Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

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

A model for graphical interaction

Computer Science/Department of Interactive Systems Report CS-R8718 April

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.).

A Model for Graphical Interaction.

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

Department of Interactive Systems
Center for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

A model for graphical interaction is presented which will allow us to precisely and formally describe many important aspects of graphical input, graphical output and various correlations between these two.

The model encapsulates the fundamental properties of elementary graphics input devices and their feedback. It also encompasses the operational modes of such devices and the screen resources they occupy. On top of this, the model allows the description of compound inputs leading to the description of arbitrary complex interaction techniques. Moreover, these interaction techniques can very precisely be controlled by the application program. The latter is of importance for incorporating such techniques in a variety of methods. One of the main achievements of the model is the encapsulation of the concept called I/O-symmetry.

Finally, it is shown how the model can be used to describe various concepts such as user freedom, direct manipulation, error recovery and dialogue scheduling. Directions for further development of the I/O-unit model will be outlined.

1980 Mathematics Subject Clasification : 69K32, 69K36.
1983 CR Categories : I.3.2, I.3.6.
Key Words & Phrases : Graphics systems, Methodology and techniques.

Note : This report will be submitted for publication elsewhere.

1. Introduction.

The introduction of standards for Computer Graphics has raised great interest in providing models for basic graphics systems. Using models makes it easier to understand the scope of the graphics functions and the functional structure of the system. Moreover, such models form the basis for giving precise definitions of the semantics of the system. They therefore are also used for developing test and certification procedures

Report CS-R8718
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

for graphics standards implementations.

Models are functional schemas which, by leaving out many details, catch the major semantic framework of a system. In this way the important properties of a system can be analyzed. Most models of graphics systems are only defined informally. For instance, the models (one for input and one for output) used in the functional description of GKS [6] are described in plain English. Nevertheless, the GKS document was and still is considered quite an improvement over other documents describing existing graphics systems.

When used in interactive graphics, the existing GKS input and output models have the major drawback that the input model is completely separated from the output model. This results in a lack of control that an application program has over the feedback given by an input device. Control over the feedback given by an input device is restricted to attribute settings at initialization time of the particular device. A more generous approach would be to allow the application program to dynamically manipulate feedback to any level of detail. This can only be achieved if the input and output models are integrated.

In this paper we will introduce a new model for interactive graphics systems. The model, which is based on the concept of I/O-units, encapsulates all properties of elementary graphics input devices and their feedback. It also encompasses the operational modes of such devices and the screen resources they occupy. On top of this the model allows the description of compound inputs, leading to the description of arbitrary complex interaction techniques. Moreover, these interaction techniques can very precisely be controlled by the application program. The latter is of importance for incorporating such techniques in a variety of methods. One of the main achievements of the model is encapsulation of I/O-symmetry.

In chapter two an input and output model for a GKS-like graphics system will be developed. This will allow us to give a simple example of such models and, at the same time, explain the shortcomings of these models when used in interactive graphics. In chapter three the new model will be introduced and it will be shown how the missing functionality is covered by the new model. Finally, in chapter four it is shown how the model can be used to describe various concepts such as user freedom, direct manipulation, error recovery and dialogue scheduling. Directions for further development of the I/O-unit model will also be outlined.

1.1. Related work.

Various efforts have been taken to apply formal specification techniques to the specification of computer graphics systems. The most notable are reported by Carson [2], Gnatz [4], Mallgren [7], Duce and Fielding [3]. The vast majority of the work done in this field concentrates on graphical output; i.e. the geometrical aspects of output as well as the controlling output functions.

Work on alternative interaction models has also been done by van den Bos [12], Anson [1], Hopgood and Duce [5], and Shaw [10]. The model proposed by van der Bos,

although parallelism is possible, is strictly synchronous. Furthermore, as Matthijs reports in [8], problems concerning input ambiguity have not (yet) been adequately solved.

2. GKS-like systems.

2.1. GKS output model.

This section introduces a simplified model describing the output functionality of a GKS-like graphics system. It is not our intention to encapsulate the GKS output functionality completely. However, the model does include the mechanisms the GKS provides for manipulating dynamic pictures, i.e. pictures that can change in real-time during interaction. In GKS, only pictures created in segments can be dynamically manipulated. The model therefore considers only segments and not so-called "primitives outside segments". Moreover, minor details such as the precise appearance of a particular output primitive (for instance, what it means when a primitive is *red*, has *thick* lines, etc.) will not be taken into account. These details, although indispensable in a practical implementation of GKS, are of no importance for dynamic aspects of picture manipulation and, hence, are left out of the model.

GKS has a fairly complex, three step, picture creation mechanism. First a segment must be created. This results in a unique segment header which contains a set of segment attributes. Then output primitives together with their attributes are entered into segments, resulting in a segment body. The output primitive attributes, which control the appearance of the primitive, are taken from a previously prepared state list. Finally, the segment must be closed. After creation of a segment, the primitives and their attributes within the segment body cannot be altered. Only the segment attributes in the header can obtain new values. Due to the static nature of a segment body, the model simplifies the multi-step segment creation process to a one-step creation of complete segments.

The simplified output model considers four categories of functions which can change the appearance of a picture. The second and third categories are considered dynamic because they can alter an already existing picture rather than add or delete something. The four function categories are:

- creation / deletion of segments.
- changing segment attributes.
- changing workstation attributes.
- changing the update state.

For each of these function categories there exists a disjunct category of data which stores the results of these functions. The data categories can be described by a graphical output state G_O, denoted as the triple $G_O = <S, W, U>$, in which:

- $S = \{ S_i \}_{i \in N_S}$
  is a list of segments having a unique index from the index set N_S. Every segment

S_i consists of a header and a body, denoted as S_{h_i} and S_{b_i}. † Only the header of each segment can be changed dynamically.

- $W = < \{Tr_i\}_{i \in N_{tr}}, \{B_{t,j}\}_{t \in P_t, j \in N_t} >$
  is the workstation state, consisting of a list of transformations Tr_i and a list of attribute bundles $B_{t,j}$ for output primitive type t with a unique index from the index set N_t. P_t is the set of output primitive types, consisting of {*polyline*, *polymarker*, *fillarea*, *text*, *cellarray*}.

- $U = <D, R, WS>$
  is the update state consisting of two value fields, D and R, denoting the deferral state and regeneration mode respectively. Furthermore there is a field, denoted as WS, which contains a set of segment indices, bundle indices and possibly transformations which are not yet effectuated on the workstation. If $WS = \varnothing$, then the actual picture is completely up to date. D and R contain control variables indicating in what way WS becomes empty, respectively non-empty.

