Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

D.A. Duce, R. van Liere, P.J.W. ten Hagen

Components, frameworks and GKS input

Computer Science/Department of Interactive Systems Report CS-R8947 November

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

D.A. Duce, R. van Liere, P.J.W. ten Hagen

Components, frameworks and GKS input

Computer Science/Department of Interactive Systems Report CS-R8947 November

---

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 Research (N.W.O.).

Components, Frameworks and GKS Input

D. A. Duce†, R. van Liere‡, P.J.W. ten Hagen‡

† *Rutherford Appleton Laboratory, Chilton, Didcot, OXON, U.K.*
‡ *CWI, Amsterdam, The Netherlands*

This paper was inspired by the Components/ Frameworks approach to a Reference Model for computer graphics, currently under discussion in the ISO computer graphics subject committee. The paper shows how a formal description of the GKS input model may be given in Hoare's CSP notation and explores some extensions in which some of the components in the GKS model are replaced by more interesting ones. The paper thus demonstrates some of the power and flexibility inherent in the Component/ Frameworks idea. The use of a formal notation led to a deepening of the authors' understanding of the input model and suggested some different ways of looking at the input model.

1980 Mathematics Subject Clasification : 69K32
1983 CR Categories : I.3.2
Key Words & Phrases : Graphics systems, formal descriptions, CSP, Component/Frameworks.

Note : This report will be submitted for publication elsewhere.

1. Introduction

This paper explores the application of a particular formal description technique, Hoare's Communicating Sequential Processes (CSP)[1] to the description of graphical input in the current generation of graphics standards, in the context of the Components and Frameworks [2],[3] approach to reference models for computer graphics.

This paper starts with an overview of the Components/ Frameworks idea followed by an overview of CSP. The next section describes the GKS input model in CSP and the following section gives some examples of how the model can be generalized. Although some of these ideas have been presented before in ISO working documents, the formulation given here is more general and more elegant, as a result of the structure of the formal description. The use of a formal description technique here has suggested new ways of presenting the GKS input model and has also suggested equivalences between the operating modes in GKS which were not previously apparent.

2. Overview of Components/ Frameworks

At the first plenary meeting of the new ISO/IEC subcommittee responsible for computer graphics, ISO/IEC JTC1/SC24, the need to review the work of its predecessor committee and to plan for the future development of graphics standards was recognized. SC24 authorised the formation of a Special Working Group to recommend a five-year strategic plan for organizing the work of SC24.[4] The Special Working Group met at Blakeney in the U.K. in April 1988. The major recommendation was that the next generation of computer graphics standards should be based firmly on a Reference Model which could identify demarcations and

Report CS-R8947
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

resolve disputes between standards. The Special Working Group also recommended a new approach to the development of standards, called the "Component/ Framework Process". Inherent in this process was a model of graphics standards which sees a graphics standard as constructed from a collection of components set in a framework. Components would include datatypes and operations. A framework is the "glue" which joins components together to form a system and performs management concerned with display and control. This model was seen as promoting harmonization between standards through the use of common components and frameworks. This idea is illustrated in Figure 1 below. Standards A, B and C each have their own frameworks. Some components are used by more than one standard, others by only one standard.

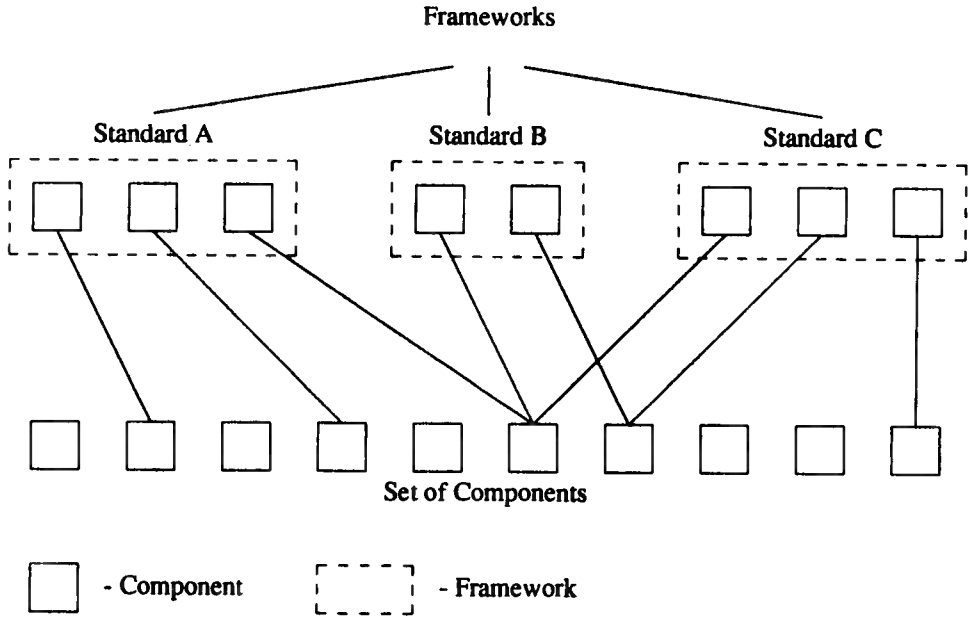

Figure 1: Components and Frameworks.

