definition:

A UIMS is a set of software tools for the design, implementation and evaluation of user interfaces which:

i. supports the production of a description of the user interface to application(s),

ii. enables the generation of the UI based on the description of (i), and

iii. optionally, provides tools for the prototyping and evaluation of the resultant UI,

where the user interface is understood to be a software module which stands between the application(s) and either the physical devices or graphics/windowing software, and which controls all interactive user input and exercises control over some element of system-generated output.

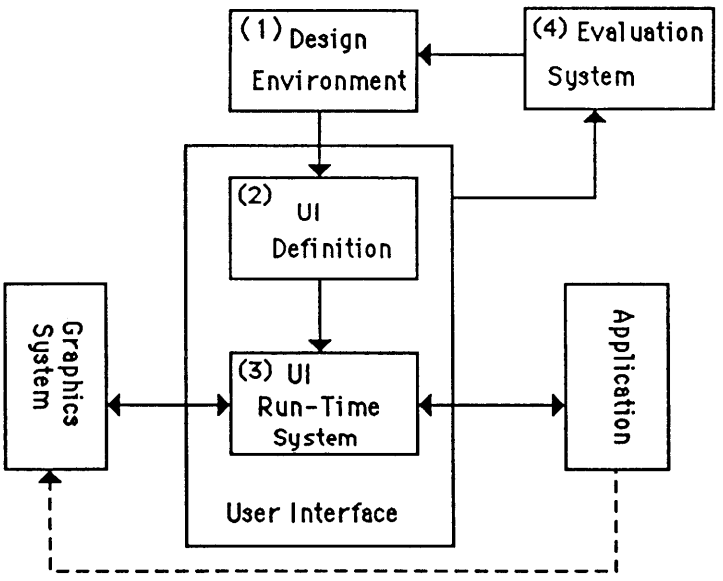

Figure 1.4 A UIMS

Figure 1.4 gives a diagrammatic representation of the UIMS as defined. It consists of four parts:

1. a design environment which produces

2. an interface definition, which defines the UI produced via the design environment and which is executed by

3. a run-time system , and

4. optionally, an evaluation environment which logs and analyses data from the performance of the UI for further refinements of the interface specification.

1.4 Advantages of a UIMS

A number of advantages have been claimed to arise from the use of a UIMS for interactive system construction. Although these have not been tested or validated formally , a review of the claimed advantages can serve as a starting point for the examination of actual UIMSs.

The advantages may be categorized as related to:

i. improving the design process,

ii. improving the efficiency of the implementation, and

iii. improving the quality of the resultant interactive systems.

It might be argued that all the advantages are, in the last analysis, related to improving the product. However, the most powerful arguments in favour of the UIMS approach have to do with making design and implementation more cost-effective or better able to incorporate the growing knowledge about human computer interaction. These arguments do not suggest that the UIMS provides user-visible features itself. Such features which would be difficult, indeed practically impossible, to provide without a UIMS will be covered under category (iii).

1.4.1 Design Advantages

One of the most compelling reasons for using a UIMS is the lack of a firm theoretical foundation for user interface design. Quite simply, not enough is known about the nature of human

computer interaction for a UI or dialogue designer to produce a specification based solely on user task requirements, functional specification of the applications, and the I/O devices/ graphics package specifications. The consequence is that UI design must be viewed differently from the conventional model of software design.

"We must accept . . . the inevitable intertwining of specification, design, and implementation....Design then becomes an iterative process, each iteration consisting of three phases: design, implementation and evaluation....The motivation to develop improved tools can therefore be seen as a desire to increase the number of iterations that we can afford to pass through in the loop. The hoped-for consequence will be an improvement in the quality of the user interface which we produce." (Buxton et al, 1983b, p.35)

Of course, traditional software design consists of design, implementation and evaluation phases and the need for looping through this process is implicitly acknowledged by the existence of an evaluation, or testing, phase, the results of which must have an influence on subsequent changes to the implementation and design. In interactive system design, though, the lack of firm design principles necessitates a greater number of iterations of this loop in order to allow experimentation with the actual performance of different design solutions before a final solution is adopted.

Where "dialogue is woven into the computer software fabric...software vendors and designers simply become committed to inferior interfaces because they are too complex and expensive to reprogram." (Ehrich, 1982a, p. 50) A UIMS overcomes this problem through making both UI specification and implementation independent of application code. Changes to the specification are reasonably inexpensive given the automatic generation of the UI from its description.

The production of prototype interfaces for the purposes of experimentation is also desirable (Thomas, 1983, p.21). Evaluation can concentrate on the effects of certain features with other features removed. Testing of the UI with real users can begin early in the design stage and can be supported throughout development (Yuntan et al, 1985, p. 247). Prototyping is facilitated by the UIMS approach since application actions of interactive sequences can be substituted by

non-computational stubs. This also means UI and application development can take place in parallel.

Olsen et al (1985b, p. 34) present an example of interface design in which the facility for fast prototyping is an essential element in the design process. Consider the user task of setting a spreadsheet cell to a numerical value, say cell C5 to the value 3.5. Olsen et al list six possible techniques (not an exhaustive list) by which this action can be carried out:

(1) typing 'set C5 to 3.5',

(2) moving a screen cursor over cell C5 with cursor keys and then typing '3.5',

(3) moving the screen cursor over cell C5 with a mouse and typing '3.5',

(4) using a mouse to select cell C5 and setting its value by setting a screen potentiometer to 3.5 by sliding the potentiometer bar with the mouse,

(5) selecting cell C5 with a joystick which has a rotary potentiometer attached to the lever. Moving the lever allows selection of the cell and rotating the lever changes its value once selected, and

(6) rotating a 3-D trackball to select cell C5 and pressing the ball down with the palm to set the value to 3.5.

Some problems with these alternative techniques can be anticipated without user trials. For instance, alternative (3) requires a hand shift from mouse to keyboard. Alternative (5) may cause loss of cell selection when the lever is rotated to set the value. However, only by carrying out tests with users and analysing the results can the severity of these problems be determined, unanticipated problems be discovered and the best technique for a given user population found. Where fast and easy prototyping is available, preferably with monitoring of input events built into the run-time environment, this sort of experimentation will be naturally encouraged. Similar problems of choice can occur at higher levels of interaction abstraction, where, for example, choices must be made about how many menu options should be offered at one time, which options should be grouped together, or which of several alternative command language syntaxes is preferable. Using a UIMS, these alternatives can be offered via isolating changes in the UI specification, followed by prototypes being produced automatically by the UI generator.

It has already been argued that the concept of UI independence which underlies the UIMS

approach leads to improvement in system design by isolating interface design issues from those of the application (see Section 1.2). Separation of interface design and the demands of the testing of prototypes leads to the responsibility for UI design being placed in the hands of individuals with special skills. Using a UIMS, with its support for prototyping and iterative design, "individuals can be trained as human factors specialists (analogous to those in traditional ergonomics or industrial engineering) who need not be familiar with details of applications systems." (Feldman et al, 1982b, p. 4) These specialists also need not be concerned with low-level details of the implementation of particular interaction techniques which reside in the graphics/windowing package or in the UIMS. (Ehrich, 1982a, p. 50)

Such UI designers may well not be skilled programmers. The language, or specification notation, in which the UI specification is described is itself part of the UI design environment. High-level abstractions built into the specification system will have a major bearing on the ability to express the structure and functional content of the UI. Properly constructed, this system can allow the application of programming aids, such as graphical specification of dialogue sequences, which may be of considerable assistance to UI designers. Libraries of interactive sequences (Tanner & Buxton, 1985, pp. 70-71) and interface frameworks or templates (Hayes, 1985, p. 164) can be provided, as well as means of adding new ones. Ultimately, the expressiveness and ease of use of the specification system are the most likely elements in the design environment to affect the efficiency of design.

1.4.2 **Implementation Advantages**

UI separation leads to advantages for application programmers. The applications programmer is freed "from low-level details [of the UI] so as to be able to concentrate on higher application-specific aspects of the user interface." (Buxton et al, 1983b, p. 35) For example, an application programmer might only know that requests to delete a segment from a picture (along with the segment id) may be received from the UI, without concerning himself with how the command was generated (e.g., menu selection, command entry), how prompts are generated, and how the segment is identified. A useful, and frequently used, analogy is with data base management systems (Tanner & Buxton, 1985; Buxton et al, 1983b; Kasik, 1982; Thomas,

1983). As Feldman et al state:

"Such modularity {the separation of the UI from the application} is analogous to the DBMS approach in handling the "back end" of a system. There an application writer can request information to be drawn from a data base with little concern for its organization or access techniques.... In the domain of interactive systems, the abstraction is done at the "front end" as well, where an application writer can request information from and display results to an interactive user without being concerned with the detailed interaction syntax or the device-level access methods." (Feldman et al, 1982b, pp. 4-5)

It also follows that application code will be more portable since detailed interaction code is removed. The UI can be altered to maximize the hardware and software support available in different installations without altering the application. Similarly, the implementation of application functions can be altered without concern for possible effects on the interface.

The UIMS approach of automatically generating the UI from its description also has implementation advantages. Most obviously, once the specification is complete, no further manual programming need be done to produce the interface; the UI generator (compiler or interpreter) does the work. Given suitable representations of the interactions in the UI description, many of the details of implementation can be left to the UI generator and need not be re-constructed for every interface (Hayes, 1985, pp. 163-4). Furthermore, low-level interaction techniques can be optimized and made applicable to all interfaces produced by the UIMS.

1.4.3 **UI Quality Improvement**

The need to learn new interfaces or dialoque styles for each new application, a common situation where the UI is embedded in the application, can place heavy and unnecessary demands on a user. This is particularly true where a number of applications are closely interleaved in use, or even used concurrently, as in an automated office environment where a word processor, electronic mail system, diary, spreadsheet and database may all be used by the same individual.

"A uniform style across applications is a great help to a user approaching an unfamiliar program - he can bring much of his "intuitions" from other programs." (Fahlman et al, 1984, p. 555) While a consistent interface can be produced without UIMS methods, a UIMS can make it much easier to provide this feature (Olsen et al, 1984b, pp. 35-6). When changes to the UI are perceived as desirable or new applications added to an integrated system, consistency of interface can be maintained via a UIMS without undue effort (Thomas, 1983, p.19).

Although general consistency is useful, it is also the case that different user populations, indeed different individuals, require different interaction styles. Most well-known is the preference of novices for menu-based systems and experienced users for command languages. Some users find mouse-based selection preferable, others function keys. Using a UIMS, different UI specifications can provide individualised interfaces to the same applications (Feldman et al, 1982b, p. 3). The UIMS execution environment can be so constructed that adaptation of the interface takes place during execution either through user selection or automatically by means of an intelligent monitor (Alty, 1984).

1.5 UIMS Problems

In spite of the numerous advantages which appear to follow from using a UIMS to produce interfaces, there is a price to pay, at least with current UIMS implementations. These will be examined in detail in later chapters, but it is perhaps helpful to enumerate here the sort of problems encountered in UIMS development.

The specification language or notation and the execution environment together determine largely the sort of interaction styles and dialogue features available to the designer. Some systems allow only textual input and output, e.g., early SYNICS (Edmonds et al, 1984a). Others do not provide facilities for globally available commands, command undo, or multi-stream (i.e., concurrent) input and output. Dialogue may be limited to menu selection, command entry or form-filling. Natural language input is not supported by any current UIMS. Most cannot change the style of dialogue dynamically at run-time based on user or application behaviour. While these limitations may be acceptable for the types of applications for which the systems were designed, they also prevent the UI designer from exploring possibly better interfaces. In general, decisions

about dialogue style should be in the hands of the designer and not constrained artificially by the system being employed.

The usability of the UIMS can be affected by several factors. Specification languages which are difficult to use are unlikely to be taken up by UI designers. The ability to change the dialogue at run-time can allow users to adjust the interface to suit them; the absence of this feature lessens the value of the UIMS product. Many UIMS require applications to be restructured, hence rewritten, limiting the applicability of the UIMS to custom-built applications. The UI execution environment, particularly if interpretative, may be too inefficient to support usable dialogues.

Also, as mentioned earlier, many UIMS cannot handle application-generated output. However, the current state of the dialogue may well depend upon application information displayed to the user. User control over the way application-generated data is displayed is a desirable feature for certain interactive systems; this could only be possible in a UIMS if it has some degree of knowledge of and control over such output.

These problems, briefly enumerated, are not presented as a critique of the UIMS approach. Rather, they serve as a set of criteria against which to judge UIMS implementations or, alternatively, as a programme for UIMS development. Viewed positively, the problems can be transformed into the following criteria for the adequacy of a UIMS:

i. The expressiveness of the method of UI specification; that is, its ability to describe a range of interactive actions, styles and techniques.

ii. The ease of use of the UI design environment.

iii. The degree to which both input and output can be controlled, including semantically significant feedback.

iv. Adaptability of the UI, i.e., the ease with which changes can be made to the interface, including the speed with which the altered UI is generated.

v. The ability to link UI to application code with minimal recoding.

vi. The efficiency of UI execution.

Chapter Two

UIMS Models

2.1 UIMS Run-time Systems

At the heart of any UIMS is the User Interface specification -- the characterisation of the UI in a form which can be executed by the run-time system. Before looking at UI specification methods, however, it is necessary to examine the various UIMS run-time systems. This is not meant to imply that the UI structures are designed to fit the run-time system; quite the reverse is usually true, with the run-time system being designed to interpret or execute dialogues of the sort described by the UI specification notation. Nevertheless, the run-time system does embody the model of the UI and an examination of the models and their exemplification in run-time systems will be helpful in understanding the UI specification methods which accompany them. A summary of the systems examined, including their UIMS models, is given in Appendix A.

2.2 Locus of Control

One way of categorizing UIMS execution models (hereafter referred to as the **UIMS model**; the design and evaluation environments will not be considered part of the model for present purposes) is in terms of the ultimate locus of control over interaction sequences. From the point of view of the user interface, control may reside either in the application or in the UIMS. The control of the interaction by the user is not treated as a third option, but is considered equivalent to UIMS control.

Application control, hereafter called **Internal control**, treats the UIMS as a source of application-formatted input and as a destination for application-formatted output. The UIMS is viewed by the application as a collection of routines which can be called whenever I/O functions must be executed. Decisions about the timing and ordering of such actions are in the hands of the application. The purpose of an internal control UIMS is to deal with the prompting, low-level feedback, interaction techniques employed and possible validation of application-meaningful input and the appearance and location of output. Fig. 2.1 shows the relation of the application and an internal control UIMS.

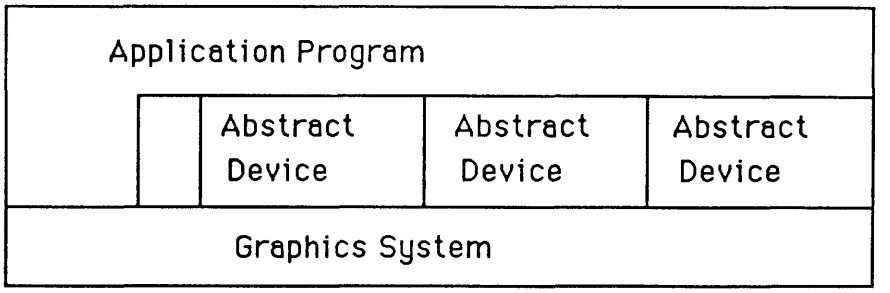

Fig. 2.1 An Internal Control UIMS (Thomas, 1983, p. 17)

External control UIMSs take away from the application the decisions about when information is passed to and from the user and the way application actions are structured. The application is called from the UIMS to carry out computational functions when necessary, given the current state of the dialogue managed by the UIMS. Fig. 2.2 is a diagram of the UIMS/application relation under external control.

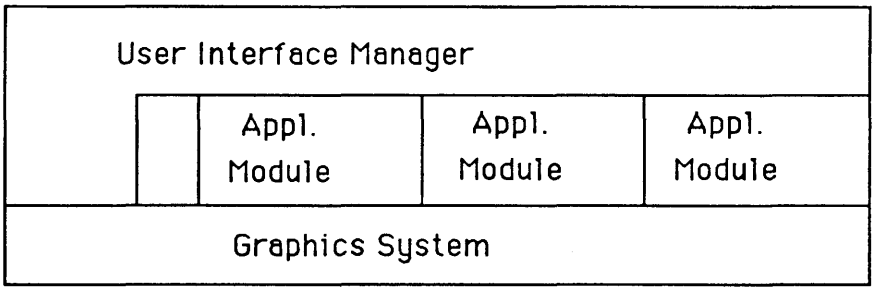

Figure 2.2 An External Control UIMS (Thomas, 1983, p. 17)

Some designers (Takala, 1985, p. 60) do not include internal control systems in the UIMS category, reserving that appellation for external control systems only. Following GIIT, however, we shall include both external and internal control systems within the UIMS definition.

Both external and internal control models are based on a more fundamental paradigm of interaction as **single-threaded**. That is, there is only a single interaction process which requires a single controller of the sequencing of processing of interactive events. However, if the interaction is conceived of as multi-threaded, it is not so clear why "control" must reside uniquely

anywhere. As Kamran and Feldman have pointed out:

"... in a multitasking or parallel processing environment, the question of "who is in control" becomes a very fuzzy issue and depends largely upon how one looks at the system. In such an environment, at any particular moment, control may reside at all levels of the [UIMS]." (Kamran & Feldman, 1983, p. 59)

Most common among actual systems are internal and external control systems. Furthermore, they tend to share a model of UIMS internal structure by which they can be distinguished. For our purposes, we shall call these the **Linguistic Model** and the **Device Model**. Their connection is shown in the table below.

Control	Model
External	Linguistic
Internal	Device
Mixed	(various)

Note that Mixed Control systems have no common model. There are too few such systems yet implemented to determine if there is a common model they employ. Whether this connection of control mode and UI model is just an accident or is an integral feature (e.g., must all external control UIMS use a linguistic model?) is a question to which we shall return.

2.3 The Linguistic Model

The most common type of UIMS so far constructed provides external control and has a "linguistic" structure. What is meant by a linguistic structure will be examined below, but it must first be made clear that we are not speaking of the nature of the UI, or Dialogue, Specification, or the notation system employed to describe the UI. Rather, we are referring to the way the UIMS run-time system is organised, including its relationship to the other components of the system (graphics system and applications modules) as well as the way the dialogue is executed. However, saying that the UIMS model is linguistic does not imply that any particular approach is

taken to UI specification.

2.3.1 **The Language Model of UI Design**

Moran (1981) and Foley & VanDam (1982) present linguistic models of interactive systems as a design tool. It is worth elaborating this model somewhat since it underlies the linguistic model of UIMSs (Takala, 1985). Indeed, it could be argued that the linguistic model is an attempt to provide an environment in which this design methodology can be carried out. In the ensuing section, Moran's Command Language Grammar will be the focus, but Foley & Van Dam's Language Model will also be examined. It should be noted that we shall not be looking at the Command Language Grammar from the point of view of its adequacy as a design method, but only as a possible structure for implementing an interactive system.

The CLG approach begins from the assertion that the starting point for interactive system design is the user's conceptual model of the system (Moran, 1981, p. 4-5). This conceptual model, and thus the CLG, consists of the following components:

Conceptual Component:	Task Level
	Semantic Level
Communication Component:	Syntactic Level
	Interaction Level
Physical Component:	(Spatial Layout Level)
	(Device Level)

The Physical Component, although part of the global UI and hence of concern for UI designers, will not be considered further here as it consists of the "givens" of the environment in which the UIMS operates and thus deals with factors outwith the concerns of UIMS design. The remaining four levels are also identified by Foley & Van Dam, except that they give different names to the parts of the model (Foley & Van Dam, 1982, pp. 220-221):

Moran	Foley & Van Dam
Task Level	Conceptual Design
Semantic Level	Semantic Design
Syntactic Level	Syntactic Design
Interaction Level	Lexical Design

These components of the interactive system become the stages in a hierarchical design method. Moran states:

"The CLG Levels are arranged so that they map onto each other in sequence. This can be interpreted as a top-down design sequence: first the conceptual model is created, then a command language that implements that conceptual model is designed." (Moran, 1981, p. 7)

The actual notation used for the CLG is not of concern here. What is important is that the UI design is viewed as equivalent to the design of a formal language with the following functional elements:

Task Level: the description of the tasks required of the system by the user.
Semantic Level: the definitions of the objects and operations on those objects required to carry out the tasks enumerated above.
Syntactic Level: the definition of allowable sequences of atomic interactive events, i.e., the tokens of the language, such that the operations of the semantic definition can be carried out.
Interaction (Lexical) Level: the specification of how the input tokens are formed by user manipulation of the system's physical devices. It must also include, where appropriate, how physical output is generated by the occurrence of output tokens.

Fundamental to this separation is the notion that each level may be completely specified without consideration of the specification of lower levels. "The levels are ordered so that the description at each level makes only a few global assumptions about the system features described in lower levels." (Moran, 1981, p. 28) Whether this decoupling of stages and the

separation of system components is adequate for describing interactive systems is an issue to which we shall return (Section 2.3.8).

Moran's CLG and the Foley & Van Dam Language Model are intended as design methods and not as implementation systems (Moran, 1981, p. 48) Nevertheless, any design methodology must have implications for the form of its implementation. Moran briefly considers a "programmable user interface" which uses "CLG structures literally as the implementation structure." This is approaching the linguistic model of the UIMS.

2.3.2 **The Seeheim Model**

Arising out of the Seeheim Workshop on User Interface Management Systems (Pfaff, 1985) has come perhaps the most fully elaborated version of the linguistic model of the UIMS. As such, it can serve as the basis for examining actual systems. The Seeheim Model consists of three basic components (Green, 1984, p. 305):

Presentation System

The presentation system deals with the processing of low-level input and output. It may be the physical devices available, a graphics system like GKS or CORE, or a set of virtual devices.

Dialogue Control System

This system uses a dialogue specification to determine the effect of processing input from the presentation systems and application modules. Such input is usually in the form of requests for application actions to be carried out and requests from the application for output to be displayed by the presentation system. The dialogue specification may be compiled into an executable module or modules or it may serve as input to a run-time interpreter.

Application Interface Model

This is "a representation of the application from the viewpoint of the user interface." (Green, 1984, p. 306). This element is the least well-developed of the three in most UIMSs, although it can be identified logically. Information about application data-structures and available application routines with which the UIMS may communicate allows the UIMS to carry out validity checking of

input and transformation of input to make it suitable for the application, thus moving recovery from some sorts of error into the interface. In many systems, this part of the UIMS is either implicit or built into the method of communication between the dialogue control system and the application.

These three components correspond to the Semantic, Syntactic and Interaction Levels of Moran's CLG, and represent a straightforward implementation of that design methodology in a run-time environment. Note, however, that while the Semantic Level of design includes the specification of the application tasks, the Application Interface Model only includes the information about the application required by the dialogue control system. This is because the Seeheim model, being a UIMS, assumes the existence of separate application modules which deal with the implementation of the tasks specified in the Task Level, or Conceptual Design, of the system. The correlation between the CLG Design Model and the Seeheim UIMS model is shown in the table below:

CLG	Seeheim Model
Task Level	–
Semantic Level	(Application Modules &) Application Interface Model
Syntactic Level	Dialogue Control System
Interaction Level	Presentation System

This analogy with language design is pushed even further by the common use of the term 'token' to refer to the "chunks of information handled by the UIMS." (Green, 1985c, p. 10)

By definition, a linguistic model of the UIMS is an external control UIMS. The dialogue control system has the responsibility for parsing sequences of UI tokens to producing meaningful units which call upon the application modules. The modules called, when and in what order, are managed by the Dialogue Control System.

While these three parts can be identified functionally in a model, they may not necessarily be implemented as separate modules. Furthermore, they may be related to one another in different ways. In the following sections a number of linguistic model UIMSs will be examined with regard to the degree of explicit implementation of these three elements, and the way in which they are

connected.

2.3.3 SYNICS

SYNICS (Edmonds, 1981; Guest, 1982; Edmonds, 1984a) and its later enhancement SYNICS2 (Edmonds & Guest, 1984b, 1985), developed by the Man-Computer Interface Research Group at Leicester Polytechnic (see also Section 3.4.1), provides an example of a UIMS in which the system structure fits the model described above. In SYNICS dialogues are specified in a Dialogue Description Language (DDL). The dialogue control system consists of a dialogue processor which uses the dialogue specification to control the path of actual dialogues, recognise and transform input, pass transformed input to the application, called the background, and pass output to the presentation system.

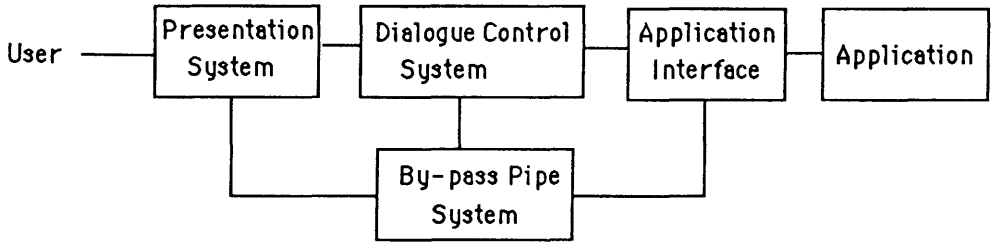

Figure 2.3 The SYNICS2 UIMS (Guest & Edmonds, 1984, p. 340)

SYNICS2 uses a restricted subset of GKS as its presentation system. The dialogue control system can accept input by means of an INQUIRE command. The input available includes device and GKS workstation information. Only GKS request mode is implemented at present. Output is similarly based on GKS except that no coordinate transformations are allowed. When large amounts of data must be input or output without any processing by the dialogue control system (e.g., bitmaps), such information can be passed directly from the presentation system to theapplication and conversely by means of a by-pass pipe (Guest and Edmonds, 1984, p. 342)

SYNICS2 takes the UIMS approach seriously in that, unlike many UIMS, all input from the user and all output from the application are mediated through the dialogue control system. Even in the case of data by-passing the control system, the process of transfer is under dialogue control.

