User Interfaces

H.J. Schouten
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

At CWI a specialised tool for the construction of user interfaces called *Dialogue Cells* has been developed. This paper describes the work, and the place of this system within the area of user interfaces.

1. Introduction

1.1. Definition of user interface

By the user interface of a computer program is meant the way the program communicates with its user. Obviously the contents of the information exchanged can be different on each execution of the program. The form of the user input may depend on the choices of the user; the form of the computer output may depend on the input values. But the allowable forms of input and the way the output depends on the input is fixed by the program and the environment. This is called *the* user interface of the program.

1.2. History of user interfaces

User interfaces have evolved in conjunction with technology. Input to the earliest computer programs was provided by setting switches. Output was the on/off status of a set of lights. The amount of information exchange was limited by the number of switches and lights of the computer. Paper tape and punched cards allowed an increase of the amount of input and output (I/O). A big step forward was the automatic translation of the I/O codes into characters. Input could be typed in and output printed. This allowed computer programs to be used interactively: the user could react to computer output immediately with further input parameters. The introduction of CRT displays allowed users to choose between temporary output on a screen, and permanent output produced on paper by separate printers. Gradually forms of input and output other than textual emerged. Graphical output could be produced both on screens and plotters. Input could be produced by pointing devices such as lightpen and mouse. Nowadays user interfaces commonly have a mixture of text and graphics output, and keyboard and mouse input.

CWI Quarterly 3, 203-218

1.3. Problems with user interfaces

The progress in computer technology has caused the limiting factor of user interface quality to be shifted from hardware to software and design. The enhanced possibilities do not automatically guarantee better user interfaces however. First it is not at all a trivial question what is a good user interface. An objective measure of quality has to be extracted from intrinsically subjective tastes of the users, in order to be able to incorporate it into the (static) definition of the user interface. Second, the programming of a user interface is, as any programming work, a hard job. The construction of user interfaces is especially difficult because it has to deal with the unpredictable behaviour of the human user. Furthermore, user interfaces tend to be complex in order to exploit the vast range of possibilities provided by the technology.

1.4. Current trends in user interfaces

The spread of computers in society has caused the group of computer users to be dominated by non-professionals. This makes the user interface a more important part of computer programs. The user interface can no longer rely either on the programming skills, or even, some knowledge of how computers work. The user interface may well become the distinguishing feature between competing programs.

The ways user interfaces adapt to these developments is (grosso modo), on the one hand by simplification and on the other hand by extension of the features. Simplification is achieved by the move from textual to graphical user interfaces. Even small home computers have system user interfaces with icons, menus, resizable windows, etc. Often a well-known situation is employed as a *metaphor* for the behaviour of a user interface. Best known is the desk top metaphor where files can be moved around as if they were sheets of paper, and icons represent objects such as printers and trash cans. Extensions to user interfaces are achieved by allowing the user to do the same thing in multiple ways and to do multiple things at the same time. An example of the former is a shortcut for (nested) menu selections. An example of the latter is the use of multiple windows, each running its own application. Although this parallelism makes the user interface per se more complicated, it can in fact simplify the work to be done by the user considerably. Consider for example the situation where the user has entered deeply into a menu hierarchy, to find out that he needs additional information to make the next selection. Without the possibility to side step into another interaction to find the information while the menu context is maintained, the user would have to backtrack through the menu hierarchy, get the information, and go back to where he was.

1.5. Research areas in user interfaces

Research on user interfaces can roughly be divided into two areas: *design* and *tools*. Here only a few of the many issues in the two areas can be touched upon.

The topic of the field of *cognitive ergonomics of human-computer interaction* is the quality of user interfaces in the broadest sense. It tries to find measures for

user interface quality and to design methods that lead to good user interfaces. It is hard to give any quantitative measures for the quality of user interfaces. However, important relations between properties of actors in the system (user, application, workstation etc.) can be given. For example, which kind of user interface is good depends very much on the kind of application and the intended group of users. In command systems newcomers tend to prefer menus, whereas experienced users often prefer command line input. The nature of the commands might pose limitations on their representation, e.g. as menus or icons. Introductions to ergonomy of human-computer interaction are e.g. Martin [1], and Card et al. [2].

Artificial intelligence techniques can be used to improve the understanding of the user by the user interface. For instance, natural language parsing allows for more human-human like conversation. Reasoning about the behaviour of users might be used to be able to adapt a user interface to a particular user.