The relationship between the PHIGS standard and the PHIGS+ proposal illustrates this idea in that PHIGS and PHIGS+ share a common framework, but differ in their choice of output primitive component and attribute component. PHIGS+ uses a richer set of components which take illumination into account. PHIGS and PHIGS+ also use the same input components.

Immediately after the Blakeney meeting, the BSI held a Reference Model meeting which produced an approach to Reference Models combining the merits of the Component/ Framework approach and an earlier BSI approach to Reference Models. The new BSI approach[2] essentially arose from the recognition of the parallels between components and abstract data types. This work drew heavily on the work of Arnold and Reynolds concerning configurable models of graphics systems and their work with Duce in the formal specification of a GKS-like output pipeline.[56] The ideas were subsequently explored further at the meeting of SC24/WG1 held in Tucson, U.S.A. in July 1988. At this meeting consideration was given to how a functional standard might be expressed in a component/ frameworks setting and how input might be treated in this way. The first (without input) proved fairly straightforward at a fairly high level of abstraction, the second proved more demanding, in part because of the lack of a suitable notation in which to describe components.

This paper is the result of work done since that meeting by the authors to explore one particular notation, which appears to offer considerable promise for expressing the components required to describe the GKS input model. The technique used is Hoare's CSP notation. A short description of this notation follows.

3. Communicating Sequential Processes

Hoare motivates CSP by a discussion of objects in the world around us which act and interact with each other in some characteristic ways.[1] The aim of CSP is to describe this characteristic behaviour. The starting point for this is to decide what kinds of events or actions will be of interest and then to choose a different name for each kind.

As an illustration, consider a simple one-place buffer. Let the event *write* correspond to the user writing a value to the buffer and *read* correspond to reading the value of the buffer. Each name actually denotes an event class, there may be many occurrences of events in a single class, separated in time.

The sets of names of events which are considered relevant to the particular description of an object is called its *alphabet*. An object cannot engage in events outside its alphabet, but the presence of a name in an object's alphabet does not imply that the object will eventually engage in that event.

In CSP occurrences of events are regarded as instantaneous or atomic actions without duration. Extended or time consuming operations can be represented by pairs of events, one denoting its start, the second its finish. During the interval between start and finish other events may occur. Time is ignored in the basic CSP model and consequently it is not meaningful to ask if one event occurs simultaneously with another. When simultaneity is important (e.g. in synchronization), it is represented as a single-event occurrence. When it is not, potentially simultaneous events are allowed to occur in any order.

CSP also does not distinguish between events initiated by an object and those initiated by some agent outside the object. This avoidance of the concept of causality leads to considerable simplification of the theory and its application.

The word *process* is used to stand for the behaviour pattern of an object. There is a convention in CSP that events are denoted by lower case words and processes by upper case words. Let x be an event and P be a process. Then the prefix notation

$$(x \rightarrow P)$$

describes an object which first engages in event x and then behaves exactly as denoted by P. This notation can be used to describe the entire behaviour of a process that eventually stops. However, for objects which continue to act and interact with their environment for as long as they are needed, and which contain repeating patterns of behaviour, this is not a convenient notation.

Consider a simplification of the 1-place buffer, an unchanging storage location which can be read. Denote the object by B, then the alphabet of B is

$$\alpha B = \{ read \}$$

An object which behaves like B after being read once would be described by

$$(read \rightarrow B)$$

The behaviour of this object is exactly like the original object B, which suggests a formulation

$$B = (read \rightarrow B)$$

This can be regarded as an implicit definition of the behaviour of B. The potentially unbounded behaviour of B is effectively defined as

$$read \rightarrow read \rightarrow read \rightarrow read \rightarrow \ldots$$

This method of process description only works if the right hand side of the equation starts with at least one event prefixed to all recursive occurrences of the process name. A process description beginning with a prefix is said to be *guarded*. In the specifications following, it is sometimes necessary to refer to the process which satisfies (i.e. is the solution of) such an equation. If $F(X)$ is a guarded expression containing the process name X, then it can be shown that the equation

$$X = F(X)$$

has a unique solution. This solution is denoted by

$$\mu X: F(X)$$

X is a bound variable whose name can be changed at will. In the example above, the solution of the recursion equation for B is

$$B = \mu X: (read \rightarrow X) = \mu Y: (read \rightarrow Y)$$

Many objects, including the one-place buffer with which this discussion started, allow their behaviour to be influenced by interaction with the environment in which they are placed. If x and y are distinct events,

$$(x \rightarrow P \mid y \rightarrow Q)$$

denotes a process which initially engages in either of the events x or y. After the first event has occurred, the process behaves as P if the first event was x or as Q if the event was y.

A description of a one-place buffer which first engages in a *write* event and then subsequently in either *read* or *write* events is

$$B = (write \rightarrow \mu X: (read \rightarrow X \mid write \rightarrow X))$$

The choice of which event will actually occur can be controlled by the environment within which the process evolves. The environment of a process may itself be described as a process. The complete system is itself a process whose behaviour is definable in terms of its component processes.

