Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

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

An approach to hierarchical input devices

Computer Science/Department of Interactive Systems Report CS-R8946 November

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

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

An approach to hierarchical input devices

Computer Science/Department of Interactive Systems Report CS-R8946 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.).

An Approach to Hierarchical Input Devices

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 shows how a formal description of the GKS input model can be extended to include hierarchically structured input devices.

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

It has been shown in[1] how a formal description of the GKS input model can be given using Hoare's CSP notation.[2] This paper considers an extension to the GKS input model which introduces hierarchically structured input devices, which is described using the same approach as that taken in.[1]

GKS logical input devices can be described in terms of a *class*, *operating modes* and *attributes*. The class defines the type of value which is returned. GKS recognizes six classes: LOCATOR, STROKE, VALUATOR, CHOICE, PICK and STRING. The operating mode determines how the input is obtained from the logical input device. Attributes are parameters which allow the application program some degree of control over the device, for example the type of prompting and echoing to be used.

The GKS input model is described in[3] in terms of six processes:

(1) measure;

(2) trigger;

(3) prompt;

(4) echo;

(5) acknowledgement;

(6) control.

The measure value of a logical input device is the type and value of the input to be returned to the application program. The measure mapping defines the relationship between input values from the physical input device by which the logical device is realized, and the measure value. The measure process maintains the current measure value of the logical input device as the operator manipulates the physical device.

The trigger is an event, which for certain styles of input determines when the measure value is returned to the application program.

The prompt indicates to the operator when the device is available for input, the echo process gives feedback of the current measure value to the operator and the acknowledgement process informs the operator of the value delivered to the application program.

Report CS-R8946
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The control process controls the operation of the device. Diagrammatically a logical input device can be represented as shown in Figure 1.

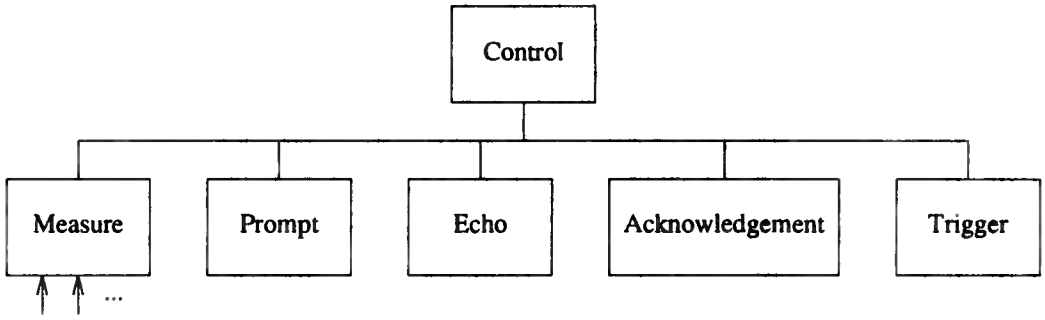

Figure 1

Logical input devices in GKS have 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.

The input subgroup at the Eurographics GKS Review Workshop[4] in 1987 spent considerable time discussing the role of logical input devices in interactive applications and arrived at the diagram shown in Figure 2. The model was that interactive techniques would use some form of composite input device, built up from logical input devices (although there might be more than one layer of composite devices) which in turn are constructed from measures and triggers which map onto physical devices. Each level in this structure has associated prompts, echo and acknowledgements which are realized using the facilities provided by the graphical output part of the graphics system.

Recent work within ISO on a reference model for computer graphics[5] and an Improved Input Model[6] is making some generalizations and simplifications to this model. The notion of logical input device traditionally applies to an input device at a certain level of abstraction. The point about logical input devices, is that they can be realized by a wide range of different physical devices and are not tied to a single physical device. The generalization that has been made now is to realize that the GKS model comprising the six processes listed above can be used to describe input devices at any number of different levels of abstraction and that a device at one level of abstraction may provide the measure or trigger processes for a device at a higher level of abstraction. Even physical input devices can be described with the notions of measure and trigger and what is thought of as a measure in GKS can be described as a logical measure which is constructed from physical measure values.

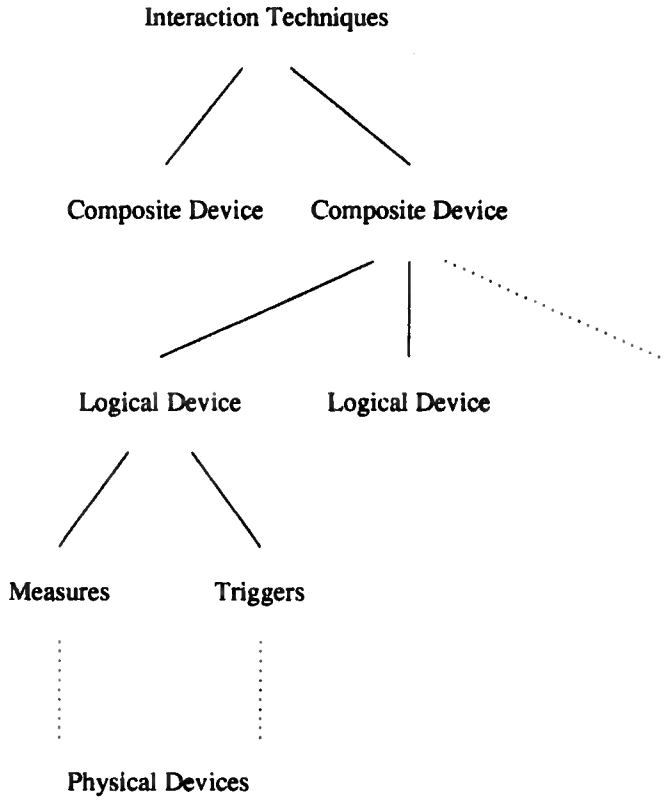

Figure 2

The remainder of this paper explores how such hierarchically structured input devices can be formally described. The aim is that the descriptions should fit within the same framework and behaviour patterns as was used for the single level of device in the GKS input specification.

