Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

H.J. Schouten, P.J.W. ten Hagen

Dialogue cell resource model and basic dialogue cells

Computer Science/Department of Interactive Systems   Report CS-R8841   October

Bibliotheek
Centrum voor Wiskunde en Informatica
Amsterdam

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.).

Copyright © Stichting Mathematisch Centrum, Amsterdam

Dialogue Cell Resource Model and Basic Dialogue Cells

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

A model for the handling of resources used in interactive programs, with emphasis on the input device part, is developed. Its connection to the I/O pair model and its usage in the dialogue cell system is described. The definition and properties of basic dialogue cells are given. A method for constructing a basic cell library is presented. A number of results from the application of the resource model on a basic cell library is shown.

Computing Reviews Categories: 1.3.5.
Mathematical Subject Classification: 69K36.
Key Words & Phrases: Resource Management, Interaction Techniques, Computer Graphics, User Interface Management Systems.
Note: This report will be submitted for publication elsewhere.

1. I/O unit and resource models

1.1. Introduction

The I/O unit model was first put forward in [9]. It was recognised to be a promising approach for modelling interaction. It is elaborated further in [3]. This model is the basis of the Dialogue Cell system [1,12].

In an actual system like Dialogue Cells, one has to cope with the limited resources screen space and input devices. This is done by employing the resource model presented in [4]. The input part of this model will be elaborated further in the third section of this chapter. The model is also the basis of resolving ambiguities in the interaction.

The subsequent chapters are devoted to how the models are applied in the Dialogue Cell system. This system is itself based on the I/O unit model. It is shown how the resource model is used for the declaration of resources. It allows the programmer to specify the kind of resource a dialogue cell needs, with the actual value being assigned in run time. This makes it possible to build a library of interaction techniques (basic cells) without fixing their exact resources in advance.

Report CS-R8841
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1.2. I/O unit model

An I/O unit is a triple

$$P = <S, I, O>$$

in which S is a symbolic name, I is an input value and O is a picture output. An input value I and a picture O are *correlated* if they are in the same I/O unit.

I/O units can be compound, i.e. made up of a number of "sub I/O units" by means of functions f_i and f_o:

$$(I, O) = (f_i\ (I_1, .., I_n), f_o\ (O_1, .., O_n)) = F((I_1, .., I_n), (O_1, .., O_n))$$

Correlations between I and O parts are maintained within compound I/O units.

The behaviour of a graphical interactive program can be described in terms of manipulating I/O units. The three components of the I/O unit then have the following semantics:

- the input part I is an abstract internal representation of a (possibly complex) user input.
- the output part O is the representation of a picture on a workstation screen. This includes both program output and feedback generated by the input device(s) used to enter the value.
- the symbolic name S, denoting the correlation between the input and the output component of the I/O unit.

Both the *i*-part as well as the -part may be empty. In many cases the meaning of the correlation is that the output part is a visualisation of the input part.

Examples of I/O-units are:

- $P = <locator, <x, y>, dr_crosshair(<x, y>)>$.
  which defines a position, $<x, y>$, and a crosshair drawn on the screen, with the cross at $<x, y>$. The name of the unit is *locator*. The function dr_crosshair() denotes the visualisation of the crosshair.
- P = *menu*, exit, *dr_menu*(*box*, *strings*, exit)>.
  A menu selection has taken place from the choice device called *menu*. The item selected is exit. The menu is visualised by the function *dr_menu()*.

I/O units are manipulated by *active transaction processes* (ATP's). An ATP is a process that handles a single, but possibly compound, transaction between a user and the computer. Each ATP produces a particular I/O unit. When the ATP is ready, the I/O unit is returned to another ATP (the consumer of the I/O unit). The consumer can apply any transformation on the I/O units by means of the functions f_i and f_o and use that to produce its own I/O unit, so that compound I/O units may be produced. An ATP can construct the I/O unit it is to produce from zero or more I/O units that it consumes itself.

An ATP can be identified by:

(1) the name of the produced I/O unit,
(2) the name of the ATP to which the I/O unit will be returned,
(3) a unique incarnation identifier, distinguishing ATP's which produce I/O units with the same name for the same consumer.

Active transaction processes are activated by a parent ATP, so that they are organised in a hierarchy. If an ATP is deactivated all its children are deactivated first. It can deactivate itself or be deactivated by its parent. An ATP can be deactivated after it has produced zero, one or more I/O units. The consumer of an I/O unit is the parent of its producer.

1.3. Resource model

1.3.1. Informal introduction

There are two types of physical resources that are typically scarce for an interactive program: screen space and input devices. Among the tasks of the program is their management. If the program consists of a number of parallelly working, independent parts, resource conflicts may arise that need to be resolved.

We introduce a model for resources such that we can exploit the hierarchical structure of the processes involved in an interactive program built in accordance with the I/O-unit model. A resource of an ATP is defined in such a way that all its descendents have resources that do not extend beyond it. This way a certain part of the processtree will not conflict with other, unrelated parts. If two ATP's dispute a resource it can be resolved by their common ancestor. By localising the resource requirements, ATP's can be defined much more independently. At the same time no resource may be wasted. In using the model it is not necessary to claim resources that will not be used after all, just to be sure that no conflicts occur. As each ATP needs both screen space (to show its *O*-part) and input devices (to obtain its *I*-part), both are part of a single resource.

1.3.2. Definitions

A **resource** consists of three components:

- A **window**, a rectangular part of the workstation screen, aligned with the axes.
- An **InputSet**, a subset of the set of input devices of the workstation.
- A **priority**, a real number.

Every ATP has associated with it a resource. They use the resource for showing their outputs and getting their inputs. They have no effect on other parts of the screen and other input devices. What exactly we mean by an input device is explained shortly hereafter.