When two processes are brought together to evolve concurrently, it is usually intended that they should interact with each other. These interactions can be regarded as events in which both processes participate. The one-place buffer described above might be placed in the context of an application program which will alternately write and then read the buffer. Such a program can be described by the process AP

$$AP = write \rightarrow read \rightarrow AP$$

The notation

$$AP \parallel B$$

denotes the process which behaves like the system composed of the two processes AP and B interacting in synchronization as described.

When processes P and Q with differing alphabets are combined to run concurrently, events that are in both their alphabets require simultaneous participation of P and Q. However, events in the alphabet of P which are not in the alphabet of Q are no concern of Q. Such events may occur independently of Q whenever P engages in them. Similarly, Q may engage alone in events which are in the alphabet of Q but not of P. Examples of this will be seen in the GKS input model specifications following.

CSP introduces special notation for a particular class of events called *communications*. A communication is an event that is described by a pair

$$c.v$$

where c is the name of a channel on which communication takes place and v is the value of the message which passes. The set of all messages which P can communicate on channel c is defined

$$\alpha c(P) = \{ v \mid c.v \in \alpha P \}$$

If v is a member of $\alpha c(P)$, a process which first outputs v on channel c and then behaves like P is defined

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

The only event in which this process is initially prepared to engage is the communication event $c.v$.

A process which is initially prepared to input any value x which can be communicated over the channel c, and then behave like $P(x)$ is defined

$$(c?x \rightarrow P(x)) = (y: \{y \mid channel(y)=c\} \rightarrow P(message(y)))$$

In the one-place buffer example given earlier, a more complete description would be

$AP = write!v \rightarrow read?v \rightarrow AP$
$B = write?v \rightarrow B_v$
$B_v = read!v \rightarrow B_v \mid write?v \rightarrow B_v$
$B \parallel AP$

Notice also here the use of subscripts to indicate the value of the state associated with a process.

4. The GKS Input Model

4.1. Introduction

The GKS input model is based on the concept of logical input devices, providing the application program with an interface which abstracts physical input devices from a particular hardware configuration. The paper by Rosenthal et al[7] gives a detailed exposition of the GKS input model.

Logical input devices are described in terms of a *class*, *operating modes* and *attributes*. These are described below.

Conceptually, logical input devices are explained in terms of two processes, the *measure process* and *trigger process*.

Classes.

The class of a logical input device defines the type of the input value which is returned. The six logical input data types are:

1. *LOCATOR:* a position in world coordinates and the associated number of the normalization transformation used to convert back from device coordinates via normalized device coordinates to world coordinates.
2. *STROKE:* similar to LOCATOR except it represents a sequence of world coordinate positions rather than a single position.
3. *VALUATOR:* a real number in some range.
4. *CHOICE:* an integer representing a selection from a set of choices.
5. *PICK:* the name of a selected segment and an identifier indicating which set of primitives in the segment has been picked.
6. *STRING:* a character string.

A particular measure value of a logical input device is defined to be the value of the physical input device transformed by a measure mapping function. GKS does not place any constraints on the realization of logical input devices in terms of physical devices. A CHOICE device could be realized by a keyboard and the operator has to type in the name of the menu item to be selected. In this case the physical input value (the string) is mapped to the corresponding CHOICE device value (integer selection number) by the measure process. The following table shows a possible relationship between strings typed and the value of the CHOICE logical input device.

""	NOCHOICE
"CREATE"	1
"REDRAW"	2
"DELETE"	3
"RETURN"	4

The measure process will always contain the current measure value of the logical input device. Usually, the measure value is echoed in some way on the screen (for instance, by echoing a cursor shape in the position that corresponds with the measure value).

How the measure value is mapped onto a value returned by a logical input device is defined differently for every input class.

Operating modes.

The operating mode indicates how the input value is obtained from the logical input device. The trigger process plays an important role in this. A trigger process is an independent, active process which for certain operating modes, when *triggered* by the user, indicates that the current measure value is to be returned to the application program.

There are three different operating modes:

- *REQUEST*. Logical input devices in REQUEST mode behave rather like FORTRAN READ. A request is made by the application program for a measure to be returned from a specified device. GKS waits until the operator has set the measure to the desired value and has activated the trigger.
- *SAMPLE*. In SAMPLE mode the current measure is returned whenever requested by the application program. No triggering is involved when a logical device is sampled so that the application program will immediately continue after issuing a sample call.
- *EVENT*. A number of input devices may be active together. Each time the trigger for a particular device is activated, the current measure value and data identifying the device are added to a single queue of input events for all the devices used in event mode. The application program can interrogate the queue to retrieve the input events. It is possible to couple more than one input device to the same trigger so that multiple events can be generated from a single trigger event.

