Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

R. van Liere, P. J.W. ten Hagen

Introduction to dialogue cells

Computer Science/Department of Interactive Systems Report CS-R8703 January

Introduction to Dialogue Cells.

R. van Liere
P. ten Hagen
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

General-purpose tools for specifying user interfaces are currently of wide interest. Of particular interest is the research done on the precise functionality of such tools. The tools presented here are based on the concept of DIALOGUE CELLS. The dialogue cell system provides a number of features seldom found in other systems, such as:

i. Various forms of user freedom.
ii. Abstraction mechanisms for input devices.
iii. Advanced application communication mechanisms.
iv. Low level workstation control.
v. High degree of portability.

This report introduces various concepts related to dialogue cells and dialogue programming environments.

1980 Math. Subject Classification: 69K34, 69K36
1983 CR Categories: I.3.4, I.3.6
Key Words & Phrases: User Interface and Management Systems, Man-Machine Interaction, Dialogue Design Tools

1. Introduction.

For the past few years research on user interfaces has mainly concentrated on two different areas. One area is concerned with *designing* the user interface itself. Topics related to this area of research have focussed on the questions of how to assure the quality of a particular user interface design. Especially ergonomical issues, such as for example the consistency of the user interface or how to use color, have been given much attention. A second area of research, and the one that we are concentrating on, is concerned with the problem of creating software *tools* to support the development of graphical user interfaces. Such tools are very much needed because the task of building good user interfaces is technically speaking far from easy. A few typical problems that such tools will have to deal with are:

- *control over the communication means.*
  Precise control over the real-time behaviour of the communication means is a condition for guaranteeing certain ergonomic requirements.
- *context switching.*
  Fast changing contexts, typical for interaction, require dynamic redistribution of sparse resources (ie. screen space, input devices, cpu support for various feedback loops, etc).
- *user freedom and system tolerance.*
  Maximizing user freedom and system tolerance requires capabilities akin to pattern recognition combined with real-time response for any successful recognition event.
- *device independency.*
  The application program, including its user interface, should be specified in a device independent way.

The tools that we propose here address the problems mentioned above. In addition, the tools are embedded in a programming environment which will provide the designer with a systematic design and development method for specifying user interfaces. These tools, all belonging to the DICE (*dia*logue *ce*lls) system, currently run on the SUN III and IBM 5080 workstations. Besides running under different operating systems (ie. UNIX 4.2 vs. UNIX V), these workstations have fundamentally

Report CS-R8703
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

different architectures (ie. raster vs. display list). Implementation issues will not be discussed in this paper (see [11]).

Of particular interest is the method used for specifying the interactive dialogue. The approach taken by us is to characterize the interactive dialogue by a syntactic description from which the user interface can automatically be generated. The syntax of the specification language , used for specifying the interactive dialogue, is such that the specification yields readable and easy maintainable code, from which the real-time behaviour can be fairly precisely understood. It has been shown that specifying user interfaces with this method can be done with greater ease and reliability than by following other "hand written" methods.

This report introduces various concepts that are included in the DICE specification method. CHAPTER two addresses various concepts and definitions related to how user interfaces are to be specified. CHAPTER three discusses a number of fundamental issues which form the basis of the specification method. CHAPTERS four and five discuss the dialogue cell system as a particular solution to the concepts addressed in chapter three. CHAPTER six gives a simplified example of a DICE program and also illustrates the points presented in this paper.

2. Specifying scenarios.

When writing an interactive program, a dialogue programmer is primarily concerned with specifying what are called *scenarios.* A scenario will define all relevant information for every, possibly complex, transaction between user and interactive program. This will include such things as the specification of the program, the resource administration, and how the user will communicate with the program. More specifically, a scenario will define both the external and internal effects the transaction has as well as all mutual relationships between these effects. In this context, external effects are defined by actions related to the visualization of the program, whereas internal effects are defined by actions which will not be visualized by the user. For instance, the user can be notified as soon as a certain calculation has successfully been completed. In this case, the scenario will precisely specify how the calculation is to be done (internal effect) and how the user is informed (external effect).

The *dialogue* constitutes all external perceptible effects of the executing scenario. By specifying a scenario, the dialogue programmer will define a *dialogue language* which is implicitly specified as a set of grammer rules. The dialogue language will statically define all possible dialogues; ie. a particular dialogue will simply be an element of the specified dialogue language. Analogous to the compiling of a program, a dialogue will be parsed in run-time by a parser. However in this case, tokens will be the inputs given by the user. The syntax of the dialogue language, implied by the grammer rules, will define the sequences of input tokens. Thus, each token is read by the parser which in turn tries to match the corresponding grammer rule. Furthermore, semantic actions can be associated to every grammer rule. These actions will be initiated by the parser as soon as the grammer rule has been matched.

As will be clarified later, it is possible to separate the dialogue part of a scenario from the associated application part. Consequently a part of the scenario, the so-called *dialogue specification*, is solely responsible for the dialogue. In order to write complex dialogue specifications, a special purpose specification language is mandatory. Syntactically, the special purpose language will contain constructs for mapping user inputs onto application data structures, and constructs for displaying output from the application. The scenario language contains, as part of the dialogue specification, all conversions between user data and application data. Such conversions encompass both data formats and control formats. The later is quite a novel feature. It allows for having a control structure most suitable for interaction next to an independent control structure which is more suitable for application programs. Ideally there should be only one, very high-level, language which is suitable as both dialogue specification and application language. The main reason for having two different languages is that the constructs of a dialogue program are significantly different from the constructs used in an application program. This will be elaborated when dealing with each particular dialogue concept. Furthermore, by having a special purpose specification language many properties of a scenario in the dialogue domain can be implicit in the semantics of the dialogue language, relieving the dialogue programmer from the need to specify these in the dialogue. It is very unlikely that a general purpose high-level language will have these properties. Although the various logical structures can be simulated in the same language as the application is written in, it will make the specification very difficult to write and read.

The two most fundamental properties of scenarios are:

- *they can be context sensitive.*
  A context sensitive interpretation can be given to the input / output by the application; ie. depending on the state of the application. The scenario will have to take into account the state of the application program when information is presented to the user.

  For example: a text editor operates in two modes; a "navigate" mode and an "edit" mode. For each mode the text editor will (re)act to user commands in a different way. The scenario can be such that, while in "navigate" mode, the editor responds by echoing commands on a different part of the screen as it would do when it is in "edit" mode. In both modes the same key strokes can be used.

- *they can be guaranteed.*
  Resources demanded by a scenario are guaranteed to be available when needed. It must be known beforehand if this is not the case so that appropriate action can be taken. As will be seen, this is of paramount importance for two derived guarantees: unambiguity and completeness of dialogues.

  The specification language contains a construct to allocate resources. Before the scenario is executed on a particular workstation, a check will be done to ensure that the requested resources are available. Resource deallocation is implicit in the semantics of the language.

FIGURE 1 summarizes the concepts being discussed. A scenario specification is written in a special purpose specification language. The scenario will then be translated in executable code which in turn will implicitly define a dialogue language. Furthermore, numerous properties are validated which will aid to the guarantee of correctness of the dialogue language. The executable code will, along with the dialogue run-time system and the application, constitute a set of allowable scenarios. During run-time a the dialogue part of the scenario will be parsed.