The four function categories can now be described as:

- S-functions: *New* (S_i), *Del* (S_i)
  for creating, respectively deleting, a segment
- H-functions: *New* (S_{h_i})
  for changing (part of) the segment header of S_i.
- W-functions: *New* (Tr_i), *New* ($B_{t,j}$)
  for changing the various components of the workstation state.
- U-functions: *New (D)*, *New (R)*, *Update ()*
  for changing the deferral state respectively the regeneration mode. The function *Update*() ensures that $WS = \varnothing$ holds.

2.2. GKS input model.

This section will first describe the input functionality of GKS and then explain which part of the graphical output state can become involved in input processes.

GKS defines a number of logical input device classes. Each abstract input device is *only* characterized by the data type it returns. Abstract input devices are mapped onto physical devices. How this is done is of no concern to GKS. It is hidden behind the device class, and thus is outside the scope of the input model.

The operational behaviour of a logical input device can be controlled at device initialization-time. This is achieved by setting various predefined attributes (cf. GKS [6]). Examples of these attributes are the echo area, the prompt echo type and the initial value of the input device. However, once an input device is initialized, the attribute

† We will not elaborate on the contents of a segment other than that the header consists of a set of attributes and that the body consists of a set of output primitives.

values cannot be changed. Due to the static nature of attribute setting, the model will consider device initialization and device activation to be combined in one operation. A logical input device can be activated in one of three modes, each of which determines the way an input value becomes available to the application program:

- in **request**-mode a single input value is taken from an input register. Reading this value implicitly terminates the activation mode.
- in **sample**-mode the input value is also taken from an input register. However, the value will change when the device is operated. As a result, the value taken is the most recently produced one. In this case reading does not terminate the activation mode.
- in **event**-mode the input value is taken from a FIFO (first in-first out) queue of input registers. Reading does not terminate the activation mode.

In request and event mode reading implies waiting if no input value is available.

GKS merges all the event registers into one queue. This makes it awkward to wait for a particular event. On the other hand it is possible to wait for any event (i.e. the queue is not empty). However, to provide an event queue, the underlying support system will have to provide a facility for asynchronous input. Therefore it would have been more adequate to provide direct access to such a facility, leaving it up to application to define a particular scheduling strategy. Access to asynchronous input could be provided through maintaining an event queue per device.

GKS requires that reading reflects the operating mode : it has three read calls, one for each mode. This is functional redundancy as the call must match the actual mode.

This can be summarized by describing a graphical input state by a pair, denoted as $G_I = <L, M>$, in which:

- $L = \{L_{c,\,i}\}_{c \in I_C,\, i \in N_C}$
  is a list of logical input devices. $L_{c,\,i}$ is the logical input device of input class C and unique index from the index set N_C.

- $M = \{M_{c,\,i}\}_{c \in I_c,\, i \in N_C}$
  is a list of logical input device modes, including their initial state. $M_{c,\,i}$ is the activation mode of the corresponding logical input device $L_{c,\,i}$.

Furthermore, there are two function categories which can operate on the input state G_I.

- M-function: New ($M_{c,\,i}$)
- L-function: Read ($L_{c,\,i}$)

Read ($L_{c,\,i}$) means sample, request or read-event depending on the mode $M_{c,\,i}$.

2.3. GKS interaction model.

The total graphical system can be viewed as the pair $G = <G_O, G_I>$. Static aspects in G are not considered since they cannot influence the interplay between input and output. This follows from the observation that G_O and G_I generate different objects. The interplay can only result from dynamic changes (alternations of an object after creation). Hence, only dynamic aspects in G will be considered.

In GKS there exist some very restricted relations between elements of G_O and G_I. These are summarized as:

- The first type of relation considers the synchronization between input and output. This can be expressed as follows:

$$\forall\, L_{c,\,i}\,,\; c \in I_C,\; i \in N_C : U\,[D] = BNI \wedge \neg\,(M_{c,\,i} = \textbf{event} \wedge Q_{c,\,i} \neq \varnothing)$$
$$=> \{U\,[WS] = \varnothing\}\; Read(L_{c,\,i})$$

  For every logical input device, $L_{c,\,i}$, if the update state is BNI (before next input) and an input value was not already available in the event queue then $\{WS = \varnothing\}$ is a precondition for the $Read(L_{c,\,i})$ function.
  The GKS standard even tries to achieve something stronger. If the update state is BNI, no new event may be created or sample values may change unless the picture is up to date. However, this request cannot always be implemented because the system cannot establish that a new value is being created by the operator. This can therefore not be certified.

- The second type of relation considers input values which refer to output elements. The pick device returns a segment index. This relation can be denoted as:

$$\forall\, L_{P,\,i},\; i \in N_P : Read\;(L_{P,\,i}) \in$$
$$\{\, x \mid x \in \{N_S \vee \varnothing\} \wedge S_x \in S \wedge S_{h_x}\,[VIS] \wedge S_{h_x}\,[DET]\,\}$$

  where $S_{h_x}\,[v]$ denotes that the header of the segment with index x contains the value v. Input from a pick device consists of a segment index of an existing segment which is visible and detectable. In the case that no segment index is picked, $Read\;(L_{P,\,i})$ will return a null value.
  The point being made is that the pick input mechanism does not alter the graphical output state. Hence the visual effects of picking a segment are not in G_O although the segment is. One would expect that the segment picked is somehow marked. The mark, however, is neither in G_O nor in G_I (i.e. it cannot be expressed in the model of GKS).
  Also the process of picking itself, e.g., how it operates on the data of G_O or how the candidates for picking are identifying themselves, is outside the model. This is generally true for GKS: the visual effects of operating the input devices cannot be controlled.

