11.1.5 Event (Event data) port

(10) Event data ports are intended for message transmission.... A receiving thread can get access to one or more data element in the queue according to the Dequeue_Protocol and Dequeued_Items properties. ...Individual element of the queue can be retrieved via the port variable using the Get_Value and Next_Value service calls. If the queue is empty the most recent data value is available.

(11) Event ports are intended for event and alarm transmission.... A receiving thread can get access to one or more events in the queue according to the Dequeue_Items property.

(9.1(16)) The AADL supports n-to-n connectivity for event and event data ports. A port may have multiple outgoing connections, i.e., its content is transmitted to multiple destinations. This means that each destination port receives an instance of the event, or event data being transmitted. (PLG claim not consistent with the above fan_out_policy ????) Similarly, event and event data ports can support multiple incoming connections resulting in sequencing and possibly queuing of incoming events and event data.

Event and event data ports can have a queue associated with them. By default, the incoming event (event data) ports of threads, devices and processors have queues.

1. **Dispatch event (event data) port**

   (C1) The ports that trigger the dispatch must have a Input_Time property value of Dispatch_Time.

   (20) If no event or event data port is explicitly connected to or associated by condition with the Dispatch port, then any incoming event or event data port can trigger the dispatch. The input of other ports that can trigger dispatch is not frozen. Input of the remaining ports is frozen according to the specified input time. ???

   (21) If event and event data ports are explicitly connected to the Dispatch port, then only one of those port will trigger the dispatch. The input of other ports that can trigger dispatch is not frozen (PLG thus simultaneity only occurs for data ports or non dispatching event). Input of the remaining ports is frozen according to the specified input time.

   (22) If a dispatch condition is specified (PLG: HOW ???, dispatch condition does not seem to be defined; is it the condition in Await_Dispatch runtime? If such, there is no hope to fully model dispatch in Signal if the condition is not written in Signal) then the logic expression determines the combination of event and event data ports that trigger a dispatch, and whose input is frozen as part of the dispatch. The input of other ports that can trigger dispatch is not frozen. Input of the remaining ports is frozen according to the specified input time.

   (23) If an event port is associated with a component (including thread) containing modes and mode transition, and the mode transition names the event port, then the arrival of an event is a mode change request and it is processed according to the mode switch semantics.

   (35) ... If such an incoming port is associated with a thread and the thread does not contain a mode transition naming the port, then the event or event data arriving at

this port is added to the queue of the port. If the thread is aperiodic or sporadic and does not have its Dispatch event connected (PLG: in the current mode) , then each event and event data arriving and queued at any incoming ports of the thread results in a separate request for thread dispatch. PLG: what about other threads ?

Dispatch event and Polychrony A Signal-process is dedicated to generate the dispatch event (only for event driven AADL-threads).

2. **Port queue**

   Queue properties for in event(-data) port:

Overflow_Handling_Protocol: enumeration (DropOldest, DropNewest, Error) Urgency: aadlinteger 0 .. value(Max_Urgency) Dequeued_Items: aadlinteger Dequeue_Protocol: enumeration (OneItem, MultipleItems, AllItems)

   (30) ... If an event arrives and the number of queued events (and any associated data) is equal to the specified queue size, then the Overflow_Handling_Protocol property determines the action. If the Overflow_Handling_Protocol property value is

   - Error, then an error occurs for the thread. ...
   - DropNewest and DropOldest, the newly arrived or oldest event in the queue event is dropped.

   (11)The number of queued event (data) elements accessible to a thread can be determined through the port variable using the Get_Count service call.

   (31) Queues will be serviced according to the Queue_Processing_Protocol, (PLG: not defined in my copy) Yue: Queue_Processing_Protocol could be one of the Supported_Queue_Processing_Protocol (which is an enumeration type specifies the set of queue processing protocols that are supported. By default is FIFO. Other protocols are project specific). by default in a first-in, first-out order (FIFO). When an event-driven thread declares multiple in event and event data ports in its type and more than one of these queues are nonempty, the port with the higher Urgency property value gets serviced first. If several ports with the same Urgency are non-empty, then the Queue_Processing_Protocol is applied across these ports and must be the same for them. In the case of FIFO the oldest event will be serviced (global FIFO). It is permitted to define and use other algorithms for picking among multiple non-empty queues. Disciplines other than FIFO may be used for managing each individual queue.