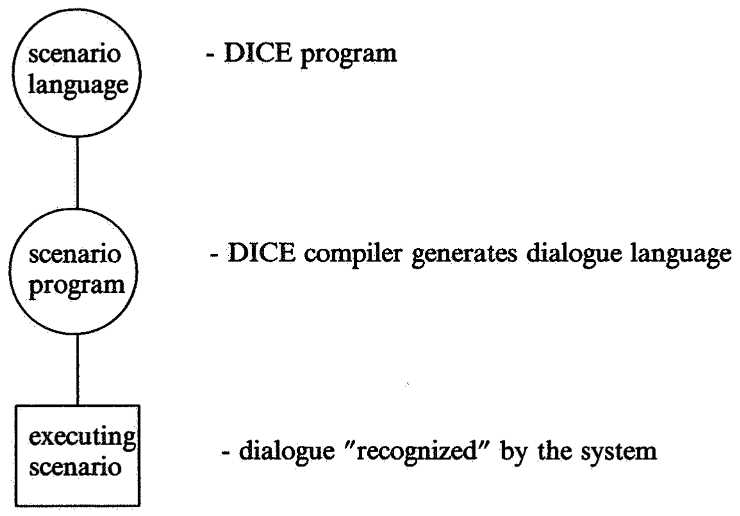
scenario language
- DICE program
scenario program
- DICE compiler generates dialogue language
executing scenario
- dialogue "recognized" by the system

FIGURE 1. - RELATIONSHIP BETWEEN SCENARIO AND DIALOGUE.

3. Fundamental concepts.

This section introduces a number of fundamental concepts of the specification language and the dialogue language. These are:

- separation of the dialogue from the application.
- parallelism.

- low level workstation control.
- result mapping / passing.

Although no level of detail or completeness is intended to be achieved, these concepts are chosen in order to provide a context to discuss issues more specific to the DICE system. Separation and low level workstation control apply to the specification language discussion whereas the concepts of parallelism and result passing are fundamental issues of the scenario language itself.

3.1. Separating the dialogue from the application.

Separation of the dialogue from the application means that there will exist a collection of program modules which are concerned exclusively with the user dialogue. These so-called dialogue modules look from the outside just like application modules. Hence, they can be easily combined with real application modules. How the various modules exchange data and control information is determined by the normal options available for each application module. From the inside, however, dialogue modules can look quite different. In our case they will consist of DICE programs. The DICE system must therefore be capable of interfacing with other programming environments according to the rules of such environments.

3.1.1. Benefits of separating dialogue from application.

Some benefits of separating the dialogue part from the application are enumerated in the following list:

- *Input correctness checking.*
  Numerous (code consuming) assertions on input data can be left up to the dialogue program. This implies that it is possible to assume, at least syntactically, correct input when writing the application program.

- *Error recovery.*
  Errors can be detected and repaired as part of the input process. Having error repair done as part of the input process allows the user to immediately correct errors. For instance, each part of a complex input can be validated (and eventually corrected) before the complex input as a whole is submitted to the application.

- *Simplifies modification.*
  The dialogue can be modified without having to change the application. Here, modification includes changing the dialogue specification as well as porting the dialogue onto different hardware.

- *Encourages modularity.*
  Separation will allow dialogue experts to specify the dialogue while application experts can concentrate on the application. Furthermore, various "plug-compatible" dialogue tools can be available in a given environment. These tools can (eventually after being tuned to own taste) be called from any part of the dialogue.

3.1.2. Requirements for separating dialogue from application.

Separating the dialogue part of the scenario from the application part is possible if the following two conditions can be ensured:

1. the application can be organized in such a way that the control structure of the application is "open". This means that the user must be able to influence the flow of control of the application.[1]

1. This is also referred to as *external control.*

2. the application may be involved in at most one interaction at any time. However, this one interaction can be arbitrarily complex.

FIGURE 2 illustrates how a dialogue can be viewed from an application. Suppose the application is made up of the set of subroutines

$$Subs = S_1,...,S_n.$$

The application program can be viewed as a tree of subroutine calls. The root of the tree is the "main" program, and dialogue is exclusively done by calling subroutines which are at the leaves of the tree. Suppose that the set of subroutines that take care of the dialogue is

$$Dias = D_1,...,D_m$$

Each D_i is responsible for receiving an arbitrary complex input token from the user and for giving the corresponding feedback. Furthermore, each D_i will eventually return an arbitrary return value to a application subroutine.

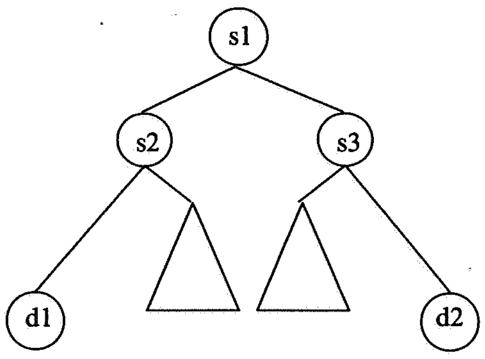

FIGURE 2. - SEPARATING THE DIALOGUE FROM THE APPLICATION.

Since each D_i can return complex application dependent tokens, some application knowledge must be included in each dialogue. This knowledge can be used to interpret (and eventually correct) input tokens. How much application knowledge should be put in a dialogue specification is an open question. To much knowledge will negatively influence the preformance of the feedback since very much checking will have to be done every time a token is entered. Common practice is to only include application knowledge which can be syntactically checked in the dialogue language. For instance, a drafting system for printed circuit boards can check the size and shapes of chips syntactically. Moreover, it can also determine the possible sequences of inputs that can be given. However, although it can be semantically verified, the drafting system should not know that a certain chip layout is illegal. Verifying a particular layout should be done by the application that receives the final result from the drafting system.

FIGURE 2 suggests that the dialogue part decomposes into a number of "dialogue subroutines". However, it is very well possible that also the dialogue part takes the form of a "dialogue subroutine tree" which contains "leaf calls" for exchanging information with the application. Whether both dialogue subroutine trees progress in parallel with a message passing protocol between them, or whether they cooperate as coroutines on request basis is up to the run time support system. The scenario specification is capable of specifying any of these situations.

3.2. Parallelism.

There are a number of reasons for introducing parallelism in a user interface:

1. Most graphics packages allow numerous basic input devices to be simultaneously active. Parallel active input devices allow users to make their own decision about which input to produce next; ie. which active device is to be used. Accordingly, on a higher syntactical level, it is possible that more than one picture can be under construction. For each picture it is immaterial that inputs

for other pictures were produced as well.

2. Several pictures presented on the screen can be correlated with each other. Assuming that different pictures are associated with different tasks, each task will then produce the corresponding feedback. The user, when changing one of the correlated pictures, may trigger various associated tasks. As a result changes on more than one place may occur simultaneously.

