RAL-91-065

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-91-065

A Formal Specification of a Graphics System in the Framework of the Computer Graphics Reference Model

D Duce and F Paterno

September 1991

A FORMAL SPECIFICATION OF A GRAPHICS SYSTEM IN THE FRAMEWORK OF THE COMPUTER GRAPHICS REFERENCE MODEL

D.Duce [+], F.Paterno'[*]

([+]) Rutherford Appleton Laboratory, Chilton, Didcot, Oxon

([*]) CNUCE - C.N.R., Via S.Maria 36, 56100 Pisa, Italy

Abstract

The Graphical Kernel System, GKS, was published as an ISO/IEC standard in 1985 and is now undergoing revision. A Reference Model for Computer Graphics reached the status of Draft International Standard in August 1991. This paper explores the use of the Reference Model as the basis for the structure of a formal description of the framework of the first draft of the revised GKS, called GKS-R. The Reference Model combines process and data views of computer graphics, and for this reason, the formal description technique LOTOS has been used for the work described here. LOTOS combines an algebraic data type definition notation ACTONE with a process description technique based on process algebras.

1.Introduction

The Computer Graphics Reference Model(CGRM)[1] provides a conceptual framework for the development and the integration of graphics systems and their related systems and it is being developed as an international standard. It is defined as five environments hierarchically layered and with the same internal logical structure.

Its main goals are: verify and refine requirements for computer graphics; identify needs for computer graphics systems and external interfaces; compare computer graphics systems. It is intended to provide a consistent terminology for computer graphics systems as well as generic framework by which all future graphical systems and relationships among them should be described. There is also an interesting discussion in progress on how systems different from graphics systems, as window systems and image processing systems, are related to this framework.

Its proposal was born from the consideration that a set of graphics systems (GKS, GKS-3D, PHIGS, PHIGS PLUS) has been produced over a ten years period with common concepts but also with incompatibilities due to the different times at which they were produced, the different people involved, and the different application areas.

It was clearly going to be difficult to produce a Reference Model of the existing set of standards with clean concepts. The approach had to be to define a Reference Model having to filter the current concepts and to use this as the basis for the next generation of graphics systems.

The aim of this present work is to provide a formal specification for the framework of GKS-R structured in the same way as the CGRM. The motivation for the work was a desire to explore the structure of the CGRM and the applicability of this particular structuring to the description of a graphics system, in a formal way.

The first ISO standard for computer graphics, the Graphical Kernel System(GKS) was published in August 1985 and is now undergoing revision. The GKS-R system considered in this paper is the result of a meeting of the GKS Rapporteur Group held in Leeds in July 1990 which has already undergone one further round of revision in February 1991, leading to a document with Committee

Draft Status. The document has to undergo much more processing before it will became an International Standard. The reason for choosing GKS-R for this exercise is that it is based on a small number of well-defined concepts and has a well-defined internal architecture that differs significantly from the current GKS standard. The GKS revision work is being carried out in the light of the insight gained during the development of the CGRM and hence is an appropriate system in which to explore the structuring used by the CGRM. A conscious effort has been made by the GKS Rapporteur Group to adhere to the CGRM. The GKS Revision activity is described in more detail in [3].

The specification is given in LOTOS[4]. LOTOS is a formal description technique, standardized by ISO/IEC. LOTOS was originally developed for the formal description of communications protocols. It contains a notation for describing communicating processes, which is based on Milner's Calculus of Communicating Systems (CCS)[5], with an algebraic notation for defining datatypes called ACTONE[6].

The CGRM combines process and datatype views of computer graphics, and the fact that LOTOS provides both process and data types descriptions was the main reason for choosing LOTOS for this work. There was also a desire to see how well an International Standard, developed within one domain of application in mind, could be applied to a different domain.

The formal definition of a general framework for graphics systems has been addressed already[7] providing a configurable model graphics system as general framework and using the Z specification language. In this approach a variety of different configurations of specialized functional units, called graphics modules, can be constructed with a range of differing functionality. A graphics system is built up from a selection of graphics modules, organized into conceptually related processing classes, termed processing strands. These form a framework on which a number of designated configurations of graphics modules, termed processing pipelines can be assembled.

Also the outline of the GKS-R functionality has been already formally specified[8] using LOTOS. In the earlier work the kernel and each workstation were associated with one specific process while, to follow the CGRM framework, a deeper decomposition into processes is needed.

LOTOS has been shown to be a useful specification language for graphics systems because it has concurrent constructs needed to describe the cooperation among output and input functionality and the possible input modes and it also allows algebraic data types specifications for defining the graphics data types exchanged among the graphics component and their processing.

The paper concludes with an example of a frequently used graphical interaction described using the specification developed here, to show that it can be used to describe a wide spectrum of interactive graphic applications.

2. The Computer Graphics Reference Model

The CGRM consists of five environments representing five different levels of abstraction in a graphics system. The environments all have the same internal architecture composed of five processes and four data structure. The functions of the environments are:

Construction environment: the application data to be displayed is prepared as a model from which specific graphics scenes can be produced. Input tokens in the instruction store are constructed in the precise form utilized by the application.

Virtual environment: a set of virtual output primitives is defined in a completely geometrically defined way. Input tokens are defined in a coordinate system used in the virtual environment.

Viewing environment: a specific view of the scene is taken, projecting the picture to be presented.

Input tokens in the selected store are elevated to the virtual environment.
Logical environment: the picture is rendered as an image ready for presentation: each graphical output primitive is associated with a complete set of properties, in a device-independent way. Input tokens are converted to device independent form with properties added to differentiate the origin of the input if required.
Physical environment: the image is presented as a display for output to a specific device. All properties associated with the physical input devices used will be known at this stage.

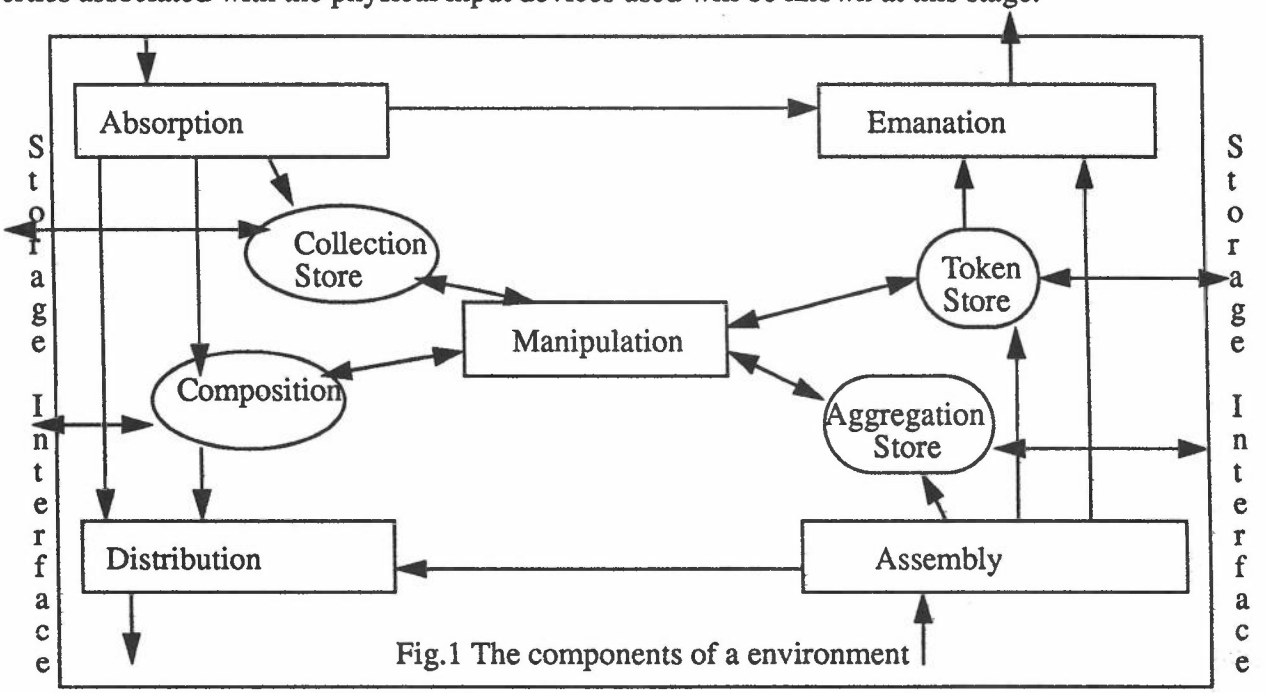

Fig.1 The components of a environment

One environment is described as five processes and four data structures(fig.1).
The processes are:
absorption: accepts output data from the next higher environment and applies the geometric and other transformations necessary to produce the data in the form appropriate to its environment. It is called *production* in the virtual environment and *rendering* in the logical environment.
manipulation: processes data from the collection store and the aggregation store, it adds data from these two components to the others and it provides linkages between input and output within a specific environment.
distribution: takes data from the current environment and passes it to the next lower environment, no transformations, geometric or otherwise, are applied to the data as it is distributed.
assembly: is the process which accepts input data from the next lower environment and processes it for inclusion in the aggregation store or token store of the current environment.
emanation: it takes input control data passed directly from the assembly process or input tokens from the token store of the current environment and passes it to the next higher environment. It is called *generation* in the virtual environment and *abstraction* in the logical environment.
The data elements are:
composition: the conceptual accumulation of the graphical output existing at any time within an environment. It is called *scene* in the logical environment and *image* in the logical environment.
collection store: a set of named and structured set of output data intended for use within an environment.

token store: it contains input tokens ready for emanation to the next higher environment. It is called *directive store* in the virtual environment and *information store* in the logical environment.
aggregation store: a set of named and structured set of input data intended for use within an environment.

3. The GKS-R case study

The key component of GKS-R is the NDC picture, which consists of a sequence of output primitives. In this specification, only the polyline primitive is considered. It suffices to represent its geometry by a list of points (the vertices of the primitive) and its attributes by the nameset attribute bound to it on creation from the GKS state list. The GKS state list consists of the current values of the GKS state, nameset attribute, list of normalization transformations (presumed ordered by viewport input priority) and associated list of normalization transformation numbers and the currently selected normalization transformation.
The NDC picture may be displayed on workstations. Multiple workstations can be in operation together. As the NDC picture is updated, the displays on the open workstations will be updated at the same time. A mechanism is provided to suspend this updating of visual effects for a particular workstation. For this specification, a workstation is represented by the workstation state list and physical picture.
The physical picture is represented as a sequence of physical primitives. A physical primitive is represented by a list of points and the values of the operator attributes, highlighting and detectability. The subsequence of the NDC picture to be displayed on a particular workstation and the values of the operator attributes are determined by selection criteria. The workstation state list consists of separate selection criteria for visibility, highlighting and detectability, the current workstation transformation and a flag used by REQUEST mode input. Sequences of output primitives can be defined as a picture part and retained in a picture part store. A picture part is thus a sequence of output primitives.
In the current GKS-R document, GKS segments are supported using the nameset mechanism and picture parts. A route direction has also been included which directs output primitives either to the NDC picture or through a by-pass channel directly to a backdrop on each workstation.
These features are not discussed in this paper.
The GKS-R operations modelled in the specification are:
OPEN GKS
CLOSE GKS

CREATE OUTPUT PRIMITIVE

SET PRIMITIVE ATTRIBUTE (nameset only)
SET NORMALIZATION TRANSFORMATION

DELETE PRIMITIVES

BEGIN PICTURE PART
END PICTURE PART
BEGIN PICTURE PART AGAIN
APPEND PICTURE PART
COPY PICTURE PART FROM PICTURE PART STORE

SET INPUT MODE
REQUEST LOCATOR
REQUEST PICK
SAMPLE LOCATOR
SAMPLE PICK
AWAIT EVENT

OPEN WORKSTATION
CLOSE WORKSTATION
SET WORKSTATION SELECTION CRITERION
SET WORKSTATION TRANSFORMATION

GKS-R combines the input functions for the different classes of input device into single functions for each operating mode. Here we have found it convenient to specify separate functions for each of the two device classes described.
GKS is a 2-dimensional system, this implies the viewing environment can be ignored because it performs only the identity transformation. The construction environment is also null. In each environments state information is distributed between the component processes. The allocation of GKS-R fuctionality to environments follows the description of the GKS functionality in the terms given in annex A of the CGRM document. In GKS the production of primitives in NDC space with attributes bound corresponds to the virtual environment. GKS workstations correspond to the logical and physical environments. Realization of bundled aspects is done in the logical environment during rendering. In GKS those aspects that are definitely geometric are bound at the virtual (NDC) environment. The individual/bundled model fits into the CGRM as long as the complete geometry is specified at the NDC level. The event queue in GKS corresponds to a token store in the virtual environment. Transformation of locator and stroke input values from NDC to WC coordinates is performed by an emanation in the virtual environment. Echoing occurs when the appropriate output primitives are created in the image.
GKS can open multiple workstations, this means that from a CGRM point of view there is a fan-out with one virtual environment and as many logical and physical environments as the number of open workstations. In this specification we consider a simplified case with a kernel (a virtual environment) and a workstation (a logical environment) statically allocated, with the workstation associated with two input devices (locator and pick) that can be reconfigured at any time in one out of the three possible input modes, and one output device.