   (32) By default, one item is dequeued and made available to the source text through the port variable. The Dequeue_Protocol property specifies different dequeuing options.

   - OneItem: (default) a single frozen item is dequeued and made available to the source text unless the queue is empty. The Next_Value service call has no effect. Yue: copy only one value at input time.

- AllItems: all items that are frozen at input time are dequeued and made available to the source text via the port variable, unless the queue is empty. Individual items become accessible as port variable value through the Next_Value service call. (PLG meaning that values remain totally ordered) Yue: copy all values. Next_Value service call is used to access a value.
- MultipleItems: multiple items can be dequeued one at a time from the frozen queue and made available to the source text via the port variable. One item is dequeued and its value made available via the port variable with each Next_Value service call. Any items not dequeued remain in the queue and are available for the next dispatch. Yue: multiple (Dequeue_Items) values are copied. Use Next_Value to access one item at a time.

(46, p.143) For each data or event data port declared for a thread, a system implementation method must provide sufficient buffer space within the associated binary image to unmarshall the value of the data type. Adequate buffer space must be allocated to store a queue of the specified size for each event data port.

3. **Port queue and Polychrony**

   A Signal-process is dedicated to manage the port queue. The queue could be any supported processing queue, by default is fist in first out. It is made of a Queue and a controler defined wrt to port queue rules.

   To deliver multiple values, one can use an array with a companion counter (the number of meaningful values in the array) or introduce a new (?) type (bounded) sequence in Signal and associated operators (size, append, next,...)

