Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P.J.W. ten Hagen, J. Derksen

Parallel input and feedback in dialogue cells

Department of Computer Science Report CS-R8413 July

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

PARALLEL INPUT AND FEEDBACK IN DIALOGUE CELLS

P.J.W. TEN HAGEN
Centre for Mathematics and Computer Science, Amsterdam

J. DERKSEN
The Netherlands Organisation for Applied Scientific Research (TNO-IBBC), Rijswijk

ABSTRACT

In this paper a specification method for interaction is outlined based on a new programming concept called 'dialogue cells'.
The method supports parallel input and separation of a dialogue part from the algorithmic part of an application.
Graphics interaction can be fully integrated. Some problems associated with parsing parallel inputs are analyzed.

1982 CR CATEGORIES: I.3.6.

KEYWORDS & PHRASES: Interaction, Man-Machine Communication, Computer Graphics.

NOTE: this report will be submitted for publication elsewhere.

Report CS-R8413
Centre for Mathematics anc Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1 Introduction

The programmer of an interactive program has a complex task to perform. In addition to the specification of correct, efficient algorithms he must pay continuous attention to the way the program will manifest itself to the observing user when run and by what means the user can influence the problem solving process. The success of the program depends as much on how easily the user can interact with it as on the efficiency and correctness of the algorithms.

In this paper we will concentrate on techniques for programming the interactive part of a program. We will not attempt to define what constitutes high quality interaction. This is a multidisciplinary subject. Psychologists (perception, cognition), ergonomists, artists, application specialists, etc., all have something to say about this. Very likely no two of them will agree about which are the more important rules and issues.

The techniques that will be presented, and the fundamentals for such techniques, ideally will be such that they can be used or misused for implementing any type of interaction. At best the programmer may discover that the techniques he is using follow some rules of thumb such as:

- give the user value for his efforts. (e.g. keep the amount of work he has to do to obtain some information sufficiently small).
- present only relevant information; remove rubbish.
- don't be ambiguous without reason.

However, no systematic is known. Sometimes one rule prevales, sometimes another. Also, many of the principles can be traded off against each other.

The particular type of interaction needed for a succesful interactive program, may depend on the application or on the available hardware. This implies that a programming method which aims at portability and device independence should not exclude any particular type of interaction.

2 User Interface

The user interface to a program is the collection of visualisation and manipulation facilities and their interpretation (right or wrong) given to them by the user. Each of these facilities as perceived by the user is made up of four entities: the physical part of a facility, the way it can be used, the way it is controlled by the program, and, its contribution to the interactive process.

The fact that each facility depends on a physical realisation is reason for questioning by many people whether generally good interfaces can be specified in a device independent manner.

Also many people believe that good interfaces cannot be described unless taking into account the general model of the whole program and the role of each interaction facility in such a model. The problems here are that different users may perceive different models and that such models are not a bases for specifying (designing) interaction, because they are difficult to formulate.

The optimists however, consider the capability of human beings to adapt and to use abstractions sufficient in order for interactive systems to be realised on a wide

variety of devices and to work with a great variety of user models. The techniques discussed in this paper attempt to support the optimists view.

The attention of researchers and system designers has, with respect to user interfaces, shifted from the conceptual model to the realm of devices and interaction techniques. The recognition that the techniques used are substantial in determining what you can say, or, that communication and expression are as important as understanding has emerged with the use of rich interaction devices.
The computer graphics experts are for the time being also the experts who can build user interfaces. It is through graphics that communicating with computers has become fun. That is to say, it has become a fun for non-experts in programming. Providing good interactive techniques is essential for good and effective interfaces. However, understanding is also essential. Many good interfaces are based on exhibiting that the user is understood (e.g. natural language interfaces). This shows that interactive techniques are ultimately to be coupled with the process of understanding.
This paper will concentrate only on the interactive techniques.

3 Interaction and Graphics

3.1 Properties of graphics interfaces

One may wonder what is so special about a computer interface involving graphics.