However, the application interface is not particularly well developed, as calls to applications are

simply embedded in the dialogue specification. Such calls consist of the passing of parameter lists (which may include command or function names) to a concurrent application process (the background process) (Edmonds, 1984, pp. 53-54).

2.3.4 The TIGER UIMS

The TIGER UIMS (Kasik, 1982) consists of a dialogue specification language, TICCL, and a run-time interpreter which uses application-specific dialogue specifications produced by TICCL as input to determine the current state of the dialogue and the effect of user interactions (i.e., which application modules to call). The system is limited to hierarchically structured menu-based dialogues.

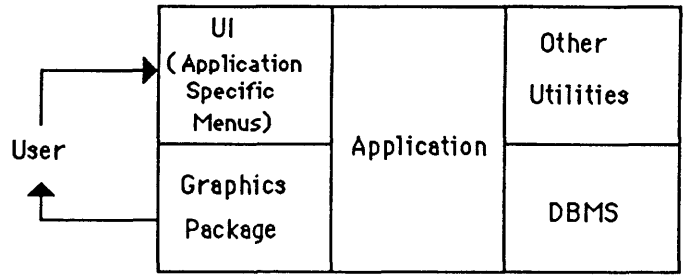

Figure 2.4 The Tiger UIMS (Kasik, 1982)

There is no independent presentation system in the TIGER UIMS, except in the sense that the dialogue control system makes use of the graphics package to accept input and to display prompt information.

All dialogue-specific input and output are handled by the UIMS, although application-specific output is not controlled. This means that the UIMS is not seen as a mediator between the user and the application with regard to all input and output; it is primarily concerned with processing user input and only handles output in so far as it applies to prompts and non-semantic feedback for user actions. This avoids a number of problems related to UI access to the application database and its relation to graphical output (i.e., the application interface model), but at the expense of limiting the applicability of the UIMS.

Since this is an external control UIMS, the sequence of actions within the dialogue is

determined by the dialogue specification. The application consists simply of a set of modules which are called at points defined in the dialogue specification. Thus, as with SYNICS2, no explicit application interface model is necessary.

2.3.5 SYNGRAPH & IPDA

The SYNGRAPH system (Olsen, 1983b), like SYNICS2, defines the dialogue specification by means of a regular grammar. The semantics of non-terminals in the specification (i.e., actions carried out during the execution of the dialogue when the non-terminal to which the actions are connected is recognized) consist of Pascal statements embedded in the dialogue specification. Thus, the application interface model need not exist explicitly; the application is embedded in the dialogue iteself. One might say that this is the limiting case of the External Control type of system and, in a sense, the reverse of conventional methods of UI implementation in which the dialogue is embedded in application code.

The Interactive Push-Down Automaton (IPDA) UIMS (Olsen, 1984a) extends the SYNGRAPH system by creating an explicit presentation system and a (minimal) application interface. As with SYNGRAPH, the dialogue is defined in a specification language, producing an IPDA which is then interpreted by the dialogue handler (Olsen, 1984a, p. 178)

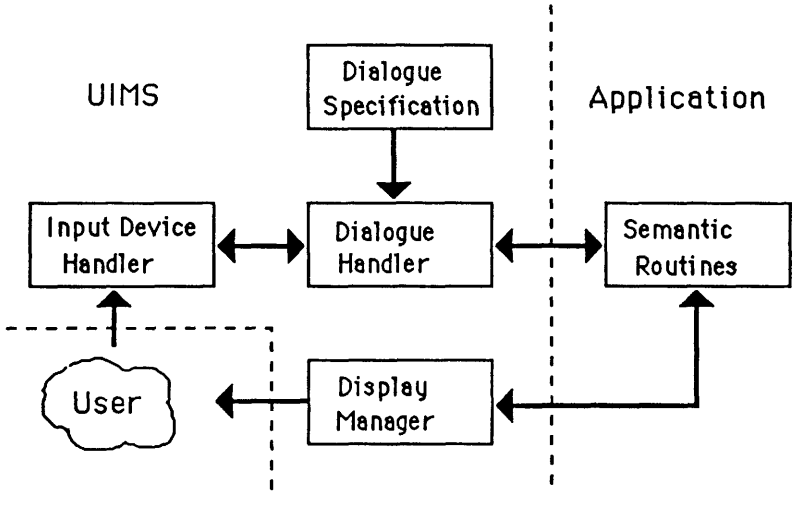

Figure 2.5 The IPDA Run-Time System

The presentation system consists of a set of input devices, which may be physical, logical or virtual. "The class of a device defines the number and types of its attributes and whether it is an

event or sampled device. A set of device classes, coupled with a set of device identifiers, constitutes the device configuration of an IPDA." (Olsen, 1984a, p. 180) Using an independent module of this nature, which provides for the construction of complex, abstract devices allows the dialogue control system to leave aside details of device configuration. Such a system may be built on top of a graphics system like GKS or CORE, but provides greater flexibility in the types of devices and input made available to dialogue control. Furthermore, it allows for the presentation system to handle prompting, echoing and acknowledgement of inputs. (Olsen, 1984a, p. 180) The IPDA system allows prompting and feedback to occur at the lexical (presentation), syntactic (dialogue control) and semantic (application) levels, although the last is not under UI control.

Like TIGER, the IPDA application interface consists of calls to application routines with required input passed as parameters in the subroutine calls, this replacing the actual application code as in SYNGRAPH. The dialogue control system is still not treated as a separate process as in SYNICS2, and while this simplifies the passing of information to the application, it limits the flow of control to a single thread. Furthermore, graphical output from the application routines is not under dialogue control, but is passed directly to the presentation system.

2.3.6 **The Toronto UIMS**

In the University of Toronto UIMS (Buxton et al, 1983b), dialogue specifications are networks of menus produced by the MENULAY system and which are input to the MAKEMENU program to produce a table-driven dialogue execution program. The application actions are separate C routines which, along with the dialogue specification, serve as input to MAKEMENU. Buxton et al described the application interface thus:

"Where the applications designer specifies names of application-specific routines, the programs generated by MAKEMENU contain unresolved external references ("hooks"). By writing functions with the required names and referencing the appropriate file names when MAKEMENU is called, the applications programmer can add any amount of application-specific programming to the layout and sequence information specified by the [dialogue] designer." (Buxton et al, 1983b, p. 37)

The presentation system is based on the GPAC graphics package (Reeves, 1978) providing generation of events from a graphics tablet with puck and keyboard. Provision of input from other devices such as a mouse, light-pens, etc. is possible (Buxton et al, 1983b, p. 40). However, there is no support for logical or virtual input devices. Furthermore, all prompting and feedback must be handled by the dialogue control system, increasing its complexity and making the distinction between lexical and syntactic feedback difficult.

The application modules which are linked to the dialogue control system by MAKEMENU may produce output independent of the UIMS. Compared to SYNICS2, this limits the generality of the UIMS as a vehicle for the modification of the appearance of the dialogue output and suffers the same design drawback as TIGER, SYNGRAPH and the IPDA.

2.3.7 **The University of Alberta UIMS**

The University of Alberta UIMS was designed to incorporate all the features in the Seeheim model, so it is not surprising that it comes closest to that model in its structure. Each of the three components of the model is implemented as a separate process which communicates by the passing of tokens, which represent events (Green, 1985b, p. 212). Following the Seeheim model, the three modules are the Presentation System, the Dialogue Control System and the Application Interface Model.

The concept of events underlies the entire system. Each of the three components consists basically of a set of event-handlers which can process events generated by the lower-level components and which themselves may output tokens which are passed to another level for processing.

The presentation system is based on a graphics/windowing system called WINDLIB. Each input event possesses a name, position and, possibly, a value (Green, 1985b, p. 208). Every window has associated with it an event handler. WINDLIB passes the event for handling to the window of the highest priority which covers the position at which the event occurred. Each window can also have a menu associated with it.

"Each menu is viewed as a collection of menu items. A menu item consists of an input token, and a text string or icon. When then menu item is selected its input token is sent to the dialogue

control component." (Green, 1985b, p. 209)

Output is handled by associating a display procedure and a window with each acceptable output token which can be processed. The display procedure may be written specially or chosen from a standard library. Some of these procedures allow the transformation of contents structures (Green, 1985b, p. 208), which are application-meaningful data structures, into graphics output primitives to display the structures.

This highly-developed presentation system allows complex virtual input devices to be programmed and made available to the dialogue control component. Similar virtual devices could be produced using, say SYNICS2 or SYNGRAPH (although not TIGER, which only allows menus), but they would have to be produced from within the dialogue control system, thus complicating the design. The limitation of one event handler, plus a single menu, to one window is a problematic restriction. The ability to define a set of virtual devices, as in the IPDA UIMS, and then bind them to windows as desired, seems a more flexible approach.

In addition, WINDLIB support also allows these event-handlers to operate concurrently. A user could start interacting with one window and then move to another, without having to complete the interaction in the first. Such multi-threaded interaction would be most difficult to produce using the earlier UIMSs described, short of producing multiple independent UIMSs.

The dialogue control component has a similar structure to that of the presentation system. It consists of a set of event handlers which can deal with tokens generated by the presentation and application interface systems, as well as those generated by other dialogue control event handlers. Each token is mapped onto events which are processed by the event handlers; the result of such processing includes changes to the values of local and global variables (i.e., changes to the dialogue state), activation and deactivation of event handlers, and the generation of tokens sent to other event handlers, the application interface model or the presentation system (Green, 1985b, p. 209-210). As with the presentation system, more than one event handler may be active simultaneously. The ability to "parse" several interaction sequences concurrently increases the range of interaction styles possible well beyond the single-thread types supported by the systems described so far.

Finally, the application interface model consists of a set of routines to process tokens passed to

it by the dialogue control system. Each token has code associated with it which is executed when that token is received. There are also local variables available, since as Green states:

"The mapping between tokens and application routines may not be one-to-one. A token may cause several application routines to be executed, or it may contain data used in a subsequent call of an application routine." (Green, 1985b, p. 210)

One useful consequence of this explicit application interface model is that parameters for a command can be collected in any order and stored in local variables in the application interface until the token arrives which "fires" the call to the application routine. A similar interaction technique would be difficult to produce in the other linguistic model UIMSs due to the embedding of application calls in a single-thread dialogue control system. Kasik's TIGER system does provide special system support for this type of interaction. It is, however, not a generalisable feature.

There is actually a fourth component present, a scheduler, which is needed to support the asynchronous communication between processes implied by the token-passing mechanism (Green, 1985b, p. 212). Each of the components has a queue onto which tokens sent to it are placed. The scheduler operates as follows:

"The scheduler examines the scheduling queue associated with each of the components and selects one of the tokens for execution. Highest priority is placed on the presentation component, and lowest priority on the application interface model queue, with the restriction that a token will not be blocked for an arbitrary (sic) long time. The scheduler then calls the appropriate routine in the receiving component to process the token." (Green, 1985b, p. 212)

The University of Altberta UIMS is the most complex of the linguistic model UIMS, at least in terms of the dialogues it can express. This is not so much a result of the separation into three components. All the linguistic model UIMS have the three components at least logically and could have the implicit components expanded into separate modules. The additional expressiveness arises from its ability to support concurrent, or multi-thread, dialogues. The existence of an

indeterminate number of independent event-handlers on three different levels, however, can result in dialogues where the behaviour of the system is difficult to predict. Some way of controlling this complexity seems in order, and the device model of UIMS provides just such a source of control.

2.3.8 **Problems with the Linguistic Model**

While the separation of the UIMS into three distinct components appears justified in terms of supporting a top-down design methodology, problems arise particularly when attempting to provide highly interactive graphical interfaces. As Kamran points out, "immediate feedback is desired at the lexical, syntactic and semantic levels while the command is still being formulated." (Kamran, 1985, p. 44) However, where the three levels are separated, the lexical analysis is complete before a token is passed to the syntactic level, i.e., the dialogue control component, and the syntactic analysis is carried out before the application is invoked. Hence, the provision of continuous multi-level feedback appears to be at odds with the linguistic model. Such continuous feedback where the three levels are not discriminable is referred to by Kamran as "dynamic semantic feedback" (Kamran, 1985, p. 45) An example is the manipulation of a graphical entity on the screen, for example, picking a picture segment and moving it to a new location. If this is done by dragging the object in real time, the object's movement can be considered to be simultaneously giving lexical feedback (echoing of the command), syntactic feedback (the very movement confirms that the action is syntactically acceptable) and semantic feedback (the desired application task is carried out before one's eyes). To implement this type of feedback with a linguistic model UIMS would require either reducing the interactive events to the smallest possible unit so that information could be passed back and forth between the components at the completion of each atomic physical device event or by providing some special-purpose run-time mechanism to allow this form of integrated feedback. In either case, the apparent value of separating of the three components would be reduced if not removed.

Furthermore, it is not at all clear that the choice of interactive device at the presentation level is independent of the appropriate syntax for interactive dialogues (Buxton, 1983a). The separation of the presentation and dialogue control levels suggests that interactive sequences can be specified at the dialogue control level independent of the sort of device used to generate the

input events. This decoupling in fact underlies the concept of logical devices in GKS. A closer relationship between the device chosen, including the particular style of interaction it supports, and the syntax of the interaction appears to be desirable.

2.4 The Device Model

One subgroup at the Seeheim workshop suggested that a UIMS model based upon "an application-dependent hierarchy of interaction blocks" (Strubbe, 1985, p. 6) was more desirable for interactive interfaces than the linguistic model. The Device Model (not to be confused with Anson's Device Model; see Section 2.4.3), is just such a model, treating the UIMS as a collection of abstract, or virtual, input and output devices. In practice, these systems usually just implement the input devices. This may appear similar to the University of Alberta UIMS, which is a linguistic model UIMS. However, in the Device Model there is no clear-cut distinction between the presentation, dialogue control and application interface components; each device deals with its own prompting and lexical feedback, has its own "parser" for operating on tokens passed to it from other devices, and may communicate directly with the application. Thus, the close coupling of lexical, syntactic and semantic levels of the dialogue seems better served by this approach.

The devices are constructed hierarchically out of lower-level devices in a fashion similar to the organisation of modules in a block-structured language. Arbitrarily complex abstract devices can be built by this method. Furthermore, concurrent activation of devices is a basic feature of the device model.

The most common view of the device model is that it is an internal control UIMS (Takala, 1985, p. 60). That is, the application program makes calls to the abstract devices when input is required or output is to be generated. Unlike the linguistic model, however, this is not a necessary condition of device model UIMSs. As shall be shown below, a device model UIMS may have the control of the interactive sequences built into the devices themselves.

2.4.1 Dialogue Cells

In the Dialogue Cell approach (Borufka et al, 1982; Pfaff et al, 1982), a UI specification consists of a hierarchically-structured set of dialogue cell definitions, each cell optionally containing sub-cells. "Complex input data structures can repeatedly be decomposed until reduced to one of

the five logical input classes. Substructures can be entered by subdialogues." (Borufka et al, 1982, p. 27) The logical input classes form the primitive dialogue cells from which all the others are built, and include the GKS logical devices (Borufka et at, 1982, pp. 26-27). These might be construed as the presentation component, except that these devices are identical in structure to all the higher-level cells constructed from them.

Each dialogue cell consists of four parts: a prompt, a symbol, an echo and a value. The **prompt** component not only presents the prompt for user input, but also enables input devices, formats the display, and initialises the value part, if necessary. The **symbol** component is the syntactic rule for completing the dialogue cell and consists of an expression built up from operations on values generated by subdialogues. Dialogue cells referred to in the symbol component may be activated concurrently. The **echo** component provides lexical and syntactic feedback to the user. Finally, the **value** component produces a value which can be returned to the calling cell or, where appropriate, to the application. (Borufka et al, 1982, pp. 28-30)

The prompt component can also present output generated by the application. As Borufka et al state:

"Pure output cells are handled through the prompt. These cells produce only output with a specified lifetime, which is set by scope rules. Although pure output is relatively unimportant in dialogues, it is reassuring to know that it can be dealt with as a special case." (Borufka et al, 1982, p. 29)

This comment indicates that the dialogue cell model has been designed to deal particularly with interactive input only. As has been argued, any adequate UIMS must be able to deal equally with both input and output. Handling output as a special case of prompting for input must be judged an ad-hoc and unacceptable technique. The mapping of a model of interactive dialogue onto the UIMS model, well provided for as far as input is concerned by the four part structure of dialogue cells, is thus destroyed. Clearly, some better model of interactive dialogue events is needed.

The dialogue cell model is an internal control UIMS. That is, when input or output is required, the application program calls a dialogue cell which can produce the needed application-meaningful value or can generate the desired output. "The root [of the tree forming

the hierarchical ordering of the dialogue cell structure] is the interface to the application." (Borufka et al, 1982, p. 30) Figure 2.6 shows this structure.

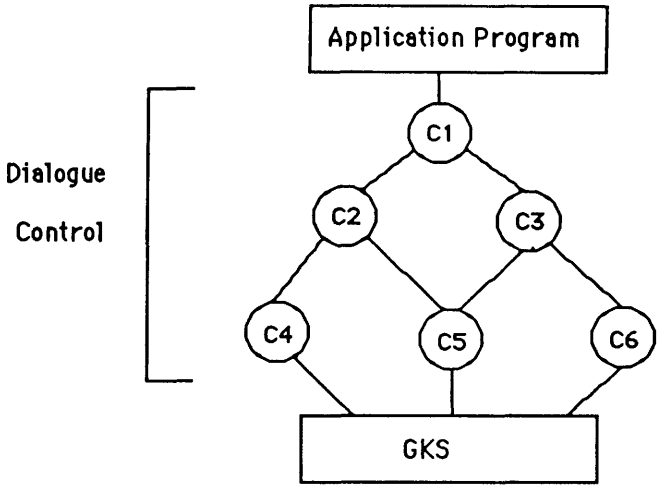

Figure 2.6 Dialogue Cell Structure

However, a later description of the model (tenHagen, 1985, p. 120) suggests that data can be sent to the application program via calls to a message-passing mechanism from within the value component. The passage in which this occurs also provides an insight into how the Dialogue Cell model would handle continuous feedback. An example is given of a dialogue cell implementation of a system to allow the positioning and labelling of a rectangle within a "map". One version allows immediate feedback of the rectangle's current state (size, location and label) as soon as any one of its parameters is changed. This is done by keeping each of the four subdialogues active (i.e., those to change the size, location, label and to signal completion) until the completion cell is satisfied, and echoing immediately any change in the current state of the rectangular object. (ten Hagen, 1985, pp. 122-123)

This would appear to satisfy Kamran's condition for continuous multiple-level feedback, except that the resultant triple (size, location and label) is only sent to the application once the entire interaction is complete. Thus, semantic feedback, e.g., the legality of the size, location and label in terms of the application model of the "map" data structure, is not made available to the user until the sequence is complete. Lexical and syntactic feedback are continuous, but semantic feedback is not. Continuous semantic feedback would only be possible if the dialogue subcells

dealing with the setting of the three parameters had knowledge of the constraints on the application data structure. And this would appear to require some sort of application interface model similar to that in the University of Alberta UIMS. Thus, the problem of providing continuous integrated feedback at all three levels is not removed by the dialogue cell approach, though. Because of the internal control structure and lack of sophisticated support for application output, any semantic feedback must be the concern of the application alone.

The organisation of a dialogue cell into prompt, symbol, echo and value is similar to a linguistic model view of the presentation system as a set of "interaction techniques" (Thomas 1985, p. 82). This is not surprising, given that the dialogue cell model is an internal control system and thus does not contain explicit dialogue control or application interface elements. Nevertheless, the combination of input expressions in the symbol parts of all the cells in a path of the hierarchy can be construed as a syntax for the interaction. There is clearly more structure to the dialogue cell model than would be found, then, in a linguistic model presentation system alone.

2.4.2 **Input Output Tools**

The Input Output Tool model (see also Section 3.3.4) also starts from a set of hierarchically-organized arbitrarily high-level abstract devices. Like the Dialogue Cell model, each input output tool has an internal structure which corresponds to a particular view of the nature of interactive events. There are four part which correspond to the four parts of a Dialogue Cell, viz., the initialisation, an input rule, a tool body and a returns expression (Van Den Bos, 1980, p. 160); there is also a possible cleanup which is performed if a special escape tool cancels the tool's activation. The correspondence between Dialogue Cell and Input Output Tool is shown in the table below:

Dialogue Cell	Input Output Tool
Prompt	Initialisation
Symbol	Input Rule
Echo	Tool Body
Value	Returns expression

An important difference between the two is the fact that the tool body may contain a set of actions, including "nested tools, statements and local procedures" (Van Den Bos, 1980, p. 160), which makes an Input Output Tool much more powerful than the dialogue cell since tools can be called as part of the actions which execute the tool, as well as providing values for the expression which comprises the input rule.

The tools are block-structured in the sense that lower-level tools must be "defined at the same or a surrounding block level in the program or else should be defined within the tool definition." (Van Den Bos, 1980, p. 161) Tools which appear in an input rule expression may be joined via concatenation ('&') which does not imply precedence in time; that is, an input rule of the form 'A & B' which refers to two tools, A and B, would call these two tools to be active concurrently. Other operators allow the specification of temporal precedence, conditional rules and iteration.

Like Dialogue Cells, there is a set of basic input tools, similar to GKS logical devices, although including a Clock and an Escape tool. The escape tool allows a user to cancel the parse of the input, followed by the deactivation of the currently active tools and the return to some earlier tool as defined in the body of the Escape tool. Kilgour mentions the problem of providing global commands (i.e., commands available at all points in a dialogue) using a model like Input Output Tools (Kilgour, 1983, p. 19). To make a command globally available throughout the dialogue, it would appear necessary to either:

i. define a tool corresponding to the global command at the top of the tool hierarchy and then produce a conditional input rule to produce a value from every lower-level tool which would activate this global tool, or

ii. provide a mechanism for "specifically listing global tools within a tool definition. Any instances of such tools encountered by the parser while trying to satisfy the input rule would be accepted but ignored, and tools listed as global in one tool definition could automatically be exported to lower-level tools." (Kilgour, 1983, p. 19)

Option ii, using a mechanism similar to the escape tool, which allows its concurrent activation at all lower points in the parse and which can be redefined as necessary, would seem to satisfy the requirement. This problem is one of several which arise from the use of a grammar-based UI specification method, and as such will be covered in the next chapter. However, it is worth noting at this point that the introduction of special-purpose tools can serve to overcome inherent

difficulties in the model.

The connection to the application in the Input Output Tool model is via statements within the tool body, similar to the Dialogue Cell approach. Application actions are thus "side effects" of the satisfaction of the tool's input rule. This would seem to imply that the Input Output Tool model is an external control UIMS although it does not employ a linguistic model. It is similar to the University of Alberta UIMS in that concurrent tools are like event handlers; however, the University of Alberta event handlers are not block-structured, nor do they possess an internal structure like the tools in the Input Output model.

2.4.3 **Anson's Device Model of Interaction**

The Device of Model of Interaction (Anson, 1982) consists of a set of abstract input devices which have (i) an external behaviour including a state visible to other devices, and the ability to signal events to other devices and carry out actions when signalled by other devices, and (ii) an implementation, which is not accessible to other devices. The devices include primitive devices as well as composite devices which are composed of one or more lower-level devices.

The application views the devices as generators of logical input events and values, with the devices translating "between these abstract notions and the physical realities of the [physical] devices being used." (Anson, 1982, p. 110) The existence of devices is dependent upon the existence of the device inside which they are declared and not on the devices which may examine their values or receive their signals; thus they are more like variables than subroutines. This differs from the Input Output Tool model in that tools only supply values to higher-level tools which call them. Also, Input Output Tools only exist while part of an active parse; Anson's devices can continue to exist even when not called; that is, they retain their value and can update their value between uses.

The Device Model is similar to the object-oriented approach of Smalltalk. Each object in Smalltalk has a defined set of messages to which it can respond by changing its state and/or sending messages to other objects. Objects can be defined to respond to messages generated by input events or the state of the physical input devices and other objects can cause modifications to the state of physical output devices as part of the response to messages (Goldberg & Robson, 1983).

Unlike Anson's Device Model, in which abstract devices are the only objects supported by the system, Smalltalk treats all system entities as independent objects capable of sending and responding to messages. There is no identifiable independent dialogue structure and thus no UIMS, although a UIMS could be implemented within Smalltalk. Also, Anson's devices can operate concurrently, while there is no system support for concurrently executing input and output devices in Smalltalk. Finally, Smalltalk objects are organised into classes in such a way that an object or subclass of objects inherits all the attributes of the class to which it belongs; there is no similar class organisation of devices in Anson's Device Model.

As with the other device models, there is no distinct presentation system. All prompting, formatting of the display, and lexical and syntactic feedback are built into the device definitions. Unlike Dialogue Cells and Input Output Tools, there is no provision for device control of application generated output. However, the model appears extendable to cover output devices which could be called by the application to produce output.

2.5 **The Dialogue Management System (Virginia Polytechnic)**

Linguistic model UIMSs offer the advantages of separate implementation of functionally distinct parts of the UI based on a well-developed design methodology, the ability to specify the syntax of the dialogue in a single description from which the dialogue control system can be generated, and a view of the application interface with allows the UI to handle both input and output equally well. However, it seems better suited to single-thread dialogues and has difficulty with the provision of continuous multi-level feedback, both of which are better dealt with via a device model of the UIMS. Furthermore, both models appear to offer either external or internal control of the dialogue, but not both.

There is another alternative offered for the structure of the UIMS, although it does not fit neatly into the other categories. The Dialogue Management System, or DMS, from Virginia Polytechnic allows separation of the UI from the application without requiring control of the interaction to reside in either component. Rather, "control is either concurrent, or it alternates among the modules." (Ehrich, 1982b, p. 328) To provide this flexibility, the DMS views the application and UI as made up of collections of concurrently executing modules which communicate with each other under the control of a run-time environment which manages process creation and termination and

synchronous interprocess communication (Ehrich, 1982b, p. 332).