LOTOS supports a multiway rendezvous protocol that is different with respect to the conventional mechanism found in languages such as ADA, OCCAM, ECSP: more than just two processes can be involved, and are not fixed, but they can vary dynamically during the evolution of the system depending on the processes activations realized (while the channels are static that means for a given set of allocated processes it is not possible to modify the processes synchronizing on each channel). The value passing on a gate can be realized along different directions and it is possible to transmit on the same gate, at different times, different data types. When the communication between processes has to be modelled, and values of basic data types have to be sent, there are two possible approaches:

- to use simple events where the value passed belongs to a structured data type (the associated commands are: process sender: ...g1!x; .../process receiver: ...g1?y:complex_type; ...)

- to use structured events, allowed by LOTOS, composing in the communication simpler data type

(the associated commands are: process sender: ...g1!x1 !x2 ... !xn; .../ process receiver: ... g1?y1:simple1_type ?y2:simple2_type ...?yn: simplen_type; ....).
We have chosen the second approach because otherwise the specification gets more complex since more data types are added only to compose smaller data types in order to specify the communication, and then we need more primitives to insert in their definition different operations to decompose them again by inquiring the value of the components of the complex data types There are two possible approaches to describe CGRM data types in LOTOS: associating with LOTOS processes whose only function is managing the access to it (we call this the CGRM data-LOTOS process approach) or as parameters of one CGRM process (we call this the CGRM data-CGRM process approach). Next two sections illustrate the results of both the approaches in the case of GKS-R for both the involved environments (virtual and logical).

4. The CGRM data-LOTOS process approach

The main GKS-R functionality has been mapped onto the CGRM in this way:
Virtual environment: normalization transformation (production) and inverse (manipulation/generation); input queue (directive store); NDC picture (scene); picture part store (collection).
Logical environment: workstation transformation (rendering) and inverse (abstraction); DC picture (image); mapping logical attributes (rendering): echo (from aggregation to image, through the workstation manipulation, for the locator; through kernel manipulation and scene, for the pick device); trigger and measure (aggregation and assembly).
You can notice in figure 2 that the workstation collection does not receive any data from the other components of the graphics system. This stresses the consideration that in the present design of the graphics system there are some features that are beyond the control of the graphic application programmer and are implementation dependent.

4.1 The realization of input primitives in the CGRM

This seems to be one of the more complicated problems. We have chosen to describe pick and locator input classes because they are most involved in the manipulation of graphics representations. The physical devices corresponding to the pick and locator logical input devices each provide raw data which define a position in device coordinates(DC).
The flow of primitives in the model depends on the operating mode of the input device. In REQUEST mode, the production receives the input primitive from the application requesting input from a particular device. The primitive is then passed as control information through the distribution and rendering to the abstraction. The abstraction then requests the current measure value of the device from the aggregation store. The current measure value is returned when the corresponding trigger fires. The current measure value delivered to the abstraction is then returned to the application after processing by the manipulation and generation.
SAMPLE mode input is similar to REQUEST mode input except that the aggregation store does not wait for a trigger to fire before returning the current measure value to the abstraction.
In EVENT mode, the AWAIT EVENT request is passed from the application to the production and then to the generation. The generation removes the first event in the input queue (directive store) and return this to the application. Events are added to the input queue by the abstraction, through the manipulation.
The workstation manipulation applies the echo function to the locator input values received from the aggregation, it uses predefined shapes received from the collection to build the echo and it returns the results to the image.

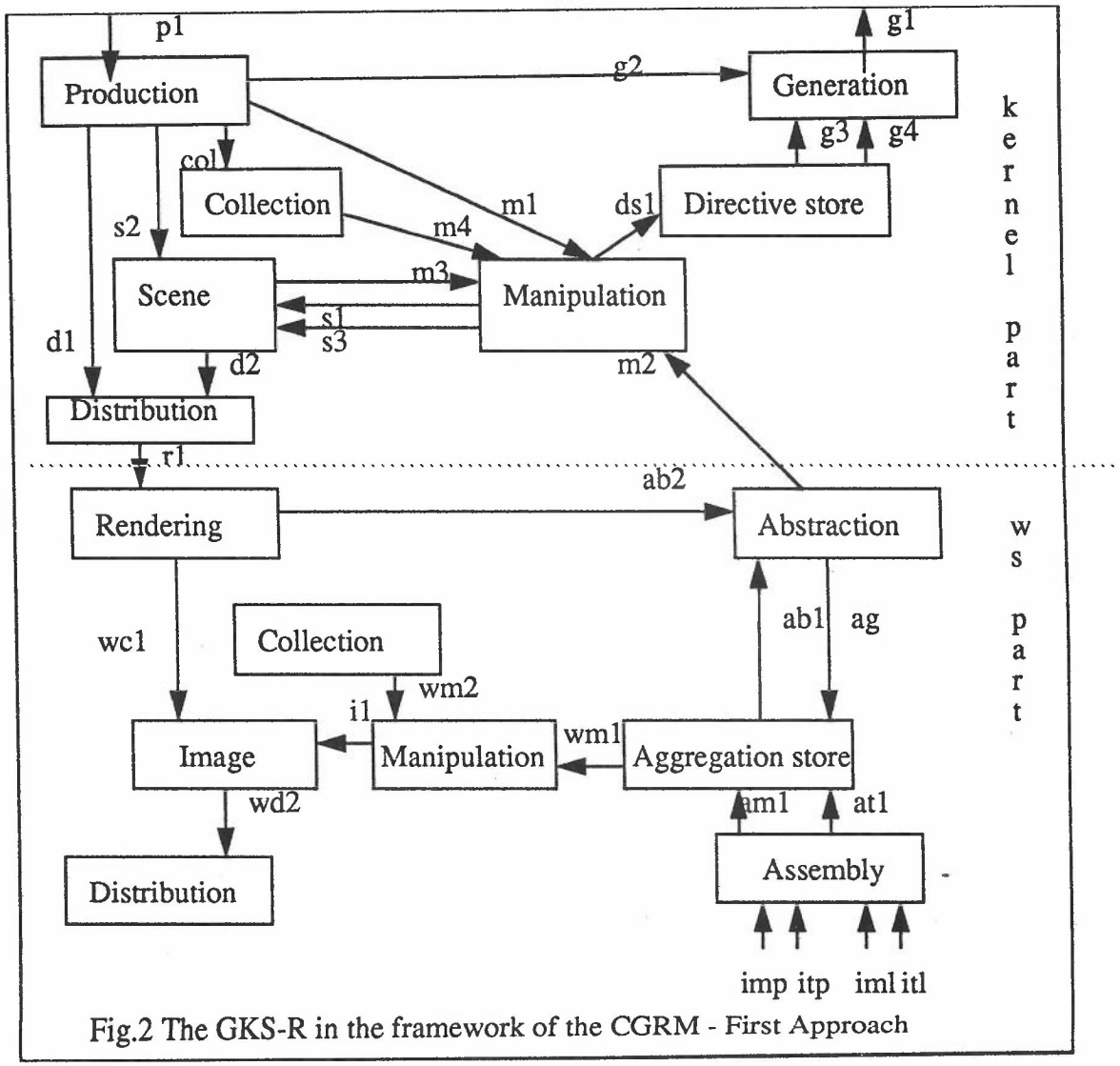

Fig.2 The GKS-R in the framework of the CGRM - First Approach

When the assembly receives new measure input or trigger values it returns them to the aggregation which in turn passes the locator measure values to the workstation manipulation and the pick value to the higher levels for echoing. When a trigger fires, if the device is in EVENT mode, the current measure value is sent to the kernel manipulation, if it is in REQUEST mode and a request from the application is pending, the current measure value is sent to the kernel manipulation and the pending request is cleared. The abstraction applies the inverse workstation transformation to input data and passes them to the kernel manipulation.

For locator devices, the token emanating from the abstraction is passed to the kernel manipulation, which determines which inverse normalization transformation should be applied to the token in order to generate the appropriate logical input value for the application program. The current values of the normalization transformation are held by the production. These values are communicated to the manipulation when required by the manipulation. The manipulation selects the appropriate transformation and creates a token in the directive store which consists of the locator position in NDC coordinates and the transformation to be applied. The transformation is actually applied by the generation. This mechanism is necessary in order to comply with the constraints on coordinate systems and the actions of each process in the environment, imposed by the CGRM.

Process	Parameters	Input data types
Production	KernelStateList	Gks_primitives
Kernel collection	PicturePartStore, Partname	Gks_primitives
Scene	Ndcpicture	Ws_primitives, PicturePart, Ndcpoint
Directive	Queue	Input_kvalue, Input_mode
Generation		Input_kvalue
Manipulation		PicturePart, Point, Input_mode, Input_class, Nameset, List_trasfcord, List_Int
Distribution		Ws_primitives, Ndc_primitives
Rendering	WsStateList	Ws_primitives
Image	Dcpicture	Dc_primitives, Dcpicture
Assembly		Bool, Point
Aggregation	Point, Bool, Input_mode	Point, Input_class, Input_prim
Abstraction	Trasfcord	Input_mode, Input_class, Ws_primitives, Point
Manipulation	Template	Template, Point

Table 1: Summary of Parameters and Data Types

For pick devices, the manipulation selects the output primitive in the scene which is 'closest' to the NDC position. The value of the nameset associated with this primitive is returned to the directive store. The specification, like GKS-R itself, does not give a precise definition of how the primitive picked is selected. The directive store contains the input queue used by devices in EVENT mode. For devices in REQUEST or SAMPLE mode, tokens are passed immediately to the generation. The parameters and data types for each process are summarized in table 1.

4.2 The virtual environment

The kernel is described by seven processes, where four have a status and three realize mainly routing of data, after processing them, to other processes.