The first important property of communicating through pictures is the directness. That is to say using pictures removes at least one level of encoding-decoding. Pictures are much closer to the concepts one wants to convey, be it for input or for output. One may consider both the method of constructing a view from a picture description, or constructing a picture description from input activities, as local activities, which have only to do with the syntax of an interaction language. This is a very important justification for situating graphics viewing facilities both for input and output in a local terminal, to be used by arbitrary applications. The fact that the CORE -once proposed- standard and to a lesser extend the GKS standard failed to support picture input at the same level as picture output, had more to do with the state of the art, than with the principles underlying the standard.

A second important property is the non-sequential nature of pictures. Two or three dimensional objects cannot be linearly ordered. Therefore a user cannot be expected to react only to the last item produced. In principle he must be able to act upon every aspect of the information presented.
The program must be prepared to receive input referring to items produced earlier but still visible. It is up to the program to reconstruct the association with the semantic context.

A third property is the opportunity to convey a meaning by changing something rather than adding something new. Hence manipulation and 'gesture' become part of the dialogue.

Last but not least the syntax of a graphics dialogue should allow many degrees of freedom and variable feedback. So that adaptation to user skills becomes feasible.

By no means can one say that today a general solution and method exists for exploiting these possibilities in interactive systems.

3.2 Introducing parallelism

The specification of complex inputs and interactions, such as those involving pictures, can be adequately done by using parallel programmings concepts. In some cases the parallelism can be convenient. However, for the general case, we intend to show it being necessary.

Modern graphics packages, like GKS, allow for basic input devices to be active in parallel. Each of these input devices has associated with it an interactive (basic) technique to produce the input items of the device. The operator at a workstation can make his own decision about which input to produce next. He might even decide to produce several inputs simultaneously.

On a higher syntactical level more than one picture may be under construction. At that level for a given picture it is immaterial that inputs for another picture were produced as well. The only way to abstract from these irrelevant details is by applying a parallel model. This will simplify and make possible the independent programming of simultaneous picture constructing processes.

A similar argument applies to a user reacting to several pictures presented on a screen. If different processes or subprograms are associated with different pictures, each one is assumed to only process the corresponding reactions. This may imply that these processes are concurrently active.

A user defining a correlation between two items presented may trigger both associated subprograms. This is an example of conveying a message by changing the information as presented. As a result changes on more than one place may occur simultaneously.

The most complex use of parallelism occurs when one tries to maximise user freedom in specifying input sequences. In between given synchronization points all permutations of an input sequence may be acceptable. This means that between these points all mechanisms that can produce symbols for the sequence must be active. If not, then the user is forced to activate every mechanism himself prior to producing an input with that mechanism.

We will present some aspects of a dialogue specification language which facilities for parallelism for the situations mentioned above.
In particular we will describe the problem of parsing input under parallel conditions and how parsers can be built to solve these problems.

4 Dialogues

In these notes a human-computer dialogue, or dialogue for short, is the exchange of information (commands, results, explanation) between a human operator and an interactive computer program in execution.
As part of writing an interactive program it is specified which information can be exchanged. Besides defining algorithms, a programmer also defines a dialogue language, i.e. he specifies which exchanges can take place at any time and what their effect (=meaning) will be. We can therefore consider a program to consist of two parts: the algorithmic part and the dialogue part. Whether both parts can be mutually separated depends on the program structure.

The state of the art very much is that the algorithmic part of a program is designed

before it is implemented whereas the dialogue part is merely designed while implemented.

For a proper method of designing a dialogue language prior to implementation three conditions have to be met:

1. The dialogue part must indeed be a separate program section (not interwoven with the algorithms).
2. The programming method for the dialogues must allow for easily readable dialogue specifications, so that a programmer can conclude what happens at the user interface when reading the program text. This is a 'structured programming' requirement for dialogues.
3. The constructs for dialogue programming must provide complete control over the physical part of the user interface, with or without device independence.

Generally speaking the program structure required for algorithms differs from dialogue structure. For instance, the algorithm may ask for parameters in a format and order quite different from the way most users would naturally provide them.