User interface design for the DMS is based on a design methodology, called SUPERMAN (Supervisory Methodology And Notation) (Yuntan, 1985) which conceives of the entire interactive program as a hierarchical structure built up of dialogue functions, computational functions and combined dialogue-computation functions. All three types may be supervisory functions, while the first two may also be worker functions. Supervisory functions are composed of other functions, combined algorithmically; sequence, conditional execution, iteration and recursion and concurrent execution are all allowed. A worker function "does not supervise other functions, but performs some work (e.g., dialogue or computation) and provides data to, or gets data from, the supervisory functions." (Yuntan, 1985, p. 250)

Dialogue worker functions are built up in an manner analogous to the device model. Each defines a single dialogue transaction. An input transaction consists of a prompt, a language input (an input rule), and a syntatic confirmation (echo) (Hartson et al, 1984, p. 59). Although not mentioned, there obviously must also be a value returned to the calling function.

The equivalent of the presentation system is a collection of input and output services; these consist of device drivers, where "each I/O device used by DMS is controlled by a program called a driver that executes in a separate process." (Ehrich, 1982a, p. 52) While this allows concurrent I/O devices, the set of devices is effectively static from the point of view of the UI designer. No facility is available for creating higher-level virtual devices. This seems particularly surprising omission given that even Ehrich observes that:

"A significant problem in designing I/O services is that it is surprisingly difficult to select a set of parameterized subroutines whose functionality covers the range of input/output transactions that might possibly be required of a hardware device in the design of a human-computer dialogue." (Ehrich, 1982a, p. 53)

In addition, since dialogue worker functions (the dialogue transactions) only occur at the leaves of the supervisory hierarchy, it does not appear possible to build up complex, abstract devices or transaction-handlers, as in the device model or the University of Alberta UIMS.

While DMS takes seriously the notion that both dialogue and computation (i.e., application)

tasks should be treated as of equal importance and hence that control should not reside uniquely in either component, the free-form relationship resulting seems to complicate design and lessen the value of the UIMS approach. In fact, the flexibility which the system allows is constrained considerably in practice, probably because of the difficulty of actually treating the interface and application modules as interleaved with one another as collections of independent communicating processes. Hartson states that the current implementation creates a program complex, where

"within a complex, the computational component contains the control structure for the application. The dialogue component is in a separate process (or set of processes) containing a collection of dialogue transactions that do not call one another and which have no shared environment. The relationships among dialogue transactions are defined by the control structures in the computational programs that invoke the dialogue transactions." (Hartson et al, 1984, p. 61)

The result, then, is a device model UI with internal control, but without the capability for dialogue functions to call one another or to share a common data structure. This is a far less powerful model than either the most powerful linguistic model UIMS or the device model UIMS.

2.6 **A Hierarchical Event-Based Model**

In reviewing the three models so far presented, and the systems which embody those models, several features seem desirable in any adequate model. These are:

i. continuous multi-level feedback,

ii. support for multi-threaded dialogues,

iii. UI control over all input and output,

iv. the ability to handle globals, undo and escape/cancel commands,

v. as far as is possible given the requirements above, the separation of functionally separate elements of the dialogue into separate modules of the UI.

In order to satisfy these conditions, a hierarchical event-based model of the UI is presented

here. It draws upon features of all three models, but is not equivalent to any of them. At its heart is the notion that dialogue control should be built around dialogue units, which can be executed concurrently. Each should be able to respond to events generated by a presentation system, by application modules, and by other dialogue units. That is, the dialogue control component must be able to satisfy requests for dialogue actions which originate from the user, the application and the UI itself; no type of request for dialogue action should be prohibited, regardless of its source.

Each dialogue unit should have an internal structure which models the functional components of an interactive transaction. Incorporating a model of the transaction in the dialogue unit structure allows a design methodology to be employed which, like CLG or the Language Model of Design, begins with a conceptual description of the application and leads to a specification of the dialogue transactions needed for the application. The linguistic model has been shown to be inadequate because of the problems relating to multi-level feedback. The device model appears more acceptable since the separation of lexical and syntactic considerations is within the device, rather than exterior to it, and can be handled at the same time. Thus, following the device model, each dialogue unit contains initialisation, prompt, activation rule, action and value and clean-up elements.

Dialogue units appear so far to resemble the event handlers in the dialogue control component of the University of Alberta UIMS. However, dialogue units are hierarchical in structure; units are defined within higher-level units. Furthermore, borrowing from the object-oriented philosophy of Smalltalk, classes of dialogue units can be defined, which allows the inheritance of attributes by subclasses of units. As shall be discussed in later chapters, this facilitates the provision of support for global commands, cancel/escape commands as well as controlling the complexity of the specification process.

So far, this model appears to be a type of device model. However, a weakness of the device model is the poorly developed interface to the application. Borrowing from the linguistic model, an application interface system is needed which mediates the communication between dialogue units and the application processes. This allows dynamic configuration of the communication channels, with the destination of messages being changeable as the result of dialogue or application actions. Also, semantic feedback requires the dialogue units to have access to information about the application data. An application interface can contain information on

constraints on application data (e.g., ranges of acceptable values for validation, screen location of picture segments), contain a copy of those parts of the application database which are significant to the UI, or it can mediate requests for information about the application data from the UI.

At the opposite end of the UI, multi-level feedback and other needs of an integrated interactive system demand that the presentation system should not be completely separated from the dialogue units. At some level there must be software modules which mediate between physical input/output devices and the dialogue units, particularly if multiple indepedent windows are desired; thus, there must be a presentation system with which dialogue units communicate, accepting input events which already have attributes like a window, location and, optionally, an additional value. However, the dialogue units should be able, for example, to alter the binding between physical and logical devices, change the size and location of windows, enable/ disable echoing and change the cursor style. Systems like GKS which do not allow full control of such features from outside the graphics/windowing system are not adequate.

A single UIMS system is not necessarily the best solution to all interactive system design needs. It may well be that for many applications a linguistic model UIMS, even one which can only handle single-threaded dialgoues, allows a simpler, and more efficient, design and implementation than using a more complex UIMS with greater functionality. However, a UIMS which has aspirations to being general purpose and able to handle highly interactive multiple applications requires a model which takes features from all the currently available models, such as the Hierarchical Event-Based Model.

Chapter Three

UI Specification

3.1 UI Models

It has already been established that UIMS are intended to assist in the design and subsequent execution of user interfaces. Whether the UIMS model is linguistic or device and whether it embodies external or internal control, there must be a method of specifying the UIs which the UIMS will execute. The specification is based on a language or notation in which the UIs can be described.

UIMS development has tended to focus on syntactic issues and notations for the dialogue control component are consequently the most highly developed, to the extent that Green has identified three UI models based on dialogue control notations (Green, 1985c, p. 14). These are:

i. context free grammars,

ii. state transition networks, and

iii. the event model.

It should be noted that this is not an exhaustive list. Other possible UI models include:

iv. Petri nets and

v. the production system model.

These models, and the notations associated with them, are not equivalent to the UIMS models discussed in the previous chapter. A linguistic model UIMS may employ a grammar-based, transition network or event-based specification of the UI, and similarly with device model UIMSs. In fact, one UIMS may support specification in more than one notation (Guest, 1982; Green 1985b). Appendix A lists the UI models of the systems examined in this chapter.

3.2 Evaluation of Dialogue Specification Methods

A user interface model, and the dialogue specification method based upon it, is a design tool and as such can be evaluated in terms of its support for the design process. Wasserman (1985) provides a set of criteria for judging specification systems:

"1) Formalism: the notation had to serve as a formal definition of the interface.

2) Completeness: The notation had to be self-contained, including user input, system output, and linkage to system operations (application code).

3) Comprehensibility: The notation had to be comprehensible both to system developers and to users (or their representatives).

4) Flexibility: The notation had to accommodate a broad variety of dialog styles. In other words, the notation could not make assumptions about the nature of the human-computer interaction....

5) Executability: The notation had to be directly executable to support prototyping, development, and testing of interactive information systems." (Wasserman, 1985, pp. 705-6)

The formalisation of a specification notation is desirable if analysis of the formal properties of the specification are to be carried out. This is important as a way of discovering ambiguities, inconsistencies and to find unreachable parts of the specified dialogue.

The last criterion, executability, is a necessary condition of any UI specification for a UIMS, at least in the sense that the specification must be transformable into an executable UI definition. The way in which this is done is significant for the design process and will be considered in the discussion of the design environment in the next chapter.

Clearly any adequate specification must capture presentation, control and application interface characteristics, but it needn't, as implied by Wasserman's second criterion, do so in a single self-contained specification. Some linguistic model UIMSs separate the specification of the presentation system from the dialogue control component, using a graphics language to specify the presentation system and then use one of the UI models listed above for dialogue control. Such a separation may be viewed as desirable, in that it keeps lexical and syntactic specification issues distinct, rather than hiding low-level specification within the dialogue control component. The question of how to specify low-level details of the interface depends to a large extent on the UIMS model chosen. Thus, the completeness criterion should be modified to state that notations should be present to capture all aspects of the UI, including all three UI components, although there need not be a single notation for all three.

This leaves the flexibility and comprehensibility critera. Given the above, it would appear that these are the primary criteria and are perhaps better described as criteria for judging:

i. the expressive power of the notation, and

ii. its ease of use.

The notation must enable the designer to produce the types of dialogue he judges desirable given user requirements and human factors principles. Restrictions based on inadequacies of the notation should not get in the way of the designer's freedom for design. Certain types of dialogue rule out certain UI models. For example, natural language dialogues cannot be described by means of a context free grammar. Some models do not support concurrency, which may be a necessary feature of some interactive dialogues.

Of course, the restrictions may simply limit the applicability of the system to certain types of interactive systems and this limitation may be deemed acceptable to the designer. For example, a specification system which only allows alphanumeric input and output may be acceptable if the computer systems it will be mounted on do not support graphical interaction. Even then, however, these system-imposed limitations prevent the exploration of alternative interactive styles which require the missing features (e.g., graphics, concurrency). Thus, even if acceptable to the dialogue designers, a limited specification system may be judged to be less adequate than one without those restrictions.

However, there are many dialogues which are capable of being defined by means of more than one user interface model. The evaluation must then be based on grounds other than the expressive power of the notation. A notation is clearly not acceptable whatever its expressive power if it is more difficult to use than, say, writing the UI in a general-purpose high-level language. The notation must be perspicacious, making evident the nature of the possible dialogues, including what user actions are allowable and the effects of those actions on the system. It should assist in the analysis of the described dialogue for possible inconsistencies and ambiguities. It should be possible to make changes to the specification without introducing unwanted side-effects.

Not mentioned by Wasserman, but also of importance, is the presence of support for generally desirable dialogue styles and techniques. This may be simple "syntactic sugar" added to the notation, but may also include features of the UI model -- mirrored by the notation and underlying UIMS run-time mechanisms -- which facilitate interactive dialogue design. Interactive styles and techniques are likely to change as new applications are developed, new I/O devices become

available, and interactive "fashions" change. Thus, it is impossible to exhaustively enumerate the syntactic sugar and special features which are wanted. Nevertheless, one can examine some of those features made available by current UI specification systems which are general in nature and should be present in any "ideal" UI specification system.

Among the features which will be considered are:

i. Global Commands

A global command is one which is valid at all times during a dialogue. A sub-global command is one which is available at all points in some specifiable part of the dialogue. At one extreme is the "modeless" dialogue in which all commands are global. At the other extreme, for example, is the hierarchical menu-based system in which only a (small) subset of all the possible commands is valid at any one time, i.e., there are no globals.

Dialogues which allow complex graphical interaction are particularly in need of a global command provision since a single "command" (syntactic unit or interactive sentence) may be internally quite complex and part-way through the issuing of the complex command the user may want to issue a command, with return to the current point in the dialogue, like a subroutine call. This is termed a re-entry command (Kasik, 1982, p. 104) For example, while placing components in a circuit diagram, one may want to zoom the view of the diagram to increase the accuracy of the placement task. Such a zoom command would have to be carried out without affecting the state of the placement task. That is, the placement task needs to be suspended while the zoom is executed, followed by a return to the placement task.

Global re-entry commands could be specified at each point where they are valid. However, their global nature means that the number of points where they must be specified will be very great, resulting in a massive specification task and an unreadable specification. What is wanted is a method to specify the scope of validity of these commands without specifying all the points in the dialogue where they are valid.

ii. Cancel/Selective Retreat/Undo Commands

It should be possible to cancel, or escape from, a current command with a return to a previous point in the dialogue. This must include cleaning up prompts, feedback and perhaps changes to the dialogue environment. A selective retreat is similar to a cancel, but to a user-determined point in the dialogue. Undo, unlike cancel and selective retreat, must handle semantic (application)

consequences of the action. It is clearly more problematic for handling within the UI itself, since there may not be simple inverses of the application actions. However, some facility to deal with undo is desirable.

iii. **Unordered command and data entry**

When, say, entering parameters to a command, the order of those parameters may be of no consequence from the user's point of view. Specifying such unordered entry can be difficult if every combination must be listed; there may unacceptable combinatorial explosion. Thus, a notational feature to allow the marking of a set of inputs as having no significant ordering is wanted.

iv. **Default values**

It is often useful to specify default values for both commands and data values. If there are standard values for parameters, it should be possible for the system to supply these, rather than the user, as long as the user can alter the defaults by replacement. Furthermore, once a user has set values of, say, parameters, the UI should be able to retain these values for future use so that they needn't be re-specified by the user. These are called "dynamic defaults" (Apperly & Spence 1983, pp. 783-4).

v. **Data validation**

One of the useful functions of the UI is to carry out validation of data entered prior to passing it to the application. It should be possible to specify within the UI the range of acceptable values of data, along with facilities for prompting the user for re-entry of data when necessary.

vi. **Macros**

When a sequence of commands must be re-used, it should be possible for the user to specify these, along with a mechanism for the UI to use the specified set rather than user-supplied individual commands. This may be done by means of the dynamic defaults, where user-supplied commands become the default and are executed by the system when appropriate until over-ridden by the user. Alternatively, a macro facility may be provided, allowing commands to be supplied from a named file rather than from physical input devices.

To summarise, the UI specification notations to be discussed below will be examined with respect to their:

i. expressive power,

ii. provision of special features for interactive systems

iii. comprehensibility and ease of use.

3.3 Grammar-based specification

In so far as a dialogue may be viewed as a sequence of user-generated (and possibly system-generated) input tokens, it is possible to define the UI by means of context-free grammar. Tokens presented to the control component are parsed using the dialogue grammar which attempts to recognise syntactically acceptable sequences and which, when such sequences or patterns are found, executes application and/or UIMS actions.

The grammar model takes a dialogue to be "a hierarchy of action patterns at various logical levels." (Hanau and Lenorovitz, 1980, p. 274) For example, consider the following user actions which occur in an interactive drawing program:

i. move cursor over "draw" box in menu and click mouse button (new pop-up menu appears with options for the draw command),

ii. move cursor over "rectangle" box in new menu and click mouse button.

Each of these actions can be viewed as an instance of the pattern "select from menu". At a higher logical level they constitute together the pattern "draw rectangle". At an even higher level this may be part of a more generalised task, perhaps "design circuit".

Viewed top-down, dialogues according to the grammar model can be progressively expanded into sequences of lower level actions until reaching primitive input events generated by physical input devices or by logical devices in a separate presentation system. Execution of a dialogue consists of attempting to recognise grammatically acceptable patterns in the input, combining these into progressively more abstract syntactic units. The equivalent of lexical tokens are the events generated by physical or logical input devices and the application actions are the semantic associate of syntactic units.

Clearly, this model takes user input as analogous to the source code of a program, the UIMS run-time system as the analogue of an interpreter (not a compiler in that the dialogue "program" is executed as it is presented to the UIMS) and the UI specification as analogous to the grammar of the language.

A grammar-based approach to UI specification has a number of advantages, the most important of which are:

i. knowledge of the construction of grammars and their parsers for non-interactive languages can be employed in dialogue design,

ii. the linguistic model of UIMS discussed in the previous chapter identifies separable components for presentation, control and interface to the application. These three components correspond to the lexical, syntactic and semantic aspects of formal languages. Thus the linguistic model of UIMSs and the grammar model of UI specification fit together neatly,

iii. certain methods of grammar description (e.g., BNF) encourage a top-down approach to UI design and produce a human-readable description of the UI,

iv. systems for the automatic construction of parsers provide a means of automatically generating UIs from a description of the dialogue "grammar".

In spite of these advantages, it must be borne in mind that the grammar model has been borrowed from an environment different from interactive UI construction, viz., the design and production of "static" languages. Therefore, the tools and techniques borrowed from static language design must be modified in various ways to accommodate the needs of interactive dialogues.

First, texts in static languages exist in their entirety before processing (e.g., for translation or compilation) begins. The inputs in a dialogue, however, may be determined in part by how the user reacts to system responses to earlier inputs. This results in more complex possible sequences of inputs and system responses than is the case with static languages, including user interruption of one "statement" or command followed by the start of another. Such an interruption may be a rejection of the earlier command (cancellation) or it may be a request to suspend the command, followed by subsequent reactivation after intermediating actions. As Jacob states, "a specification of an interactive language must capture more information about the behaviour of the system than one for a static language." (Jacob, 1983a, p. 4)

Second, static language descriptions conventionally treat the source as textual in nature, while dialogues must cater for a range of input actions using a variety of devices other than the keyboard, including tablet, mouse and lightpen input of cursor location, function keys and so on.

A grammar-based dialogue description which treats its primitive tokens as identical to user-generated input actions at physical device level, as a static language grammar might treat characters in the input stream as its tokens, may obscure the correspondence of cognitively significant user tasks with application actions. For example, such a view of interactive tokens would have to distinguish a function key press from a mouse button click, even though there is no difference between them from the syntactic or semantic point of view. Jacob suggests treating values generated by virtual input devices as the terminal tokens, leaving the detail of how physical device level actions generate these tokens to the specification of the virtual devices (Jacob, 1983a, p. 5) This approach fits the linguistic UIMS model in that virtual devices can be seen as part of the presentation component with the values associated with input tokens passed to the control component across the presentation/control interface. Nevertheless, the nature of the terminal symbols in a grammar-based dialogue is not clear cut. Various grammar-based systems have dealt with this issue in a number of ways, as shall be described below.

3.3.1 BNF-type specification

A Backus-Naur Form (BNF) description of a language consists of a set of production rules. Each rule consists of a nonterminal symbol on the left-hand side which is replaceable by (defined by) the symbols occurring on the right-hand side of the rule. Terminal symbols may occur on the right-hand, but not the left-hand, side of a rule; that is, they may replace, but not be replaced by, other symbols. Terminals consititute the tokens of the language. An expression or sentence (i.e., string of tokens) is grammatical if and only if it can be derived from a distinguished starting symbol via a sequence of applications of the production rules of the language.

The BNF-type notations for UI specification have generally been altered from the pure BNF described above to make them more suitable for describing interactive dialogues. The main addition has been the association of an action with a production rule, such that the action is carried out when the rule is satisfied. These "actions can include prompts and other feedback to the user and also actions that involve processing by the rest of the system." (Jacob, 1983a, p. 5) Some systems introduce an environment, i.e., variables, which can be altered via such associated actions. Also, the definition of production rules requires operations over the symbols (terminals and nonterminals) appearing on the right-hand side. These operations may include sequence,

conjunction and alternation as well as iteration and conditionals. The richness of the set of operations available largely determines the expressiveness of the notation.

3.3.2 The Interactive Dialogue Synthesizer (IDS)

The Interactive Dialogue Synthesizer, or IDS, (Hanau and Lenorovitz, 1980) is an example of a UI specification notation using BNF-type productions to specify the dialogue. In this system production rules are written thus:

<operation> = <poly-line> ! <circle> ! <rectangle>

where expressions in angled brackets are nonterminals, '=' separates the two sides of the rule and '!' indicates alternative productions. Several enhancements to standard BNF are used, including:

i. Repetition, which allows repeated parsing of a bracketed set of symbols.

<draw> = <initialise> (<operation>) (1 1000) <terminate>

In this example, <operation> would be repeated up to 1000 times, unless <operation> failed, in which case the parser would move to <terminate>. Failure is itself a syntactic element which may appear as a symbol in a production (Hanau and Lenorovitz, 1980, p. 274)

ii. Semantic Actions.

A string enclosed by single quotes indicates an application action to be carried out. It may appear on the right-hand side of a production. No parameter-passing mechanism is supported, but a set of registers and a stack are available for holding values generated by physical devices and various contextual variables. These may be accessed by application routines.

There is no clear distinction between reading interactive input and executing application actions. Both occur indiscriminately as semantic actions associated with nonterminals. For example, consider these two productions:

<circle> = "o" <locator> 'mark-center' <locator> 'compute radius;draw-circle'

<locator> = ("?" 'poll-lpen' ! "=") <e-list>

(Hanau and Lenorvitz, 1980, p. 274)

The semantic action 'poll-lpen' is clearly part of the presentation system while 'compute-radius' is an application action. A cleaner design would differentiate among prompting, reading input, generating feedback and executing application modules. In IDS, all four are lumped together as

semantic actions.

IDS allows full access to physical devices. The construction of higher-level logical devices is the responsibility of the dialogue designer and all such detail appears in the UI specification. In other words, the division bewteen presentation and control components occurs at the physical device level. Hanau and Lenorovitz see this as an advantage:

"The level of detail at which semantics is specified in this kind of system is highly variable, ranging from selection among a limited set of very high-level, application-oriented actions, to more flexible, more general, but lower-level actions." (Hanau and Lenorvitz, 1980, p. 277)

While this flexibility and control over every level of the dialogue are desirable features of a UI specification system, it does not follow that all dialogue specification need be carried out as part of the control component.

No facilities are included in IDS to allow escape from command sequences. Consideration, but no solution, is given to the semantic difficulties of backup from parsing dead-ends, viz., it may be difficult or impossible to undo executed semantic actions (Hanau and Lenorovitz, 1980, p. 276). This is a problem for any interactive dialogue system. However, the lack of tools for defining the syntax of command escape means that escape must be hand-coded where needed.

Also, global and subglobals commands are not considered. Such commands would have to be entered as alternative productions in all relevant rules. Not only is this tedious, but it produces an unnecessarily confusing and complex specification. The hierarchical nature of BNF production rules lends itself to the exportation of attributes and alternatives from higher to lower level productions. The possibilities of this characteristic of the production rule approach are not exploited by IDS.

3.3.3 SYNGRAPH

The SYNGRAPH system is a user interface generator like IDS which also uses a BNF type of grammar to specify the dialogue. It is more sophisticated than IDS, however, providing mechanisms to handle many of the deficiencies of IDS.

As in IDS, concatentation (indicated by sequence), alternation (represented by '|') and

repetition (shown by curly braces) are all allowed. Some tokens generated by the presentation component return values and these must be immediately followed by parameters to hold the value(s) returned. Such values may be passed to application actions, which are expressed as Pascal code, delimited by '(#' and '#)'. Finally, information for control of semantic action execution and prompt generation are included in the left-hand side of a production.

Part of a SYNGRAPH specification is shown below:

```
DRAWCOM : heading = (##); [newlevel]
          ::= draw {DRAWINSTR (##) '/' }

DRAWINSTR : heading (##); [rubout]
          vars = (# x1, y1, x2, y2 : real;
                    thiswall : walltype; #);
          ::= wall
              loca (# x1, y1 #)
              b1
              loca (# x2, y2 #)
              b1   (# drawwall(x1,y1,x2,y2)  #)
             |
              window
              picwall (# thiswall  #)
              loca (# x1,y1 #)
              b1
              width  (#  w  #)
              (# drawwindow (thiswall, x1,y1,x2,y2)  #)
```

(Olsen, 1983a, p. 114)

In this example, variables declared as local within DRAWINSTR are used for passing positional and pick information to the semantic actions 'drawwall' and 'drawwindow'. Certain expressions (draw, wall, window) refer to selectable menu items. Prompts for these are generated via a special lookahead mechanism, which will be discussed below, while their display and selection are handled at the lexical (presentation) level. 'b1' and 'loca' also refer to input devices, button click and locator respectively. Note that the locator returns x,y values which are assigned to the variables x1 and y1 or x2 and y2. Semantic actions are Pascal code, in this case procedure names; variables and parameters are also in Pascal.

Unlike IDS, SYNGRAPH includes a separate specification of nonterminals which provides a set of virtual input devices as tokens for the grammar. They are "categorized by the type of semantic information they return, the kind of prompting they require, the echo feedback required when selected and the method of acquiring input from the token." (Olsen and Dempsey, 1983a, p. 113) It is stated that user-defined (or better, designer-defined) virtual devices are envisaged as a

further development of the system (Olsen and Dempsey, 1983a, p. 113).

Although prompting, acquisition of input and low-level feedback are the concern of the lexical component, and do not appear explicitly in the syntactical specification of the dialogue, the state of the parse does affect the input devices. For example, menu items to be displayed and input devices to be enabled depend upon the current state of the dialogue. This is handled automatically by the parser as follows: when encountering a non-terminal declared as starting a new level of discourse (see DRAWCOM in the example above), the parser looks ahead to display and activate those menu items in succeeding non-terminals which are not themselves headers of new discourse levels. Thus, the menu item "draw" in DRAWCOM would be enabled, as would "wall" and "window" in DRAWINSTR since DRAWINSTR does not start a new level of discourse. Similarly, all subsumed devices required for input in the current level of discourse are enabled, in this case "loca" and "b1". Appropriate declaration of a nonterminal as a new discourse level is the responsibility of the dialogue designer.

A generalisation of this approach (Olsen, 1984a) views the UIMS run-time system as an interactive push-down automaton (examined in Section 3.3.3) , of which SYNGRAPH is an example. In this later development, input is received from input devices, which may be logical, virtual or actual. A device handler looks after enabling and disabling of devices, as well as prompting and low-level feedback. It also resolves conflicts of reference (e.g., in pick actions) via ordering the priority of devices in the order of enabling.