  The event queue is structured as a queue of event reports. The event queue is interrogated by the GKS function AWAIT EVENT. This function removes the event report at the top of the queue, writes it into a buffer known as the *current event report* and returns the identification of the device which produced the report (workstation identifier, input class, logical input device number) to the application program. If the queue is empty, GKS is suspended until either input arrives (in which case the function behaves as before) or a timeout period expires (in which case input class NONE is returned), whichever happens first. GKS provides a set of functions, one for each device class, which return the logical input value contained in the current event report.

Attributes.

Attributes are used to parameterize the initialization of a logical input device. Most attributes have to do with how and where input devices produce echos on the screen. Attributes include initial values, prompt / echo types, activation modes and echo areas. Data records provide the application program a means to parameterize the logical input device in a device dependent manner. For instance, an entry in the data record can specify which mouse button will be used to trigger a locator device.

4.2. Structure of the Specification

This section describes the overall structure of the CSP specification of the GKS input model. The following sections elaborate the specification in detail. Figure 2 illustrates the overall structure.

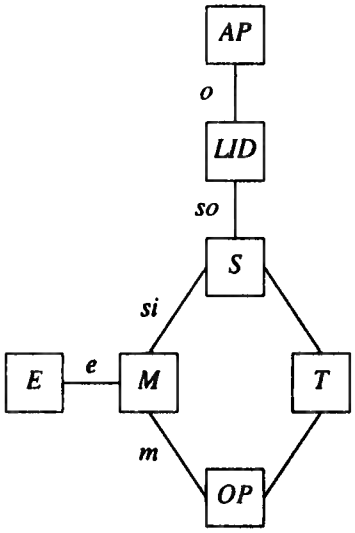

Figure 2: Structure of the Specification

The first attempt at a CSP specification of GKS input used different structures to describe each of the operating modes. By examining the resulting specifications; it was realized that each mode could be described in terms of five processes (*LID*, *S*, *E*, *M* and *T*). The following specifications are therefore presented within this framework; some of the components which populate this framework have different descriptions in the different operating modes.

The specification starts from the realization that there are three key objects, the application program, the operator, and the logical input device which is the connection between the two. For simplicity we will only consider a single logical input device here. The application program and the operator provide the environment for the logical input device and to describe the system as a whole it is convenient to describe also what actions the operator may perform and what functions the application may invoke. These are all modelled as events in the alphabet of the CSP processes describing the application (*AP*) and the operator (*OP*).

The process *LID* describes the behaviour of a logical input device in terms of a measure process *M*, an echo process *E*, a trigger process *T* and a storage process *S*. It was the realization that each operating mode could be described in terms of processes of these kinds which led to the model presented here. In the case of SAMPLE mode the trigger process is null. It will be seen that the echo and measure processes are the same in all the operating modes. The different modes present different opportunities to the operator and application and have different storage components.

Each of the processes will now be discussed in turn for each of the operating modes. Section 4.7 discusses how this generalizes to the case of more than one device.

4.3. Application Program (AP)

The application program may set the mode of a logical input device and invoke functions appropriate to that mode. Some slight simplifications are made to the GKS model. GKS allows the application to set an initial value for a device in the appropriate workstation statelist. Here initialization is not considered as it is essentially orthogonal to the remainder of the specification. Secondly, when a device is placed in SAMPLE or EVENT modes, measure and trigger (in the case of EVENT mode only) processes are created immediately for the device and it becomes active. When a device is set into REQUEST mode, the measure and trigger processes are not created until the REQUEST <device class> function is invoked. Essentially the device is in a quiescent state until this latter function is invoked. The specification given here reflects this by explicitly introducing a *set–quiescent–mode* event.

The behaviour of the application program is described by the process:

$$AP = set\text{-}mode\text{-}quiescent \rightarrow AP$$
$$\quad | (request \rightarrow REQUEST \,||\, E_0 \,||\, M_0 \,||\, T \,||\, OP^R \,||\, S^R \,||\, LID^R)$$
$$\quad | (set\text{-}mode\text{-}sample \rightarrow SAMPLE \,||\, E_0 \,||\, M_0 \,||\, OP^S \,||\, S^S \,||\, LID^S)$$
$$\quad | (set\text{-}mode\text{-}event \rightarrow EVENT \,||\, E_0 \,||\, M_0 \,||\, T \,||\, OP^E \,||\, S^E_{<>} \,||\, LID^E)$$

This specification states that the logical input device can be set into the quiescent mode, REQUEST mode, SAMPLE mode or EVENT mode. The *request* event corresponds to an invocation of the GKS REQUEST <device class> functions. From quiescent mode it can be set into any of the other modes. From the other modes, the behaviour is described by a composition of control (*REQUEST*, *SAMPLE*, *EVENT*), storage, measure, trigger echo and operator processes. Superscripts are used to denote processes which are different for the different modes. Subscripts are used to denote the initial states of processes.

4.4. REQUEST Mode.

4.4.1. The operator process, OP^R.

The operator of a logical input device can change the value of the device's measure process or fire the trigger process. The trigger firing can be represented as an event *trigger* and setting a new measure value by outputting a value v on channel m. The behaviour of the operator is then characterized by the process OP^R defined as:

$$OP^R = (m!v \rightarrow OP^R) \,|\, (trigger \rightarrow STOP_{OP^R})$$

This means that the operator can choose to change the value of the measure or fire the trigger. Once the trigger has fired, the interaction with that device terminates.