4.1 The separation of algorithm and dialogue

A separate dialogue module would provide a number of advantages:

- The algorithmic part is relieved from all details concerning in- an output such as: test whether input is correct, check whether output is possible right now.
- One can alter the dialogue part (e.g. adapt to new hardware), without having to change the algorithms.
- Dialogue specialists can write a dialogue most suitable for a given environment and/or application.
- Interactive techniques may become available as (sub)dialogue libraries.
- A dialogue run time support system may provide facilities for user help, error recovery and audit-trails.

The most simple and maybe adequate method for separating algorithms from dialogues is by introducing dialogue procedures. They would, when called, look quite similar to, say, subroutines.
However, the dialogue procedure itself may be specified by a method or language quite different from the general purpose programming language used for the algorithms.

Dialogue procedures provide a structure different from input- or menu-driven systems. Here the program block- and subroutine structure can be seen as a directed graph (the arrows are the procedure calls or block nestings), the top levels of which are the input handlers.

One can imagine that dialogue procedures in turn can call algorithmic routines, for instance, to do conversions of input data. At run time dialogue and algorithm can be conceptually two cooperating processes which exchange messages (=procedure calls). The language construct that will be presented here for specifying a dialogue can represent dialogues where both processes can cause side effects in the other.
Current practice does not allow for such complicated structures. Especially situations requiring backtracking over algorithms or dialogues cannot be dealt with satisfactory. A better strategy seeks to avoid the necessity of such mechanisms.
As we will see this can be achieved without becoming too restrictive.

4.2 Dialogue Cells

The language construct that will be presented here for specifying a dialogue procedure is called **dialogue cell.**
A dialogue cell is a unit which can completely specify one step in a dialogue. Each step comprises the following parts:

- an action from the user,
- the corresponding external reaction from the system,
- the effect on the internal state of the system
- the environment and conditions in which the action takes place.

Dialogue cells can be organised in hierarchies. This means that a big unit can be built from a combination of smaller units. In a hierarchy the way smaller units are combined can be precisely controlled. The overall action of two combined units is determined by the individual actions and the combining operator. This hierarchy therefore allows at the same time the specification of dialogue syntax (how words combine to sentences), the internal semantics (how a structured state vector is changed component by component), how the effect of changing the visible interface is built from little changes and last but not least how sub-dialogues get activated and deactivated.
The four groups of dialogue cell activities as given here are specified in four separate cell components that make up a dialogue cell.
These components are:

Prompt: The section which initialises the sub-dialogue and activates all sub-cells required. In addition it informs the user that it is ready to accept the input.

The appearance of the prompt can vary from a simple 'continue' sign to a whole new screen set up plus initial message (question) for the user. This means that it can express where the initiative lies, or, who asks the question and who provides the answers.

Symbol: The section which specifies the syntax of an input sentence. It will tell how the input monitor will try to read the input words and when the dialogue cell input is completed.

First of all specifying how the system will read symbols is different from specifying how (and in what order) a user must produce the inputs. Only in strictly synchronous, sequential systems will these two be the same.
Sub-symbols (i.e. 'words' of the sentence) are produced either by the user directly (basic symbols) or are obtained from sub-cells. The mechanisms to provide basic symbols are integrated in the dialogue system as special dialogue cells, so-called *basic cells.*

Echo: The acceptance of a symbol by a dialogue cell causes a message to be sent back to the user.

These pictorial messages (echoes) may remain visible when the dialogue cell is completed (global echo) or may have to be removed (local echo) when the cell is deactivated. Thus the echo mechanism is the basis for dynamic control of the screen.

Value: Each symbol accepted has a value associated with it. This value may with or without conversion contribute to the complex value to be produced by the cell.

Also here a local-and-global value scheme is used which is controlled through the

cell hierarchy. The value section contains an extensive mapping mechanism including input validity checks. The result of mapping and checking may be fed back to the symbol section by means of synchronisation tokens which can be used to select among alternatives or end repetitions. In this way attributed grammars can be handled.