The IPDA design keeps the division between presentation and control clear and the interface between lexical and syntactic components clean. However, some features of the design limit the construction of adequate dialogues. While devices may pass values to the control component, both in event and sample modes, the only effect the control component can have on the devices is to make enable/disable or acquire/release calls to the device handlers (Olsen, 1984a, p. 180) Thus, application-related prompt or feedback must either be built into the virtual devices (in Olsen's example of lexical specification, devices include an application-independent locator and application-dependent icons for menu selection (Olsen and Dempsey, 1983a, p. 113)) or be carried out via application-dependent semantic routines. Although not discussed, higher-level feedback regarding the state of the dialogue parse must also be carried out by semantic routines which alone can produce output. This problem arises because the dialogue control mechanism,

which interprets the dialogue specification, cannot dynamically configure or control input devices nor can it generate or control output. Underlying the problem is a system limitation of SYNGRAPH which allows only one-way value-passing from the lexical to the syntactic component and the placement of output generation (apart from input echoing and low-level feedback) in the semantic routines.

Systems like IDS and SYNGRAPH which employ a top-down parse of a grammatical specification of a dialogue run into a problem due to possible backing up of a parse. In a standard compiler, a parse may continue down a path which eventually fails, subsequently backing up to attempt to traverse another path of the parse tree. However, in an interactive system, this is unacceptable since (i) semantic actions may have been executed which cannot be undone and (2) feedback may have been presented to the user which cannot be recalled. As Hanau and Lenorovitz state:

"While a compiler may reasonably "pop" symbols added to a symbol table on such a false path, an interactive system can hardly advise the user to "ignore all messages since...." (Hanau and Lenorovitz, 1980, p. 276)

Hanau and Lenorovitz suggest that this problem occurs solely because of the association of semantic actions, including feedback, with dialogue syntax, and that it is the dialogue designer's responsibility to ensure that such backing up either cannot occur or does not cause problems if it does. Olsen, however, points out an additional cause of such problems. Sampled input cannot be used to select transitions, nor does the occurrence of a sampled input token indicate when the sampling should occur. From the point of view of the parse, sampled input acts like semantic actions. (Olsen and Dempsey, 1983a, p. 114)

Olsen's solution is to pre-process the specification in order to additionally associate a "trigger" event with each transition which has only sampled input or a semantic action associated with it. These triggers are event inputs which occur subsequently in the parse tree and which then determine the transition to be taken and the time at which sampling should take place (Olsen & Dempsey, 1983a, pp. 114-115). The details of the method by which this is done will be examined in Sec. 3.4.4 which discusses Olsen's IPDA. It should be noted that in SYNGRAPH pick inputs

are also treated like sampled inputs since a sampled tablet is used to implement picking. Thus, each pick must have an associated trigger and, along with tablet/trigger pairs used for inputting points, the following event inputs are associated with earlier sampled tablet input as triggers. This, however, is an implementation detail of SYNGRAPH; if picking were implemented as a virtual device generating an input event, the trigger association would not be necessary. It is thus still the case that the basic division is between event input which can label valid transitions and sampled input which cannot.

Having produced a specification such that every transition has an associated trigger, it is still possible that a parse will "fail" due to user-generated rejection of the path. While IDS provides no method to handle this, SYNGRAPH has two mechanisms built into the grammar. First, a non-terminal may be declared to support rubout (see DRAWINSTR in the example above). Such nonterminals are parsed twice: first skipping semantic actions and second, when fully recognised, only carrying out the associated actions. This is analogous to processing text a line at a time, only processing the input string when a newline character is encountered. (Olsen and Dempsey, 1983a, p. 118) Prior to complete recognition of the non-terminal, it is possible to cancel the nonterminal without requiring semantic "undo". As attractive as this facility may appear, it will not handle the common interactive situation in which the semantic actions must take place to generate feedback necessary for the user to proceed with a given task.

Also, cancellation of a non-terminal is supported via a special "escape" clause associated with that nonterminal. When a cancel token occurs in the input, the escape clause for this nonterminal, if any, is parsed, and the parse then returns to the previous higher-level nonterminal. This process continues until an escape clause is encountered which includes a 'handled' token. This mechanism provides the sort of special feature useful for dialogue designers in specifying system behaviour for a commonly-encountered situation in interactive dialogues. The 'handled' mechanism in particular allows escape to return control to any previous point in the dialogue, as specified by the designer. However, no user control is allowed over the degree of "retreat" in cancellation.

The rubout and cancel facilities are global commands which are built into the system. Along with help and device reset, they are special actions associated with special tokens which need not be explicitly included in the definition of a nonterminal (Olsen and Dempsey, 1983a, p. 117) Rather,

these tokens are implicitly included in every production and hence are valid transitions at all points in the dialogue. However, recognition of one of these globals does not affect the state of the parse; after carrying out the global action, the parse continues at the point at which the global token was encountered. In a linguistically-based dialogue, global commands must either be specified in every production or must, as with SYNGRAPH, be treated as special tokens handled by the parser and removed from the input so that they do not appear as invalid tokens to the nonterminals. While the SYNGRAPH approach is clearly preferable to the first alternative, only a handful of globals are supported and these are pre-determined by the system, not by the UI specification. Furthermore, there is no support for sub-globals, that is, nonterminals valid at all times during the parse of another specified nonterminal.

3.3.4 Input-Output Tools

Input-Output Tools is a device model UIMS which uses a grammar-based spcification for the UI. As described in Section 2.3.2, there are five parts to an Input-Output Tool, the init section, the input rule, the cleanup section, the tool body and the output rule. Of these, the syntax of the tool is found in the input rule, which consists of an expression made up of other input-output tools (recursion is not allowed) which are joined together by means of a set of operators. The occurrence of tools within input rules results in a hierarchical structuring of rules similar to the structuring of production rules in a BNF grammar. The similarity is further confirmed by the statement that:

"an entire program [the UI] is defined as an input-output tool or, to be more precise, as a tree of tools, the root of which may be considered as the name of the tool program. Starting a tool program is equivalent to activating the highest level input-output tool. A single parser controls the tool program." (van den Bos et al, 1983, p. 250)

Thus, the set of tools are like a set of production rules, with the input rules forming the equivalent of the BNF definition of a nonterminal.

The input rule expressions have a rich set of operators available by which to join the consituent tools. These include, as with IDS and SYNGRAPH, sequence (';'), alternation ('+') and inifinite

iteration ('$'). Repetition with a stated terminator is defined as follows:

T1 * T2

where T1 is repeated until T2 is satisfied. An alternative allows the number of repetitions to be specified as a range:

T1*i..k

would indicate a repetition of at least i and at most k invocations of tool T1. The "significance" operator allows the tool in the left part of the expression to be terminated when another tool is satisfied. For example, in

T1 ? T2

T1 is considered successfully completed at any point when T2 occurs. If the input rule were "y;e;s;return" it would be matched by "y;return"and "y;e;return" as well. This is only possible, however, if the truncated input is not ambiguous with respect to other alternatives; it must uniquely determine the input rule (van den Bos et al, 1983, p. 249).

The most powerful operator, which gives considerably greater expressive power compared to the other grammar-based systems, is the "interleaving" operator. The input rule

T1 & T2 & T3

would be satisfied by T1, T2 and T3 but in any order. This allows the system to simulate concurrently activated tools and to specify, say, the collection of parameter values in arbitrary order.

Also, unlike the previous systems examined, pre- and post-tests can be carried out on tools in an expression. This results in the possibility of conditional activation of tools. The effect of a pre-test is to make the tool associated with the test an active component of the rule only if the test is successful. A post-test, the value of which may be affected by the tool with which it is associated, results in its associated tool failing if the test fails. For example, in

bool: T1 ; T2: ch = "a"

the value of the boolean variable bool would determine whether T1 was activated. If bool were false, the input expression would be equivalent to:

T2: ch = "a"

The post-test for T2 checks the value of the char variable ch. after the termination of T2. If ch is not "a" then T2 would fail. In the event that both tests fail, the effect of pre- and post-tests is

different. If pre-tests results in a failure of the entire expression (i.e., if it becomes an empty expression) the expression simply fails, with possible consequences further up the parse tree. However, if post-tests fail and no alternatives exist, then "the failing tool(s) and the descendants, following cleanup and termination, are activated again." (van den Bos et al, 1983, p. 251). This difference in the effects of the two tests seems to be unfortunate. Particularly the automatic re-activation of the expression in the case of the post-test might well cause confusion for the user unless appropriate feedback is built into the cleanup part of the tool, and this may be difficult to do if the expression is a complex one, with several possible alternative post-tested tools. A similar, but more general problem arises from the fact that at any point in the parse, if the input token cannot be matched to the current expression, it is rejected and the parse attempted again (van den Bos et al, 1983, p. 250). This could also cause user disorientation if proper feedback is not provided.

Nevertheless, the existence of conditional activation of tools gives much more expressive power than the previous grammar-based specifications examined. The post-test facility allows for validation of the range of input values presented and the pre-test facility allows tools to be activated conditionally on user-action resulting in the change of the value of the pre-test condition. However, the only expressions which appear allowable in tests are tests of equality and set membership of variables local to the tools. The dynamic change of state resulting from changes to application data structures should also be able to activate or deactivate tools in the input rule. In general, what is wanted is as rich a set of operators and operands in the conditions as exist in the input rules themselves.

The categorisation of all other Input-Output Tool characteristics into four separate components gives an easier to read specification than the undifferentiated actions mixed into the nonterminal definitions as in IDS and SYNGRAPH. Thus, in the Input-Output Tool, all semantic actions are specified in a distinct tool body. Prompts and the initialisation of local variables occurs in the init section. The separation of solely prompt information as in SYNGRAPH's nonterminal heading is perhaps preferable, although on this point there is little to choose between the two approaches.

Neither of the other notations treats the problem of the clean termination of non-matched alternatives when one of several alternatives is successful. This task, carried out by the cleanup section of an Input-Output Tool, is particularly useful for removing prompts which are no longer

valid.

Like SYNGRAPH the Input-Output Tool system makes provision for the cancellation of commands. This is done by means of a special **escape** tool. The escape tool may be (re)defined at any level in the tool hierarchy, but does not appear in any input rule. If its own input rule is matched, it causes the currently active tools to excute their cleanup and terminate, followed by a re-activation of the tool within which the escape tool is defined. No single mechanism is provided for multilevel escape as in SYNGRAPH, although "in a single IOT several escape tools may be active simultaneously at different levels. In this way one may fine-tune the amount of backing-up." (van den Bos et al, 1983, p. 252)

The escape tool is the only such globally available tool. Any other global or sub-global tools would have to be present as alternatives in all appropriate input expressions. The same mechanism used for escape could be expanded, however, to provide such global tools. Given the hierarchical nature of the system, implementation of sub-globals available at all points below a given level would be relatively straightforward.

The output rule allows for the passing of parameters from one level to another without the need for global registers as in IDS or global variables as in SYNGRAPH. To show the different methods of parameter passing and to generally illustrate the differences between Input-Output Tool specification and SYNGRAPH, a sample specification from each is given here, based on a rubber-band line-drawing facility:

SYNGRAPH Specification

```
RUBBERBAND : heading = (##);
        vars = (# sx,sy,fx,fy : real;
                    line_present : boolean #);
        ::= (#line_present := false #)
             STARTPT
             ENDPT
              (# sendline(sx,sy,fx,fy) #);

STARTPT : heading = (##);
          ::=
           loca (# sx, sy #)
           button

        escape ::= (# undo_line #) handled;
```

```
ENDPT : heading = (##);
        ::=
            {loca (#fx,fy#)
            clocktick
           (# if line_present then delete_line;
                 move(sx,sy);
                 draw(fx,fy);
                 line_present:= true #)
         button (# setend (fx,fy #) }

        escape ::= (#undo_line #) handled ;
```

Input-Output Tools Specification

```
tool Rubberband  =

        input Beginpoint (real xb,yb); Endpoint(real xe,ye) end
        output (real uxb,uyb,uxe,uye) end

        bool line_present;

        tool Beginpoint = input Locator(real x,y)*1..$Button[1] end
            output(x,y) end
        end

        tool Endpoint= input Band(real x,y)*1..$Button[2] end
            output(x,y) end

            tool Band=input Clock[0..1];Locator(real x,y) end
                output (x,y) end
                if line_present then delete_line fi;
                move(xb,yb);draw(x,y); line_present := true
            end
            remove_cursor
        end

        init line_present := false;
              insert_cursor (0,0)
        end

        cleanup remove_cursor end

end
```

(van den Bos et al, 1983, p. 254)

In order to make the two specifications comparable, it was necessary to invent a clock input primitive for SYNGRAPH which would serve as a trigger for the locator sampling and the recurrent redrawing of the line in ENDPT. Without it, the subsequent button press would become the trigger for the locator sampling; thus, the line would only be drawn once, viz., when the button was pressed. Also, some additional parts of the IOT specification have been omitted to simplify the comparison.

The parameters to Beginpoint and Endpoint in the IOT specification clearly indicate the number and type of data expected to be returned from these tools. When each of these tools is successfully completed, their output parameters are made available to the calling tool Rubberband via the parameters listed in the invocation. This is much clearer than the use of globals in the SYNGRAPH specification; just by examining the nonterminal Rubberband it is not possible to determine the information which will be provided by STARTPT and ENDPT. Nevertheless, the IOT only allows passing of information in one direction -- from a tool to its calling tool via the output rule. The additional ability to pass information from the calling tool to the called tool would be preferable to the present need to use globals (e.g., note the occurrence of 'xb' and 'yb' in the tool body of tool Band).

An examination of the similarity of structure of these two specifications reveals the extent to which the IOT model fits into the grammar-based specification category. What it does is to structure and amplify more fully those features of the specification notation which are peculiar to the needs of interactive systems. Some degree of concurrency is also supported. As has been noted, some of the extensions to the specification system do not go far enough. Although it is called an Input-Output Tool model, no provision is made for control of application generated output, nor does it appear that the application can generate events at the level of the primitive Input-Output Tools, apart from basic system generated events such as the Clock Tool. Furthermore, any deficiencies in either the grammar-based approach to specification in principle or in the device model of UIMSs still remain.

3.4 State Transition Network Specification

A State Transition Network can be used to define the grammar of a language. It is thus an alternative to BNF and, not surprisingly, has also been used as the basis for the specification of interactive systems. Indeed, the use of State Transition Networks to define interactive systems has a longer history than the use of BNF (Newman, 1968; Parnas, 1969).

The simplest state transition network is one which can define a finite-state automaton. It consists of a set of states and a set of transitions, each of which joins two states. The network can be represented as a directed graph, with each state represented by a node (conventionally a circle) and each transition by an arc (conventionally a directed line). There is one initial state and

one or more terminal states, joined to zero or more other states. Each arc is labelled with a token; transition from one state to another takes place on the occurrence of the token in the input stream. When used for dialogue specification, each token would represent a user-generated input.

However, just as with BNF, extensions and modifications have been made to the state transition network notation to make it more suitable for dialogue specification. Some of these modifications have been "borrowed" from notations already developed for the definition of static languages. Two such extensions are recursive and augmented transition networks.

A Recursive Transition Network, or RTN, (Woods, 1970) is a transition network in which the arcs may be labelled with the names of states as well as with tokens. By so doing, complex networks can be broken up into named subnetworks and it is possible to make these named subnetworks correspond to meaningful segments of the diagram. This is similar to the use of nonterminals in a BNF definition.

An Augmented Transition Network, or ATN, (Woods, 1970) adds two new features to the basic transition network model. First, arcs can be labelled with predicate evaluations as conditions for the successful traversal of the arc. These predicates may include the testing of the values of registers. Also, if the network is also an RTN, the subnetworks called may also return values which are tested via conditions. Second, actions can be associated with the transitions, such that successful traversal results in the execution of the action.

Cockton (1985) also describes a Dialogue Augmented Transition Network which is basically an ATN with a rich set of input tokens, including input from graphical devices, transition conditions which can operate on environment data objects, the ability to produce output via dialogue actions and a separate application process or program "which executes dialogue free application actions." (Cockton, 1985, p.140) All of the current UIMS which employ a transition network notation for dialogue specification either fit into this category or at least approach it.

Cockton also notes that actions, including application procedure calls, environment manipulation and passing data to the presentation system, may be associated with:

i. successful transitions (the usual definition of an ATN),

ii. condition evaluation (as side effects), or

iii. a state.

Where actions are restricted to transitions only (similar to IOT where actions can only occur in the tool body and side effects of conditional tests are not allowed), the resultant notation is called by Cockton a Transition Procedure Transition Network. A network allowing actions only at states is called a State Procedure Transition Network (Cockton, 1985, p. 141). In principle, there seems to be little difference between actions associated with transitions and those associated with states. Cockton points out, however, that where actions are associated with transitions only, they can be treated, along with the transition condition, as arc labels, thus allowing analysis using path algebra techniques (Cockton, 1985, p. 141; Alty, 1984b) If the actions in a State Procedure Transition Network occur only on entry to the state and if there is no other system effect from entering a state, it would appear possible to translate a State Procedure Transition Network into a Transition Procedure Transition Network by propogating the actions at the states back onto the arcs. However, another problem arising from the occurrence of actions at several points in the network (e.g., condition side-effects, transitions and states) is that it may be difficult to determine from the resultant network the ordering of actions (Jacob, 1983a, p. 13).

If a rich set of input tokens, including graphical input devices, are added to an ATN and actions can include both application actions and the generation of output, a powerful notation would be available by which to specify user interfaces. It is worth comparing such a notation to the Input-Output Tool model. Note that IOT allows conditional tests on constituent tools in its input rules, which is similar to the conditions on arcs in an ATN. The actions carried out in a tool body are analogous to the actions associated with transition labels and the value(s) returned by successfully terminating tools are like the values returned by subnetworks. In fact, the similarlities between fully developed BNF and transition network notations are greater than their differences, at least in terms of expressive power.

An RTN is equivalent in expressive power to a context-free grammar, and both an ATN and BNF with associated actions and environmental data objects are equivalent to a Turing machine (Jacob, 1983a, p. 16). There seems, then, to be no grounds for choosing between them in terms of expressive power. Nevertheless, it does not follow that a BNF description and a transition network description of a dialogue will be equivalent in their ease of use or comprehensibility.

Guest's observations of programmers' reactions to a BNF-based specification of SYNICS dialogues (see Section 2.3.3 and Section 3.4.1 below) seems to confirm this view:

"When SYNICS was offered to programmers, who already had a set task to be completed, they almost all rejected it and wrote their own routines. When asked why they did not use the tool the most common reply was that "it looked too complex to use"....They all used the argument that they could not understand the input language structure and operation." (Guest, 1982, p. 264)

The problem encountered here is open to a number of explanations. Guest offers the comment that "most programmers tend to think and program in procedural terms, yet a grammar and rules for this type of system are declarative." (Guest, 1982, p. 267) However, it must also be considered that an interactive dialogue is essentially a temporal sequence of actions. This sequencing of system states is not captured in any grammatical description in an easily conceivable way. Jacob suggests that a transition network better models the notion of the options available to a user from a given point in the dialogue (Jacob, 1983a, p. 10). Whatever the reason, transition networks appear easier to use. A transition network specification language, DDL, was added to SYNICS and user reaction to this was positive. "All the work with the dialogue system given to undergraduates was not supported by any documentation, yet they all managed to produce a working system." (Guest, 1982, p. 278)

3.4.1 SYNICS/DDL

The SYNICS system (see Section 2.3.3), developed by the Man-Computer Interface Research Group at Leiceister Polytechnic, is a syntax-directed translator which takes BNF type dialogue specifications and produces output for a table-driver parser. The Dialogue Design Language, or DDL, defines the dialogue in terms of a transition network. A later version, SYNICS2, introduces significant improvements to the system, among them some changes to the specification language. In the following account, SYNICS and DDL will form the basis of the examination, but the alterations introduced by SYNICS2 will also be considered.

DDL defines a dialogue as a set of nodes, each of which defines a possible dialogue event (Edmonds, 1984a, p. 5). Each node consists of :

i. an initial statement, including a unique identifying number

ii. optional statements specifying actions related to output to the display

iii. an input mode, allowing acceptance of a line of input from the terminal or the application program

iv. a set of statements defining possible transitions and the conditions under which those transitions would be taken.

A sample node might look like this:

```
AT NODE 10
    ; This is a SYNICS node.
    ; It will merely transfer control to another node
    ; on the basis of the key pressed.
    REMARK  The above three lines will be displayed when this node
                becomes activated. It will then wait for input from the
            keyboard.
    TO 20 IF '1'
    TO 30 IF '2'
    GO TO 999
```

The node identifier, which may be any number from 1 to 999, is not meaningful and, therefore, not particularly helpful in making the specification understandable. SYNICS2 improves on this by allowing meaningful names to be used; the node identifier would look like this in SYNICS2:

DIALOGUE EVENT <node_name>.

The output statements immediately following the node identifier only allow textual output along with some terminal control statements, such as clear screen, reverse video and so on, since SYNICS only handles textual input and output. These limitations have been removed by SYNICS2 which uses a restricted subset of GKS as its presentation system. This allows GKS logical device and workstation information to be accepted as input via an INQUIRE statement. Also, GKS forms the basis of dialogue output, although there are limitations, e.g., no co-ordinate transformations are allowed (Guest and Edmonds, 1984, pp. 341-2).

In both systems, output actions are associated with the state rather than with transitions. Since all output must occur before input is accepted or the transition conditions are evaluated, such output must be viewed as prompt information. No provision is made within a node for the cancelling of prompts, echoing or feedback; these, along with any cleanup when the node is exited must be handled by other nodes.

The input source statements of SYNICS allows the designer to accept input for evaluation either from the user (the default option) or from the application. This is a particularly nice feature, allowing the application to control the path of the dialogue when desired. Also, it could be seen

as the basis of a facility for using command macros, with input specified as coming from a named file. In SYNICS, there is only one input device, the keyboard, but in SYNICS2 the source statement can also specify the device from which input is accepted (Guest and Edmonds, 1984, p. 341)

The list of possible transitions is made up of statements expressing either conditional or unconditional transition. Conditional transitions are based on matching the input with a rule. The rules, which may be defined in a separate general rule block, are expressed in a BNF-type language allowing the definition of nonterminals based on input tokens and the concatenation and alternation operators. This is a much less well-developed notation than those available in any of the BNF-based specifications described earlier and limits the type of transition conditions which may be expressed. Furthermore, these conditions cannot include any contextual variables declared within the dialogue itself.

Transitions are always to named nodes. There is a facility for calling a named node with subsequent return. This CALL statement must be the only statement in the node in which it appears and can have a list of up to 20 parameters holding values from either input, the application or string literals. Note that one cannot call named subnetworks, but only named nodes. Return is by means of a RETURN clause which may appear by itself in its own node or as part of a transition rule.

Each transition rule may have associated with it both a text string to be passed to the application program and a statement assigning values to contextual variables; these are executed if the transition is successful. This is the fundamental mechanism for passing information to the application. The requirement that only textual information can be passed seems overly restrictive. This is augmented in SYNICS2 by a by-pass pipe mechanism which allows large amounts of data (e.g., bitmaps) to be passed directly from the presentation system to the application without any processing by dialogue control. Also, it would appear that numerical values can be passed to applications in SYNICS2 (Edmonds, 1984b, p. 55; see example below) Still, a more general facility for passing data of a range of types to the application would appear to be more desirable. Furthermore, the separation of output actions executed on entry to the state on the one hand, and application actions and environment manipulation on successful transition on the other is an unnecessary complication which makes the system less perspicuous.

Unlike Input-Output Tools, only one node can be "active" at a time, so there is no concurrency, real or simulated. The association of output actions with states means that only one such set of actions will be executed at a time (e.g., only one prompt available at a time), hence there is no need for a cleanup routine.

There is a special "help" statement in SYNICS2 which can be placed in any node. This will output a textual string if the "help" token is encountered, with a return to the node in which the help statement is defined. Such help statements cannot be defined for a set of nodes. Furthermore, there is no provision for escape or globally available commands.

To illustrate some of these points and to allow comparison with the grammar-based specifications, a sample SYNICS2 specification is presented, again based on a rubber-band line-drawing factility:

```
DIALOGUE EVENT initialise
        CLEAR SCREEN
        ENABLE CURSOR
        ENTER EVENT start
END EVENT initialise

DIALOGUE EVENT start
        ENTER EVENT line IF
            BUTTON ["start:"CURSX,CURSY]
                <SX=CURSX, SY=CURSY>
END EVENT start

DIALOGUE EVENT line
        CLEAR SCREEN
        LINE(SY,SY,CURSX,CURSY)

            RETURN IF
                BUTTON ["finish:", CURSX,CURSY]
            ENTER EVENT end IF
                TEXT "cut" ["cancel"]
END EVENT line

DIALOGUE EVENT end
        CLEAR SCREEN
        RETURN
END EVENT end
```

(Edmonds, 1984b, p. 55)

Note that the returns in event **line** and event **end** are to a dialogue event not here specified. The lack of a built-in escape facility means that the application must be invoked to cancel the interaction on receipt of the textual input "cut" in event **line**. Cancel is not allowed at any other point simply because it has not been specified. Since parameters can only be passed via sub-node calls, there is no way of passing the start points from event **start** to event **line**; it is

done by the global variables SX and SY.

The re-drawing of the line in event line until the button is pressed is accomplished automatically because in SYNICS if no transition is successful, the node is re-executed. Since output actions may be present, this means that the output will be re-generated. This may be desirable in some cases, but certainly not all. Some way of making the output generation conditional would be preferable, as would be the case if all output were generated on transition traversal rather than entry to a state.

3.4.2 The CONNECT System

The CONNECT system from Strathclyde University employs an ATN model of the UI. All input and output are textual, and application actions are BASIC programs which are loaded prior to run-time or can be loaded into six pre-designated overlay areas at run-time. Passing of values between the UI and application routines is via the manipulation of global variables accessible to both (Alty and Brooks, 1985, pp. 341-2). Thus, user/UI and UI application communication are somewhat primitive and similar to the original SYNICS, although SYNICS supports parameter passing between UI and application even in its early version.