- The locator device returns a position and a reference to the transformation which was used to map the position from screen coordinates into some internal

coordinate system. This coordinate system is one that can be used to generate output from because it appears in the workstation state. GKS describes the mechanism by which the locator device determines the transformation fairly precisely. There is, however, no description of the visual effects.
All coordinate transformations in the workstation state list contain a clipping rectangle in virtual workstation coordinates. All screen coordinates have the common mapping on these virtual workstation coordinates. The locator input is mapped from screen to virtual workstation coordinates. Moreover, there is an unambiguous way to determine the clipping rectangle. All transformations are sorted on the basis of assigned priorities. The clipping rectangle with highest priority that contains the locator point identifies the transformation used.
In the output model we presented, this is all contained in $G_O\ [W\ [Tr_i]]$. There is a function which, given a coordinate value, will return the transformation index. The fact that in GKS the information about transformations is distributed over several state lists is immaterial here. The relation between a locator point and the transformation system can be summarized as follows: †

$$\forall\ i \in N_L\colon Read\ (L_{L,\ i}) \in$$

$$\{\ <ntr_{maxp},\ pos> \mid ntr_{maxp} = maxp\ (\ \{\ ntr \mid ntr \in \{Tr_i\} \wedge pos \in ntr\ [clip]\ \}\ )\ \}$$

The position is transformed by the transformation with highest priority whose clipping rectangle contains the position. *maxp*() is a function that, given a set of transformations and a particular position, will return the transformation with the highest priority which contains the position in its clipping area.
This locator input device does reveal a serious deficiency. It is important that the transformation list $\{Tr_i\}$ is properly set-up (done through *New*(Tr_i)-functions) prior to a locator device activation. Two locator devices may require different orderings for Tr_i. Hence, if these settings are in conflict the locator devices cannot be simultaneously active. In most cases it would be sufficient for both devices to select their substate of Tr_i. The substates may be without mutual conflicts. This is a typical example of both input and output needing to access the same state.

This completes the relations between graphical input and output that can be made in GKS. It looks very much like a unsystematic set of facilities, certainly compared to how systematically output and input themselves are structured. Before developing a model for interaction, we will summarize the deficiences of the GKS-like model:

- Every interactive graphics system uses dynamic pictures to provide real time feedback for the logical input devices. The implementation must have a private graphics output system to provide this feedback. Such pictures could be drawn using GKS-like output segments. However, in reality this part of the picture making is entirely beyond program control. In order to change this, the method for controlling the picture must accept other sources of picture creation and also allow input processing to directly control parts of the picture even if they originally were

† A similar relation can be defined between transformations and stroke input.

created by way of output. In addition, such changes must be able to occur simultaneously because input devices can be active simultaneously.

- A second deficiency of GKS is the lack of control the program has over the update state. GKS very crudely redraws all segments at update. This is because there is no way to determine a minimal set of segments for which redraw for a particular update is necessary and sufficient. This is probably the most serious obstacle to be overcome for efficient, program controlled, feedback.

3. Graphical interaction through I/O-units.

3.1. Introduction.

The most basic concept that we try to capture with input/output units is the association between a picture generated on the screen and an internal representation of that picture. For instance, a geometric modeler maintains an internal representation of the geometry of an object. A visualization algorithm can generate a view of this object using the geometric description. If the internal model changes, the picture should change accordingly and vice-versa. This means that the geometric model and its visualization are correlated. We say that they belong to one I/O-unit.

Of course there are also much more elementary units. For instance, a point on the screen can be represented by an $<x,y>$-value and a small marker drawn at the point. The value of the point and the marker belong to one unit. Moreover, the point may be part of the marker's description. Hence, there may be redundancy in the unit.

Conceptually, the I/O-unit model, $G_{I/O}$, builds on top of the GKS-like input and output models. $G_{I/O}$ includes additional functionality that allows the interactive program to avoid the deficiencies cited in the previous chapter. In particular, the following points are of interest (both will be elaborated in the subsequent sections) :

- the feedback provided by input devices is included in G_O. This allows the program unconstrained access to the feedback mechanism.
- a sophisticated synchronization mechanism is provided. This will allow the program a more refined control over the update state.

In order to judge the validity and usability of this description method, the following issues must be addressed:

- Is it possible, by using various predefined composition rules, to describe large compound units in terms of smaller units? If this is not so, which constraints must be put on the object being described.
- If such a composition is possible, will it then be possible to visualize the composition process itself? The extent to which this can be done dynamically is perhaps the best way to characterize the interactivity of computer graphics.

We will address both issues in subsequent sections, but first we will give a formal introduction to I/O-units. The rest of the chapter is organized as follows: first we will show

how I/O-units, I/O-unit environments and so-called transaction units are characterized. Then we will define three categories of semantic actions associated with I/O-units, environments and transaction units. The three categories of semantic actions will define:

- how an I/O-unit is merged into an environment.
- how an I/O-unit environment comes into existence or ceases to exist.
- how I/O-units themselves are created and when they are merged into another environment.

3.2. I/O-units.

An I/O-unit is a triple :

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

in which S is a symbolic name, I is a (possibly complex) input value, O is a (possibly complex) picture output. [†] An input value, i, and a picture, o, are said to be *correlated* if there is an I/O-unit S such that $I_S = i$ and $O_S = o$.

An interactive graphics system can be described in terms of manipulating I/O-units. The three components of the I/O-unit are characterized as follows:

- the input part, i, of the unit is an abstract internal representation of a (possibly complex) input from the user.
- the output part, o, of the unit is the specification of a picture on the screen. All pictorial parts of the unit, whether produced by the program or the system (i.e. such as echos), are in the o-part of the unit.
  The key point is that there is only one specification language for output. This means that *both* program output as well as feedback generated by input devices are produced by the same logical process. This is what is referred to as *input-output symmetry*.
- the symbolic name, s, which will be called the *symbol*. The symbol stands for the particular correlation between the input and output component of that I/O-unit. In many cases the meaning of the correlation is that the output part is a visualization of the input part. However, entirely different meanings can be given to a correlation as well.

Both the i-part as well as the o-part may be empty: $<S_1, \varnothing, O_1>$ respectively $<S_2, I_2, \varnothing>$. In case of strict separation of input and output (as in GKS-like systems) one could say all I/O-units have an empty component.

Examples of I/O-units are:

- $P = <locator, <x, y>, dr\_xhair\ (<x, y>)>$.
  which defines a position , $<x, y>$, and a cross-hair drawn on the screen, with the cross at $<x, y>$. The name of the unit is *locator*. The function *dr_xhair* ()

[†] Sometimes the notation $<i_s, o_s>$ or $<i,o>_s$ is used for the same concept $<S, I, O>$

denotes the visualization of the cross-hair at position $<x, y>$.

- $P = <$*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 visualized by the function *dr_menu ()*, which places the *strings* in a *box* with **exit** being highlighted to indicate the selection.
- $P = <$*bspline*, $<pt_1, pt_2, \cdots, pt_n>$, *dr_curve* ($<pt_1, pt_2, \cdots, pt_n>$).
  A sequence of points is input. The display shows a bspline-curve through these points.

In each example the pictures drawn by *dr_xhair* (), *dr_menu* () and *dr_curve* () are actually defined using graphics output functions (for instance those of G_O may have been used).