3. The user is given control over the order in which input tokens are to be given. Assuming that a certain input token sequence is required, the user is thus allowed to give any permutation of this sequence. This allows the user to switch between pictures without having to activate the corresponding devices before doing so.

The only way for the dialogue programmer to efficiently deal with the problems arising from the above mentioned topics, is by introducing a parallel model (see [4]). A parallel model will simplify and make possible the independent programming of simultaneous picture constructing tasks.

3.2.1. Kinds of parallelism.

There exist two ways to introduce parallelism in the scenario system. They both deal with the dialogue part. A further kind of parallelism, in the application part of the scenario, is not discussed in this paper. The first kind of parallelism is to give the user the freedom of determining how and when the dialogue can be started. To the extent that the user has this freedom, the system must provide all input mechanisms simultaneously. Associated with this parallelism is a recognition task of the run-time system to decide which of the available inputs (individual or together) can be reacted to. This kind of parallelism is called *early activation* because the input mechanisms are activated before the system needs the result: the system may sooner or later deal with it but is not necessarily waiting for it. A different kind of parallelism is introduced by *syntax.* In this case the grammer specifies a number of alternatives which may be under construction simultaneously. The system would be able to proceed as soon as one or, depending on the grammer rule, all of the components become available.

How the system deals with the first kind of parallelism depends on who has the initiative in the dialogue. There are three different ways that the system can receive a particular input. These are:

1. *user and system synchronize.*
   In this case the dialogue cannot proceed until the user gives the requested input token. This is similar to the request mode in GKS. Request mode is necessary when synchronization is required between the dialogue and the user.

2. *the initiative is at the system.*
   In this case the dialogue is running without waiting for a particular input token from the user. However, when the run-time system needs a token, it will sample the corresponding input mechanism. The latest input token, which may or may not be valid, is taken. This is similar to the sample mode in GKS.

3. *the initiative is at the user.*
   The dialogue keeps running until one or more particular inputs are needed. The dialogue will get the input at user defined time. This is similar to the event mode in GKS. However, in this case, each subdialogue has its own event queue. Another advantage of event mode is that the user may supply inputs even before the system is ready to accept these.

The syntactic parallelism differs from early activation by the fact that in addition to user freedom also some system freedom is introduced. The system may determine independently from the user which of the available alternative components it will process. The introduced parallelism allows the user to supply sequences of input tokens in an arbitrary order. This means that each device which can receive an input token must be active. If not, the user will be forced to activate every device prior to supplying an input token with that device. However, multiple active input devices can cause input ambiguities, ie. it is not known from which device an input token originated (see section [4.2]). The system contains a mechanism to automatically and adequately deal with such situations.

3.3. Workstation control.

The amount of control that the dialogue programmer can have on the workstation should be unlimited. This implies that it should be possible to use all (advanced) features offered by a particular workstation. However, the specification method is such that workstation control must be done in a device independent way.

In particular, there are two issues which must be addressed. These are related to, firstly, the optimizing of the update mechanism and, secondly, to the amount of control available feedback techniques.

3.3.1. Optimizing facilities.

On all workstations output picture generation during dialogues is incremental. This is done by managing a so-called display list which represents the objects being displayed. Incrementing the display list is not a burden on the response of the workstation. However, after every input / output transaction, an update of the screen is necessary; ie. the complete display list will have to be redrawn. This forms the bottleneck of all workstations and it is said that the speed of an update is proportional to the quality of the workstation. There are a number of facilities which can cut down on the amount of work to be done at update time. These facilities will organize the display list in such a way that it is known beforehand which part of the display list minimally is to be executed to visualize an input / output transaction. These facilities are very low level and none will be accessible to the dialogue programmer. However, on a higher level, these facilities can be translated into techniques which the dialogue programmer will be able to influence. These techniques are of cardinal importance if preformance guarantees are to be made about dialogues. These techniques will include:

- *Update control.*
  There is be a "performance" table attached to the workstation which indicates the speed of an update. In this way the dialogue programmer can program the workstation when to do an update; ie. the dialogue programmer can hold back time consuming updates and do these at a more appropriate time.

- *Positioning of viewports.*
  Since the dialogue programmer knows the position of the active window, there must be a mechanism which enables him to inform the workstation that the picture changes occur only on that part of the screen where the window resides. Ideally, this screen managing mechanism should be as low-level as possible so that hardware facilities might eventually be used.

- *Buffering graphical output.*
  The dialogue programmer is also offered a facility which allows the buffering of output information. This seems to be contrary to the strategy which supports the idea that what the user does can be directly observed. The buffering facility will allow the dialogue programmer to make the tradeoff between buffered and non buffered input / output. The following two categories are distinguished, leaving the decision of what belongs to which category to the dialogue programmer :
  - pure output: this is buffered and can be delayed.
  - feedback: this is non buffered.

  Furthermore the programmer must be able to specify synchronization points when the buffer is to be flushed.

3.3.2. Feedback techniques.

Interactive feedback techniques are defined to be (application dependent) ways of interacting with the application program. Basic feedback techniques can be categorized in: construction techniques (rubber boxing / banding, gridding, continuous selecting / dragging, etc.), picking techniques (various cursor shapes), illustrating techniques (dependency relations through animations) and command techniques (icons, menus, function keys, etc.). More complex feedback can be a combination of basic feedback techniques allowing the programmer to offer the user more informative feedback.

The following properties should be applicable to each feedback technique:

- *definable.*
  There should be a rich set of different feedback techniques (called a prompt / echo type library) at every workstation. These are selectable and form the basic set of feedback techniques which will be used by the dialogue programmer to define elementary feedback. Furthermore, there must be a mechanism to define specific feedback techniques in terms of more elementary ones.

- *honest.*
  This means that input feedback will be shown if and only if these inputs can be given. This implies that the feedback will appear only when the corresponding input device can be used (ie. the device is activated or the feedback (re)enters the echo area belonging to the device). The feedback will disappear when no use is made of the device any more (ie. the device is deactivated or the feedback exits the echo area belonging to the device).

- *triggered.*
  The feedback technique must give a reaction at the moment of actual input. This could vary from a simple acknowledgement to a complete highlighting of a picture.

Feedback should be done at workstation implying that the mechanism on the workstation which is responsible for giving feedback will have to be programmable. Only in this way can real-time response be provided.

3.4. Result mapping / passing.

This section introduces two different interaction models, the so-called orthodox model and the input / output pair model. After giving various (convincing) arguments in against the orthodox model, it will be shown in what way the input / output model can be used.

Figure 3 illustrates the orthodox interaction model, in which all input tokens come from one input stream and output tokens go out on one output stream. Furthermore, some form of synchronization on tokens is offered between the input / output streams. A crucial property of this model is that, even though a number of different devices may provide input tokens, they will be accepted sequentially by one input stream.

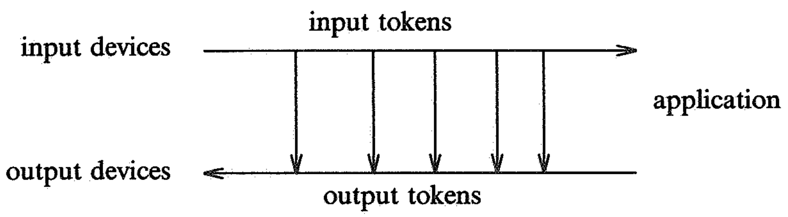

FIGURE 3. - ONE, SYNCHRONIZED, STREAM OF INPUT TOKENS.