Unlike SYNICS, where all nodes have the same basic structure, CONNECT nodes are differentiated into several types. These are:

communicator node - which prompt for and accept user input (a newline terminated string)

task node - an appplication action

subnet - which calls named subnet, with return to the calling node on exit

termination node - end node; in subnet, causes return

assister node - assists in transforming transition conditions; may produce output

(Alty & Brooks, 1985, p. 336)

The division of nodes into types arises partly from the placement of actions at nodes, rather than as a consequence of arc transition as in SYNICS. While the SYNICS approach of linking actions to condition evaluation may be overly "complex and indecomposible" (Cockton, 1985, p. 143), the creation of a variety of node types is unnecessarily complex and restrictive. Even with application actions located at states, there is no need to restrict the possible actions to application action or prompt or output.

A major innovation of CONNECT is its provision for switching transitions on or off dynamically at

run-time. Transition conditions are boolean predicates on the value of environment variables. Up to three variables may be joined by logical alternation (Alty and Brooks, 1985, p. 336) a strangely restrictive limitation on the construction of conditions.

These environment variables are global and may be examined and set by application action, user input or by a production system. The production system consists of set of production rules which can cause setting of environment variables if their condition is satisfied. The conditions of production rules are similar to the transition conditions. The production system is called by the run-time executor prior to user input and can thus change the transition from the current node based on the current network state, including previous path traversal. At its simplest, an arc with the condition TXT1$="on" can have that transition switched off by the production system (or indeed an application routine) changing the value of TXT1$ to "off".

Although this is quite a general facility, it is designed to allow switching between parallel disjoint networks with switchable transitions between them at particular points in the dialogue. These parallel networks might be alternative interfaces to the same application. An example implementation of CONNECT is described which provides three interfaces to CP/M: novice, intermediate and advanced. Switching between them is actually by means of function key presses from the user (Alty and Brooks, 1985, p. 344).

The idea of parallel networks is a valuable enhancement of transition network based UIs, but the structure of CONNECT is not necessary to implement it. Apart from the run-time production system, the same facility is implementable in any transition network UIMS which allows transitions based on conditions over user input, application action and UI actions.

3.4.3 **USE Transition Diagrams**

The User Software Engineering system (Wasserman, 1984, 1985) is a complete design methodology for user interfaces. It employs transition diagrams for specification. Since the system was originally developed for the production of interfaces to database systems, it is presently restricted to textual input and output, although extensions to handle graphical input and output are envisaged (Wasserman, 1985, p. 711).

There are three components to a USE transition diagram. These are the node, the arc and the operation. A node can have output text associated with it, like SYNICS. Nodes are connected by

arcs, which may be labelled either by a string literal, which is to be matched with user input, or with a named diagram. Finally, operations are application actions which are also associated with arcs, but in a way which makes them separate network elements, as will be described below. Each diagram must have one start node and one or more exit nodes.

The specification of rules for transition traversal are quite limited. There are a number of special features for specifying buffered and unbuffered keyboard input, variable length strings and control characters. However, the only operation over these tokens is alternation. If a called network is the arc label, it may return a value to the calling network; however, this value can only be a non-negative integer (Wassserman, 1985, p. 704). No parameters can be passed to the called network, although global contextual variables are available.

There is no provision, as in SYNICS, for accepting input from the application. However, the special operation elements serve this function, at least to an extent. An action, represented by a box, is an application module and can be written in any high-level language. When such an application box appears on an arc, control is passed to the module. When it completes, it may return a value which can determine the path to the next node; again, these values must be non-negative integers. An application box, then, is a special form of node which has an application action rather than an output action associated with it. There is a restriction that no two application boxes may be adjacent.

The way invocation of applications is carried out seems unfortunate in that it produces two types of nodes with different characteristics. It also means that there are two corresponding types of arcs with different types of labels. The result is a much more complex diagram with no increase in expressive power, since the SYNICS method of allowing input to be accepted from the application has the same effect.

3.4.4 The Interactive Push-Down Automaton (IPDA)

The Interactive Push-Down Automaton (Olsen, 1984a) was developed from SYNGRAPH (see Sections 2.3.5 and 3.3.3), but uses a state transition model for specification. Compared to the systems so far examined, the IPDA is much more complex with a more developed set of transition conditions and network actions.

The IPDA is a Transition Procedure Transition Network. That is, all actions take place as the

result of arc traversals. The nodes, or nonterminals of the IPDA definition, have no actions associated with them, although a node has associated with it a set of attributes which provide an environment, used for passing values from one node to another. Also, each node has a set of transitions. The transitions available from a given node, and the state of the system on entering that node, depends upon the previous state of the system. In addition to the normal state of the system, there are two other pervasive states, called the escape state and the reenter state. "The set of legal transitions that the parser may take includes not only those leaving the current state but also those leaving the reenter and escape states of the current nonterminal." (Olsen, 1984a, p. 183) The function of the two pervasive states will be considered further below.

Each transition consists of a current state, a next state, a selector, an action and a priority. (Olsen. 1984a, p. 183) The selector and action are a pair and can consist of several types. Selectors may be:

i. a device identifier: this may be either a sampled device or an event. The possible actions are either sample or consume, respectively.

ii. a nonterminal, or node: the action here is a call of the node, causing the current state to be saved and control passed to the called nonterminal.

iii. NIL and NOW: these are immediate transitions which do not require a wait for input. They are associated either with an application action or with a return from a called node.

iv. COND: this is a conditional transition and is associated with a predicate formed by an application function, perhaps with state contextual variables as parameters. Unlike SYNICS there is no language for producing expressions for the condition predicates; rather, the condition is evaluated by an application function which is written in a high-level language. Although this means that the predicates can have a rich set of operators and a complex syntax, the condition itself is not part of the UI specification. This appears to be a defect in that the formation of the predicates either forces the UI designer to concern himself with the application or he must rely on the application to provide the conditions needed.

v. FOLLOW, ESCAPE_FOLLOW, and REENTER_FOLLOW: each of these selectors determines the form of return from a called node. A FOLLOW results in a normal return, an ESCAPE_FOLLOW returns to the escape state of the node and the

REENTER_FOLLOW returns to the reenter state of the node, each of these states determining the possible transitions which can be taken. Each is associated with its corresponding return action, viz., return, escape_return and reenter_return.

The complexity apparent here results from the more highly developed graphical input possible and from the special provision for escape from commands and reenter commands. Looking first at the graphical input, the separation of the device identifier from the input action is the key to handling sampled input. As was noted in discussing SYNGRAPH, sampled input requires some sort of trigger to determine when the sampling should take place. The solution is to process the specification such that sampled device/sample action pairs are combined with event device/consume event pairs. In the following example, the selector is given above the line and the action below the line. Fig. 3.1a shows the specification as written and Fig. 3.1b the result of processing to provide a trigger for the sampled input (Olsen, 1984a, p. 189).

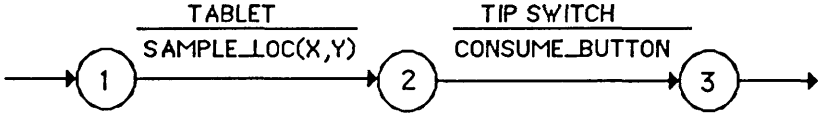

Figure 3.1a Sampling without trigger

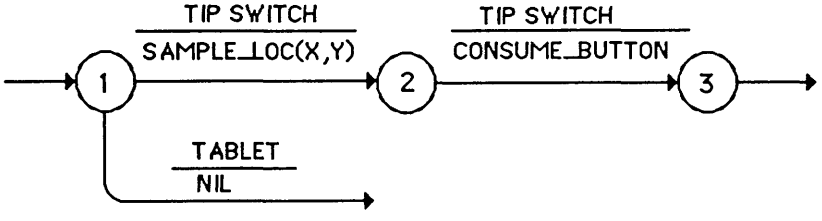

Figure 3.1b Sampling with trigger

Note that the Tip Switch input is now associated with the assignment of locator position (which is assigned to the state variables x and y). The switch event, however, is not removed from the input queue until after state 2. Also a new transition is added from state 1 to enable the tablet even

though the sampling is not done until the switch is pressed, since a device is enabled when its identifier is encountered in a transition selector.

Although this approach works, it could be argued that the same result would be produced by continual sampling with only the last value saved, i.e., the value when the switch is pressed. Event start and event line in the SYNICS example (Sec. 3.4.1) uses this method. The Olsen method of trigger assignment seems unnecessarily complex, although it may be argued that it prevents busy looping.

The second source of complexity is less easy to remove. IPDA is the only transition network notation which attempts to provide support for cancel and re-entry of commands. A re-entry command, as described in Sec 3.1, is one which allows a return to the point in the dialogue at which the transition was made. Unlike an normal return which returns to the point after the call was made (i.e., it passes on the the next node), a re-entry return must return to the point before the call was made. Furthermore, any changes to state variables must be retained during the return; they cannot be removed by the popping of the stack. Similarly with a cancel command which must, upon execution of its cleanup actions return to the nonterminal above the calling nonterminal. Multi-level escape, which is provided for in SYNGRAPH by the absence of the "handled" token in an escape nonterminal, can be carried out in IPDA by a sequence of ESCAPE_FOLLOW returns, finished by an ordinary RETURN. (Olsen, 1984a, p. 201)

Globally available commands are not catered for by any similar mechanism, although a special reenter nonterminal can be attached as a transition from all nonterminals to implement a global command. (Olsen, 1984a, p.210) A method to provide subglobals in a simple way is not present. Such subglobal commands would have to be added manually.

Olsen also considers the input of data in an arbitrary order. His method consists of using a set of flags corresponding to the successful collection of each item of data and a COND predicate which tests whether all the flags are set. By looping through a set of transitions each of which calls a nonterminal to collect one item and then testing for all flags being set, the items can be collected in any order (Olsen, 1984a, p. 201). Although this could be implemented in any of the three transition networks we have examined, it is notable for being the only one where this feature is considered.

The IPDA is clearly the most powerful of the systems examined, but has paid for this power by

increasing the complexity of the notation. It is arguable that the trade-off of power for comprehensibilty is a fair one and that the IPDA should be judged the superior system as a result. However, the value of a system depends crucially on whether it will actually be used. In the case of IPDA, by handling sampled input differently and hiding the mechanism of cancel and reenter commands in the run-time system, a better notation is likely to result.

3.5 Specification of event-based UIs

A third model of the UI upon which to base the specification notation views the dialogue as a set of events generated by user interaction with the input devices, by application actions and by the occurrence of other events (i.e., events generated internally by the UI). On this view, the UI consists of event-handlers which respond to events sent to them by producing output, sending information to the application, generating new events and changing the state of the system. Since an event can be sent to more than one event-handler, there can be several event-handlers active simultaneously. "The set of event-handlers active at any one time defines the legal user actions at that point in the dialogue." (Green 84, p. 309) This means an event model UI can specify multi-threaded dialogues in which the user can interact with several separate sub-dialogues without any change of "mode". (Green 85a, p. 93, 1984, p. 309)

This approach is somewhat similar to Input-Output Tools. The operators for input rules in the Input-Output Tool model which allow concurrent activation of tools gives a similar expressive power to event model UIs. However, IOT can only pass their output tokens to the calling tool. Furthermore, the activation of an IOT takes places as the result of the parse of an input rule for a higher-level IOT, while in the event model event-handlers can be activated as part of the action of processing an event.

The event model is also similar to the object-oriented approach of Smalltalk. An event-handler could be modelled as a Smalltalk object in which the "events" were the acceptable messages which could be received by that object and the actions were the methods associated with those messages. Smalltalk does not combine all UI functions into a separable component in the UIMS way, but if it did, or if a UIMS were implemented in Smalltalk, then the specification of the UI would consist of defining I/O event-handling objects.

3.5.1 **Petri Nets**

Petri nets provide a notation for "modelling of systems of events in which it is possible for some events to occur concurrently but there are constraints on the occurrence, precedence, or frequency of these occurrences." (Peterson, 1977, p. 223) It would appear, then, that Petri Nets are a candidate for defining event-based UIs.

A Petri Net consists of a graph containing two types of nodes: places, or states, and transitions. Arcs connect each type of node to the other. More than one arc may enter or exit from either type of node. Transitions may have conditions associated with them. Flow of control in a Petri Net is shown by tokens which may pass from one state to another. When all the states leading to a given transition have tokens, that transition is **enabled** and may **fire** if its condition is met. This removes the tokens from the transition's input state(s) and results in tokens being deposited in its output state(s). Fig 3.2a shows a Petri Net with transition t1 and t2 enabled, while Fig 3.2b shows the result of t1 firing.

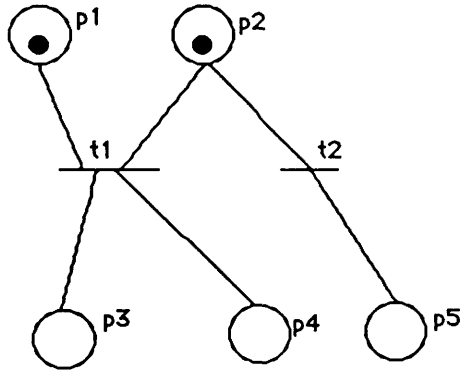

Figure 3.2a Net Before Firing

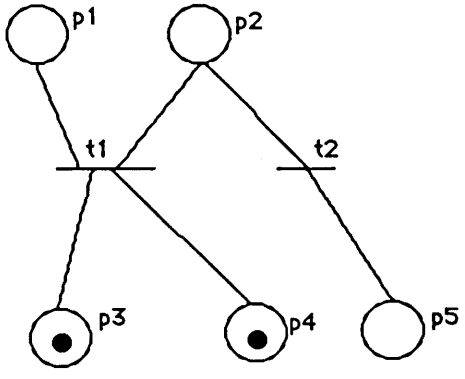

Figure 3.2b Net After Firing

The Petri Net is similar to a state transition network. Indeed, state transtion networks form a subclass of Petri Nets where each transition has one input and one output state (Peterson, 1977, p. 247). The action box notation of Wasserman's USE diagrams function a bit like special Petri Net transitions which can have one input state, but multiple output states. The fundamental advantage of Petri Nets from the point of view of dialogue specification is the ability to describe situations where several transitions are enabled at the same time, perhaps in widely separated parts of the network.

Barron (1982) describes an extension to the TAXIS language for the design of interactive information systems which uses Petri Nets to define dialogues. Dialogues consist of networks, called Scripts, which are made up of states and transitions. A state may have the values on or off, which correspond to being marked with a token. Transitions have associated with them a set of input states, a set of output states, a condition and a set of actions, much like conventional Transition Procedure Dialogue Networks (see Sect 3.3.3). When a transition's input states are all on (marked), that transition is enabled (i.e., it attempts to satisfy its condition); when satisfied, the transition fires and its actions are executed, including turning on all its output states (Barron, 1982, p. 14).

The notation is clearly more powerful than augmented transition networks, but it has not yet been used by any UIMS; even the script notation and its accompanying syntax, called Interpret (Pilote, 1983), are design tools only; they are not currently able to be used to automatically generate UIs (Pilote, 1983, p. 131). This would suggest that the notation may not be preferable to other currently available ones (e.g., IOT or the event-handlers of the University of Alberta UIMS (see below)). Also, Jacob suggests that the notation is "unnecessarily complex." (1983a, p. 17)

The advantages of the Petri Net approach lie largely in the ability to carry out formal analysis of the network, e.g., to discover transitions which are never enabled or possible deadlock (Peterson, 1977, pp.238-240). In a complex concurrent system, such formal analysis is highly desirable. But it does not follow that the notation must be used for the specification as long as the notation used can be translated into a Petri net.

3.5.2 The University of Alberta UIMS

In the University of Alberta UIMS, the UI consists of events and event-handlers. Events are produced upon generation of tokens by the presentation system or the application interface (see Section 2.2.7) or internally by event-handlers. Tokens have a possible one-to-many mapping to events, allowing one token to produce several events, each dealt with by a different event-handler. Each event handler has

i. a list of tokens which it can handle, each associated with an event name

ii. a declaration of local variables; these can be of any type definable in the C language

iii. a list of event declarations. Each has an event name, followed by C statements which

are executed when the named event is received by the event-handler.

There is a superficial resemblance to a Smalltalk object, with tokens analogous to messages and actions, including token generation, analogous to methods. Furthermore, the definition is of an event handler as a type; in an executing UI, there may be a number of separate instances of a given event-handler, similar to Smalltalk classes and instances. However, unlike Smalltalk, there is no subclass mechanism, thus attributes cannot be inherited by one event-handler from another. There are no class (static) variables, which in Smalltalk allow values to be retained between invocations of an object. Furthermore, there is no mechanism for the recipient event-handler to return values to the object sending the event. Rather, the sending of tokens is asynchronous. Thus, event-handlers are to be viewed as concurrently executing processes (although the concurrency may be simulated in the implementation). (Green, 1985b, p. 209)

Among the actions which can be executed when an event is handled are several special ones. New instances of event-handlers can be created and existing event-handlers terminated. A nice feature is that when an event-handler is terminated, it is first sent a finish event by the system (Green, 1984b, p. 210), allowing clean termination, including possible deallocation of resources, somewhat like the cleanup rule in the Input-Output Tool. Tokens can be sent to the presentation system to produce output and to the application interface to pass information to the application process. New internal events can also be generated and sent to other active event-handlers.

At its simplest, an event-handler can be reduced to an analogue of a single node of a transition network and its transitions. The set of input tokens define the selectors for the transitions and the action of creating a new event-handler is the next node (followed, presumably by its own termination). Other actions are associated with the traversal of the arc. Thus, specifications using a transition network notation are translatable into the event-based notation. Such a facility exists for the University of Alberta UIMS (Green, 1984b, p. 210) In general, for two notations N1 and N2, if N1 is at least as expressively powerful as N2, then UIs defined in N2 can be translated into N1. This implies that the choice of design notation is less important than the choice of language used for generating the UI specification object which is used for run-time execution. Since the event-based notation is equivalent to a Turing Machine (Green, 1984b, p. 207), UIs described in, say, a context-free grammar can be converted to an equivalent event-based description, although not conversely. Along with the current translator for transition network specifications, a

grammar-based notation and translator is envisaged (Green, 1984b, p. 210). The same equivalence was exploited by SYNICS in producing both BNF and transition network specification systems for the SYNICS system, of which the transition network description is now the standard.

Although there are no built-in facilities for special UI features, the fact that event-handlers can be active concurrently means that global commands can be easily described. They will be created at the start of the dialogue and hence can respond to user input at any time during the dialogue (Green, 1984b, p. 208).

In spite of the undoubted power of the system, it is doubtful if in a complex UI specification, the behaviour of the UI at different points in the dialogue can be easily predicted since a number of separate event-handlers will be active and the set of current tokens in the input queue(s) could be quite large. The hierarchical structure of the Input-Output Tool model and its division into separate functionally significant parts places a constraint on UI structure that may well assist in the analysis of designs. Another alternative to handling this complexity would be to produce a Petri Net of the UI so that formal analysis could be carried out.

3.5.3 **Production Systems**

Another approach to UI specification which allows the description of multi-threaded dialogues is the production system (hereafter, PS) model. It is perhaps significant that no working system using this method has appeared between the first suggestion of the approach in 1980 (Hopgood & Duce, 1980) and the most recent discussion (Duce, 1985), but the system should be examined nevertheless.

A production system consists of three parts:

a short term memory (STM) which contains the current input tokens, and functions somewhat like an input queue,

a long term memory (LTM) holding production rules which consist of rules determining the actions to be performed (and possibly tokens to be generated) upon the occurrence of tokens in the STM

a central processing unit (CPU) which at each processing cycle examines the rules in the LTM to find those matched by tokens in STM, subsequently executing the matched rule(s) according to a processing algorithm (Duce, 1985, p. 251-2).

The production rules of a production system are entirely different from those of a BNF definition. The left-hand side of the PS production rule consists of those tokens which will fire the rule, if present in STM. The right-hand side contains the actions to be taken and, optionally, further tokens to be generated and placed in STM. Thus, a production rule here is more like the label on a transition in a transition network, with the left-hand side taking the place of the transition selector.

The behaviour of the PS depends crucially on the way in which the STM handles tokens, particularly how tokens are removed. Duce (1985, p. 252) mentions the following:

i. tokens are removed from the STM after each processing cycle,

ii. tokens are only removed if matched with the LH of a production rule,

iii. token values replaced, or overwritten (like a continually sampled device), or

iv. tokens are removed at a given time (requiring each token to have a time tag).

As we shall see, option ii gives the system its particular strength, but all the options could be used to capture the needs of UI description.

Equally important in determining PS behaviour is the method by which production rules are chosen for firing. Unlike BNF which has its rules hierarchically organised or transition networks in which all legal transitions from a state must have unique selectors, the left-hand side of several production rules may be matched by the contents of the STM. How, then, is the CPU to decide which rule to fire? One possibility is that the rules are given priority ordering (statically or dynamically) and the CPU fires the highest priority rule which is matched. Or, if concurrency is allowed, several rules may be fired simultaneously. (Duce, 1985, p. 252)

Duce presents an example using option (ii) for STM token removal and priority ordering of production rule firing. The example shows the apparent advantages of production system specification for capturing multi-threaded dialogue over the transition network approach. Figure 3.3a shows the LTM for a production system specification of a rubber-band line drawing facility. The input events include buttons (b1-3), a locator showing the current position of the tracing device (X) and a system generated flag (S). Priority of selection is given by the ordering of the rules, with the rules higher in the list having higher priority. Figure 3.3b gives a similar specification for a timer. To join these two "sub-dialogues" together so that the user can interact with both of them at the same time, the two sets of production rules need only be combined with

the rules in figure 3.3b appended to those in figure 3.3a. If this were to be specified in a transition network, the two diagrams would combine to form an extremely complex one, suffering from a combinatorial explosion of transitions (Duce, 1985, p. 253).

B3	->	<store end point>
S X	->	<display rubber band line> S
B2	->	<store start point> S
X	->	<display cursor>
B1	->	<enable tracking device>

Figure 3.3a Rubber-band line LTM

B6	->	<stop timer>
S1 T	->	<update clock and timer> S1
B5	->	<start timer> S1
T	->	<update clock>
B4	->	<enable clock>

Figure 3.3b Timer LTM

This appearance of advantage suffers somewhat when the affect of the ordering of the rules is considered. Consider the case in which the user starts the timer (b5), then starts the line-drawing (b1), intending to stop the timer when the line-drawing is completed. If we assume that the redrawing token (X) and the timer update (T) are generated at each cycle during the re-drawing of the line, then the timer update will never be executed, as its priority is too low. In essence, the priority of the rubber-band drawing dialogue "locks out" the timer. Duce suggests several possible solutions to this problem:

i. as the line-drawing and timer are separate processes, the system could be constructed to model the two processes, with separate STM and production rules for each,

ii. change the STM so that the timer overwrites the existing current time whenever the token is generated (option iii above). Thus, token generation itself will handle updating the time and the elapsed time can be determined by the time token present when the timer is finally stopped.

iii. if the time between cycles is sufficiently high, then only infrequently will both the sampled locator and clock update will occur simultaneously,

iv. dyanamically changing the order of production rules on the basis of the contents of the STM, so that the timer update can be handled when present.

Remember that the advantage claimed for the PS approach lies in the ability to specify interleaved commands more easily and clearly than by other means, particularly transition networks. Solutions (ii)-(iv) do not solve the problem generally; they simply provide an ad hoc solution for one discovered "lockout". As the UI becomes more complex, and the number of

production rules becomes larger, the number of such conflicts is bound to increase. Thus, the apparent advantage seems to dissolve into a trade-off of one type of complexity for another. Solution (i) separates the specification of the two sub-dialogues once again, placing the "interleaving" in the run-time support for concurrency of processes. It is difficult in this case to see the advantage of PS specification over transition networks for the separate sub-dialogues. Furthermore, other notations, e.g., Input Output Tools, are capable of expressing the interleaving of dialogue interactions.

Chapter Four

The UI Design Environment

4.1 UI Design Methods

The user interface design environment of a UIMS consists basically of those tools which assist in the UI design process and include both specification tools and prototyping/testing tools. The nature of the tools required in a UI design environment will depend upon the individuals involved in the process and the tasks which they will carry out. The tasks to be carried out may include definition of the UI in a UI specification notation, the alteration of the UI during testing, and the adaptation of the UI to customize it for particular users or classes of users. The individuals engaged in these activities may include programmers, with or without extensive knowledge of user interface design, dialogue designers with limited programming skills and users with little or no programming skill; indeed, the agent of UI modification may even be the system itself. Roughly, the division of responsibility looks something like this:

	UI description	Testing/Modification	Adaptation
Programmer	x	x	
UI Designer	x	x	x
User		(x)	x

It is difficult to generalize about the division of labour within the UI design process, particularly as no generally accepted methodology has emerged which has had widespread use. Nevertheless, one important point arises from the participation of UI designers and end users in the UI design process, viz., that an acceptable UI design environment must take into account the limited programming skills of those involved.

Tanner and Buxton (1985) point out the conflict which exists between the desire for ease of use, motivated by the need to accommodate non-expert programmers in the design process, and descriptive power, required if the UIs capable of being designed are not to be constrained by

the design environment. They contrast **glue systems** at one extreme from **module builders** at the other. A glue system is one which provides a library of "dialogue cells" from which the designer may select and which may be "glued together" to form the resulting interface. Menulay is given as an example of such a system, where individual screen menus are available from a library (and can be created via the Menulay interactive specification system) and joined together to form a network of linked menus. The structure of individual cells or menus is built into the UIMS and cannot be altered by the dialogue designer. Module-building systems, on the other hand, allow the specification of the "low level details of the dialogue structure" via a specification language (p. 71).

The contrast described by Tanner and Buxton is perhaps too extreme. Clearly, a glue system requires some method by which the individual cells in the library are created (a way of constructing the UI modules) and a module building system can, and usually does, have a means for naming components of the UI (e.g., sub-networks, commands, event-handlers) so that they can be referred to and called from several points in the UI definition. The problem is actually more general and is similar to that occurring in computer assisted learning where authoring languages have been designed to allow non-programmers access to special facilities to assist them in producing CAL programs. Features of an authoring language which simplify the specification task also limit the designer from expressing designs which do not fit neatly into the model which underlies the authoring language structure (Short et al, 1982).