Several ATP's can share a single resource, because a child ATP can inherit its parent resource. However, resources that are not inherited are unique. Two incarnations of the same ATP are not allowed to be active in parallel as child of the same consumer. Even if they would have different resources, it would not be possible for the user to determine beforehand which incarnation would have produced the result. The identification of the resource acts as the unique incarnation identifier mentioned above. Uniqueness of resources means that for two ATP's D_1 and D_2 following condition holds:

$$Window_{D_1} \cap Window_{D_2} = \emptyset \wedge InputSet_{D_1} \cap InputSet_{D_2} = \emptyset \wedge Prio_{D_1} \neq Prio_{D_2}$$

Two windows have a non-empty intersection if they overlap on the workstation screen. The priority determines which one is on top. Two InputSets intersect if they have an input device in common (see section 1.4.3).

When an ATP comes into existence, its associated resource is said to become *virtually active*, or if the ATP inherited its parents resource, it already was virtually active. So, any resource is virtually active only once. Reversely, for every virtually active resource there exists an associated ATP. A resource becoming virtually active has the effect that its window will appear on the workstation screen and that any of the input devices in the InputSet of the resource may be activated from that moment on.

1.3.3. Window organisation

A **WindowClass** is a set of non-intersecting rectangles in the virtual space [0,1] × [0,1]. The *instantiation* of a WindowClass is the mapping of a WindowClass onto a window. The result is a **WindowClassInstance**, a set of tiled rectangles within the window. There is one predefined window defining the workstation screen. Other windows are elements of some WindowClassInstance.

A WindowClass can have multiple instantiations. It serves as a template which is mapped to concrete windows by an instantiation.

1.4. The input device model

1.4.1. Trigger and measure processes

Every workstation defines a set of available **trigger processes** and **measure processes**, connected one-to-one to physical input devices. (So, on every workstation there is a fixed set of these processes.) A trigger process notifies its clients when the (physical) trigger it is connected to fires. Firing can be caused by pushing a mouse button, but also by moving the mouse, the cursor leaving or entering a region etc. Clients of trigger processes are processes that have notified it to be interested in its events. (We shall see that that are in fact certain ATP's.)

A measure process maintains the value register associated with an input device. The user can change the value by operating the associated *measure handle*, e.g. the mouse. When the value changes, the measure process may show an echo of the current value on the workstation screen.

The available measure processes determine what kind of input values of the workstation can be accessed, *not* the physical input devices themselves. For example, if only a *string* measure process is present on a workstation, only string values can be obtained. If a *character* measure process is present, individual characters can be obtained.

Trigger processes and measure processes might not be mutually independent. For example, a move of the mouse changes the current measure, but might also cause a trigger process to fire. In other words, a physical trigger may be both a trigger for a trigger process, and a measure handle for a measure process. However, trigger and measure processes are required to be independent among themselves. For example, if a character measure process is present, no string measure process can also be present. In this way, measure and trigger processes can be seen as atomic entities.

There is no implicit connection between trigger processes and measure processes in our model. For example, although the mouse buttons are physically located on the mouse, it is not required that the mouse measure process always connects to one of the mouse button trigger processes.

For purposes that will become clear later on, we introduce on every workstation a **NIL trigger process** T_0, that is not connected to any physical input device and never fires.

Every workstation WS thus defines a set $IDEV_{WS}$ with:

$$IDEV_{WS} = \{T_0, T_1, \cdots, T_{n_{WS}}\} \times \{M_1, \cdots, M_{m_{WS}}\}$$
$$\forall i, j \leq n_{WS}: T_i \textit{ independent of } T_j$$
$$\forall i, j \leq m_{WS}: M_i \textit{ independent of } M_j$$

1.4.2. Comparison with GKS

Although the notion of measure and trigger processes is known from GKS [5] and related standards, there is an important difference between the GKS notion and ours. In GKS, trigger and measure processes are linked to logical input devices, and are only in existence when the corresponding logical input device is active. Since there are only a fixed number of logical input devices on any workstation, also the maximum number of measure and trigger processes is fixed. In our definition trigger and measure processes are connected to physical input devices and do always exist while the

workstation is open. There is no constraint on the number of clients of trigger and measure processes. These clients are roughly comparable to GKS logical input devices.

Furthermore, in GKS the correspondence between measure and trigger processes (in our sense) with a logical input device is determined by the implementation. In our model there is no predefined connection.

1.4.3. Input device organisation

An **InputDev** of a workstation is defined as a trigger - measure process pair:

$$InputDev = (T, M)$$

An **InputSet** is defined as a set of InputDevs:

$$InputSet = \{ InputDev_1, ..., InputDev_n \}$$

An **InputClass** is defined as a set of InputSets:

$$InputClass = \{ InputSet_1, \cdots, InputSet_m \}$$

An InputDev is **active** if there is a client (ATP) connected to its measure and trigger process.

The trigger component of an InputDev may be T_0. Such an input device will never produce a result by itself. Its measure value can be sampled however. The equality of two InputDevs is determined by the following rule:

$$(T, M) = (T', M') \Leftrightarrow (T = T' \neq T_0) \vee ((T = T_0 \vee T' = T_0) \wedge M = M')$$

In other words, two input devices are considered equal when either they share a non-empty trigger, or when they share the empty trigger and also their measure processes are equal. By the uniqueness of resources, this means that whenever a user fires some trigger within a certain window, that only one event will be generated, i.e. only one client will be notified. If the user operates a measure handle, this will cause a different result of subsequent samples of the measure process for one client only.

The measure value that is returned to a client can be compound. For example, when $T = mousebutton$ and $M = mouse$, then both the identification of the picture (in GKS a segment name and a pick identifier) at the mouse position and the mouse position itself might be returned. In GKS one would have to activate both a PICK device and a LOCATOR device that use the same physical trigger to achieve this result. However, whether these (logical) devices do share a trigger cannot be determined. Furthermore, the application is responsible for matching the two incoming events.