The orthodox model is frequently used by associating conceptual / semantical / syntactical / lexical levels to the stream of tokens (see [1]). In this way the application can group a number of lexical tokens in a syntactical unit and thus reason about groups of tokens. This can also be done to a group of syntactical units to derive a semantical unit. The GKS approach is to introduce six different input streams, each corresponding to what is called a logical device stream. However, this approach is still based on the sequential nature of the input tokens.

When writing interactive programs the orthodox model has two serious deficiencies:

1. For interactive programs it is mandatory that input and output can be done at all times very efficiently. In the orthodox model this is not necessarily the case. The main reason being that it

is not known which output token belongs to a received input token (and vice versa). This leads to a sometimes complex administration in order to decide to which interaction task an input token belongs.

2. Another deficiency arises from the fact that, when examining a complete interaction, there is no unique separation between syntax and semantics. Syntax on one level of the interaction can be semantics on another (and vice versa). This is because every interaction has different levels of abstraction implying that tokens on one level can have meaning but on another can not. However, when looking at only one level, the separation between syntax and semantics can be made.

The problems mentioned above can be solved by considering every transaction to be represented by a so-called io-unit.[1] A io-unit consists of a (possibly empty) input token, a (possibly empty) output token and an identification (in our case given as an index). As will be seen, input resp. output tokens can be arbitrarily complex. The notation used for an io-unit is: (i_1,o_1). The dialogue can simply be viewed as to perform the sequence of tokens:

$$(i_1,o_1),\ (i_2,o_2),\ \ldots,\ (i_n,o_n)$$

It is important to note that a particular io-unit is directly identifyable given its identification. The io-unit model also includes a mechanism to define a hierarchy of io-units. The input resp. output token of a unit at a particular level in the hierarchy will become a local input resp. output value of the one higher level unit. At the lowest level of the hierarchy are units which receive input tokens from physical devices. The corresponding output is the low level feedback of that device. The hierarchical approach provides the dialogue programmer with a mechanism to define arbitrary complex interactions without giving up the input output coupling. For example:

$$(I,O) = (((i_1,o_1),\ (i_2,o_2)),\ (i_3,o_3))$$

Here input I is defined to be composed of two io-units $((i_1,o_1),(i_2,o_2))$ while output O is defined to be result from the io-unit (i_3,o_3), cf. figure 4. This implies that in the case of $I = ((i_1,o_1),\ (i_2,o_2))$, o_1 and o_2 are local outputs which will be of no concern on the level of I. I itself is a composition of i_1 and i_2. This example illustrates that the higher level unit $(I,\ O)$ can decide which functions of both input and output parts it receives will become part of its own input and output. Each level also can redefine the relation between the I and O part as well as add entirely new input and output functions.

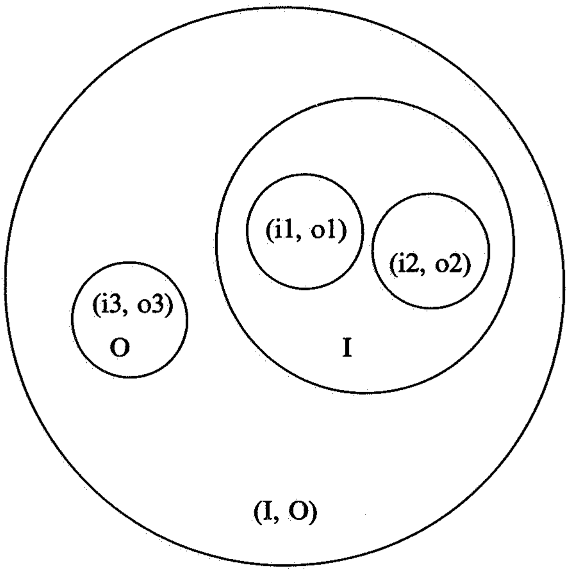

FIGURE 4. - HIERARCHY OF INPUT / OUTPUT UNITS.

1. Io-units were first introduced at the Seillac II workshop [1]. Originally the io-unit model was designed to define hierarchical input devices (ie. the root of the hierarchy can be viewed as is device which returns a token). It was labeled as one of the more "promising approaches".

Associated with each unit is a (possibly empty) action body. Action bodies are used to transform received input resp. output tokens onto (parts of) the resulting input resp. output token of the unit. By transforming tokens this way, different (application dependent) interpretations can be given to input resp. output tokens at different levels in the hierarchy. These semantic actions for transforming input tokens are called *value mappings* whereas transformations of output tokens are called *echo mappings.*

The io-unit model, as presented above, does not include control; ie. nothing is mentioned about when and in what order a result of a subunit will be received. Since this can be done in numerous ways, control issues will be discussed in the context of dialogue cells themselves (see section [4.3]).

3.4.1. Value and echo mappings.

A hierarchy of io-units can be viewed as the exchange of tokens between the user and the application program in execution. However, tokens required by the application are usually not structured in the same way as the user would provide them. Conversions of input tokens will take place before they are presented to the program. A number of, possibly application dependent, routines must be available to do these conversions. These routines must be supplied by the application programmer and can be used freely in every dialogue subroutine.[1] However, it should be noted that these calculations are done locally in a certain unit and have no side effects on other units.

The io-unit model is strictly hierarchical: relationships between subunits can only be validated in the common "enclosing" unit of the subunits. For example, suppose a unit (*unit C*) consists of two subunits (*unit A* and *unit B*) each having input tokens which are constrained by some (application dependent) measure. A relationship such as "the input of A must be twice as large as the input of B" can be validated only in unit *C*, because the value mapping of unit *A* is ignorant of the tokens of unit *B*. However, the relationship "my output token must be larger than four" can be validated in unit *A*.

3.4.2. Result passing.

If the input resp. output token produced by one unit is complex and structured (such as picture representations or large sets of application data), the unit may be decomposed into a number of subunits with each subunit contributing to the compound unit. Each subunit will contain all actions required to produce its token. A result passing mechanism is provided to pass tokens from subunits to their parent unit. The result passing mechanism is the only way tokens can be given to other units.

The following figure illustrates the result passing mechanism. Conceptually, there will be a buffer in which the units put their results (eg. i/o tokens). This buffer is needed because a number of units, each producing tokens, can be simultaneously active.[2] Tokens will reside in the buffer until they are needed by the "enclosing" unit. The result passing mechanism will extract results from the buffer and pass these to the corresponding unit.

1. It is in principle possible that a conversion routine can in turn start a dialogue causing recursiveness.
2. It is even possible to give input tokens before the corresponding unit is ready to accept this token (type, mouse ahead).