4.4.2. The application program, *REQUEST*.

The application program can receive a logical input value from a channel o. The interaction with the device then terminates and the device returns to the quiescent state. The behaviour of the application program in REQUEST mode is described by the process:

$$REQUEST = o?v \rightarrow AP$$

4.4.3. The measure process, M.

The behaviour is described by:

$$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \,|\, (get\_m \rightarrow si!v \rightarrow M_v)$$

The communication over channel m corresponds to the operator setting a new measure value, which is then transmitted to the echo process over channel e. The logical input device may request the current measure value by the event *get_m*. The value is returned along channel *si*. The special value 0 (process M_0) denotes the measure process initialized to the initial measure value recorded in the workstation state list.

Strictly speaking, the value communicated over channel m from the operator is a *physical* input value. The value associated with the state of the measure process and communicated along channel *si* is a logical input value. If f denotes the physical to logical value mapping, the first choice in the behaviour above could be written as :

$$M_v = (m?v' \rightarrow e!f\,(v') \rightarrow M_{f(v')})$$

This also makes it clear that it is the logical rather than the physical value which is echoed (see 4.4.5). The measure process is the point in the specification where the physical to logical mapping occurs. For example, if a keyboard is used to implement a CHOICE device, f might be the function:

$$f = \{\ \text{""} \rightarrow \text{NOCHOICE, "CREATE"} \rightarrow 1, \text{"REDRAW"} \rightarrow 2, \text{"DELETE"} \rightarrow 3, \text{"RETURN"} \rightarrow 4\ \}$$

The mapping could be specified precisely using a notation such as Z[8] in combination with CSP,[9] but that would take us rather beyond the scope of this present paper. The intention should, however, be clear.

4.4.4. The trigger process, T.

The behaviour of the trigger process is just :

$$T = trigger \rightarrow trigger_s \rightarrow T$$

The event *trigger* is shared by the operator and the trigger process. The event $trigger_s$ is shared by the trigger and storage processes.

4.4.5. The echo process, E.

The echo process can receive a value on channel e (from the measure process), and echo it on the display.

$$E_v = e?v' \rightarrow E_{v'}$$

The special value 0 (process E_0) denotes the echo process which echoes the initial measure value recorded in the workstation state list.

4.4.6. The logical input device, LID^R.

The logical input device reads a logical input value from the storage channel, *so*, and delivers it to the application program on channel *o*. The behaviour is defined by:

$$LID^R = get\_s \rightarrow so?v \rightarrow o!v \rightarrow LID^R$$

4.4.7. The storage process, S^R.

The storage process is initiated by the event *get_s*, awaits the trigger firing event $trigger_s$, initiates the event *get_m* to get the current value of the measure process, then reads the current value of the measure on channel *si* and transmits this value to the LID^R process on channel *so*. This behaviour is defined by:

$$S^R = (get\_s \rightarrow trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^R)$$

The storage process is effectively providing a one-place buffer between the measure process and the application program. The value of the measure process transmitted to the application is the value current when the trigger fires.

4.4.8. Remarks on REQUEST mode.

- The table below shows the alphabets of each of the processes in the specification. The left hand column lists all the possible event classes. An '×' underneath a process indicates that the corresponding event is in the alphabet of that process.

	OP^R	*REQUEST*	*M*	*T*	LID^R	*E*	S^R
m.v	×		×				
trigger	×			×			
o.v		×			×		
e.v			×			×	
get_m			×				×
si.v			×				×
$trigger_s$				×			×
get_s					×		×
so.v					×		×

- The application process, *REQUEST*, does not exhibit any choice. The choice in the system is made by the operator, who can choose when to vary the measure and when to fire the trigger. This is shown clearly in the specification.
- The specification also shows clearly that the device is put into REQUEST mode by the application. Once the trigger has fired, a logical input value is returned to the application program and interaction with the device ceases until it is put into REQUEST mode again by the application program at which point a new measure process is created.

4.5. SAMPLE Mode.

4.5.1. The operator process, OP^S.

The operator of a logical input device can only change the value of the device's measure process. The behaviour of the operator is then characterized by the process OP^S defined as:

$$OP^S = (m!v \rightarrow OP^S)$$

4.5.2. The application program, *SAMPLE*.

The application program can engage in three events, *set–mode–quiescent* which returns the device to the quiescent state, *sample* requesting a logical input value from the device, and receiving a logical input value from it on channel *o*. The behaviour of the application program is described by the process:

$$SAMPLE = (\mu X : sample \rightarrow o?v \rightarrow X) \mid (set\text{–}mode\text{–}quiescent \rightarrow AP)$$

Notice that once the device is in SAMPLE mode, the application program can sample the device any number of times before returning it to the quiescent state.

