ESPRESSO AADL Digest Report

ESPRESSO Team

March 23, 2011

Foreword

This document is a draft that contains many copy/past from SAE AS5506A [2]. Its purpose is to propose a synthetic view of AADL behavior aspects, asserted by uninterpreted citations, and related to Polychronous model.

Paragraph formats

The following formats are used:

This is a citation extracted from SAE AS5506A.

This is a property definition.

This is a comment/proposal related to Signal/Polychrony.

This is a short specific comment or question.

Note: the notation "+" represents alternative choice, "**opt**()" represents optional and "**list**()" represents repeatable.

Contents

1 AADL purpose and organization

(1.1(1)) The purpose of the AADL is to provide a standard and sufficiently precise (machine-processable) way of modeling the architecture of an embedded, real-time system, such as an avionics system or automotive control system, to permit analysis of its properties, and to support the predictable integration of its implementation...

(1(8))...The standard specifies relevant characteristics of the detailed design and implementation descriptions, such as source text written in a programming language or hardware description language, from an external (black box) perspective. These relevant characteristics are specified as AADL component properties, and as rules of conformance between the properties and the described components.

(1.1(2)) The AADL describes application software and execution platform components of a system, and the way in which components are assembled to form a complete system or subsystem. The language addresses the needs of system developers in that it can describe common functional (control and data flow) interfacing idioms as well as performance-critical aspects relating to timing, resource allocation, fault-tolerance, safety and certification.

2 AADL components, packages and annexes

2.1 Specification

(3(2)) An AADL specification consists of global AADL declarations and AADL declarations. The global AADL declarations are comprised of package specifications that contain globally accessible AADL declarations and property set declarations. AADL declarations include component types, component implementations, feature group types, and annex libraries. AADL declarations can be declared in packages and are therefore accessible to other packages, or they can be declared directly in an AADL specification and not be accessible to packages...

Abstract syntax

$$AADL\_specification \ ::= \ Package\_spec + Property\_set$$

2.2 Package

(4.2(1)) A package provides a way to organize component types, component implementations, feature group types, and annex libraries into related sets of declarations by introducing separate namespaces. Package names built using identifiers separated by double colons (“::) ... In other words, complete_sys :: first_independent ::

fuel_flow is distinct from complete_sys ::second_independent ::fuel_flow. Packages cannot be declared inside other packages.

(4.1(1)) ...The content of packages, e.g., classifiers, can be referenced from anywhere by qualifying the classifier reference with the package name. The content of property sets, i.e., property type, property constant and property definitions, can be referenced from within anywhere by qualifying the property type, constant, or property reference with the property set name. Component classifiers, feature group types, and annex libraries that are declared directly in an AADLspecification are anonymous declarations. They are considered to reside in a local name space and can only be referenced by another anonymous declaration.

Abstract syntax

$$
\begin{aligned}
&Package\_spec \ ::= \ packageID \times \mathbf{opt}(Public\_package\_declarations) \\
&\quad \times \mathbf{opt}(Private\_package\_declarations) \times \mathbf{opt}(\mathbf{list}(Property)) \\
&Package\_declarations \ ::= \ Package\_category \times \mathbf{list}(Name\_Visibility\_declaration) \\
&\quad \times \mathbf{list}(AADL\_declaration) \\
&Package\_category \ ::= \{private, public\} \\
&Private\_package\_declarations \ ::= \ Package\_declarations([Package\_category = private]) \\
&Public\_package\_declarations \ ::= \ Package\_declarations([Package\_category = public]) \\
&AADL\_declaration \ ::= \ Classifier\_declaration + Annex\_library \\
&Classifier\_declaration \ ::= \ Software + Execution\_platform + Composite \\
&Software \ ::= \ Data + Subprogram + Subprogram\_group + Thread \\
&\quad + Thread\_group + Process \\
&Execution\_platform \ ::= \ Memory + Processor + Bus + Device \\
&\quad + Virtual\_processor + Virtual\_bus \\
&Composite \ ::= \ System \\
&Name\_Visibility\_declaration \ ::= \ Import\_declaration + Alias\_declaration \\
&Import\_declaration \ ::= \ \mathbf{list}((Package\_name + property\_set\_ID)) \\
&Alias\_declaration \ ::= \ defining\_ID \times Package\_name \\
&Annex\_library \ ::= \ annexID \times Annex\_spec \\
&Property \ ::= \ property\_name\_ID \times Assignment \times In\_binding \\
&Assignment \ ::= \ property\_value \\
&In\_binding \ ::= \ \mathbf{list}(platform\_component\_reference)
\end{aligned}
$$

2.3 Components

(4(2)) A component represents some hardware or software entity that is part of a system being modeled in AADL. A component has a component type, which defines a functional interface.

(4.4(1)) ... Every component implementation is associated with a component type. A component type may have zero or more component implementations declared.

(4(6)) Components can be declared in terms of other components by refining and extending existing component types and component implementations. This permits partially complete component type and implementation declarations to act as templates that may have explicit parameter (prototype) specifications. Such templates can represent a common basis for the evolution of a family of related component types and implementations.

(4(2)) The component type acts as the specification of a component that other components can operate against. It consists of features, flows, and property associations.

(4(3)) A feature models a characteristic of a component that is visible to other components. Features are named, externally visible parts of the component type, and are used to exchange control and data via connections with other components...

(8(1)) ...The four categories of features are: port, subprogram, parameters, and subcomponent access.

(8.1(1)) Feature groups represent groups of component features or feature groups.

(10(1)) A flow is a logical flow of data and control through a sequence of threads, processors, devices, and port connections or data access connections. A component can have a flow specification, which specifies whether a component is a flow source, i.e., the flow starts within the component, a flow sink, i.e., the flow ends within the component, or there exists a flow path through the component, i.e., from one of its incoming ports to one of its outgoing ports.

(10(2)) The purpose of providing the capability of specifying end-to-end flows is to support various forms of flow analysis, such as end-to-end timing and latency, reliability, numerical error propagation, Quality of Service (QoS) and resource management based on operational flows...

(3(4)) A component implementation specifies an internal structure in terms of subcomponents, connections between the features of those subcomponents, flows across a sequence of subcomponents, modes to represent operational states, and properties.

(4(4)) ... Component implementations represent variants of a component that adhere to the same interface, but may have different property values and realiza-

tions... Subcomponents are instantiations of component classifiers, i.e., component types and implementations.

(3(12)) "Features and flow specifications of component types (...) subcomponents, connections, flows, and modes of component implementations may have incomplete specifications. These (...) act as templates that can be parameterized by specifying prototypes. These specifications may be later refined in (...) extensions with the completion of classifier references and property associations. Component type extensions can also introduce additional features, flow specifications, and properties. Such extensions can add new subcomponents, connections, flows, modes, and properties to component implementations.

2.3.1 Category of component (AADL 4.3, 4.4)

- **abstract**: generic that can be refined into 2...10 (3(2)).
- **Software components**
  1. **data**: represents static data in source text (3(15))
  2. **subprogram (- group)**: represents source text that is executed sequentially (3(17))
  3. **thread (- group)**: models concurrent tasks (3(18))
  4. **process**: models space partition in terms of virtual address spaces (3(20))
- **Execution platform components**
  1. **(virtual -) processor** (3(22))
  2. **memory** (3(24))
  3. **(virtual -) bus** (3(25))
  4. **device** (3(27))
- **Compositional components** system (3(28))

2.3.2 Prototype(4.7, for information)

(1) Prototypes represent parameters for component type, component implementation, and feature group type declarations. They allow classifiers to be supplied when a component type, component implementation, or feature group is being extended. The prototypes can be referenced in place of classifiers in feature declarations, in subcomponent declarations, and as prototype bindings. The latter allows parameterization via prototype to be propagated down the system hierarchy.

2.3.3 Component description (AADL 3, 4.3, 4.4)

In this document each of the component descriptions contains a structure table that lists the categories of elements that can belong or not to a component. The property is present in all components, and thus implicit in those tables.

1. **Component type (AADL 3, 4.3)**

   (4.3(1)) A component type specifies the external interface of a component that its implementations satisfy.

   (4.3(5)) Component types can declare prototypes, i.e., classifier parameters that are used in features. The prototype bindings are supplied when the component types is being extended or used in subcomponent declarations.

   Component type elements A component type specifies a functional interface in terms of:

   (a) **features** (3(6)) that can be
      i. **ports** (to support data/control directional flows)
      ii. **subprograms** (synchronous procedure call)
      iii. (shared) **access** to data, subprograms(- group), bus

   (b) **flow specification** (3(32)) (across a sequence of subcomponents)

   (c) **modes** (3(29)) represent operational states of components in the modeled physical system; a mode change can change the set of active components and connections.

   (d) **properties** (3(9))property has a name, a type and a value

   Syntax to be added

2. **Feature group (AADL 8.1, for information)**

   (8(3)) Feature groups represent groups of component features. Feature groups can contain feature groups. Feature groups can be used anywhere features can be used. Within a component, the features of a feature group can be connected to individually. Outside a component, feature groups can be connected as a single unit.

   (8.1(L2))A feature group type can be declared to be the inverse of another feature group type, as indicated by the reserved words inverse of and the name of a feature group type.

(8.1(5)) The inverse of reserved words of a feature group type declaration indicate that the feature group type represents the complement to the referenced feature group type.

Two feature group types are considered to complement each other if the following holds:

(8.1(L9)) The number of feature or feature groups contained in the feature group and its complement must be identical;

(8.1(L10)) Each of the declared features or feature groups in a feature group must be a pair-wise complement with that in the feature group complement, with pairs determined by declaration order....

(8.1(L11)) If both feature group types have zero features, then they are considered to complement each other;

(8.1(L12)) Ports are pair-wise complementary if they have complementary direction (out / in, in / out, in out / in out), and are of the same port type. In the case of event data or data ports, the data component classifier reference must be identical;

(8.1(L13)) Access features are pair-wise complementary if they have complementary access direction (requires / provides, provides / requires), and have matching access classifiers with the matching criteria being identity.

3. **Component implementation(AADL 3, 4.4)**

(4(4))...A component implementation specifies the realization of a component variant, i.e., an internal structure for a component as an assembly of subcomponents.

Syntax to be added

Subcomponent inclusion

(a) **Software components**

(b) **Execution platform components**

- (virtual) bus may contain virtual bus.
- Device may contain bus.
- Memory may contain memory, bus.
- Processor may contain virtual processor, memory, (virtual) bus.
- Virtual processor may contain virtual processor, virtual bus.

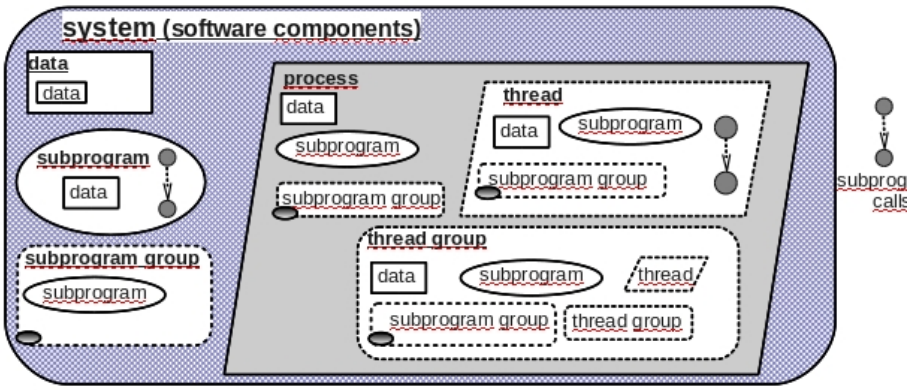


Figure 1: Subcomponent inclusion

(c) **System** may contain data, subprogram (group), process, (virtual) processor, memory, (virtual) bus, device, system

2.3.4 Connections

(3(31)) AADL connections specify patterns of control and data flow between individual components at runtime. A semantic connection can be made between

a data component and threads that access the data component for data access connections,

a subprogram component and threads that require call access to the subprogram,

two threads,

the event port of a thread, device, or processor and a mode transition for mode transition connections.

a thread and a device or processor for port connections,

a bus component and buses, memory, processor, and device components for bus access connections,

(3(31)) ...A semantic connection is represented by a set of one or more connection declarations that follow the component hierarchy from the ultimate connection source to the ultimate connection destination.