Design and subsequent implementation of a user interface is hard without specialised tools. Two types of such tools have been developed. A *toolkit* is a set of standard interaction techniques from which user interfaces can be composed. The tools are fixed or parameterisable. They form a library of features common to many user interfaces. A glue system is provided to control the composition of the tools. Toolkits are embedded in a general purpose programming language. They usually come with window manager environments. The toolkit of the X window system [3] is a prototypical example. *User interface management systems* (*UIMS*s) are more comprehensive tools. They come from the area of computer graphics rather than window managers. They are also older. The term was introduced by Kasik [4] but comparable systems existed much earlier, e.g. Sketchpad [5]. At CWI, research has been performed on one such system, called **Dialogue Cells** (DICE for short). Therefore UIMSs are introduced more thoroughly here. Introductions to tools in general and UIMSs are Myers [6] and Hartson and Hix [7].

2. User Interface Management Systems

2.1. Separation of the user interface

The underlying idea of user interface management systems is that the user interface can be *separated* from the application. They are regarded as two loosely connected modules. The functionality of the application does not depend on how the user provides it with information. Therefore the user interface can be designed, developed and maintained independent of the application, as long as the interface between the two is fixed.

The separation of user interface and application allows the use of a specification language especially intended for user interfaces. This requires a clear view of what interaction is, what aspects are involved and how they relate. In other words, what is necessary is a *model of interaction*. A user interface specification language then can have primitives that address the aspects of the model, with its relations reflected in the language.

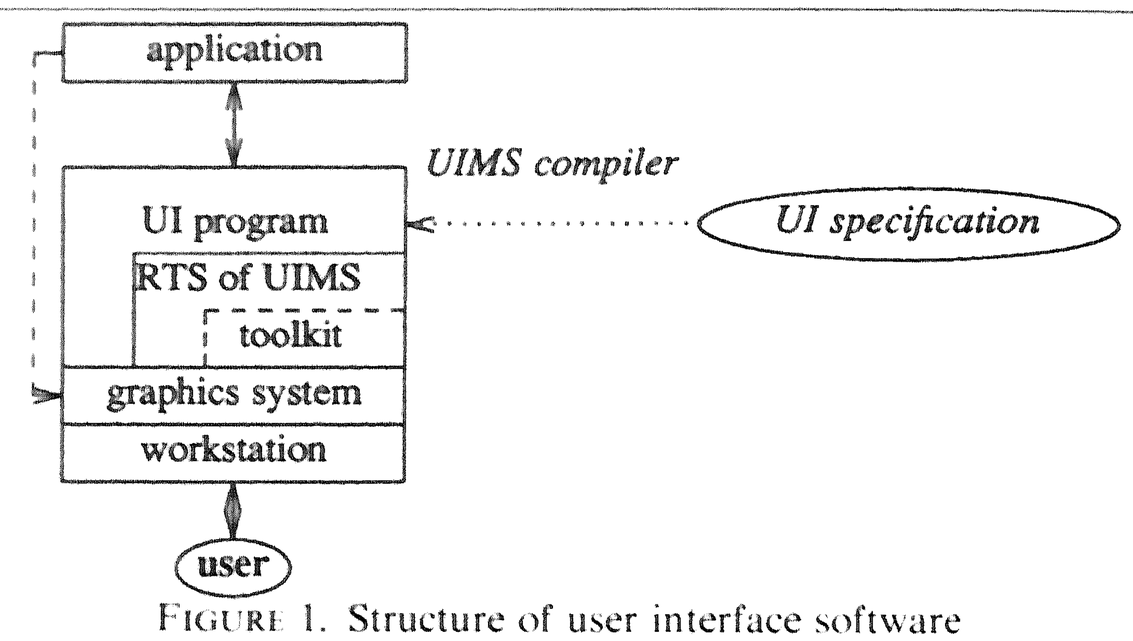

FIGURE 1. Structure of user interface software

The structure of a user interface is depicted in Figure 1. The user interacts with an application through several layers. He is in direct contact with the workstation by means of the input devices and screen. These are driven from a graphics system and/or window management system. These shield the hardware intricacies from the layers of top of them and provide the basic primitives for drawing and graphical input. Some low level interactions of the user interface are provided by a toolkit of interaction techniques. The latter is part of the run-time system of the UIMS. The run-time system is also responsible for managing the resources of the workstation. The user interface program defines the complete user interface. It defines when and which interaction techniques are available to the user, what kind of feedback should be provided, which results are sent to the application etc. Feedback of non-standard interaction techniques is directed to the graphics system directly. Some UIMSs require the application to provide the feedback; they manage only the input.

The user interface program can be specified in a special purpose language provided by the UIMS. A compiler generates the user interface program from the specification.

Apart from improvements in the design and development process of user interfaces, the recognition of user interfaces as a separate module can possibly lead to the improvement of the user interfaces themselves. Both in design and development, specialists for user interfaces can be appointed. The hope is that these user interface designers and dialogue programmers can create better user interfaces than application programmers.