4.5.3. The measure process, M.

The measure process is exactly the same as for REQUEST mode. The behaviour is described by:

$$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \mid (get\_m \rightarrow si!v \rightarrow M_v)$$

4.5.4. The echo process, E.

The echo process is exactly the same as for REQUEST mode. The echo process can receive a value on channel *e*, and echo it on the display.

$$E_v = e?v' \rightarrow E_{v'}$$

4.5.5. The logical input device, LID^S.

The application can sample the logical input device (*sample*), reading a logical input value from the storage process and delivering it to the application program on channel *o*. The behaviour is described by:

$$LID^S = sample \rightarrow get\_s \rightarrow so?v \rightarrow o!v \rightarrow LID^S$$

The event *sample* is also contained in the alphabet of the process *SAMPLE*. The description of SAMPLE input differs from that of REQUEST input because in the latter the device reverts to the quiescent state after one value has been returned to the application program, whereas in SAMPLE input the device remains active and may be sampled any number of times before being explicitly returned to the quiescent state.

4.5.6. The storage process, S^S.

The storage process is similar to the process S^R in REQUEST mode, except that there is no involvement of the trigger process. The value delivered is the value current when the application invokes the sample function.

$$S^S = (get\_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^S)$$

4.5.7. Remarks on SAMPLE mode.

- The table below shows the alphabets of the processes in this specification. Note that the trigger process is not used in the specification.

	OP^S	SAMPLE	M	LID^S	E	S^S
m.v	×		×			
o.v		×		×		
e.v			×		×	
get_m			×			×
si.v			×			×
get_s				×		×
so.v				×		×
sample		×		×		
set–mode–quiescent		×				

- Figure 3 illustrates the structure of this specification. It can be seen that this is just a special case of the general case shown in Figure 2.

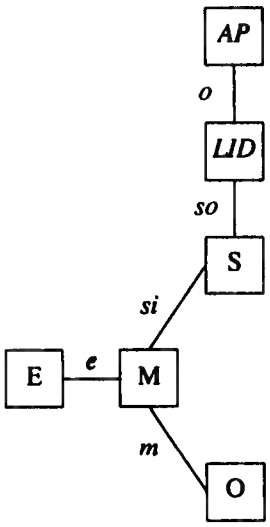

Figure 3: Structure of the SAMPLE Specification

- The application program can decide when to sample the device, that is when to invoke the SAMPLE <device> function (event *sample*). The operator can alter the value of the device's measure, but that is the only option offered to the operator.
- The device remains in SAMPLE mode until the application chooses to return it to the quiescent state. In GKS the application would set the device directly into one of the other operating modes or reinitialize the device in SAMPLE mode. Here we select a new mode in two stages, returning first to the quiescent state before selecting the new mode. This gives a tidier specification.
- The measure process in SAMPLE mode is identical to the measure process in REQUEST mode. The form of the storage processes in these two modes shows clearly the role of the trigger in REQUEST mode input.

4.6. EVENT Mode.

4.6.1. The operator process, OP^E.

The operator of a logical input device in EVENT mode can change the value of the device's measure process or fire the trigger process. The trigger firing can be represented as an event *trigger* and setting a new measure value by outputting a value v on channel m. The behaviour of the operator is then characterized by the process OP^E defined as:

$$OP^E = (m!v \rightarrow OP^E) \mid (trigger \rightarrow OP^E)$$

4.6.2. The application program, *EVENT*.

The application program can engage in three events, *set–mode–quiescent* which returns the device to the quiescent state, *await–event* requesting a logical input value from the storage and receiving a logical input value from it channel *o*. The behaviour of the application program is described by the process:

$$EVENT = \mu X.\ (await\_event \rightarrow o?v \rightarrow X) \mid (set–mode–quiescent \rightarrow AP)$$

4.6.3. The measure process, M.

This is identical to the measure process in SAMPLE and REQUEST modes. The behaviour is described by:

$$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \mid (get\_m \rightarrow si!v \rightarrow M_v)$$

4.6.4. The trigger process, T.

This is identical to REQUEST mode.

$$T = trigger \rightarrow trigger_s \rightarrow T$$

4.6.5. The echo process, E.

The echo process can receive a value on channel *e*, and echo it on the display. This is identical to REQUEST and SAMPLE modes.

$$E_v = e?v' \rightarrow E_{v'}$$

4.6.6. The logical input device, LID^E.

The logical input device reads a logical input value from the storage channel, *so*, and delivers it to the application program on channel *o*. The behaviour is described by:

$$LID^E = await\_event \rightarrow get\_s \rightarrow so?v \rightarrow o!v \rightarrow LID^E$$

There is a similarity with the process LID^S in that the device remains active when await event has completed.

4.6.7. The storage process, S^E.

This process represents the major difference between EVENT mode and REQUEST and SAMPLE modes. In the other two modes the storage process does not retain values. In EVENT mode, trigger firing results in values being sent to the storage process. Their consumption awaits a *get_s* event from the logical input device.

In GKS, the storage discipline is a queue. Values are added to one end of the queue by *get_m* and removed from the other by *get_s*. Subscripts to the process name are used to indicate the state of the queue before and after each event which modifies the queue.