The production applies the normalization transformation to the output primitives and associates the current name set by the wctondc function. It also manages most of the kernel state list (some data are managed by other processes, for example the current picture part open is in the kernel collection). It receives commands from the application on the p1 channel. Most of the kernel control commands will modify its status, the workstation control commands are sent to the distribution(d1 channel), the output commands are sent to the collection if a picture part is open otherwise to the scene. The input instructions are sent to the distribution except for the await event command that is sent to the generation. After receiving an input instruction it has always to syncronize with the generation waiting until the latter sends the input result to the application. This is needed to prevent new instructions being received before the previous input instruction has been completely executed to avoid inconsistencies. In Lotos two or more processes synchronize when all are offering the same value on the same gate. It can also synchronize with the manipulation when the list of normalization transformations and the list of associated numbers are requested.

specification CGRM_GKS_R_subset : noexit

library

BOOLEAN, ELEMENT, SET, NATURALNUMBER, INTEGER, LIST_INTEGER, REAL, STRING

endlib

The following expression is the parallel composition of the thirteen processes, with the initial values of their parameters, which describe the GKS-R system. The workstation identifier of the workstation instance is wsx.

behaviour

((((((((((production(mk_krn_list(ppcl, empyNS, emptylist, emptylist, identity))|[col]| kernel_collection(emptyPPS, null))|[s2]| scene(emptyNDCP))|[s1, s3, m3, m4, p1, m1] | manipulation)|[ds1]| directive(emptyqueue))|[g1, g2, g3, g4]|generation)|[d1, d2]|distribution)|[r1]| rendering (initial_wsStateLs (wsx))|[wc1]| image (emptydcp))|[wm2, i1]| manipulation (arrow))|[wm1]|aggregation(0, 0, 0, 0, false, false, request, request))|[ab1, ab2, ag, m2]|abstraction(identity))|[am1, at1]|assembly)|[wm2]|ws_collection)

where

```
process production[p1, d1, s2, col, m1, g2](k:KernelStateList) :noexit :=
(p1?pr:Gks_prim;
    ([inq(pr) eq opengks] -> production(k)
    []
    [inq(pr) eq closegks] -> exit
    []
    [inq(pr) eq openws] -> d1!pr; production(opws(k, pr))
    []
    [inq(pr) eq closews] -> d1!pr; production(clws(k, pr))
    []
    [inq(pr) eq outputprim] -> ([inq_state(k) eq ppcl] -> s2!wctondc(pr, k); production(k)
                                []
                                [inq_state(k) eq ppop] -> col!wctondc(pr, k); production(k))
    []
    [inq(pr) eq deleteprimitive] -> s2!pr; production(k)
    []
    [inq(pr) eq selectnormalizationtransformation] -> production(sel_tn(k, pr))
    []
    [inq(pr) eq setnamesetattribute] -> production(set_ns(k, pr))
    []
    [inq(pr) eq setworkstationtransformation] -> d1!pr; production(k)
    []
    [inq(pr) eq setvisualeffect] -> d1!pr; production(k)
    []
    [inq(pr) eq setworkstationcriteria] -> d1!pr; production(k)
    []
    [inq(pr) eq beginpicturepart] -> col!pr; production(bpp(k))
    []
    [inq(pr) eq beginpicturepartagain] -> col!pr; production(bpp(k))
    []
    [inq(pr) eq closepicturepart] -> col!pr; production(cpp(k))
    []
    [inq(pr) eq appendpicturepart] -> col!pr; production(k)
    []
    [inq(pr) eq copypicturepartfrompps] -> col!pr; production(k)
    []
    [inq(pr) eq setinputmode] -> d1!pr; production(k)
    []
    [inq(pr) eq requestlocator] -> d1!pr; g2!resin; production(k)
    []
    [inq(pr) eq requestpick] -> d1!pr; g2!resin; production(k)
    []
    [inq(pr) eq samplelocator] -> d1!pr; g2!resin; production(k)
    []
    [inq(pr) eq samplepick] -> d1!pr; g2!resin; production(k)
    []
    [inq(pr) eq awaitinput] -> g2!awaitinput; g2!resin; production(k))
[]m1!inq_trn(k) !inq_lntrns(k); production(k))
endproc
```

The collection manages the picture part store and has also, as parameter, the name of the currently open picture part. When it receives a copypicturepartfrompps instruction it sends the related instructions to the manipulation.

```
process kernel_collection[col, m4](pps: PPS, pn: Partname) :noexit :=
col?pr:Ws_prim;
   ([inq(pr) eq outputprim] -> collection(addPP(pn, pr, pps), pn)
   []][inq(pr) eq beginpicturepart] -> collection(begpicpart(pps, pr), getpn(pr))
   []][inq(pr) eq appendpicturepart] -> collection(append(pps, pr), pn)
   []][inq(pr) eq copypicturepartfrompps] -> m4!cpfpps(pps, pr); collection(pps, pn)
   []][inq(pr) eq beginpicturepartagain] -> collection(pps, getpn(pr))
   []][inq(pr) eq closepicturepart] -> collection(pps, null))
endproc
```

The scene manages the NDC picture. It receives single output and control output primitives from the production or a picture part from the manipulation. The new NDC picture is sent to the workstation for display. The scene can also receive an NDC position from the manipulation. The nameset of the output primitive 'closest' to this position is returned to the manipulation. Echoing of the pick input device takes the form of highlighting the primitive selected. This is achieved by adding the name hl to the nameset of the primitive selected; the workstation selection criterion for highlighting is set such that primitives containing the name hl will be highlighted. The name hl will be deleted from the nameset of the other primitives in the scene, thus cancelling the echo of any previously picked primitives.

```
process scene[d2, m3, s1, s2, s3](ndcp: Ndcpicture) :noexit :=
(s2?pr:Ws_prim;
          ([inq(pr) eq outputprim] -> d2!pr; scene(addndc(pr, ndcp))
          []][inq(pr) eq deleteprimitive] -> d2!del(getsel(pr), ndcp); scene(del(getsel(pr), ndcp)))
[]s3?pp:PicturePart; d2!addpp(pp, ndcp); scene(addpp(pp, ndcp))
[]s1?p:Ndcpoint ?s:Select; m3!get_ns(detect(p, ndcp));
            [det(get_ns(detect(p, ndcp)), s)] -> d2!eco(rmec(ndcp), p); scene(eco(rmec(ndcp), p))
            [not(det(get_ns(detect(p, ndcp)), s)] -> scene(ndcp))
endproc
```

The directive manages the input queue. In our example we consider only locator and pick values. In both cases it receives them from the kernel manipulation. For locator input it receives also the associated inverse transformation, chosen from the normalization transformations at the moment at which the point is received. For pick input, the pick value consists of a nameset. These values will pass through the generation that will apply the inverse normalization transformation to locator input. This process also receives from the manipulation the locator and pick input values for devices in REQUEST or SAMPLE mode that have to be passed to the generation. When the application asks for an event from the input queue it is communicated by the generation.

```
process directive[g3, g4, ds1](q: Queue):noexit :=
(g4!top(q); directive(remove(q))
[] ds1?id:In_kvalue ?md:In_mode;
              ([md eq event] -> directive(add(id, q))
              [][(md eq sample) or (md eq request)] -> g3!id; directive(q))
)endproc
```

The generation receives from the directive store the input values, on g3 for SAMPLE and EVENT mode and on g4 for EVENT mode, to pass to the upper levels. For locator input it receives the point in normalized coordinates and the associated inverse normalization transformation and applies the transformation, delivering a position in world coordinates and the associated normalization transformation number.
It synchronizes with the production after the application requests an input value from the queue and then it waits for a value from the directive.

```
process generation[g1, g2, g3, g4] :noexit :=
(g3?im:In_kvalue; ([inq_cl(im) eq locator] ->g1!trasf_n(im); g2!resin; generation
                   [][inq_cl(im) eq pick] -> g1!im; g2!resin; generation)
[]g2!awaitinput; g4?im:In_kvalue; ([inq_cl(im) eq locator] ->g1!trasf_n(im); g2!resin; generation
                                   [][inq_cl(im) eq pick] -> g1!im; g2!resin; generation)
) endproc
```

The manipulation serves two purposes: transmission of picture parts from the picture part store to the scene, generation of logical input values from data provided by the abstraction. For locator input, the manipulation selects which normalization transformation is to be used to transform the NDC coordinate point defined by the abstraction to a point in WC required by the application. The current values of the normalization transformation are held in the kernel state list in the production. The manipulation selects the normalization transformation whose viewport contains the NDC point. In the case that the viewport of more than one transformation contains the point, the transformation selected is that with highest viewport input priority. The CGRM requires that the generation applies the inverse normalization transformation and so the manipulation delivers a token consisting of the NDC point, the inverse normalization transformation that has to be applied and the number of the normalization transformation concerned. It is necessary to store the normalization transformation itself in the input token, rather than a pointer to the transformation (i.e. the transformation number), because it is possible for the application to change the normalization transformation between selection by the manipulation and application by the generation.
For pick input, the manipulation inquires from the scene the nameset of the output primitive closest to the point delivered by the abstraction. The scene echoes, by highlighting, the primitive selected.

```
process manipulation[m4, m3, s1, s3, m1, m2, ds1] :noexit :=
(m4?pp:PicturePart; s3!pp; manipulation
[]m2?im: Point ?md: In_mode ?cl: Input_class;
    ([cl eq pick] -> m2?s:Select; s1!im !s; m3?ns: Nameset; ds1!mk_pick_kvalue(ns) !md; manip-
     ulation
     [][cl eq locator] -> m1?tn:List_Trasfcord ?ln: List_Int; ds1!mk_loc_kvalue(im, get_tr
     (inp_trasf(tn, ln, im)), get_in(inp_trasf(tn, ln, im))) !md; manipulation)
) endproc
```

The distribution transmits data to the lower level.

```
process distribution[d1, d2, r1] :noexit :=
(d1?pr1:Ws_prim; r1!pr1; distribution
[] d2?pr2:Ndc_primitives; r1!pr2; distribution
) endproc
```

4.3 The logical environment

The rendering realizes the workstation transformation and manages the workstation state list. It receives output, control and input primitives. The latter are switched to the abstraction. Commands to update the image only have an effect if the visual effects state is ALLOWED (represented here by the value true).

```
process rendering[r1, wc1, ab2](w: WsStateList) :noexit :=
(r1?pr: Ws_prim;
   ([inq(pr) eq outputprim] ->
         ([sel(get_ns(pr), inq_wcri(w, visib)) and inq_ve(w)] -> wc1!wstrasf(pr, w); rendering(w)
         []][not(sel(get_ns(pr)), inq_wcri(w, visib)) or not(inq_ve(w))] -> rendering(w))
   [][inq(pr) eq ndcpicture] ->([inq_ve(w)] -> wc1!ndcptodcp(pr, w); rendering(w)
                                []][not(inq_ve(w))] -> rendering(w))
   [][inq(pr) eq openworkstation] -> rendering(initial_wsStateLs(wsx))
   [][inq(pr) eq closeworkstation] -> exit
   [][inq(pr) eq setworkstationcriteria] -> rendering(setcri(w, pr))
   [][inq(pr) eq setworkstationtransformation] -> ab2!pr; rendering(set_wstr(w, pr))
   [][inq(pr) eq setvisualeffect] -> rendering(ve(w, pr))
   [][inq(pr) eq setinputmode] -> ab2!pr; rendering(w)
   [][inq(pr) eq requestlocator] -> ab2!pr; rendering(w)
   [][inq(pr) eq requestpick] -> ab2!pr; rendering(w)
   [][inq(pr) eq samplelocator] -> ab2!pr; rendering(w)
   [][inq(pr) eq samplepick] -> ab2!pr; rendering(w))
[]ab2!inq_wcri(w, detect); rendering(w));
endproc
```

The image is involved to update the DC picture and it sends to the distribution all the DC picture updates and the echo for the measure value. When it receives a new instance of the locator echo, the previous one is removed before adding the new one.

```
process image[wc1, i1, wd2] (dcp: Dcpicture) :noexit :=
(wc1?prw: Dc_primitives; wd2!prw; ([inq(prw) eq outputprim] -> image(adddcp(prw, dcp))
                                  []][inq(prw) eq dcpicture] -> image(adddcpic(prw, dcp)))
[]i1?im: Dcpicture; wd2!adddcpic(im, rem_cursor(dcp)); image(adddcpic(im, rem_cursor(dcp))))
endproc
```

The assembly process receives the input for the trigger and the measure from the input devices (for pick and locator devices it is a point), and then passes these values (for the trigger only the indication of the device where it has been satisfied) to the aggregation store where the logical processing is realized.

```
process assembly[itl, iml, itp, imp, am1, at1] :noexit :=
(itl?it:Bool; at1!locator; assembly
[]iml?im1:Point; am1!im1 !locator; assembly
[]itp?it:Bool;  at1!pick; assembly
[]imp?im2:Point; am1!im2 !pick; assembly)
endproc
```

