EUROGRAPHICS '90 / C.E. Vandoni and D.A. Duce (Editors)
Elsevier Science Publishers B.V. (North-Holland)

AN APPROACH TO THE FORMAL SPECIFICATION OF THE COMPONENTS OF AN INTERACTION

Giorgio P. Faconti, Fabio Paterno'

C.N.R. - Istituto CNUCE
Pisa, Italy

In this paper we present the preliminary results from a work aiming to the formal specification af a model suitable for the description of interactive graphics program within the framework defined by the Reference Model for Computer Graphics Systems, actually under development within the International Organization for Standardization. The architecture defined by the Computer Graphics Reference Model, at its actual state of development, is shortly presented with particular attention paid to the concepts used in the paper. Following, the components of a basic interaction are identified and described as a set of independent communicating processes, referred to as an interactor. The relationships between interactors are described in terms of the communication between their component processes by using ECSP–like constructs.

1.INTRODUCTION

In the last few years a large amount of work in human–computer communication has been devoted to the description and specification of user interfaces. The models and tools used for the design and implementation of such interfaces are often referred to as User Interface Management Systems (UIMS) although the literature doesn't have a consistent view of this term. In fact, the traditional view of a UIMS is a piece of software which controls all communication between the user and the application program; the underlying idea being that of a strong separation of the semantic component and the user interface component.

In the majority of the cases, the specification of the man–machine dialogue is based on the linguistic model first proposed by Foley and Wallace [1] and subsequently reinforced by Foley and Van Dam [2], and adopted by the Seeheim model of UIMS [3]. This model defines a layered architecture of UIMS composed by a Presentation, a Dialogue Control, and an Application layers that correspond to the lexical, syntactic, and semantic analysis phases derived from the theory of formal languages and from compiler practice (figure 1).

The model by itself doesn't specify how to separate an interface into the layers making often difficult to understand where the boundaries between these layers lie. It substantially defines a framework within which UIMS may be described rather than providing for a functional description, either formally specified or not, of a system.

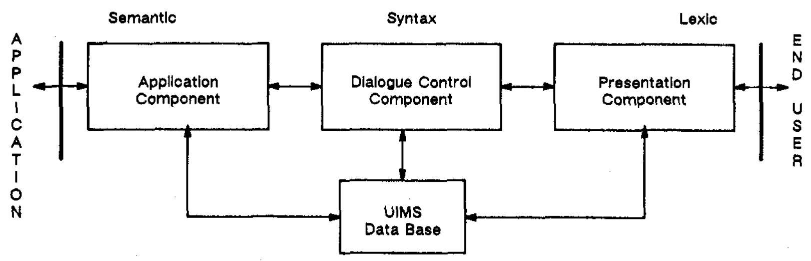

Figure 1. The Seeheim model of UIMS Structure.

As a result, many of the actual UIMS are dialogue specification systems (that is they address only the specification of the control of the dialogue) built on top of a generic graphics system from which the presentation component is constructed. They are mainly based on sequential control models that may be specified by using context–free grammars, state transition diagrams, or equivalent specification techniques [4]. However, such models fail in describing dialogue styles based on the direct manipulation of graphics images, as recently demonstrated by several authors [5, 6, 7, 8, 9].

As the presentation component inherits the input model of the underlying graphics system, the limitations of actual UIMS may be partially explained with the limitations found in the input model of graphics system. The model we are addressing here, although being adopted by standardized graphics systems as GKS and PHIGS [10, 11], has received severe criticisms [12, 13, 14, 15, 16, 17, 18] even before it reached its final state. The most controversial issues here are that:

- the model effectively hides from the application the specific peculiarities of hardware. Although this is undoubtedly an advantage taking into consideration the portability of applications across systems, for the purposes of graphics user interfaces development there is a need for the programmer to be able to have full control over the mapping mechanism between logical and physical input devices.

- logical input devices belong to a set of predefined classes distinguished by the type of data they return to the application, and there is no way to specify input data types outside of the predefined set.