Tanner and Buxton suggest that the solution to the problem is a system which incorporates a module-building facility, presumably to be used by experienced programmers, and a glue system which allows modules to be joined together, a task suitable for UI designers and even users (p. 71). However, this solution is still based on a view of UIs in which the links among UI modules or components are relatively simple, such as hierarchical menus, where the links are specifiable without the use of complex control structures or reference to complex contextual data-structures. A general UI design environment cannot make such assumptions and thus requires a different solution to the ease-of-use/descriptive power problem.

The conflict between ease of use and descriptive power exists only if it is assumed that a single method of specification is to be provided. The combined glue/module builder proposal of Tanner and Buxton affords two specification techniques, but is not sufficiently general to be a model for

any UI design environment. A general UI specification language is required by which to describe and specify the entire UI, to allow for formal analysis where appropriate, to provide a human-readable description, and to prevent restrictions on the types of UI which may be created. In addition, tools must be provided to augment the language which facilitate a range of methods for specification of various aspects of the UI.

Olsen et al (1984b) list three specification techniques: notation, example and composition (p. 39). Specification by notation requires a dialogue specification language. It is, as has been argued, fundamental to a sufficiently powerful UI design environment. Specification by example refers to the production of generalizable formats, templates or structures. Finally, specification by composition is equivalent to a glue system and may include provision for macros, module libraries and the use of templates (perhaps themselves specified by example). Clearly, apart from specification by notation, it is unlikely that any complex UI could be fully specified by any one technique; at least, we do not have the tools at present to allow this (Olsen et al, 1984b, p. 39). However, specification by example is a natural method for specifying screen displays and menus. Specification by composition can be incorporated into systems based on specification by notation; it is also useful for specifying form-based interfaces .

UI design is an instance of problem-solving and "suitable representations of problems are crucial to solution finding...." (Shneiderman, 1983, p. 63) It is the representation which underlies and allows particular specification techniques to be used. Specification by example of the UI appearance requires a graphical representation of that appearance which can be manipulated. Specification by composition of UI modules requires a representation of those modules which allows them to be named and hence referred to. Both of these are obvious examples, but the possibility of finding new and more powerful techniques for UI specification depends upon appropriate choice of representation of the UI or its parts; it requires "powerful metaphors" which allow the designer to make use of features of the representation itself to assist in the specification process.

There is evidence that graphical representations provide a useful aid to design problems. Carroll et al (1980) found that there was a significant advantage both in terms of speed and accuracy of solution of a spatial over a temporal representation of an isomorphic design problem. Where the designer can interact directly with a graphical representation of the UI and not have to

translate the result into a UI specification notation, the design process should be more efficient, both because of the time saved by avoiding the translation process and because there is no notation to learn. It is to such interactive graphical design tools for UI specification that we now turn.

4.2 Interactive Graphical Specification

Where a UI communicates with its users via text only, a non-interactive specification of the presentation is adequate. In Rapid/Use, for example, output is specified as text strings along with special terminal control commands for such functions as cursor control, tab setting and screen partition. (Wasserman, 1985, pp. 701-2). However, where a UI uses icons and other graphical objects for user manipulation and uses changes to graphical objects as feedback, a textual specification is less satisfactory. The complexities of such a specification may be daunting for the dialogue or graphic designer, resulting in less acceptable UI presentation systems and inhibting the exploration of alternative designs. Additionally, the specification of a graphically-oriented presentation system is a "logical candidate for a graphical specification tool." (Olsen, 1985a, p. 126)

Also, some notations for the UI control component are more amenable to a graphical representation than others. For example, a transition network can be represented by a transition diagram. If the dialogue structure is displayed as such a diagram, the representation can be drawn interactively and the results automatically translated into a representation appropriate for use by the UIMS run-time system.

Thus, at least some aspects of both UI presentation and control are open to interactive graphical specification. As shall be shown in the discussion below, there are limitations to both. If there is an ideal interactive specification system, it is one which integrates the types of graphical representations mentioned above with other specification tools in such a way that the system as a whole makes the least demands on learning and programming skills while providing maximum descriptive power.

4.2.1 Interactive Graphical Specification of the Presentation Component

Menulay is an example of a system which allows interactive graphical specification of the

appearance of the interface. Fundamental to the Menulay design philosophy is the idea that the "specification technique uses the same devices and interfaces as those the end-user will ultimately face." (Buxton et al, 1983b, p. 40) This statement is ambiguous and can mean either that the design environment uses the same type of interface as the ultimate system or that the designer is actually creating and manipulating the very same (graphical) objects which the end-user will interact with. Both interpretations are true of Menulay.

The Menulay system limits itself to menu-based interfaces, at least in the sense that all interaction takes place via user selection of light-buttons, which may be pictures or text; selection of a light button can cause designer-specified application code to be executed, possibly resulting in changes to the display. The design environment itself is menu-based in this sense. Pictures can be called up from a library (or created via a graphical editor) and, along with text, placed on the screen and sized. Certain items may be turned into light buttons and the application function linked to that button by name. The actual code for the application functions must be written separately in a high-level language.

There are several advantages to this approach. The designer is working in an interactive, graphical environment; no notation or language must be learned or used. The designer can see during the design process exactly what the user will see when the interface is used. Since the appearance of the display is a crucial factor in the effectiveness of interactive graphical interfaces, the ability to be able to work with the visual image of the interface rather than with a textual description is a powerful tool. In addition, graphic designers and end-users can be involved early in the design process.

However, Menulay requires pre-processing of the menu descriptions and linking with the application functions each time the design is prototyped (Buxton, 1983b, p. 37). Thus, while the image of the interface can be manipulated directly, the designer cannot immediately see the dynamic results of his work. Also, the appeal of the Menulay design environment relies largely on the relatively simple types of interface which are supported. Basically, the designer can specify only screen displays, light buttons and the names of the functions which the light buttons select. Complex relationships between input and feedback in the dialogue (i.e., a complex dialogue control component) cannot be specified, except via side-effects of the application code, which weakens the boundary between interface and application.

The design principle -- work in and with the same environment which the user will eventually use -- is still worth pursuing both because it allows the designer to see the end results of his design decisions immediately and because users can be involved in design early and often. Whether a more powerful (i.e., general) UI design system can follow this principle is less certain. It is presented here, then, as a guideline in the production of a design environment; one desirable feature to be provided to whatever extent possible given the other demands of UI design.

4.2.2 Graphical Specification of Dialogue Control

Where some component of the UI can be modelled by objects which are naturally represented graphically, it is possible to specify that component via manipulation of the graphical representation. These objects may not correspond to any objects of which the end-user will be aware. They may constitute an abstract description of interest only to the designer.

State transition networks are commonly represented by transition diagrams and thus it is not surprising that interactive graphical specification of dialogue control has been developed for several systems which employ a transition network model of the UI control component.

USE transition diagrams (see Section 3.4.3) can be edited interactively via a transition diagram editor, called TDE (Wasserman, 1985a, p. 707). A dialogue description is generated automaticaly from this diagram which is executable via a transition diagram interpreter (TDI). Subdiagrams may be named in a diagram and edited separately. The interactive graphical nature of the specification process applies only to creating and naming nodes, joining them by transitions and labelling the transitions with the input conditions. The output generated on entry to a node must be specified textually and the semantic actions associated with transitions must be produced by means of a general high-level language.

Jacob (1985) describes a similar graphical specification system based on a transition network notation for UI definition. As the notation allows more complex conditions for transitions than does USE, these are specified separately (and textually) from the diagram just as are semantic actions "to avoid clutter." (Jacob, 1985, pp. 53-4) It is suggested that the definition of the tokens of the language, both input and output, in terms of physical device events and attributes is carried out via the same type of transition diagrams, although the details of this are not clear (Jacob, 1985, pp. 54-55).

The Dialog Management System of Virginia Polytechnic Institute (see Section 2.5) employs a different model of dialogue control and consequently a different notation for transition diagrams. Nevertheless, the various functions making up a supervisory flow diagram (SFD) are representable via geometric objects and their hierarchical organisation can be represented by directed lines joining them. A Graphical Programming Language (Hartson et al, 1984, p. 59) allows interactive graphical editing of these diagrams in a fashion similar to the graphical editors for transition diagrams.

It is important to be clear about what such graphical editors do and do not provide. Transition diagrams are a useful design tool for transition networks because significant features of the network are represented by visually significant features of the graphical representation. The interactive editing of such diagrams simply removes a step of manual translation from the diagram to another notation which is either usable directly for interpretation or which can be translated into a machine-readable form for execution. However, not all elements in the UI have obvious graphical representations in the diagram; these must still be defined in some other way, e.g., by a textual notation. Jacob (1985, p. 55) notes that we await useful visual metaphors by which to specify these other elements, for example, the semantic actions. However, even if such metaphors were developed, the resulting variety of radically different metaphors might produce more confusion in the design process than a single, unified, albeit textual, notation.

Also, graphical programming fits certain notations better than others. There are several transition diagram editors, but it is difficult to conceive of a graphical editor for a BNF notation, except in the trivial sense of a form-based editor with production rule templates. Similar problems arise in imagining graphical editors for event-based UIs (however, see Section 4.4). While this may simply mean that more imaginative metaphors are required, it also suggests that the attributes of these models do not map onto clearly-grasped features of graphical objects which are sufficiently familiar. The value of a metaphor, visual or otherwise, depends upon the user of the metaphor understanding the relationships of its parts better than he understands the relationships of the parts of the original objects. If obscure features of unfamiliar graphical objects are required to represent a UI component, that representation is likely to be worthless for design purposes.

A graphical editor of the dialogue syntax cannot by itself help the designer in conceptualising

the resulting UI. Wasserman comments that "we have observed that understandability of the diagrams is enhanced by the use of sample screens, either using TDI [the transition diagram interpreter] or hand drawn screens, to show the typical displays from specific nodes." (1985, p. 712) Thus a graphical editor for the presentation system (see Section 4.1.1) is a desirable additional tool. Also, while the dynamic behaviour of the interface may be made clearer to a designer by the use of a graphical representation of the control component, the predictability of behaviour is likely to be lost where many subdialogues are involved. In such cases, the ability to move easily from editing of the UI to prototype execution and back again may be far more important for efficient design than whether the UI is specified graphically or textually.

4.3 Prototyping

Interactivity in design can refer to the highly interactive nature of editing of graphical representations of interface appearance or structure, as described in the previous section. However, it may also refer more generally to a design environment in which the consequences of design decisions are perceived with little delay. For interface design, this can be implemented through a close link between producing or modifying the UI specification and creating or changing the resultant UI, especially in the form of prototypes.

Given the lack of generally accepted valid principles for UI design, prototype UIs are desirable so that testing and evaluation of the design can be carried out continuously in the design process, and the results fed back into further iterations of the define-implement-test loop (Hanau and Lenorovitz, 1980, p. 271). A number of iterations must be expected in order to produce an acceptable UI, so the speed of prototype generation is a major factor in efficient UI design.

Along with speed of production, the content of the prototype is important. The prototype may be a full, albeit tentative, version of the ultimate system, including the application functionality. However, the prototype may also contain only selected components of the final system. Such a restricted prototype enables evaluation of purely human factors aspects of the design. For example, Hanau and Lenorovitz describe separate prototyping tools for displays (1980, pp. 272-3) and command syntax (the IDS, see Section 3.3.2). Furthermore, UI design can proceed in parallel with the design and implementation of the application functions.

Clearly, prototypes can be constructed with UIMSs which do not have special prototyping tools.

The University of Toronto UIMS, which includes the Menulay system for interactive graphical specification of the UI displays, uses a further program, Makemenu, which generates C programs from the Menulay menu specifications plus any additional application functions (Buxton et al, 1983b, p. 37). A restricted prototype can thus be created by producing stubs in place of the full application routines. Since Makemenu produces C source code, compiliation is still necessary. Each time changes are made to the specification and a new prototype is desired, these changes can be made either via Menulay (followed by another use of Makemenu) or via textual editing of the C source code produced by Makemenu; in either case, the UI must be recompiled.

The RAPID/USE system includes a prototyping tool called a Transition Diagram Interpreter. This takes a textual description of the USE transition diagrams (see Section 3.4.3), called a TDI script, and produces an interpreter for the associated TDI script. No application actions need be referenced as long as dummy action labels are included in the specification (Wasserman and Shewmake, 1985, p. 200). The script which is interpreted can be replaced by other scripts, allowing quick testing of alternative versions of the interface (Wasserman and Shewmake, 1985, p. 203). When changes are required, these may be made via the Transition Diagram Editor or via direct textual editing of the TDI script. In either case, no recompiliation is needed as the dialogue script is directly interpreted.

Such a prototype contains no application actions. It can be useful for evaluating the appearance of displays and the structure of the dialogue, but cannot of itself assist in evaluating those aspects of the UI which depend upon the application, such as semantic feedback or transitions which depend upon application-generated values. A further tool, called the Action Linker, associates application routines, written in a supported high-level language (C, Fortran77, Pascal or a database manipulation language, PLAIN). (Wasserman and Shewmake, 1985, p. 205) The Action Linker uses a TDI script and the files containing the action code to produce an executable UI program.

Thus, while the TDI provides a useful tool for the prototyping of application-independent dialogue structures, the need to use the Action Linker to produce a full prototype weakens the resultant design environment. More desirable would be a design environment which allows alteration to the specification to take place while the prototype is running and after the application routines have been included. Such a system is described by Tanner and Buxton and the process

is called by them "Suspended-Time UI Definition/Modification." (1985, pp. 72-3) While the examples they give of such modification only cover presentation-level changes such as altering input devices or changing the appearance of a light-button, run-time modification can be more extensive, and might well include the dialogue structure and links to application routines.

Jacob (1983b) describes a RTN notation which is directly executable by means of an interpreter. In a later version (Jacob, 1985) which includes support for graphical editing of the UI structure, the design environment uses two sets of windows, one of which - the simulator - displays the system under construction and one (the programming windows) which displays the graphical representation of the system with the current state marked. (Jacob, 1985, pp. 55-56) The designer may interact with the simulator as if he were the user in order to test prototype behaviour or he may interact with the programming windows to alter the specification. "Since the state diagram language is entirely interpreted, the program can be modified arbirtrarily while it is running." (Jacob, 1985, p. 56) Like RAPID/USE, unspecified action modules can be replaced by stubs; in fact, they can be added automatically by the UI structure editor (Jacob, 1985, p. 33) Unlike RAPID/USE, the prototype executor will also operate when application actions are supplied. This is particular important for testing, as noted by Carrol and Rosson:

"However, it is also critical that working prototypes be available as early as possible, as many problems can be expected to occur when the user is required to deal with the full complexity of the system." (1985, pp. 24-5)

Action routines, since they are separately edited and are compiled, cannot be altered in this run-time mode. Jacob suggests implementing semantic actions via a separate visual, interpreted language (Jacob, 1985, p. 55), but does not consider what such a language would look like.

The ZOG menu-based UIMS incorporates a run-time editor, ZED, which allows editing of individual ZOG frames, or nodes of the menu network, at all times during executing of a ZOG net. ZED allows frames to be added, deleted, copied, moved and linked to other points in the network, and also allows the contents of individual frames to be edited (Robertson et al, 1981, p. 476). Empty subnets, called schemas, can be created and re-used in various parts of the network with editing to provide content for the frames. In addition to called ZED as a global option from any

frame, ZED is also entered automatically if a user selects an option which has no associated frame. A new ZOG network can thus be created "by systematically stepping through the options and creating new frames at each step." (Robertson et al, 1981, p. 477)

ZED is not designed as a prototyping tool. However, the notion of embedding the design tools in the run-time environment increases the flexibility and efficiency of prototype creation and use. There is no longer a clear distinction between creating, prototyping, editing and using the resultant system. Each type of task requires different tools, but in ZOG these tools, such as they are, are always available; it is a "modeless" design environment. Unfortunately, ZOG does not include the graphical editing tools of systems like RAPID/USE or Jacob's executable specifications, but is not difficult to envisage their inclusion in the system via a different editor, particularly given the transition network structure of the system.

A "modeless" design environment blurs the distinction between prototype and final system. Changes to the UI may take place continuously throughout its creation and those changes will be reflected immediately in the running "prototype". Alterations may be made in the same way to allow customization of the UI for particular uses or classes of users. Prototyping and user-adaptation become aspects of the same process, rather than separate processes.

The power of such a system may be too great. Edmonds (1981) identifies three classes of human agent involved in UI change or adaptation:

i. computer specialists

ii. trained users (e.g., dialogue designers)

iii. any user (1981, p. 404)

He suggests that the facilities or tools available to each class should match the skills and needs of that class. Thus, class (i) would have access to tools for creating/modifying the UI structure and its links to application routines, while class (ii) would be able to design screen layouts, alter output messages and perhaps command lexemes. Class (iii), according to Edmonds, would be allowed, perhaps encouraged, to create command abbreviations or to ask for expert designers to alter dialogue structure, based on system logging of commands used and errors made. "None of these facilities [for the ordinary user] will exist in a given dialogue unless specified by the designer." (Edmonds, 1981, p. 419)

It is difficult, without extensive further research, to determine the type and complexity of

dialogue editing which users need and can handle. Experience with ZOG suggests that experienced computer users without specialised dialogue design skills are capable of using ZED for editing network structure and content. Nevertheless, one may wish to prevent users from altering the structure of a complex dialogue since they might well introduce logical errors which they will be unable to correct. A modeless design environment which possesses all the relevant editing and design tools at all times can still restrict access to these tools to desginated classes of users, e.g., by restricting access to the tools themselves or by marking certain parts of the dialogue specification as alterable only by such designated users. The latter approach seems more flexible than restrictions on access to tools. It would allow the non-specialist user the opportunity to develop design skills by experimenting with altering a portion of the dialogue structure while still preventing modification of critical (perhaps shared) portions of the structure.

4.4 Towards a Modeless Design Environment

So far, the systems employing interactive design environments have been based on UIs with an underlying transition network model. This is probably accountable for by the ease with which a graphical representation, and hence interactive graphical editors, can be designed for the network model. However, this is not a necessary connection.

The Programming by Rehearsal authoring system (Gould and Finzer, 1984), although not a UIMS, provides an example of an approach which might used as a model for a UI design environment. Programming by Rehearsal (hereafter, PbR) is a system, implemented in Smalltalk-80, for the creation of educational programs involving complex graphical interaction, without the need for programming skills on the part of the designer.

The PbR design environment consists of a set of objects. These objects, like all Smalltalk objects, can send and receive messages and contain methods, or actions, which are carried out when associated messages are received. The fundamental objects in PbR, all of which have visible representations, include:

i. performers - the "necessary parts of a graphical production whose purpose is educational" (Gould and Finzer, 1984, p. 13), and

ii. stages - windows into which performers may be placed.

The basic classes of performers include string, integer, clock, list, boolean objects as well as a

collection of graphical objects. These objects when combined and edited form the final program. Each object consists of an amalgam of interface features (the input action which causes them to execute their default action; the display effects when they carry out their action) and application features (usually minimal, but may include string manipulation, list processing and arithmetic operations). Performers respond to a number of pre-defined messages: they may be moved, sized, copied and turned into a light button. They also possess a default action which will be executed if, when serving as a light button, they are selected. The action on selection may be changed by textually editing or replacing the Smalltalk statements which constitute its default action.

The design environment is modeless in the sense that all objects respond immediately and visibly to any changes in their definition. It is fundamentally a glue system, in the Tanner and Buxton sense, since the design process consists largely of selecting pre-existing objects from supplied libraries, called Troupes, and linking them via copying into a stage window, plus supplying parameters to messages and some (minimal) writing of Smalltalk code to define their effects on one another.

The power of PbR lies in its providing a set of objects which already possess many, if not most, of the features required for the final program. Where design decisions must be taken, these can be carried out largely by interactive graphical manipulation of the objects' visible representations. Clearly, the set of objects needed for a general interface design environment will be different, but the method of design may well be transferable.

Lieberman's EZWin system provides just such a set of general interface objects. EZWin is a "graphical interface kit" (Lieberman, 1985, p. 181) which assists in the design of graphical menu-based interfaces by providing basic facilities used by such systems, such as mouse tracking, menu administration and alternative command syntax. An EZWin program consists of three types of objects:

i. an EZWin object containing all the other objects in the program and exercising control over the interactions among the objects,

ii. presentation objects, which are graphical representations of application-significant objects,

iii. command objects, which have a selectable graphical representation (icon or text), an argument

list (themselves objects) and an action to be executed when the command object is selected and its arguments have been supplied. (Lieberman, 1985, p. 182)

There is no interactive design environment as in PbR, although the similarity of the systems suggests such an environment could be created; the major problem would be the construction of an appropriate library of interface objects.

Thus, the types of objects in EZWin and the design environment of PbR point towards a UI design environment which is both powerful and easy to use for non-programmers. Both systems, however, make no clear distinction between UI and application aspects of the system; objects have input rules, display actions and application actions bundled together. While this is natural given the heavily graphical nature of the applications which they are used to produce, it is not clear if the approach is applicable given applications with more complex command structures and more abstract application tasks. Lieberman states that EZWIN is "an editor for presentation objects." (1985, p. 182) Also, Gould and Finzer express doubt about the general applicability of PbR:

"As designers create increasingly large and sophisticated productions, they may find complete instantiation [visible representation] to be a nuisance....[And] it may be difficult to build productions which access large amounts of data. At some point, the concreteness may become a barrier rather than an advantage." (1984, p. 70)

Nevertheless, an object-oriented interactive design environment, linked to an event-based UIMS, such as the University of Alberta UIMS, offers the hope of a modeless design environment with interactive visual specification of both dialogue and application components of the system. Further research is needed to establish a suitable underlying UI model, a set of generic UI objects and the basic messages and methods required of these objects.

4.5 Evaluation Tools

Evaluation of a specification requires additional tools which may be thought of as constituting an evaluation environment. Nevertheless, they must be available during the design process and be closely connected to the tools constituting the design environment. The tools fall into two

categories:

i. those for formal analysis of the UI specification

ii. those for gathering and analysing performance data.

4.5.1 Formal Analysis

By using a formal model of the UI control component, such as BNF or transition networks, it is possible to formally analyse the resulting specification with respect to the satisfaction of certain design criteria or simply to acquire information about the specification which is useful for design decisions. Thus, the UI specification notation itself can be thought of as both a design tool and an evaluation tool.

Reisner (1981) employs a BNF description of interactive systems as the basis for analysing the complexity of the designs, in terms of three features of the command language:

i. the number of terminal symbols,

ii. the length of syntactically acceptable sentences for particular tasks (i.e., the length of command strings)

iii. the comparative length of sentences which are semantically equivalent.

(Reisner, 1981, pp. 232-3)

Underlying this analysis are several human factors design principles concerning the usability of an interactive system, viz., (i) the shorter a sentence, the easier it is to learn, use and remember and (ii) a set of semantically equivalent, or similar, commands are easier to learn, use and remember if their syntax is also equivalent or similar. The BNF notation assisted Reisner in expressing hypotheses about user behaviour with particular comparable commands in two similar interactive systems, Robart I and Robart II. Studies carried out on subjects using these systems tended to confirm the design principles expressed above. Reisner suggests, therefore, that the a priori formal analysis of interfaces based on such human factors principles is useful as a design tool independent of(although not in place of) actual testing of prototypes. (Reisner, 1981, p. 237)

While the validity of Reisner's experiments - and hence of the design principles they support- have been questioned (Carroll and Rosson, 1985, p. 10), it is still true that formal analysis of UI features can serve as the basis for empirical studies of alternative designs. "The precision of a formal description also helps in formulating clear, testable hypotheses about design decisions."

(Reisner, 1981, p. 237) Even Carroll and Rosson, who reject the value of formal analyses of interactive systems, admit the usefulness of such analyses as the starting point for the empirical testing of interfaces.

Beyond the formal UI models themselves, other formal tools have been employed as adjuncts of UIs using a transition network control model. Alty (1984b) describes the use of path algebras for the analysis of various features of the CONNECT UIMS. A path algebra representation of the network, with appropriate labelling of the arcs, or transitions, allows certain formal features of the network to be investigated, either manually or via an automated path algebra analysis system (Alty & Ritchie, 1985b). Alty provides an example of determining the display state of the system after several changes occurring as a result of traversal of several nodes of the network, where each node causes a (partial) alteration of the display. (Alty, 1984b, pp. 125-30) If this were the only application of path algebras, it would be of little use. However, the algebra can also be used to:

i. determine the number of steps (nodes) between two states, perhaps to discover the complexity of paths,

ii. analyse the frequency of certain paths, where arcs are labelled by frequency of use in previous trials,

iii. analysing retracing of paths where points of difficulty resulting from lack of inverse actions can be identified.

For CONNECT, one of main purposes of the path algebra representation results from the inclusion of an expert system to alter the transitions available from nodes. The expert system uses production rules where the conditions are tests on the occurrence of particular path fragments, matching them against specified features of the network. The algebra provides a means by which the path fragments can be represented (Alty, 1984b).

4.5.2 **Performance Measurement**

If the testing of a UI is to go beyond the recording of users' subjective reactions, some method(s) are needed to record user and system behaviour. The user's behaviour can be monitored by trained observers, time-stamped video or by keeping a run-time log of user interactions. System behaviour, of course, requires a log of system responses to user input, time-stamped and integrated with the log of user actions.

A number of UIMSs keep logs of user interactions. If asked for, ZOG records user keystrokes and system responses, time-stamped in milliseconds (Robertson et al, 1981, pp. 475-6). Similar logging facilities are available in RAPID/USE (Wasserman, 1985, p. 712). The University of Toronto UIMS produces a file of time-stamped records, including "the x and y tablet co-ordinates of the cursor and the input event which was activated." (Buxton et al, 1983, p. 41)

The nature of the logged data clearly depends upon what is to be done with it. Keystroke, or low-level graphical interactions, can be analysed for user errors and for the patter of cursor-(hence, hand-) movement during a user session (Buxton et al, 1983, p. 41). This form of data must be collected by means of facilities built into the input device-handling mechanisms. Also, significant for some forms of evaluation are the application modules called, the commands executed and the help and editing facillities accessed. These are better logged at a higher-level in the run-time system. ZOG, for example, keeps separate data on overall system use and on a user's access of ZOG frames and options (Robertson et al, 1981, pp. 475-6).