   In the current translation, a FIFO queue is provided (the other Supported_Queue_Processing_Protocol has not been implemented yet). A FIFO library is defined to: create a FIFO, set messages to a FIFO, and send out messages from a FIFO, etc. The FIFO management will be implemented in external C code. Other types of queues could be developed later.

```
process CREATE_FIFO =
(? string fifo_name;
   integer fifo_size;
   Data_type default_msg;
 ! ID_type fifo_ID; )
 pragmas
       Comment "create a FIFO"
     end pragmas
(| fifo_ID := FIFO_RECORD{}(fifo_name, fifo_size, default_msg)
 |);

process FIFO_RECORD =
( ? string fifo_name;
    integer fifo_size;
    Data_type default_msg;
  ! ID_type fifo_ID;)
```

```
pragmas
         C_Code "GLOBAL_FIFO_MANAGER->FIFORecord(&i1,&i2, &i3, &o1)"
end pragmas;

process In_FIFO =
{integer k;}
(? ID_type fifo_ID; [k]Data_type x_in;)
 pragmas
       Comment "send x_in (k elements) to FIFO"
     end pragmas
(| fifo_ID ^= x_in
 | fifo := FIFO_CHECKID{}(fifo_ID)
 | fifo_new := SET_MESSAGE{k}(fifo_ID, fifo, x_in)
 |)
where
  FIFO_type fifo, fifo_new;
end;

process FIFO_CHECKID =
( ? ID_type fifo_ID;
  ! FIFO_type fifo_out;)
pragmas
         C_Code "GLOBAL_FIFO_MANAGER->FIFOCheckID(&i1,&o1)"
end pragmas;

process SET_MESSAGE =
{integer k;}
(? ID_type fifo_ID; FIFO_type fifo; [k]Data_type x_in;
 ! FIFO_type fifo_new;)
 pragmas
       Comment "add [k]x_in at the end of fifo_ID "
     end pragmas
(| fifo_new.name := fifo.name
 | fifo_new.size := fifo.size
 | fifo_new.nbmsg := (fifo.nbmsg + k) when ((fifo.nbmsg + k)<=fifo.size)
                     default fifo.size

| array i to (MAX_QUEUE_SIZE-1) of
     fifo_new.fifo_queue[i] := if (i<fifo.nbmsg) then fifo.fifo_queue[i]
       else if ((i<fifo.size) and (i<(fifo.nbmsg+k))) then x_in[i-fifo.nbmsg]
       else fifo.fifo_queue[i]
   end
| FIFO_UPDATE(fifo_ID,fifo_new)
 |);

process FIFO_UPDATE =
(? ID_type fifo_ID; FIFO_type fifo;)
 pragmas
       Comment "update the fifo recored"
       C_Code "GLOBAL_FIFO_MANAGER->FIFOUpdate(&i1,&i2)"
end pragmas;

process Out_FIFO =
{integer k; Data_type default_msg;}
```

```
(? ID_type fifo_ID;
 ! [k]Data_type x_out; )
 pragmas
       Comment "get k elements (x_out) from FIFO, (suppose that k<fifo.nbmsg)"
     end pragmas
(| fifo := FIFO_CHECKID{}(fifo_ID)
 | (fifo_new, x_out) := GET_MESSAGE{k, default_msg}(fifo_ID, fifo)
 |)
where
  FIFO_type fifo, fifo_new;
end;

process GET_MESSAGE =
{integer k; Data_type default_msg;}
(? ID_type fifo_ID; FIFO_type fifo;
 ! FIFO_type fifo_new; [k]Data_type x_out;)
 pragmas
       Comment "from fifo_ID, get k elements, return [k]x_out"
     end pragmas
(| fifo_new.name := fifo.name
 | fifo_new.size := fifo.size
 | fifo_new.nbmsg := (fifo.nbmsg - k) when ((fifo.nbmsg - k)>0)
                     default 0

 | array i to (MAX_QUEUE_SIZE-1) of
     fifo_new.fifo_queue[i] := if ((i<(fifo.nbmsg-k)) and (fifo.nbmsg>k))
       then fifo.fifo_queue[k+i]
       else if ( (i>=(fifo.nbmsg-k)) and (fifo.nbmsg>k))
       then default_msg
       else default_msg
   end
 | array i to k of
     x_out[i] := if (i < fifo.nbmsg) then fifo.fifo_queue[i]
                     else default_msg
                     end
 | FIFO_UPDATE(fifo_ID, fifo_new)
 |);

process Move_k_FIFO =
{integer k; Data_type default_msg;}
(? ID_type fifo_send, fifo_receive; event Move_time;
 ! [k]Data_type x_move;)
 pragmas
       Comment "at Move_time, move k elements from fifo_send to fifo_receive"
     end pragmas
(| x_move := Out_FIFO{k, default_msg}(AT(fifo_send, Move_time))
 | In_FIFO{k}(AT(fifo_receive, Move_time),x_move)
 |);

process Move_all_FIFO =
{Data_type default_msg;}
(? ID_type fifo_send_ID, fifo_receive_ID; event Move_time;)
 pragmas
       Comment "at Move_time, move k elements from  fifo_send to fifo_receive"
```

```
     end pragmas
(| fifo_send := FIFO_CHECKID{}(AT(fifo_send_ID, Move_time))
 | fifo_receive := FIFO_CHECKID{}(AT(fifo_receive_ID, Move_time))
 | fifo_send_new.name := fifo_send.name
 | fifo_send_new.size := fifo_send.size
 | fifo_send_new.nbmsg := 0

| array i to (MAX_QUEUE_SIZE-1) of
     fifo_send_new.fifo_queue[i] := AT(default_msg, Move_time)
   end
 | FIFO_UPDATE(fifo_send_ID, fifo_send_new)

 | fifo_receive_new.name := fifo_receive.name
 | fifo_receive_new.size := fifo_receive.size
 | fifo_receive_new.nbmsg := (fifo_receive.nbmsg + fifo_send.nbmsg)
      when ((fifo_receive.nbmsg + fifo_send.nbmsg)<fifo_receive.size)
      default fifo_receive.size

| array i to (MAX_QUEUE_SIZE-1) of
     fifo_receive_new.fifo_queue[i] :=
     if (i<fifo_receive.nbmsg)
     then AT(fifo_receive.fifo_queue[i], Move_time)
     else AT(fifo_send.fifo_queue[i-fifo_receive.nbmsg], Move_time)
     end
 | FIFO_UPDATE(fifo_receive_ID, fifo_receive_new)
 |)
where
  FIFO_type fifo_send, fifo_receive, fifo_send_new, fifo_receive_new;
end;
```