- there exists a lack of uniformity among the different logical device classes with respect to their behaviour.

- input data types do not match with output data types leaving to the application the task of providing a consistent mapping between the two sets.

- it is undetermined how to relate prompt, echo, and acknowledgment functionalities to output.

Recently, the ISO/IEC subcommittee responsible for computer graphics, ISO/IEC JTC 1/SC 24, realized the need of developing an improved graphics input model to overcome this difficulties and authorized a Study Group to carry on the work. The results are reported in [19], where the major advances which have been reached can be summarized through the concepts of improved application control, I/O symmetry, extensibility of logical input classes, and input device composition.
ISO/IEC JTC 1/SC 24/WG 1 is also developing a Reference Model for Computer Graphics Systems with the aim of defining an architecture for computer graphics. The purpose of the Reference Model is to define the internal behaviour of graphics systems and establish the relationship between the concept which make up the Reference Model itself [20].

Explorative works have already been undertaken in order to identify notations which appear particularly suitable for formally specifying the components required to describe the improved input model within the framework set by the Reference Model [21, 22].

Starting from this premises, our goal is to specify the basic components from which interactive graphical programs can be modeled. Such components are described as a set of interaction units, namely interactors, each consisting of an input part, that has the task of managing measure values, and an output part, that contains corresponding pictures and provides for feedback. We differ from the previous approaches in that the internal components of interactors are differently specified and the concept of collection, as described in the Reference Model, is used to define both the class of an interactor as well as its appearance and behaviour. Moreover, to describe the communication between processes we use constructs derived from the Extended Communicating Sequential Processes (ECSP) [23], which allow for the specification of dynamically defined communication channels.

2. THE REFERENCE MODEL FOR COMPUTER GRAPHICS SYSTEMS

In this paragraph, a short overview of the Reference Model for Computer Graphics is given with particular attention to the concepts of which we are making use to describe an interaction. The detailed description of the model may be found in [20].

Within the Reference Model for Computer Graphics Systems, computer graphics is described as an environmental approach. An environment consists of output and input processes, a picture, a set of collections, a token store and possibly associated state information, defined at a specific coordinate space, as shown in figure 2.

The Reference Model describes five environments respectively called the application, the virtual, the projection, the logical, and the physical environments. Each of them is distinguished by the set of operations that output and input processes respectively perform on data stored in the picture and in the token store, and by the level of abstraction at which data are represented. The five environments are always present in the description of a graphics system but some of them may be transparent or null.

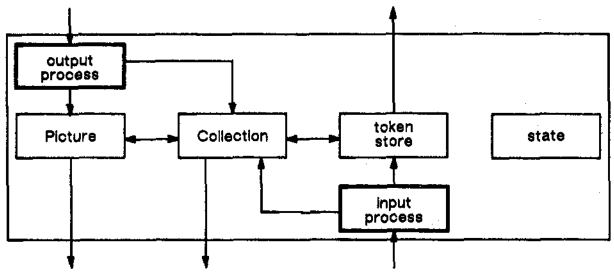

Figure 2. An generic environment of the Reference Model for Computer Graphics

2.1 Collections

A collection is a named structured assembly of entities which can be transformed either into a (part of a) picture within the same environment by the traversal operation and/or into the corresponding collection within the next lower level by the output process. A partially evaluated collection can be computed with data from the token store to provide for values of specific input data types. This provides with the capability of extending the input device classes so that they are able to return values of data types that can be dynamically defined depending on a specific instance of graphics system, and also of controlling the feedback provided by a specific instance of an input device.

2.2 Pictures

A picture is defined as a spacially structured set of output primitives at a given environment level. Output primitives are the atomic units from which graphical output is composed and are defined as:

<output_primitive>::=<output_primitive_class><geometric_shape><properties>.

Pictures are transformed from one environment level to the next lower environment level by the output process. Equivalent result can be achieved by processing the collection making up the picture at a given environment rather than processing the picture itself.