The aggregation process receives from the asssembly new measure values and trigger signals. From the application through the rendering and the abstraction it receives input commands. Its status consists, for each device, of the current measure value, the input mode and a boolean indicating if an input in REQUEST mode is pending. When a new pick measure is received, it is sent to the kernel level where the echo is realized. The token transmitted includes a mode field, which can take the values REQUEST, SAMPLE, EVENT and NEW. NEW means that the value is related only to a change of the current measure and not to an instruction or a trigger event. While the new locator measures are sent to the image, through the manipulation, for echoing.

```
process aggregation[ab1, ag, wm1, at1, am1](iml,imp: Point, pndl,pndb: Bool, lm,pm: In_mode)
:noexit :=
(am1?im:Point ?cl:Input_class;
      ([cl eq locator] -> wm1!im; aggregation(im, imp, pndl, pndp, lm, pm)
      []][cl eq pick] -> ab1!im !new !pick; aggregation(iml, im, pndl, pndp, lm, pm))
[]at1?it:Input_class;
  ([it eq locator] ->
      ([lm eq event] -> ab1!iml !lm !it; aggregation(iml, imp, pndl, pndp, lm, pm)
      []][pndl eq true] -> ab1!iml !lm !it; aggregation(iml, imp, false, pndp, lm, pm)
      []][(lm eq sample) or ((lm eq request) and (pndl eq false))] -> aggregation(iml, imp, pndl,
      pndp, lm, pm))
   []][it eq pick] ->
       ([pm eq event] -> ab1!imp !pm !it; aggregation(iml, imp, pndl, pndp, lm, pm)
       []][pndp eq true] -> ab1!imp !pm !it; aggregation(iml, imp, pndl, false, lm, pm)
       []][(pm eq sample) or ((pm eq request) and (pndp eq false))] -> aggregation(iml, imp, pndl,
      pndp, lm, pm)))
[]ag?prw:Input_prim;
      ([inq(prw) eq requestlocator] -> aggregation(iml, imp, true, pndp, lm, pm)
      []][inq(prw) eq requestpick] -> aggregation(iml, imp, pndl, true, lm, pm)
      []][inq(prw) eq samplelocator] -> ab1!iml; aggregation(iml, imp, pndl, pndp, lm, pm)
      []][inq(prw) eq samplepick] -> ab1!imp; aggregation(iml, imp, pndl, pndp, lm, pm)
      []][inq(prw) eq setinputmode] ->
                  ([inqd(prw) eq locator]->aggregation(iml, imp, pndl, pndp, inqm(prw), pm)
                  []][inqd(prw) eq pick]-> aggregation(iml, imp, pndl, pndp, lm, inqm(prw))))
) endproc
```

The abstraction receives requests for input for devices in REQUEST and SAMPLE mode, from the application through the production, distribution and rendering. The abstraction communicates the requests to the aggregation. The tokens generated are then communicated to the kernel manipulation. The abstraction applies the inverse of the current workstation transformation to positional data, to generate corresponding points in NDC space. Whenever the workstation transformation is changed, the new value is communicated to the abstraction by the rendering, so that the state of the abstraction mantains the current value of the workstation transformation.

```
process abstraction[ab1, ab2, ag, m2](tn:Trasfcord) :noexit :=
(ab1?im:Point ?md:In_mode ?cl:Input_class; m2!trasf(im, tn) !md !cl; abstraction(tn)
[] ab2?prw:Ws_prim;
      ([inq(prw) eq requestlocator] -> ag!prw; ab1?im:Point; m2!trasf(im, tn) !request !locator;
                                    abstraction(tn)
```

```
    [][inq(prw) eq requestpick] -> ag!prw; ab1?im:Point; ab2?s:Select; m2!trasf(im, tn) !request
                                   !pick; m2!s; abstraction(tn)
    [][inq(prw) eq samplelocator] -> ag!prw; ab1?im:Point; m2!trasf(im, tn) !sample !locator;
                                   abstraction(tn)
    [][inq(prw) eq samplepick] -> ag!prw; ab1?im:Point; ab2?s:Select; m2!trasf(im, tn) !sample
                                   !pick; m2!s; abstraction(tn)
    [][inq(prw) eq setinputmode] -> ag!prw; abstraction(tn)
    [][inq(prw) eq setworkstationtransformation] -> abstraction(inv_trasf(gettr(pr))))
)endproc
```

The manipulation, at the workstation level, is responsible for realizing the echo functionality for the locator device. The manipulation holds one template echo, received from the collection, which may be instantiated at a particular position to produce an echo of the cursor position. The result is incorporated into the image.

```
process manipulation [i1, wm2, wm1] (cur: Template) :noexit :=
(wm2?z:Template; manipulation (z)
[]wm1?im: Point; i1!mk_cursor(im, cur); manipulation(cur))
endproc
```

5. The CGRM data-CGRM process approach

In this approach collection, composition, aggregation store and token store are treated as parameters of the CGRM processes. We use the same data types that have been defined for the first approach and are described in the appendix. The main problem is to decide the association of CGRM data with CGRM processes. The first solution investigated was to put all the data structures into the manipulation. This decreases the number of processes to define but causes an overloading of functionality in the manipulation. The result is a manipulation specification that is not easily readable and an unbalanced distribution of processing among the components. For these reasons the approach chosen associates the directive store with the generation. In this way it is possible to simplify the manipulation specification and obtain a more natural description because the directive store has then the function of mantaining data until the generation receives a request from the higher levels to interrogate the store, an operation that can be immediately executed if the data are stored locally. Consideration was also given to associating the composition with the distribution, but this was rejected because of the tight coupling between the composition and the manipulation in the virtual environment. Echoing of input devices and pick input require close coupling.

5.1 The virtual environment

When the production receives output primitives it delivers them to the manipulation with the kernel status to provide information whether to add them to the NDC picture or to the collection.The m1 gate is used to pass the list of normalization transformations to the manipulation when locator input has to be processed.

```
process production[p1, d1, m1, g2](k:KernelStateList) :noexit :=
(p1?pr:Gks_prim;
   ([inq(pr) eq opengks] -> production(k)
   [][inq(pr) eq closegks] -> exit
   [][inq(pr) eq openws] -> d1!pr; production(opws(k,pr))
```

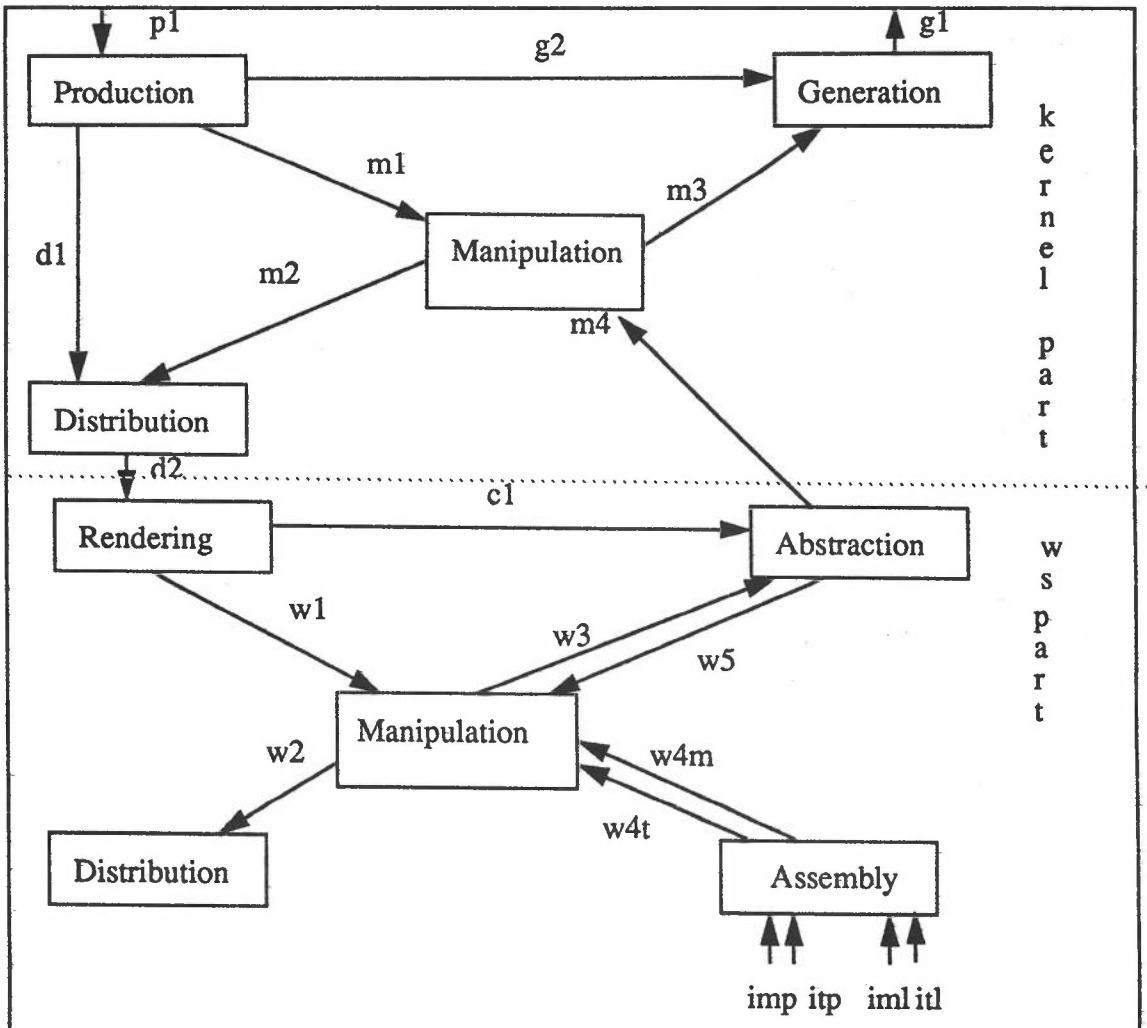

Fig.3 The GKS-R in the framework of the CGRM - Second Approach

[][inq(pr) eq closews] -> d1!pr; production(clws(k, pr))
[][inq(pr) eq outputprim] -> m1!wctondc(pr, k); m1!inq_state(k); production(k)
[][inq(pr) eq deleteprimitive] -> m1!pr; production(k)
[][inq(pr) eq selectnormalizationtransformation] -> production(sel_tn(k, pr))
[][inq(pr) eq setnamesetattribute] -> production(set_ns(k, pr))
[][inq(pr) eq setworkstationtransformation] -> d1!pr; production(k)
[][inq(pr) eq setvisualeffect] -> d1!pr; production(k)
[][inq(pr) eq setworkstationcriteria] -> d1!pr; production(k)
[][inq(pr) eq beginpicturepart] -> m1!pr; production(bpp(k))
[][inq(pr) eq beginpicturepartagain] -> m1!pr; production(bpp(k))
[][inq(pr) eq closepicturepart] -> m1!pr; production(cpp(k))
[][inq(pr) eq appendpicturepart] -> m1!pr; production(k)
[][inq(pr) eq copypicturepartfrompps] -> m1!pr; production(k)
[][inq(pr) eq setinputmode] -> d1!pr; production(k)
[][inq(pr) eq requestlocator] -> d1!pr; g2!resin; production(k)
[][inq(pr) eq requestpick] -> d1!pr; g2!resin; production(k)

```
   [][inq(pr) eq samplelocator] -> d1!pr; g2!resin; production(k)
   [][inq(pr) eq samplepick] -> d1!pr; g2!resin; production(k)
   [][inq(pr) eq awaitinput] -> g2!awaitin; g2!resin; production(k))
[]m1!inq_trn(k) !inq_lntrns(k); production(k))
endproc
```

The production and the generation can synchronize when the production receives an await event instruction from the application, then the generation gets the element at top of the event queue on the m5 gate and passes it to the application. The generation uses the m3 gate to receive input in SAMPLE or EVENT mode.

```
process generation[g1, g2, m3](q: Queue):noexit :=
(m3?im:In_kvalue ?md: In_mode;
  ([md eq event] -> generation(add(im, q))
  []([md eq sample) or (md eq request)] -> ([cl eq locator] -> g1!trasf_n(im); g2!resin;
generation(q)
                                          [][cl eq pick] -> g1!im; g2!resin; generation(q)))
[]g2!awaitin; ([inq_cl(top(q)) eq locator] -> g1!trasf_n(top(q)); g2!resin; generation(remove(q))
            [][inq_cl(top(q)) eq pick] -> g1!top(q); g2!resin; generation(remove(q))))
endproc
```