3 Data

(5.1(1))A data component type represents a data type in source text. The internal structure of a source text data type, e.g., the instance variables of a class or the fields of a record, is represented by data subcomponents in a data component implementation. Provides subprogram access features of a data component type can model the concept of methods on a class or operations on an abstract data type. If provides subprogram access features are declared, the data component may only be accessed through the subprograms...

A data component implementation represents the internal structure of a data component type.

A data subcomponent represents a data instance, i.e., data in the source text that is potentially sharable between threads and persists across thread dispatches.

(5.1(2)) A data component classifier, i.e., a data component type name or a data component type and implementation name pair (separated by a dot .), is used as data type indicator in port declarations, subprogram parameter declarations, and data subcomponent declarations.

(5.1(4)) References to data components are modeled through provides and requires data access. Threads, processes, systems, and subprogram may access data by reference.

3.1 Data component

- Data components classifiers represent data types.
- Data subcomponents represent static data in source text. Only those components that explicitly declare required data access can access such sharable data subcomponents. Data subcomponents can be shared within the same process and across processes (if supported by the runtime system).
- When declared in a subprogram, that data subcomponent represents a local variable. This data can not be made accessible outside the subprogram through a provides data access declaration.
- Data subcomponents that are not declared in subprograms can be shared between threads.
- References to data components are modeled through provides and requires data access.
- Data component classifier references are also used to specify the data type for data (event data) ports as well as subprogram parameters.

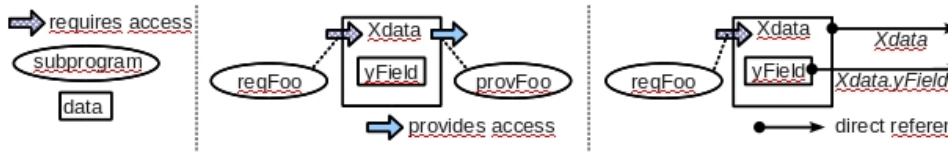


Figure 2: Data component graphical notation

Abstract syntax (5.1(10))A data component type can have zero data implementation.

The table is in contradiction with (L3) A data implementation can contain abstract, data and subprogram subcomponents, and data property associations.

A data type does not provide data access (to its subcomponents). This point is discussed in the draft. A port connection can be established between a port P and an element E in a provided data access DA to P by DA.E.

Type		**Implementation**	
Features	*Provides* subprog. access	**Subcomponents**	abstract
	Requires subprog. (- group) access		data
		Subprog. calls	*NO*
		Connections	access
Flow spec, Mode	*NO*	**Flows, Modes**	*NO*

Figure 3: Data type and implementation

$$Data \ ::= \ Data\_type + Data\_implementation$$

- **Data_type**

$$Data\_type ::= dataID \times \mathbf{opt}(\mathbf{list}(Data\_feature)) \times \mathbf{opt}(\mathbf{list}(Data\_property))$$
$$Data\_feature \ ::= \ Feature\_group + Provides\_subprogram\_access + Requires\_subprogram\_access + Requires\_subprogram\_group\_access$$

- **Data_implementation**

$$Data\_implementation ::= dataID \times \mathbf{opt}(\mathbf{list}(Data\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Data\_property))$$

$$Data\_subcomponent ::= subcomponentID \times Data\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$$

$$Data\_subcomponent\_reference ::= dataID + subprogramID$$

3.2 Standard properties

This section gives some standard properties that could be applied to data component.

1. Properties related to source text

   Type_Source_Name: **aadlstring applies to** (data, port, subprogram);
   Source_Name: **aadlstring applies to** (data, port, subprogram, parameter);
   Source_Text: **inherit list of aadlstring applies to**
   (data, port, subprogram, thread, thread group, process, system, memory, bus, device, processor, parameter, feature group, package);
   Source_Language: **inherit list of** Supported_Source_Languages

2. Properties specifying memory requirements

   Source_Data_Size: Size **applies to**
   (data, thread, thread group, process, system, subprogram, processor, device);
   Allowed_Memory_Binding_Class: **inherit list of classifier** (memory, system, processor)
   Allowed_Memory_Binding: **inherit list of reference** (memory, system, processor)
   Actual_Memory_Binding: **inherit list of reference** (memory)
   Base_Address: **aadlinteger** 0 .. Max_Base_Address
   Source_Code_Size: Size

3. Data sharing properties

   (a) **Access_Right**. This property specifies the form of access that is permitted for a component. It could be *read_only, write_only, read_write, by_method*. Default value is *read_write*.

   Access_Right : Access_Rights ⇒ read_write **applies to**
   (data, bus, data access, bus acess);
   Access_Rights : **type enumeration** (read_only, write_only, read_write, by_method);

(b) **Concurrency_Control_Protocol**. This property specifies the concurrency control protocol used to ensure mutually exclusive access to a shared data component.

(5) Shared data may be accessed by multiple threads. Such potential concurrent access is controlled according to the Concurrency_Control_Protocol. (PLG not specified)

```
Concurrency_Control_Protocol: Supported_Concurrency_Control_Protocols
    applies to (data);
Supported_Concurrency_Control_Protocols: type enumeration
    (None_Specified, < project-specified >);
```

Default value is *None_Specified*: no concurrency control protocol.

AADLv2 does not specify the detailed project-specified protocols, but gives some example concurrency control protocols: *Interrupt_Masking, Maximum_Priority, Priority_Inheritance, Priority_Ceiling, Spin_Lock* and *Semaphore*. [1] implemented four kinds of concurrency control protocol: *NoneSpecified, Lock, BIP, PCP*.

When a thread enters a critical region (when it is accessing a shared data component), a **Get_Resource** operation is performed on the shared data component. When it exit from a critical region, a **Release_Resource** operation is performed.

3.3 Data component access

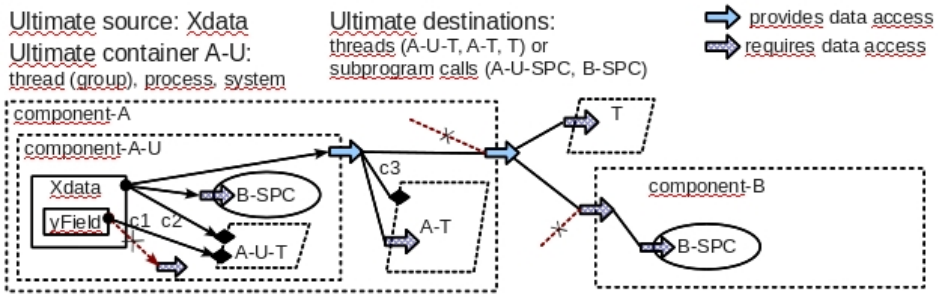


Figure 4: Data component access

c1, c2, c3 are port connections, others are data access connections. It is not clear if connecting data elements in a data to requires_data_access is possible: in the rules one can see data_subcomponent_identifier . provides_data_access_identifier,

but in the corresponding table, a data component does not provide data access to its subcomponents..

(6) Input_Time and Output_Time specify the time range over which a component has read or write access to a shared data component. The value of a shared data component is read or written through the use of a data variable that represents the shared data component, or through Get_Value and Put_Value service calls. Write access immediately updates the shared data component.

(7) Input_Rate and Output_Rate specify the rate at which a shared data component is accessed. The input rate specifies read accesses while the output rate specifies write accesses.

Yue: Input_Time and Output_Time properties are associated to ports, not to data. Maybe here the Access_Time property should be used, which is associated to data access and specifies the time range over which a component has access to a shared data component. (refer to Section 11.4 for detail)

3.4 Behavior: critical region

(5.1(17)) Concurrent access to shared data is coordinated according to the concurrency control protocol specified by the Concurrency_Control_Protocol property value associated with the data component. A thread is considered to be in a critical region when it is accessing a shared data component. When a thread enters a critical region a Get_Resource operation is performed on the shared data component. Upon exit from a critical region a Release_Resource operation is performed ... . If multiple data components with concurrency control protocols are accessed by a thread, the critical regions may be nested, i.e., the Get_Resource and Release_Resource operations are pair-wise nested for each data component. Furthermore, deadlock avoidance among threads accessing the same set of shared data components is assured by proper nesting of the critical regions across all of the threads.

The *Get_Resource* and *Release_Resource* runtime services represent an abstract interface for functions that perform locking and unlocking of resources according to the specified concurrency control protocol.

```
subprogram Get_Resource
features
  resource: in parameter <implementation-specific
            representation of one or more resources>;
end Get_Resource;

subprogram Release_Resource
features
```

```
  resource: in parameter <implementation-specific
            representation of one or more resources>;
end Release_Resource;
```

(5.1(30)) The concurrency control protocol can be implemented through a number of concurrency control mechanisms such as mutex, lock, semaphore, or priority ceiling protocol. Appropriate concurrency control state is associated with the shared data component to maintain concurrency control. The protocol implementation must provide appropriate implementations of the Get_Resource and Release_Resource operations.

(5.4.3 (42)) The time a thread resides in a critical region in worst case is the duration of executing one thread dispatch.

Supported_Concurrency_Control_Protocols are not defined by the standard. Examples are given in AADL A.2. By default there is no control protocol.

3.5 Data in Polychrony

TODO. Clarify semantics of Data ↔ Data event connections

3.5.1 Data type

Data type can be represented by a free clock signal or signal structure containing inner data as fields. The *provides subprogram access* features of a data component type can model the concept of methods on a class of operations on an abstract data type.

For example:

```
data Message
  features
    updata_message: provides subprogram access Update_address;
end Message;

data implementation Message.impl
  subcomponents
    name: data aadlstring;
    size: data aadlinteger;
    text: data aadlstring;
end Message.impl;

subprogram Update_address
  features
```

```
    message: in parameter Message;
end Update_address;
```

A corresponding Signal structure:

$$type\ Message\ =\ struct\ (\ string\ name;\ integer\ size;\ string\ text;\ );$$

Simple data There is no problem to represent data types in Signal: at worst external types can be used to represent AADL types.

There are two categories of Signal-signals: the free clock1 signals (constants, state variables and free variables2) and the clocked signals (the use of which can generate clock constraints). An AADL-data can be represented as a free clock signal.

The problem of multiple accesses during a logical instant exists as for other AADL features.

In this case:

$$\text{Let } x = (x_1, x_2, x_3) \in Data\_type$$
$$\text{where: } x_1 \in dataID,\quad x_2 \in Data\_feature,\quad x_3 \in Data\_property$$

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

Compound data A data that contains inner data can be represented by a free clock structure that contains inner data as fields.

In this case:

Let $x = (x_1, x_2, x_3) \in Data\_type,\ y = (y_1, y_2, y_3, y_4) \in Data\_omplementation$
where: $x_1 \in dataID$
$x_2 \in Data\_feature$
$x_3 \in Data\_property$
$y_1 \in dataID$
$y_2 \in Data\_subcomponent$
$y_3 \in Connection$
$y_4 \in Data\_property$
and: $x_1\ and\ y_1 are\ compatible$

$$\mathbf{DataTranslation}(x) = \mathbf{type}\ y'_1 = struct(y'_2)$$
$$y'_1 = \mathbf{IDTranslation}(y_1) = y_1$$
$$y'_2 = \mathbf{DataSubcomponentTranslation}(y_2)$$

Let $z = (z_1, z_2, z_3, z_4) \in Data\_subcomponent$
where: $z_1 \in subcomponentID$
$z_2 \in Data\_subcomponent\_reference$
$z_3 \in Property$
$z_4 \in In\_modes$

$$\mathbf{DataSubcomponentTranslation}(z) = \begin{cases} \mathbf{DataTranslation}(\mathbf{z_2})\ z'_1 & \text{if } z_2 \in dataID \\ \mathbf{SubprogramTranslation}(z_2)\ z'_1, & \text{if } z_2 \in subprogramI \end{cases}$$

$$z'_1 = \mathbf{IDTranslation}(z_1) = z_1$$

3.5.2 Data subcomponent (declared in a subprogram)

A local data in the subprogram. It could be accessible only inside the subprogram.

A data subcomponent declared in a subprogram could be represented as a local variable of the subprogram. This local variable contains a type (translated from the corresponding data type) and a name (comes from the reference name). (refer to Section 4 for detail.)

3.5.3 Protected data (data subcomponent not in a subprogram)

A Data that provides subprogram access can be represented by a (see) Signal server when concurrent access requires asynchronous features.

- A data subcomponent represents static data in the source text. Data in the source text that is sharable between threads.
- A data that provides subprogram access can be represented by a Signal server when concurrent access requires asynchronous features.