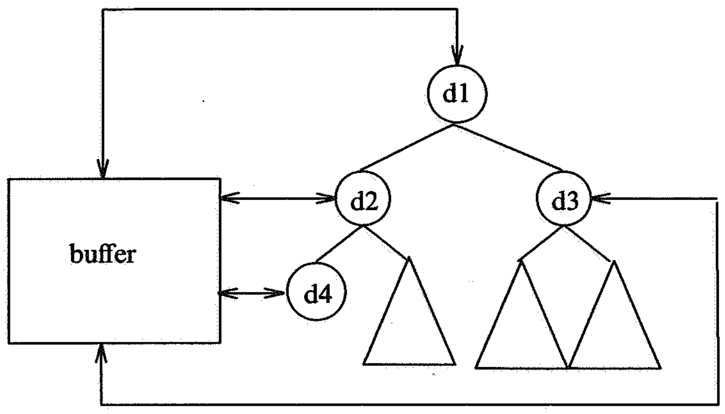

FIGURE 5. - PARALLEL RESULT PASSING.

3.4.3. Advantages of I/O units.

The io-unit approach has a number of advantages over the orthodox model:

1. Viewing each interaction as an input / output pair allows the dialogue programmer to associate input with output (and vice versa) at any level of detail. Correlating input with output in this way will make the interaction easier to understand.

2. By introducing hierarchy a separation between syntax and semantics can be made. Action bodies in one unit can associate meaning to received input tokens while the final result of the unit will merely be an input token for the enclosing unit (which in turn can associate a completely different meaning to its input). This approach will be useful for designing the dialogue, understanding what the dialogue does and finally programming the dialogue.

3. This model includes support for both sequential and parallel dialogues. Parallel hierarchical dialogues can be accomplished by activating subunits asynchronously. This can be done by activating subunits in different modes or by supplying a number of combining operators (see section [4.2]).

   Note that the input / output model can easily be implemented on a multi-processor system (ie. each unit will claim its own processor). Such a scheme will require one additional processor which acts as a manager for the activation and result passing between units. Currently everything is implemented in one process. The run-time system includes a dispatcher which schedules the tasks that need to be done by each unit.

4. Dialogue cells.

The dialogue cell system is based on the input / output unit model. Each io-unit is specified as a *dialogue cell.* A dialogue cell is specified in terms of *subcells* which will supply the "enclosing" cell with input resp. output tokens. These tokens are also referred to as *dialogue cell symbols.* Specifying subcells is done in a so-called *symbol expression* which consists of a sequence of subcells and combining operators. The overall result of two combined subcells is determined by the individual results of the subcells and the combining operator. The total external effect of a cell can be thought of as consisting of two parts. One part specifies the external effect during execution, the other specifies the final result. The external effect during execution, which is specified in the body of the cell, is temporal and is undone when the cell is deactivated. The final result, which is specified in the cell header, will remain after the cell has completed execution. Furthermore, every dialogue cell is activated in a particular environment which, after being inherited from the "enclosing" cell, can only be manipulated local to a cell itself. Each environment will contain the necessary information to execute the dialogue cell.

Evaluating a dialogue cell consists of the following seven steps:

- the initial parameters, including the environment, are elaborated.
- all subcells receive necessary information to activate.
- for each subcell[1]
  - -- the subcell is evaluated.
  - -- the corresponding value mapping action is preformed.
  - -- the corresponding echo action is preformed.
- upon termination of the cell, the result token is delivered.
- all local effects are removed.

Each step is either explicitly programmed by the dialogue programmer or it is implicitly done by the run-time system. For instance, value and echo actions are programmed, whereas all obsolete local effects are removed implicitly by the run-time system.

4.1. Dialogue cells and dialogue cell hierarchies.

The *dialogue cell header* specifies all information needed for properly invoking a dialogue cell; ie. its name, parameters and return type.

Each *dialogue cell body* consists of the following components which, as a whole, completely specify one step of the dialogue:

- *prompt component*: specifies the initialization of the cell.
- *symbol component*: specifies the symbol expression of the cell.
- *value component*: specifies the transformations of inputs from subcells.
- *echo component*: specifies the graphical output of the cell.

The semantics (see [5]) of the dialogue language define a number of implicit relationships between cell components. These relationships include the ability to specify synchronization, data exchange and value correspondence between components and aid the dialogue programmer to minimize the specification of external references.

The rest of this section will discuss each component and the implicit relationships in more detail without paying attention to the actual syntax of the specification language (see [7]).

4.1.1. The cell header.

The cell header specifies the name of the cell, its parameters and the type of the external result. The parameter part consists of parameters of two different classes:

- *environmental parameters*:
  Environmental parameters specify in which environment the cell is invoked. Environment parameters also specify how the cell can be used. What belongs in the environment is discussed in more detail in section [4.2].

- *non-environmental parameters*:
  Non-environmental parameters specify all other things concerning the value and echo data. Non-environmental parameters can be compared to normal parameters in conventional programming languages.

Associated with parameter passing are two default mechanisms which allow leaving parameters

1. More precise would be to say that this depends on the activation mode of the subcell and the combining operator of the symbol expression.

unspecified when a dialogue cell is called. The first default mechanism will allow each environment parameter to be inherited from the "enclosing" cell. This allows a dialogue cell to be activated in a context dependent way. The second default mechanism will allow the dialogue programmer to give parameters (static) default values when a cell is invoked. If such a parameter is not explicitly given in the actual parameter list, the default value will be assigned. These two default mechanisms make it possible to specify a dialogue cell with many parameters without the necessity for much overhead when only a few are needed for a particular activation. The rich and flexible parameter passing mechanism allows the dialogue programmer to design cells in a "dialogue independent" way. For instance, both workstation and application constraints can be parameterized outside the cell so that the same cell can be used in different dialogues.

The result type is also specified in the cell header. The type of the cell is the type of the value returned by the cell. By associating a type to each cell, a typing schema can be followed by the compiler to ensure that type errors can be eliminated at compile time. Also included in the dialogue specification language is a subtyping schema which allows the dialogue programmer to define subtypes over complex input configurations (see [6]). For instance, the type of the input configuration specified as "a and b and c" is the record consisting of the result types of cell a, b and c. Similarly, if the input configuration is specified as "a or b or c", the resulting type is the variant record consisting of the result types of cell a, b and c. Subtyping will allow the programmer to manipulate just one element of the resulting type in a flexible and natural way.

4.1.2. The prompt component.

The prompt component guides the user and prepares the system for input actions. Prompts may appear on the screen as cursors, messages, pictures or as menus. The prompt component will initialize the environment of the cell, (re)initialize subcells, and initialize the value and echo components.

If the cell is decomposed into subcells, the prompts are likewise refined. The initial prompt will allow the user to discover in which context the task is invoked. On the other hand, previous prompts in the hierarchy are no longer meaningful if the required task completed. When more than one prompt is associated to one resource, precedence rules cause enclosing prompts to be switched off temporarily.

4.1.3. The symbol component.

The symbol component specifies the symbol expression of the cell, which consists of a sequence of subcells and combining operators (the combining operators are: *and*, *or*, *case*, *sequential* and *iteration*). Analogous to expressions in programming languages, symbol expressions can explicitly be decomposed into subexpressions. This will give the dialogue programmer the possibility to define any input configuration. Since every cell has a corresponding type, a type can be associated to every symbol (sub)expression. Which type this will be depends on the combining operator of the subexpression.