This process is strongly modified with respect to the previous approach because now it has to realize directly the processing of the main data structures(except the input queue). When an output primitive or a picture part is added to the NDCpicture it is sent toward the lower levels, by the m2 gate, to be added, after transformation, to the DC picture.

```
process manipulation[m1, m2, m3, m4](pps: PPS, pn: Partname, ndcp: Ndcpicture) :noexit :=
( m4?p: Point ?md: In_mode ?cl: Input_class;
  ([cl eq pick] -> [det(get_ns(detect(p, ndcp)), s)] -> m2!eco(rmec(ndcp), p);
      m3!mk_pick_kvalue (get_ns( detect(p, ndcp)))!md;
        ([det(get_ns(detect(p, ndcp)), s)] -> manipulation(pps, pn,  eco(rmec(ndcp), p))
        [not(det(get_ns(detect(p, ndcp)), s))]) -> manipulation(pps, pn, ndcp))
   [][cl eq locator] -> m1?tn:List_Trasfcord ?ln: List_Int;
      m3!mk_loc_kvalue(p, get_tr (inp_trasf(tn, ln, p)), get_in(inp_trasf(tn, ln, p)))!md;
      manipulation(pps, pn, ndp)
      )
[]m1?pr:Gks_prim;
   ([inq(pr) eq outputprim] -> m1?st:Status;
                              ([st eq ppop] -> manipulation(addPP(pn, pr, pps), pn, ndp)
                              [][st eq ppcl ] -> m2!pr; manipulation(pps, pn, addndc(pr, ndcp)))
     [][inq(pr) eq beginpicturepart] -> manipulation(begpicpart(pps, pr), getpn(pr), ndp)
     [][inq(pr) eq appendpicturepart] -> manipulation(append(pr, pps), pn, ndp)
     [][inq(pr) eq copypicturepartfrompps] -> m2!cpfpps(pr, pps); manipulation(pps, pn, addpp
                                          (cpfpps (pr, pps), ndp))
     [][inq(pr) eq beginpicturepartagain] -> manipulation(pps, getpn(pr), ndp)
     [][inq(pr) eq closepicturepart] -> manipulation(pps, null, ndp))
     [][inq(pr) eq deleteprimitive] -> m2!del(getsel(pr), ndcp); manipulation(pps, pn,
                                          del(getsel(pr), ndcp)))
endproc
```

The distribution transmits data to the lower level.

```
process distribution[d1, d2, m2] :noexit :=
(d1?pr1:Ws_prim; d2!pr1; distribution
[] m2?pr2:Ndc_primitives; d2!pr2; distribution
) endproc
```

5.2 The logical environment

We assume wsx is the workstation identifier of the workstation.

```
process rendering[d2, w1, c1](w: WsStateList) :noexit :=
(d2?pr: Ws_prim;
   ([inq(pr) eq outputprim] ->(
      [sel(get_ns(pr), inq_wcri(w, visib)) and inq_ve(w)] -> w1!wstrasf(pr, w); rendering(w)
      [][not(sel(get_ns(pr)), inq_wcri(w, visib)) or not(inq_ve(w))] -> rendering(w))
   [][inq(pr) eq Ndcpicture] ->(
            [inq_ve(w)] -> w1!ndcptodcp(pr, w); rendering(w)
             [][not(inq_ve(w))] -> rendering(w))
   [][inq(pr) eq openworkstation] -> rendering(initial_wsStateLs(wsx))
   [][inq(pr) eq closeworkstation] -> exit
   [][inq(pr) eq setworkstationcriteria] -> rendering(setcri(w, pr))
   [][inq(pr) eq setworkstationtransformation] -> c1!pr; rendering(set_wstr(w, pr))
   [][inq(pr) eq setvisualeffect] -> rendering(ve(w, pr))
   [][inq(pr) eq setinputmode] -> c1!pr; rendering(w)
   [][inq(pr) eq requestlocator] -> c1!pr; rendering(w)
   [][inq(pr) eq requestpick] -> c1!pr; rendering(w)
   [][inq(pr) eq samplelocator] -> c1!pr; rendering(w)
   [][inq(pr) eq samplepick] -> c1!pr; rendering(w))
[]c1!inq_wcri(w, detect); rendering(w))
endproc

process assembly[itl, iml, itp, imp, w4m, w4t] :noexit :=
(itl?it:Bool; w4t!locator; assembly
[]iml?im1:Point;    w4m!im1 !locator; assembly
[]itp?it:Bool;  w4t!pick; assembly
[]imp?im2:input_data;    w4m!im2 !pick; assembly)
endproc

process abstraction[w3, w5, c1, m4](tn:Trasfcord) :noexit :=
(w3?im:Point ?md:In_mode ?cl:Input_class; m4!trasf(im, tn) !md !cl; abstraction(tn)
[] c1?prw:Ws_prim;
      ([inq(prw) eq requestlocator] ->  w5!prw; w3?im:Point; m4!trasf(im, tn) !request !locator;
                                        abstraction(tn)
      [][inq(prw) eq requestpick] ->  w5!prw; w3?im:Point; c1?s:Select; m4!trasf(im, tn) !request
                                        !pick; m4!s; abstraction(tn)
```

```
    [][inq(prw) eq samplelocator] ->  w5!prw; w3?im:Point; m4!trasf(im, tn) !sample !locator;
                                      abstraction(tn)
    [][inq(prw) eq samplepick] -> w5!prw; w3?im:Point; c1?s:Select; m4!trasf(im, tn) !sample
                                      !pick; m4!s; abstraction(tn)
    [][inq(prw) eq setinputmode] -> w5!prw; abstraction(tn)
    [][inq(pr) eq setworkstationtransformation] -> abstraction(inv_trasf(gettr(pr))))
)endproc
```

In this approach the workstation manipulation status has to consist also of the status of the logical input devices.

```
process manipulation [w1, w2, w3, w5, w4m, w4t] (cur: Template, iml,imp: Point, pndl, pndb:
Bool, lm,pm: In_mode, dcp: Dcpicture) :noexit :=
(w1?pr: Dc_primitives; w2!pr;
    ([inq(pr) eq outputprim] -> manipulation(cur, iml, imp, pndl, pndp, lm, pm, adddcp(pr, dcp))
   [][inq(pr) eq dcpicture] -> manipulation(cur, iml, imp, pndl, pndp, lm, pm, adddcpic(pr, dcp)))
[]w4m?im:Point ?cl:Input_class;
    ([cl eq locator] -> w2!adddcpic(mk_cursor(im, cur), rem_cursor(dcp)); manipulation (cur,
                    im, imp, pndl, pndp, lm, pm, adddcpic(im, rem_cursor(dcp)))
    [][cl eq pick] -> w3!im !new !pick; manipulation(cur, iml, im, pndl, pndp, lm, pm, dcp))
[]w4t?it:Input_class;
  ([it eq locator] ->
    ([lm eq event]-> w3!iml !lm !it; manipulation(cur, iml, imp, pndl, pndp, lm, pm, dcp)
    [][pndl eq true]-> w3!iml !lm !it; manipulation( cur, iml, imp, false, pndp, lm, pm, dcp)
    [][(lm eq sample) or ((lm eq request) and (pndl eq false))] -> manipulation( iml, imp, pndl,
                    pndp, lm, pm, dcp))
   [][it eq pick] ->
    ([pm eq event]-> w3!imp !pm !it; manipulation(cur, iml, imp, pndl, pndp, lm, pm, dcp)
    [][pndp eq true]-> w3!imp !pm !it; manipulation(cur, iml, imp, pndl, false, lm, pm, dcp)
    [][(pm eq sample) or ((pm eq request) and (pndp eq false))] -> manipulation( cur, iml, imp,
                    pndl, pndp, lm, pm, dcp))
[]w5?pr:Input_prim;
    ([inq(pr) eq requestlocator] -> manipulation(cur, iml, imp, true, pndp, lm, pm, dcp)
    [][inq(pr) eq requestpick] -> manipulation(cur, iml, imp, pndl, true, lm, pm, dcp)
    [][inq(pr) eq samplelocator] ->  w3!iml; manipulation(cur, iml, imp, pndl, pndp, lm, pm, dcp)
    [][inq(pr) eq samplepick] -> w3!imp; manipulation(cur, iml, imp, pndl, pndp, lm, pm, dcp)
    [][inq(pr) eq setinputmode] ->
            ([inqd(pr) eq locator] -> manipulation(cur, iml, imp, pndl, pndp, inqm(pr), pm, dcp)
            [][inqd(pr) eq pick] -> manipulation(cur, iml, imp, pndl, pndp, lm, inqm(pr), dcp))))
) endproc
```

6. A graphical interaction described in the specified framework

The components of the CGRM that are involved in typical graphical interactions will be illustrated following the structure of the first specification discussed(CGRM data structure-LOTOS processes approach). The example chosen for this purpose is a simple graphical editor. The initial layout has on the left a column composed of a vertical sequence of graphical rectangles each one associated with a basic graphic element (i.e. polygon, rectangle, circle, ...). The main area is used to compose

the design. For interaction a three button mouse is used. The possible interactions are:
- to select a basic element from the column and to place a new instance in the graphic area. The event, to position it, is a button press and while pressed feedback shows the current position of the new graphical item depending on the cursor position.
- the primitives in the graphic area can be selected by button press, when selected they are high-lighted.
- an operation on the selected primitive (i.e. move, resize, set_attributes, ..) can be realized by a pop-up menu activated by the third button press.

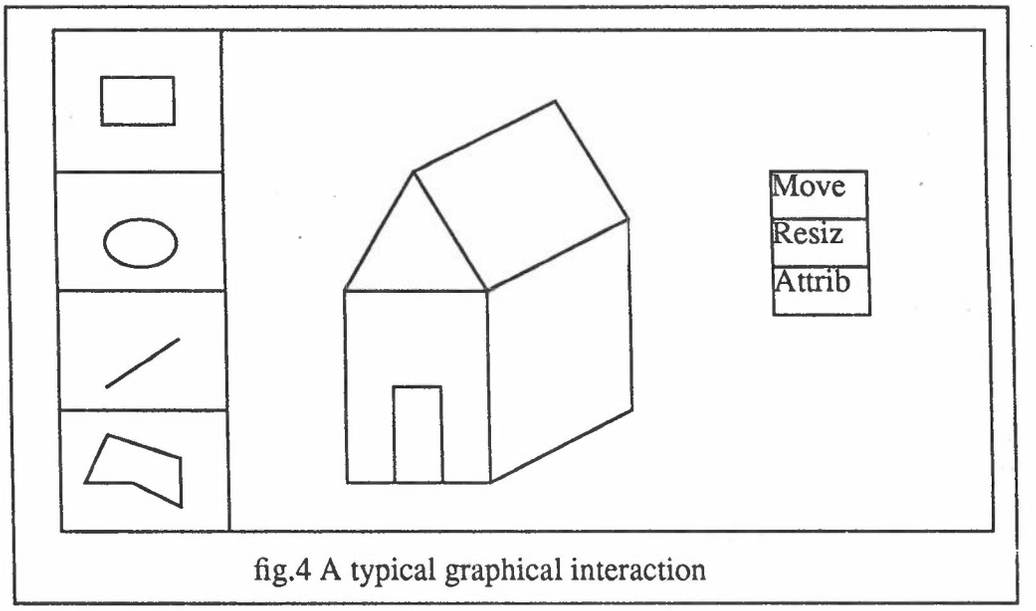

fig.4 A typical graphical interaction

The interaction can be described by five input devices:
- one pick device associated with the side column, whose trigger condition is button 1 pressed and cursor position inside the column;
- one pick input device associated with the graphic area, whose trigger condition is button 2 pressed and cursor position inside the graphic area;
- a locator device to place a new instance in the graphic area, whose trigger condition is button 1 pressed and cursor position in the graphic area;
- a locator device activated by button 3 pressed and cursor position in the graphic area, this device is used to define a position at which to display a pop-up menu;
- a choice device to select an item of the pop-up menu, whose trigger is button 3 release.

The two pick devices and the second locator device (associated with the pop-up menu) are in EVENT mode because the user has to be free to choose which action to realize, the first locator is in REQUEST mode, because a request locator instruction has to be perfomed, after a pick on the side column, to indicate where the new instance has to be placed. Also the choice device is in REQUEST mode and it is requested from the application after receiving an event from the second locator. When it is requested the menu with its items appears and the element selected by the mouse position, when the button is up, is sent to the application.
At the beginning only the three devices in EVENT mode are activated. In both the pick devices the primitives detected by the position of the cursor are highlighted. This is obtained by sending the cursor positions from the assembly through aggregation, abstraction, and manipulation, to the

scene. Here their name set is changed so as to satisfy the selection criterion for highlighting of the workstation. The updated primitives are distributed for display on the workstation.
The application after receiving a pick input from the side column starts a request locator instruction. This means a new echo (the geometric shape selected) related to the cursor position is displayed through the workstation manipulation, when it is in the graphic area. If the trigger fires the current position is sent to the application, after transformation by the inverse workstation transformation (abstraction) and the appropriate inverse normalization transformation (kernel manipulation and generation). Then the application will send the graphical primitives to display the selected object in the selected position.
Manipulation of a displayed instance of a graphical element is achieved by selecting the instance using the second mouse button. Pressing the third button then generates an event that is sent to the application to ask for a request choice primitive associated with the pop-up menu that will appear in the position previously selected, and when the buttom is up, the selected choice is sent to the application. Then the parameters of the select operation have to be provided by the operator: this can be easily achieved in different ways.