3.3. I/O-unit environments.

Every I/O-unit contains a picture fragment and a data fragment. If such a unit exists then there also exists an environment to which it belongs. This environment is in charge for maintaining the picture as well as an internal information structure. The I/O-unit contributes its picture and data fragment to the relevant portions of the environment.

An environment consists of a picture environment component, a data environment component and a set of relations between elements of both components. The relations explain how correlations between the input and output component can have their effect in processing I/O-units. They also explain how new I/O-units can emerge from an environment.

A complete environment is denoted as $E = <D, R, U>$ in which D is a data environment, R is a picture environment and U is a set of relations between elements of D and R. In the current version of the model U is not explicitly defined. This is, temporarily, acceptable because U can be empty and the relations may exist implicitly in the functions that will process I/O-units.

3.3.1. The picture environment.

A picture environment is denoted as $R = <Tr, A, \{ r_i \}_{i \in N_R}>$ in which

- Tr is a transformation and clipping rectangle.
- $A = < \{B_t\}_{t \in P_t} >$ is an attribute state containing an attribute bundle for each type of output primitive.
- $\{ r_i \}_{i \in N_R}$ is a list of so-called *radicals*. These are compound picture primitives, similar to segments, but with a different controlling functionality. Unlike segments, radicals do not contain their own clipping rectangle.
  Radicals are created by adding or deleting output primitives to, respectively from, them. Every primitive added to a radical automatically gets an identifier. In every picture environment the pair $<$*radical identifier*, *primitive identifier*$>$ is guaranteed to be unique. The output primitives can have individual or bundled attributes (cf. GKS). Furthermore, individual radicals can as a whole be transformed, highlighted, made visible / invisible and detectable / undetectable.

Each picture environment is self-contained. After each change, an environment is automatically updated as soon as possible (ASAP (cf. GKS)). It has all necessary information for producing graphical output. The clipping rectangle of *Tr* applies to all radicals in the environment. The transformation part of *Tr* contains one matrix transformation which can dynamically be changed. This transformation also applies to all radicals in the environment. Furthermore, the clipping rectangle can have a shielding background color. The coordinate system of *Tr* and its' radicals are always defined when the environment exists. How this is done will be defined when the interplay between environments is discussed.

For the aspects of the model that we are considering, it is not necessary to explain the radical system in detail. This has been done in [9] .

Several picture environments may exist simultaneously. A run-time system could automatically maintain information about the overlap between environments on the display surface, by using the available clipping and shielding attribute values. The overlap information is essential to minimize the redraw actions during updates.

Radicals can be part of only one environment at a time. The only exception to this rule is the picking mechanism which, as will be described below, has restricted access to other environments.

All picture fragments of I/O-units are elements that can be extracted from or added to environments. For example: a global transformation, an attribute bundle, a radical or a list of radicals. The following table displays a list of radical functions for altering and accessing an existing picture environment.

Control functions	Changing the environment	Radical manipulation	Primitive manipulation	Inquiry functions
CREATE ENV	SET LINETYPE	CREATE RAD	REPL. DATA	INQUIRE ENV
DELETE ENV	SET MARKERSIZE	SET VISIBILITY	CHANGE TYPE	INQUIRE RAD
UPDATE ENV	...	SET AREA STYLE	DELETE IND. ATTR	SET IND. LINETYPE
SEND RAD		SET TEXT HEIGHT	ADD POLYLINE	SET IND. MARKERSIZE
ACCEPT RAD		REMOVE PRIM	...	...
		SHIFT RAD		
		CONCAT RAD		

3.3.2. The data environment.

The data environment is much more schematically defined. This is because the actual data types and data manipulations are defined by the application shell around the interactive graphics part.

The scheme of the data environment is as follows: There is a list of data types given which can occur in I/O-units. For each of these data types operations for adding or removing values of such data types are given. In addition, a data environment may have local data types and message data types for building up temporary local data or communicating data elsewhere.

A data environment consists of three data sets, one for each type category. Additional

functions can be supplied for type conversion between the various categories and for sending messages to the application shell. There are also functions which can create new picture fragments from elements of the data environment. All these functions necessarily reside in the application shell.

A data environment is denoted as the triple:

$$D = <D_i, D_l, D_m>$$

in which D_i, D_l and D_m are sets of data items whose types occur in the input type list. Each data items of D can be referenced individually.

3.3.3. Environment structures.

Every environment belongs to a so-called *active transaction process*. An active transaction process is a process which creates a particular I/O-unit. The active transaction process is characterized by a descriptor which contains:

- the name of the I/O-unit produced by the transaction,
- the name of the transaction process to which this I/O-unit will be transferred,
- a so-called incarnation identifier which assigns a unique identification to each descriptor which has the same name for either producer or consumer of this I/O-unit.

For each active transaction process there exists a so-called *transaction unit* which is defined as follows:

$$T = <S_c, M, N, S_p, \{<I, O>\}^* >$$

in which S_c is the consumer of the I/O-unit $<S_p, I, O>$ produced by S_p. M is the activation mode of T and N is the incarnation identifier. Hence S_c, M, N and S_p together form the descriptor, so the descriptor is contained in the transaction unit.

The transaction unit $T = T_p$ is said to belong to S_p and will exists as long as S_p is active (i.e. producing I/O-units for S_c). S_p will be activated by the active transaction process belonging to S_c and will eventually also be deactivated by S_c.

If T_p exists then S_c and S_p are both active and there exists a relationship between E_c and E_p, where E_c is the environment belonging to T_c and E_p is the environment belonging to T_p. The relationship can be characterized by:

- E_p is created by the active transaction process S_c and initially

  $$E_p = <<D_{ic}, D_{lc}, D_{mc}>, <Tr_c, A_c, \varnothing>, U_c>$$

  The initial environment inherits the transformation and attributes from the activator. It has no radicals and the shielding is off.
  This environment is further initialized by the active transaction process S_p. During initialization the clipping rectangle and shield may be redefined. From then on they will remain static until the transaction is deactivated and the environment is

removed.

- The data type of the *I*-unit produced by E_p belongs to D_c. I/O-units cannot return values or pictures of unknown type.
  If specified in the creation parameters, then E_p may initially contain radicals from E_c. This means that these radicals are removed from E_c. They may, if necessary, be returned with the I/O-unit. This conforms to the rule that radicals can be an element of one environment at a time.
- There is a default environment, denoted as E_0, which is given to a root transaction, i.e. a transaction which has no consumer. This default environment also defines an initial (absolute) coordinate space. Hence, through inheritance, all environments are guaranteed to have a defined coordinate space.
- The picture environment allows a given environment read-only access to another environment (usually a producer environment). This is for picking a radical. The picked radical can be individually redrawn using the attributes from the pick environment. The I/O-unit returned by the pick process can contain the radical identifier. Only in the case that the environments are the same, can the I/O-unit return the radical as well. It is then removed and transferred to the consumers' environment.