2.2. Definition of User Interface Management System

In the literature the term User Interface Management System is rarely defined precisely. As a consequence (or maybe the other way around!) many different systems call themselves UIMSs. Here a User Interface Management System is regarded to be a system that is both a tool to create the user interface part of

interactive programs, and a run-time system necessary to implement the user interface constructed with the tool.

Most commonly the form of this tool is a language which is especially suited for describing interaction. The language has primitives to describe the control flow within the user interface, the layout of the screen, etc. A set of predefined interaction techniques is usually also provided. The development tools can include support for debugging and evaluation as well.

2.3. Specification of graphical interaction

Interaction can be described in any general purpose programming language and traditionally it is. But it is more convenient to describe the tasks that typically have to be performed by interactive programs in terms that better correspond to the higher level of abstraction in which dialogue programmers think about these tasks. The special purpose language of user interface management systems allows for this. Some tasks that are typical for interactive programs are:

- *Control structure of input.* This includes the work to make input values available to the application at the moment it needs them. The proper input mechanisms must be made available to the user so that it is possible for him to enter these inputs before or at the moment it is necessary. If a single complex input value for the application requires multiple inputs from the user, the way the complex value is structured into simple inputs has to be described.
- *Resource management.* To make input mechanisms available to a user, physical input devices must be allocated to the mechanism and deallocated. Likewise, the screen space necessary to show the feedback must be allocated and deallocated.

  It is a well-known problem from text based terminal use, that feedback from the keyboard can get mingled with application generated output. In graphical user interfaces, with a larger number of input devices and accompanying feedback, which is not expected to interfere with graphical output, this problem is greatly increased. Therefore it is necessary to order them in space and/or time.
- *Input mapping.* The values that the application needs may be in a format that is not usable for direct user input. A simple example is the application requiring a number, that is entered by the user as a string. In cases like this the user interface must map inputs from one format to another.

  The feedback and application output must be specified. The language interface to the graphics system used in the UIMS can be chosen such that it is easier to describe the graphics system concepts, than in general purpose programming languages.

The specification language of UIMSs have provisions for the specification of these tasks. In general the semantics of the primitives for these provisions are more complex than in general purpose programming languages.

The specification of interaction can be done in several ways. The traditional approach is to construct a text file containing the specification and afterwards

compile and run it. Specification of graphical output for user interfaces may be done interactively as well. There are other tasks that can be specified by an interactive graphical interface as well. The feasibility of this approach was shown among others by Myers and Buxton [8].

2.4. Transaction model of interaction

A UIMS works according to some model of the interaction between a user and a computer program. A user interface can be designed according to that model and then constructed using the UIMS. The model of interaction used in DICE is the *transaction model*. In this model the issuing of a command with its parameters by a user, together with the computer feedback is regarded as one entity. Such an entity is called a *transaction*. So a transaction is the execution of an interaction unit in an interactive program. Transactions can consist of a number of simpler transactions, i.e. transactions are nested. A single utilisation of an input device with its feedback is the most basic transaction from which more complex transactions are built. The dialogue as a whole is the enclosing transaction.

2.5. Model of UIMSs

A classical model of a UIMS is the Seeheim [9] model depicted in Figure 2.

The model sees a UIMS as being composed of three components exchanging *tokens*. The presentation component is responsible for the interface to the user. It manages the input and output devices. The dialogue control component manages the structure of the dialogue. It tells the presentation component which tokens may be received, and controls which tokens are accepted and in what order. The application interface component manages the interface to the application. The functionality of the application that is required in the user interface is accessed through it. It regulates the control between the dialogue control component and the application.