The AWAIT EVENT function in GKS returns a NONE value to the application program if the queue is empty when the function is invoked and no input is added to the queue before a timeout period has expired. Timeout is indicated in this model by the event *time_out*. It is not further defined here.

In GKS the AWAIT EVENT function returns the identification of the device from which the event at the top of the queue originated and moves this event description to the current event report. Events are retrieved from the current event report by GET <device class> functions, one for each type of device. In this specification, the current event report is not described, it is merely stated that the logical input value is returned to the application program through channel *o*.

$S^E_{s<v>} = (\ get\_s \rightarrow so!v \rightarrow S^E_s)$
$S^E_s = (\ trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>s})$
$S^E_{<>} = (\ get\_s \rightarrow (\ time\_out \rightarrow so!NONE \rightarrow S^E_{<>} \mid trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^E_{<>}))$
$\quad \mid (\ trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>})$

4.6.8. Remarks on EVENT mode.

- The table below shows the alphabets of the processes in the specification of EVENT mode input.

	OP^E	*EVENT*	*M*	*T*	LID^E	*E*	S^E
m.v	×		×				
trigger	×			×			
o.v		×			×		
e.v			×			×	
get_m			×				×
si.v			×				×
$trigger_s$				×			×
get_s					×		×
so.v					×		×
set-mode-quiescent		×					
await-event		×			×		
timeout							×

- The application process *EVENT* is very similar to the corresponding process for SAMPLE mode input. The application can decide when to ask for an input value. In the EVENT case however, the storage component does not immediately request the current value of the measure; instead it looks to see if a value is stored or if not, awaits the arrival of a new event until a timeout expires.
- The measure and trigger processes are identical to those in REQUEST mode. The operator process is similar except that in EVENT mode a trigger firing does not terminate the interaction with the device, so the operator may generate more than one value. It is an application program action which terminates the device. The difference in behaviour is accounted for by the different storage components in the two systems and the different application process behaviours.

4.7. Multiple Devices

Although the specification given here only considers a single logical input device, multiple devices can easily be described by amending the description of the process *AP* and introducing clones of the other processes, with appropriate names (for example prefixed by the device name which is unique). If the devices are independent, then the event names need to be prefixed by the device name to make them unique. If the devices are not independent, for example if two devices share the same trigger, then events which are common have the same name. Recall that in CSP events which are in the alphabets of two processes require their simultaneous participation. Thus in this example, the trigger firing would automatically go to both devices because it is in the alphabets of the operator and device processes.

5. Extensions

5.1. Introduction.

In this section some simple extensions to the input model are discussed. Essentially these involve replacing components in the framework described here. The extensions to be discussed are logical input device types and composite devices, storage strategies and the interaction between input and output.

5.2. New device types.

As noted earlier, the GKS input model defines six classes of logical input device, each corresponding to a particular type of input value. The type of the logical input value associated with the logical input device has not featured at all in the specification given here. In fact the specification is completely independent of the type of the input value, provided that the type is consistent throughout the specification. This means that new classes of logical input device can be introduced very easily, merely by substituting components which can handle the new datatype. No change is needed to the specification to describe such systems.

In EVENT mode input, GKS allows any particular trigger to be associated with more than one measure process. Then when the trigger fires, multiple event reports are added to the queue and marked as simultaneous events. There is no analogue of this mechanism in SAMPLE or REQUEST modes. This restriction is unfortunate because this mechanism provides a nice way to accommodate devices such as the Tektronix cross-hairs and the mouse, each of which can be viewed as a composite of a LOCATOR and CHOICE device.

Such devices can be incorporated into this specification fairly easily. There are two ways to do this, the first involves generalizing the framework given here to incorporate multiple measure processes, one for each of the basic measure types which make up the device. A slightly more elegant way to do this comes from the recognition that there is one measure value associated with the device, which happens to be composed to two basic types, LOCATOR and CHOICE. We will illustrate this with the mouse device. Suppose we have a three-button mouse. The value of the device can be expressed as a LOCATOR and CHOICE value, as an ordered pair of the form (l, c), where l is of type LOCATOR, indicating the position of the device, and c is of type CHOICE, indicating which, if any, of the buttons are depressed. The specification of such a device can be obtained from that given here by consistently substituting (l, c) for v throughout the specification.

Note that the operator now generates events of the form

$$m!(l, c)$$

so the operator has been implicitly retrained! Notice that this works perfectly well in SAMPLE and REQUEST modes as well as EVENT mode and that the notion of simultaneous events has been replaced by the simpler notion of cartesian product datatypes. All we need is a way of delivering values of this type to the application program. If the only mechanism for doing this in a particular programming language involves a notion of simultaneous events, then this should not be cluttering the specification for more flexible programming languages.