Note furthermore that, other than for WindowClasses, it is allowed for InputSets within an InputClass to have a non-empty intersection.

The *instantiation* of an InputClass IC relative to InputSet IS_0 is the mapping to another InputClass, defined as the largest subset of IC such that all its InputSets are subsets of the reference InputSet:

$$instantiation_{IS_0}(IC) = \{ IS \mid IS \in IC \wedge IS \subset IS_0 \}$$

So the effect is to remove those InputSets from IC that are not contained within the reference InputSet IS_0. The result of such a mapping is called an **InputClassInstance**.

There is one default InputSet, being the total set of InputDevs on the workstation $IDEV_{WS}$. Every

other InputSet is an element of some InputClassInstance.

1.5. Operations in the resource model

There are four operations in the resource model: allocation and deallocation of resources and activation and deactivation of input devices. Reallocation of resources is not considered in this paper, see [4].

1.5.1. Handout of resources

Allocation of a resource is done on request of an ATP by a special process, the **Resource Manager**. All interactive resources are managed by the resource manager; it has a total overview of the resources in use. The resource manager has to comply with requests, deciding on the basis of its knowledge of:

- the total available screen space of the workstation
- the available set of input devices of the workstation
- the history of the requests done so far.
- the resource of the parent of the requesting ATP

When an ATP is specified, it is in general not known in which context it will be activated. By virtue of the resource model still it can make a meaningful resource request. It issues a request by means of giving the Resource Manager three parameters being:

- a WindowClass *WC*, or a window identifier w_0
- an InputClass *IC*, or an InputSet identifier i_0
- a priority p.

If a w_0 is in the parameters, this is the ATP's parent's window. Similarly, if a i_0 is in the parameters, this is its parent's InputSet.

The resource manager can comply with the request by selecting a (W, I, P) triple, with

$$W \in instantiation_{W_{parent}}(WC) \vee W = w_0$$
$$I \in instantiation_{I_{parent}}(WC) \vee I = i_0$$
$$P = p$$

such that the uniqueness requirement is met.
If the request cannot be satisfied there are not enough resources available on the workstation.

The selected resource is furthermore made virtually active by the Resource Manager, which means that if a new window was selected, it will be shown on the screen. The (W, I, P) triple, the resource descriptor, is returned to the ATP.

An algorithm to handout resources to requesting ATP's (ignoring priorities for simplicity) is given below.

R_{parent} is the resource of the parent of the requesting ATP. *In_use* is a set of InputDevs associated with a resource. It represents the union of all InputSets of its descendents, that have been given a resource with the same window component.

```
foreach w ∈ WCInst ∨ w = w0
        foreach i ∈ ICInst ∨ i = i0
                if w = window(R_parent) then
                        if in_use(R_parent) ∩ i = ∅ then
                                in_use(R_parent) := in_use(R_parent) ∪ i;
                                return (w, i);
```

```
                fi
        else
                in_use(R_requestor) := ∅;
                return(w, i);
        fi
    end;
end;
return (no resource available);
```

The effect of this algorithm is that when a new window is allocated, any InputSet in the InputClassInstance is allowed, and when the parent window is inherited, an ATP can get an InputSet that contains InputDevs that were already allocated to one of its ancestors, but not to any of its brothers, nephews, etc.

Clearly, selecting the first InputSet in the given InputClassInstance that satisfies the condition is not always the optimal solution. Selecting the first one might cause a subsequent request to fail, while if a following InputSet had been selected, there would still have been a (w, i, p) triple available. For an optimal handout, therefore, either handouts that have already been done must be changed, or the algorithm must have knowledge of the requests still to come. The first is not an option, since suddenly changing a resource that the user is already using results in a bad user interface.

We will see that where applying this model to the dialogue cell system, it is possible to find an optimal handout on the basis of the second option, and furthermore, whether the ATP's can run on the workstation at all.

Deallocation of a resource must be done when an ATP is deactivated. If the resource contained a not inherited window, this window is removed from the screen, else if the InputSet was not inherited, the InputSet of the resource being deallocated is taken out of $in_use(R_{parent})$, so that these input device may subsequently be used by other ATP's within the parent window..

1.5.2. Activation of input devices

Active transaction processes can activate one of the InputDevs in their InputSet in their window, i.e. they connect to the measure and trigger process of the InputDev, supplying them with the required parameters. This may only be done by ATP's that have no children. Such an ATP should have a singleton InputSet. As an InputSet reflects the set of input devices that an ATP needs to perform its task (i.e. produce an I/O unit), there is no need to have more than one InputDev in the InputSet if only one is used.

When an ATP that has activated an InputDev is deactivated, it deactivates its InputDev.

2. Application of the models in the Dialogue Cell system

2.1. Dialogue Cells

The dialogue cell system is a User Interface Management System; a system for specifying the user interface part of an interactive program in a more consistent and easier way than by "programming by hand". It includes a specification language and a runtime system to realise the specification. The dialogue cell system follows the I/O unit model. Each ATP is specified as a dialogue cell in the dialogue cell language.

A dialogue cell contains a description of the control structure of its direct descendents (subcells), called the *symbol expression*, a grammar rule containing the names of subcells and a number of operators, together determining the order in which subcell results are handled. The symbol expression allows for subcells to be activated in parallel. For example, the rule *A and B* activates both *A* and *B*

at the same time. The user is free to enter input for A and B in any order.

Dialogue cells can be activated in on of three modes: *request*, *event* and *sample*. Cells in request mode are activated when their parent is ready to accept their result, cells in event mode are activated as soon as their parent are activated. So are cells in sample mode, but they must always have a value available; it will be read by the parent as soon as it is ready for it, without waiting for the subcell to be ready as in event mode.