The semantics of the symbol component is such that whenever a symbol is received a *trigger action*, corresponding to the name of the symbol, is executed in the value and echo parts of the cell. Triggering is also possible for subexpressions, allowing the dialogue programmer to immediately validate, map and acknowledge composite symbols.

4.1.4. The value component.

Input symbols are generally not of the same type required by the application or other dialogue cells. Usually, some kind of type conversion and computation is necessary to produce the result of the cell. These conversions and computations are done in the value component.

Every value trigger consists of a sequence of *value rules* each describing when, how and where the accepted symbol is mapped onto an internal value (see [6]). Mappings consist of a sequence of transformations, each of which can pre / post conditioned by predicates, from values in one domain onto another. Furthermore, associated with each value rule is a status variable. If any of the specified pre / post conditions fail, the status variable is set to false and the automatic error recovery mechanism is invoked. This will cause the corresponding trigger to be reactivated so that the user can immediately correct the error.

4.1.5. The echo component.

The echo component will inform the user how the dialogue cell interpreted the input symbols entered. Like prompts, echos may appear on the screen as cursors, messages, pictures or menus. Furthermore, echos can be converted to result values and can be passed over to the calling cell.

It should be noted that the echo component will include a parallel interface to the workstation. The parallel interface will have to administrate and update all graphical attributes needed by the currently active dialogue cells. This implies that when a number of cells output pictures simultaneously, the parallel interface must ensure that the corresponding attributes are properly set (see [3]).

4.1.6. Relationships between components.

All direct reactions of the dialogue system to a certain input are specified as part of the same input configuration. Only in this way may the programmer expect to be able to judge the effect of each input / output transaction. This is especially true for relationships between input values and feedback (ie. correlations). Numerous implicit correlations between input and output are kept intact by the dialogue system without the dialogue programmer having to specify these. The following is a list of these implicit relationships:

- *implicit triggering.*
  As seen above, a trigger action is automatically executed each time an input configuration is accepted. To achieve this, the corresponding subexpression and action must be specified in the value / echo part. This will have the effect that the action belonging to the trigger will be executed when the input configuration is accepted.

- *implicit picture result passing.*
  Whenever a subcell passes a picture as a result to its "enclosing" cell, the dialogue system must automatically insert the result in the window of the "enclosing" cell. All environment dependent transformations and attributes of the picture must eventually be updated. Passing picture results is done automatically and is of no concern to the dialogue programmer.

- *implicit garbage collection.*
  Whenever a cell is deactivated all lingering echos belonging to that cell will be removed. Lingering echos include such things as pictures cursors, messages, or menus. Since this cleaning up process can be done automatically, it is of no concern to the dialogue programmer to administrate and update these echos. Hence, it is up to the dialogue run-time system to keep track of everything on the screen and to remove superfluous output.

- *data exchange.*
  The general result passing facility is completely transparent to the dialogue programmer. A symbol name in the body of a cell is implicitly a reference to the result of the cell. How results are passed and where the result data is kept is hidden from the dialogue programmer.

  During run-time, result administration is done by the buffer manager (the so-called *input pool manager*). When and where the results will be extracted is determined by the input pool scheduler, which will schedule dialogue cells in increasing priority.

- *(re)activation.*
  As mentioned earlier, there are three different activation modes: request, sample and event. The later two , so-called *early activations*, allow the user to provide input tokens even before the run-time system is ready to accept them. It is the systems responsibility to implicitly do all activation administration. This includes, for example, the input queue administration and resource

(de)activation.

For example, given the following screen layout:

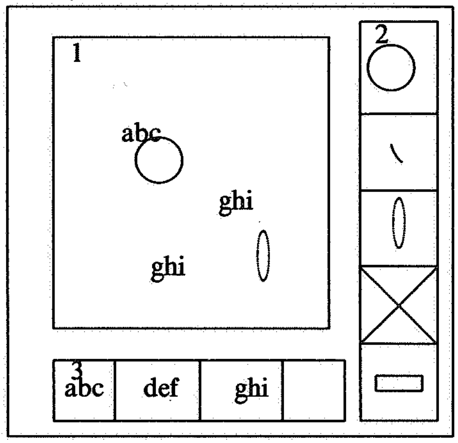

FIGURE 6. - SCREEN WITH THREE CORRELATED WINDOWS.

Suppose an object is being created in window 1. Windows 2 and 3 are used as temporary windows in which arbitrary actions can take place (these actions are related to the creation of the object in window 1). When the object in window 1 is finally created, all aspects correlated with the creation of the object will be automatically removed (in this case windows 2 and 3). The results produced in window 2 and 3 are placed automatically in window 1.

4.2. Dialogue cell environments.

Every dialogue cell is obliged to operate in an particular environment. The environment will contain information necessary to execute the cell, including such things as: information for the parser, scheduler, resource manager, and the error recovery mechanism, predefined system dependent values, user defined types, primitive functions which can be used for local calculations, application dependent functions which can be called, communication functions (messages) for exchanging information with the application.

A complete dialogue cell environment can be very complex providing many facilities. For every component in the environment it must be defined how every individual dialogue cell will interact with it. This is done explicitly or implicitly depending on the component and on the context in which the cell is to be activated.

The environment consists of a static and a dynamic part. The static environment is given in the specification of the cell and will be verifyed during compile time, while the dynamic environment contains the actual values given to the static components. The following is an list of some of the facilities which are included in an static environment:

- *Resources*:
  Every dialogue cell must declare which resources it needs. Resource declarations are done in the prompt section by specifying the amount as well as structure of the resources. This amount will be automatically claimed at activation time of the cell and returned upon exiting the cell. Resource structure is specified in terms of virtual resources. During run time virtual resources are mapped onto physical resources. The compiler will statically ensure that these physical resources are available when needed. It may be obvious that when two subdialogues use the same physical resources simultaneously an ambiguity exists. It cannot be decided to which dialogue cell the tokens produced via these resources belong. Ambiguities are solved by assuring that all resources are uniquely assigned to requesting cells. This is done by the resource manager[1]

1. See [8] for a detailed discussion on how ambiguities are solved by the resource manager. Ambiguities are resolved by hav-

which is embedded in the run-time system.

By defining resources in such a way, the dialogue programmer has no concern on how to administrate and update the resources. It should be noted that if a cell does not declare a resource, the resource of the "enclosing" cell will (implicitly) be inherited. In this way echo actions can be specified relatively to another cell.

- *Error handling*:
  A flexible error recovery mechanism is essential in every dialogue system. Every time an error is detected the run-time system automatically invokes an error handling mechanism (see [10]). The following properties hold for every error situation:
  - the user can never invoke an unexpected operation.
  - only "value" errors are possible, "type" errors are impossible.
  - error recovery will demand the reallocation of the deallocated resources (ie. resources will have to be restored in order to correct the error).

  Each cell can supply attributes which will aid the error handling mechanism in the recovery process. Default is to restart the input configuration which supplied the (faulty) value. However, it is possible to program the error handling mechanism to do (application dependent) recovery actions. Furthermore, the user will have some control during run-time over the error recovering process. For instance, the user can place the corrected token in front of the queue of already processed tokens.