8. Conclusions

This work shows that LOTOS can be used as a specification language for graphics systems because it provides a rigorous tool to describe the cooperation among the components and the definition and processing of graphics data types. A parallel approach can provide more information to the users and the implementators of the graphics system regarding the cooperation among its components. It facilitates refinement to parallel language implementations, such as Occam.
We investigated two possible approaches in the decomposition of the graphics system following the structure of the CGRM. Both approaches use the same set of basic graphics data type for the definition of their parameters and input data types. Their algeabric definitions can be found in the appendix.
If we compare the two approaches we can note the first one requires a larger number of processes and this allows all the possible accesses to a particular CGRM data structure to be shown in a clearer way. The second one is more compact. A choice between the two approach is probably driven by the taste and the background of the user of the formal specification. Future work is planned to formally explore their equivalence.
We can conclude that a description of the architecture of the New GKS is realizable using the CGRM framework (this was not evident at the beginning). It also pushes towards to a more parallel description: if we compare this work to the previous attempt to specify GKS-R functionality by LOTOS, we notice that the system, in both approaches, has been divided into a higher number of components (in the previous case it was just divided into kernel and workstation processes).

Acknowledgments

This work was carried out whilst Fabio Paterno' was a Visiting Scientist at Rutherford Appleton Laboratory. Support from Consiglio Nazionale delle Ricerche and Rutherford Appleton Laboratory is gratefully acknowledged.

Bibliography

[1] ISO/IEC DIS 11 072. Information Processing Systems. Computer Graphics. Computer Graph-

ics Reference Model. ISO Centrals Secretariat, Geneva, 1991.
[2] ISO/IEC. Information Processing Systems. Computer Graphics. New Graphical Kernel System (GKS-R) functional description.
[3]K.W.Brodlie, D.A.Duce, F.R.A.Hopgood. "The New Graphical Kernel System". Computer Aided Design, N.4,1991.
[4]T.Bolognesi, H.Brinksma. "Introduction to the ISO Specification Language LOTOS". Computer Networks and ISDN Systems, vol.14, pp.25-59,1987.
[5]R.Milner. "A Calculus for Communicating Systems". LNCS 92, Springer Verlag, 1980.
[6]H.Ehrig, B.Mahr. "Fundamentals of Algebraic Specification 1 - Equations and Initial Semantics". Berlin. 1985. Springer-Verlag.
[7] D.B.Arnold, D.A.Duce, G.J.Reynolds. "An Approach to the Formal Specification of Configurable Models of Graphics Systems". Proceedings of Eurographics Conference 1987. Amsterdam
[8] D.Duce, F.Paterno'."LOTOS Description of GKS-R functionality". Eurographics Workshop Formal Methods in Computer Graphics. Marina di Carrara. 1991.

Appendix A

In this appendix the data types used for the parameters and the channel data types are defined. The naming conventions are variables and operators in lower case, sort names with initials in upper case.

A generic transformation is described by six coefficients, representing the composition of basic transformation along the two directions. In our specification it is used for coordinates transformations that are defined by two translations and one scale transformation. The mult operation realizes a matrix product between square matrixes with the first two columns from the Trasfcord data and the third from the identity matrix. It returns a Trasfcord that is the resulting matrix without the third column.

```
type TRASFCORD is REAL
sorts Trasfcord
opns mk_trasfcor: Real, Real, Real, Real, Real, Real      -> Trasfcord
     inv_trasf: Trasfcord                                  -> Trasfcord
     mult: Trasfcord, Trasfcord                            -> Trasfcord
     identity:                                             -> Trasfcord
eqns forall a,b,c,d,e,f,g,h,z,l,m,n: Real ofsort Trasfcord
     identity = mk_trasfcor(1, 0, 0, 0, 1, 0);
     mult(mk_trasfcor(a, b, c, d, e, f), mk_trasfcor(g, h, z, l, m, n)) = mk_trasfcor(a*g+d*h,
     b*g+e*h, c*g+f*h+z, a*l+d*m, b*l+e*m, c*l+f*m+n);
     mult(mk_trasfcor(a, b, c, d, e, f), inv_trasf(mk_trasfcor(a, b, c, d, e, f))) = identity;
endtype

type POINT is TRASFCORD
sorts Point
opns mk_point: Real, Real                                  -> Point
     get_x: Point                                          -> Real
     get_y: Point                                          -> Real
```

```
    trasf: Point, Trasfcord                                          -> Point
eqns forall x,y,a,b,c,d,e,f:Real  ofsort Real
      get_x(mk_point(x, y)) = x;
      get_y(mk_point(x, y)) = y;
      ofsort Point
      trasf(mk_point(x, y), mk_trasfcor(a, b, c, d, e, f))=mk_point(x*a+y *b + c, x* d+y * e+ f);
endtype

type NDCPOINT is POINT
sorts Ndcpoint
      eqns  forall x,y:Real   ofsort Bool
      get_x(mk_point(x,y))ge 0 and get_x(mk_point(x,y))le 1  = true;
      get_y(mk_point(x,y))ge 0 and get_y(mk_point(x,y))le 1  = true;
endtype

type POINTS is POINT
sorts Points
opns emptypoints :                                                   -> Points
      addpoint  : Point, Points                                      -> Points
      trasfpoints: Points, Trasfcord                                 -> Points
eqns forall pn : Point, pnts : Points, mat : Trasfcord   ofsort Points
      trasfpoints(emptypoints, mat) = emptypoints;
      trasfpoints(addpoint(pn, emptypoints), mat) = addpoint(trasf(pn, mat), emptypoints);
      trasfpoints(addpoint(pn,pnts), mat) = addpoint(trasf(pn, mat), trasfpoints(pnts, mat));
endtype

type NDCPOINTS is POINTS
sorts Ndcpoints
      eqns  forall x,y:Real   ofsort Bool
      get_x(mk_point(x,y))ge 0 and get_x(mk_point(x,y))le 1  = true;
      get_y(mk_point(x,y))ge 0 and get_y(mk_point(x,y))le 1  = true;
endtype

This data type allows associates integers and normalization transformations.
type VIEW_INP is TRASFCORD
sorts View_inp
opns mk_view_inp:Int, Trasfcord                                      ->View_inp
      get_in: View_inp                                               -> Int
      get_tr: View_inp                                               -> Trasfcord
eqns forall num: Int, tr: Trasfcord ofsort Int
      get_in(mk_view_inp(num, tr)) = num;
      ofsort Trasfcord
```

```
    get_tr(mk_view_inp(num, tr)) = tr;
endtype


type VIEWPORT is POINT
sorts Viewport
opns mk_vp: Point, Point                                    -> Viewport
endtype
```

The list of coordinate trasformations has been introduced to define the operation to detect the inverse normalization transformation to apply to locator input. The inp_trasf operation returns the inverse of the normalization transformation containing the point with the highest priority and its associated number.

```
type LIST_TRASFCORD is TRASFCORD, VIEW_INP, VIEWPORT, LIST_INT
sorts List_Trasfcord
opns emptylist:                                             -> List_Trasfcord
     add_trasf: List_Trasfcord, Trasfcord, Viewport         -> List_Trasfcord
     inp_trasf: List_Trasfcord, List_Int, Point             -> View_inp
     gett: Int, List_Int, List_Trasfcord                    -> Trasfcord
eqns forall ltr: List_Trasfcord, t,tj: Trasfcord, ii,j: Int, li: List_Int, vp: Viewport, x,x1,x2, y,y1,y2:
Real ofsort Trasfcord
     if (ii eq j) =>
     gett(ii, add_int(li, j), add_trasf(ltr, tj, vp)) = trj;
     if (ii neq j) =>
     gett(ii, add_int(li, j), add_trasf(ltr, tj, vp)) = gett(ii, li, ltr);
     ofsort View_inp
     if ((x1 leq x) and (x leq x2) and (y1 leq y) and (y leq y2)) =>
     inp_trasf(add_trasf(ltr, t, mk_vp(mk_point(x1, y1), mk_point(x2, y2))), add_int(li, ii),
        mk_point(x,y)) = mk_view_inp(ii, inv_trasf(t));
     if not((x1 leq x) and (x leq x2) and (y1 leq y) and (y leq y2)) =>
     inp_trasf(add_trasf(ltr, t, mk_vp(mk_point(x1, y1), mk_point(x2, y2))), add_int(li, ii),
        mk_point(x,y)) = inp_trasf(ltr, li, mk_point(x,y));
endtype


type LOCATOR is POINT
sorts Locator_data
opns mk_locator: Point, Int                                 -> Locator_data
endtype
```

The type name set is obtained from the definition of set from the basic LOTOS library by renaming (one of the features provided by LOTOS to build structured specifications).

```
type NAME is
sorts Name, Partname, Wsid
opns n1, n2, n3 :                                           -> Name
```

```
        null, p1, p2, p3  :                                   -> Partname
        w1, w2, w3  :                                         -> Wsid
        pptoname: Partname                                    -> Name
        wsidtoname: Wsid                                      -> Name
endtype

type NAMESET is
SET renamedby
sortnames Nameset for set
                Name for element
opnnames      emptyNS for { }
endtype

type ATTRIBUTES is
sorts Highlt, Detectab
opns highlighted:                                             -> Highlt
        normal:                                               -> Highlt
        detectable:                                           -> Detectab
        undetectable:                                         -> Detectab
endtype
```

Here only the 'contains' selection criterion is described but other selection criteria can be added in a similar way. They are stored in the workstation state list to filter the primitives arriving from the kernel.

```
type SELECTION is NAMESET, BOOL, ATTRIBUTES
sorts Select
opns contains: Nameset                                        -> Select
        sel: Nameset, Select                                  -> Bool
        det: Nameset, Select                                  -> Detectab
        high: Nameset, Select                                 -> Highlt
        selectall, rejectall:                                 -> Select
eqns forall ns, ns1, ns2 : Nameset, sv, sh : Select
         ofsort Bool
        sel(ns1, contains(ns2)) = ns1 includes ns2;
        sel(ns1, contains(emptyNS)) = (ns1 eq emptyNS);
        sel(ns, contains((ns1) union (ns2))) = (sel(ns, contains(ns1))) and (sel(ns, contains(ns2)));
        ofsort Highlt
        sel(ns, sv) =>
        det(ns, sv) = highlighted;
        not(sel(ns, sv)) =>
        det(ns, sv) = normal;
        ofsort Detectab
        sel(ns, sh) =>
        high(ns, sh) = detectable;
        not(sel(ns, sh)) =>
        high(ns, sh) = undetectable;
```

endtype

This type has been introduced to manage different types of selection criteria.
type CRITERION is
sorts Criterion
opns visib: -> Criterion
detect: -> Criterion
highligh: -> Criterion
endtype