3.4. Transaction handling.

If an active transaction process S_p produces I/O-units for S_c, then there is a transaction, denoted as:

$$T_p = <S_c, M_p, N_p, S_p, \{<I, O>\}^*>.$$

T_p exists only for the purpose of properly transferring the I/O-unit $<S_p, I, O>$ to S_c.

M_p, denoted as $M_p = <due\_flag, fill\_flag, mode>$, is a compound unit which characterizes the state of the transaction unit. The due_flag is either **true** or **false**. If **true** then the consumer is ready to accept the I/O-unit. As long the due_flag is **false**, the I/O-unit cannot be transferred to S_c. The fill_flag is also either **true** or **false**. If **false**, the producer S_p has not (yet) produced an I/O-unit for S_c. Hence, both flags must be **true** in order for S_c to be able to proceed with S_p.

The field M [*mode*] can have values:

- **request**. In request mode T_p is initialized as:

  $$T_p = <S_c, M_p\ [due\_flag = \textbf{true}, fill\_flag = \textbf{false}], N_p, S_p, \{<\varnothing, \varnothing>\}>.$$

  As soon as an I/O-unit is produced

  $$T_p = <S_c, M_p\ [due\_flag = \textbf{true}, fill\_flag = \textbf{true}], N_p, S_p, \{<I_p, O_p>\}>.$$

  T_p is ready to be processed by S_c.
  In the case of request mode, T_p will be removed as soon as S_c consumed the I/O-unit (i.e., the transaction process S_p is deactivated as soon it produces one I/O-unit). †

† Whenever S_c consumes the I/O-unit produced by S_p, we say that the I/O-unit is *transferred.* The source and destination of the transferred I/O-unit is always unique (it is recorded in the

- **sample**. In sample mode initially:

$$T_p = <S_c, M_p\ [due\_flag = \textbf{false}, fill\_flag = \textbf{true}], N_p, S_p, \{<I_{ci}, O_{ci}>\}>.$$

$<I_{ci}, O_{ci}>$ is an initial I/O-unit assigned to S_p , which may be provided by S_c, through a parameter passing mechanism. Each time S_p produces an I/O-unit the value T_p $[<I_{ci}, O_{ci}>]$ is overwritten. Each time the due_flag becomes **true** the current I/O-unit is transferred.
In the case of sample mode, S_c must remove T_p explicitly.

- **event**. In event mode, initially:

$$T_p = <S_c, M_p\ [due\_flag = \textbf{false}, fill\_flag = \textbf{false}], N_p, S_p, \{<\varnothing, \varnothing>\}>.$$

Each time an I/O-unit is produced, the fill flag is set **true** and the I/O-unit is appended to $P_p : <S_p, I^n, O^n> \rightarrow <S_p, I^{n+1}, O^{n+1}>$ resulting in the transaction unit:

$$T_p = <S_c, M_p\ [due\_flag = \textbf{false}, fill\_flag = \textbf{true}], N_p, S_p, \{<I, O>\}^{n+1}>.$$

When both flags become **true** the first (oldest) I/O-unit will be transferred to T_c.
In the case of event mode, S_c must also remove T_p explicitly. In the case of **request** and **event** mode, when the I/O-unit is transferred it is removed from the producers environment. In **sample** mode, a copy of the I/O-unit is transferred.

We have now introduced the concepts:

- I/O-units
- environments
- transactions

Using these concepts we can now describe the complete model for graphical interaction based on I/O-units. This model will contain two abstract machines:

- *the transaction mediator.* The transaction mediator is responsible for activating and deactivating transaction units. Furthermore, at the proper time, it will transfer all produced I/O-units to the consumers' active transaction process.
- *the transaction processor.* The transaction processor executes every individual active transaction process. Every step in this execution consumes one I/O-unit. Sometimes a step may result in sending / receiving messages to / from the external world. Moreover, as in the case of **sample** and **event** mode, a active transaction process may reach the final step several times. Each time a final step is reached an I/O-unit will be created and transferred to another active transaction.

The transaction processor interfaces with the transaction mediator and vice-versa. Each time certain conditions are satisfied the interface may be involved in sending control signals or transferring I/O-units for a particular process.

We will now describe the transaction mediator and the parts of a transaction process

---

corresponding transaction unit).

that interface to the mediator.

3.4.1. The transaction mediator.

The mediator maintains a transaction unit pool, denoted as:

$$TP = \{T_s\}_{s \in N_T}$$

which is the set of transaction units for which an active transaction process exists. The mediator can execute the following instructions on the pool:

- *New* (S_c, S_p, *mode*)
  A new transaction unit is added to the pool. The function *New* calculates the value of the incarnation identifier, N, itself. The initialization of the mode field, *Mode*, is as described in the section on transaction handling. In case M [*mode*] = **sample**, *New* will also call the function *Produce* (S_c, S_p, $<I_{ip}, O_{op}>$) where $<I_{ip}, O_{op}>$ is the initial value of the producer (see the *Produce* function below).
- *Due* (S_c, S_p, N)
  The transaction unit, uniquely identified by S_c , S_p and N, has its' due_flag set to **true**.
- *Produce* (S_c, N, S_p, $<I, O>$)
  A new I/O-unit is written to the transaction unit. How this is done was given in the section on transaction handling. The fill_flag is set to **true**.
- *Select* ()
  The mediator selects a transaction unit from the set of transaction units for which T [M [*due_flag* = **true**, *fill_flag* = **true**]]. The selected I/O-unit is transferred to the consumer transaction process, S_c. This transaction process will make the execution step for $<S_p, I, O>$. The due_flag of the selected transaction is set to **false**.
  If more than one candidate for selection exists, the mediator will use an arbitration algorithm based on a priority driven scheduling strategy. This scheduling strategy will not be discussed here.
  If no candidate for *select* exists, the selection process goes into a waiting state. After a change in the pool, caused by some of the other functions, a new selection may be attempted.
- *Remove* (S_c, S_p, N)
  The transaction unit, uniquely identified by S_c , S_p and N, is removed from the pool of transaction units. If I/O-units in this transaction unit are still pending, they will be removed as well.