Data subcomponent declared in a thread or process It could be shared between threads. Figure 5 gives an example of two threads want to access a shared data. The three components are declared in a same process. *Thread* require data access by feature *requires_data_access*, and *Thread2* provides data access by feature *provides_data_access*.

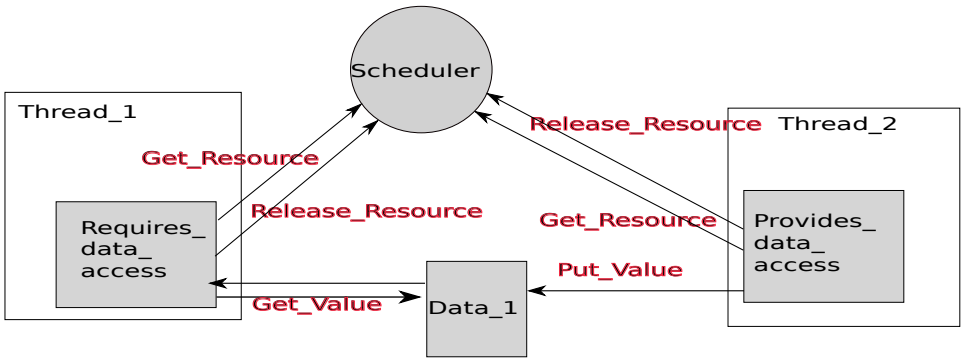


Figure 5: Data concurrency control

Two service calls *Get_Resource* and *Release_Resource* are performed to access the shared data. The *scheduler* will decide which thread could get the resource. *Get_Value* service call returns the current value, and *Put_value* updates the data value.

```
process Get_Resource(? resource; ! boolean return;)
process Release_Resource(? resource; ! boolean return;)
process Get_Value(? component; ! value;)
process Put_Value(? component; value;)
```

In this case, the data is interpreted as a Signal signal, and a shared resource control (a scheduler) is added.

4 Subprogram

(5.2-(1)) A subprogram component represents sequentially executed source text that is called with parameters. A subprogram may not have any state that persists beyond the call (static data). Subprograms can have local variables that are represented by data subcomponents in the subprogram implementation.

(8.3(4)) A subprogram that is accessed by more than one component is shared and must be reentrant. The shared subprogram may be called by multiple threads. This may result in concurrent access to shared data components.

A subprogram models callable source text that is executed sequentially. A subprogram may be called by multiple threads or subprograms.

4.1 Subprogram component

(5.2-(6)) A subprogram type declaration specifies all interactions of the subprogram with other parts of the application source text. Subprogram parameters are specified as features of a subprogram type .... This includes in and in out parameters passed into a subprogram and out and in out parameters returned from a subprogram on a call, events being raised from within the subprogram through its out event port and out event data port, required access to static data by the subprogram are specified as part of the features subclause of a subprogram type declaration, and required access to subprograms that are contained in another component and are called by this subprogram.

4.1.1 Structure

<table>
<tr><th colspan="2">Type</th><th colspan="2">Implementation</th></tr>
<tr><td rowspan="5">Features</td><td>Out event (-data) port</td><td rowspan="2">Subcomponents</td><td>Abstract</td></tr>
<tr><td>Feature group</td><td>Data</td></tr>
<tr><td>Requires data access</td><td colspan="2"></td></tr>
<tr><td>Requires subprog. (- group) access</td><td>Subprog. calls</td><td>yes</td></tr>
<tr><td>Parameter</td><td>Connections</td><td>yes</td></tr>
<tr><td>Flow spec, Mode</td><td>yes</td><td>Flows, Modes</td><td>yes</td></tr>
</table>

Figure 6: Subprogram structure

4.1.2 Abstract syntax

$$Subprogram \ ::= \ Subprogram\_type + Subprogram\_implementation$$

Subprogram_type

$$Subprogram\_type \ ::= \ subprogramID \times \mathbf{opt}(\mathbf{list}(Subprogram\_feature)) \times \mathbf{opt}(\mathbf{list}(Flow\_spec)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Subprogram\_property))$$

$$Subprogram\_feature \ ::= \ out\_event\_port + out\_event\_data\_port + Feature\_group + Requires\_data\_access + Requires\_subprogram\_access + Requires\_subprogram\_group\_access + Parameter$$

Subprogram_implementation

$$Subprogram\_implementation \ ::= \ subprogramID \times \mathbf{opt}(\mathbf{list}(Subprogram\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Subprogram\_call)) \times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Flow\_implementation)) \times \mathbf{opt}(\mathbf{list}(End\_to\_end\_flow)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Subprogram\_property))$$

$$Subprogram\_subcomponent \ ::= \ subcomponentID \times Subprogram\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$$

$$Subprogram\_subcomponent\_reference \ ::= \ dataID$$

1. *out_event_port* is a *Event_port* whose *port_direction* is *out*.
2. *out_event_data_port* is a *Event_data_port* whose *port_direction* is *out*.

4.1.3 Standard properties

- Properties related to source text

Source_Name
Source_Text
Source_Language
Type_Source_Name

- Properties specifying memory requirements

Source_Code_Size
Source_Data_Size
Source_Stack_Size
Source_Heap_Size
Allowed_Memory_Binding_Class
Allowed_Memory_Binding
Actual_Memory_Binding
Acceptable_Array_Size

- Execution related properties

Compute_Execution_Time: Time_Range
Compute_Deadline: Time
Client_Subprogram_Execution_Time: Time_Range

- Remote subprogram call related properties.

Subprogram_Call_Type specifies whether the call is to be performed synchronous or semi-synchronous. In case of a semi-synchronous call, the use of the result is may be suspended until the result is available.

Subprogram_Call_Type: **enumeration** (Synchronous, SemiSynchronous)
⇒ Synchronous **applies to** (subprogram);
Allowed_Subprogram_Call_Binding: **inherit list of reference**
(bus, processor, device) **applies to** (subprogram, thread,
thread group, process, system);
Actual_Subprogram_Call_Binding: **inherit list of reference**
(bus, processor, memory, device) **applies to** (subprogram);

- Other properties.

Urgency
Reference_Processor: **inherit classifier** (processor)
Classifier_Substitution_Rule

4.2 Subprogram call

5.2-(2)) Subprograms can be called from threads and from other subprograms. These calls are sequential calls local to the virtual address space of the thread. Subprograms can also be called remotely from threads in other virtual address spaces. A subprogram call sequence is declared in a thread implementation or in a subprogram implementation. Subprogram call sequences may be mode-specific. Subprogram calls may be local, i.e., to an instance of the subprogram in the same process as the caller, or they may be remote, i.e., to subprogram instances in other processes.

(5.2-(C2)) A subprogram call must reference a subprogram implementation.
(PLG: subprogram calls can be queued)

4.3 Behavior

(2) For parameter connections, data transfer occurs at the time of the subprogram call and call return. In the case of subprogram calls to remote subprograms, the data is first transferred to a local proxy and from there passed to the remote subprogram.

(5.2-(14)) Ordering of subprogram calls is by default the order of the subprogram call declarations. Annex-specific notations, e.g., the Behavior Annex, can be introduced to allow for other call order specifications, such as conditional calls and iterations.

(5.2-(15)) The flow of parameter values between subprogram calls as well as to and from ports of enclosing threads is specified through parameter connection declarations.

(5.2-(L3)) Only one subprogram call sequence can apply to a given mode. In other words, a mode identifier can be specified in the in_modes subclause of at most one subprogram call sequence.

((5.2-(19) A subprogram is executed within the calling AADL-thread or within a called component while calling AADL-thread is suspended. It is executed within a called component when the call refers to:

- Subprogram access to subprogram component in another AADL-thread,
- Subprogram access to a provides subprogram access feature in a device,
- Subprogram access of a processor (operating system),
- Subprogram classifier and the call has a subprogram call binding property that refers to provides subprogram access in other AADL-thread.

In all other cases execution remains within the calling AADL-thread.

4.4 Subprogram in Polychrony

If there is no recursive call, one can consider a subprogram as a standard aperiodic thread that has a dispatch event to which all calls are connected.

A subprogram seems to be a standard aperiodic AADL-thread with specific syntactic synchronous signals named parameters. A subprogram differs from a standard thread in the computing of C and T and the thread scheduling: when an AADL-thread TH1send values to a standard AADL-thread TH2, the execution of TH1 code is not necessary stopped. And TH2 cannot send values to TH1 in the

same period. At the opposite, if a thread TH1 send input parameters to a thread subprogram THS2, those parameters are immediately sent, TH1 is suspended (with its C remaining equal to 1) awaiting for output parameters from THS2 in the same period.

A parameter can be seen as a data port; the Input_Time of an input parameter is the dispatch event of THS2, the Output_Time of an output parameter is the complete event of THS2. Parameters are connected by immediate connection.

Multiple calls in the same logical instant are analogous to simultaneous arrivals of dispatching events in an aperiodic AADL-thread. To guarantee the correct synchronization of those parameters, the input parameters are grouped in a Signal structure type, and thus received as a single event data port named InParameter connected to the dispatch port of the thread. The same is done for out parameters grouped in ReturnResult. In out parameters are split on InParameter and ReturnResult fields. The InParameter and OutParameter have the suitable properties. They are connected with respect to expected calling behavior.

To check: is a Subprogram call considered as a dispatch event of the AADL-thread that provides the Subprogram access ? If not, when is the call executed ?

Finally a subprogram can be implemented as a Signal-procedure if such a feature is added to Signal.

A subprogram can be considered as a standard aperiodic thread that has a dispatch event to which all calls are connected.

A parameter can be seen as a data port: the **Input_Time** is dispatch, and **Output_Time** is complete.

Synchronous call The input parameters are grouped in a Signal structure, named *InParameter*, and the out parameters are grouped in *ReturnResult*. In out parameters are split on *InParameter* and *ReturnResult* fields. Figure 7.

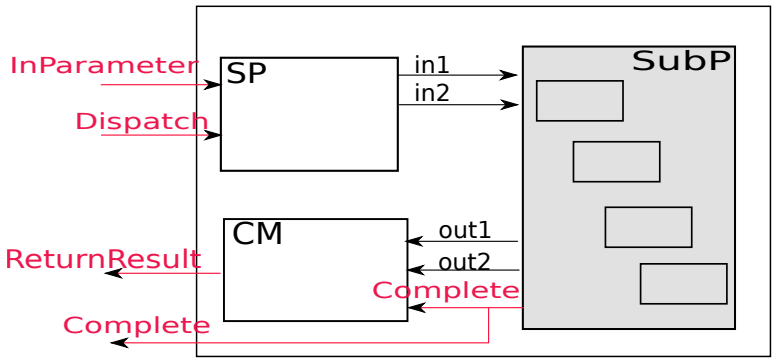


Figure 7: Subprogram

Semi-synchronous call The use of the result is may be suspended until the result is available. When?

5 Subprogram group

(5.3-(1)) Subprogram groups represent groups of subprogram features, i.e., libraries of subprograms. Subprogram groups can be made accessible to other components through subprogram group access features and subprogram group access connections. This grouping concept allows the number of connection declarations to be reduced, especially at higher levels of a system when a number of provided subprograms from one subcomponent and its contained subcomponents must be connected to requires subprogram access in another subcomponent and its contained subcomponents. The content of a subprogram group is declared through a subprogram group type declaration. This declaration is then referenced when subprogram groups are declared as subcomponents.

A subprogram represents a subprogram library.

5.1 Abstract syntax

$Subprogram\_group$ ::= $Subprogram\_group\_type + Subprogram\_group\_implementation$

Subprogram_group_type

$Subprogram\_group\_type$ ::= $subprogram\_group\_ID \times \mathbf{opt}(\mathbf{list}(Subprogram\_group\_feature))$
$\times \mathbf{opt}(\mathbf{list}(Subprogram\_group\_property))$
$Subprogram\_group\_feature$ ::= $Feature\_group + Subprogram\_access$
$+ Requires\_subprogram\_group\_access$

Subprogram_group_implementation

$Subprogram\_group\_implementation$ ::= $subprogram\_group\_ID$
$\times \mathbf{opt}(\mathbf{list}(Subprogram\_group\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Connection))$
$\times \mathbf{opt}(\mathbf{list}(Subprogram\_group\_property))$
$Subprogram\_group\_subcomponent$ ::= $subcomponentID$
$\times Subprogram\_group\_subcomponent\_reference$
$\times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$
$Subprogram\_group\_subcomponent\_reference$ ::= $subprogramID$

5.2 Standard properties

1. Reference_Processor: **inherit classifier** (processor)

6 Thread

(5.4-(1)) A thread represents a sequential flow of control that executes instructions within a binary image produced from source text. A thread models a schedulable unit that transitions between various scheduling states. A thread always executes within the virtual address space of a process, i.e., the binary images making up the virtual address space must be loaded before any thread can execute in that virtual address space.