The results produced by dialogue cells have a *value* (internal) and an *echo* (external) part, together forming an I/O unit. They are handled by the consumer together, so that the correlation between them is maintained. For example, the *value* could be a point and the *echo* a marker at that position. Pictures are handled by an underlying, GKS-like graphics system [2].

Each dialogue cell has its own resource, that it gets either by inheritance from its parent, or by declaration. The declarations are handled by the resource manager, which is a component of the run time system. For a more thorough introduction to Dialogue Cells see [1] or [12].

Since a dialogue cell program is declared as a static tree, and resource declarations within it are constant, it is possible to find the complete resource requirements of the dialogue in compile time. An optimal handout can be found, and furthermore, whether the dialogue can be run on the workstation at all.

An algorithm for this is the recursive application of the algorithm given above, backtracking to the next higher level when a resource request fails. If the root cells resource request fails, the dialogue cannot be run. The algorithm depends on the form of the symbol expression and the activation mode of the subcells.

2.2. Resource declarations

The declaration of a resource by a dialogue cell can take one of the three following forms:

$r = \langle WC_0, IC_0, prio \rangle;$
$r = \langle -, IC_0, prio \rangle;$
$r = \langle WC_0, -, prio \rangle;$

WC_0 is a previously declared Window Class, and IC_0 is a previously declared Input Class. *Prio* is a real number. If the window class parameter is omitted, the resource inherits the window of the parent of the cell. Similar, if the input class parameter is omitted, the parent InputSet is inherited. There is no need for a declaration for which both parameters are omitted, since that is default.

The declaration of a WindowClass looks like:

$WClass\ WC = \langle Window, Window, \cdots \rangle;$

A Window declaration looks like:

$\{ llx, lly, urx, ury \}$

with:
llx = lower left x coordinate,
lly = lower left y coordinate,
urx = upper right x coordinate,
ury = upper right y coordinate,
and $llx < urx \wedge lly < ury$.

It is not allowed that Windows within a WindowClass declarations overlap. Furthermore, all coordinates must lie between 0.0 and 1.0.

The declaration of an Input Class has one of two forms:

$IC = <InputSet,\ InputSet,\ \cdots >;$
or
$IC = InputClass + InputClass + \cdots ;$

The second form is called a *compound InputClass*. An InputSet declaration looks like:

$\{(T_1, M_1), (T_2, M_2), \cdots \}$

All T_i and M_j are trigger resp. measure processes of the workstation at hand, that can be referred to symbolically.

The composition of two InputClasses is defined as:

$IC_0 = IC_1 + IC_2 \vee$
$IC_0 = \{IS \mid \exists IS_1 \in IC_1\ \exists IS_2 \in IC_2 : IS = IS_1 \cup IS_2 \}$

As can easily be shown, the composition operator is associative.

Take as an example the InputClass $IC_1 = \{\{(M_1, M)\}, \{(M_2, M)\}\}$ and $IC_2 = \{\{(M_3, M)\}, \{(M_4, M)\}\}$, for two dialogue cells that use a mouse with either button 1 or button 2 resp. button 3 or button 4. The composition $IC_0 = IC_1 + IC_2$ is
$IC_0 = \{\{(M_1, M),(M_3, M)\}, \{(M_1, M), (M_4, M)\}, \{(M_2, M), (M_3, M)\}, \{(M_2, M), (M_4, M)\}\}$, for a composite dialogue cell that can use any of these combinations of two input devices.

The input component of a dialogue cell resource is an InputSet. This InputSet defines a set of InputDevs, that the dialogue cell needs to complete its task. By declaring multiple InputSets in a single InputClass, the programmer of the dialogue cell gives the resource manager the freedom to choose any one of them at run time. Because it is not known beforehand which one will be chosen, each on its own must be sufficient for the completion of the dialogue cell.
The option to construct InputSets from existing InputClasses allows the programmer to symbolically address the InputClasses of other dialogue cells, in particular that of library basic cells (see chapter 3).
Any dialogue cell that does not declare an InputClass in a resource declaration, inherits its parents InputSet, i.e. any of them (even if there is more than one child) can use that set of InputDevs.

2.3. Ambiguity prevention

When two or more dialogue cells are active in parallel, there is the risk of ambiguities. That is, one user input might cause more then one event. This is no problem if this is what the programmer meant to happen, but this can better be modelled by having the measure process return a compound value consisting of the two values that are intended to be returned in the simultaneous events.
However if two unrelated dialogue cells were active in the same window, using the same trigger for their input, there would be no way for the user to give an input to one dialogue cell and not to the other. This situation is prevented by:
(1) the definition of equality of input devices (section 1.4.3),

(2) the uniqueness requirement of resources (section 1.3.1) as enforced by the resource handout algorithm (section 1.5.1),
(3) the InputSet declaration requirement for dialogue cells that activate an input device (section 3.3).

This can be shown by assuming the opposite: two dialogue cells D_1 and D_2 having activated the same input device I in the same window. Then they have a common ancestor D_0 with $window(D_0) = window(D_1) = window(D_2)$, and no new windows have been declared on the paths from D_0 to D_1 and from D_0 to D_2.
I is within the InputSet of D_1. By recursion this means that I is in the InputSet of the first dialogue cell D'_1 along the path from D_0 to D_1 that declared a resource. By (3) such a dialogue cell must exist. Similarly, $I \in InputSet(D'_2)$.
However, after D'_1 was given its resource, $I \in in_use(R_{D_0})$ so by (2), D'_2 could never have been given an InputSet containing I.