For the remainder of this paper, the term *input device* will be used in preference to the term *logical input device* in recognition of the fact that input devices in the paper may be at many levels of abstraction in addition to that normally associated with logical input devices.

2. Review of the GKS Input Model Description

This section summarizes the formal description of GKS input contained in.[1] The description is given in Hoare's CSP notation, for a full explanation of which the reader is referred to Hoare's book.[2] A brief introduction to the key ideas is given here, which in conjunction with[1] should be adequate for an understanding of this paper.

Hoare, in the preface to his book, writes "The most obvious application of the new ideas is to the specification, design, and implementation of computer systems which continuously act and interact with their environment. The basic idea is that these systems can be readily decomposed into subsystems which operate concurrently and interact with each other as well as with their common environment. The parallel composition of subsystems is as simple as the sequential composition of lines or statements in a conventional programming language". The GKS input model would seem to lend itself naturally to expression in this language, describable as it is in terms of the subsystems listed in the introduction.

CSP sets out to describe the behaviour patterns of objects. The first step is to decide what kinds of event or action will be of interest, and to give a different name for each kind. Each event name then describes a *class* of events; there may be many occurrences of events in a single class, separated in time.

The set of names of events which are used to describe a particular object are called its *alphabet*. Selection of the alphabet focuses attention on the properties and actions of the object that are important and deliberately ignores events of lesser interest. For the specification to be presented here, prompting and acknowledgement are deliberately ignored and no events corresponding to these actions appear in any of the processes.

CSP regards occurrences of events as instantaneous or atomic actions without duration. The exact timing of events is also ignored in CSP; where timing concerns are important, these are to be treated separately from the logical correctness of the design. When simultaneity of a pair of events is important (as, for example, in synchronization), it is represented as a single event occurrence; when it is not, potentially simultaneous events are allowed to be recorded in either order.

In CSP, the behaviour pattern of an object is termed a *process*. If *trigger* is an event and T is a process, then:

$$(trigger \rightarrow T)$$

denotes an object that first engages in the event *trigger* and then behaves like T. The event *trigger* has to be in the alphabet of T.

This prefix notation can be used to describe the entire behaviour of an object that eventually stops; however, to use it in the way described above would be extremely tedious. Recursion can be used to give an implicit definition of the behaviour of an object. For example, an object T which first engages in the event *trigger* and then continues to behave exactly like T can be defined implicitly by the equation:

$$T = (trigger \rightarrow T)$$

A process description which begins with a prefix is said to be *guarded*. If $F(X)$ is a guarded expression containing the process name X, and A is the alphabet of X, then it turns out that the equation:

$$X = F(X)$$

has a unique solution with alphabet A, which it is sometimes convenient to denote by

$$\mu X{:}A.F(X)$$

where X is a local name which can be changed at will. The alphabet A is often omitted when it is clear from the content or context of the process. The process T above could be expressed in this notation as

$$\mu T.(trigger \rightarrow T)$$

Objects frequently exhibit choice in their behaviour through interaction with the environment. If $trigger_1$ and $trigger_2$ are distinct events,

$$(trigger_1 \rightarrow T_1) \mid (trigger_2 \rightarrow T_2)$$

describes an object which initially engages in either of the events $trigger_1$ or $trigger_2$. The behaviour following event $trigger_1$ is described by T_1 and that following $trigger_2$ by T_2. The choice between T_1 and T_2 is determined by the first event that actually occurs.

The process with alphabet A which never engages in any of the events of A is called $STOP_A$.

The next operator to be discussed is the concurrency operator '||'. When processes are combined to evolve concurrently, it is usually intended that they will interact with each other. The interactions can be thought of as events that require the simultaneous participation of the processes involved. Considering first the case of interaction, assume two processes with the same alphabet are combined, then each event that actually occurs must be a possible event in the independent behaviour of each process separately. The notation:

$$P \parallel Q$$

denotes the process composed of processes P and Q interacting in lock-step synchronization in this way.

If P and Q have different alphabets, then events that are in both their alphabets require the simultaneous participation of both P and Q. Events in the alphabet of P and not Q are of no concern to Q and hence can occur independently of Q whenever P engages in them. Similarly events in the alphabet of Q which

are not in the alphabet of P can be engaged in by Q independently of P.

The choice operator $(x{:}B \rightarrow P(x))$ defines a process which exhibits a range of possible behaviours depending on which event x from the set B occurs and the concurrency operator $\|$ permits some other process to make a selection between the alternatives in B. Whenever there is more than one event possible, the choice between them is made by the environment of the process. Such processes are called *deterministic*.

Sometimes the selection between alternatives is not influenced by the environment, but is made internally by the process in an arbitrary or *non-deterministic* fashion. The choice is not controlled by the environment. This kind of non-determinism arises from a deliberate decision to ignore the factors which influence the selection. It is used below in the description of the process *OP* which describes the behaviour of the operator of an input device in setting a new measure value or firing the trigger. A non-deterministic process is used because we deliberately choose to ignore the factors that lead the operator to make these selections. The notation

$$P \sqcap Q$$

denotes the process which behaves either like P or like Q. The operator

$$\prod_{x:S} P(x)$$

where S is a finite nonempty set, is a multiple choice operator which denotes the non-deterministic choice between the alternative events in the set S.

The final piece of notation to be introduced is used to describe the communication of a value between processes. A communication is an event described by a pair $c.v$ where c is the name of the channel over which the communication takes place, and v is the value of the message which passes. A process which first outputs v on the channel c and then behaves like P is denoted

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

and a process which is initially prepared to input any value x communicable on the channel c and then behave like $P(x)$ is denoted:

$$(c?x \rightarrow P(x))$$

By convention, channels are only used for communication in one direction and between only one pair of processes.