2.3 Token Store

The token store is composed by input primitives at a given environment level. They are the atomic units from which graphical input is composed ad are defined as:

<input_primitive>::=<input_primitive_class><geometric_shape><properties>.

Specific operations exist for assembling the token store at one environment to generate some part of the token store at the next higher environment level.

3. THE INTERACTION COMPONENTS

We describe the components of an interaction in term of processes. The environment which a device participating in an interaction belongs to, can be found out only when creating an instance of that device. The behaviour of the processes composing a device is specified in a uniform manner throughout the model independently from the levels of abstractions. Both logical and physical devices are described in term of I/O units, or interactors, that are made of an input module, an output module and a control component. They are specified by the triple

Interactor = <*CONTROL, IN, OUT*>.

The control component behaves the same across all the interactors in the system. It is responsible for the initialization of the interactor as well as for the control over the communication with others interactors in the system. The input component is an abstract internal representation of a possibly complex user input, while the output component is the representation of a picture that is in relation with the user input. Either the IN–component or the OUT–component may be empty, in which cases pure input and pure output devices can be realized.

The IN–component is defined as a trigger–measure–traversal triple:

IN = <*TRIGGER, MEASURE, I–TRAVERSAL*>,

and the OUT–component is defined as a feedback–traversal pair:

OUT= <*FEEDBACK, 0_TRAVERSAL*>.

When both IN–component and OUT–component are present (i.e. are not empty) within an interactor they use the same traversal component. Then the most general form of interactor is:

Interactor = <*CONTROL, MEASURE, TRIGGER, FEEDBACK, TRAVERSAL*>

as shown in figure 3.

Within a graphics environment several interactors may exist and be active at the same time. The picture, the collection, and the token store of that environment are defined as the composition of the corresponding entities found within the interactors, while the state information are represented by the number and classes of interactors that exist in the environment itself.

Interactors are distinguished by classes that univoquely specifiy the data type generated by the interactors belonging to a specific class. Let C be the set of classes actually available in a system and T the set of entities that can be stored in I–TRAVERSAL, then the class of an interactor is specified as:

$InteractorClass = f(e) \mid e \in T \quad \text{and} \quad f: T \rightarrow C.$

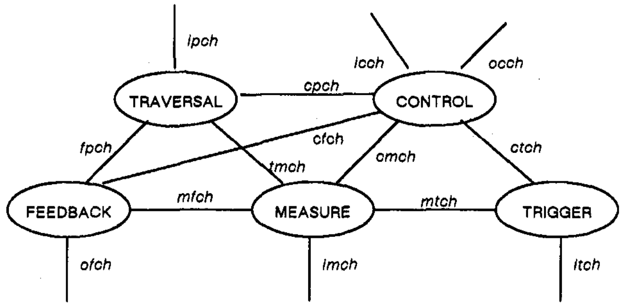

Figure 3. Processes of an Interactor and associated communication channels distinguished by type constructors

Interactors may be grouped into a set defined as:

$InteractorSet = \{ Interactor_1, Interactor_2, ... \}.$

Binding an InteractorSet to an InteractorClass defines a subclass for that class which completely defines its input data type. This can be specified as:

$InteractorSubclass = g(e,i) \mid e \in T \wedge i \in \Pi\ InteractorClass_j$

and

$g : T \times \Pi\ InteractorClass_j \rightarrow C$

where $InteractorClass_j$ is the class of the jth component interactor of the InteractorSet.

4. THE INTERACTOR PROCESSES

An interactor is modeled as a set of comunicating processes each one realizing one of the components previously identified. The behaviour of an interactor can then be modelled though the following ECSP construct:

```
Interactor ::
begin
    activate
      CONTROL;
      TRIGGER;
      MEASURE;
      TRAVERSAL;
      FEEDBACK
    end
    terminate (succ)
  end
```

4.1 The Control Process

The control process is responsible for the overall operation of an interactor and its associated component processes. It notifies lower level interactors of the triggers and measures is interested in, and receives similar requests from higher level interactors. When the trigger firing report is received, it reads from the measure process the value of the appropriate input primitive which was current at the time of the trigger firing, and sends it to all the higher level interactors that have declared an interest in receiving that value.
The control process also receives requests from the application to modify the appearance and the behaviour of the containing interactor. If the sender has the authorization to perform the operation, the traversal process is notified and the request can be honoured.