6.1 Structure

(from 5.4-(2)) An AADL-thread contains a predeclared in event port named Dispatch, and two predeclared out event ports named Complete and Error; those ports cannot be user-declared (L3). As other ports, they may be connected (or not).

Type		*Implementation*	
Features	port	**Subcomponents**	abstract
	Feature group		data
	Provides, Requires data access		subprogram(- group)
	Provides, Requires subprog. (- group) access	**Subprog. calls**	*yes*
		Connections	*yes*
Flow spec, Mode	*yes*	**Flows, Modes**	*yes*

Figure 8: Thread structure

6.2 Abstract syntax

$$Thread ::= Thread\_type + Thread\_implementation$$

Thread_type

$$Thread\_type ::= threadID \times \mathbf{opt}(\mathbf{list}(Thread\_feature)) \times \mathbf{opt}(\mathbf{list}(Flow\_spec)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Thread\_property))$$

$$Thread\_feature ::= Port + Feature\_group + Data\_access + Subprogram\_access + Subprogram\_group\_access$$

Thread_implementation

$$Thread\_implementation ::= threadID \times \mathbf{opt}(\mathbf{list}(Thread\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Subprogram\_call)) \times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Flow\_implementation)) \times \mathbf{opt}(\mathbf{list}(End\_to\_end\_flow)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Thread\_property)) \times \mathbf{opt}(\mathbf{list}(Annex\_subclause))$$

$$Thread\_subcomponent ::= subcomponentID \times Thread\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$$

$$Thread\_subcomponent\_reference ::= dataID + subprogramID + subprogram\_group\_ID$$

$$Subprogram\_call ::= subprogram\_call\_ID \times Called\_subprogram \times \mathbf{opt}(\mathbf{list}(Subprogram\_call\_property))$$

$$Called\_subprogram ::= subprogramID + provides\_subprogram\_access\_ID + requires\_subprogram\_access\_ID$$

$$Annex\_subclause ::= annexID \times Annex\_spec \times \mathbf{opt}(In\_modes)$$

6.3 Standard properties

This section gives a brief view of standard properties that are related to thread. Some of them are also available for thread groups.

Properties		Thread	Thread group
Deployment	Allowed_{Processor, Memory, Connection}_Binding_Class	X	X
	Allowed_{Processor, Memory, Connection}_Binding	X	X
	Actual_{Processor, Memory, Connection}_Binding	X	X
	Allowed_Subprogram_Call_Binding	X	X
Thread	Dispatch_Protocol	X	
	Dispatch_Trigger	X	
	POSIX_Scheduling_Policy	X	X
	Priority	X	X
	Criticality	X	X
	Time_Slot	X	X
	Resumption_Policy	X	X
	Active_Thread_Handling_Protocol	X	X
	Active_Thread_Queue_Handling_Protocol	X	X
	Synchronized_Component	X	X
Timing	{Active, Compute, Deactivate}_Deadline	X	
	{Active, Compute, Deactivate}_Execution_Time	X	
	Deadline	X	X
	First_Dispatch_Time	X	X
	Dispatch_Jitter	X	X
	Dispatch_Offset	X	
	{Finalize, Initialize, Recover}_Deadline	X	
	{Finalize, Initialize, Recover}_Execution_Time	X	
	Period	X	X
	Reference_Processor	X	X
Memory	Source_Code_Size	X	X
	Source_Data_Size	X	X
	Source_Heap_Size	X	
	Source_Stack_Size	X	
Programming	{Activate,Compute, Deactivate}_Entrypoint	X	
	{Activate,Compute, Deactivate}_Entrypoint_Call_Sequence	X	
	{Activate,Compute, Deactivate}_Entrypoint_Source_Text	X	
	{Finalize,Initialize, Recover}_Entrypoint	X	
	{Finalize,Initialize, Recover}_Entrypoint_Call_Sequence	X	
	{Finalize,Initialize, Recover}_Entrypoint_Source_Text	X	
	Source_Language	X	X
	Source_Text	X	X

The table gives all the predeclared thread properties. Some of them will be taken into account in our translation.

1. **Dispatch_Protocol** specifies the dispatch behavior for a thread.

   Dispatch_Protocol: Supported_Dispatch_Protocols

2. **Period** (mandatory if Dispatch_Protocol is periodic or sporadic)

   Period: **inherit** Time **applies to**
   (thread, thread group, process, system, device, virtual processor);

3. Dispatch_Offset: Time (only if Dispatch_Protocol is periodic)

4. **Deadline** specifies the maximum amount of time allowed between a thread dispatch and the time that thread begins waiting for another dispatch.

   Deadline: **inherit** Time $Rightarrow$ Period
   applies to (thread, thread group, process, system, device);

5. **Priority** specifies the priority of the thread that is taken into consideration by some scheduling protocols in scheduling the execution order of threads.

   Priority: **inherit aadlinteger applies to**
   (thread, thread group, process, system, device);

6. Properties specifying execution entrypoints and timing constraints: those properties are defined for STEP in Initialize, Compute, Activate, Deactivate, Recover, Finalize

   STEP_Execution_Time: Time_Range
   STEP_Deadline: Time
   STEP_Entrypoint, STEP_Entrypoint_{Call_Sequence, Source_Text}

7. Properties related to mode switching and scheduling

   Synchronized_Component: **inherit aadlboolean** ⇒ true
   Active_Thread_Handling_Protocol: **inherit**
   Supported_Active_Thread_Handling_Protocols ⇒ abort
   Active_Thread_Queue_Handling_Protocol: **inherit enumeration**
   (flush, hold) ⇒ flush
   Activation_Mode: **enumeration** (initial, resume)

6.4 Behavior

(5.4-(2))... A thread can be active in a particular mode and inactive in another mode. As a result a thread may transition between an active and inactive state as part of a mode switch. Only active threads can be dispatched and scheduled for execution. Threads can be dispatched periodically or as the result of explicitly modeled events

that arrive at event ports, event data ports, or at a predeclared in event port called Dispatch. Completion of the execution of a thread dispatch will result in an event being delivered through the predeclared Complete event out port if it is connected.

(5.4-(3)) If the thread execution results in a fault that is detected, the source text may handle the error. If the error is not handled in the source text, the thread is requested to recover and prepare for the next dispatch. If an error is considered thread unrecoverable, its occurrence is propagated as an event through the predeclared Error out event data port.

(5.4.3 (39)) A scheduler selects one thread from the set of threads in the ready state to run on one processor according to a specified scheduling protocol. It ensures that only one thread is in the running state on a particular processor.

6.4.1 Predeclared ports

- **Dispatch**: If this port is connected, (ie is a destination in a connection), then the arrival of an event results in the dispatch of the AADL-thread. Events arriving on other (data-) event do not dispatch the AADL-process but are queued. (PLG: Dispatch event Overflow_Handling_Protocol cannot be defined ??)
- **Complete**: If this port is connected, an event is raised when the execution of the AADL-thread completes. (PLG: no possible overflow)
- **Error**: If this port is connected, an event is raised when an unrecoverable error is detected.(PLG: execution is stopped; no possible overflow)

6.4.2 Real time counters

An AADL-thread THREAD holds two timing values: C which is its actual execution time, and T which is its elapsed time. C and T are times in the reference time of the processor (PROC) THREAD executes on . The actual execution time is the time accumulating while THREAD actually runs on PROC; the elapsed time is the time accumulating since the last dispatch of THREAD. In nominal behavior, C and T are reset to 0 when the AADL-process is dispatched (C:=0, T:=0 in automata), C continuously increases when THREAD is computing (δC=1 in automata), T continuously increases until THREAD completion (δT=1 in automata) (PLG: there is here some personal interpretation concerning T). δX=0 means that X remains unchanged.

6.4.3 Dispatch_Protocol

(5.4.1(28)) The Dispatch_Protocol property of a thread determines the characteristics of dispatch requests to the thread. The Enabled function determines when a transition to performing thread computation will occur. The Wait_For_Dispatch invariant captures the condition under which the Enabled function is evaluated. The consequence of a dispatch is the execution of the entrypoint source text code sequence Subprogram access at its current execution position. This position is set to the first step in the code sequence and reset upon completion.

(5.4.1(16)) ...If a dispatch request is received for a thread while the thread is in the compute state, this dispatch request is handled according to the specified Overflow_Handling_Protocol for the event or event data port of the thread.

An AADL-thread THREAD can have one of the following Dispatch_Protocols:

1. **periodic**(29,30): a dispatch request is issued to THREAD at time intervals of the specified Period property value. THREAD can have a **Dispatch_Offset** property value, set to 0 by default, that allows user defined alignment of logically synchronous AADL-threads. Arrival of event (-data) will not result in a dispatch. Events and event data are accessible (PLG ????) to a periodic AADL-thread. (PLG: clarify event (-data) queuing).

   (a) Enabled is **T = Period+Dispatch_Offset**

   (b) Wait_For_Dispatch is **T** ≤ **Period+Dispatch_Offset**.

   (c) The dispatch occurs at (PLG: immediately after) **T = Period+Dispatch_Offset**.

2. **aperiodic**(31): a dispatch request is issued to THREAD when a triggering event occurs; there is no constraint on the inter-arrival time of triggering events. A triggering event occurs when:

   (a) an event (-data) arrives at an event (-data) port of THREAD with empty queue

   (b) a subprogram call arrives at a *provides* access feature of THREAD

   (c) THREAD raises its *complete* event and an event is already queued in some of its event (-data) port features

3. **sporadic**(32): dispatch requests are the same as in the aperiodic Dispatch_Protocol, but the time interval between successive dispatch requests will never be less than the associated **Period** property value.

4. **timed**(33): dispatch requests are the same as in the aperiodic Dispatch_Protocol, but the time interval between two successive dispatch requests will never be more than the associated Period property value. Thus an event time-out is raised to Dispatch if **T = Period**. The **Dispatch_Offset** property does not apply. (PLG contradiction with definition of Period p. 268, where Period is not allowed here).

5. **hybrid**(34): dispatch requests are those of the aperiodic Dispatch_Protocol, completed by those of the periodic Dispatch_Protocol, for which a periodic clock T_p is required; thus a supplementary event is raised to dispatch when T_p = $Period$. The **Dispatch_Offset** property does not apply. (PLG contradiction with definition of Period p. 268, where Period is not allowed here).

6. **background**(36): the AADL-thread is dispatched immediately upon completion of its initialization entrypoint execution. A background AADL-thread is Mode insensitive.

(5.4(9)) For periodic threads arrival of events or event data will not result in a dispatch. Events and event data are accessible to a periodic thread...

(5.4.6??(86)) A method of implementing a system must support the periodic dispatch protocol. A method of implementation may support only a subset of the other standard dispatch protocols. A method of implementation may support additional dispatch protocols not defined in this standard.

6.4.4 Thread states and state transition

(5.4.1 (15)) When a mode switch is initiated, a thread that is part of the old mode and not part of the new mode exits the mode by transitioning to the suspended awaiting mode (SAM) state after performing thread deactivation during the mode change in progress system state (see Figure 20). If the thread is periodic and its Synchronized_Component property is true, then its period is taken into consideration to determine the actual mode switch time (???? see Sections 12 and 13.3 for detailed timing semantics of a mode switch). If an aperiodic or a sporadic thread is executing a dispatch when the mode switch is initiated, its execution is handled according to the Active_Thread_Handling_Protocol property.

A thread that is not part of the old mode and part of the new mode enters the mode by transitioning to the suspended awaiting dispatch (SAD) state after performing thread activation.

(5.4.3 (39) A thread initially enters the ready state. A scheduler selects one thread from the set of threads in the ready state to run on one processor according

to a specified scheduling protocol. It ensures that only one thread is in the running state on a particular processor.