Here some operations which modify the kernel or the workstations status are defined.
type CONTROL_PRIMITIVE is NAME, TRASFCORD, NAMESET, CRITERION
sorts Control_prim
opns mk_opws: Wsid -> Control_prim
mk_clws: Wsid -> Control_prim
mk_delete_primitives: Select -> Control_prim
mk_begin_picturepart: Partname -> Control_prim
mk_begin_picturepartagain: Partname -> Control_prim
mk_close_picturepart: Partname -> Control_prim
mk_appendpp: Partname, Partname, Trasfcord, Nameset -> Control_prim
mk_copyppfrompps: Partname, Select, Trasfcord, Nameset -> Control_prim
mk_sel_cri: Wsid, Criterion, Select -> Control_prim
mk_set_visef: Wsid, Bool -> Control_prim
mk_sel_norm_transf: Int -> Control_prim
mk_set_ws_transf: Trasfcord -> Control_prim
mk_set_ns_attr: Nameset -> Control_prim
inqcp: Control_prim -> String
getwsid: Control_prim -> Wsid
getsel: Control_prim -> Select
gettr: Control_prim -> Trasfcord
getpn: Control_prim -> Partname
eqns forall ws : Wsid, bol : Bool, ns : Nameset, pn,pn1 :Partname, sel : Select, tr: Trasfcord, n: Int, cr: Criterion ofsort String
inqcp(mk_opws(ws)) = openws;
inqcp(mk_clws(ws)) = closews;
inqcp(mk_delete_primitives(sel)) = deleteprimitive;
inqcp(mk_begin_picturepart(pn)) = beginpicturepart;
inqcp(mk_begin_picturepartagain(pn)) = beginpicturepartagain;
inqcp(mk_close_picturepart(pn)) = closepicturepart;
inqcp(mk_app_picture_part(pn1, pn, tr, ns)) = appendpicturepart;
inqcp(mk_sel_cri(ws, cr, sel)) = setselectioncriteria;
inqcp(mk_set_vis_ef(ws, bol)) = setvisualeffect;
inqcp(mk_copyppfrompps(pn, sel, tr, ns)) = copypicturepartfrompps;
inqcp(mk_sel_norm_transf(n)) = selectnormalizationtransformation;
inqcp(mk_set_ws_transf(tr)) = setworkstationtransformation;
inqcp(mk_set_ns_attr(ns)) = setnamesetattribute

```
        ofsort Wsid
        getwsid(mk_opws( ws)) = ws;
        getwsid(mk_clws(ws)) = ws;
        getwsid(mk_sel_cri(ws, cr, sel)) = ws;
        getwsid(mk_set_visef(ws, bol)) = ws;
        ofsort Select
        getsel(mk_delete_primitives(sel)) = sel;
        ofsort Trasfcord
        gettr(mk_set_ws_trasnf(tr)) = tr;
        ofsort Partname
        getpn(mk_begin_picture_part(pn)) = pn;
        getpn(mk_begin_picture_part_again(pn)) = pn;
        getpn(mk_copyppfrompps(pn, sel, tr, ns)) = pn;
endtype

type INPUT_CLASS is
sorts Input_class
opns  locator:                                          -> Input_class
      pick:                                             -> Input_class
endtype
```

This is the data type for the elements of the input queue.

```
type INPUT_KVALUE is NDCPOINT, LOCATOR, NAMESET, INPUT_CLASS
sorts In_kvalue
opns mk_loc_kvalue: Ndcpoint, Trasfcord, Int            -> In_kvalue
     mk_pick_kvalue:  Nameset                           -> In_kvalue
     null_element:                                      -> In_kvalue
     trasf_n: In_kvalue                                 -> Locator_data
     inq_cl: In_kvalue                                  -> Input_class
eqns forall tn : Trasfcord, pn : Ndcpoint, numb : Int, ns: Nameset ofsort In_kvalue
     trasf_n(mk_loc_kvalue(pn, tn, numb)) = mk_locator(trasf(pn, tn), numb);
     ofsort Input_class
     inq_cl(mk_loc_kvalue(pn, tn, numb)) = locator;
     inq_cl(mk_pick_kvalue(ns)) = pick;
endtype
```

The 'new' mode has been introduced to indicate to the manipulation that the current pick value has been sent to it only to realize an echo.

```
type INPUT_MODE is
sorts In_mode
opns event :                                            -> In_mode
     sample:                                            -> In_mode
     request:                                           -> In_mode
```

```
        new:                                                    -> In_mode
endtype

type INPUT_PRIMITIVE is STRING, INPUT_MODE, INPUT_CLASS
sorts Input_prim
opns mk_requestlocator:                                         -> Input_prim
        mk_requestpick:                                         -> Input_prim
        mk_samplelocator:                                       -> Input_prim
        mk_samplepick:                                          -> Input_prim
        mk_awaitinput :                                         -> Input_prim
        mk_setinputmode: Wsid, Input_class, In_mode             -> Input_prim
        inqip: Input_prim                                       -> String
        inqd: Input_prim                                        -> Input_class
        inqm: Input_prim                                        -> In_mode
eqns    forall ws: Wsid, cl:Input_class, md: In_mode ofsort String
        inqip(mk_requestlocator) = requestlocator;
        inqip(mk_requestpick) = requestpick;
        inqip(mk_samplelocator) = samplelocator;
        inqip(mk_samplepick)  = samplepick;
        inqip(mk_awaitinput)  = awaitinput;
        inqip(mk_setinputmode(ws, cl, md)) = setinputmode;
        ofsort Input_class
        inqd(mk_setinputmode(ws, cl, md)) = cl;
        ofsort In_mode
        inqm(mk_setinputmode(ws, cl, md)) = md;
endtype

type GKS_PRIMITIVE is  OUTPUT_PRIMITIVE , CONTROL_PRIMITIVE , INPUT_PRIMI-
TIVE
sorts Gks_prim
opns mkgp1: Output_prim                                         -> Gks_prim
        mkgp2: Control_prim                                     -> Gks_prim
        mkgp3: Input_prim                                       -> Gks_prim
        inq: Gks_prim                                           -> String
eqns  forall op : Output_prim, cp : Control_prim, ip : Input_prim ofsort String
        inq(mkgp1(op)) = inqop(op);
        inq(mkgp2(cp)) = inqcp(cp);
        inq(mkgp3(ip)) = inqip(ip);
endtype

 type STATUS is
 sorts Status
 opns   ppop :                                                  -> Status
        ppcl :                                                  -> Status
 endtype
```

When a workstation is opened its name is added to the name set of the kernel state list. In this

specification we assume that a list of normalization transformations have been previously defined and it is possible to modify the current normalization transformation by the Select Normalization Transformation instruction.

```
type KERNELSTATELIST  is BOOL, NAMESET, PICTUREPART, LIST_TRASFCORD
sorts  KernelStateList
opns mk_krn_list: Status, Nameset, List_Trasfcord, List_Int, Trasfcord     -> KernelStateList
       opws: KernelStateList,    Control_prim                              -> KernelStateList
       clws: KernelStateList,    Control_prim                              -> KernelStateList
       bpp: KernelStateList                                                -> KernelStateList
       cpp: KernelStateList                                                -> KernelStateList
       sel_tn: KernelStateList, Control_prim                               -> KernelStateList
       set_ns: KernelStateList, Control_prim                               -> KernelStateList
       inq_ns: KernelStateList                                             -> Nameset
       inq_state: KernelStateList                                          -> Status
       cur_trasf: KernelStateList                                          -> Trasfcord
       inq_trn:KernelStateList                                             -> List_Trasfcord
       inq_lntrns: KernelStateList                                         -> List_Int
eqns forall st : Status, ns,ns1 : Nameset, tr: List_Trasfcord, nt: List_Int, t,t1: Trasfcord, w1: Wsid,
  n: Int ofsort   Nameset
       inq_ns(mk_krn_list(st, ns, tr, nt, t)) = ns;
       ofsort Trasfcord
       cur_trasf(mk_krn_list(st, ns, tr, nt, t)) = t;
       ofsort Status
       inq_state(mk_krn_list(st, ns, tr, nt, t)) = st;
       ofsort List_Trasfcord
       inq_trn(mk_krn_list(st, ns, tr, nt, t)) = tr;
       ofsort List_Int
       inq_ltrn(mk_krn_list(st, ns, tr, nt, t)) = nt
       ofsort KernelStateList
       opws(mk_krn_list(st, ns, tr, nt, t), mk_opws(w1)) = mk_krn_list(st, ns union {w1}, tr, nt, t);
       clws(mk_krn_list(st, ns, tr, nt, t), mk_clws(w1)) = mk_krn_list(st, ns remove {w1}, tr, nt, t);
       bpp(mk_krn_list(ppcl, ns, tr, nt, t)) = mk_krn_list(ppop, ns, tr, nt, t);
       cpp(mk_krn_list(ppop, ns, tr, nt, t)) = mk_krn_list(ppcl, ns, tr, nt, t);
       sel_tn(mk_krn_list(st, ns, tr, nt, t1), mk_sel_norm_transf(n)) = mk_krn_list(st, ns, tr, nt,
        gett(n, nt, tr));
       set_ns(mk_krn_list(st, ns, tr, nt, t1), mk_set_ns_attr(ns1)) = mk_krn_list(st, ns1, tr, nt, t);
endtype

type OUTPUT_PRIMITIVE is POINTS, KERNELSTATELIST, NDC_PRIMITIVE
sorts Output_prim
opns mk_out_prim:  Points                                                  -> Output_prim
       wctondc: Output_prim, KernelStateList                               -> Ndc_prim
       inqop: Output_prim                                                  -> String
eqns forall pns : Points, k : KernelStateList   ofsort String
        inqop(mk_out_prim(pns)) = outputprim;
```

```
    ofsort Ndc_prim
    wctondc(mk_out_prim(pns), k) = mk_ndcpr(trasfpoints(pns, cur_trasf(k)), inq_ns(k));
endtype
```

The realization of echo of the pick device is amongst the functions defined in the type NDCPICTURE. The detect function returns the picked primitive (only one) and eco realizes highlighting by adding the element hl to the name set of the picked primitive. The last equation of the eco function indicates that when a new primitive is highlighted any previous picked primitive is returned to the normal state.

```
type NDCPICTURE is NDC_PRIMITIVE, PICTUREPART
sorts Ndcpicture
opns emptyNDCP:                                                    -> Ndcpicture
     mkNDCP: Ndc_prim                                              -> Ndcpicture
     addndc: Ndc_prim, Ndcpicture                                  -> Ndcpicture
     del: Select, Ndcpicture                                       -> Ndcpicture
     detect: Point,  Ndcpicture                                    -> Ndc_prim
     addpp: PicturePart, Ndcpicture                                -> Ndcpicture
     eco: Ndcpicture, Point                                        -> Ndcpicture
     rmec: Ndcpicture                                              -> Ndcpicture
eqns forall crit : Select, pn, ps : Points, ns: Nameset, ndcp : Ndcpicture, pp : PicturePart
     ofsort Ndcpicture
     del(crit, emptyNDCP) = emptyNDCP;
     sel(get_ns(mk_ndcpr(ps, ns)), crit)  eq true =>
     del(crit, addndc(mk_ndcpr(ps, ns), ndcp))=del(crit, ndcp);
     sel(get_ns(mk_ndcpr(ps, ns)), crit)  eq false =>
     del(crit, addndc(mk_ndcpr(ps, ns), ndcp)) = addndc(mk_ndcpr(ps, ns), del(crit, ndcp));
     addpp(emptypp, ndcp) = ndcp;
     addpp(addprpp(mk_ndcpr(ps, ns), pp), ndcp) = addndc(mk_ndcpr(ps, ns),addpp(pp, ndcp));
     eco(emptyNDCP, pn) = emptyNDCP;
     detect(pn, ndcp) = mk_ndcpr(ps, ns) =>
     eco(addndc(mk_ndcpr(ps, ns), ndcp), pn) = addndc( mk_ndcpr(ps, ns union {hl}), ndcp);
     not(detect(pn, ndcp)) = mk_ndcpr(ps, ns) =>
     eco(addndc(mk_ndcpr(ps, ns), ndcp), pn) = addndc( mk_ndcpr(ps, ns), eco(ndcp, pn));
     rmec(addndc(mk_ndcpr(ps, ns), ndcp)) = addndc(mk_ndcpr(ps, ns remove{hl}), rmec(ndcp));
endtype
```