4.3 The Basic cycle of a Dialogue Cell

What happens when a dialogue cell becomes active?
The answer is: its four components become (simultaneously) active. However, implicit in the dialogue cell semantics are synchronisation rules and some form of abstract data typing. We will give an informal description here because the understanding is more important than the precise form.

In each of the sections names of sub-cells may be used. It is understood however, that when a prompt part names a dialogue cell it only refers to the prompt section of that sub-cell. A similar rule holds for all other sections.

The next rule is that only symbols appearing in the symbol part of a cell may appear in the other parts. This guarantees that only cells are activated which inputs can be read and also that values and echoes of cells that are to be produced stem from active cells.

An additional very important rule states that symbols can only be read when a cell is active (i.e. the prompt section must be executed before symbol parsing) and the value and echo actions specified for a symbol must be triggered by the parser. This guarantees that value expressions cannot be evaluated further unless all values required become available and all output echoes (e.g. pictures) produced are either prompts or echoes.

A refinement of this rule allows for mutual synchronisation between value and echo, so that the change to the internal state can be visualised. An active cell proceeds for each symbol in the symbol expression through the sequence, $P \rightarrow S \rightarrow V \rightarrow E$. When the last symbol is encountered the cell will deliver a symbol, value and echo to the external environment.
After delivery of the external results the cell is ready for a new complete basic cycle. Whether this will happen or not is controlled by the surrounding cell.

4.4 The activation of dialogue cells

A dialogue cell, as we have seen, is either activated by the algorithmic part as dialogue procedure (root-cell), or it is activated as sub-cell of a dialogue cell closer to the root-cell. This activation can generally be either synchronous or asynchronous. In the synchronous case a cell is activated just prior to the action by the symbol part to accept this symbol. The activating dialogue cell is forced to wait until the symbol asked for has been produced. The cell is deactivated immediately after its symbol has been accepted. This activation mode is a generalisation of the REQUEST mode defined for GKS input. Similarly the asynchronous activation is a generalisation of the GKS concepts SAMPLE and EVENT.

All asynchronously used sub-cells of a dialogue cell are activated in the prompt section as part of the initialisation. So all these subcells are active simultaneously. This has two important consequences:

1. The user is allowed to produce input long before it is being used and he **can** produce input in an order quite different from the order in which the symbol part will accept them. This is indicated with the term **early activation.** Symbols produced before they will be used are called **early symbols.**
2. Symbols may be produced in parallel, the symbol expression of many cells may simultaneously attempt to accept the symbol produced for it. Therefore the symbol parsing process goes on in many rules at the same time although ultimately one start symbol will be produced. This requires a so-called dynamic multi-stream parser.

In order for both user and programmer to limit the complexity of this parallel parser a number of restrictions must be imposed on the dialogue cell semantics. These will be discussed in the next section.

Simultaneously active dialogue cells model the actual situation of many graphics workstations, having for instance, a keyboard, menu on the screen and a locator all simultaneously enabled. They then leave it up to the user which one to select.

A dialogue cell activated in SAMPLE mode puts the initiative with the computer program. The program can decide at any time to 'sample' the value produced, even before the user has provided one, or, after the user has provided several, in which case all but the last get lost.

A dialogue cell activated in EVENT mode gives the initiative to the user. Firstly the program is forced to wait if no symbol is yet produced. Secondly, the program is forced to read all symbols produced one after the other from a queue. Unlike GKS, the various event queues of the individual dialogue cells are not merged into one.

5 Input

5.1 Input Parsing

The symbol part of a dialogue cell consist of a grammar rule. This rule directly controls the input parser. Two kinds of problems with respect to input parsing must be dealt with.
The first one has to do with ambiguous grammars. An ambiguous situation occurs when a symbol is produced which might fit in more than one alternative. Then the parser cannot decide where to continue. In addition, a parallel parse over a non-ambiguous grammar may introduce ambiguities, because more than one active rule (= symbol part of the active cell) may require such a symbol. Ambiguous situations are generally speaking, not wanted in interactive dialogues because they leave the user confused. They either force the system into multiple reaction (try all possibilities for some time) or they postpone one reaction or they choose one arbitrarily. Each of those has its own contribution to confusion to make.