States and "normal" transitions (assert ignored) let **SRC** in {**AADL-process, vprocessor, processor, system**}

- **TH**: AADL-thread halted(14), (AADL-thread not in a current Mode)

  $(?\, loaded(AADL-process)\ !\, dispatch\ initialization) : T := 0,\ TH \rightarrow [PTI]$

  $(?\ AADL-threadexit(Mode)\ \vee\ ?\ AADL-threadenter(Mode)) : TH \rightarrow TH$

- **[PTI]**: performing AADL-thread initialization, (AADL-thread not in a current Mode)

  let $initialization\ completed\ =\ (started(system) \wedge ?\, complete\ initialization)$

  $\delta T = 1,\ \delta C \in \{0,1\}?$

  (THREAD is not part of the initial mode $\wedge$ initialization completed): $[PTI] \rightarrow SAM$

  (THREAD is part of the initial mode $\wedge$ initialization completed): $[PTI] \rightarrow SAD$

  PLG: Mode change during initialization?

- **SAM**: suspended awaiting mode(15) (AADL-thread not in a current Mode)

  $\delta T = \delta C = 0?$

  $(?\, AADL\text{–}thread\ enter(Mode) \wedge !\, dispatch\ activation) : T := 0,\ SAM \rightarrow [PTA]$

  $(stop(SRC)) : T := 0,\ SAM \rightarrow [PTF]$

- $[PTF]$: performing AADL-thread finalize (AADL-thread not in a current Mode)

  $\delta T = 1,\ \delta C \in \{0,1\}?$

  $(stopped(AADL-process)) : [PTF] \rightarrow TH$

  PLG: Mode change during finalize?

- **[PTD]**: performing AADL-thread deactivation, (AADL-thread not in a current Mode)

  $\delta T = 1,\ \delta C \in \{0,1\}?$

$(?\ complete\ deactivation) :\ [PTD] \rightarrow SAM$

PLG: Mode change during deactivation?

- **[PTA]**: performing AADL-thread activation, (AADL-thread in current cMode)

  $\delta T = 1,\ \delta C \in \{0,1\}?$

  $(?\ complete\ activation) :\ [PTA] \rightarrow SAD$

  $(stop(SRC)) :\ T := 0,\ [PTA] \rightarrow [PTF]$

  PLG: exit(cMode) during activation?

- **SAD**: suspended awaiting dispatch(16) (AADL-thread in current cMode)

  $\delta T = \ \delta C = \ 0?$

  $(Enabled(T) \wedge !\ dispatch\ computation) :\ T := 0,\ SAD \rightarrow [PTC]$

  $(stop(SRC)) :\ T := 0,\ SAD \rightarrow [PTF]$

  $(?AADL-thread\ exit(cMode)) :\ T := 0,\ SAD \rightarrow [PTD]$

- **[PTC]**: performing AADL-thread computation, (AADL-thread in possibly suspended current cMode)

  $\delta T = 1?\ \delta C \in \{0,1\}(seeinnerstate)$

  $(?\ complete\ activation) :\ [PTA] \rightarrow SAD$

  PLG: AADL-thread exit(cMode) during computation in inner transitions

  PLG: stop(SRC) during computation ?

- PLG: compute state, used in (5.4.1(16)) is not defined. In (16) we have If a dispatch request is received for an AADL-thread while the AADL-thread is in the compute state, this dispatch request is handled according to the specified Overflow_Handling_Protocol for the event or event data port of the AADL-thread. (???). Probably means super state performing AADL-thread computation

performing thread computation: inner states and transitions (AADL-thread in possibly suspended current cMode)

- **PTC.ready**:
  - $\delta C = 0$
  - $?\ resume :\rightarrow PTC.Running$

- **PTC.Running**:
  - δC = 1
  - ? $preempt$: → $PTC.Ready$
  - ! $complete$: → $null\ state$
  - $(AADL\text{–}thread\ is\ background \wedge ?\ exit(cMode))$: → $PTC.Awaiting\ resume$
  - ! $call\ server\ subprogram$: → $PTC.Awaiting\ return$
  - ! $Get\_Resource$: → $PTC.Awaiting\ resource$
- AADL-thread is background **PTC.Awaiting resume**
  - δC = 0
  - ? $enter(cMode)$: → $PTC.ready$
- **PTC.Awaiting return**
  - δC = 0
  - ? $return\ server\ subprogram$: → $PTC.ready$
- **PTC.Awaiting resource**
  - δC = 0
  - ? $Release\_Resource$: → $PTC.Awaiting\ resource$

Specific states and "abnormal" transitions (see 5.4)

6.5 Thread in Polychrony

One can propose a uniform view of AADL-threads.

6.5.1 Expressiveness

The preemption mechanism cannot be fully described in Signal, due to possible invisible side effects. All other AADL mechanism can be described in full Signal (ie non endochronous Signal-processes). One suppose that an AADL-thread is split into atomic actions that contains no more than one external interaction (value output/input, subprogram call,...). If the source language is Signal, this splitting can be automatic (gray box construction).

6.5.2 Uniform view

A background AADL-thread is considered as an aperiodic AADL-process with one single dispatch and lowest priority.

An AADL-thread T is translated into a Signal-process P that has the same input/output as T (ports). This Signal-process P is embedded in a process that provides to P, accurate synchronizations and communications. One can find below the coarse principles to do it.

1. P is (automatically) structured into atomic components following the gray box principles (extended to IO equivalence, refining the I equivalence)

2. P is then nested in a container C_P that insures the correct scheduling and data transmission using FIFOs for event(data) ports thanks to a synchronization Signal-process SP and a communication manager CM. SP and CM communicate (for instance to determine the complete event).

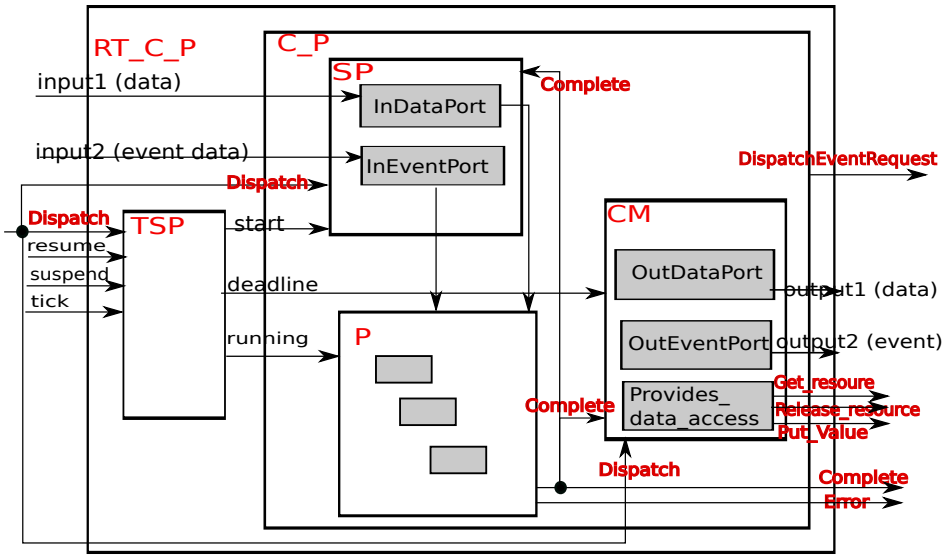
   - The synchronization Signal-process SP owns the Signal-signals of P completed by the (data events or) events found in the (fig.5,6): ? event dispatch, event complete, event data mode, Get_resource, Release_resource. It is built thanks to properties of T (including the dispatch_property) and the Polychrony standard gray box scheduler of P. SP describes the (logical) clock behaviors resulting from T and inner features properties. SP has a companion Signal-process TSP that interfaces logical events and real-time. SP is composed with a companion Signal-process that manages timing constraints (intervals). It builds the dispatch event according to the dispatch protocol from DispatchEventRequest and PeriodEvent.
   - The communication manager CM interacts with (contains ?) port FIFOs to schedule event (data) actual delivering taking into account T and port properties (such as priorities,...). It has its own inner clock. For aperiodic, sporadic, timed, hybrid AADL-threads, it generates the Boolean Signal-signal DispatchEventRequest at each occurrence of complete event. DispatchEventRequest is false if all FIFO are empty, true otherwise. It generates the events required by ports synchronizations.

3. 1.The container C_P is used as a component in a real time container RT_C_P. C_P is composed with the companion Signal-process TSP that interfaces logical events and real-time:

   TSP has the time unit as input (or the time value in the current hyperperiod) and computes C and T (or TSP receives T and computes C,...). TSP generates

timed events resulting from time properties such as DeadlineEvent,... For a periodic, sporadic, timed, hybrid AADL-process TSP generates an event PeriodEvent.

In Figure 9, a thread is interpreted as a real-time container *RT_C_P*. It is composed of a timing environment *TSP* and a container *C_P*.



P: have the same input/output as thread T
SP: a synchronization process
CM: a communication manager,
interact with port FIFO to schedule event (data) actual delivering
C_P: a container, insure correct scheduling and data transmission
TSP: a timing environment, generate timed events resulting from time properties
RT_C_P: a real-time container

Figure 9: Thread

- The timing environment *TSP* handles the time properties, and generates timed events, such as *start* and *deadline*.
- The container *C_P* insures the correct data transmission (*SP* and *CM*) and execution (*P*).
- *SP* contains all the in ports and requires (data, bus or subprogram) access feature.

- *CM* contains all the out ports and provides (data, bus or subprogram) access feature. For aperiodic, sporadic threads, it generates a boolean signal *DispatchEventRequest* at each occurance of complete event.
- *P* is a synchronous computation process. When it finishes, a *Complete* event is sent out. An *Error* event is generated when an unrecoverable error occurs.

Problem: Thread mode transition.

6.5.3 Remaining questions

1. Errors, event abort
2. Provides/require access
3. The details of the above description and the combination with other features translation may raise new problems.
4. Loops in the scheduler due to multiple input/output during the AADL logical time (i.e. dispatch-complete interval). Sequence type in Signal ?.
5. Define the precisely the morphism that transforms the Signal step gray box scheduler into an oversampled scheduler (ie input are cells, a single black box runs in an oversampled instant, following the initial static graph)

7 Thread group

(1)A thread group represents an organizational component to logically group threads contained in processes. The type of a thread group component specifies the features and required subcomponent access through which threads contained in a thread group interact with components outside the thread group. Thread group implementations represent the contained threads and their connectivity. Thread groups can have multiple modes, each representing a possibly different configuration of subcomponents, their connections, and mode-specific property associations. Thread groups can be hierarchically nested.

PLG: An AADL-thread group has properties such as period, deadline,...priority,... What are the relations of these properties/constraints with the same properties in inner features ?

A thread group represents an organizational component to logically group thread contained in processes. A thread group does not represent a virtual address space nor does it represent a unit of execution. It must be directly or indirectly contained within a process.

7.1 Abstract syntax

$$Thread\_group \ ::= \ Thread\_group\_type + Thread\_group\_implementation$$

Thread_group_type

$$\begin{aligned} Thread\_group\_type \ ::= \ & thread\_group\_ID \times \mathbf{opt}(\mathbf{list}(Thread\_group\_feature)) \\ & \times \mathbf{opt}(\mathbf{list}(Flow\_spec)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Thread\_group\_property)) \\ Thread\_group\_feature \ ::= \ & Port + Feature\_group + Data\_access \\ & + Subprogram\_access + Subprogram\_group\_access \end{aligned}$$

Thread_group_implementation

$$\begin{aligned} Thread\_group\_implementation \ ::= \ & thread\_group\_ID \\ & \times \mathbf{opt}(\mathbf{list}(Thread\_group\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Connection)) \\ & \times \mathbf{opt}(\mathbf{list}(Flow\_implementation)) \times \mathbf{opt}(\mathbf{list}(End\_to\_end\_flow)) \\ & \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Thread\_group\_property)) \\ Thread\_group\_subcomponent \ ::= \ & subcomponentID \\ & \times Thread\_group\_subcomponent\_reference \\ & \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes) \\ Thread\_group\_subcomponent\_reference \ ::= \ & dataID + subprogramID \\ & + subprogram\_group\_ID + threadID + thread\_group\_ID \end{aligned}$$

7.2 Standard properties

The thread group properties could refer to the thread properties.

8 Process

(1)A process represents a virtual address space. The Runtime_Protection process property indicates whether this virtual address space is runtime protected, i.e., it represents a space partition unit whose boundaries are enforced at runtime. The virtual address space contains the program formed by the source text associated with the process and its subcomponents. A complete implementation of a process must contain at least one thread or thread group subcomponent.