The operations allowed on the picture part are: to add a primitive to a picture part (addprpp), to select a picture part obtained by the primitives of a picture part satisfying a selection criterion (selpp), to add a name set to the primitives of a picture part (addns), to append a picture part to another (appendpp), to transform the primitives of a picture part (trasfpp).

```
type PICTUREPART is NDCPRIMITIVE, POINTS
sorts PicturePart
opns emptypp:                                                      -> PicturePart
     addprpp: Ndc_prim, PicturePart                                -> PicturePart
     addns: Nameset,  PicturePart                                  -> PicturePart
     appendpp: PicturePart, PicturePart                            -> PicturePart
```

```
    trasfpp: PicturePart, Trasfcord                          -> PicturePart
    selpp: PicturePart, Select, trascord, Nameset            -> PicturePart
eqns forall ns,ns1 : Nameset, cri : Select, mat : Trasfcord, pnts : Points, ndcp : Ndcpicture, pp,pp1:
PicturePart ofsort PicturePart
    selpp(emptypp, cri,  mat, ns) = emptypp;
    sel(ns, cri) eq true =>
    selpp(addprpp(mk_ndcpr(pnts, ns), pp), cri, mat, ns1) = addprpp(trasf (mk_ndcpr(pnts, ns
    union ns1), mat), selpp(pp, cri, mat, ns1));
    sel(ns, cri) eq false =>
    selpp(addprpp(mk_ndcpr(pnts, ns), pp), cri, mat, ns1) = selpp(pp, cri, mat, ns1);
    trasfpp(emptypp) = emptypp;
    trasfpp(addprpp(mk_ndcpr( pnts, ns), pp), mat)  = addprpp(mk_ndcpr(trasfpoints(pnts, mat),
    ns), trasfpp(pp, mat));
    addns(ns1,emptypp) = emptypp;
    addns(ns1, addprpp(mk_ndcpr(pnts, ns), pp)) = addprpp(mk_ndcpr(pnts, ns union ns1), pp);
    appendpp(emptypp, pp) = pp;
    appendpp(addprpp(p, pp), pp1) = addprpp(p, appendpp(pp, pp1));
endtype
```

In the Picture Part Store are defined operations to make a PPS, to unify two PPS, to begin a new picture part in the PPS, to append a picture part to a PPS, to get a picture part from a PPS and to add a primitive to a PPS.

```
type PICTUREPARTSTORE is PICTUREPART, CONTROL_PRIMITIVE
sorts PPS
    opns emptyPPS:                                           -> PPS
    mkPPS: Partname, PicturePart                             -> PPS
    _union_: PPS,  PPS                                       -> PPS
    begpicpart: PPS, Control_prim                            -> PPS
    append: PPS, Control_prim                                -> PPS
    addPP: Partname, Ndc_prim, PPS                           -> PPS
    getpp: Partname, PPS                                     -> PicturePart
    cpfpps:  PPS, Control_prim                               -> PicturePart
eqns forall pn1,pn2: Partname, p: Ndc_prim, pps1, pps2, pps3: PPS, ns: Nameset, mat: Trasfcord
ofsort PPS
    emptyPPS union pps1 = pps1;
    pps1 union pps1 = pps1;
    pps1 union pps2 = pps2 union pps1;
    pps1 union (pps2 union pps3) = (pps1 union pps2) union pps3;
    begpicpart(pps1, mk_begin_picturepart(pn1)) = mkPPS(pn1, emptyPP) union (pps1);
    begpicpart(pps1, mk_begin_picturepartagain(pn1)) = mkPPS(pn1, emptyPP) union (pps1);
    append(pps1, mk_appendpp(pn1, pn2, mat, ns)) = mkPPS(pn1, appendpp(addns(ns,
        trasf(getpp(pn1, pps1), mat)), getpp(pn2, pps1)) union deletepicturepart(pn1, pps1));
    addPP(pn1, p, addPP(pn2, p, pps3)) = addPP(pn2, p, addPP(pn1, p, pps3));
    addPP(pn, p, pps1) union pps2 = addPP(pn, p, pps1 union pps2);
    addPP(pn1, p, emptyPPS) = emptyPPS;
    pn1 eq pn2 =>
```

```
        addPP(pn1, p, mkPPS(pn2, pp)) = mkPPS(pn2, addprpp(p, pp));
        pn1 ne pn2 =>
        addPP(pn1, p, mkPPS(pn2, pp)) = mkPPS(pn2, pp);
        addPP(pn1, p, pps1 union pps2) = addPP(pn1, p, pps1) union addPP(pn1, p, pps2);
        ofsort PicturePart
        getpp(pn1, emptyPPS) = emptypp;
        pn1 eq pn2 =>
        getpp(pn1, mkPPS(pn2, pp) union pps1) = pp;
        pn1 ne pn2 =>
        getpp(pn1, mkPPS(pn2, pp) union pps1) =  getpp(pn1, pps1);
        cpfpps(pps1, mk_copyppfrompps(pn, sel, tr, ns)) = selpp(getpp(pn, pps1), sel, tr, ns);
endtype
```

The described queue is the input one where can be added or removed locator and pick elements.

```
type QUEUE is INPUT_KVALUE
sorts Queue
opns emptyq:                                                    -> Queue
        add: In_kvalue, Queue                                   -> Queue
        top: Queue                                              -> In_kvalue
        remove: Queue                                           -> Queue
eqns forall x : In_kvalue, q : Queue ofsort In_kvalue
        top(emptyq)=null_element;
        top(add(x, emptyq))=x;
        q neq emptyq =>
        top(add(x, q)) = top(q);
        ofsort Queue
        remove(add(x, emptyq)) = emptyq;
        q neq emtpyq =>
        remove(add(x, q)) = add(x, remove(q));
endtype
```

This data type represents a output primitive after Production processing.

```
type NDC_PRIMITIVE is WORKSTATIONSTATELIST, SELECTION, NDCPOINTS, STRING
sorts Ndc_prim
opns mk_ndcpr:  Ndcpoints, Nameset                              -> Ndc_prim
        wstrasf: Ndc_prim, WsStateList                          -> Dc_prim
        inqop: Ndc_prim                                         -> String
        get_ns: Ndc_prim                                        -> Nameset
eqns forall pnts : Points, ns : Nameset, wsl : WsStateList   ofsort String
        inqop(mk_ndcpr(pnts, ns)) =  outputprim;
        ofsort Nameset
        get_ns(mk_ndcpr(pnts, ns)) = ns;
        ofsort Dc_prim
        wstrasf(mk_ndcpr(pnts, ns), wsl) = mk_dc_prim(trasfpoints(pnts, inq_wstr (wsl)),
                        det (inq_wcri(wsl, detect), ns), high(inq_wcri(wsl, highligh), ns));
endtype
```

```
type NDC_PRIMITIVES is NDC_PRIMITIVE, NDCPICTURE, STRING
sorts Ndc_primitives
opns mk_ndcps1: Ndc_prim                                   -> Ndc_primitives
     mk_ndcps2: Ndcpicture                                 -> Ndc_primitives
     inqndcps: Ndc_primitives                              -> String
eqns forall pr: Ndc_prim, ndcp: Ndcpicture   ofsort String
     inqndcps(mk_ndcps1(pr)) = inqop(pr);
     inqndcps(mk_ndcps2(ndcp)) = ndcpicture;
endtype
```

Input and control primitives arrive at the workstation level at the same gate and so are grouped in the same data type, with output primitives.

```
type WS_PRIMITIVE is  CONTROL_PRIMITIVE, INPUT_PRIMITIVE, NDC_PRIMITIVES
sorts Ws_prim
opns mkwp1: Control_prim                                   -> Ws_prim
     mkwp2: Input_prim                                     -> Ws_prim
     mkwp3: Ndc_primitives                                 -> Ws_prim
     inq: Ws_prim                                          -> String
eqns  forall cp : Control_prim, ip : Input_prim ofsort String
     inq(mkwp1(cp)) = inqcp(cp);
     inq(mkwp2(ip)) = inqip(ip);
     inq(mkwp3(ip)) = inqndcps(ip)
endtype

 type WORKSTATIONSTATELIST is SELECTION, TRASFCORD, CONTROL_PRIMITIVE
 sorts WsStateList
opns mkwsStLs: Select, Select, Select, Trasfcord, Bool     -> WsStateList
     initial_wsStateLs: Wsid                               -> WsStateList
     ve:WsStateList, Control_prim                          -> WsStateList
     set_cri: WsStateList, Control_prim                    -> WsStateList
     set_wstr: WsStateList, Control_prim                   -> WsStateList
     inq_wstr: WsStateList                                 -> Trasfcord
     inq_ve: WsStateList                                   -> Bool
     inq_wcri: WsStateList, Criterion                      -> Select
eqns forall s,s1,s2,s3 : Select, bl,bl1: Bool, tr,tr1: Trasfcord, wsx: Wsid ofsort WsStateList
     initial_wsStateLs(wsx) = mkwsStLs(contains(wsx), contains(hl), selectall, identity, true);
     ve(mkwsStLs(s, s1, s2, tr, bl), mk_set_visef(wsx, bl1)) = mkwsStLs(s, s1, s2, tr, bl1);
     set_cri(mkwsStLs(s1, s2, s3, tr, bl), mk_sel_cri(wsx, visib, s)) = mkwsStLs(s, s2, s3, tr, bl);
     set_cri(mkwsStLs(s1, s2, s3, tr, bl), mk_sel_cri(wsx, detect, s)) = mkwsStLs(s1, s, s3, tr, bl);
     set_cri(mkwsStLs(s1, s2, s3, tr, bl), mk_sel_cri(wsx, highligh, s)) = mkwsStLs(s1, s2, s, tr, bl);
     set_wstr(mkwsStLs(s1, s2, s3, tr, bl), mk_set_ws_transf(tr1)) = mkwsStLs(s1, s2, s3, tr1, bl);
     ofsort  Bool
     inq_ve(mkwsStLs(s1, s, s3, tr, bl)) = bl;
```

```
    ofsort Select
    inq_wcri(mkwsStLs(s1, s2, s3, tr, bl), visib) = s1;
    inq_wcri(mkwsStLs(s1, s2, s3, tr, bl), detect) = s2;
    inq_wcri(mkwsStLs(s1, s2, s3, tr, bl), highligh) = s3;
    ofsort Trasfcord
    inq_wstr(mkwsStLs(s1, s2, s3, tr, bl)) = tr;
endtype
```

This type has been introduced because defining a DC primitive with highlighting and detectability attributes it cannot be seen as an instance of a generic output primitive

```
type DCPRIMITIVE is
sorts Dc_prim
opns mk_dc_prim: Points, Detectab, Highlt                    -> Dc_prim
endtype
```

This description of the dcpicture allows the composition of the dcprimitives arriving from the rendering and the echo of the input devices arriving from the aggregation. A cursor is defined as a dcpicture generated from a point and a predefined shape.

```
type DCPICTURE is NDC_PRIMITIVES, DCPRIMITIVE, WORKSTATIONSTATELIST
sorts Dcpicture
opns emptydcp:                                               -> Dcpicture
      adddcp: Dc_prim, Dcpicture                             -> Dcpicture
      adddcpic: Dcpicture, Dcpicture                         -> Dcpicture
      mk_cursor: Point, Template                             -> Dcpicture
      rem_cursor: Dcpicture                                  -> Dcpicture
      ndcptodcp: Ndcpicture, WsStateList                     -> Dcpicture
eqns forall dcp1 : Dcpicture, pn : Point, ps: Points, tm : Template, wsl: WsStaeteList, ns:Nameset,
ndcp: Ndcpicture ofsort Dcpicture
      ndcptodcp(emptyNDCP) = emptydcp;
      sel(ns, inq_wcri(wsl, visib)) =>
      ndcptodcp(addndc(mk_ndcpr(ps, ns), ndcp), wsl) = adddcp(wstrasf(mk_ndcpr(ps, ns), wsl),
      ndcptodcp(ndcp, wsl));
      not(sel(ns, inq_wcri(wsl, visib))) =>
      ndcptodcp(addndc(mk_ndcpr(ps, ns), ndcp), wsl) = ndcptodcp(ndcp, wsl);
      rem_cursor(adddcpic(mk_cursor(pn, tm), dcp1)) = dcp1;
endtype
```

```
type DC_PRIMITIVES is OUTPUT_PRIMITIVE, STRING, ATTRIBUTES
sorts Dc_primitives
opns mk_dcps1: Output_prim, Highlt, Detectab                 -> Dc_primitives
      mk_dcps2: Dcpicture                                    -> Dc_primitives
      inq: Dc_primitives                                     -> String
eqns forall pr: Output_prim, ndcp: Ndcpicture ofsort String
      inq(mk_ndcps1(pr)) = inqop(pr);
      inq(mk_ndcps2(ndcp)) = dcpicture;
```

```
endtype

These are some possible cursor types
type TEMPLATE is
sorts Template
opns arrow:                                   -> Template
        cup_of_tea:                           -> Template
        hour_glass:                           -> Template
endtype
 endspec
```