A user who makes an error in an ambiguous situation forces the system to reject, without having the possibility to direct the user where to go.

A dialogue cell system is provided with a strong mechanism to dynamically remove ambiguities. First observe that in order to be able to obtain a symbol, a sub-cell or basic cell must be activated first. The run time system can easily detect by a lock-out mechanism that the subcell is already activated. Disambiguating means that it will in this case activate a different instance of the same cell, which will be guaranteed to manifest itself to the user in a different way (e.g. on a different place on the screen, or with a different cursor and prompt or eventually at a different time (local

synchronisation by priority)).
This mechanism is closely coupled to the resource manager which assigns and deassigns hardware and firmware interaction resources.

This strategy is based upon a principle which is very fundamental. That is the strict separation of type and value. In a dialogue cell type is the realm of the symbol part whereas value is in the realm of the value part.
For instance, a syntax rule which has two alternatives:

A: all letters and digits
B: all punctuation marks and zero.

is ambiguous when a zero is encountered.
In a dialogue cell either A and B would use different (virtual) keyboards or the same keyboard at different times, or there is only one syntax rule (say C), and the value mechanism associates a value with C in all cases. So, the ambiguity is either removed before the activation or it is not a syntactic but a semantic ambiguity.
An additional benefit is that the system is at first instance always prepared to accept all user inputs possible. The inputs it cannot accept are not enabled.

5.2 Syntactic specification

The two types of parallelism have different syntax. The parallelism based on early activations is specified by the **prompt-component** by selecting an asynchronous mode (e.g. SAMPLE and EVENT). A cell which is asynchronously active can only be deactivated through the deactivation of the parent cell.

The parallelism based on syntactic branching (AND- and OR-branches) is specified as part of the **symbol-component.** The activation of synchronous (REQUEST) cells in a given parallel branche is completely independent from the progress in the corresponding other parallel branches.
Through activation of a cell also its prompt section is activated. As a result a whole series of subcells may become active as well. For each active cell the parser (re)opens the corresponding syntax rule in the parse table. A parse cursor is initialised on the first symbol(s) of the rule and the input will be scanned for such a symbol (cf. input pool section 5.3). Simultaneous active dialogue cells produce external echoes and values which will be further dealt with by the parent cell. When exactly this will take place will only depend on the status of the parent cell. The handover of value and echo result is controlled by the input pool.

5.3 The input pool

When the parallel production of symbols for input is served by only one processor one needs a scheduler to determine which early symbol will next be processed by the parser. We will now show how this arbitration mechanism can influence the behaviour of the system as perceived by the user. The situation we are confronted with is as follows:
Each time an active cell produces a symbol this symbol can either immediately be consumed by the requesting cell (synchronous mode) or it is placed in the cell's corresponding sample or event register. In case the requesting cell is not immediately scheduled, the symbol is placed in the request register.
The input pool is the collection of all request sample and event registers.
Only a subset of the symbols in the pool is candidate for reading by the parser. All others are, one could say, produced too early.
Each time an elementary parse action has been completed, (i.e. the parser moves to the next symbol of the symbol expression), the arbitration unit can be activated to

decide which symbol for which cell to parse next.
The decision procedure chosen, favours high level cells over low level cells, and also favours cells in left branches over cells in right branches, provided the higher level parse pointer has not proceeded beyond those branches. This is achieved by considering the syntax graph which is implied by the dialogue cell hierarchy and appending a preorder code to each dialogue cell, being the preorder code from the node in the corresponding graph. The root will get the highest code. Then the scheduler will search the pool for the symbol with the highest preorder code.

The effect of all this at the user interface is that the highest level result implied by any input is always produced first and also that cells trying to empty an event queue have priority over cells adding a new symbol to the queue.

6 Example

Suppose a user is working with a graphics terminal and wants to specify a text, a position and a rectangular box. These actions may be part of, say, adding legends to a blind map.