(13)This standard permits dynamic virtual memory management or dynamic library linking after process loading has completed and thread execution has started.

However, a method for implementing a system must assure that all deadline properties will be satisfied to the required level of assurance for each thread.

8.1 Structure

Type		*Implementation*	
Features	port	**Subcomponents**	abstract
	Feature group		data
	Provides, *Requires* data access		subprogram(- group)
	Provides, *Requires* subprog. (- group) access		thread(- group)
		Subprog. calls	*NO*
		Connections	*yes*
Flow spec, Mode	*yes*	**Flows, Modes**	*yes*

Figure 10: Process structure

8.2 Abstract syntax

A process represents a virtual address space. Threads of a process must be explicitly declared.

$$Process \ ::= \ Process\_type + Process\_implementation$$

Process_type

$$Process\_type \ ::= \ processID \times \mathbf{opt}(\mathbf{list}(Process\_feature)) \times \mathbf{opt}(\mathbf{list}(Flow\_spec)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Process\_property))$$

$$Process\_feature \ ::= \ Port + Feature\_group + Data\_access + Subprogram\_access + Subprogram\_group\_access$$

Process_implementation

$Process\_implementation$ ::= $processID \times \mathbf{opt}(\mathbf{list}(Process\_subcomponent))8$
$\times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Flow\_implementation)) \times \mathbf{opt}(\mathbf{list}(End\_to\_end\_flow))8$
$\times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Process\_property))$
$Process\_subcomponent$ ::= $subcomponentID \times Process\_subcomponent\_reference8$
$\times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$
$Process\_subcomponent\_reference$ ::= $dataID + subprogramID$
$+ subprogram\_group\_ID + threadID + thread\_group\_ID$

8.3 Standard properties

- Deployment properties:

Allowed_{Processor, Memory, Connection}_Binding_Class
Allowed_{Processor, Memory, Connection}_Binding
Actual_{Processor, Memory, Connection}_Binding
Allowed_Subprogram_Call_Binding
Not_Collocated
Collocated

- Predeclared thread properties.

Priority
Time_Slot
Resumption_Policy
Active_Thread_Handling_Protocol
Active_Thread_Queue_Handling_Protocol
Runtime_Protection
Synchronized_Component

- Timing properties

Deadline
Load_Deadline
Period
Startup_Deadline
Startup_Execution_time
Reference_Processor

- Predeclared memory properties.

Source_Code_Size
Source_Data_Size

- Predeclared programming properties

Source_Language
Source_Text

PLG: An AADL-process has properties such as period, deadline,...priority,... What are the relations of these properties/constraints with the same properties in inner features ?

8.4 Process and Polychrony

Same question as for AADL-thread group concerning inheritance.

A standard Signal-process ? Following AADL definition of an AADL-process (a –virtual– address space), a notion of Object-process in Signal (or any other name) can correspond to shared variable scopes.

Check period, time-out, ... properties. They might impact connection delay by accumulation.

9 Execution platform components

9.1 Processor

A processor is an abstraction of hardware and software that is responsible for scheduling and executing threads and virtual processors that are bound to it.

9.1.1 Abstract syntax

$$Processor \ ::= \ Processor\_type + Processor\_implementation$$

Processor_type

$$Processor\_type \ ::= \ processorID \times \mathbf{opt}(\mathbf{list}(Processor\_feature)) \times \mathbf{opt}(\mathbf{list}(Flow\_type)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Processor\_property))$$

$$Processor\_feature \ ::= \ Provides\_subprogram\_access + Provides\_subprogram\_group\_access + Port + Feature\_group + Bus\_access + Feature\_group$$

Processor_implementation

$$Processor\_implementation ::= processorID \times \mathbf{opt}(\mathbf{list}(Processor\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Flow\_implementation)) \times \mathbf{opt}(\mathbf{list}(End\_to\_end\_flow)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Processor\_property))$$

$$Processor\_subcomponent ::= subcomponentID \times Processor\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$$

$$Processor\_subcomponent\_reference ::= memoryID + busID + virtual\_processor\_ID + virtual\_bus\_ID$$

9.1.2 Standard properties

1. Deployment properties

Allowed_Memory_Binding_Class
Allowed_Memory_Binding
Actual_Memory_Binding
Provided_Virtual_Bus_Class
Provided_Connection_Quality_Of_Service
Allowed_Period
Scheduling_Protocol
Preemptive_Scheduler
Thread_Limit
Priority_Map
Priority_Mapping
Priority_Range

2. Thread properties

Resumption_Policy
Deactivation_Policy

3. Timing properties.

Startup_Deadline
Startup_Execution_Time
Clock_Jitter
Clock_Period
Clock_Period_Range
Process_Swap_Execution_Time
Scaling_Factor
Scheduler_Quantum
Thread_Swap_Execution_Time
Frame_Period
Slot_Time

4. Memory properties

Assign_Time
Source_Code_Size
Source_Data_Size
Source_Stack_Size

5. Programming properties.

Source_Language
Source_Text
Supported_Source_Language
Hardware_Description_Source_Text
Hardware_Source_Language

6. Modeling properties.

Implemented_As

9.2 Virtual processor

A virtual processor represents a logical resource that is capable of scheduling and executing threads and other virtual processors bound to them.

9.2.1 Abstract syntax

$Virtual\_processor$ ::= $Virtual\_processor\_type + Virtual\_processor\_implementation$

Virtual_processor_type

$$Virtual\_processor\_type ::= virtual\_System\_subcomponent\_referenceprocessor\_ID \times \mathbf{opt}(\mathbf{list}(Virtual\_processor\_feature)) \times \mathbf{opt}(\mathbf{list}(Flow\_type)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Virtual\_processor\_property))$$

$$Virtual\_processor\_feature ::= Provides\_subprogram\_access + Provides\_subprogram\_group\_access + Port + Feature\_group$$

Virtual_processor_implementation

$$Virtual\_processor\_implementation ::= virtual\_processor\_ID \times \mathbf{opt}(\mathbf{list}(Virtual\_processor\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Flow\_implementation)) \times \mathbf{opt}(\mathbf{list}(End\_to\_end\_flow)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Virtual\_processor\_property))$$

$$Virtual\_processor\_subcomponent ::= subcomponentID \times Virtual\_processor\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$$

$$Virtual\_processor\_subcomponent\_reference ::= virtual\_processor\_ID + virtual\_bus\_ID$$

9.2.2 Standard properties

1. Deployment properties

Allowed_Processor_Binding_Class
Allowed_Processor_Binding
Actual_Processor_Binding
Provided_Virtual_Bus_Class
Provided_Connection_Quality_Of_Service
Scheduling_Protocol

2. Thread properties

Time_Slot
Deactivation_Policy

3. Timing properties.

Execution_Time
Period
Startup_Deadline
Startup_Execution_Time
Frame_Period
Slot_Time

4. Programming properties.

Supported_Source_Language

5. Modeling properties.

Implemented_As

9.3 Memory

A memory represents an execution platform component that stores code and data binaries.

9.3.1 Abstract syntax

$$Memory ::= Memory\_type + Memory\_implementation$$

Memory_type

$$Memory\_type ::= memoryID \times \mathbf{opt}(\mathbf{list}(Memory\_feature)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Memory\_property))$$
$$Memory\_feature ::= Bus\_access + Feature\_group$$

Memory_implementation

$$Memory\_implementation ::= memoryID \times \mathbf{opt}(\mathbf{list}(Memory\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Memory\_property))$$
$$Memory\_subcomponent ::= subcomponentID \times Memory\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$$
$$Memory\_subcomponent\_reference ::= memoryID + busID$$

9.3.2 Standard properties

1. Deployment properties.

Provided_Virtual_Bus_Class Provided_Connection_Quality_Of_Service Memory_Protocol

2. Thread properties.

Resumption_Policy

3. Predeclared memory properties.

Byte_Count
Word_Size
Word_Space
Write_Time

4. Programming properties.

Source_Text
Hardware_Description_Source_Text
Hardware_Source_Language

5. Modeling properties.

Implemented_As

9.4 Bus

A bus represents an execution platform component that can exchange control and data between memories, processors and devices.

9.4.1 Abstract syntax

$$Bus \ ::= \ Bus\_type + Bus\_implementation$$

Bus_type

$$Bus\_type \ ::= \ busID \times \mathbf{opt}(\mathbf{list}(Bus\_feature)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Bus\_property))$$
$$Bus\_feature \ ::= \ Requires\_bus\_access + Feature\_group$$

Bus_implementation

$$Bus\_implementation \ ::= \ busID \times \mathbf{opt}(\mathbf{list}(Bus\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Bus\_property))$$
$$Bus\_subcomponent \ ::= \ subcomponentID \times Bus\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property\_assocaition)) \times \mathbf{opt}(In\_modes)$$
$$Bus\_subcomponent\_reference \ ::= \ virtual\_bus\_ID$$

9.4.2 Standard properties

1. Deployment properties.

Provided_Connection_Quality_Of_Service
Allowed_Connection_Type
Allowed_Physical_Access_Class
Allowed_Physical_Access

2. Thread properties.

Resumption_Policy

3. Communication properties.

Transmission_Type
Transmission_Time
Latency

4. Memory properties.

Access_Right
Allowed_Message_Size

5. Programming properties.

Source_Language
Source_Text
Hardware_Description_Source_Text
Hardware_Source_Language

6. Modeling properties

Implemented_As

9.5 Virtual bus

A virtual bus represents logical bus abstraction, such as a virtual channel or communication protocol.

9.5.1 Abstract syntax

$$Virtual\_bus \ ::= \ Virtual\_bus\_type + Virtual\_bus\_implementation$$

Virtual_bus_type

$$Virtual\_bus\_type ::= virtual\_bus\_ID \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Virtual\_bus\_property))$$

Virtual_bus_implementation

$$Virtual\_bus\_implementation ::= busID \times \mathbf{opt}(\mathbf{list}(Virtual\_bus\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Virtual\_bus\_property))$$

$$Virtual\_bus\_subcomponent ::= subcomponentID \times Virtual\_bus\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$$

$$Virtual\_bus\_subcomponent\_reference ::= virtual\_bus\_ID$$

9.5.2 Standard properties

1. Deployment properties.

Allowed_Connection_Binding_Class
Allowed_Connection_Binding
Actual_Connection_Binding
Provided_Virtual_Bus_Class
Required_Virtual_Bus_Class
Provided_Connection_Quality_Of_Service
Required_Connection_Quality_Of_Service

2. Communication properties.

Transmission_Type

3. Modeling properties

Implemented_As

9.6 Device

A device represents dedicated hardware within the system, entities in the external environment, or entities that interface with the external environment.

9.6.1 Abstract syntax

$$Device ::= Device\_type + Device\_implementation$$

Device_type

$$Device\_type ::= deviceID \times \mathbf{opt}(\mathbf{list}(Device\_feature)) \times \mathbf{opt}(\mathbf{list}(Flow\_spec)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Device\_property))$$

$$Device\_feature ::= Port + Feature\_group + Provides\_subprogram\_access + Provides\_subprogram\_group\_access + Bus\_access$$

Device_implementation

$$Device\_implementation ::= deviceID \times \mathbf{opt}(\mathbf{list}(Device\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Flow\_implementation)) \times \mathbf{opt}(\mathbf{list}(End\_to\_end\_flow)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(Device\_property))$$

$$Device\_subcomponent ::= subcomponentID \times Device\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes)$$

$$Device\_subcomponent\_reference ::= busID + virtual\_bus\_id$$

9.6.2 Standard properties

1. Deployment properties.

Allowed_{Processor, Memory}_Binding_Class
Allowed_{Processor, Memory}_Binding
Actual_{Processor, Memory}_Binding
Provided_Virtual_Bus_Class
Provided_Connection_Quality_Of_Service
Allowed_Connection_Type

2. Thread properties.

Dispatch_Trigger
Resumption_Policy

3. Timing properties.

Compute_Deadline
Compute_Execution_Time
Deadline
Period

4. Memory properties.

Source_Code_Size
Source_Data_Size
Source_Stack_Size

5. Programming properties.

{Activate, Compute, Deactivate, Finalize, Initialize, Recover}_Entrypoint
{Activate, Compute, Deactivate, Finalize, Initialize, Recover}_Entrypoint_Call_Sequence
{Activate, Compute}_Entrypoint_Source_Text
Source_Language
Source_Text
Hardware_Description_Source_Text
Hardware_Source_Language

6. Modeling properties.

Implemented_As

10 System

(1) A system represents an assembly of interacting application software, execution platform, and system components. Systems can have multiple modes, each representing a possibly different configuration of components and their connectivity contained in the system. Systems may require access to data and bus components declared outside the system and may provide access to data and bus components declared within. Systems may be hierarchically nested.