4. **In event (event data) port and Polychrony**

   An event (event data) port could be represented by a pair of QUEUEs (*Ex-QUEUE* and *Frozen-QUEUE*) and a container of constraints. (Figure 18). *Ex-QUEUE* receives inputs from other threads (e.g., In_FIFO() process). At *InEvent* (constraint by **Input_Time** in *Between*() process), move (*Frozen*) all the actual elements from *Ex-QUEUE* to *Frozen-QUEUE* (Move_all_FIFO()). The inputs arrived after the *InEvent* will be available at the next *InEvent*.

   A number of elements (this number is determined by the **Dequeue_Protocol** property) in *Frozen-FIFO* will be dequeued (Dequeue() process) into an array (i_accessible). Any items not dequeued remain in the *Frozen-QUEUE* and are available for the next *InEvent*. The elements in i_accessible array could be used one at a time through *Next_Value* service call.

   At **InEvent**, frozen the inputs: copy all elements of *Ex-QUEUE* into (*Frozen-QUEUE*), and dequeue k elements of *Frozen-QUEUE*. These processes are defined in library AADL_EVENTPORT and AADL_EVENTDATAPORT.

```
process Frozen_event_port =
{Data_type default_msg;}
```

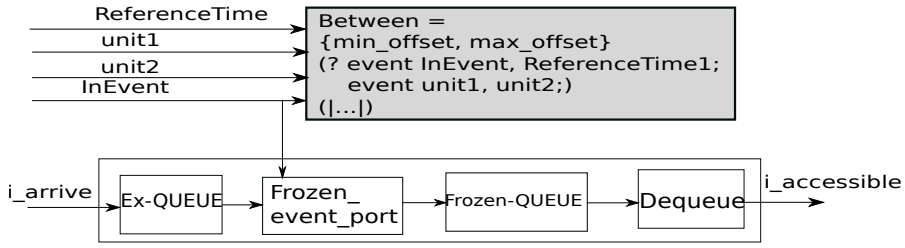

Figure 18: In event port

```
(? ID_type fifo_send_ID, fifo_receive_ID; event Move_time;)
(| Move_all_FIFO{default_msg}(fifo_send_ID, fifo_receive_ID, Move_time)
 |);

process Dequeue_event =
{integer dequeue_number; Data_type default_msg;}
(? ID_type fifo_ID;
 ! [dequeue_number]Data_type dequeue_sequence;)
(| dequeue_sequence := Out_FIFO{dequeue_number,default_msg}(fifo_ID)
 |)
where
  FIFO_type fifo;
end;

process Frozen_event_data_port =
{Data_type default_msg;}
(? ID_type fifo_send_ID, fifo_receive_ID; event Move_time;)
(| Move_all_FIFO{default_msg}(fifo_send_ID, fifo_receive_ID, Move_time)
 |);

process Dequeue_event_data =
{integer dequeue_number; Data_type default_msg;}
(? ID_type fifo_ID;
 ! [dequeue_number]Data_type dequeue_sequence;)
(| dequeue_sequence := Out_FIFO{dequeue_number,default_msg}(fifo_ID)
 |)
where
  FIFO_type fifo;
end;
```

The *dequeue_number* k is decided by **Dequeue_Protocol** and **Dequeued_Items** property. The detailed interpretation of these two properties could be found in 14.4.