Once collected, tools for analyzing the data are required. This may include the number and location of errors, average response time, listing of traversal paths through hierarchical menus. Buxton also notes the value of an evaluator being able to user a keystroke-level file of a user session as input to the UI, in order to "play back" and observe a user session, a facility available in the University of Toronto UIMS (Buxton et al, 1983, p. 41) and in the CONNECT system (Alty and Brooks, 1985a, p pp. 340-1). To allow such play-back, the run-time system must be designed to accept input from a specified file as well as from ordinary input devices.

4.6 **An Ideal Design Environment**

There is no such thing as an ideal design environment. The tools which support efficient and effective UI design will depend to a large extent upon the nature of the UIMS run-time system structure and the UI control model, as well as on the growing sophistication of formal analysis of usability requirements and the empirical study of user performance.

Nevertheless, one can summarize the previous discussion by establishing a set of desirable features of a design environment:

i. Where possible, design of displays should be graphical and interactive, using the same facilities available to the end user.

ii. Where the dialogue control structure allows, specification should employ editing of a graphical representation of the structure, with automatic generation of a textual human-readable version. This textual version should be directly executable or be translatable automatically into an executable specification. If necessary for performance reasons, compilation of the system should be postponed until design is complete.

iii. Prototypes should be generated automatically from the specification at any point in the design process, including automatic provision of stubs for missing semantic actions.

iv. It should be possible to make modifications to the prototype while it is running, or by suspending execution. Where possible, both the specification (in graphical or textual form) and the prototype should be visible simultaneously during modification.

v. Editing and design tools should be available in the final system. Where necessary, restriction of editing facilities should be by means of marking the dialogue specification.

vi. Where appropriate, tools should be available to analyze the formal structure of the dialogue.

vii. The run-time system should be capable of logging user interactions at lexical ("key-stroke"), syntactic (command) and semantic (application action) levels. The system should be able to accept input from a specified file as well as from physical input devices.

Chapter 5

A Proposed UIMS

5.1 Background and Overview of GUIDE

GUIDE, a Graphical User Interface Design Environment, is a design for an external control UIMS which attempts to incorporate a number of the features of the UIMSs discussed above. The initial ideas for GUIDE arose from a study of graphical input carried out by A.C. Kilgour (1983) and work done by the author on systems for the construction of simulations for Computer Assisted Learning applications. This led to a description of GUIDE (Gray & Kilgour, 1985) and preliminary efforts towards the implementation of a prototype (See Section 5.6). At present, work is continuing on the completion of the prototype and an effort is being made to secure funds to further the resesarch.

The fundamental design goal for GUIDE is the production of a system which is:

universal — all interactions between user and system are handled; all input and output are mediated by UIMS

graphical — communication via graphical entities is supported, both during design and at run-time

hierarchical — complex dialogue sequences can be built from lower-level ones

extensible — new dialogue elements can be built from pre-existing ones and kept for future use

adaptable — changes to the UI can be made without extensive programming effort and reflected immediately in the UI, thus supporting fast prototyping

A description of these goals and their connection with current systems is given for the run-time system in Section 2.6 and for the design environment in Section 4.6.

GUIDE provides a UI specification model and notation which is based on transition networks, but borrows the hierarchical structuring of the device model UIs (Section 2.4). Concurrent execution of dialogue and application is supported as is the concurrent execution of separate elements of the dialogue.

To support a fully interactive design environment, the UI specification is executed by a run-time

interpreter. The design environment allows construction of dialogues from a library of dialogue subroutines, and also permits the definition and incorporation of entirely new subroutines. The dialogue can be modified and prototyped without any special pre-processing before execution. The full GUIDE system and its components are shown in figure 5.1.

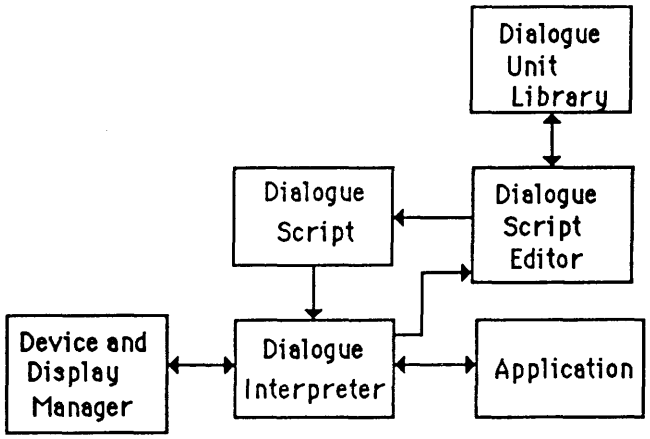

Figure 5.1 The GUIDE UIMS

The decision to favour a hierarchical dialogue structure in GUIDE suggested an implementation which embeds the dialogue specification in the Unix file system, as a collection of mostly human-readable files, each defining an element of the dialogue structure. Such an implementation makes the Unix programming tools easily accessible to the design environment, as well as providing support for concurrency and interprocess communication. We shall hereafter refer to this proposed implementation as the "Unix implementation of GUIDE," without thereby implying the existence of other implementations or implying that the current prototype incorporates all the features of the proposed Unix implementation of the full GUIDE system.

5.2 Dialogue Structure

An interactive dialogue in GUIDE is defined in terms of a Dialogue Script (hereafter, DS). A DS consists of a set of Dialogue Units (DUs). Each DU, when activated, may cause system action(s) to take place and a new set of DUs to be made available for activation by the user (or, alternatively, by the system itself). DUs are structured in such a way that a DS may represent a transition network or a hierarchically constructed "device" similar to the Input Output Tool model (van den Bos, 1980, 1983) or the dialogue cell concept of Borufka et al (1982).

Each DU consists of the following elements:

name: the name by which the DU is known to the DI

entry action: statements or processes executed when the DU is activated

option list: a list of subDUs selectable from this DU; each option in the list includes the following information:

- **name**: name of the option
- **type**: independent, subDU, endDU; each of these types may be further qualified as global(this affects the way the DI handles the option)
- **rep**: the representation of this option for selection purposes
- **condition**: an input rule which must be satisfied for option activation
- **parameter list**: a list of parameters passed to the option on activation
- **return list**: a list of values returned to the calling DU

default type: determines if static or dynamic defaults (or both) are allowed

default option: an option from the list selected by default; this is the static default; if default type specifies a static default, default option must hold an option name

exit action: statements or processes executed on DU termination

A BNF definition of a Dialogue Script is given in the Appendix B.

5.2.1 The DU name

The DU name is simply a string identifier used to uniquely specify the DU. In the Unix implementation, the DU name is the name of the Unix directory where the DU definition is located.

5.2.2 DU Actions

DU actions fall into two broad categories: communication with the user, both input and output, and communication with the application. In the first category are those functions which affect the display and which affect the input devices available via the device and display managers. These

include the following:

create_win (location, size)	a function to create a new window; returns an identifier for the new window
open_win (win_id)	displays window and makes it the recipient of input
close_win (win_id)	removes window from display and detaches input devices
clear (win_id)	clears window win_id
menu (style, loc, title, option_list)	creates a menu with specified style at loc on display with heading title and a set of selectable options given in the option_list; returns an identifier
remove_menu (menu_id)	removes menu referred to by menu_id
menu_map (loc, title)	creates a menu map (see Section 5.4); returns an identifier
add_level (map_id, option_list)	adds a level to the menu map containing the selectable options given in the option_list
remove_map (map_id)	removes menu map referred to by map_id
sample_loc (win)	returns x & y coordinates of cursor in window win; win must be open

In addition, there is a set of functions for output, including text display and line_drawing primitives. A larger and more powerful set of functions in this area is envisaged with the addition of GKS to the specification of the device and display manager.

Communication with the application is via a single function:

execute (process_name,mode, parameter_list)

In the Unix implementation of GUIDE, execution of application routines depends upon the process creation and interprocess communication facilities of Unix. process_name is the name of a file containing an executable process image or Shell script. When called, the DI forks and starts the new process. The parent process, i.e., the DI, either continues concurrently or waits, depending upon the execution mode given in the mode parameter. The parameter_list is passed by means of a socket which is bound to both DI and application process at the time of creation.

Values returned from application to the dialogue are passed by means of the same socket. In the case of concurrent execution, the DI maintains a list of all application processes created and will kill them on DU termination or abnormal abortion of the DU.

Additional actions necessary are:

rep(opt_name)	returns the representation of an option in the current option list
rep_list	returns an array of representations of the options in the current option list
dyn_default(set)	takes a boolean argument; turns the dynamic default on or off
quit	terminates the execution of the Dialogue Script
none	no effect; no action is executed

Local variables can also be declared in the entry action and are available until termination of the DU. For example, they can be passed to subsequent DUs or application processes as parameters or assigned values in the actions.

5.2.3 **The Options**

Options are themselves DUs, but the way they function is determined largely by their type. When an independent DU option is selected, the predecessor is immediately terminated after execution of its exit action and control is passed to the new DU. Thus, a DS constructed solely of independent DUs is simply a transition network, where the option condition is the transition condition. When a subDU option is selected, the result is like a subroutine call; the subDU is activated and, when it terminates, control is returned to the calling DU at the point at which the subDU was called. Since it was called while option selection was being made, the parent DU attempts once again to activate one of its options; i.e., it does not terminate. An endDU is like a subDU, except that when it returns control to the calling DU, that DU executes its exit action and terminates.

SubDU and endDU options increase the functionality of the system in several ways. First, they provide a mechanism by which a DS can be decomposed into named components. Second, a DS composed of subDUs and endDUs is inherently hierarchical. This has advantages for both the user and the designer of such dialogues in that the complexity of the resulting dialogue is limited by its hierarchical structure; the behaviour of the dialogue is more easily comprehended and

predicted. The difficulty of expressing certain interactive sequences via a hierarchical approach can be overcome both by employing independent DUs where necessary or by using certain navigation aids provided by GUIDE which enhance the hierarchial structure (see Section 5.3).

The activation condition for an option is a Boolean expression on the current input, which is held in a global environment variable accessible to the interpreter and nameable in conditions, the value of local variables or a combination of both. Devices include menus, menu maps (a special form of menu; see Section 5.3), hit areas in output windows and newline terminated textual input. Where no condition is present, a subDU will be called and will execute concurrently, allowing the simultaneous activation of several subDUs. The termination of the calling DU will cause the subDUs to be terminated. This is a departure from Input Output Tools and Dialogue Cells, both of which employ expressions over lower-level tools or cells. However, since subDUS can return a value or values to the calling DU, predicates based on these values can be used to express the termination conditions of the DU, giving the same functionality of the Input Output Tool and Dialogue Cell syntax analysis approach. The value of the GUIDE method is that one is not tied to a syntax analysis model exclusively.

As an example of this ability to model the syntax analysis approach in GUIDE, consider the following example. In the Input Output Tool model, the input rule for a tool A might include two further tools, B and C. The expression 'B & C' (see Section 3.3.4) as an input rule states that the tool A is satisfied (terminated) if both B and C are satisfied, but in any order. To express the same thing with GUIDE a further DU D would be needed. The result is as follows:

```
DU A

entry action:        boolean B_done, C_done;
                     B_done := false;
                     C_done := false;
.
.
.
option list
        DU B
                     type: subDU
                     cond:
                     ...
                     return (B_done)
        DU C
                     type: subDU
                     cond:
                     ...
                     return (C_done)
```

```
DU D
        type: endDU
        cond: B_done & C_done
        ...
```

DU D is a "dummy" DU which does nothing except provide an exit condition for DU A. DU A will not terminate until the condition for DU D is satisfied, since it is the only exit option for DU A, and DU D will not be activated until both DU B and DU C have terminated, albeit in any order.

Finally, the fact that subDUs return to the calling DU without causing that DU to terminate enables certain attractive interactive styles to be specified easily. Consider the following examples of a simple menu which makes available two options, plus the alternative to terminate. The first version uses a transition network approach and uses independent DUs, while the second employs subDUs. For the sake of clarity, no application actions are included. Figures 5.2a and 5.2b give diagrammatic representations of each example.

Example A: **Simple Menu: Independent DU version**

```
DU1
        entry action:       int win1, menu1;
                            win1:=createwin;
                            menu1:= menu (100,100, "Please choose:", rep_list)
        option list
                    DU2
                            type: ind
                            rep: "action A"
                            cond: menuhit(menu1,1)
                            param: win1,menu1
                            return: null
                    DU3
                            type: ind
                            rep: "action B"
                            cond: menuhit(menu1,2)
                            param: win1,menu1
                            return: null
                    DU4
                            type: ind
                            rep: "quit"
                            cond: menuhit(menu1,3)
                            param: win1
                            return: null

DU2
        entry action:       remove_menu(menu1)
                            display(win1,10,10, "You are at action A. Press button to
                                continue. ")
        option list:
                    DU1
                            rep: null
                            cond: button_press
                            param: null
                            return: null
```

exit action: clear (win1)

DU3

entry action: remove_menu(menu1)
display (win1, 10, 10, "You are at action A. Press button to continue.")

option list:

DU1

rep: null
cond: button_press
param: null
return: null

exit action: clear (win1)

DU4

entry action: clear(win1)
quit

option list: null

exit action: none

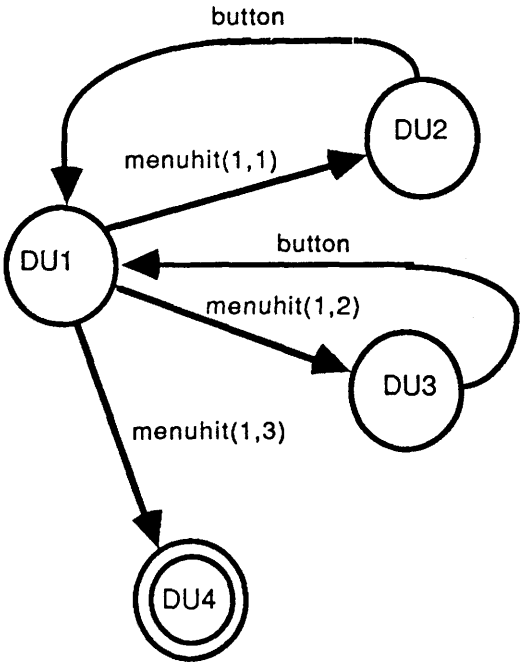

Figure 5.2a Simple menu: independent DU version

Example B: **Simple Menu: using subDUs**

DU1

entry action: int win1, menu1;
win1:=createwin;
menu1:= menu (100,100, "Please choose:", rep_list)

```
option list:
                DU2
                        type: sub
                        rep: "action A"
                        cond: menuhit(menu1,1)
                        param: win1,menu1
                        return: null
                DU3
                        type: sub
                        rep: "action B"
                        cond: menuhit(menu1,2)
                        param: win1,menu1
                        return: null
                DU4
                        type: end
                        rep: "quit"
                        cond: menuhit(menu1,3)
                        param: win1
                        return: null

exit action:            quit

DU2
        entry action:   clear(win1)
                        display(win1, 10,10, "You are at action A. ")
        option list:    null

        exit action:    null

DU3
        entry action:   clear(win1)
                        display (win1, 10, 10, "You are at action B." )
        option list:    null

        exit action:    clear (win1)

DU4
        entry action:   clear(win1)
        option list:    null
        exit action:    null
```

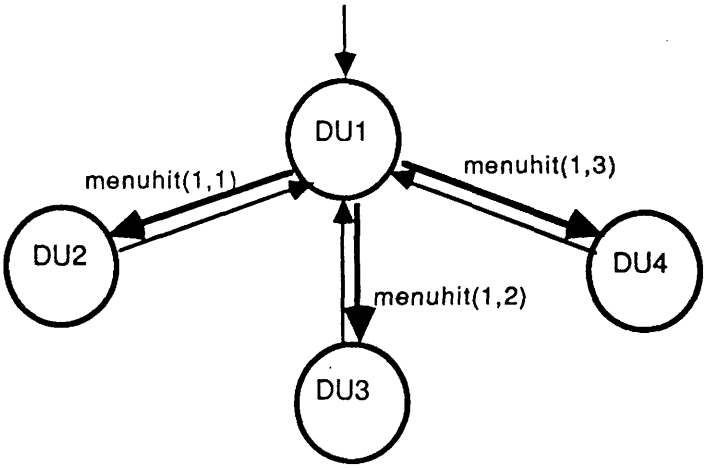

Figure 5.2b Simple Menu: using subDUs

While both versions can be expressed in GUIDE and both are equivalent in their functionality, version B uses a hierarchical structuring of the dialogue. The menu remains active after each selection without requiring any explicit transition back to DU 1. The interactive sequence ends once DU4 has been selected since it is an "end" option. The advantages of this approach become more apparent in a larger DS of which these four DUs might form a part. Thus, if control were to return to another DU after this sequence were complete, it is easier to follow the flow of control in version B where the start and end states are identical (i.e., DU1) than in version A where start and end states are distinct with, perhaps, a number of intervening states. Furthermore, the hierarchy enhancements to be discussed below are based on the mechanism provided by sub and end DU options.

5.2.4 **The Default Option**

If specified, a default option for a DU will be selected on activation without examination of input. The default will be taken only once during a given DU activation. Where a default is set for an independent or end option, the effect is equivalent to an immediate unconditional transition. Such DUs can be used for setting parameters, carrying out actions, etc. Where a default is set for a subDU, its return allows user selection of other options, perhaps to alter a value set in the default.

Defaults may be either static or dynamic or both. This is specified by the default_type. A static default value is specified during design, while a dynamic default is the last option selected by the user (i.e., during the last activation of the DU). A dynamic default will override a static default where both are available for a DU. However, dynamic defaults are only set if both specified for the DU during design and if the dynamic default flag is set at run-time; this is an action function and hence the dynamic default may be turned on or off via a DU (global or non-global) as determined by the DS definition.

5.3 **The Dialogue Interpreter**

The function of the Dialogue Interpreter is to execute a given Dialogue Script. The basic interpreter routine takes a DS and a start DU as its parameters. In the Unix implementation, only one argument is necessary, as a DS is named by the directory name of its root DU. The skeleton

form of this routine is:

```
interpret(DS)
.
.
.
finished= FALSE;
current_DU= DS;
.
.
.
while ( finished == FALSE)  {
        current_DU = execute (current_DU);
        if  (current_DU.end)
                finished=TRUE;
        }
```

The heart of the interpreter Is thus the execute function. In its simple version (without support for selection of options above the current point in the hierarchy), it looks like this:

```
execute(DU)
.
.
.
finish = FALSE;

while (finish== FALSE)  {

        entry_action(DU);
        next_DU = get_next_DU(DU);
        switch type(next_DU)  {
                case sub :      execute (next_DU);
                                break;
                case end :      execute (next_DU);
                                finish = true;
                                break;
                case ind :      finish = true;
                                break;
                case error:     report_err;
                                break;
        }
}

exit_action;

}
```

The function get_next_DU checks for a default option and, if present, returns this DU; otherwise it takes the next event off the input queue and attempts satisfy one of the option conditions, first attempting to match options from the current DU and then attempting to match any global options. If no match is found, an error is returned.

The run-time environment includes a number of features to overcome many of the objections to hierarchically organised dialogues, including

i. display of all available commands and the currently selected commands,

ii. global and subglobal commands,

iii. the ability to create at run-time default paths through the hierarchy.

These features, described by Apperley & Spence (1983) and Kasik (1983), are supported by the interpreter. The operation of default option specification has already been discussed (Section 5.2).

When the interpreter encounters a global command, it is placed on a global DU list which is maintained separately from the ordinary options from the current DU. Globals are treated like ordinary options from the current DU except that they are checked by get_next_DU after checking for a match with the options in the current DU's option list. Also, when a DU is terminated, any globals in its option list are removed from the global DU list. Thus, global commands are only available at any point in the hierarchy at or below the point where they are defined.

One objection to hierarchical dialogues is the danger of the user "losing the place" in the hierarchy, with resultant disorientation. In GUIDE, all DUs at the same or higher level in the hierarchy as the currently active DU may be selected if they are selectable from a "menu map". The notion of the menu map is similar in concept to that of the SMALLTALK browser (Goldberg, 1984). Consider the following example of a hierarchical dialogue for interaction with a simulation of a one-way linear list:

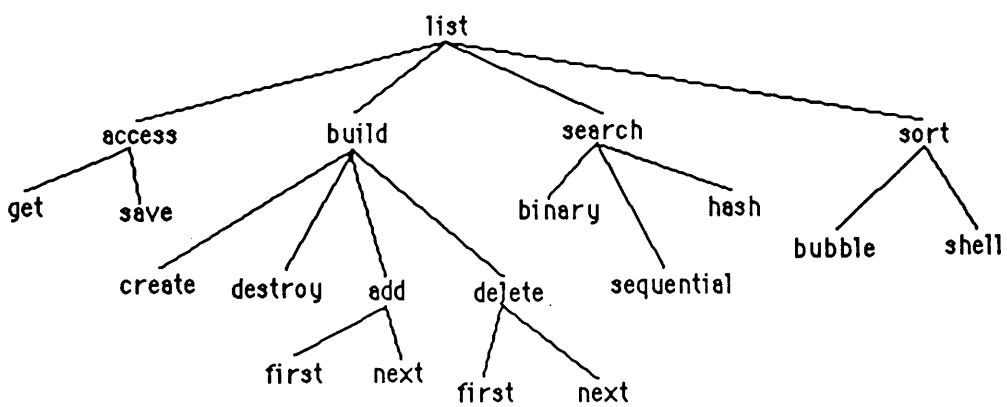

Figure 5.3 One-way linear list simulation

If all these DUs were selectable by menu map and the current DU were DU "add", then the menu map would look like this:

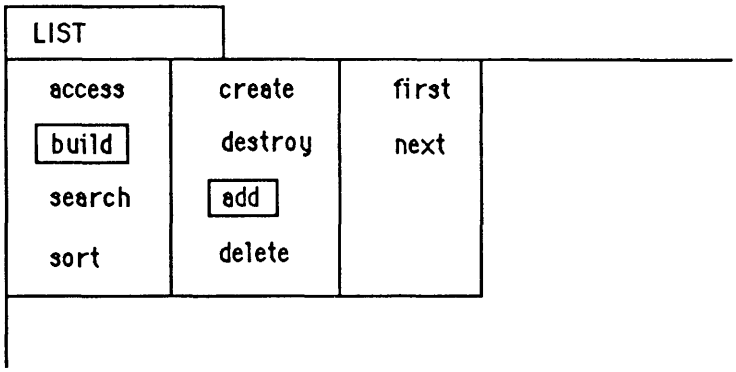

Figure 5.4a Menu map before selection

All displayed DUs are selectable. If DU "access" were now selected, the menu map would change to:

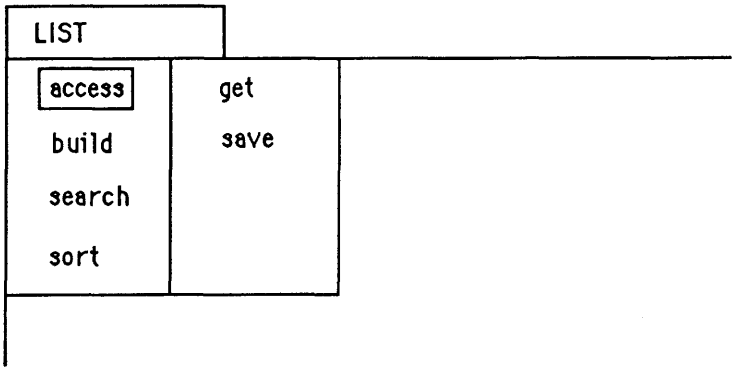

Figure 5.4b Menu map after selection

The menu map state and low-level prompting and alterations of the display are handled by the device manager.

Adding the ability to select options above the current level in the hierarchy (called "implied reject processing" by Kasik (1982))complicates the implementation of the "execute" function by requiring it to handle backtracking. The execute routine, with this backtracking facility, could now be as follows:

```
DU_type
execute(DU)
.
.
.
next_DU.uptree = FALSE;
finish = FALSE;

while (finish == FALSE) {

        if (next_DU.uptree==FALSE ) {

                entry_action(DU);
                next_DU = get_next_DU(DU);
        }

        switch type(next_DU)  {
                case backtrack: if (next_DU.name==DU.name)
                                        next_DU.uptree = FALSE;
                                else    {
                                        next_DU.uptree = TRUE;
                                        finish = TRUE;
                                };
                                break;
                case sub :      next_DU = execute (next_DU);
                                break;
                case end :      next_DU = execute (next_DU);
                                finish = TRUE;
                                break;
                case ind :      finish = TRUE;
                                break;
                case error:     report_err;
                                break;
        }
}

exit_action;
return(next_DU);
}
```

Note that the value returned by execute is now used to pass the value of the selected DU back up the hierarchy. If the selection is higher up the DS tree, function type will return the value backtrack; this will cause the uptree flag to be set unless or until the selected DU is reached. Note that this will only work with subDUs and endDUs. The uptree flag is an additional field in the DU data structure in addition to those described in Section 5.2. The DI keeps lists of currently selectable DUs (i.e., the current path down the hierarchy plus the global option list and the option list from the currently active DU). The additional fields only exist in the entries in this list, not in the DS itself.

5.4 The Device and Display Manager

The DI communicates with a separate process, called the Device and Display Manager (the DDM), which handles the collection of input and the production of output. Requests for the creation of windows and the creation and enabling of menus, browsers and pick devices are sent to the DDM by the DI. Once enabled, an input device will generate events which are placed on a single input queue for the DI. Requests for output are also sent to the DDM and include the list of functions given in the description of the action instructions in Section 5.2. Ultimately, the DDM should have available the full functionality of GKS.