With this background, the CSP specification of the GKS input model contained in[1] excluding prompting and acknowledgement is now briefly reviewed. Logical input devices are described in terms of an operator process, *OP*, a measure, *M*, trigger, *T*, and echo, *E*, processes and a control process, *LID*. In addition *EVENT* mode requires a storage process, *S*. For each of the operating modes, the processes communicate as shown in Figure 3.

The REQUEST mode processes are described below. Process descriptions for the other operating modes follow similar lines and are given in Table 1.

Operator process, *OP*

In REQUEST mode, the operator can either set a new measure value from the values available (denoted by the set *V*) or fire the trigger. In the latter case the process can engage in no further behaviour. The operator process is modelled as a non-deterministic process. The mechanism by which the operator decides what selection to make is not modelled. The process description is:

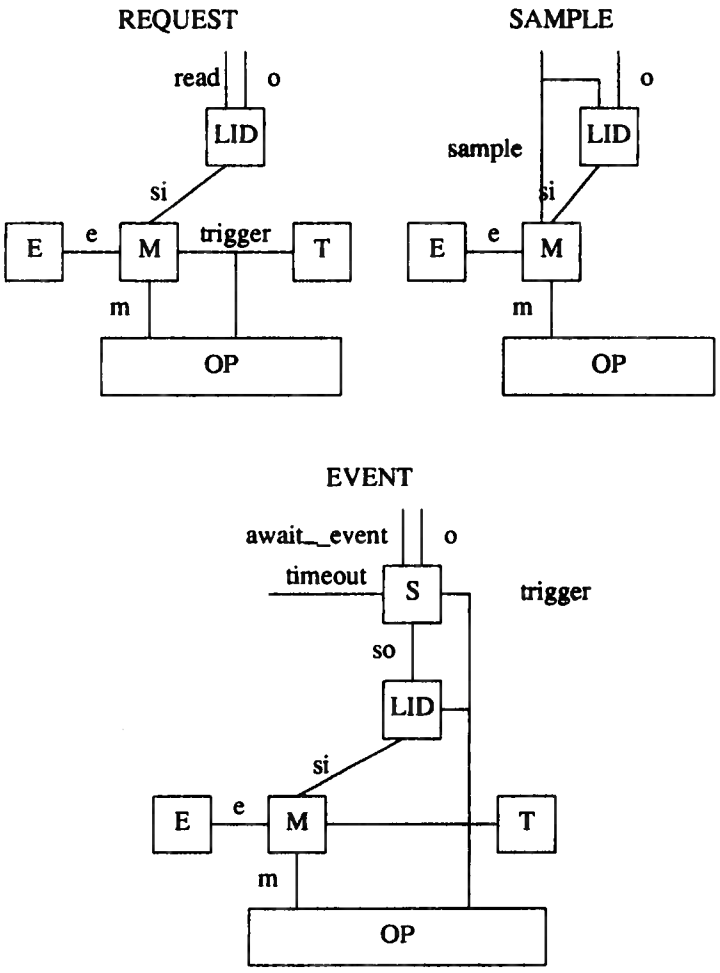

Figure 3

$$OP = (\prod_{v:V} m!v \rightarrow OP) \,\Pi\, (trigger \rightarrow STOP_{\alpha OP})$$

αOP denotes the alphabet of the process OP.

Measure process, M

When the operator sets a new measure value, the state of the measure process is updated to contain the new value. The measure process recording the current measure value v is denoted M_v. The new value is communicated to the echo process. When the trigger fires, the current measure value is communicated to the control process over channel si and the process engages in no further activity.

$$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \mid (trigger \rightarrow si!v \rightarrow STOP_{\alpha M})$$

Trigger process, T

When the trigger fires, this process engages in no further activity.

$$T = (trigger \rightarrow STOP_{\alpha T})$$

Echo process, *E*

This process echoes the value communicated on channel *e*. The mechanism by which the value is echoed is not described. The state of the echo process records the value echoed.

Control process, *LID*

The application program requests a value from the input device, denoted by the event *read*. When the trigger has been fired by the operator, the current measure value is communicated to the control process along channel *si* and communicated to the application program along channel *so*. Thereafter the process engages in no further actions.

$$LID = (read \rightarrow si?v \rightarrow o!v \rightarrow STOP_{\alpha LID})$$

The overall behaviour is described by the composition of the above processes:

$$OP \parallel M \parallel T \parallel E \parallel LID$$

The definitions of each of the processes in each of the operating modes are given in Table 1.

	REQUEST	*SAMPLE*	*EVENT*
OP	$(\prod_{v:V} m!v \rightarrow OP)$ $\Pi\ (trigger \rightarrow STOP_{\alpha OP})$	$(\prod_{v:V} m!v \rightarrow OP)$	$(\prod_{v:V} m!v \rightarrow OP)$ $\Pi\ (trigger \rightarrow OP)$
M	$(m?v' \rightarrow e!v' \rightarrow M_{v'})$ $\mid (trigger \rightarrow si!v \rightarrow STOP_{\alpha M})$	$(m?v' \rightarrow e!v' \rightarrow M_{v'})$ $\mid (sample \rightarrow si!v \rightarrow M_v)$	$(m?v' \rightarrow e!v' \rightarrow M_{v'})$ $\mid (trigger \rightarrow si!v \rightarrow M_v)$
T	$(trigger \rightarrow STOP_{\alpha T})$		$(trigger \rightarrow T)$
E	$(e?v' \rightarrow E_{v'})$	$(e?v' \rightarrow E_{v'})$	$(e?v' \rightarrow E_{v'})$
LID	$(read \rightarrow si?v \rightarrow o!v \rightarrow STOP_{\alpha LID})$	$(sample \rightarrow si?v \rightarrow o!v \rightarrow LID)$	$(trigger \rightarrow si?v \rightarrow so!v \rightarrow LID)$

$S_{q<s>} = (await\_event \rightarrow o!s \rightarrow S_q) \mid (trigger \rightarrow so?s' \rightarrow S_{<s'>q<s>})$
$S_{<>} = (await\_event \rightarrow ((time\_out \rightarrow o!NONE \rightarrow S_{<>}) \mid (trigger \rightarrow so?v \rightarrow o!v \rightarrow S_{<>})))$
$\quad \mid (trigger \rightarrow so?v \rightarrow S_{<v>})$

Table 1

3. Description of Hierarchical Input Devices

3.1. Introduction

The intention is to describe hierarchical input devices using the components and descriptions given above at each level. Thus a measure process at one level might receive input values from an input device at a lower level, as illustrated in Figure 4. Here the operator may manipulate the measure and trigger processes of the input device at level 1 and the trigger process of the device at level 2. The measure process at level 2 is manipulated by the input device at level 1.