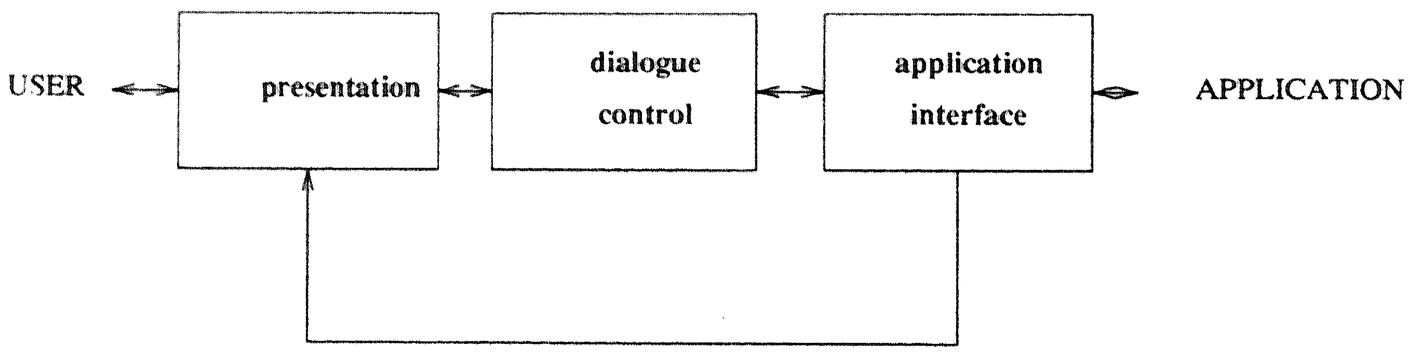

FIGURE 2. The Seeheim model of UIMS

2.6. Dialogue control model

The application interface component is the least well developed among these three components. A workshop in Seattle [10] attempted to shed more light on this, but still more work in this area has to be done. Most research in the UIMS area has been devoted to the dialogue control component. UIMSs use

a model for the control structure of the dialogues they support. This model is the basis of the dialogue control component. Much research has been devoted to the usefulness of different models for this component. The main three types of model that have been developed are *event*, *grammar* and *transition network* models. Each will be discussed briefly. An overview of these models and their relative power is given in Green [11].

2.6.1. The event model. In an *event* model, control is regulated by the occurrence of events. Events are either generated by the user or the program, including events generated inside the user interface and events generated by the application. For every type of event there is a registered function that reacts to that event. Typical functionality of these functions include the generation of feedback, the enable/disabling of event types and the generation of follow up events.

2.6.2. The grammar model. In a *grammar* model, interaction is viewed as a language which user and program use to communicate. A UIMS utilising this model provides a parser that parses the input token stream. The accepted language depends on the grammar that is provided by the dialogue programmer. The potential power depends on the class of the grammars that can be used. Context free grammars are common place, since there are well-known parsing techniques for them, but other types of grammars can be used as well. Actions can be performed on shift and reduce steps of the parser.

2.6.3. The transition network model. In the *transition network* model the user interface is in one of a finite set of states. In each state a finite set of transitions can be made, depending on the user action. When the user does an action, the corresponding transition to a new state is performed and the associated reaction is performed. The power of the model is enhanced to that of context free grammars if *recursive transition networks* (*RTN*) are allowed. The power of context-sensitive grammars can be achieved by *augmented transition networks* (*ATN*), in which each state can contain registers with associated functions. The functions can determine, based on the value of the register, which transition is to be made.

2.7. Multi-threaded dialogues

A *multi-threaded dialogue* is the conduction of multiple independent or communicating dialogues at the same time. The user is free to switch from one thread to another at any point in time. The aforementioned example of the hierarchical menus indicates that it is important for UIMSs to support this. The model of the dialogue control component must encompass the ability to do the switching. It might also provide the control over the time periods or dialogue states when the switching can be done. Such dialogues can be described easily in the event model. It can also be described by certain classes of grammars. The DICE system is an example of the latter.

2.8. Types of interaction control

Two basically different ways of how user inputs influence the flow of control in a user interface can be distinguished.

- *Internal control.* Here the program is in charge of the interaction. The transactions are initiated by the program. The program runs until it needs some information from the user. It prompts the user to enter the information and then waits until he has done that. In this scenario the user acts as a source of information to the program.
- *External control.* Here the user initiates the transactions. The program is responsible for enabling the input devices. It must provide functions for each possible type of user input to implement the reactions. In this scenario users are more free in the choice of transaction. They have the feeling of controlling the program instead of the reverse. Therefore users like this type of interaction better than internal control.

For complex transactions external control is not completely achievable. There are two reasons for that:

- It may be inconvenient to the user. If the user is busy with one coherent transaction at a time, it is not necessary to have transactions for unrelated transactions available. Furthermore, there is a logical structure over the transactions in general. Transactions may not be meaningful in certain contexts. The application normally imposes a logical ordering on the transactions.
- It cannot be implemented on workstations with limited resources. If several transactions require the same resource and this resource cannot be shared, they cannot be available at the same time. A resource is, for example, an input device. It cannot be shared if the program bases its decision for which transaction the input is on the used resource.

The usual approach to this, is to have *moded* dialogues. The dialogue is subdivided into modes. In each mode a set of transactions is available that do not conflict and which is not too complex, so that external control is possible within that mode. Special transactions are added to allow the user to navigate between modes. If the user is required to finish the transaction (or set of transactions) associated with a mode before he can navigate to a different mode, the interaction type on the mode level is internal control. Such an interaction type is called a *mixed mode* or *mixed initiative* interaction.