5.5 The Design Environment

5.5.1 The DS Editor

Just as the ZOG frame editor, Zed, is available from any ZOG frame (Robertson et al, 1981, pp. 476-7) , the GUIDE dialogue design system is available as a global option throughout the execution of a GUIDE dialogue script. The same DS editor is used whether creating a new Dialogue Script or modifying an existing one.

Creating a new DS can be done by using the dialogue creation program which creates a new root DU containing only the global options "quit" and "edit", where the "edit" option is the DS editor. In the Unix implementation, the create_DS program takes a dialogue script name as its argument; it creates a new sub-directory with this name from the current directory, copies into it the requisite files and sub-directories for the DU elements and then calls the DI to begin execution of this skeleton DS. Editing can now proceed by selecting the "edit" option.

5.5.2 DS Editor Functions

5.5.2.1 Editing Individual DUs

When the "edit" option is selected the user is shown a frame displaying the definitions of the elements of the current DU. Each element may be edited textually. Where the representation element is iconic, an icon editor can be called up to modify this.

Options can be renamed, added to or deleted from the option list. When adding an option, it may be newly created or may be copied (i) from another part of the DS or (ii) from the DU library.

Application routines referred to in the action elements can be shell scripts or executable process images. In the latter case, they can be programs written in any language and subsequently compiled. GUIDE does not support the creation of such routines.

5.5.2.2 Sub-dialogues and the DU Library

When an option is added to the option list by copying, it may be specified as an individual DU or the entire sub-dialogue beginning at that DU. A sub-dialogue consists of a DU, its options, and all the sub-DUs at lower levels of the hierarchy beginning with this DU. Obviously, deleting an option deletes the entire sub-dialogue to which it refers.

An added option may be taken from a DU library. This consists of basic DUs and sub-dialogues, including those for:

i. a standard "menu" DU

ii. sub-dialogues for textual command entry

iii. sub-dialogues for entering graphical primitives, including lines, rectangles and polygons,

iv. a parameter collection sub-dialogue.

Additional DUs and sub-dialogues may be added to the library.

In the Unix implementation, a DU or sub-dialogue may be copied directly from another DS. This is possible since the DS description is simply a part of the Unix file structure.

Once copied into a DS, DUs and sub-dialogues from whatever source can then be edited to suit the particular requirements of the DS being designed.

5.5.3 Protoyping

Since DS creation and modification take place from within the run-time execution system, there is no separate prototyping system. However, support for the prototyping process is provided for by the fact that where the activation condition for an option is not explicitly specified, it is taken by default to be a menu option with the DU name as the representation.

Separate versions of a particular UI can be generated by creating new empty DSs, copying the DUs from the original and then making the appropriate modifications to the new versions.

5.5.4 Performance Measurement

If requested, every input event is logged, time-stamped and saved in a monitor file. Following Buxton et al(1983), when execution begins, a monitor file may be specified as the source of input in place of the standard input. When using input from this source, it may either be run in "real time", as determined by the time-stamping, or single-stepped by the evaluator.

5.5.5 Dialogue Design Agents

By making the dialogue design system part of the dialogue execution environment, it becomes possible to present users with the opportunity to alter the interface to suit themselves. It is not clear, however, that users should be given freedom to modify all aspects of the dialogue, particularly the inter-relationships or the structure of the DUs, since this may result in syntactic and logical errors which the user is not capable of correcting. Furthermore, if the dialogue has been designed with reference to the results of user performance with prototypes, the efficiency and effectiveness of the design may well be compromised by modifications introduced by end users.

An end user might still be safely allowed to make minor adjustments, e.g., to alter the representation of options, the location and size of browsers, menus and windows (if these are not already user adjustable via the window manager) and the cursor shape. Determining the degree of user adaptability of a dialogue should be the responsibility of the designer and in GUIDE this amounts to marking DU elements as editable or not by particular classes of users. In the Unix implementation, this is handled by using the file permission mechanism of the Unix file system. For the DU elements in a DS, the creator (designer) establishes the write permission for a privileged group (the designers) and for the end users. Particular sub-dialogues may also have greater or lesser write permission allowed to the user population, depending upon how crucial it is that that section of the dialogue remain unchanged. DU elements which cannot be changed are marked as such in the DS editor.

5.6 The Unix Implementation of GUIDE

As has been mentioned above, the Unix implementation of GUIDE proposes the use of the Unix hierarchical file structure for storing the DS definition. Thus, each DU is stored in a file

directory. Each element in the DU definition is kept in a separate file. Options are found in sub-directories of a parent DUs directory. Thus, to execute a DS, the DI traverses a complete subtree of the file system. This has advantages for the rapid production of prototypes, in that each element of a DU can have the Unix utilities applied to it for editing or examination.

A limited prototype of GUIDE has been implemented in C, initially on a PERQ 1, later transferred to a PERQ 2, running under PNX 2.0. The PERQ Window Management System, WMS, and the UKC "Higher-Level" Window I/O Interface were used as the presentation system. This prototype has been used just to test the interpreter algorithm and the basic mapping of the DS into the Unix file structure. It supports only menu map input and textual output and not been tested with any real applications. In particular, the interactive DS editor and the separate Device and Display Manager have not yet been implemented.

However, even this initial work has emphasized:

i. that the overheads of reading and writing to the file system are sufficiently large to make this implementation acceptable only for menu-based systems which do not require real-time graphical manipulation. The DS, however, is amenable to translation into a non-human readable form, resident in main memory, which should have acceptable response times for most applications.

ii. a sufficiently powerful graphics system is needed, probably GKS in the first instance, although problems with GKS as a presentation system suggest that it may require ultimate replacement in GUIDE (see Section 2.5).

iii. an interprocess communication facility, such as the socket mechanism in Berkeley Unix 4.2, is necessary to handle communication between the Input/Display Manager and the DI on the one hand and between the DI and the applications on the other.

5.7 **Further Developments**

The ultimate goal of the GUIDE design system is to allow creation and modification of Dialogue Scripts to take place to the greatest extent possible via graphical interaction between the designer and the system. As described above, the DS editor only allows the modification of the currently selected DU. Navigation through the DS structure takes place outside the editor. An enhancement to the editor would present a graphical representation of the dialogue structure, as a transition diagram or tree. The definition of the currently selected DU would be separately

displayed and could be edited as before. In addition, though, the designer could add, delete, move or copy DUs by direct manipulation of the graphical representation of the DS. Also, it would allow movement to new parts of the DS without exiting the editor.

The DS notation and DS editor allow the production of dialogues without the need for programming in a general-purpose high-level language. The next step is to enable the designer to interact with the display objects to produce the dialogue, "rehearsing" the dialogue somewhat like educational designers produce CAL programs via the Programming by Rehearsal system (Gould and Finzer, 1984). It is not clear at present whether or not this will require a radical restructuring of the underlying UI model, perhaps adopting a model akin to that of the EZWin system (Lieberman, 1985). However, the design environment for GUIDE might be made more interactive by replacing textual editing by interactive graphical specification where possible. For example, when a new DU is created, the designer could be allowed to choose a selection device (menu, menu map, text window). When chosen, that device would appear on the screen for locating and sizing. If a menu, options could be typed in (if text) or drawn/selected and placed (if iconic). Default options could be specified by picking. Each option could then be chosen and the design process repeated.

This would still leave the problem of specifying application actions. As Jacob has noted (1985, p. 55), this problem awaits the design of graphical languages for the specification of general-purpose programs and is beyond the scope of GUIDE. The state of developments in such visual programming environments also suggests that considerable work is needed before a workable system is produced (Glinert & Tanimoto, 1984). However, for circumscribed application areas, a library of application routines might be made available to the UI designer which could be attached to the DUs, i.e., incorporated in the action element, by simple selection.

Another area of enhancement arises from the need to increase the performance of the run-time system. As mentioned in Section 5.6 above, embedding the DS in the Unix hierarchical file structure produces overheads of file access required for reading DU element definitions which are too great for a final UI with response times acceptable to end users. Thus, after prototyping, the DS needs to be pre-processed to allow compilation into an executable program.

Systems like CONNECT (Alty, 1984) feature system adaptation of the dialogue based on user behaviour. Although the provision of intelligent knowledge-based dialogue systems or

sub-systems is not envisaged for GUIDE, it is compatible with such developments. The equivalent of switching between parallel networks in CONNECT could be implemented in GUIDE by the provision of alternative versions of the options in a DU. Determination of the appropriate option could be achieved by an intelligent sub-system of the DI which selected the option version to be used.

Appendix A

User Interface Management Systems

Name	Institution	Authors	Sections	UIMS Model	UI Model	Design	Notes
CONNECT	Strathclyde Univ.	Alty	3.4.2	Linguistic	TN	IP/IS R	text only
Dialogue Cells	Technical Univ. of Darmstadt; Amsterdam Mathematical Centre	Borufka, Kuhlmann, ten Hagen	2.4.1	Device	BNF	--	
Input Output Tools	University of Nijmegen	van den Bos, Plasmeijer, Hartel	2.4.2/ 3.3.4	Device	BNF	--	concurrency supported
Device Model of Interaction	Northeastern University	Anson	2.4.3	Device	--	--	concurrency supported
Dialogue Management System	Virginia Polytechnic Institute	Hartson, Ehrich	2.5	--	"Super-visory" Structure	GP/GS R	unique spec. system
Interactive Dialogue Synthesizer	Martin Marietta Aerospace	Hanau, Lenorovitz	3.2.2	Linguistic	BNF	--	
IPDA	Arizona State University	Olsen	2.3.5/ 3.4.3	Linguistic	TN	GP	
SYNICS	Man-Computer Interface Research Gp, Leicester Poly	Edmonds,Guest	2.3.3/ 3.4.1	Linguistic	TN	--	text only

Name	Institution	Authors	Sections	UIMS Model	UI Model	Design	Notes
SYNGRAPH	Arizona State University	Olsen,Dempsey	2.3.5/ 3.2.3	Linguistic	BNF	--	
TIGER UIMS	Boeing Computer Services	Kasik	2.3.4	Linguistic	tree	--	triply-linked tree struct
Toronto UIMS	Universi.y of Toronto	Buxton	2.3.6	Linguistic	TN	GP/I R	menus only
Univ of Alberta UIMS	University of Alberta	Green	2.3.7/ 3.5.2	Linguistic	Event/ TN		Seeheim Model
User Software Engineering System	University of California, San Francisco	Wasserman	3.4.3	Linguistic	TN	GS/R	text only
GUIDE	University of Glasgow	Gray, Kilgour	Chap. 5	Device	TN	GS/I/R	hierarchical

GP= Graphical Specification of Presentation System
I = Interactive Design

GS=Graphical Specification of Structure
R = Rapid Prototyping

Appendix B

BNF definition of Dialogue Script

<Dialogue Script> ::= { <Dialogue Unit> }

<Dialogue Unit> ::= <DU name> <entry action> <option list> <default type> <default>
<exit action>

<DU name> ::= <Unix directory name>

<entry action> ::= <action description>

<option list> ::= { <option description> }

<action description> ::= <filename of executable file>

<option description> ::= <option name> <option type> <rep> <conditon> <parameter list>
<return list>

<option name> ::= <DU name>

<option type> ::= <type descriptor> (GLOBAL)

<type descriptor> ::= IND| SUB | END

<rep> ::= <string> | < icon>

<condition> ::= <simple condition > | <simple condition> <connective> <simple condition> |
NOT (<condition> | (<condition>) |<empty>

<connective> ::= AND | OR

<simple condition> ::= <term> | <simple condition> <relation> <term>

<relation> ::= == | != | > | >= | < | <=

<term> ::= <input test> | <factor> | <term> <operation> <factor>

<input test> ::= <menu hit> | <map hit> | <win hit> | <string>

<menu hit> ::= MENUHIT (<menu id> , <option number>)

<map hit> ::= MAPHIT (<map id> , <level>, <option number>)

<window hit> ::= WINHIT (<win id>, <top left>, <bottom right>)

<operator> ::= + | - | * | /

<factor> ::= <variable> | <constant>

<parameter list> ::= <var list>

<return list> ::= <var list>

<var list> ::= <variable> {, <variable> } | <empty>

<default type> ::= STATIC | DYNAMIC | BOTH | NONE

<default> ::= <option name> | <empty>

<exit action> ::= <action description>

References

Allen, Robert B. (1982) Cognitive factors in human interaction with computers. In Albert Badre and Ben Shneiderman, eds. Directions in Human/Computer Interaction. Ablex, 1982. pp. 1-26.

Alty, J. L. (1984a) Use of path algebras in an interactive adaptive dialogue system. In Interact '84. First IFIP Conference on Human/Computer Interaction. London, 1984. pp. 321-324.

Alty, J.L. (1984b) The application of path algebras to interactive dialogue design. Behaviour and Information Technology 3,2 (84). pp. 119-132.

Alty, J.L. and Brooks, A. (1985a) Microtechnology and user friendly systems - the CONNECT dialogue executor. Journal of Microcomputer Applications, 8 (85). pp. 333-346.

Alty, J. L. and Ritchie, R. A. (1985b) A path algebra support facility for interactive dialogue designers. In Peter Johnson and Stephen Cook, eds. People and Computers: Designing the Interface. Cambridge Univ. Press,1985. pp. 128-137.

Anson, Ed (1982) The Device Model of Interaction. Computer Graphics 16,3 (July 82). pp. 107-114.

Apperley, M D and Spence, R (1983) Hierarchical dialogue structures in interactive computer systems. Software Prac & Experience 13 (83). pp. 777-790.

Barron, John (1982) Dialogue and process design for interactive information systems using Taxis. Proc of the ACM SIGOA Conf on Office Information Systems. 1982. pp. 12-20.

Borufka, H.G., Kuhlmann, H.W. and ten Hagen, P.J.W. (1982) Dialogue cells: a method for defining interactions. IEEE Computer Graphics and Applications 2,5 (July 82). pp. 25-33.

van den Bos, Jan. (1980) High-level graphics input tools and their semantics. In Richard Guedj et al, eds. Methodology of Interaction. North-Holland, 1980. pp. 159-169.

van den Bos, Jan, Plasmeijer, Marinus J. and Hartel, Pieter H (1983) Input-Output Tools: A language for interactive and real-time systems. IEEE Trans Software Eng 9.3 (May83). pp. 247-259.

Buxton, W. (1983a) Lexical and pragmatic considerations of input structures. ACM Computer Graphics 17,1 (Jan 83). pp. 31-37.

Buxton, W., Lamb, M.R., Sherman, D. and Smith, K.E.(1983b) Towards a comprehensive user interface management system. ACM Computer Graphics 17,3 (July 83). pp. 35-42.

Carroll, John M., Thomas, John C. and Malhotra, Ashok (1980) Presentation and representation in design problem-solving. British Journal of Psychology, 71 (80). pp. 143-153.

Carroll, John M. and Rosson, Mary Beth (1985) Usability specifications as a tool in iterative development. In Rex Hartson, ed. Advances in Human-Computer Interaction, Vol. I. Ablex, 1985. pp. 1-28.

Cockton, Gilbert (1985) Three transition network dialogue management systems. In Peter Johnson and Stephen Cook, eds. People and Computers: Designing the Interface. Cambridge Univ. Press,1985. pp. 138-147.

Duce, D. A. (1985) Concerning the specification of user interfaces. Computer Graphics Forum 4 (85). pp. 251-258.

Edmonds, E. A. (1981) Adaptive man-computer interfaces. In M.J. Coombs and J.L. Alty, eds. Computing Skills and the User Interface. Academic Press,1981. pp. 389-426.

Edmonds, Ernest (1982) The man-computer interface: a note on concepts and design. IJMMS 16,3 (April 82). pp. 231-236.

Edmonds, E A, Guest, S. and Pollard, A. (1984a) A User Guide for SYNICS/DDL. Leicester Polytechnic. 1984.

Edmonds, Ernest and Guest, Stephen (1984b) The SYNICS2 User Interface Manager. In Interact '84. First IFIP Conference on Human-Computer Interaction. London, 1984. pp. 53-56.

Edmonds, E. and Guest, S. (1985) The Unification of a Dialogue Manager and a Graphics System. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983. Springer-Verlag, 1983. pp. 155-159.

Ehrich, Roger W. (1982a) DMS - An environment for building and testing human-computer interfaces. International Conference on Cybernetics & Society. Oct., 1982. pp. 50-54.

Ehrich, Roger W. (1982b) DMS - A system for defining and managing human-computer dialogues. IFAC Analysis, Design & Evaluation of Man-Machine Systems, Baden-Baden, 1982. pp. 327-334.

Fahlman, Scott E. and Harbison, Samuel P. (1985) The Spice Project. In Barstow et al, eds. Interactive Programming Environments. McGraw-Hill, 1984. pp. 546-557.

Feldman, Michael B. and Rogers, George T. (1982a) Toward the design and development of style-independent interactive systems. Report GWU-IIST-82-09. George Washington University, 1982.

Feldman, Michael B., Rogers, George T., Kamran, Abid and Wenner, Patricia A. (1982b) Software engineering efforts for style-independent interactive systems. Report GWU-IIST-82-14. George Washington University, 1982.

Foley, James D. and Van Dam, Andries. (1982) Fundamentals of Interactive Computer Graphics. Addison-Wesley, 1982.

Glinert, Ephraim P. and Tanimoto, Steven L. (1984) Pict: an interactive graphical programming environment. IEEE Computer 17,11 (Nov 84). pp. 7-25.

Goldberg, Adele and Robson, David (1983) Smalltalk-80. The Language and Its Implementation. Addison-Wesley, 1983.

Gould, Laura and Finzer, William (1984) Programming by Rehearsal. Report SCL-81-1. Xerox PARC, 1984.

Gray, P. D. and Kilgour, A.C. (1985) GUIDE: A Unix-based dialogue design system. In Peter Johnson and Stephen Cook, eds. People and Computers: Designing the Interface. Cambridge Univ. Press,1985. pp. 148-160.

Green, Mark (1984) Report on Dialogue Specification Tools. Computer Graphics Forum 3,4 (DEC 84). pp. 305-313.

Green, M. (1985a) Design notations and user interface management systems. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983 Springer-Verlag, 1985. pp. 89-107.

Green, Mark. (1985b) The University of Alberta User Interface Management System. ACM Computer Graphics 19,3 (July 85). pp. 205-213.

Green, M. (1985c) Report on Dialogue Specification Tools. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983. Springer-Verlag, 1983. pp. 9-20.

Guest, Stephen P. (1982) The use of software tools for dialogue design. IJMMS 16,3 (Apr 82). pp. 263-285.

Guest, Steve and Edmonds, Ernest (1984) Graphical Support in a User Interface Management System. In K. Bo and H.A. Tucker, eds. Eurographics '84. Elsevier, 1984. pp. 339-347.

Hanau, P R and Lenorowitz, M M. (1980) Prototyping and simulation tools for user-computer dialogue design. ACM Computer Graphics 14,3 (July 80). pp. 271-278.

Hartson, H. Rex, Johnson, Deborah, H. and Ehrich, Roger W. (1984) A human-computer management system. In Interact '84. First IFIP Conference on Human-Computer Interaction. London, 1984. pp. 57-61.

Hayes, Philip J. (1985) Executable interface definitions using form-based interface abstractions. In H. Rex Hartson, ed. Advances in Human-Computer Interaction. Ablex, 1985. pp. 161-189.

Hopgood, F.R.A. and Duce, David (1980) A production system approach to interactive graphic program design. In Richard Guedj et al, eds. Methodology of Interaction. North-Holland, 1980. pp. 247-259.

Hopgood, F.R.A., Duce, D.A.; Gallop, J.R. and Sutcliffe, D.C. (1983) Introduction to the Graphical Kernal System (GKS). Academic Press, 1983.

Jacob, Robert J K (1983a) Survey and examples of specification techniques for user-computer interfaces. Comp Sci and Systems Branch, Naval Research Laboratory. Wash. D.C. 1983.

Jacob, Robert J.K. (1983b) Executable specifications for a human-computer interface. In Ann Janda, ed. Human Factors in Computing Systems. Proceedings of CHI '83. ACM, 1983. pp. 28-34.

Jacob, Robert J.K. (1985) A state transition diagram language for visual programming. IEEE Computer (Aug 85). pp. 51-59.

Kasik, D. (1982) A user interface management system. ACM Computer Graphics 16,4 (July 82). pp. 99-106.

Kamran, Abid and Feldman, Michael B. (1983) Graphics programming independent of interaction techniques and styles. ACM Computer Graphics 17,1 (Jan 83). pp. 58-66.

Kamran, A. (1985) Issues pertaining to the design of a user interface management system. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983. Springer-Verlag, 1983. pp. 43-48.

Kilgour, A.C. (1983) Graphical Input Study. Report to SERC Special Interest Group on Tools for Interactive Programs. University of Glasgow, Computing Science Dept. Report CSC/83/R1. July, 1983.

Lieberman, Henry (1985) There's more to menu systems than meets the screen. ACM Computer Graphics 19,3 (85). pp. 181-189.

Moran, T.P. (1981) The Command Language Grammar: A representation of the user interface of interactive computer systems. IJMMS 15 (81). pp. 3-50.

Newman, William M. (1968) A system for interactive graphical programming. IFIP Spring Joint Computer Conference, 1968. pp. 47-54.

Olsen, Dan and Dempsey, Elizabeth (1983a) Syntax directed graphical interaction. ACM SIGPLAN 18,6 (June 83). pp. 112-117.

Olsen, Dan and Dempsey, Elizabeth (1983b) SYNGRAPH: a graphical user interface generator. ACM Computer Graphics 17,3 (July 83). pp. 43-50.

Olsen, Dan R. (1984a) Pushdown automata for user interface management. ACM Trans. Graphics 3,3 (July 84). pp. 177-203.

Olsen, Dan R., Buxton, William, Ehrich, Roger, Kasik, David, Rhyne, James R. and Sibert, John (1984b) A context for user interface management. IEEE Computer Graphics and Applications (Dec. 84). pp. 33-42.

Olsen, Jr., D.R. (1985a) Presentational, Syntactic and Semantic Components of Interactive Dialogue Specification. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983. Springer-Verlag, 1983. pp. 125-133.

Olsen, Jr., Dan R., Dempsey, Elizabeth, P. and Rogge, Roy. (1985b) Input/output linkage in a user interface management system. ACM Computer Graphics 19,3 (July, 85). pp. 191-197.

Parnas, David L (1969) On the use of transition diagrams in the design of a user interface for an interactive computer system. Proc. ACM Conference, 1969. pp. 379-385.

Peterson, James L (1977) Petri Nets. Computing Surveys 9,3 (Sept 77). pp. 223-252.

Pfaff, Gunther, Kuhlmann and Hanusa, Henning (1982) Constructing user interfaces based on logical input devices. IEEE Computer 15 (Nov 82). pp. 62-68.

Pilote, Michel (1983) A programming language framework for designing user interfaces. ACM SIGPLAN 18,6 (June 83). pp. 118-136.

Reeves, William T. (1978) A device-independent graphics system in a minicomputer time-sharing environment. Technical Report CSRG-93, Computer Systems Research Group, University of Toronto. August, 1978.

Reisner, P. (1981) Formal grammar and human factors design of an interactive graphics system. IEEE Trans on Software Engineering 7,2 (March 81). pp. 229-240.

Robertson, G., McCracken, D. and Newell, A. (1981) The ZOG approach to man-machine communication. IJMMS 14,4 (May 81). pp. 461-488.

Shneiderman, Ben (1983) Direct manipulation: a step beyond programming languages. IEEE Computer (Aug, 83). pp. 57-69.

Short, J., Brown, R., Kilgour, A.C. & Dodani, M.H. (1982) The application of microcomputers to English essay writing: a comparison of two authoring systems. In Lewis & Tagg (eds.) Involving Micros in Education. North-Holland, 1982. pp. 53-61.

Sibert, J., Belliardi, R. and Kamran, A. (1985) Some thoughts on the interface between user interface management systems and applications software. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983 Springer-Verlag, 1985. pp. 183-192.

Strubbe, H.J. (1985) Report on role, model, structure and construction of a UIMS. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983. Springer-Verlag, 1983. pp. 3.-8.

Takala, T. (1985) Communication mediator -- a structure for UIMS. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983. Springer-Verlag, 1983. pp. 59-66.

Tanner, P.P. and Buxton, W.A.S. (1985) Some issues in future user interface management systems (UIMS) development. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983 Springer-Verlag, 1985. pp. 67-79.

ten Hagen, P.J.W. and Derksen, J. (1985) Parallel input and feedback in dialogue cells. In Gunther Pfaff, ed. Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983. Springer-Verlag, 1983. pp. 109-124.

Thomas, J.J. (1983) Graphical input interaction technique workshop summary. ACM Computer Graphics 17,1 (Jan, 83). pp. 5-30.

Thomas, J.J. (1985) Architecture for a user-interface management system. In Gunther Pfaff, ed., Proc. of the Workshop on User Interface Management Systems, Seeheim, Nov., 1983 Springer-Verlag, 1985. pp. 81-85.

Wasserman, Anthony I. (1984) Developing interactive information systems with the user software engineering methodology. In Interact '84. First IFIP Conference on Human-Computer Interaction. London, 1984. pp. 471-477.

Wasserman, Anthony I (1985a) Extending state transition diagrams for the specification of human-computer interaction. IEEE Trans Software Eng 11,8 (Aug 85). pp. 699-713.

Wasserman, Anthony I. and Shewmake, David T. (1985b) The role of prototypes in the user software engineering (USE) methodology. In Hartson (ed.) Advances in Human-Computer Interaction, Vol. I. Ablex, 1985. pp. 191-209.

Woods, W A (1970) Transition network grammars for natural language analysis. Comm ACM 13,10 (Oct 70). pp. 591-606.

Yuntan, Tamar & Hartson, H. Rex. (1985) A SUPERvisory Methodology and Notation (SUPERMAN) for human-compter systems development. In H. Rex Hartson, ed. Advances in Human-Computer Interaction. Ablex, 1985. pp. 243-281.