The program will arrange the text in such way that it fills the box and next will place the box at the position given.
We will compare three different dialogues for the same problem, both from the point of view of feedback quality and conciseness of dialogue. The second and third solution are given as transformations of the specification of the first solution. In this way also the dialogue cell specification method itself will be illustrated.
In each of the solutions a requirement is to be met saying that the user may choose the order of speçifying the values (text, pos., box) at will. At any point in the dialogue he must be able to look at the result of combining the current values of the triple. In addition he may save, discard or change the result.
The three solutions will offer an increasing amount of feedback and support to the user. Among other things this may lead to a simpler, more effective, dialogue.

6.1 First solutions: use a keyword menue

The user is given the possibility to choose any action by presenting him a keyword menue. After selecting a menue item he is prompted for the corresponding action and the result is shown. Then the menue reappears asking for the next action. This is a strictly synchronous approach. Only one activity at the time is possible. Nevertheless, this is a very commonly used way of providing such facilities. In many implementations of similar dialogues the menue is up on the screen continuously. However, sometimes it is usable an sometimes it is not. The improvement of the solution in the first example over this situation is that the menue will only be presented when a selection is expected from the user.
We will now briefly describe the dialogue cell program for this problem. The language syntax can be seen as an extension to the syntax of the programming language C.
The overall structure of the program consists of a list of typedefinitions, resource declarations and dialogue cell definitions. One of the dialogue cells is marked as the main cell. It will be called from the application within which this program is embedded.
The typedefinitions prepare for the use of variables specific to the dialogue program. The resource declarations define resource claims, such as a part of the screen or a particular cursor. They can then be used by the various dialogue cells. Resource declarations are used to partly define a resource strategy e.g., screen layout.

```
/*   This is a comment line

/*   Resources

/*   A window is defined by its lower left and upper right corner in virtual screen
     coordinates ([0  ,1]:[0,1]).

WINDOW

                w text      =    { 0.0, 0.0, 0.5, 0.2 }
                w menue     =    { 0.0, 0.2, 0.2, 1.0 }
                w box       =    { 0.5, 0.0, 1.0, 0.2 }
                w screen    =    { 0.2, 0.2, 1.0, 1.0 }
/*   Type definitions, including global initialization

TYPEDEF STRUCT
```

MENUE ptb menue = The 'ptb menue' defined here will be handled over as a parameter to a basic cell called 'menue'. This dialogue cell will allow the user to select a menue item. To this end it will present the menue in a window (also a parameter) on the screen. The activitation of the cell 'menue' will take place in the prompt-section of the calling cell e.g. through the rule: REQUEST menue (ptbmenue, wmenue) ...:
The user will initiate each action, either his own or by the system, by a menue choice, as follows:

'text' for typing a string
'box' for specifying a rectangular box around the text,
'pos' for specifying a position for the text in the window wscreen,
'do' for showing the result of combining text, box and pos,
'keep' for storing the result in external store (e.d. variables, files),
'next' for setting up for a new triple,
'stop' for exiting the dialogue.