The approach will be illustrated through a number of examples.

3.2. STROKE input

3.2.1. Description

The STROKE logical input device in GKS returns a sequence of positions in world coordinates, the LOCATOR logical input device returns a single position in world coordinates. It is possible to think of a STROKE device being constructed from a LOCATOR device which defines individual positions. GKS is more complicated than this in that the conversion from world coordinates is done using the highest priority normalization transformation within whose viewport all the points lie. In this example, this complication and the details of coordinate system mapping in general are not described, but could be included in a

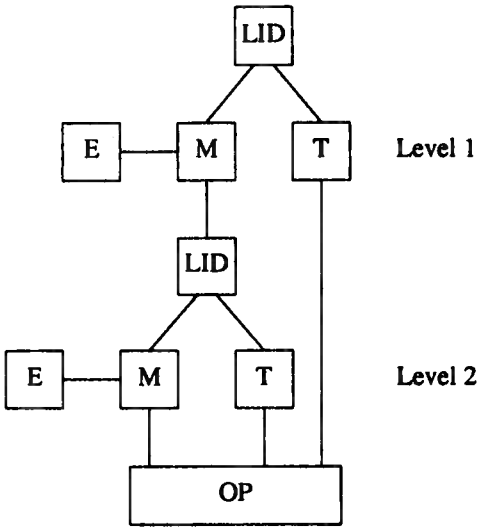

Figure 4

complete description of the LOCATOR and STROKE measure processes.

Process descriptions follow. The description will be given for EVENT mode operation of the STROKE device. The processes and communications channels are shown in Figure 5.

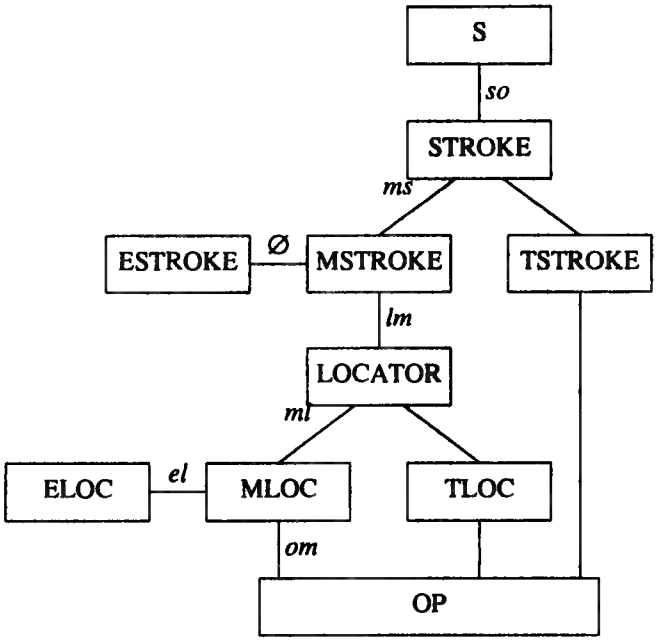

Figure 5

The operator process, *OP*

The operator of the STROKE input device can change the value of the LOCATOR measure process, fire the LOCATOR trigger to define a point in the STROKE and fire the STROKE trigger to indicate definition of a complete STROKE. The behaviour is characterized by the process:

$$OP = (\prod_{m\,:\,Point} om!m \rightarrow OP)\ \Pi\ (loctrigger \rightarrow OP)\ \ \Pi\ (stroketrigger \rightarrow OP) \qquad (1)$$

The LOCATOR measure process, *MLOC*

The measure process receives new measure values from the operator, communicates the new value to the echo process, and communicates the current value over channel *lm* when the LOCATOR trigger fires.

$$MLOC_v = (om?v' \rightarrow el!v' \rightarrow MLOC_{v'}) \mid (loctrigger \rightarrow ml!v \rightarrow MLOC_v) \qquad (2)$$

The LOCATOR echo process, *ELOC*

This process receives a LOCATOR measure value from the measure process and echoes it.

$$ELOC_v = (el?v' \rightarrow ELOC_{v'}) \qquad (3)$$

The LOCATOR trigger process, *TLOC*

The behaviour is just:

$$TLOC = (loctrigger \rightarrow TLOC) \qquad (4)$$

The LOCATOR input device control process, *LOCATOR*

This process receives the current measure value when the LOCATOR trigger fires and communicates it over channel *lm*.

$$LOCATOR = (loctrigger \rightarrow ml?v \rightarrow lm!v \rightarrow LOCATOR) \qquad (5)$$

The STROKE measure process, *MSTROKE*

This process receives new measure values from the LOCATOR input device and communicates the value to the STROKE echo process. This process occupies the position that the storage process would occupy if the LOCATOR device were used in EVENT mode in the normal way. The current stroke value is communicated over channel *ms* when the STROKE trigger fires. A STROKE is represented as a sequence of points. The notation $<v>$ denotes the sequence containing just the point v and juxtaposition of sequences denotes concatenation.

$$MSTROKE_s = (lm?v' \rightarrow \varnothing!s{<}v'{>} \rightarrow MSTROKE_{s\,<v'>}) \mid (stroketrigger \rightarrow ms!s \rightarrow MSTROKE_{<>}) \qquad (6)$$

After the *stroketrigger* fires the process reverts to a state in which the current measure value is the empty stroke. This corresponds to clearing the device's buffer and is the action taken by the GKS STROKE device. If this is not done, each subsequently generated stroke would have the previously generated stroke as a prefix.