3. Dialogue cells

A user interface consists of a number of nested transactions. These transactions are seen as separate processes, called *dialogue cells*. When the user has performed the input actions necessary for the transaction, it produces some result. Since transactions are nested, dialogue cells are organised hierarchically.

A dialogue cell can delegate part of its work to children dialogue cells (subcells). Dialogue cells deliver their results to their parent. Dialogue cells that are the leaves of the process tree get their results from a user input value. Higher level dialogue cells compute their results from one or more subcell

results. The results of the root dialogue cell are the ultimate results of the dialogue that are delivered to the application.

3.1. The dialogue component of DICE

The DICE system has a dialogue component based on the grammar model. Each dialogue cell program defines an input language. The allowed sentences of the languages can be produced from a set of grammar rules, defined within the program. The inputs of the user are the terminals of the language. Each grammar rule defines one non-terminal. Each dialogue cell, except for a leaf of the process tree, possesses exactly one grammar rule. So there is a 1-1 correspondence between higher level dialogue cells and non-terminals in the input language. The leaves form the set of terminals of the input language. The results of dialogue cells are the tokens of the input language. An example of this is shown in Figure 3.

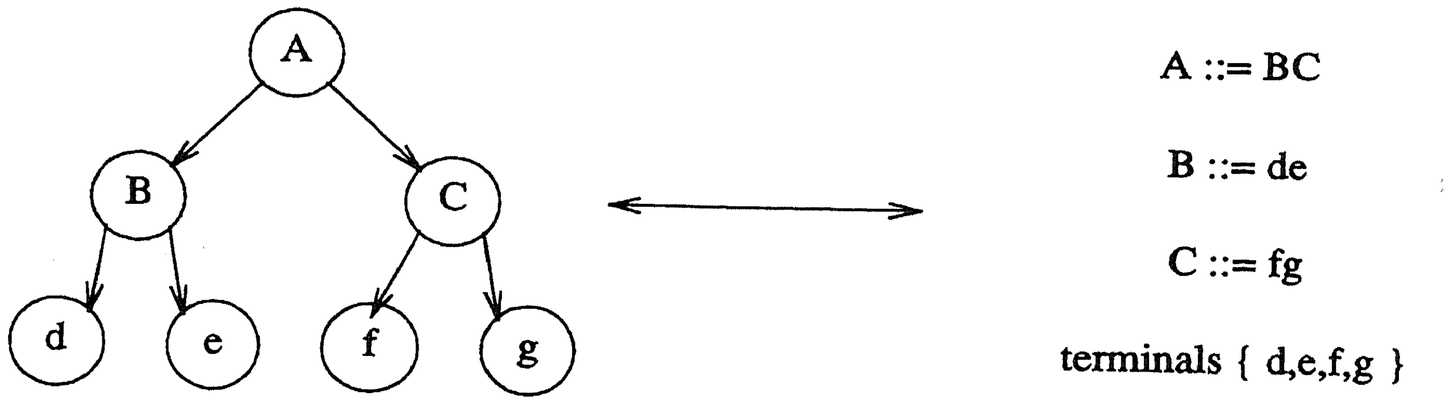

FIGURE 3. Correspondence of DICE and input grammar

The grammar rules that are used in dialogue cells are regular expressions, extended with pseudo variables. The latter allow that the language recognised by a dialogue does not only depend on the grammar rule, but on the value associated with the input tokens as well. Thus the power of the input language is extended beyond the class of regular languages. In fact, it can be proven that all context free languages and some context sensitive languages can be generated.

3.2. Activation of dialogue cells

The grammar rules of the dialogue cells determine the syntax of the acceptance of the result of children dialogue cells. The children can only produce results when they are active. Therefore dialogue cells are activated and deactivated by their parent. It is clear that if a dialogue cell's parser is ready to accept one or more of the child dialogue cells' results, these child dialogue cells must be active in order for the parser to be able to proceed. This does not mean that a child dialogue cell cannot be activated *before* its parent is ready to accept its result, which is called *early activation*. DICE supports early activation by separating the moment of activation and deactivation of children dialogue cells from the moment the parser is ready to accept a token, which allows users to

provide inputs before the program is ready to accept them. Early activation can be regarded as a generalisation of type ahead.

3.3. Description of the system

3.3.1. The grammar rules. The grammar rules of dialogue cells are regular expressions consisting of subcell tokens and operators. The operators modify the allowed ordering of the acceptance of the subcell tokens in the grammar rule. The grammar rule of a dialogue cell is called a *symbol expression*. The subcell tokens are called *symbols* in the sequel.

A *subexpression* is either a single symbol, a subexpression within parentheses or an expression with operators as described below. These will recursively define a subexpression. A symbol expression is simply a subexpression. The operators are described below. In the descriptions *A* and *B* stand for arbitrary subexpressions.