3. Basic dialogue cells

3.1. Interaction techniques

An **interaction technique** is a method of the user entering a value or command to a computer and accompanying feedback from the computer. This may involve sophisticated graphical input and output devices. An interaction technique is not only the input of a single, atomic value and its accompanying feedback, but can extend to a coherent sequence of user actions and computer reactions. However, a sequence that is too long, or too complicated is not considered an interaction technique (otherwise the whole execution of an interactive program would be an interaction technique). Where to draw the borderline is unclear. As a consequence, it is hard to give a formal definition of the concept of interaction technique. In the context of the dialogue cell system we can define an interaction technique to be a **basic dialogue cell.**
A basic set of interaction techniques can, for example, be found in [8].

3.2. Definition and properties of basic cells

When writing a dialogue cell program, the programmer effectively defines a language (i.e. the language that computer program and user use to communicate). The terminal symbols of this language are the dialogue cells that have no subcells. These are called **basic dialogue cells** or *basic cells* for short. They take their input directly from the user. As any other dialogue cell, the input type and the feedback is described completely in these cells. They rely on the graphics system underlying dialogue cells to realise these descriptions, and to actually obtain the input value.
Basic cells offer a standardised interface to their users, namely that of the dialogue cell language. On the other hand, the implementor can use all input and feedback features that are available on a particular workstation. The physical resources used to obtain a value are an integral part of a basic cell. The reason for this is that, although the actual devices used to create a value may be of no importance to the application program that needs that value, it can make a big difference to the user. Therefore the dialogue programmer needs control over this. Furthermore, the physical resources associated with basic cells are used by the dialogue cell system to ensure that no ambiguous situation can occur (see [4] and section 1.5.2).

The approach of having the resources as an integral part of basic cells has the disadvantage that basic cells may be device dependent. Some basic cells will therefore not run on certain workstations. The mechanism of the dialogue cell language to construct cells from combination of others, offers a

solution here: such a basic cell can be simulated by using others that are available. By keeping the interface to the cell the same, higher level dialogue cells can use this cell the same way as if it was directly present on the workstation. A difference that may propagate to the dialogue program, is the difference in resources the simulated cell uses. For example, if a basic cell uses a keyboard to simulate a position input that is done by a mouse on a different workstation, this can have consequences for the rest of the dialogue as the keyboard is now tight up to this cell.

The mechanism of simulation can hide the workstation dependency of basic cells from the programmer (apart from resources). For the user there may still be differences, for example response time, available feedback types, etc.

Dialogue cells simulating basic cells in this way are also considered to be basic cells, although they actually have subcells. But it is immaterial whether the cell has been implemented on the workstation directly or within the dialogue cell system.

We can take this argument one step further. Since the capabilities of workstations grow continuously, it is conceivable to have dialogue cells that are not directly implementable on any existing workstation, but may be in the future. This can only be the case for dialogue cells that do not rely on a particular application program to perform their work. We therefore consider any dialogue cell, that does not depend on a application, to be a basic cell. To distinguish them from other basic cells, we will adopt the term *atomic cell* for basic cells that are implemented directly on a workstation.

3.3. Atomic cells

An **atomic cell** is a dialogue cell that has no symbol expression. It gets its input by connecting to a measure process and connecting that to a trigger process if it is activated in request or event mode. The trigger process notifies the measure process that a result may be sent to its client atomic cells. Whether this is done, depends on the current measure; the atomic cells tell the measure processes what are legal values for it at start up. For example, a locator value must lie within a certain window. The trigger process can be parameterised too; for example, a mouse button trigger process might be instructed only to fire for a certain atomic cell if the button goes up. An atomic cell activated in sample mode continuously samples the measure until it is deactivated. Its trigger processes is T_0, irrespective of the trigger process component of the selected InputSet of its declared InputClass. However, these components may not be omitted, since the atomic cell does not know itself whether it will be activated in sample mode; that is controlled by its parent.

The atomic cell is responsible for showing an echo on the screen. This is done by delegating the task to the measure process, giving it the necessary information at start up time. The current measure value, possibly mapped, is produced as result of the atomic cell.

When an atomic cell samples its measure process, the measure value at that moment might not be correct. In that case, it returns an *error status*, allowing the parent of the atomic cell to react as it sees fit. An atomic cell in request or event mode can return an error status if the associated *break* action for the input device is triggered. The break action consists of a combination of the normal trigger for the atomic cell and a special global break key, e.g. the shift or escape key.

3.4. Example

A workstation has a mouse with three buttons, with trigger processes T_1, T_2, T_3, and a keyboard with trigger process T_4, that fires when any key is hit. There are measure processes M_1 for the mouse and M_2 for the key values.

$$IDEV_{WS} = \{(T_i, M_j)\}_{i=1..4,\, j=1,2}$$

The following atomic cells are constructed:

L_1 with InputSet $\{(T_1, M_1)\}$, which returns a point.
L_2 with InputSet $\{(T_2, M_1)\}$, which returns a point.
L_3 with InputSet $\{(T_3, M_1)\}$, which returns a point.
C_1 with InputSet $\{(T_4, M_2)\}$, which returns a character.

A string basic cell can be specified for this workstation as follows (in an informal notation):

DICE str: String, Void
PROMPT
  initprompt
    InputClass IC $= <\{(T_4, M_2)\}>$;
    Resource R $= <-, IC, 0.0>$;
  initsymbol
    C_1: **event**
SYMBOL
  * (C_1)
VALUE
  C_1: if (C_1 == NEWLINE) stop iteration;
    else add C_1 to string;
  return: string;
ECHO
  C_1: if (C_1 != NEWLINE) **echo** C_1;
END

Note that this basic cell does not declare a window; it uses the window inherited from its parent. The InputClass consists of only one InputSet of only one InputDev, being the one *str*'s subcell will use. So this is not strictly necessary, but now the system can at the moment of activation of the basic cell already check the resource requirements (see section 4.3).