The STROKE echo process, *ESTROKE*

This process echoes the stroke:

$$ESTROKE_s = (\varnothing?v \rightarrow ESTROKE_{s'}) \qquad (7)$$

The STROKE trigger process, *TSTROKE*

The behaviour is just:

$$TSTROKE = (stroketrigger \rightarrow TSTROKE) \qquad (8)$$

The STROKE input device control process, *STROKE*

This process receives the current stroke measure value when the trigger fires and communicates it to the storage process along channel *so*.

$$STROKE = (stroketrigger \rightarrow ms?s \rightarrow so!s \rightarrow STROKE) \tag{9}$$

The storage process, *S*

The storage process is the normal storage process for EVENT mode input.

$$S_{q\,<s>} = (\, await\_event \rightarrow o!s \rightarrow S_q) \mid (\, stroketrigger \rightarrow so?s' \rightarrow S_{<s'>\,q\,<s>}) \tag{10}$$

$$\begin{aligned} S_{<>} = (\, await\_event \rightarrow ((\, time\_out \rightarrow o!NONE \rightarrow S_{<>}\,) \mid \\ (\, stroketrigger \rightarrow so?v \rightarrow o!v \rightarrow S_{<>}))) \\ \mid (\, stroketrigger \rightarrow so?v \rightarrow S_{<v>}) \end{aligned} \tag{11}$$

The application process, *AP*

The application process can interrogate the storage process through *await_event*.

$$AP = (await\_event \rightarrow o?s \rightarrow AP) \tag{12}$$

The overall system

The overall system behaviour is described by the composition of the above processes:

$$\begin{aligned} SYS = {} & OP \parallel MLOC \parallel ELOC \parallel TLOC \parallel LOCATOR \parallel \\ & MSTROKE \parallel ESTROKE \parallel TSTROKE \parallel STROKE \parallel \\ & S \parallel AP \end{aligned} \tag{13}$$

Questions of input device initialization and termination have been ignored in this description.

3.2.2. Discussion

The operator process in equation (1) corresponds to the form of the event mode operator process in Table 1. A generalization is made to accept multiple triggers.

The LOCATOR measure (2), echo (3) and trigger (4) processes all correspond to the form for EVENT mode measure, echo and trigger processes given in Table 1. The LOCATOR control process, *LOCATOR* (5) also corresponds to the general form in Table 1. The process *MSTROKE* occupies the position that the storage process would occupy if the LOCATOR device were used in EVENT mode in the normal way. The STROKE processes (6) - (9) are exactly the same as the general form in Table 1, as is the storage process ((10) and (11)).

The input device described is an EVENT mode device, built using another EVENT mode device. A STROKE input device operating in SAMPLE mode is obtained by removing the *stroketrigger* event from the operator process, eliminating the storage process and stroke trigger process and replacing *await_event* in *AP* by *samplestroke* and *stroketrigger* in *MSTROKE* and *STROKE* by *samplestroke*. The output of *STROKE* is directed to the application rather than the storage component. Sampling a stroke does not remove the portion of the stroke defined up to that moment. The resulting equations are:

$$OP = (\prod_{m\,:\,Point} om!m \rightarrow OP) \,\Pi\, (loctrigger \rightarrow OP)$$

$$MSTROKE_s = (lm?s<v'> \rightarrow \varnothing!s<v'> \rightarrow MSTROKE_{s<v'>}) \mid (samplestroke \rightarrow ms!s \rightarrow MSTROKE_s)$$

$$STROKE = (samplestroke \rightarrow ms?s \rightarrow o!s \rightarrow STROKE)$$

$$AP = (samplestroke \rightarrow o?s \rightarrow AP)$$

Notice that the LOCATOR part of the description remains unchanged and that this input device continues to operate in an EVENT type mode. Sampling a STROKE device in this way returns to the application program the portion of the stroke which the operator has actually defined. The point that the operator is currently defining (if any) is not included in the stroke measure and so is not returned to the application.

3.3. String input

3.3.1. Description

This example follows the same general approach as the previous example, but is slightly more complex. The example is character string input, but the character strings may contain codes to change font and character size. Although the font and size coding will not be defined here, it may help the reader to give an example of what this might look like. Suppose the operator wanted to input the string:

A **Fancy** String

This might be represented by the input value:

A \fBFancy\fP \s14String\s0

The operator is provided with a keyboard to input characters, a choice device to select fonts and a choice device to select character size.

The input device hierarchy is shown in Figure 6.