  Error handling is also used to allow the user to skip back in the dialogue, correct / change something and then continue *without* having to repeat that part of the dialogue which was skipped.

- *Predefined data types*:
  The dialogue system provides a rich set of elementary data types which can be used freely in cell specifications. These include data types for manipulating the underlying workstation. The environment will contain a list data types visible to the cell. This allows the dialogue system to do the necessary type checking and memory allocation.

  Apart from the elementary ones, additional data types can be made available in the dialogue cell environment. For instance, collections of icons and text fonts, or (application dependent) picture building facilities built on top of the radical system. Once these are installed, they can be used freely as system data types. This mechanism allows the dialogue system to be enhanced in an application dependent way.

- *Calculation functions*:
  The environment will contain libraries of functions to enhance the value mapping mechanism. Such functions can be the basis for simplifying (application dependent) dialogues as well as implementing application oriented feedback mechanisms. Analogous to predefined data types, the environment will contain a list of calculation functions which can be called in the value part of the cell. Building such libraries may require expert knowledge about the dialogue cell system and language. However, the use of such libraries is simple and straightforward.

- *Communication functions*:
  The normal mechanism to exchange result values with applications through root cells will be extended by allowing the exchange of intermediate results with every active dialogue cells. A library of message functions can be made available for such purposes. It is important to realize that the dialogue cell system will constrain such a message mechanism to the extent necessary for guaranteeing understandable and manageable dialogues. For instance, messages must be cancelable as part of error recovery procedures.

The dynamic environment of the dialogue cell will contain the actual state of every component at execution time. For instance, which resources have been assigned to the dialogue cell incarnation, what message has been sent, what calculating functions can be executed, or, what the error status of the dialogue cell is.

ing the dialogue programmer specify a workstation dependent resource model which is verified at compile time. The resource model will serve as a framework for the resource manager to distribute resource requests.

5. Dialogue Cell programming environment.

The dialogue cell programming environment consists of a number of tools and other support facilities which will aid the programmer to implement dialogues. These tools can be seen as an enhancement to the DICE specification method. The facilities belonging to the minimal support system include:

- the compiler, which will translate the DICE specification into executable code. The compiler will also do a number of workstation dependent assertions on the specifications.
- the multi-stream parser and scheduler, which are responsible for controlling the flow of the dialogue.
- the resource manager which will dynamically manage all resource requests. As pointed out before, resource ambiguity is solved by the resource manager.
- the error recovery mechanism.
- the radical system which assures the parallel access to the workstation.
- a small set of basic cells. Basic cells will be discussed in the next section.

5.1. Basic cell libraries.

The hierarchical approach allows a uniform treatment of basic cells as well as dialogue cells. The body of a basic dialogue cell is a set of instructions of the underlying support system. The basic cell header specifies how this body can be invoked, how it can be parameterized and what token type it will return. What kind of user actions are involved in executing the basic cell is part of the operations manual of the implementation (see [2]).

At the very lowest level, basic cells will replace the six GKS input classes. These cells will produce their prompt (selectable from a set defined by the standard), have as their symbol part the operation of the logical input device that also produces an echo, and have as their value result the primitive GKS input value. This approach will assure flexibility and device independence. It is only these six basic cells which are workstation dependent. These are the cells which will have to be rewritten in order to port dialogues onto other workstations. TABLE 1 gives a list of the six most elementary basic cells in the DICE system. It must be noted, however, that the choice of these six cells will influence the level of control the programmer has over the physical hardware device. For instance, by only having *string* as a basic cell the programmer cannot control a particular character device.

basic-cell	return-type
locator	Wc
choice	Choice
pick	Pickid
valuator	Value
string	String
stroke	Wc [1..n]

TABLE 1. THE SIX MOST ELEMENTARY BASIC CELLS.

Using the lowest-level layer, multiple higher-level (even application dependent) libraries of cells can be constructed.

5.2. Dialogue tools.

5.2.1. Trace and history facilities.

The *tracer* tool will allow the user to replay any part of the dialogue. A traced dialogue can be stopped at an arbitrary point and be continued from that point. Furthermore, the tracer supplies information on the flow of the dialogue such as which cells were executed, which tokens were not consumed, etc. This information can be used for profiling a number of dialogue different sessions.

5.2.2. Debugger.

The *interactive graphical debugger* (see [9]) will aid the dialogue programmer in correcting the dialogue. All normal debug operations can be used such as the setting of breakpoints, inspecting data, etc. The debugger is a so-called interactive graphical debugger because it presents the cell hierarchy on the screen allowing the dialogue programmer to examine the (static as well as dynamic) call graphs. Debug operations are invoked by pointing to the call graph. For instance, a breakpoint is set by picking a cell.

5.2.3. Picture editor.

Every program which will allow interactive picture manipulation should include a *picture editor*. The picture editor, which in fact is simply a dialogue cell, includes operations such as search for a picture element, insert / change / delete a picture element, etc. The basic picture editor is available in the programming environment and can easily be tuned to do application dependent operations. For instance, a chip designer is only interested in editors which can edit transistors and registers. The dialogue cells which know something about the transistors and registers can be built on top of the basic picture editor.

5.2.4. Graphical data base manager.

A *graphical data base manager* which keeps control of all graphical data base objects is also provided in the dialogue cell programming environment. The data base manager will aid the dialogue programmer in inserting and extracting graphical objects (ie cell results) from the background store. Due to the hierarchical nature of graphical objects, the relations between data base objects tend to be quite complex. However, these complexities should be transparent to the dialogue programmer.

6. An example.

This example illustrates how a dialogue of a continuously changing picture can be specified. The dialogue allows the user to simultaneously drag two objects in two disjunct viewports. Dragging is done with the locator device by pointing to a new position in the viewport. When satisfied with the position of the object, the user will use the choice device to stop that part of the dialogue. The dialogue specification is:

```
    /* keywords and types are given in bold    */

dice locator : Wc
dice choice  (String str) : Choice

dice my_dragger  (WClass w, IClass c) : Picture
prompt
    initprompt                          /* declarations           */
        Trigger n_locator;
        Wc cur_locator;
        Resource r = <w, c>
    initsymbol
        choice ("stop"): event,       /* choice event mode      */
        locator          : sample;    /* locator sample mode    */
    initvalue                         /* initialise values      */
        n_locator  := continue;
    initecho                            /* initialise echo        */
        give_feedback (null);
symbol
    *(choice or locator) * n_locator
value
    locator: cur_locator := locator;          /* result of locator    */
    choice : n_locator   := stop;           /* controls symbol expr */
echo
```

```
22.     locator : give_feedback (cur_locator);  /* redraw feedback       */
23. end


24. dice my_dialogue : null
25. prompt
26.     initprompt                         /* declarations          */
27.         WClass wc = {<0.0, 0.0, 0.5, 0.5>, <0.5, 0.5, 1.0, 1.0>};
28.         IClass icl = {cursl};
29.     initsymbol
30.         my_dragger (wc, icl): event,      /* event mode    */
31. symbol
32.     my_dragger ; my_dragger
33. end
```

Both the dialogue cells *locator* and *choice* are basic cells. Only the headers are specified here (lines 1, 2). These cells return a value of a point in world coordinates resp. a choice value.