4. Libraries of basic cells

4.1. Use of basic cell libraries

A library of basic cells is in principle workstation dependent. The reason is primarily that workstations can have widely different input and feedback facilities. However, by the mechanism of simulation, basic cells that are used on one workstation can often be made available on others too. This means that if resource difference is no problem, a dialogue cell program using such a basic cell can be ported to another workstation without modification, and without much change to the resulting user interface.

The possibility for a dialogue programmer to take a basic interaction task from a library, instead of having to reprogram it himself over and over, is not only a matter of ease and time saving. Also, when the same basic cells are used consequently, the resulting user interface will be more consistent. The users will know how to perform the basic tasks beforehand from other programs.

4.2. Classification

The set of atomic cells that is available on a workstation is limited by the number of input facilities it features, but the number of basic cells that can be built on top of them is unlimited. For the construction of basic cell libraries it is necessary to have a method to decide what kind of basic cells should be included in such a library. As this library is intended for general use, we cannot look at an application to find out what is needed. However, this also means that the number of types of objects that basic cells in a library are dealing with is limited; no application specific objects are to be created.

This property implies that the number of possible type combinations of the *(value, echo)* return type pair is limited. Therefore a table of basic cell classes can be constructed. Figure 1 shows an extract.

Value return type	Echo return type
Void	Void
Point	Marker
String	Text
Int	Primitive attribute value
Real	Thermometer
Array of Points	Polyline
Transformation Matrix	Radical transformation

FIGURE 1

Not every (*value*, *echo*) type combination is meaningful. In accordance with the concept of I/O units, a correlation between the value and echo result must exist. Consequently only meaningful combinations are included.

Another classification of interaction techniques can be found in [8]. This classification is based on the semantic action performed by the interaction technique: selection, positioning etc. The visible effect is regarded as a part of a particular interaction task, not as a undetachable part of the result.

To offer a complete library of basic cells, the designer of the library must provide a basic cell for every class in the table above. Furthermore, a basic cell must be present for every input device available on the workstation, that is useful for the type of *(value, echo)*.

4.3. Design considerations of basic cells and libraries

The type of the internal and external value (i.e. the *I* and the *O* part of its I/O unit) a basic cell returns, and the type of input resource used to obtain it are fixed for every basic cell. Every other aspect of the cell can, in principle, be parameterised: the prompt and echo, initial values, etc. In general it is up to the designer what is fixed in the basic cell and what is parameterisable. A number of considerations must be made:

- To be of general use, the basic cell must be able to show different dynamic behaviour in different contexts. Only the global use of the cell must be fixed. As an obvious example, a basic cell that returns a choice on a menu must have as a fixed feedback type the menu, but the number of entries, the strings and the indication of the current value may be parameterisable.
- To allow for different types of users, a basic cell may offer the possibility to input a value in one of several ways (using the *OR* operator in the symbol expression). Any user can then select the mechanism he prefers. This mechanism increases user freedom. However, this feature uses up more resources. On a workstation with limited resources, this kind of basic cells may be not appropriate.

- Every basic cell in a library must declare its own resource. This has two advantages. First, the user of the basic cell can immediately see what InputSet is allowed for the parent of the basic cell so that it can use it as a subcell. Second, looking at the basic cell library from the outside, there is no difference between basic and atomic cells: both kinds are fully characterised by the interface to its parent (parameters and return types) and the required resources.

If the basic cell is to operate in the window of its parent, then only a list of possible InputSets (its InputClass) must be specified, otherwise also a window has to be declared. In both cases there cannot be a problem if a descendent declares a window, because that will be instantiated relative to the basic cell's window or a subwindow thereof.

Parameters are given to dialogue cells in a fashion similar to procedures of programming languages. Parameters are passed to each incarnation of a dialogue cell only at activation time. The parameters are used to steer the behaviour of the dialogue cell; for example the exact feedback produced can be steered by using the parameters in the *echo* part of the dialogue cell, where the visual effect is described. If the application is to control the behaviour of a basic cell, it must set the value of an actual parameter in the parent of the basic cell before activation. Note that, in that case, by definition the parent is not a basic cell.

4.4. Results of the models for basic cell libraries

A basic cell library is extendible with basic cells that return any type of value, obtained by any kind of physical input device, showing arbitrary echo's. Still, the access of these facilities can take place within the same type of interface, namely the dialogue cell language. On the other hand, workstation capabilities in terms of input devices and locally supported prompt and echo facilities, span a wide range, but are fixed (and thus not extendible) for an individual workstation.

The resource model provides the mechanism to connect the basic cell interface to physical input devices. By allowing the use of arbitrary combinations of trigger and measure processes, any available input facility can be addressed through the basic cell interface. This mechanism is analogous to the one proposed by Duce in [11] as an improvement of the GKS standard.

By means of the resource model and the resource declaration method a programmer can abstract from the resources needed by library basic cells. For windows this can be done because the resources declared by basic cells are relative to its parent, so their exact position is depending on that of its activator, so that a basic cell can be activated by several dialogue cells without conflicts. For InputSets the abstraction is done by addressing them symbolically with the mechanism of the compound InputClass, where the component InputClasses are InputClasses of library basic cells.

If a dialogue cell's InputSet comes from a compound InputClass, with one of the components being a library basic cell that is a subcell of the dialogue cell, then the library basic cell always will have enough InputDevs available (i.e. a non-empty InputClassInstance). This can be seen as follows.

Say IS_p is the InputSet of the dialogue cell. By definition there is a $IS_{bc} \in IC_{bc}$ such that $IS_{bc} \subset IS_p$. By definition of instantiation this means that $IS_{bc} \in Instantiation_{IS_p}(IS_{bc})$.