A simplified version of how the control process behaves, is described through the ECSP constructs that follow:

```
CONTROL :: var     declaration_of_local_variables
   begin
      list_of_clients := empty
      TRIGGER ! ctch(list_of_trigger_input_processes)
      MEASURE ! cmch(list_of_measure_input_processes)
      FEEDBACK ! cfch(list_of_filtering_conditions)
      for i=1 to i=number_of_input_processes do
                     list_of_input_processes(i) ! occh(interactor_name)
      repetitive
         priority (n); TRIGGER ? ctch(trigger_input)
            → MEASURE ? cmch(measure_input)
               compose_output_record
               if list_of_clients_not_empty then
                     begin
                        for i=1 to i=number_of_clients do
                           begin
                              reader := list_of_clients(i)
                              reader ! occh(output_record)
                           end
                     end
      % priority (n-1); SENDER : {any} SENDER ? icch(client)
            → validate_client.name_and_add_to_list
               if client.request_not_empty then do
                    begin
                       validate_client.request
                       if request = new_collection then do
                          TRAVERSAL ! cpch(client)
                       if request = new_feedback then do
                          FEEDBACK ! cfch(list_of_filtering_conditions)
                       if request = new_trigger_fire then do
                          TRIGGER ! ctch(list_of_firing_conditions)
                       if request = new_trigger_input then do
                          TRIGGER ! ctch(list_of_trigger_input_processes)
                    end
```

```
        end
        terminate (succ)
    end
```

4.2 Trigger and Measure Processes

Traditionally an input device has been described in term of measure and trigger processes. A measure is defined as a process that, when activated, continously updates the data value for the data type defined by that device, while a trigger has been defined as a set of conditions which, when satisfied, identifies a significant moment in time. This way of distinguishing between measures and triggers founds its justification in that triggers traditionally defined what a user had to do to engage an interaction, and were generally intended as the manipulation of real input devices.

In an improved input model, a trigger firing doen't necessarily deals with real devices; rather, it may be generated from within any device, either logical or physical, although it can trigger only if it was itself just triggered. Conceptually, trigger and measure processes both read a set of input values, apply to them a function, and produce a new data value to which different semantics apply.

From the previous considerations, triggers and measures show an uniform behaviour with respect to input data. However while the measure uniquely identifies the type of data generated by an interactor (i.e. its class), the triggers just validate the current measure. Formally:

$$G = \{ g \mid g : T \times \Pi\ InteractorClass_j \rightarrow C_m \}$$

$$\forall g \,.\, \exists F_g : \Pi\ InteractorClass_j \rightarrow C_m$$

where G is the set of functions g defined in the previous paragraph, T is the set of entities in I–TRAVERSAL, $InteractorClass_j$ is the class of the jth component interactor of an InteractorSet, Cm is the set of classes for the measure processes, and Fg is the measure function. In the simple case where

$$F_g = identity$$

a measure process behaves exactly as the measure of a composite input device as described in [22].

Triggers are distinguished from measures in that the following conditions apply:

$$\forall g \,.\, \exists F_t : \Pi\ InteractorClass_j \rightarrow C_t$$

$$C_t = \{\ \textbf{false}, \textbf{true}\}$$

where Ft is the trigger function, and Ct is the unique data type returned by the trigger process.