3.3.1.1. The sequence operator. The sequence operator ; is binary. It separates two subexpressions. The operator denotes sequential acceptance of the subexpressions. Subexpression *A* ;*B* means that the parser will first accept *A*, and after it has completely dealt with *A*, it accepts *B*.

The next example shows the symbol expression of a simple input technique to produce a rectangular box:

loc 1 ; *loc* 2

where both return a location on the screen result. The result of *loc* 1 is used for the fixed point of the rubberbox echo of *loc* 2. Therefore they must be entered sequentially. The resulting points together are used to form a rectangular box.

3.3.1.2. The repetition operators. There are two operators for repetition, both denoted by *. One is a postfix, and the other is a prefix operator.

The expression (*A*)**var* means that the parser accepts one or more instances of the subexpression *A*. *Var* is a boolean variable, taking one of the values *CONTINUE* or *STOP*. The value determines whether or not a new iteration is due. The variable is defined by the programmer and can be set in a trigger rule (see Section 3.4). The value assigned to the variable will in general depend on the results of the dialogue cells in subexpression *A*.

EXAMPLE.

(*pick*)* *CONTorSTOP*

where *pick* returns an identification of a picture element. Multiple picture elements can be chosen by the user until some stop criterion is met. Then *CONTorSTOP* can be assigned the value *STOP*, and the iteration is halted.

The expression *var**(*A*) means that the parser accepts zero or more instances of the subexpression *A*.

The difference between the two operators is, that in the postfix case, the parser control variable is examined after a subexpression has been accepted, and in the prefix case this is done beforehand. So they are comparable to *repeat* and *while* constructs in programming languages.

3.3.1.3. The case operator. The *case* operator is an $n-ary$ operator, with $n \geqslant 2$. It is written as

case A **of** $(B_1, B_2, \ldots, B_n)$.

Both A and B_1 mandatory. When faced with this case expression, the parser first accepts A, and then, depending on the value associated with A, accepts either B_1 or B_2 or ... or B_n. Dialogue cells symbols that are not selected are ignored.

Again the value associated with a symbol controls the parser. An integer variable must be set to determine the subexpression to be parsed next. The variable will in general depend on the value of A. With the case operator, alternative paths in the interaction can be taken. A common interaction technique for which the case operator is used is a menu. With the repetition and case operators the flow of control of the interaction can be made dependent on the semantics of the interaction.

3.3.1.4. The or operator. The | operator is a binary operator. When the parser reaches the expression $A|B$, it accepts either subexpression A or subexpression B, whichever is available first. If both are available at the moment the parser reaches the expression, one is chosen nondeterministically. The or operator allows for the provision of multiple interaction techniques for the same type of result.

EXAMPLE.

loc 1|*coordinates*

where *coordinates* is a subcell accepting coordinates typed in on the keyboard. *Loc* 1 can be used to enter the location by a pointing device.

3.3.1.5. The and operator. The & operator is a n-ary operator, with $n \geqslant 2$. When the parser reaches the expression $A_1 \& A_2 \& \cdots \& A_n$, it tries to recognise all A_i, $i \leqslant n$ but it accepts them in any order. The whole expression is recognised when all A_i have been accepted. Multiple symbols may be available before the parser is ready to accept them. The parser will accept them in a nondeterministic order. By using this operator the user is not forced into a particular order of providing inputs, when the application does not require so.

3.3.2. Activation moment control. The moment of activation and deactivation of dialogue cells is controlled by the parent's symbol expression and the *activation mode* of the dialogue cell. The activation mode has one of the values *request*, *event* and *sample*. These will be explained below.

The activation mode is determined by the parent; different instances of the same dialogue cell can be active in different modes. The activation mode of the parent has no effect on the activation mode of the children. In the hierarchy of dialogue cells any combination of modes is allowed.

3.3.2.1. Activation in event or sample mode. When a cell is activated in event or sample mode, it is activated immediately after its parent is activated and deactivated immediately before its parent is deactivated. The difference between event and sample mode is semantical. If a subcell is activated in event mode, the parent gets a result when the subcell provides one; the parent obtains a result from a subcell in sample mode immediately without waiting for it to provide one. For the moment of activation there is no difference.

3.3.2.2. Activation in request mode. The moment of activation of a dialogue cell in request mode depends on the symbol expression of its parent. It becomes active immediately before the parent's parser reaches its symbol in the symbol expression. It is deactivated immediately after the parent has received its result. The activation of dialogue cells in request mode in complex symbol expressions is described below.