An advantage of allowing InputSets in an InputClass to intersect is that the programmer does not need to know their exact contents. Dialogue cell programs can run unmodified on different workstations if the same basic cells and sufficient resources are available.

The mechanism, provided by the dialogue cell language to build higher order basic cells from existing ones, allowing for arbitrary type conversions, control structures within the input device mechanism and prompts and echoes, allows one to extend a basic cell library with arbitrary interaction techniques.

5. Implementation

5.1. The graphics system used in Dialogue Cells

The graphics functions of the DICE system are based on GKS [5]. Because GKS is insufficient for the functionality of the graphics of Dialogue Cells, it was extended and adapted in three ways:

- A windowing mechanism called *grouping* was added. It allows for segments to be handled in groups, that have a common clipping window (the group window). Groups have several properties that are taken advantage of in the Dialogue Cell system. [6]
- An interface to GKS was built on top of the previous extension to allow for the handling of output from several parallel dialogue cells. This *radical system* also extends the functionality of GKS with editing functions. [2]
- Extensions to the input part of GKS, as described below.

When an atomic cell is activated or deactivated, it addresses its measure and trigger process through a GKS like DI/DD interface. All measure process initialisation functions need the following set of parameter types:

- *Mode.* (request, event, sample)
- *Prompt echo type.* This is mapped from the feedback parameters supplied to the basic cell.
- *Prompt echo data.* This includes the strings for a menu, the fixed point of a rubberband etc. The input type determines the kind of data that is necessary.
- *Result value register.* By supplying this register, the result of a user input can directly be written on the appropriate place for the scheduler to find it.

The initialisation function will use these parameters to activate the physical input device belonging to the atomic cell. A request mode parameter will result in the device being activated in event mode, with the device being disabled directly after one input. This is done, because there may be several atomic cells active in request mode in parallel. The DICE system will at appropriate times call the update function. This function will update both output and input, i.e. all input events in the workstation queue will be fetched and written to its result value register. Devices in sample mode are sampled. Because this function is always called before a dialogue cell is scheduled, the sample register of an atomic cell, will always contain the most recent value.

5.2. Currently implemented basic cells

A set of basic cells has been implemented that exemplifies the characteristics of individual basic cells and libraries. There are two sets of basic cells: one for the IBM 5080 workstation and one for the SUN workstation. Although these workstations have widely different input features, most basic cells could be ported from one to the other with no or small changes. Some can only be used on one of the two.

Atomic cells for all six GKS input classes for all triggers have been implemented. For all, the prompt echo type is a parameter of the atomic cell.

The SUN workstation has a menu prompt echo type, which shows the current choice as a diapositive box. The IBM 5080 does not have this feature, there it has been simulated by creating a menu radical from the parameters, using a pick device to obtain an input (the echo of the pick device highlights the entry), and transforming the pick identifier into a choice on input. Neither workstation has the drag feature. On both this was simulated by a basic cell that has a locator subcell in sample mode.

On the IBM 5080 a set of basic cells is developed that produce a radical containing one primitive of one of the 8 types, including the attributes. The interactive editing of the (partly) specified primitive and the changing of attributes by a number of subcells, made these into rather complicated cells.

Still they can be regarded as basic cells, because no application knowledge is needed to create the primitives. Furthermore, these basic cells support I/O symmetry, because they can be regarded as virtual input devices, returning a primitive.

The SUN workstation has two measure process, the mouse (*M*) and the keyboard (*KB*), which has a string valued measure. There are five trigger processes; three for the three mouse buttons (M_1, M_2, M_3), one that fires when the mouse is moved (M_4) and one for the keyboard (*KB*, when *return* is hit). There are basic cells *point*, that uses one of M_1, M_2 and M_3, together with *mouse*. Basic cell *stop* uses exactly the same InputClass. They can be active in parallel as childs of a basic cell *polyline*, when that has at least two mouse buttons in its InputSet; the resource manager assigns one to *point* and one to *stop*.

EXAMPLE OF 'POLYLINE' BASIC CELL:

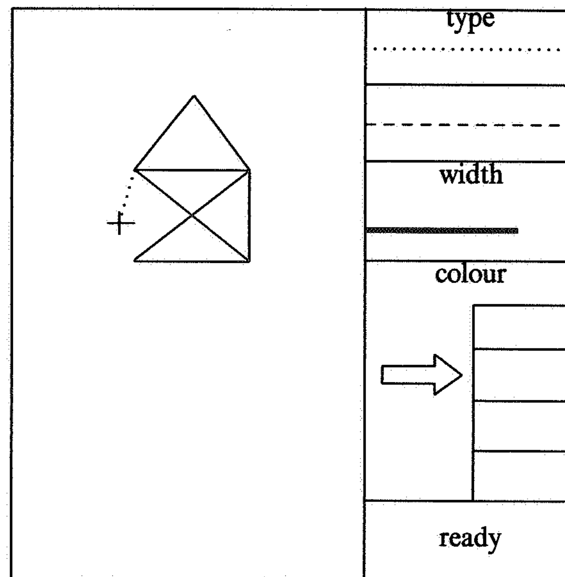

FIGURE 2

```
DICE polyline : Coords, Rad *
PROMPT
      initprompt
            WClass wcl = < rel, {{0.0, 0.0, 0.6, 1.0}}>;
            IClass icl = < {(M1,M), (M2,M)} >;
            Resource r = < wcl, icl, 0.0 >;
            Coords cds;
            Point pt;
      /* cds contains the array of points returned by point, pt the last */
            Rad  *rad
      /* radical containing the polyline */
      initsymbol
            stop: event;
            point(pt, RUBBERBAND): request
SYMBOL
```

```
      | (stop, *(point))
      /* symbol expression: stop OR (infinite) LOOP of points */
VALUE
      point:      /* code to add point to cds struct */
      return: cds
ECHO
      point: rad := pline = { cds->nbr, cds->pts };
      return: rad
END
```