The three buttons on the mouse might also act as triggers for the device. Suppose the events $trigger_1$, $trigger_2$, $trigger_3$, denote the action of depressing the respective buttons. If each of these can trigger REQUEST or EVENT input, we can describe this by substituting a process T with the following description:

$$T = (\, trigger_1 \rightarrow trigger_s \rightarrow T \,) \mid (\, trigger_2 \rightarrow trigger_s \rightarrow T \,) \mid (\, trigger_3 \rightarrow trigger_s \rightarrow T \,)$$

5.3. Storage strategies.

The second extension we discuss here concerns storage strategies. It has been seen that the form of the process S plays an important role in determining the overall system behaviour. Interesting systems can be obtained by taking the framework given here and substituting a different storage component. A simple example will illustrate the point. In GKS EVENT mode input, the storage strategy is a queue. For whatever reason, one might want to use a last-in-first-out strategy or stack, instead. A component to do this has a very simple description:

$$\begin{aligned}
S^E_{s<v>} &= (get_s \rightarrow so!v \rightarrow S^E_s) \\
S^E_{<>} &= (\, get_s \rightarrow (\, time_out \rightarrow so!NONE \rightarrow S^E_{<>} \mid trigger_s \rightarrow get_m \rightarrow si?v \rightarrow so!v \rightarrow S^E_{<>} \,)) \\
&\quad \mid (\, trigger_s \rightarrow get_m \rightarrow si?v \rightarrow S^E_{<v>} \,) \\
S^E_s &= (\, trigger_s \rightarrow get_m \rightarrow si?v \rightarrow S^E_{s<v>})
\end{aligned}$$

This is a simple example but hopefully it illustrates the point that by changing components, usefully different functionality can be obtained.

5.4. The relationship between input and output.

A workstation display surface can be represented as a process D, which can accept display events generated by the application program (through the output pipeline of the graphics system). This behaviour could be described by:

$$D_p = (display?p' \rightarrow D_{p\&p'})$$

Here the process D is receiving communications along channel *display*. The values passed are rendered output primitives, denoted by the variable p'. The primitive p' is combined with the existing state of D to yield a new state $p\ \&p'$ which represents the display space incorporating the new output primitive. The operator '$\&$' is not further defined here.

The echo process also generates graphical output, and the interaction between the echo and display processes could be expressed by:

$$D_p = (display?p' \rightarrow D_{p\&p'}) \mid (echo?p' \rightarrow D_{p\&p'})$$
$$E = (e?v' \rightarrow echo!r(v') \rightarrow E)$$

Here the function r generates the rendered primitive corresponding to the current measure value v'. In reality the operations necessary to update an echo are more complicated than those given here, but the above behaviours should give a flavour for how this approach could be used.

Following on from this, it is clear how interactions between other processes in the input model and the display process could be described. It should also be clear that if the output pipeline is described in a similar manner, then interactions between input and output can be readily described; for example the echo process might interact with the output pipeline at a higher level using facilities in the output pipeline to construct the graphical object representing the echo. It should also be clear that this approach could be used to describe systems in which values derived from input devices are used to control the graphical output, for example to determine the parameters of transformations, by introducing appropriate communications. This suggests some interesting directions for further work.

6. Conclusions

This paper has demonstrated how a Component/ Frameworks style description of the GKS input model can be given in the CSP notation. The specification clearly demonstrates how it is possible to substitute components within a framework and some simple interesting extensions to the model have been described.

The exercise has deepened the authors' understanding of the input model, and it is hoped the readers' also, by demonstrating the role of the storage component in each of the input modes and showing precisely where, and by whom, choices may be made in each of the operating modes.

7. Acknowledgements

The authors are grateful to colleagues in ISO/IEC JTC1/SSC24/WG1 for fruitful discussions on how to describe input in a component framework setting, in particular to Tom Morrissey.

References

1. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, London (1985).
2. ISO, "Contributions on a Computer Graphics Reference Model," ISO / IEC JTC1 / SC24 N4, ISO Central Secretariat (1988).
3. ISO, "The Use of the Component / Framework Description Technique in the Specificaiton of Computer Graphics Standards," ISO / IEC JTC1 / SC24 N178, ISO Central Secretariat (1988).
4. G.S. Carson, "The Future of ISO Graphics Standards," *IEEE Computer Graphics and Applications*(7), pp. 82-83 (1988).
5. D.B. Arnold, G. Hall, and G.J. Reynolds, "Proposals for Configurable Models for Graphics Systems," *Computer Graphics Forum* 3(3), pp. 201-208 (1984).
6. D.B. Arnold, D.A. Duce, and G.J. Reynolds, "An Approach to the Formal Specification of

Configurable Methods of Graphics Systems,'' pp. 439-463 in *European Computer Graphics Conference and Exhibition*, ed. G. Marechal, North-Holland, Amsterdam (1987).

7. D. Rosenthal, J. C. Michener, G. Pfaff, R. Kessener, and M. Sabin, ''The detailed semantics of graphics input devices,'' *Computer Graphics* **16**(3), pp. 33-38 (July 1982).
8. I. Hayes, *Specification Case Studies*, Prentice-Hall International, London (1987).
9. J.C.P. Woodcock, ''A Strategy for the Correct Implementation of Communicating Processes,'' *Le Premier Seminaire International sur le Genie Logiciel*, Oran, Algeria (1988).