Only the first three actions require further input. This input is collected by three subcells called 'text', 'box' and 'pos' respectively. 'do', 'keep' and 'next' will cause interval systems actions whcih also may create further pictorial output.
We will now give the main dialogue cell with explanatory comment lines:

```
/*
MAIN DICE legend:  STRUCT (STRING, LOC [2]) [ ]; SEG [ ]

/*   The result produced by legend is an array of structs describing text in a box,
/*   represented by: a) STRING, LOC [2] and b) a SEG
/*   The first is the internal representation, the second is the picture

PROMPT

/*   the prompt section of the dialogue cell body
```

```
STRUCT (STRING, Lo [2] tbox = NIL
/*    local variable, initialised to undefined
/*    activitation of subcells in REQUEST mode

TEXT tx = NIL

/*    local picture var. of type TEXT.

REQUEST       menue (w menue) (ptb-menue),
              text  (w-text),
              box   (w-box),
              pos   (w-screen);

ECHO OFF menue;

/*    each of the subcells gets a window on the screen
/*    the ECHO OFF switch will make the menue disappear
/*    after each selection

SYMBOL

/*    the symbol section
      CASE menue IN(

      'text'        :   text;
      'box'         :   box;
      'pos'         :   pos;
      'stop'        :    $
                    :   CONT;
      ) * CONT
/*    This expression in a nutshell, contains the whole control structure.
/*    i.e. an endless loop: menue * CONT, with an exit rule (
/*    internal to the loop
/*    the CASE construct which is embedded, allows for
/*    entering the sub-dialogue branches one at
/*.   a time

VALUE

/*    in the value part internal representations are obtained
/*    either locally or globally (through a SEND rule),
/*    from mapping subcell results-values.

      menue ('do')   → tbox = (text, (pos, pos + box));
      menue ('keep') → SEND (tbox);
      menue ('next') → tbox = NIL;

/*    'do' generates the internal representation of the triple
/*    'keep' stores it away.
/*    'next' cleans the variable, if not kept, then it is lost.
/*    this implements a user rejection after checking ('do') the
/*    visual result (see ECHO).
```

```
ECHO
        menue ('do')        →       tx = TEXT (fit(text,pos,box));
        menue ('keep')      →       CREATE SEG (rect(p,b),tx);
                                    REMOVE tx;
        menue ('next')      →       REMOVE tx;

/*  'do' generates a picture primitive using a utility routine
    to fit the three parameters, amd producing a TEXT primitive
/*  every picture primitive value assigned
/*  is also visible on the screen
/*  it removes the now redundant tx
/*  'next' removes (if still required) the current tx

END
```

The subcells called are **menue, text, pos** and **box.** They each have a separate window on the screen. This makes it possible, for instance. that **pos** and **box** use the same (hardware) cursor for specifying a position. There can be no confusion about the questions for whom (**pos** or **box**) a position is provided. The system may emphasise this by providing different cursor feedback for the two uses. This will become even more interesting in the next improved version.

6.2 Second solution: continuously active input mechanisms

The high price for user freedom to be paid in the previous solution is the elaborate way of indicating the next input type through a menue-keyword. In the next solution the system presents all relevant input options simultaneously. The user simply by using the right mechanism (e.g. keyboard for text, cursor in either menue window, pos window or box window) implicitely indicates what he is going to do.
Consequently, the keywords 'text', 'pos' and 'box' can be removed from the menue.
At any time he can still select 'do', 'keep', 'next' and 'stop' with the same effect as in the previous solution.
We will not repeat the complete program here but only give those lines which hav been changed,

```
        Menue dkn menue     =       { 4, ('do','keep','next','stop')
                                    };

        DICE legend ...

        PROMPT ...

REQUEST menue (w menue) (dkn menue);
SAMPLE text (wtext), box (wbox), pos (wscreen)

...
/*  the reduced menue is still used in a synchronous mode
/*  but now this mode will be used to initiate some action
/*  on previously provided other inputs
/*  text, box and pos are immediately (early) activated
/*  and their most recent input as provided by the user is kept in
/*  the input pool.
```

```
/*   While the input is in the pool, its echo remains
/*   on the screen, so that the user can see what he has done.

SYMBOL          CASE menue IN (
                'do'  : (text AND pos AND box);
                'stop': $
                      : CONT
                ) * CONT;

/*   A 'do' selection in the menue will immediately lead to
/*   simultaneously sampling of the current  text, pos and
/*   box values. The VALUE mechanism will next provide a new
/*   triple echo based on the sampled result.
.
.
.

END

/*   no more changes are needed to achieve this improvement.
/*   in fact the dialogue has been simplified, as is also apparent
/*   from the simplified SYMBOL part, and yet the systems gives
/*   better support.
```

The continuously active cells **text, pos** and **box** will only become deactivated when the calling cell (**legend**) is deactivated. This will occur after a 'stop' selection.