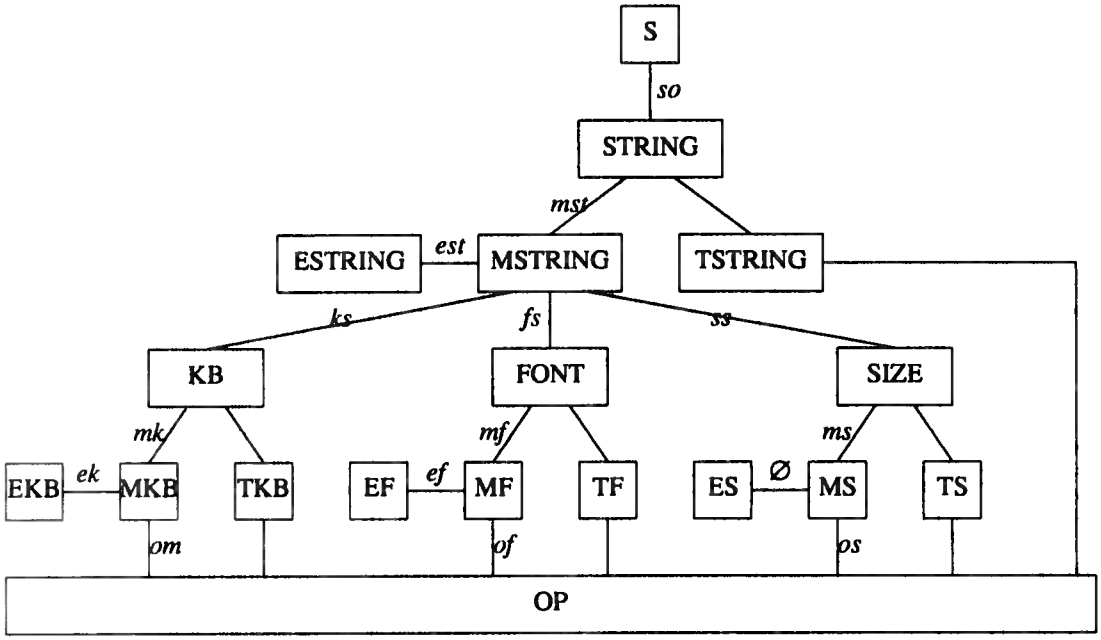

Figure 6

The operator process, *OP*

The operator of the string input device can input a character, select a font and select a character size. The behaviour is characterized by:

$$OP = (\prod_{k:Char} om!k \rightarrow OP) \,\Pi\, (\prod_{f:Font} of!f \rightarrow OP) \,\Pi\, (\prod_{s:Size} os!s \rightarrow OP) \,\Pi$$
$$(keypress \rightarrow OP) \,\Pi\, (setfont \rightarrow OP) \,\Pi\, (setsize \rightarrow OP) \,\Pi$$
$$(endstring \rightarrow OP)$$

The operator has available four triggers, *keypress*, *setfont*, *setsize*, and *endstring*. The first deserves a short explanation. The trigger *keypress* corresponds to the action of depressing a key. First the event *om!k* occurs which transmits the new character to the keyboard measure process. The event *keypress* then transmits this value to the measure process of the string device.

The remaining processes follow exactly the same style as the first example.

The keyboard measure process, *MKB*

The measure process receives new measure values from the operator and communicates the value to the echo process. The *keypress* event causes the current measure to be communicated to the string measure process, *MSTRING*, through the keyboard control process.

$$MKB_k = (om?k' \rightarrow ek!k' \rightarrow MKB_{k'}) \mid (keypress \rightarrow mk!k \rightarrow MKB_k)$$

The keyboard echo process, *EKB*

The keyboard echo process echoes the last key pressed.

$$EKB_k = (ek?k' \rightarrow EKB_{k'})$$

The keyboard trigger process, *TKB*

The behaviour is:

$$TKB = (keypress \rightarrow TKB)$$

The keyboard input device control process, *KB*

This process receives the current character measure from process *MKB* when a *keypress* event occurs and communicates it over channel *ks* to the string measure process, *MSTRING*.

$$KB = (keypress \rightarrow mk?k \rightarrow ks!k \rightarrow KB)$$

The font measure process, *MF*

The font measure process receives new measure values from the operator and communicates the new value to the echo process. This allows the operator to step through the available choices for fonts. The event *setfont* causes the current measure value (the currently selected font) to be communicated to the string measure process, *MSTRING*, through the font input device control process, *FONT*.

$$MF_f = (of?f' \rightarrow ef!f' \rightarrow MF_{f'}) \mid (setfont \rightarrow mf!f \rightarrow MF_f)$$

The font echo process, *EF*

The font echo process echoes the current value of the font measure process.

$$EF_f = (ef?f' \rightarrow EF_{f'})$$

The font trigger process, *TF*

The behaviour is:

$$TF = (setfont \rightarrow TF)$$

The font input device control process, *FONT*

When the *setfont* trigger fires, the current value of the font measure process is transmitted to the string measure process, *MSTRING*.

$$FONT = (setfont \rightarrow mf?f \rightarrow fs!f \rightarrow FONT)$$

The size measure process, *MS*

The size measure process is identical in form to the font measure process. The event *setsize* causes the current measure value (the currently selected size) to be communicated to the string measure process, *MSTRING*, through the size input device control process, *SIZE*.

$$MS_s = (os?s' \rightarrow \varnothing!s' \rightarrow MS_{s'}) \mid (setsize \rightarrow ms!s \rightarrow MS_s)$$

The size echo process, *ES*

The size echo process echoes the current size.

$$ES_s = (\varnothing?s' \rightarrow ES_{s'})$$

The size trigger process, *TS*

The behaviour is:

$$TS = (setsize \rightarrow TS)$$

The size input device control process, *SIZE*

When the *setsize* trigger fires, the current value of the size measure process is transmitted to the string measure process, *MSTRING*.

$$SIZE = (setsize \rightarrow ms?s \rightarrow ss!s \rightarrow SIZE)$$

The string measure process, *MSTRING*

This measure process accepts inputs from 3 different sources (the keyboard, font and size input device control processes) and maintains a current input value obtained by concatenating these values in the order in which they arrive. When the *endstring* trigger fires, the current value of the string measure process is communicated to the string input device control process, *STRING*.

$$\begin{aligned} MSTRING_{st} = {} & (ks?k \rightarrow est!st\langle k\rangle \rightarrow MSTRING_{st\langle k\rangle}) \mid \\ & (fs?f \rightarrow est!st\langle f\rangle \rightarrow MSTRING_{st\langle f\rangle}) \mid \\ & (ss?s \rightarrow est!st\langle s\rangle \rightarrow MSTRING_{st\langle s\rangle}) \mid \\ & (endstring \rightarrow mst!st \rightarrow MSTRING_{\langle\rangle}) \end{aligned}$$

The *endstring* trigger here is defined to be a destructive read on the input device in that the device's buffer is cleared by the trigger.