For the subexpression $A;B$, dialogue cell A is activated as soon as this subexpression is reached. A is deactivated when its parent has received its result and then B is activated.

If the subexpression is $*(A)$, A is activated when the stop criterion is not met. A is deactivated after its parent has received its result. This means that for each iteration, A is activated and deactivated. For the subexpression $(A)*$ the activation and deactivation scheme is identical.

For the subexpression **case** A **of** $B_1,\ldots,B_n$, A is activated immediately when this subexpression is reached by the parser. It is deactivated when its parent has received its result. If, in the case expression, A selects B_i, B_i is activated immediately when it is selected, and deactivated when its parent has received its result. No other B_i's in request mode are activated.

In the subexpression $A|B$, both A and B are activated as soon as the parser has reached this subexpression. Both A and B are deactivated as soon as their parent has received the result of either A or B.

Similarly, for subexpression $A\&B$, both A and B are activated as soon as this subexpression is reached. They are deactivated when its parent has received its result. The & and | operators, and the event and sample activation modes provide ways to activate dialogue cells in parallel. They allow for the implementation of external control and mixed initiative type dialogues.

3.4. Trigger rules

Whenever a symbol of a symbol expression is parsed, an associated semantic action can be performed. This is similar to formal language parsing, where the associated actions generate the object code. Here the semantic actions generate the reaction of the dialogue program to the user input that caused the production of the parsed symbol.

The actions are said to be *triggered* by acceptance of a symbol by the parser. The actions together with the identification of the subexpression with which they are associated, are called *trigger rules*. From within the trigger rules the possibility exists to communicate with the application program. This serves two purposes:

First, the application can be requested to perform a calculation using a value returned by a subcell. It may be not practical to let the dialogue cell perform the calculation itself; this might necessitate copying a substantial part of the application functionality. The purpose of the calculation may be the transformation of the result to an appropriate data type, or checking the result for semantical correctness. The result may also be used to perform semantic feedback.

Second, an intermediate result may be passed to the application. If this was not possible, the dialogue would have to stop here. A new dialogue would have to be started for the rest of the interaction. This is inconvenient if a large amount of context is shared by the interactions.

3.5. Resource management

Resource management was mentioned as one of the tasks of a UIMS. The DICE system provides a method of specification of hierarchical resources and the run-time support system includes a sub-system to realise the resource specification.

There are two types of physical resources that are typically scarce for an interactive program: (graphical) output resources and input devices. Among the tasks of an interactive program is their management. A DICE program consists of a number of concurrently working, rather independent processes, each possibly requiring resources. Some may compete for the same resource. This needs to be resolved in a systematic way, so that resource conflict resolution can be automated. Each dialogue cell needs both screen space (to show its feedback) and input devices (to obtain its input values). Therefore a dialogue cell has a resource which is a combination of a window and a set of input devices.

A dialogue cell can request one of a specified set of windows. This allows the same dialogue cell to be used in multiple windows. A window can be defined to be relative to the dialogue cell's window, or to be on a fixed position. The first option allows the visible part of a dialogue cell to be contained in that of its parent, so that the dialogue cell hierarchy is conveyed to the user. The second option is available to dialogue cells that need a window at a fixed position and with fixed size, such as dedicated error and help message windows. Within an absolute window, a window hierarchy can be started again for more complex dialogue cells. If no window is requested, the parent's window is inherited.

A dialogue cell can request one of a specified set of input device sets. Similarly to windows, this allows the same dialogue cell to be used many times each possibly having a different set of input devices. An input device set is that which is needed by the dialogue cell and its children to perform its work.

Therefore it is always a subset of the set of input devices of the parent.

The user is in an ambiguous situation when he/she can perform an input that can have multiple meanings. In a dialogue cell program this means that multiple dialogue cells utilise the same input device, while no distinction can be made based on the window. This is undesirable because the user cannot predict what the result of the input would be. In DICE such a situation cannot arise, because every dialogue cell is assigned a unique resource; no two active dialogue cells can have both the same window and the same input device. This uniqueness is also used to resolve ambiguities in the input language grammar.

3.6. Basic dialogue cells

A dialogue cell program effectively defines a language (i.e. the language that the computer program and the user use to communicate). The terminal symbols of this language are the dialogue cells that have no subcells. These are called *basic dialogue cells* or *basic cells* for short. They take their input directly from the user. As any other dialogue cell, the input type and the feedback is described completely in these cells. They rely on the graphics system underlying dialogue cells to realise these descriptions, and to actually obtain the input value.

It might not be possible to implement some basic cells on some workstations, because they require resources that are not available on the workstations. Therefore the definition of basic cell is extended in two steps to avoid it to be workstation dependent.