Dequeue_Protocol	*Dequeue_number*
AllItems	actual elements number of *EX-FIFO*
MultipleItems	value of **Dequeued_Items**
OneItem	1

(a) **In_event_port**

$$\text{Let } x = (x_1, \{x_2, x_3, x_4, x_5, \dots\}) \in In\_event\_port,$$
$$\text{where : } x_1 \in portID$$
$$x_2 = Input\_Time \in Port\_property$$
$$x_3 = Queue\_Size \in Port\_property$$
$$x_4 = Dequeue\_Protocol \in Port\_property$$
$$x_5 = Dequeue\_Items \in Port\_property$$

A notation $EventPortTranslation()$ represents the translation from AADL to Signal.

Let data port x is in the context of a component $\mathscr{C}$. An in event port in a context $\mathscr{C}$ is presented as an instance of a Signal process.

In case of Input_Time contains only one reference time. $Count(V(x_2)) = 1$.

$$\text{Let } x2 = (x_{21},\ x_{22}, x_{23},\ x_{24}, x_{25}) \in Input\_Time$$
$$\text{where: } x_{21}, x_{23} \in aadlinteger$$
$$x_{22}, x_{24} \in Time\_Unit$$
$$x_{25} \in IO\_Reference\_Time$$

$$\mathbf{EventPortTranslation}(x, \mathscr{C}) =$$
$$x'_1 :: i\_accessible := InEventPort\{m, n, size, k\}$$
$$(i\_arrive,\ InEvent,\ Reference\_Time,\ unit1,\ unit2)$$

where i_arrive, $InEvent$, $Reference\_Time$, $unit1$, $unit2$ come from the context $\mathscr{C}$

$$x'_1 = \mathbf{IDTranslation}(x_1) = x_1$$
$$m = \mathbf{PropertyTranslation}(x_{21})$$
$$n = \mathbf{PropertyTranslation}(x_{23})$$
$$size = \mathbf{PropertyTranslation}(x_3)$$
$$k = \begin{cases} size, & \text{if } V(x_4) = \text{AllItems} \\ \mathbf{PropertyTranslation}(x_5), & \text{if } V(x_4) = \text{MultiItems} \\ 1, & \text{if } V(x_4) = \text{OneItem} \end{cases}$$
$$Reference\_Time = \begin{cases} Dispatch, & \text{if } V(x_{25}) = \text{Dispatch} \\ Start, & \text{if } V(x_{25}) = \text{Start} \\ Complete, & \text{if } V(x_{25}) = \text{Complete} \end{cases}$$
$$unit1 = \mathbf{PropertyTranslation}(x_{22})$$
$$unit2 = \mathbf{PropertyTranslation}(x_{24})$$

The *InEventPort* process is defined in library AADL_EVENTPORT. (It is not allowed to have an array of events, we use an array of Data_type (integer) instead: whenever a value exists, it represents a corresponding event.)

```
process InEventPort =
{integer min_offset, max_offset, size, dequeue_number;
 Data_type def_msg;}
(? event i_arrive;
   event InEvent, ReferenceTime, unit1, unit2; string Port_name;
 ! [dequeue_number]Data_type i_accessible;)
(| ex_FIFO_ID := CREATE_FIFO("ex_FIFO", size, def_msg)
 | frozen_FIFO_ID := CREATE_FIFO("frozen_FIFO", size, def_msg)
 | x_in[0] := DEFAULT_MSG when ^i_arrive
 | In_FIFO{1}(ex_FIFO_ID,x_in)
 | e1:: Frozen_event_port{def_msg}(ex_FIFO_ID, frozen_FIFO_ID, InEvent)
 | Between{min_offset, max_offset}(InEvent, ReferenceTime, unit1, unit2)
 | e2:: i_accessible :=
           Dequeue_event{dequeue_number, def_msg}(frozen_FIFO_ID)
 | e1-->e2
 |)
where
  ID_type ex_FIFO_ID, frozen_FIFO_ID;
  [1]Data_type x_in;
  label e1, e2;
end;
```

The *Dequeue_Items* (x_5) and *Queue_Size* (x_3) property associations are interpreted in the property section 14.4.

In case of Input_Time contains a list of reference time. $Count(V(x_2)) = 1$. This case is not considered yet.