The string echo process, *ESTRING*

This follows the normal pattern of behaviour:

$$ESTRING_{st} = (est?st' \rightarrow ESTRING_{st'})$$

The string trigger process, *TSTRING*

This follows the normal pattern:

$$TSTRING = (endstring \rightarrow TSTRING)$$

The string input device control process, *STRING*

This process receives the current string measure value from the string measure process when the *endstring* trigger fires and communicates this value to the storage process, *S*.

$$STRING = (endstring \rightarrow mst?st \rightarrow so!st \rightarrow STRING)$$

The equations of the storage process, *S*, are not given here, but are identical in form to those given in the previous section.

3.3.2. Discussion

This example has described an input device whose measure process has 3 inputs, each provided by a separate input device. The equations follow exactly the same form as those given in Table 1.

3.4. Bicycle input

3.4.1. Description

This example is an elaboration of an example given in.[4] It describes an input device whose values are bicycles. A bicycle is defined as a frame plus two wheels. A wheel is defined as a rim and (in this case) an unspecified number of spokes. Rims, spokes and frames are treated as primitive input types, but it should be clear how these could be constructed from more primitive input types.

This example is considered here because it involves more levels than the examples given previously. The hierarchy is shown in Figure 7.

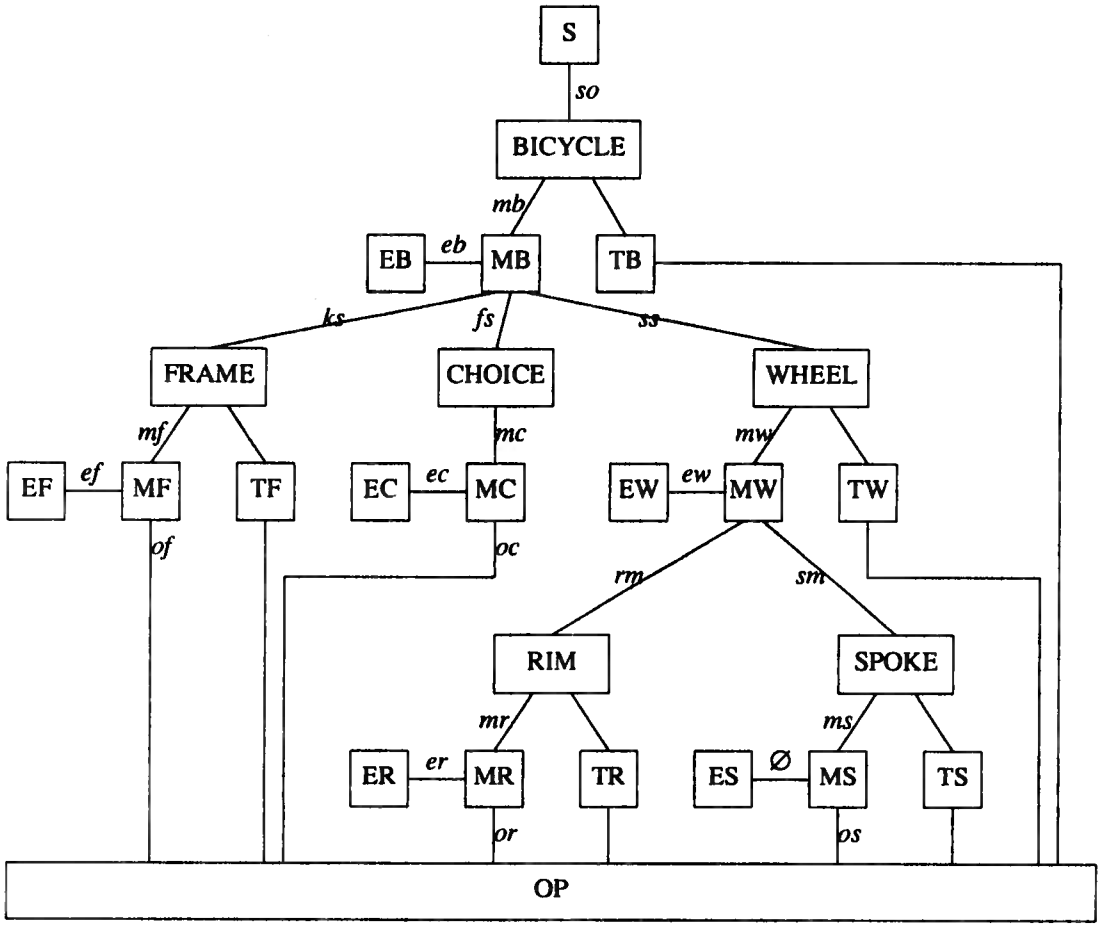

Figure 7

The operator process, *OP*

The operator of the bicycle input device can input a rim, spoke and frame, and indicate whether a particular wheel is the front wheel (*Fwheel*) or back wheel (*Bwheel*). The triggers *endrim*, *endspoke*, *endwheel* and *endbicycle* indicate that the definition of a rim, spoke, wheel or bicycle, respectively, is complete. The rim, spoke, and frame input devices allow the operator to select one of the available values for these entities, firing the corresponding trigger indicates that the current value is the one chosen.

$$OP = (\prod_{r:Rim} or!r \rightarrow OP) \,\Pi\, (\prod_{s:Spoke} os!s \rightarrow OP) \,\Pi\, (\prod_{f:Frame} of!f \rightarrow OP) \,\Pi$$
$$(oc!Fwheel \rightarrow OP) \,\Pi\, (oc!Bwheel \rightarrow OP)$$
$$(endrim \rightarrow OP) \,\Pi\, (endspoke \rightarrow OP) \,\Pi\, (endwheel \rightarrow OP) \,\Pi$$
$$(endbicycle \rightarrow OP)$$

The rim input device

The behaviour of the rim input device measure (*MR*), echo (*ER*), trigger (*TR*) and control (*RIM*) processes are given by the equations:

$$MR_r = (or?r' \rightarrow er!r' \rightarrow MR_{r'}) \mid (endrim \rightarrow mr!r \rightarrow MR_r)$$
$$ER_r = (er?r' \rightarrow ER_{r'})$$
$$TR = (endrim \rightarrow TR)$$
$$RIM = (endrim \rightarrow mr?r \rightarrow rm!r \rightarrow RIM)$$

The measure process, *MR*, communicates a new measure value from the operator to the echo process, and when the *endrim* trigger fires, communicates the current measure value to the rim input device control process.