PLG: A system has properties such as period, deadline,... What are the relations of these properties/constraints with the same properties in inner features ?

A system represents an assembly of interacting application software, execution platform ans system components.

10.1 Abstract syntax

$$System \ ::= \ System\_type + System\_implementation \tag{1}$$

System_type

$$System\_type \ ::= \ systemID \times \mathbf{opt}(\mathbf{list}(System\_feature)) \times \mathbf{opt}(\mathbf{list}(Flow\_spec)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(System\_property)) \tag{2}$$

$$System\_feature \ ::= \ Port + Feature\_group + Subprogram\_access + Subprogram\_group\_access + Bus\_access + Data\_access \tag{3}$$

System_implementation

$$System\_implementation ::= systemID \times \mathbf{opt}(\mathbf{list}(System\_subcomponent)) \times \mathbf{opt}(\mathbf{list}(Connection)) \times \mathbf{opt}(\mathbf{list}(Flow\_implementation)) \times \mathbf{opt}(\mathbf{list}(End\_to\_end\_flow)) \times \mathbf{opt}(\mathbf{list}(Modes)) \times \mathbf{opt}(\mathbf{list}(System\_property)) \quad (4)$$

$$System\_subcomponent ::= subcomponentID \times System\_subcomponent\_reference \times \mathbf{opt}(\mathbf{list}(Property)) \times \mathbf{opt}(In\_modes) \quad (5)$$

$$System\_subcomponent\_reference ::= dataID + subprogramID + subprogram\_group\_ID + processID + processorID + virtual\_processor\_ID + memoryID + busID + virtual\_bus\_ID + deviceID + systemID \quad (6)$$

10.2 Standard properties

1. Deployment properties:

Allowed_{Processor, Memory, Connection}_Binding_Class
Allowed_{Processor, Memory, Connection}_Binding
Actual_{Processor, Memory, Connection}_Binding
Allowed_Subprogram_Call_Binding
Provided_Virtual_Bus_Class
Provided_Connection_Quality_Of_Service
Not_Collocated
Collocated
Allowed_Period

2. Predeclared thread properties.

Priority
Time_Slot
Resumption_Policy
Active_Thread_Handling_Protocol
Active_Thread_Queue_Handling_Protocol
Runtime_Protection
Synchronized_Component

3. Timing properties

Deadline
Load_Deadline
Period
Startup_Deadline
Startup_Execution_time
Clock_Jitter
Clock_Period
Clock_Period_Range
Reference_Processor
Scaling_Factor
Thread_Swap_Execution_Time

4. Predeclared memory properties.

Source_Code_Size
Source_Data_Size

5. Predeclared programming properties

Source_Language
Source_Text
Supported_Source_Language
Hardware_Description_Source_Text
Hardware_Source_Language

6. Predeclared modeling properties.

Implemented_As

10.3 Component binding

(13(3)) A system instance is completely instantiated and bound if all threads are ultimately bound to a processor, all source text making up process address spaces are bound to memory, connections are bound to buses if their ultimate source and destinations are bound to different processors, and subprogram calls are bound to remote subprograms as necessary.

(13(C1))Every mode-specific configuration of a system instance must have a binding of every process component to a (set of) memory component(s), and a binding of every thread component to a (set of) processor(s).

(C13(2) In the case of dynamic process loading, the actual binding may change at runtime. In the case of tightly coupled multi-processor configurations, such as dual core processors, the actual thread binding may change between members of an actual binding set of processors as these processors service a common set of thread ready queues.

(C13(4) A software component may be bound to multiple memory components.

(C13(5) A thread must be bound to a one or more processors. If it is bound to multiple processors, the processors share a ready queue, i.e., the thread execute on one processor at a time.

(13(C6) Multiple threads can be bound to a single processor.

10.4 System operation mode

(13) The set of all mode transitions specified for all components of a system instance form a set of concurrent mode transitions, called system operation modes (SOM). The set of possible SOMs is the cross product of the sets of modes for each component. That is, a SOM is a set of component modes, one mode for each component of the system. The initial SOM is the set of initial modes for each component. (PLG: this suggest a Global mode)

(14) The discrete variable Mode denotes a SOM. That is, the variable Mode denotes a possible discrete state that is defined by the mode hybrid semantic diagrams. Note that the value of Mode will in general change at various instants of time during system operation, although not in a continuous time-varying way.

(15) The SOM transition is requested whenever a mode transition in any component in the system instance is requested by the arrival of an event. A single event can trigger a mode switch request in one or more components. In a synchronized system, this event occurs logically simultaneously for all components, i.e., the resulting component mode switch requests are treated as a single SOM transition request.

(16) A mode transition of a thread internal mode, i.e., a mode declared in the thread or one of its subprograms, that is triggered by the component itself or is triggered by an event coming in through an event port of the thread, takes place at the next thread dispatch; if the event triggers both a mode transition and a dispatch, then the dispatch is considered to be the next dispatch.

(18) If several events occur logically simultaneously and are semantically connected to transitions in different components that lead out of their current mode or to different transitions out of the same mode in one component, then events are considered to have an implementation-dependent order that determines the mode transition for the mode switch resulting in the other events being ignored. (PLG: does this mean no queuing of mode transition triggers ?)

(19) After a SOM transition request has occurred, the actual SOM transition occurs in zero time, if no periodic threads are part of the old mode, otherwise, it occurs at the hyperperiod boundary of the old SOM...During that time, the system

continues to operate in the old SOM and additional events that would result in a SOM transition from the current SOM are ignored.

(20) ... The hyperperiod is determined by the periods of those periodic threads whose Synchronzied_Component property is true and that are active in a given SOM. If this set of threads is empty, the mode transition is initiated immediately.

(21) At the time of actual SOM transition, the transition is performed to the new SOM that contains the destination modes of the requested component mode switch(es). The hyperperiod for the mode transition is determined by the set of thread to be active in the new SOM.

(22) A runtime transition between SOMs requires a non-zero interval of time, during which the system is said to be in transition between two system modes of operation. While a system is in transition, excluding the instants of time at the start and end of a transition, all arriving events that appear in transition edge declarations are ignored and will not cause any mode change.

(23) At the instant of time the mode-transition-in-progress state is entered, connections that are part of the old SOM and not part of the new SOM are disabled. For data connections, this means that the data value is not transferred into the in data port variable of the newly disabled thread.

(24) At the instant of time the mode-transition-in-progress state is entered, data is transferred logically simultaneously for all connections that are declared to be part of any of the component mode transitions making up the SOM transition. For data connections, this means that the data is transferred from the out data port such that its value becomes available at the first dispatch of the receiving thread.

(25) At the instant of time the mode-transition-in-progress state is entered, connections that are not part of the old SOM and part of the new SOM are enabled. For data connections, this means that the data value of a transition connection is transferred into the in data port variable of the newly enabled thread. If the in data port of the destination thread is not the destination of a transition connection, the data value of the out data port of the source thread is transferred into the in data port variable of the newly enabled thread. If the source thread is also activated as part of the mode transition, its out data port value is transferred after the thread completes its activate entrypoint execution.

(26) When the mode-transition-in-progress state is entered, thread exit(Mode) is triggered for all threads that are part of the old mode and not part of the new mode. This results in the execution of deactivation entrypoints for those threads (see Figure 5) as described in Section 12.

(27) In addition, at the time the mode-transition-in-progress state is entered, thread enter(Mode) is triggered for threads that are part of the new mode and not part of the old mode. This permits those threads to execute their activation entrypoints (see Figure 5). In addition, for periodic threads this is immediately followed

by their first compute entrypoint dispatch as described in Section 12.

(29) While the system is in the mode-transition-in-progress state, threads that are part of the old and new SOM continue to operate normally. SOM transition requests as resulting from raise events are ignored while the system instance is in the mode-transition-in-progress state.

(30) The system instance remains in the mode-transition-in-progress state until the next hyperperiod. This hyperperiod is determined by new SOM according to the rules stated earlier. At that time, the system instance enters current_system_operation_mode state and starts responding to new requests for SOM transition. (TG: what does it mean if there is no periodic thread in the new mode and how is it compatible with the protocols to handle threads that are in the performing computation state at the time instant of actual mode switch? cf. 12 (22))

PLG: what about queues and other pending actions

10.5 AADL and physical time

10.5.1 Perfect/unperfect real time(5.4.(5,6))

(13.3(11), p. 234) In a synchronized system, periodic threads are dispatched simultaneously with respect to a global clock. The hyperperiod of a set of periodic threads (PLG: sharing the same time reference) is defined to be the least common multiple of the periods of those threads.

(5.4.5 (61))...In the concurrent hybrid automata model for the complete system, ST is a single real-valued variable shared by all threads that is never reset and whose rate is 1 in all states. ST is called the reference timeline.

(5.4.5 (62)) Two periodic threads are said to be synchronized if, whenever they are both active in the current system mode of operation, they are logically dispatched simultaneously at (...) their hyperperiod. Two threads are logically dispatched simultaneously if the order in which all exchanges of control and data at that dispatch event are identical to the order that would occur if those dispatches were exactly dispatched simultaneously in true and perfect real time

(PLG ??? notion not defined in the standard). If all periodic threads contained in an application system are synchronized, then that application system is said to be synchronized.

(5.4.5 (64)) Within a synchronization domain, perfect synchronization may not occur in a physical system. (...) it is the responsibility of each physical implementation to take these imperfections into account when providing the synchronization domain for programmers (e.g. make sure your message transmission schedule includes enough margin for the message to get there by the time it is needed, taking into account these various effects in your particular implementation).

(5.4.6 (68)) Message-passing semantics of communication and thread execution is represented by aperiodic threads whose dispatch is triggered by arrival of messages and message may be queued in the event data port. This communication paradigm is insensitive to time, thus, not affected by multiple synchronization domains.

(5.4.6 (69)) Sampled data-stream semantics of communication and thread execution is represented by periodic threads and data ports. In this case the sampling of the input is sensitive to the reference time. AADL distinguishes between immediate and delayed connections for deterministic sampling, and sampling connections for non-deterministic sampling. Similarly, a periodic thread may non-deterministically sample event ports and event data ports, e.g., a health monitor sampling an alarm queue. Deterministic communication minimizes latency jitter, while non-deterministic communication can result in latency jitter in units of the sampling rate, the latter often leading to instability of latency sensitive applications such as control systems.

(5.4.6 (70)) In general, communication timing of immediate and delayed connections cannot be guaranteed when the connection crosses synchronization domains. In other words, those connections become sampling connections.

10.5.2 Asynchronous system (5.4.6)

In this section (???), one found:

(5.4.6(75)) The Await_Dispatch runtime service takes a mask and a trigger condition function as parameter. The mask specifies which ports are being considered in triggering the next dispatch of a thread. The trigger condition function, if present, is evaluated on the ports identified in the mask to determine when a dispatch should occur.

```
subprogram Await_Dispatch
  features
    PortMask: in parameter; -- List of ports that can trigger a dispatch
    ConditionFunction: subprogram;
end Await_Dispatch;
```

10.6 System and Polychrony

Same question as for AADL-thread group and process concerning inheritance.

Because in Signal, there is no notion of hardware component, a system is a composition of Signal-processes.

Check period, time-out, ... properties. They might impact connection delay by accumulation.

11 Features and shared access

A feature is a part of a component type definition that specifies how that component interfaces with other component.

A feature could be a port, subcomponent access, parameter or feature group.

(3) Port features represent a communication interface for the exchange of data and events between components.

(4) Subprogram access features represent access to a subprogram to be called from other components, and the need for a component to call a subprogram instance locally or to call a subprogram remotely.

(5) Subprogram group access features represent sharing and required access to a subprogram library.

(6) Parameter features represent data values that can be passed into and out of subprograms.

(7) Data subcomponent access represents communication via shared access to data components.

(8) Bus subcomponent access represents physical connectivity of processors, memory, devices, and buses through buses.

(3) Feature groups represent groups of component features. Feature groups can contain feature groups. Feature groups can be used anywhere features can be used.