(b) **In_event_data_port**

$$\text{Let } x = (x_1, \{x_2, x_3, x_4, x_5, \dots\}, x_6) \in In\_event\_data\_port,$$

where : $x_1 \in portID$

$x_2 = Input\_Time \in Port\_property$

$x_3 = Queue\_Size \in Port\_property$

$x_4 = Dequeue\_Protocol \in Port\_property$

$x_5 = Dequeue\_Items \in Port\_property$

$x_6 \in Data\_reference$

The event data port interpretation is almost the same as the event port translation, except that an interpretation of the data type is added, and the management of data (we could directly use array of data).

A notation $EventDataPortTranslation()$ represents the translation from AADL to Signal. Let data port x is in the context of a component $\mathscr{C}$. An in event port in a context $\mathscr{C}$ is presented as an instance of a Signal process.

In case of Input_Time contains only one reference time. $Count(V(x_2))$ = 1.

$$\begin{aligned}\text{Let } x2 &= (x_{21},\ x_{22}, x_{23},\ x_{24}, x_{25}) \in Input\_Time\\ \text{where: } x_{21}, x_{23} &\in aadlinteger\\ x_{22}, x_{24} &\in Time\_Unit\\ x_{25} &\in IO\_Reference\_Time\end{aligned}$$

$$\begin{aligned}&\mathbf{EventDataPortTranslation}(x, \mathscr{C}) \ =\\ &\quad x'_1 :: i\_accessible := InEventPort\{m, n, size, k\}\\ &\quad (i\_arrive,\ InEvent,\ Reference\_Time,\ unit1,\ unit2)\end{aligned}$$

where i_arrive, $InEvent$, $Reference\_Time$, $unit1$, $unit2$ come from the context $\mathscr{C}$

$$Reference\_Time = \begin{cases} Dispatch, & \text{if } V(x_{25}) = \text{Dispatch}\\ Start, & \text{if } V(x_{25}) = \text{Start}\\ Complete, & \text{if } V(x_{25}) = \text{Complete}\end{cases}$$

$$m = \mathbf{PropertyTranslation}(x_{21})$$

$$n = \mathbf{PropertyTranslation}(x_{23})$$

$$size = \mathbf{PropertyTranslation}(x_3)$$

$$k = \begin{cases} size, & \text{if } V(x_4) = \text{AllItems}\\ \mathbf{PropertyTranslation}(x_5), & \text{if } V(x_4) = \text{MultiItems}\\ 1, & \text{if } V(x_4) = \text{OneItem}\end{cases}$$

$$x'_1 = \mathbf{IDTranslation}(x_1) = x_1$$

$$unit1 = \mathbf{PropertyTranslation}(x_{22})$$

$$unit2 = \mathbf{PropertyTranslation}(x_{24})$$

$$Data\_type = \mathbf{DataReferenceTranslation}(x_2)$$

The *InEventDataPort* process is defined in library AADL_EVENTDATAPORT.

```
process InEventDataPort =
{integer min_offset, max_offset, size, dequeue_number; Data_type def_msg;}
(? Data_type i_arrive;
   event InEvent, ReferenceTime, unit1, unit2; string Port_name;
 ! [dequeue_number]Data_type i_accessible;)
(| ex_FIFO_ID := CREATE_FIFO("ex_FIFO", size, def_msg)
 | frozen_FIFO_ID := CREATE_FIFO("frozen_FIFO", size, def_msg)
 | x_in[0] := i_arrive
 | In_FIFO{1}(ex_FIFO_ID,x_in)
 | e1:: Frozen_event_data_port{def_msg}(ex_FIFO_ID, frozen_FIFO_ID, InEvent)
 | Between{min_offset, max_offset}(InEvent, ReferenceTime, unit1, unit2)
 | e2:: i_accessible :=
          Dequeue_event_data{dequeue_number, def_msg}(frozen_FIFO_ID)
```

```
 | e1-->e2
 |)
where
  ID_type ex_FIFO_ID, frozen_FIFO_ID;
  [1]Data_type x_in;
  label e1, e2;
end;
```

where $data\_type$ = $\mathbf{IDTranslation}(x_6)$

In case of Input_Time contains a list of reference time. $Count(V(x_2))$ = 1. Use *SeveralBetween* as time constraint.