A trigger is said to have fired when A = true at a given instant in time, that is the conditions are meet. This can be formalized by saying that given the left and right neighbourhoods of the instant in time $t_0 \in [\, 0, t_n \,]$ such that:

$$U^- = \{\, t_0 \in [\, 0, t_n \,] \mid t \in]\, t_0 - \epsilon ,\ t_0 [\, \} \ \forall\, \epsilon > 0 ,$$

$$U^+ = \{\, t_0 \in [\, 0, t_n \,] \mid t \in]\, t_0 ,\ t_0 + \epsilon [\, \} \ \forall\, \epsilon > 0 ,$$

then the following conditions apply:

$$F(t_0) = \textbf{true} \ \text{ if } \exists\, U^-, U^+ \mid F(t_1) \neq F(t_2) \ ,\ \forall t_1 \in U^-, \forall\, t_2 \in U^+$$

that describes a generic trigger (i.e. a key event), and

$$F(t_0)^- = \textbf{true} \ \text{ if } \exists\, U^-, U^+ \mid F(t) = \textbf{true} \ ,\ \forall t \in U^-, \ \text{and} \quad F(t) = \textbf{false} \ ,\ \forall\, t \in U^+$$

$$F(t_0)^+ = \textbf{true} \ \text{ if } \exists\, U^-, U^+ \mid F(t) = \textbf{false} ,\ \forall t \in U^-, \ \text{and} \quad F(t) = \textbf{true} \ ,\ \forall\, t \in U^+$$

that describe specific triggers (i.e. respectively a key press and a key release).

The behaviour of the trigger process is described by the following simplified ECSP-like program, where it is shown that the trigger is made of repetitive loop with three alternatives: first reads and eventually. Following, it enters a repetitive loop with two alternatives that show different priorities:

- in the first alternative, the trigger reads from the control process the list of processes that are the producers of the measures to be monitored.

- in the second alternative, the trigger reads the information necessary to build up a P_function which identifies the conditions for the trigger firing.

- the third alternative is engaged when the communication with the control process has been cleared. It enters a repetitive loop where it reads data from the included in the list of the producers, possibly transforms them in order to be evaluated from the P_function, and if the firing conditions are met, it notifies both the measure process and the control process.

```
TRIGGER :: var        declaration_of_local_variables
  begin
    P_function := is_true
    stored_input := null_value
    repetitive
      trigger := true
      priority (n); CONTROL ? ctch(list_of_input_processes)
    % priority (n); CONTROL ? ctch(list_of_firing_conditions)
          → build_P_function_from_list_of_firing_conditions
    % priority (n-1);
        repetitive
          trigger; (SENDER : {list_of_input_processes}) SENDER ? itch(input)
```

```
                → stored_input := store_input(input)
                  if P_function(stored_input) then begin
                                                    MEASURE ! mtch(true)
                                                    CONTROL ! ctch(stored_input)
                                                    trigger := flase
                                                   end
            end
        end
        terminate (succ)
    end
```

The behaviour of the measure process is described by a similar program. It should be noted that with respect to processes providing with input measures, the measure process behaves exactly like a trigger process.

```
  MEASURE :: var     declaration_of_local_variables
    begin
      F_function := identity
      P_function := identity
      stored_input := null_value
      CONTROL ? cmch(list_of_input_processes)
      TRAVERSAL ? tmch(entities)
      build_P_function_from_entities_and_list_of_input_processes
      repetitive
        priority (n); TRIGGER ? mtch(trigger)
          → CONTROL ! cmch(P_function(stored_input))
            FEEDBACK ! mfch(stored_input, end_loop)
      % priority (n-1);
          (SENDER:{list_of_processes})SENDER ? imch(input)
          → stored_input := store_input(input)
            FEEDBACK ! mfch(stored_input)
      end
      terminate (succ)
    end
```

4.3 The Traversal Process

This process is composed by a collection and by a process performing the traversal operation as defined in the Reference Model for Computer Graphics. The traversal process refers both to input and output components of an interactor, however the result of traversing a collection provides for different semantics. As for the input side, the subsequent filtering of the traversal operation uniquely defines the data type generated by a specific interactor, that is it defines its class. As the for the output side, the traversal of a collection produces a picture that represents the appearance and behaviour of the interactor as perceived by the user. A given graphics system completely defines the entities which can be made part of a collection, and consequently the set of the classes of the available interactors for that system.