The spoke input device

This is identical in form to the rim input device.

$$MS_s = (os?s' \rightarrow \varnothing!s' \rightarrow MS_{s'}) \mid (endspoke \rightarrow ms!s \rightarrow MS_s)$$
$$ES_s = (\varnothing?s' \rightarrow ES_{s'})$$
$$TS = (endspoke \rightarrow TS)$$
$$SPOKE = (endspoke \rightarrow ms?s \rightarrow sm!s \rightarrow SPOKE)$$

The wheel input device

A wheel is represented by a rim and a sequence of spokes. The current measure value of a wheel is thus denoted by a pair (r, s), where r denotes the rim and s the sequence of spokes. The wheel measure process receives new rim and spoke values from the RIM and SPOKE input devices. The rim value replaces the existing rim value in the measure and a new spoke value is concatenated to the existing sequence of spokes. When the *endwheel* trigger fires, the current wheel measure is communicated to the bicycle measure process, *MB*, through the wheel input device control process, *WHEEL*. The wheel process reverts to an initial measure value denoted by the pair $(r_0, <>)$, a default rim with no spokes.

$$MW_{(r,s)} = (rm?r' \rightarrow ew!(r',s) \rightarrow MW_{(r',s)}) \mid (sm?sp \rightarrow ew!(r,s<sp>) \rightarrow MW_{(r,s<sp>)}) \mid$$
$$(endwheel \rightarrow mw!(r,s) \rightarrow MW_{(r_0,<>)})$$
$$EW_w = (ew?w' \rightarrow EW_{w'})$$
$$TW = (endwheel \rightarrow TW)$$
$$WHEEL = (endwheel \rightarrow mw?w \rightarrow wm!w \rightarrow WHEEL)$$

The frame input device

The behaviour of this device is identical to the rim and spoke input devices.

$$MF_f = (of?f' \rightarrow ef!f' \rightarrow MF_{f'}) \mid (endframe \rightarrow mf!f \rightarrow MF_f)$$
$$EF_f = (ef?f' \rightarrow EF_{f'})$$
$$TF = (endframe \rightarrow TF)$$
$$FRAME = (endframe \rightarrow mf?f \rightarrow fm!f \rightarrow FRAME)$$

The choice input device

The operator uses this device to indicate which wheel is being defined, the front wheel (value *Fwheel*) or the rear wheel (value *Rwheel*). This device operates in SAMPLE mode. The behaviour is:

$$CM_c = (oc?c' \rightarrow ec!c' \rightarrow CM_{c'}) \mid (samplec \rightarrow mc!c \rightarrow CM_c)$$
$$EC_c = (ec?c' \rightarrow EC_{c'})$$
$$CHOICE = (samplec \rightarrow mc?c \rightarrow cm!c \rightarrow CHOICE)$$

The event *samplec* communicates the current choice value to the measure of the bicycle input device, *MB*.

The bicycle input device

A bicycle consists of a front wheel, a frame and a backwheel. The current measure value of the bicycle device is represented by a tuple (w_1,f,w_2). When a wheel is communicated to the device by the *WHEEL* process, the CHOICE input device is sampled to determine whether a wheel delivered to it is the front wheel or the back wheel of the bicycle. New wheel or new frame values supplied by the operator through the WHEEL and FRAME devices replace existing values in the current measure. The trigger *bicycle* causes the value of the current bicycle measure to be communicated to the storage process.

$$MB_{(w_1,f,w_2)} = (wm?w \rightarrow samplec \rightarrow ((cm?Fwheel \rightarrow eb!(w_1,f,w) \rightarrow MB_{(w_1,f,w)}) \mid (cm?Bwheel \rightarrow eb!(w,f,w_2) \rightarrow MB_{(w,f,w_2)})) \mid (fm?f' \rightarrow eb!(w_1,f',w_2) \rightarrow MB_{(w_1,f',w_2)}) \mid (endbicycle \rightarrow mb!(w_1,f,w_2) \rightarrow MB_{(w_1,f,w_2)})$$
$$EB_b = (eb?b' \rightarrow EB_b')$$
$$TB = (endbicycle \rightarrow TB)$$
$$BICYCLE = (endbicycle \rightarrow mb?b \rightarrow so!b \rightarrow BICYCLE)$$

3.4.2. Discussion

This example has shown how an input device with a 3-level hierarchy can be constructed. The principles are exactly the same as in the previous examples.

4. Conclusions

This paper has shown how hierarchically structured input devices can be described as a composition of single level devices. The description follows exactly the framework developed in the earlier paper.

References

1. D.A. Duce, P.J.W. ten Hagen, and R. van Liere, "Components, Frameworks and GKS Input," in *EUROGRAPHICS '89 European Computer Graphics Conference and Exhibition*, ed. F.R.A. Hopgood and W. Strasser, North-Holland, Amsterdam (1989).
2. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, London (1985).
3. ISO, "Information processing systems - Computer graphics - Graphical Kernel System (GKS)," ISO 7942, ISO Central Secretariat (1985).
4. Eurographics Association, *Proceedings of the GKS Review*, Eurographics Association, P.O. Box 16, 1288 Aire-la-Ville, Switzerland (1987).
5. ISO, "Information processing systems - Computer graphics - Reference model of computer graphics," Second Working Draft, RM/20 (January 1989).
6. ISO, *Report of the Improved Graphical Input Model Special Rapporteur Group*, ISO Central Secretariat (1989). (In Preparation)