FIGURE 3: (INCOMPLETE) SPECIFICATION OF BASIC CELL 'POLYLINE'

REFERENCES

1. P.J.W. ten Hagen, R. van Liere: *Introduction to Dialogue Cells*
CWI report CS-R8703, 1987

2. P.J.W. ten Hagen, H.J Schouten: *Parallel graphical output from Dialogue Cells*
Proc. of Eurographics, Amsterdam, Netherlands, 1987

3. P.J.W. ten Hagen, R. van Liere: *A Model of Graphical Interaction*
Proc. of NATO Conf. on Foundations of Computer Graphics and CAD, ed. by R.A Earnshaw, Castelvecchio Pascoli, Italy, 1987

4. R. van Liere, P.J.W ten Hagen: *Resource management in DICE*
CWI report CS-R8746, 1987

5. *Graphical Kernel System Functional Description*
ISO/TC97/SC5/WG2, 1985

6. P.J.W. ten Hagen, M.M. de Ruiter: *Segment grouping, an extension to the graphical kernel system*
CWI report CS-R8623, 1986

7. M. Green: *A Catalogue of Graphical Interaction Techniques*
Computer Graphics, jan. 1983

8. J.D. Foley, V.L. Wallace, P. Chan: *The Human Factors of Computer Graphics Interaction Techniques*
Report GWU-IIST-81-03, The George Washington University, 1981

9. IFIP WG 5.2 Workshop on methodology of Interaction, Seillac, 1979
ed. by R. Guedj

10. Proceedings of the workshop on User Interface Management Systems, Seeheim, 1983
ed. by G.E. Pfaff, 1985

11. D. Duce: *Configurable input devices.*
in: Proc. of Eurographics Workshop on GKS review, Manchester, UK, 1987, ed. by W.T. Hewitt

12. Borufka, ten Hagen, Kuhlman: *Dialogue Cells, a method for Defining Interactions* IEEE Computer Graphics & Applications, 1982, vol 2, nr 5

13. R.J.K. Jacob: *A specification language for direct manipulation user interfaces* ACM Transactions on Graphics, 1984, nr 4

14. D.S.H. Rosenthal, J.C. Michener, G. Pfaff, R. Kessener & M. Sabin: *The detailed semantics of graphical input devices* Computer Graphics, july 1982

APPENDIX

Algorithm to find an optimal resource handout for a given dialogue cell program.

The idea of the algorithm is to simulate the activation tree of the dialogue cell program and linearise that. There are two operators for which we cannot know in compile time which dialogue cells will be activated: the *right star* and the *case* operator. However, since we must deal with the worst case, we can handle these as if all operands of these operators would be activated. Note that the algortihm is not intended to be optimal; it is only to show that it is possible to find an optimal resource handout in compile time.

The algorithm consists of two steps. First a table is built with each entry containing:
- a dialogue cell incarnation identifier
- an activate/deactivate flag
- a number

The number will be initialised to 1 for all entries. Its use will become clear in the second step. The table is constructed by stepping through the dialogue cell tree and within the dialogue cells through the symbol expression in the following way.

(1) Start at the root dialogue cell.
(2) Add (*cell id, act flag*) to the table.
(3) Recursively add (*cell id, act flag*) for all subcells in event or sample mode (top down).
(4) Step through the symbol expression in preorder, doing the following. When arrived at a leaf (a dialogue cell) do:
- If it is in request mode add (*cellid,act flag*) to the table. Repeat this procedure recursively from step 3 onward.
- If the cell in request mode was in the operand list of a sequential operator add (*cell id, deact flag*) to the table.
- When arrived at the end of the operand list of a parallel operator, add (*cell id, deact flag*) to the table for all cells in request mode in the operand list.

(5) Recursively add (*cell id, deact flag*) for all subcells in event or sample mode (bottom up).
(6) Add (*cell id, deact flag*) to the table.

In the second step the table and the resource allocation algorithm are used to determine whether the given dialogue cell program can run on the workstation at hand. We will refer to the algorithm as *alloc_res*. We add to it a new parameter being a number. The idea of this parameter is the following. Given the WClass and IClass parameters the set of possible assignments is WClass $\times$ IClass. As both are finite, this set is enumerable. The number parameter of alloc_res is the index in this set which is the first that will be tried by the algorithm. Furthermore, for every resource request, the maximum value of the index will be denoted as N_R. The function *dealloc_res* frees the resource. *Alloc_res* returns a boolean value which tells whether the request was succesful. If *alloc_res* succeeds for all dialogue cells in the tabl , the dialogue cell program can run on the workstation, and the set of resource indices for each incarnation represents an optimal resource distribution.

```
cur_table_index := 1;
while cur_table_index <= table_size and cur_table_index > 0
        if declares_resource(table[cur_table_index].cell) = FALSE then
                        cur_table_index := cur_table_index + 1;
                        continue with next iteration;
        fi;
        if table[cur_table_index].flag = ACT then
                    cur_iter := table[cur_table_index].nbr;
```

```
            while cur_iter <= N_R and alloc_res(R_table[cur_table_index].cell, cur_iter) = FALSE
                    cur_iter := cur_iter + 1;
            end;
            if cur_iter > N_R then
                    table[cur_table_index].nbr := 1;
                    cur_table_index := cur_table_index - 1;
            else
                    table[cur_table_index].nbr = cur_iter + 1;
                    cur_table_index := cur_table_index + 1;
            fi;
    else
            dealloc_res(R_table[cur_table_index].cell);
    fi;
end;
if cur_table_index = 0 then
        return FAILURE;
else
        return SUCCES;
fi;
```