A simple ECSP program describing the behaviour of this process may look as follows:

```
TRAVERSAL :: var      declaration_of_local_variables
  begin
    MEASURE ! tmch(entities)
    FEEDBACK ! fpch(default_behaviour)
    repetitive
      CONTROL ? cpch(client)
      → client.name ? ipch(new_collection)
        if client.request = new_feedback then do
                                FEEDBACK ! fpch(traversal_of_new_collection)
    end
    terminate (succ)
  end
```

4.4 The Feedback Process

The feedback process within a specific interactor is responsible for generating an output picture that gives the appearance of that interactor at a given moment in time. Whenever a measure process is engaged in an interaction, its value is made available to the corresponding feedback process. This value is used to apply a filtering function to the result of the traversal operation on a collection. The resulting picture is added to the current picture for display purposes.

A simple ECSP program describing the behaviour of this process may look as follows:

```
FEEDBACK ::   var declaration_of_local_variables
  begin
    repetitive
      new_loop := true
      priority (n); CONTROL ? cfch(list_of_filtering_conditions)
        → build_F_function_from_list_of_filtering_conditions
    % priority (n-1); TRAVERSAL ? fpch(traversal_of_collection)
        → CURRENT_PICTURE ! ofch(F_function(traversal_of_collection, def))
    % priority (n-2);
        repetitive
          new_loop; MEASURE ? mfch(new)
          → CURRENT_PICTURE ! ofch(F_function(traversal_of_collection, new))
            if ( new = end_loop) new_loop := false
        end
    end
    terminate (succ)
  end
```

5. CONCLUSIONS

The interactors model is being developed for the purposes of describing graphical interactive programs in the framework defined by the Reference Model for Computer

Graphics. Although it is at a very early stage of development it is very promising for its applicability to a wide range of applications and environments. Especially, it has been proved to be useful in the specification of user interfaces relying on multi–tread input and on multiple feedback.

Further researches are expected to be carried on to refine the model especially on the control of the activation of processes, on the consideration of usability of asynchronous symmetric channels, and on the capability of dynamically defining output measure data types.

6. REFERENCES

[1] J.D. Foley, V.L. Wallace, The Art of Natural Graphic Man–Machine Conversation, Proceedings of IEEE 62, 1974.

[2] J.D. Foley, A. Van Dam, Fundamental of Interactive Computer Graphics, Addison–Wesley, 1982.

[3] M. Green, Report on Dialogue Specification Tools, User lnterface Management Systems, G. Pfaff ed., Springer–Verlag, 1985.

[4] B. Betts et al., Goals and Objectives for User Interface Software, Computer Graphics, 2(21), 1987.

[5] D. Olsen et at., A Context for User Interface Management, IEEE Computer Graphics & Application, 12, 1984.

[6] K.A. Lantz, Multi–process Structuring of User Interface Software, Computer Graphics, 2(21), 1987.

[7] R.D. Hill, M. Herrmann, The Structure of Tube - A Tool for Implementing Advanced User Interfaces, Proceedings of EUROGRAPHlCS'89, Hamburg, F.R.G., 1989.

[8] W. Hubner, M. de Lancastre, Towards an Object–Oriented Interaction Model for Graphics User Interfaces, Computer Graphics Forum, 3(8), 1989.

[9] J.R. Dance et al., The Run–time Structure of UIMS–Supported Applications, Computer Graphics, 2(21), 1987.

[10] ISO/IS 7942, lnpormation processing systems, Computer Graphics, Graphical Kelnel System - Functional Description, 1985.

[11] ISO/IS 9592:1989, Information processing systems, Computer Graphics, Programmers Hierarchical Interactive Graphics System - Functional Description , 1989.

[12] R.A. Guedj at al., Proceedings of Seillac II, lFlP Workshop on methodology of Interaction, Seillac, France, May 1979, North–Holland, 1980.