A basic cell that is not directly implementable on a workstation can be simulated by using other basic cells that are available. By keeping the interface to the dialogue cell the same, higher level dialogue cells can use this cell the same way as if it was directly present on the workstation.

Since the capabilities of workstations are growing continuously, it is conceivable to have dialogue cells that are not directly implementable on any existing workstation, but may be in the future. This can only be the case for dialogue cells that do not rely on a particular application program, i.e. dialogue cells that have no interface to an application. Therefore any dialogue cell, that does not depend on an application, is called a basic cell as well.

Now an interaction technique is simply a basic cell. A library of basic cells is provided with the dialogue cell system. It serves the same purpose as a toolkit.

3.7. Dialogue cell picture environments

The dialogue cell system is intended to be device independent. Although resource availability changes with workstations, by the appropriate choice of dialogue cells, a programmer can specify a dialogue that can run on several workstations. Therefore the I/O is specified in a device independent way. Needless to say that it will in general be graphical. To support this, a device independent graphics system is used as part of the run time support system, which is called the *radical system*.

A dialogue cell being a process has its own data set, which includes its graphical output. Such a disjunct output data set, together with the functions operating on it that do not interfere with other output, is called an *environment*. An environment is like a small, local graphics system of its own. However, it has implicit relations with other existing environments. These relations determine how coexisting local environments share a workstation. This includes the ordering of picture elements from different environments on the screen and the determination of which environment will receive which input. Furthermore, environments are related to be able to exchange output.

Environments form the basis for handling parallel output and input; for each dialogue cell one environment is created. The functions within an environment can execute in parallel, which allows modular specification of the dialogue cells.

Every environment has its own world coordinate system. Output of an environment is specified in these coordinates. The extent of a world coordinate system is defined by the process owning the environment as a rectangular area aligned with the principal axes. During display, world coordinates are mapped to device coordinates by a mapping which maps the world coordinate system extent to the window of the dialogue cell.

A *radical* is the central picture element of the radical system. The term radical originates from chemistry. It is chosen to indicate that radicals react vividly with their environment.

A radical is a compound picture element consisting of an arbitrary number of output primitives (polylines, polymarkers, etc.). No primitives can exist outside radicals. Radicals and primitives have attributes that control their appearance.

4. Conclusion

In the DICE system the concepts of grammar based interaction control are combined with concurrent dialogues. The former allows an interaction language to be specified directly as a grammar, while the latter improves the freedom of choice for the user. The hierarchy of transactions models the way the result of the interaction is obtained from simple inputs to ever more complex structures. The employed context independent resource model allows dialogue cell programs to be constructed from reusable components, such as a library of basic cells. The DICE system has been implemented on a number of workstations and window systems. Plate VIII shows a snapshot of an interactive session programmed in the dialogue cell language.

The system does not guarantee good user interfaces, but provides a way to construct them. The emphasis was not on generally useful elements of user interfaces, such as help systems and undo support, but rather on the basic layer in order to be able to create such features. A next step might be to investigate how well these things fit in DICE. The strictly hierarchical process organisation turned out to be unwieldy for some applications. Research is ini-

tiated to how this can be made more flexible, without sacrificing the interaction model. This must result in the successor of DICE, called *Transaction Cells*.

REFERENCES

1. J. MARTIN (1973). *Design of Man-Computer Dialogues,* Prentice-Hall.
2. S.K. CARD, T.P. MORAN, A. NEWELL (1983). *The Psychology of Human Computer Interaction,* Lawrence Erlbaum.
3. R.W. SCHEIFLER, J. GETTYS (1986). The X window system. *ACM Transactions on Graphics* (April).
4. D.J. KASIK (1982). A user interface management system. *Computer Graphics 16 (3),* 99-106.
5. I.E. SUTHERLAND (1963). SKETCHPAD: A man-machine graphical communication system. *Proceedings SJCC '63.*
6. B.A. MYERS (1989). Tools for creating user interfaces: an introduction and survey. *IEEE Software* (January).
7. H.R. HARTSON, D. HIX (1989). Toward empirically derived methodologies and tools for human-computer interface development. *International Journal of Man-Machine Studies 31 (4),* 477-494.
8. B.A. MYERS, W. BUXTON (1986). Creating highly interactive and graphical user interfaces by demonstration. *Proceedings of SIGGRAPH '86,* 249-258.
9. G.E. PFAFF (1985). User Interface Management Systems. *Proceedings of the Seeheim 1983 Workshop on UIMS's.*
10. M. GREEN ET AL. (1987). Workshop on UIMS. *Computer Graphics, Special Issue.*
11. M. GREEN (1986). A survey of three dialogue models. *ACM Transactions on Graphics 5 (3),* 244-275.