5. Out event (event data) port and Polychrony

   The Output is stored in a out-QUEUE, and sent out at *OutEvent* time (Figure 19). The *OutEvent* is restricted by a constraint *Between* (or *SeveralBetween*). A *Distributer* selects the recipients depending on the **Fan_Out_Policy**.

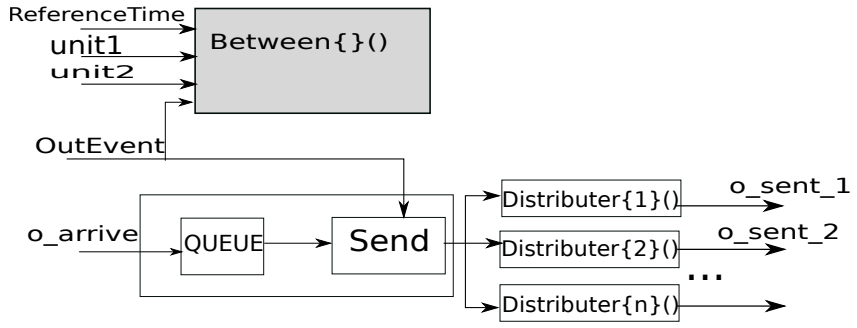

Figure 19: Out event port

The Send process is defined in library AADL_EVENTPORT and AADL_EVENTDATAPORT.

```
process Send_event =
{Data_type def_msg;}
(? ID_type fifo_ID; event OutEvent;
 ! event o;)
  (| x_out:= Out_FIFO{1, def_msg}(AT(fifo_ID,OutEvent))
   | o := when ^x_out[0]
   | )
where
  [1]Data_type x_out;
end;

process Send_event_data =
{Data_type def_msg;}
(? ID_type fifo_ID; event OutEvent;
 ! o;)
  (| x_out:= Out_FIFO{1, def_msg}(AT(fifo_ID,OutEvent))
   | o := x_out[0]
```

```
   | )
where
  [1]Data_type x_out;
end;
```

(a) **Out_event_port**

Let $x = (x_1, \{x_2, x_3, x_4 \dots\}) \in Out\_event\_port$
where: $x_1 \in portID$
$x_2 \in Output\_Time$
$x_3 \in Queue\_Size$
$x_4 \in Fan\_Out\_Policy$

In case of Output_Time contains only one reference time. $Count(V(x_2)) = 1$.

Let $x2 = (x_{21},\ x_{22}, x_{23},\ x_{24}, x_{25}) \in Input\_Time$
where: $x_{21}, x_{23} \in aadlinteger$
$x_{22}, x_{24} \in Time\_Unit$
$x_{25} \in IO\_Reference\_Time$

$\mathbf{EventPortTranslation}(x, \mathscr{C}) =$
$x'_1 :: OutEventPort\{m, n, size\}$
$(o\_arrive,\ OutEvent,\ Reference\_Time,\ unit1,\ unit2)$
where $o\_arrive,\ InEvent,\ Reference\_Time,\ unit1,\ unit2$ come from the context $\mathscr{C}$

$$Reference\_Time = \begin{cases} Dispatch, & \text{if } V(x_{25}) = \text{Dispatch} \\ Start, & \text{if } V(x_{25}) = \text{Start} \\ Complete, & \text{if } V(x_{25}) = \text{Complete} \end{cases}$$

$m = \mathbf{PropertyTranslation}(x_{21})$
$n = \mathbf{PropertyTranslation}(x_{23})$
$size = \mathbf{PropertyTranslation}(x_3)$
$x'_1 = \mathbf{IDTranslation}(x_1) = x_1$
$unit1 = \mathbf{PropertyTranslation}(x_{22})$
$unit2 = \mathbf{PropertyTranslation}(x_{24})$