The model is not making any restrictions for parallelism. This means that several concurrent active transaction processes might be communicating with the mediator. Also, the mediator might continue selecting until all selectable I/O-units have been transferred to a transaction process. We simply assume that the pool administration is sufficiently equipped with semaphores to allow concurrent access.

3.4.2. The transaction processor.

The transaction processor conceptually processes all active transaction units simultaneously. For each active process the transaction processor is either:

- waiting for an I/O-unit (which will be given to the transaction processor by a *select ()*-function of the mediator).
- or is busy processing an I/O-unit. Initially, after having activated a root transaction, which in turn has activated a whole cascade of sub-transactions, everything will wait until an operator provides input. This input will be be converted to a I/O-unit by basic active transaction processes. Processing of transaction units will stop when the root-transaction process completes.

A transaction process description has four different sections:

- **prompt** section.
- **symbol** section.
- **value** section.
- **echo** section.

The prompt and symbol section contain (among other things) all the necessary mediator interface calls for this transaction. The value and echo section specify how every received I/O-unit contributes to the data, respectively the picture environment, of that transaction. The value and echo sections together synchronously specify how the I/O-units of this transaction process are produced.

The active transaction process can be denoted as:

$$AT = <PROMPT,\ SYMBOL,\ VALUE,\ ECHO,\ P>$$

in which $P = <S,\ I,\ O>$ denotes the I/O-unit that will be produced by the transaction process. The name of the transaction process is $AT\ [P\ [S]]$. It is usually clear from the context when the name of the transaction process or the name of the I/O-unit is meant.

We will now discuss each section in more detail and emphasize the way I/O-units can be combined into new I/O-units.

3.4.2.1. The prompt section.

The prompt section consists of a list of different symbols (i.e. I/O-unit symbolic names) and an operating mode for each name. Furthermore, the prompt section specifies the initial environment. For the data environment, both data structures and initial values can be specified. The initial environment is generated when the transaction process is activated.

The prompt section is denoted as:

$$PROMPT = < \{ <S_i,\ M_i,\ <I,\ O>_i >\}_{i \in N_a}, init\ (<D,\ R>) >$$

in which N_a is an index set. Every S_i is a producer of I/O-units for S. M_i is the mode

in which the transaction process S_i is operating. $<I,\ O>_i$ is the optional initial value.

init ($<D,\ R>$) specifies the initial environment. The specifications of the initial environment may somehow be incomplete. The missing parts are either inherited, passed through as parameters or are defaulted. These mechanisms have been omitted in this introduction of the model.

The *PROMPT* section issues the *New* (S_c, S_p,*mode*)-calls to the mediator for every $S \in \{ S_i \}$. This either happens at initialization-time (i.e. in **event** and **sample** mode) or when an I/O-unit becomes due (i.e. in **request** mode).

3.4.2.2. The symbol section.

The symbol section is denoted as:

$$SYMBOL = g \in RegExpr\ (<S_{i \in N_a} \cup \$,\ Ops>)$$

in which g is a regular expression [†] consisting of S_i and $\$$ as operands and a fixed operator set. Each S_i must belong to the list of symbols specified in the prompt section. Operators define, for any term of g, how it synchronizes with other terms with respect to becoming **due**. The operator set contains the following operators:

- *";"*, the sequence operator.
  S_1 ; S_2 means: S_2 becomes due as soon as S_1 has been consumed.
- *"$\wedge$"*, the and operator.
  $S_1 \wedge S_2$ means: S_1 and S_2 become due simultaneously, each one becomes undue only after being consumed.
- *"$\vee$"*, the or operator.
  $S_1 \vee S_2$ means: S_1 and S_2 become due simultaneously, as soon as one has been consumed will all become undue.
- *"* cond"* or *"cond *"*, the repetition operator.
  S * *cond* means: S stays due until *cond* becomes **false**.
- *"case"*, the branching operator.
  based on the value of the *case*-identifier one of a number of alternatives becomes due.

The *SYMBOL* section issues one or more *Due* (S_c, S_p,N)-calls depending on the progress through the regular expression. Each time the regular expression is completely evaluated (i.e. the last symbol has been consumed), the function *Produce* (S_c, N, S_p,$<I,\ O>$) is called. Synchronization rules with the VALUE and ECHO part guarantee that an I/O-unit is ready for to be transferred to the mediator at that moment.

[†] We will not elaborate on the syntactical structure of the regular expression. The syntax is quite arbitrary. However, we do assume that regular expressions can be decomposed of a number terms.

3.4.3. Transaction hierarchy.

The transaction processes activated by an active transaction process, S, are called sub-transactions of S. S is the consumer of the I/O-units of its sub-transactions. These sub-transactions produce I/O-units for S. The producer-consumer relation induces a hierarchy among transaction processes. This hierarchy, determines the activation chains; i.e. if S gets activated it will in turn activate all its' sub-transactions and so on. Finally, there are elementary transactions processes which have no descendents. Elementary transaction units can be mapped onto physical devices, or they can represent basic interaction techniques that come with a certain workstation environment.

The PROMPT section can be characterized as the section which controls the *creation* of the I/O-units needed for that transaction. The SYMBOL section part defines independently the *order* in which the I/O-units can be consumed. This is done by determining, after each step in the transaction process, which I/O-units are due.

3.5. Merging I/O-units into the environment.

If an I/O-unit, P_p, is produced then the transaction unit associated with P_p, say T_p, determines where P_p has to go. Sooner or later (how this is done will be explained below) P_p will be merged into it's consumers' transaction process S_c. Hence P_p migrates from E_p to E_c.

The merging is specified in the VALUE and ECHO sections. If no merging action is specified explicitly, the default merge action will result in (assuming that $P_p = <S_p, I_p, O_p>$):

- $D_c\,[S_p] = I_p$.
  D_c contains a variable, named $D_c\,[S_p]$, and I_p is assigned to this variable.
- $R_c \cup new\,(O_p)$.
  The picture fragment O_p is added to R_c, through *SEND RAD* and *ACCEPT RAD* function.

If a merge action is specified in the transaction process S_c with respect to S_p, then S_c will contain two rules, which have S_p as label, i.e. S_p: D-statement and S_p: R-statement. These rules exist in addition to the default rules (so they may assume that the merge has already taken place).

Merging actions for D and R will occur *simultaneously* and an update will occur only after all rules labeled by S_p have been evaluated. For instance, if O_p is a list of radicals to be added to R_c, then the whole list will appear in E_c as a result of one update action. Hence, I/O-units and merge rules can define the "update chunks".