6.3 Third solution: real-time, integrated feedback

In the last solution the triple: **text, box** and **position**, is immediately updated each time one of the three current values changes. The result is dispalyed in the map. This means that instead of having a 'do' keyword in the menue, input of one of the triple values must trigger the do-action.
Depending on how, for instance, the sub-cell **pos** works, one can achieve either a reappearance of the triple each time a user acknowledges a new position in **pos** (i.e.: the triple 'jumps' on the map) or the triple will be 'dragged' over the map into the right position, provided that the position mechanism of **pos** can supply new positions in a 'continuous' mode.
Again we will only give the changes with respect to the first splution.

The entry 'next' in the menue can also be discarded, provided that either an empty string (**text**) or an zero sized **box** (lower left = upper right position) will effectively cause removal of the current triple from the map.

```
MENUE triple = { 2, ('keep', 'stop');
DICE legend ...
PROMPT
EVENT menue (w menue) w triple);
    EVENT text (w text), box (w box), pos (w screen);

/*   all input mechanisms are permanently presented on the screen.
```

```
/*   each, when satisfied by the user may cause some action.

SYMBOL
    (CASE menue IN (
    'stop' : $
           : CONT
       )
    OR (text OR pos OR box)
    ) * CONT;
/*   each of the four subcells are in a parallel OR-construct
/*   which is cyclically kept active until via the menue
/*   'stop' the dialogue is terminated.

VALUE

        (text OR posOR box)      →      tbox = (text, (pos, pos + box));
        menue ('keep')           →      SEND (tbox);
        ...

ECHO

        (text OR pos OR box)     →   tx = TEXT (fit (text,pos, box));
        .
        .
        .

END
```

In the Value-part and ECHO-part now the occurence of the sub-expression (text OR pas OR box) in SYMBOL will cause the 'do' effect.
The updating of the triple can be done in an efficient way if the corresponding utility can see which parameters have changed since the previous update. (e.g. if only pos is changed, the text need not be reshaped, but only translated.

The three variants of the solution for the same problem demonstrates the great variety in dialogue forms one can have and at the same time that the specifications method can accomodate this variety. As long as no pertinent rules for user interface programming can be given, this flexible approach seems the best way. The method allows for easy program transformations to obtain a different external behaviour of the system after trial sessions.

7 Conclusions.

In this paper the situation of a user at a powerful interactive graphic terminal has been analysed, leading to the formulation of two kinds of parallel input. One is the early activation of input mechanisms which allows a user to select any of them at any time, even to use them simultaneously. The other kind of parallellism results from the input syntax, allowing a syntax rule to have more than one simultaneously possible alternative.

An input parser for such a form of input has next been described. Some important properties of the parser are:

1. The possibility to have more than one parse in progress, using inputs from parallel input streams.
2. The possibility to dynamically add and remove syntax rules from the 'currently active grammar'.
3. The ability to remove ambiguities occuring in the current active grammar through (re)assignment of resources to input mechanisms.

Special attention has been paid to user freedom in providing his input. This has been modelled by allowing the user input syntax to be a set of variants over the strict input syntax as used by the parser. The buffer mechanism required for coupling user syntax variants to the input syntax, the input pool, has been introduced and some of its properties are described. This results in a tolerant user interface system which can adequately handle the common situation that production of inputs is done different from accepting it. The input mechanisms organization plays a vital role, they can be hierarchically organised, meaning that complex mechanisms are built from simple ones. Every mechanism is completely described in one unit, called dialogue cell. Every dialogue cell defines the input type and the actual value of that type produced.
By relating the input type to the mechanism used, the parser can maintain a status which only forces it to reject input on the basis of wrong value, never on the basis of wrong type.

The desirable property for a user interface which results from this is that every input the user is capable of producing will be, at first instance, acceptable to the system.

On existing single processor systems, the parallel parsing of input needs to be simulated. For this purpose a scheduler is needed to decide which of the possible input symbols is to be parsed next. The effect of the scheduling algorithms on the user interface has been described.

In future reports a number of further aspects of dialogue cells will be described. Among those are: the interface of the dialogue cell system to an interactive graphics system, the design and implementation of basic dialogue cell libraries, the dialogue value process which maps the input values in internal data structures, automatic error recovery in dialogue cells, picture maintanence, screen management for dialogue cells etc.