The *OutEventPort* process is defined in library AADL_EVENTPORT.

```
process OutEventPort =
{ integer min_offset, max_offset, size; Data_type def_msg;}
  (?  event o_arrive;
      event OutEvent, Reference_Time, unit1, unit2; string Port_name;
   !  event o_accessible;)
  (| out_FIFO_ID := CREATE_FIFO("out_FIFO", size, def_msg)
   | x_out[0] := DEFAULT_MSG when ^o_arrive
   | In_FIFO{1}(out_FIFO_ID,x_out)
   | o_accessible := Send_event{def_msg}(out_FIFO_ID, OutEvent)
   | Between{min_offset, max_offset}
           (OutEvent, Reference_Time, unit1, unit2)
   | )
   where
     o1;
     ID_type out_FIFO_ID;
     [1]Data_type x_out;
   end;
```

The *Dequeue_Items* (x_5) and *Queue_Size* (x_3) property associations are interpreted in the property section 14.4.

The Distributers are added to link the out event port and its associated connections. The details interpretation of out port and Fan_Out_Policy have been introduced in Section 11.1.3.

In case of Onput_Time contains a list of reference time. $Count(V(x_2)) = 1$. This case is left for further study.

(b) **Out_event_data_port**

$$\text{Let } x = (x_1, \{x_2, x_3, x_4, \dots\}, x_5) \in Out\_event\_data\_port$$
$$\text{where: } x_1 \in portID$$
$$x_2 = Output\_Time \in Port\_property$$
$$x_3 = Queue\_Size \in Port\_property$$
$$x_4 = Fan\_Out\_Policy \in Port\_property$$
$$x_5 \in Data\_reference$$

The out event data port interpretation is similar to the out event port translation, except that an interpretation of the data type is added.

$$\textbf{EventDataPortTranslation}(x, \mathscr{C}) =$$
$$x_1' :: i\_accessible := OutEventDataPort\{m, n, size\}$$
$$(o\_arrive,\ OutEvent,\ Reference\_Time,\ unit1,\ unit2)$$

where o_arrive, $InEvent$, $Reference\_Time$, $unit1$, $unit2$ come from the context $\mathscr{C}$

$$Reference\_Time = \begin{cases} Dispatch, & \text{if } V(x_{25}) = \text{Dispatch} \\ Start, & \text{if } V(x_{25}) = \text{Start} \\ Complete, & \text{if } V(x_{25}) = \text{Complete} \end{cases}$$

$$m = \textbf{PropertyTranslation}(x_{21})$$
$$n = \textbf{PropertyTranslation}(x_{23})$$
$$size = \textbf{PropertyTranslation}(x_3)$$
$$x_1' = \textbf{IDTranslation}(x_1) = x_1$$
$$unit1 = \textbf{PropertyTranslation}(x_{22})$$
$$unit2 = \textbf{PropertyTranslation}(x_{24})$$
$$data\_type = \textbf{IDTranslation}(x_5)$$

The process OutEventDataPort() is defined in library AADL_EVENTDATAPORT.

```
process OutEventDataPort =
{ integer min_offset, max_offset, size; Data_type def_msg;}
  (?  Data_type o_arrive;
      event OutEvent, Reference_Time, unit1, unit2;
      string Port_name;
   !  o_accessible;)
  (| out_FIFO_ID := CREATE_FIFO("out_FIFO", size, def_msg)
   | x_out[0] := o_arrive
   | In_FIFO{1}(out_FIFO_ID,x_out)
   | o_accessible := Send_event_data{def_msg}(out_FIFO_ID, OutEvent)
   | Between{min_offset, max_offset}
           (OutEvent, Reference_Time, unit1, unit2)
   | )
   where
     o1;
     ID_type out_FIFO_ID;
     [1]Data_type x_out;
   end;
```

In case of Output_Time contains a list of reference time. $Count(V(x_2)) =$ 1. Not considered yet.

6. In out event (event data) port and Polychrony Separated as in and out event (event data) ports?