We will not describe the merging functions themselves. We will only give a framework in which these functions are activated. Within this framework, the merging functions can be different for each transaction process.

3.5.1. The value section.

The VALUE section consists of a set of rules such that

$$VALUE = <VR_i>_{i=1,...,n}$$

in which VR_i is a rule of the form:

$$VR = <TRIGGER, IMPORT, TRANSPORT, EXPORT, STATUS>$$

We will only consider the *TRIGGER* and *IMPORT* in more detail. The intuitive meaning of each component is as follows:

- *TRIGGER* is a so-called *input configuration*. It is a proper sub-expression of the grammar rule given in the *SYMBOL* section. Minimally, all individual names of I/O-units (i.e. S_i, $i \in N_a$) appear as such at least once as a *TRIGGER* rule, either explicitly or as default rule. The meaning is that each time this sub-expression is satisfied, the "trigger fires" and the VALUE rule is evaluated.

  A default rule has the form:

  $$S_i : D_c\ [S_p] = P_i\ [I]$$

  or written in the *VR* framework:

  $$VR = <S_i, D_c\ [S_p] = P_i\ [I], \varnothing, \varnothing, \varnothing>$$

  Minimally, the input component of the I/O-unit (P_i) with name S_i is upon selection assigned to an internal variable s_i. In this way subsequent rules can access $P_i\ [I]$ from the data environment.
- *IMPORT* is an expression for merging the input part into the data environment. The expression can use value range tests and conditions on the actual value of $P_i\ [I]$. In particular, these tests may fail. This means that the input, and hence the entire I/O-unit, must be rejected. How this situation is handled will be discussed in the section on error recovery.
- *TRANSPORT* and *EXPORT* will contain expressions for adding fragments of *I* to other data , respectively, sending a message to the outside world.

Generally speaking *IMPORT* selects (a part of) the input structure and maps it onto an internal value. The further processing of *TRANSPORT* and *EXPORT* can use this selected and mapped result.

The *VALUE* section contains one special rule which has the *$*-sign as trigger. The *EXPORT* part of the *$*-rule will contain an expression which produces the *I*-part of the I/O-unit. The *$*-sign is equivalent with the entire regular expression of *SYMBOL*.

3.5.2. The echo section.

The ECHO section also consists of a set of rules.

$$ECHO = <ER_i>_{i=1,...,n}$$

in which ER_i is a rule of the form:

$$ER = <TRIGGER,\ PICTURE>$$

TRIGGER has exactly the same form and meaning as in the *VALUE* case. Not all triggers need to appear in both sections. However, if they do, then both triggers fire at the same time. This is the case in particular for the default rules for single I/O-units, which appear in both sections.

The default rule for the *ECHO* part is:

$$S_i : P_i\ [O]$$

indicating that the output part of the I/O-unit is added to the new picture environment.

The *ECHO* section also contains one special rule which has the *$*-sign as trigger. The *PICTURE* part of the *$*-rule will contain an expression which produces the *O*-part of the I/O-unit. The *$*-sign is equivalent with the entire regular expression of *SYMBOL*.

4. Interaction described through I/O-units.

In this chapter a number of concepts which play an important role in interactive systems design will be described in terms of the I/O-model. The extent to which this can be done well justifies using such a model as a basis for implementation. Otherwise, the model would have a more explanatory use which we also consider of importance.

4.1. Interactive techniques.

An interactive technique consists of a set of manipulations of one or more physical input devices for producing a certain fixed type of result. Examples are:

- A certain way of selecting from a menu (which may involve stepping through a hierarchy or scrolling menu items).
- A method for interactively drawing polygons with a built in eraser.
- A method for picking a picture element form a given set, which may involve a highlight function to ease selecting the correct element.

Typically, there is a clear distinction between the visualizations during the use of the technique and visualization after a certain result has been confirmed by the operator.

An interactive technique can be represented as a transaction process. The I/O-unit provided by this transaction encapsulates the visualization and the abstract data representation of the confirmed result. The visualization during the use of the technique will appear as part of the environment of the corresponding active transaction process. This will automatically disappear when the transaction is deactivated.

Interaction techniques can be provided in the model as elementary techniques or as compound techniques using more elementary techniques.

4.2. Resource management.

The screen space and the physical devices can be considered as resources to be claimed by active transaction processes. The total screen space can, by subdividing or through an overlap strategy or through clipping, become a much larger virtual screen space. Similarly keyboards, mice, button boxes, etc, can through use of selectable icons, hot keys or various buttons become multiple keyboards, cursors, etc. The system can represent this by a resource structure and a resource management strategy.

The model uses this resource structure in two ways:

- The picture part of the environment can through its' attribute specification identify virtual input resources. Through the transformation it can specify a relative or an absolute portion of the screen space. A relative screen portion means relative to the inherited screen space.
  The inheritance mechanism provides the elementary transactions with resource attributes to be mapped on the physical devices. This mapping defines for each basic device where it is defined on the screen, how its' cursor operates, etc.
  The mapping is such that when the combination of resource attributes is unique *and* the mapping succeeds, then the visualization of the devices is visibly distinguishable from any other resource. In this way the user cannot get confused.
- Whenever an incarnation identifier has to be calculated (i.e. during activation of a transaction process), the algorithm guarantees that the resource attributes for two transactions producing the same I/O-unit (for a different consumer or for two different requests from the same consumer) will have different resources attributes. For this, it is essential that part of the resource assignment can be done dynamically. This can be expressed by saying that there is a one-to-one correspondence between incarnation identifier and resource identifiers.

4.3. User freedom.

User freedom is characterized by the amount of flexibility the user has on the order in which input is to be provided. Given a set of symbols to be provided, the maximal freedom would be to allow for all permutations of the set. The minimal amount of user freedom would be to allow only one permutation.

In the model the user freedom span can be calculated for each transaction process. This can be given as a fraction of the total freedom as follows: all sub-transactions activated in **request**-mode allow for only one occurrence. The other symbols can be permuted, albeit that if the same symbol occurs more than once their order is fixed; i.e. multiple occurrences count as one. Hence, a good indication of user freedom is the fraction of the sub-transactions activated in **sample** or **event** mode.

Note that user freedom is only restricted by synchronization with the SYMBOL expression through **request** mode. The SYMBOL expression also introduces some form of user freedom through parallel $\wedge$ and $\vee$ branches. This means that multiple requests occurring in parallel branches can also be permuted as far as the user reaction is concerned. This will lead to slight adjustments of the user freedom fraction calculation.