The dialogue cell *my_dragger* is parameterized with a window class and a cursor class (line 3). These will be used by the resource manager to allocate the declared resource (line 8). Note that these are formal parameters; *my_dragger* will be called with actual parameters (in this case with actual viewport and cursor classes (line 30)). The effect of the symbol expression (line 17) is that the cell *locator* will continuously be polled until the trigger variable *n_trigger* is set to **stop**. Polling is done because *locator* is activated in sample mode, (line 11) leaving the initiative to the system. The value of *n_trigger* changes when the cell *choice* is triggered; this is specified in the trigger value part of the cell *choice* (line 20). Since the cell *choice* is started in event mode, the user can determine the moment that this triggering is done. Each time the echo trigger part of the cell *locator* is executed (line 22), the picture is redrawn in a new position. For simplicity reasons, function *give_feedback ()* will do the redrawing. Note that application dependent feedback can be given by parameterizing *my_dragger* with application dependent parameters. The *prompt* section is the initialization part of the cell which includes the section that initializes the subcells. In this particular case the prompt section specifies how the cells *locator* and *choice* are to be activated (line 10, 11). Furthermore, state values are declared and initialized (lines 6, 7, 8, 13) and feedback is initialized (line 15).

The cell *my_dialogue* activates the cell *my_dragger* two times in a sequence (line 32). However, since *my_dragger* is activated in event mode (line 30), two different incarnations will be made. This will mean that a resource will be allocated for each incarnation of *my_dragger*. The resource manager will assure that the resources are unique, implying that both incarnations are activated in different viewports (line 27, 28). The class definitions are specified in virtual coordinates which will be mapped in run-time onto device coordinates.

FIGURE 7 illustrates how *my_dialogue* can appear on the screen at a certain moment of the dialogue. The object being drawn by *give_feedback ()* is a circle with two tangent lines. The cursor, in this case the cross-hair, is in the upper right viewport. Note that both incarnations of the cell *my_dragger* have a menu of one item which appears on the lower right side of the viewport in which *my_dragger* was activated and that clipping is done on the object in the lower left viewport.

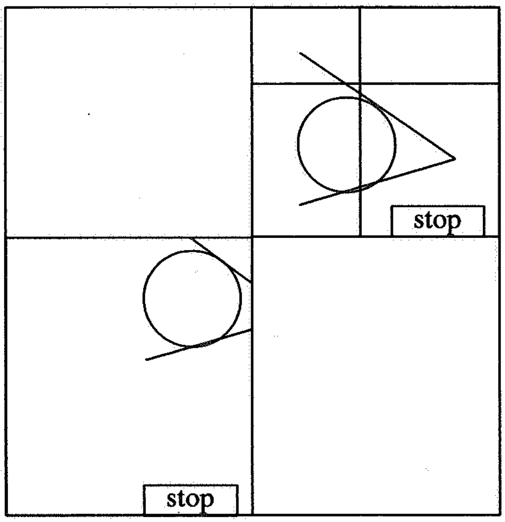

FIGURE 7. - DIALOGUE "MY_DIALOGUE".

Finally, by using *my_dialogue* as example, a number of issues discussed in this report are recapitulated:

- The scenario can be split in a part that specifies the control flow of the dialogue (done in the symbol expressions and symbol initializations) and a part that does data conversions (done in the value resp. echo parts).

- The top-down approach in designing the dialogue. The cells *locator*, *choice* and *my_dragger* can be designed and changed without influencing the resources of *my_dialogue*. Given the two different default parameter mechanisms, the cells can be programmed in a somewhat dialogue independent way so that they may also be used in other dialogues.

- Objects can be dragged in parallel. This is because, even though *my_dialogue* activates *my_dragger* in sequence, both cells are activated in event mode causing early activations. Each object will be bound to the viewport which belongs to the resource of the incarnation of the cell. By moving the cursor of the *locator* device in one viewport, the system can tell which object is being referred to.
  The second kind of parallelism can be seen in the symbol expression of cell *my_dragger*. The **or** operator will cause both cells *choice* and *locator* to be activated in parallel.

- Implicit triggering is done each time cell *locator* is sampled. The value trigger and corresponding semantic action is specified in line 19. The echo trigger and corresponding semantic action is specified in line 21. Furthermore, implicit triggering is also done after each termination of cell *choice*.

- When viewed as an io-unit, the dialogue can be written as:
  my_dialogue = (((*my_dragger*), (*my_dragger*)), (–, –))
  my_dragger = (((*locator*), (*choice*)), (–,*give_feedback*))
  locator = (*input_from_loc_device*, *feedback_from_loc_device*)
  choice = (*input_from_cho_device*, *feedback_from_cho_device*)

- Resource management is done in virtual coordinates. This will allow the dialogue programmer to specify cells relative to the position of the parent cells. Resources will be mapped onto the viewport which contains cell *my_dialogue* (ie. if cell *my_dialogue* is activated two times, the window class will be mapped onto two different viewports). Note that cell *locator* implicitly inherits its resource from the cell *my_dragger*.

- Garbage left on the screen when the corresponding cell is deactivated is automatically removed (in this case the menu of each choice device). The placed object of cell *my_dragger* is returned as a picture to the cell *my_dialogue*. Note that if *my_dragger* would not return a result, then the picture would be automatically removed after *my_dragger* terminates.

REFERENCES

1. R. GUEDJ, EDITOR, *IFIP workshop on methodology of interaction*, North-Holland, Amsterdam, 1980.
2. H. J. SCHOUTEN, *Basic cells on the IBM 5080*, CWI IS memorandum, Amsterdam, 1986.
3. H. J. SCHOUTEN, *Radicals - a paralell interface to GKS*, CWI report (to appear), Amsterdam, 1986.
4. P. J. W. TEN HAGEN and J. DERKSEN, *Parallel input and feedback in dialogue cells*, CWI CS-R8413, Amsterdam, 1985.
5. P. J. W. TEN HAGEN, *The semantics of dialogue cells*, CWI IS memorandum, Amsterdam, 1985.
6. P. J. W. TEN HAGEN and W. ESHUIS, *Value mapping in DICE*, CWI IS memorandum, Amsterdam, 1986.
7. R. VAN LIERE, *The DICE language manual*, CWI IS memorandum, Amsterdam, 1986.
8. R. VAN LIERE, *Resource managing in DICE*, CWI report (to appear), Amsterdam, 1986.
9. R. VAN LIERE and H. J. SCHOUTEN, *Debugging in DICE*, CWI IS memorandum, Amsterdam, 1986.
10. R. VAN LIERE and P. J. W. TEN HAGEN, *Error handling in DICE*, CWI IS memorandum, Amsterdam, 1987.
11. R. VAN LIERE, H. J. SCHOUTEN and P. J. W. TEN HAGEN, *Implementation strategies of DICE*, CWI report (to appear), Amsterdam, 1987.