[13] H.G. Borufka, P.J.W. Ten Hagen, H.W. Kuhlmann, Dialogue Cells, a method for defining interaction, IEEE Computer Graphics & Applications, 2(5), 1982.

[14] D.S.H. Rosenthal, J.C. Michener, G. Pfaff, R. Kessner, M. Sabin, The detailed semantic of graphical input devices, Computer Graphics, 16(3), 1982

[15] D.S.H. Rosenthal, Managing Graphical Resources, Computer Graphics, 1983.

[16] G. Pfaff, Proceedings of IFIP Workshop on User Interface Management Systems, Seeheim, F.R.G., Springer–Veriag, 1985.

[17] R.vanLiere, P.J.W.tenHagen, Logical Input Devices and Interaction, Report of Center for Mathematics and Computer Science, 1987,

[18] D. Duce, Configurable Input Devices, Proceedings of Eurographics Workshop on GKS Review, ed. W.T. Hewitt, Manchester, U.K., 1987.

[19] ISO/IEC JTC 1/SC 24 N 353, Final Report of the Study Group on an Improved Graphics Input Model, 1989.

[20] ISO/JTC 1/SC 24/WG 1/N 84, Information processing systems, Computer Graphics, Computer Graphics Reference Model, 1989.

[21] D.A.Duce, R.vanLiere, P.J.W.tenHagen, Components, Framework and GKS Input, Proceedings of Eurographics'89, North–Holland, 1989.

[22] D.A.Duce, R.vanLiere, P.J.W.tenHagen, An Approach to Hierarchical Input Devices, Report of Center for Mathematics and Computer Science, 1989

[23] F.Baiardi, M.Vanneschi, Linguaggi per la Programmazione Concorrente , Milan, Italy, Franco Angeli Libri, 1985.

7. APPENDIX - A short introduction to ECSP

ECSP (Extended Communicating Sequential Processes) is a programming language developped at the Computer Science Department of the University of Pisa.

ECSP constructs extend the best known CSP in that the communication primitives allow for the definition of asymmetric synchronous and symmetric asynchronous channels as well as the traditional symmetric synchronous ones.

A channel is identified by the triplet:

    <CANDIDATE_SENDER> <RECEIVER> <CONSTRUCTOR>

where CANDIDATE–SENDERand RECEIVER may be either explicit names of processes or variables of the type process–name, and CONSTRUCTOR may be either an explicit constructor type or a variable of the type channel–name.

Because of the components of the specification of a channel being variables, both the partners engaged in an interaction and the type of data they are exchanging can be defined by applying a function. This allows for dynamic allocation of statically defined channels to processes and for dynamic binding of data types to channels. Consequently a channel is itself specified to be a data type defined by the process that is the receiver of the messages carried over that channel. The syntax of the communication primitives becomes:

P!OP(expr)

for the write operation on all types of channels

P?OP(v)

for the read operation from a symmetric synchronous channel

(P : { P1,...,Pn}) P?OP(v)

for the read operation from an asymmetric synchronous channel and

buffer from P pattern (T constructor OP) length N;
...
P?OP(v)

respectively for the declaration and for the read operation from a symmetric asynchronous channel. P, P1, ..., Pn are process–namevariables or constants; OP is a channel–name variable or a constant type constructor; expr is an expression whose value is transmitted over the channel; v is the target variable of the message; T is the message type; N is a constant or an integer variable indicating the number of buffers on the channel (that is the channel contains a FIFO queue with N positions).

At any time the set of the channels of a program must be unambiguous: for each communication between two processes a message might be send only through one channel. That is the following applies: let C = C1, ..., Cn be a set of channels and let i and j be so that Ci=(seti, P, Ti) and Cj=(setj, P, Tj). Then Ti=Tj and $seti \cap setj \neq 0$ implies i=j.

In order to deal with nondeterministic events the alternative (%) and repetitive guarded construct have been introduced where a guard may be specified as a combination of boolean expressions and read operations. It is also possible to specify a priority by associating to each guard a variable or an integer constant.