4.4. Error handling.

Associated with error handling are a number of system properties which will make errors easier to correct.

- The first property is the occurrence of unexpected events. In the model, I/O-units can only be produced *after* the corresponding transaction process has been activated. This is also true for the lowest level of transactions. Hence, no unexpected unit can occur. Also, for each produced I/O-unit, the consumer is known.
- The second property is unexpected forms. This means that a data item has been produced that does not fit the subsequent processing; i.e. based on its' form, no function for that form can be found. The model associates an abstract data type for each component of the I/O-unit. Both producer and consumer adhere to this data type. Hence, type errors do not occur.
- The third property is unexpected values of the otherwise proper type. These errors can indeed occur in the model. However, the environment and the testing conditions for the value, which are explicitly present in the *IMPORT* section of the corresponding *VALUE* rule, guarantee that such an error gets detected immediately. As a result error recovery / logging procedures have maximum information about the error available. This situation can well be exploited for sophisticated trace and recovery facilities.
  Such an error recovery process can generally be described as a generic transaction process, where the I/O-unit to be corrected is the parameter.

4.5. Direct manipulation.

Direct manipulation is an ergonomic ideal especially for naive users who have no training in using encoded messages. Direct manipulation uses a number of basic techniques which are directly applied to achieving a particular goal. Associated with direct manipulation is the fact that the basic techniques are all used at the same semantic level.

Given a basic technique, B, and an interactive manipulation function, M, both of which can be described in a transaction process, then we can say that: B directly manipulates M for a given effect, E, (i.e. a change of the environment of M) if the following two conditions are met :

- B is a sub-transaction of M; this means that the I/O-units produced by B go to M directly.
- M contains a VALUE and ECHO rule triggered by B, which causes E.

According to this definition one can try to achieve all manipulations of a given set $\{M_c\}$ directly.

5. Conclusions and future work.

In this paper a model has been introduced bringing together many concepts relevant for graphics interaction. This model has classified many difficulties inherent to interactive graphics system design.

The model has already been applied for providing the semantic framework for

designing and implementing a specification language for graphical dialogues (see [11]).

A comparison to GKS has been made, mainly to illustrate how far GKS-like systems are separated from truly supporting interactive programming.

In the near future the model will be extended to encompass concepts like parameterization of transaction processes (i.e generic transactions) and so-called conference transactions (i.e. a many-to-many producer / consumer relationship). Furthermore, all parts which have been introduced only informally as well as many details which have been left out will be incorporated in a more formal way. The ultimate goal is to produce an entirely formally specified complete model.

Appendix 1: The (incomplete) GKS output model.

<graphical_output> ::= <picture> <wsstate> <u_state>

<picture> ::= **listof** ()
 ::= <sheader> **listof** (<primatt>)
<sheader> ::= <name> <strafo> <shili> <svis> <sdetect>
<primatt> ::= <outprim> | <prattri>
<outprim> ::= <polyline> | <polymark> | <fill_area> | <text>
<prattri> ::= <indi_attr> | <bund_attr>

<wsstate> ::= <wstrafo> **listof** (<attrbundle>)

<u_state> ::= <deferral> <regenerate> **listof** (<u_delay>)
<u_delay> ::= | <wstrafo> | <attrbundle>

The application program has functions which can control:

- segment manipulation.
  new : × <graphical_output> → <graphical_output>
  del : × <graphical_output> → <graphical_output>
  new : <sheader> × <graphical_output> → <graphical_output>

- the workstation state.
  new : <wstrafo> × <graphical_output> → <graphical_output>
  new : <attrbundle> × <graphical_output> → <graphical_output>

- the update state.
  new : <deferral> × <graphical_output> → <graphical_output>
  new : <regenerate> × <graphical_output> → <graphical_output>
  update : <graphical_output> → <graphical_output>

Appendix 2: The (incomplete) GKS input model.

<graphical_input> ::= **listof** (<i_device>)

<i_device> ::= <i_state> <i_mode>
<i_state> ::= <i_value> **listof** (<i_attr>)
<i_value> ::= <point> | <float> | <enum> | <string> | | **listof** (<point>)
<i_attr> ::= <echo_type> <echo_area>
<echo_type> ::= <cursor> |
<echo_area> ::= <point> <point>

<i_mode> ::= <request> | <sample> | <event>

The application program has functions which can control:

- input device initialization.
  new : <i_mode> × <graphical_input> → <graphical_input>

- input device reading.
  read : <i_device> × <graphical_input> → <graphical_input>

References:

1. E. ANSON, The device model of interaction, *Computer graphics*, **16**(3), July 1982, pp. 107-114.
2. G. S. CARSON, The specification of computer graphics systems, *IEEE Computer Graphics and Applications*, (September 1983), pp. 27-41.
3. D. A. DUCE and E. V. C. FIELDING, Towards a formal specification of the GKS output primitives, in *European Computer Graphics Conference and Exhibition*, A. A. G. REQUICHA (ed.), North-Holland, Amsterdam, (1986), pp. 307-323.
4. R. GNATZ, An algebraic approach to the standardization and the certification of graphics software, *Computer Graphics Forum*, **2**(2/3), 1983, pp. 153-166.
5. F. R. A. HOPGOOD and D. A. DUCE, A production system approach to interactive program design, in *Methodology of interaction*, R. A. GUEDJ (ed.), North-Holland, Amsterdam, (1980).
6. ISO, Information processing system - Computer graphics - Graphical Kernel System (GKS) functional description, ISO 7942, ISO Central Secretariat, (August 1982).
7. W. R. MALLGREN, Formal specifications of graphic data types, *ACM Transactions on Programming Languages and Systems*, **4** , October 1982, pp. 687-710.
8. J. MATTHIJS, Recent experiences with input handling at PMA., in *User interface management systems*, G. E. PFAFF (ed.), Springer-Verlag, Heidelberg, (1983).
9. H. J. SCHOUTEN, *Radicals - a parallel interface to GKS*, CWI report (to appear), Amsterdam, (1987).
10. A. SHAW, On the specification of graphics command languages, in *Methodology of interaction*, R. A. GUEDJ (ed.), North-Holland, Amsterdam, (1980).
11. R. VAN LIERE and P. J. W. TEN HAGEN, *Introduction to dialogue cells*, CWI report (CS-R8703), Amsterdam, (1987).
12. J. VAN DEN BOS, M. J. PLASMEIJER and P. H. HARTEL, Input-Output tools: a language facility for interactive and real-time systems, *IEEE Transactions on Software Engineering*, **9** , 1983, pp. 247-259.