Abstract syntax of Feature

$$Feature ::= Port + Parameter + Subcomponent\_access + Feature\_group$$

$$Subcomponent\_access ::= Subprogram\_access + Subprogram\_group\_access + Data\_access + Bus\_access$$

11.1 Port

(1) Ports are logical connection points between components that can be used for the transfer of control and data between threads or between a thread and a processor or device. Ports are directional, i.e., an output port is connected to an input port. Ports can pass data, events, or both. Data transferred through ports is typed..... Incoming events may trigger thread dispatch or mode transitions. Properties specify the input and output timing characteristics of ports. Actual event and data transfer may be initiated by the runtime system of the execution platform or by Send_Output runtime service calls in the application source text.

AADL distinguishes between three port categories: *data port*, *event port* and *event data port*.

From the perspective of the application source text, data ports are accessible in the source text as data variables, event ports represent event queues whose size is accessible, event data ports represent message queues whose content can be retrieved.

An example:

```
thread threadA
  features
    portA: in data port {Timing => immediate };
    portB: in event port;
    portC: out event data port dataA
           {Output_Time => (Completion, 0.0ns .. 0.0ns)};
end threadA;
```

11.1.1 Abstract syntax of *Port*

$$Port ::= Event\_port + Data\_port + Event\_data\_port \quad (7)$$

$$Basic\_port ::= portID \times Port\_direction \times \mathbf{opt}(\mathbf{list}(Port\_property)) \quad (8)$$

$$Event\_port ::= Basic\_port \quad (9)$$

$$Data\_OR\_Eventdata\_port ::= Port\_triggering \times \mathbf{opt}(Data\_reference) \times Basic\_port \quad (10)$$

$$Data\_port ::= Data\_OR\_Eventdata\_port([Port\_triggering = no]) \quad (11)$$

$$Event\_data\_port ::= Data\_OR\_Eventdata\_port([Port\_triggering = on]) \quad (12)$$

$$Port\_direction ::= \{in, out, \{in\ out\}\} \quad (13)$$

$$Port\_triggering ::= \{on, no\} \quad (14)$$

$$Data\_reference ::= dataID \quad (15)$$

Note:

1. A *Port* belongs to three categories: *Data_port*, *Event_port* and *Event_data_port* (7).
2. A *Event_port* is specified by a *portID*, *Port_direction* and an optional list of *Port_property* (8).
3. A *Data_port* is a *Data_OR_Eventdata_port* whose *Port_triggering* is *no* (11).
4. A *Event_data_port* is a *Data_OR_Eventdata_port* whose *Port_triggering* is *on* (12).

5. *Port_direction* could be either *in*, *out*, or *in out* (13).

6. *portID* adheres to the naming rules specified for all identifiers.

7. A *Port_property* could be *Input_Time*, *Output_Time*, *Timing*, *Fan_out_policy* property association and many others. The following table gives a brief view of properties that are associated to ports. The property should be applied to the corresponding port categories.

Property	In port			Out port		
	Event	Data	Event data	Event	Data	Event data
Input_Time	X	X	X			
Output_Time				X	X	X
Source_Name	X	X	X	X	X	X
Source_Text	X	X	X	X	X	X
Type_Source_Name	X	X	X	X	X	X
Required_Connection	X	X	X	X	X	X
Allowed_Connection_Binding_Class	X	X	X	X	X	X
Device_Register_Address	X	X	X	X	X	X
Timing		X			X	
Input_Rate	X	X	X			
Output_Rate				X	X	X
Compute_Entrypoint	X		X			
Compute_Entrypoint_Call_Sequence	X		X			
Compute_Entrypoint_Source_Text	X		X			
Compute_Execution_Time	X		X			
Compute_Deadline	X		X			
Allowed_Memory_Binding_Class		X	X		X	X
Allowed_Memory_Binding		X	X		X	X
Actual_Memory_Binding		X	X		X	X
Overflow_Handling_Protocol	X		X			
Queue_Size	X		X	X		X
Queue_Processing_Protocol	X		X	X		X
Dequeued_Items	X		X			
Dequeue_Protocol	X		X			
Fan_Out_Policy				X	X	X
Urgency	X	X	X			
Transmission_Type		X			X	
Base_Address	X	X	X	X	X	X

11.1.2 Standard properties

This section gives an explanation of some of the standard properties listed in the above table.

1. Properties related to source text (...)

   Source_Name: **aadlstring applies to** (data, port, subprogram, parameter);
   Source_Text : **inherit list of aadlstring applies to**
   (data, port, subprogram, thread, thread group, process, system, memory, bus, device, processor, parameter, feature group, package);

2. Properties related to memory space (binding,...)

   Device_Register_Address: **aadlinteger applies to** (port, feature group)

3. Property related to port connections

   Required_Connection : **aadlboolean** ⇒ **true applies to** (feature)

4. Properties related to IO policy

   (a) **Input_Time** property can be used to explicitly specify an input time for ports. This property could have a list of values, which indicates that input is frozen multiple times for the execution during one dispatch. The default value is dispatch with zero offset.

   Input_Time: **list of** IO_Time_Spec ⇒ (Time ⇒ Dispatch;
   Offset ⇒ 0.0 ns .. 0.0 ns;) **applies** to (port);
   IO_Time_Spec : **type record** (Offset : TimeRange;
   Time : IO_Reference_Time;);

   Each possible value is a pair of *Time* (possible values: *Dispatch_Time* by default, *Start*, *Completion* and *NoIO*) and a time range *Offset* (0.0 ns .. 0.0 ns by default).

   The **IO_Time_Spec** property specifies the amount of execution time *Offset* relative to a *Time* at which input or output occurs. The value consists of a reference point and time range pair.

   Input_Time possible ReferencePoint:

   - **Dispatch_Time**: (the default value) input is frozen at dispatch time; the time reference is clock time.
     T = 0.
   - **Start**: input is frozen at a specified amount of execution time into the execution. The time is within the specified time range. The time range must have positive values.
     $Start\_Time_{low} \leq C \leq Start\_Time_{high}$.

- **Completion**: input is frozen at a specified amount of execution time relative to execution completion. The time is within the specified time range. A negative time range indicates execution time before completion.
  $(C_{complete}+Completion\_Time_{low}) \leq C \leq (C_{complete}+Completion\_Time_{high})$
  where $C_{complete}$ represents the value of c at completion time.
- NoIO: input is not frozen. In other words, the port is excluded from making new input available to the source text. This allows users to specify that a subset of ports to provide input. The property value can be mode specific, i.e., a port can be excluded in one mode and included in another mode.

The content of incoming ports are frozen at a specified time (Yue: Input_Time.) This means that the content of the port that is accessible to the recipient does not change during the execution of a dispatch (Yue: Input_Time, not really dispatch time) even though the sender may send new values.

Frozen: From the point of Input_Time on, any new arrived data (or event, or event data) is not accessible until the next Input_Time (Figure 11).

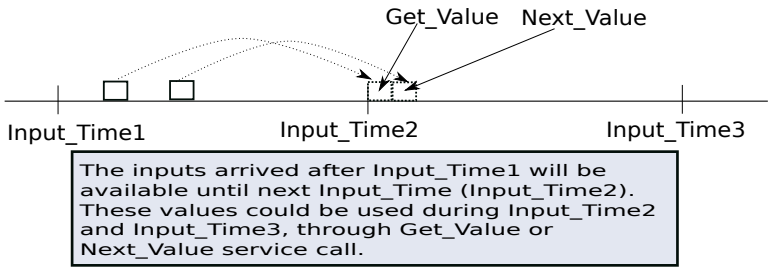


Figure 11: Input frozen

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.

In AS5506A v2, p136, "The Input_Time can be done for all ports by specifying the property value for the thread". But the Input_Time property is defined only applies to a port, but not to a thread (p262).

(b) **Output_Time** specifies the amount of execution time until completion at which output becomes available. This property indicates the time output is transmitted to connected components. The default value is completion with zero offset. Possible value is a pair of reference time

Time (*Start*, *Completion* by default, *Deadline* and *NoIO*) and a time range *Offset* (0.0 ns .. 0.0 ns by default).

Output_Time: **list of** IO_Time_Spec ⇒ (Time ⇒ Completion;
Offset ⇒ 0.0 ns .. 0.0 ns;) **applies** to (port);

Output_Time possible ReferencePoint:

- **Start_Time**: output is transmitted at a specified amount of execution time into the execution. The time is within the specified time range. The time range must have positive values.
  $Start\_Time_{low} \leq C \leq Start\_Time_{high}$.
- **Completion**: output is transmitted at a specified amount of execution time relative to execution completion. The time is within the specified time range. A negative time range indicates execution time before completion.
  $(C_{complete}+Completion\_Time_{low}) \leq C \leq (C_{complete}+Completion\_Time_{high})$
  where $C_{complete}$ represents the value of c at completion time.
  The default is completion time with a time range of zero, i.e., it occurs at $C = C_{complete}$
- **Deadline**: (the default value) ; output is transmitted at deadline time; the time reference is clock time.
  $T = Deadline$.
- **NoIO**: output is not transmitted . In other words, the port is excluded from making new output from the source text. This allows users to specify that a subset of ports to provide output. The property value can be mode specific, i.e., a port can be excluded in one mode and included in another mode.

The output will be transmitted immediately if it is called by a Send_output service call, otherwise it will be sent out at next Output_Time. (Figure 12)

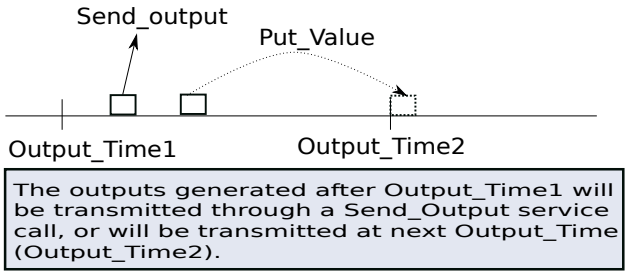


Figure 12: Output_Time

Input_Time and **Output_Time** can have a list of values. Two Signal events *InEvent* and *OutEvent* are used to represent the **Input_Time** and **Output_Time**.

(c) **Input_Rate** specifies the number of inputs per dispatch or per second of data, events, event data or subprogram calls. One input per thread dispatch by default.

```
Input_Rate: Rate_Spec ⇒ (Value_Range ⇒ 1.0 .. 1.0;
    Rate_Unit ⇒ PerDispatch; Rate_Distribution ⇒ Fixed) applies to (port);
Rate_Spec : type record (Value_Range : aadlreal;
    Rate_Unit: units (PerSecond, PerDisptach);
    Rate_Distribution : Supported_Distributions; );
```

(d) **Output_Rate** specifies the number of outputs per dispatch or per second of data, events, event data or initiations of subprogram calls. One output per thread dispatch and *fixed* distribution by default.

```
Output_Rate: Rate_Spec ⇒ (Value_Range ⇒ 1.0 .. 1.0;
    Rate_Unit ⇒ PerDispatch; Rate_Distribution ⇒ Fixed) applies to (port);
```

(e) **Timing** property specifies the connection type of a data port.

```
Timing : enumeration (sampled, immediate, delayed)
    ⇒ sampled applies to (port);
```

i. If Timing is declared as immediate, then Output_Time is Completion and Input_Time is Start. The defined Input_Time and Output_Time are ignored.

ii. If Timing is declared as delayed, then Output_Time is Deadline and Input_Time is Dispatch.

Timing	Input_Time	Output_Time
immediate	*Start*	*Completion*
delayed	*Dispatch*	*Deadline*

(f) **Fan_Out_Policy** property specifies how the output is distributed to multiple recipients of a port with multiple outgoing connections. Default value is *Broadcast*.

```
Fan_Out_Policy: enumeration (Broadcast, RoundRobin, Selective, OnDemand)
    applies to (port);
```

The Fan_Out_Policy property indicates whether the output is passed to all recipients (Broadcast), to the next recipient ready to be dispatched (OnDemand), or the output is distributed evenly to the recipients (RoundRobin). If the property is not specified the default is Broadcast. If the fan out policy is OnDemand, a queue may be associated with the port through the use of the appropriate queue properties.

PLG: the exact title for these queue properties is In port queue properties ; more over the complementary wording in 8.2.3 does not mention queues associated with output ????. Moreover an AADL-thread can probably wait for dispatch coming from several ports; if it is dispatched from one source, it should cancel the other demands,....???? it's a very costly protocol

A controler (Distributer) is needed to choose the recipients of an out port. (Figure 13.)

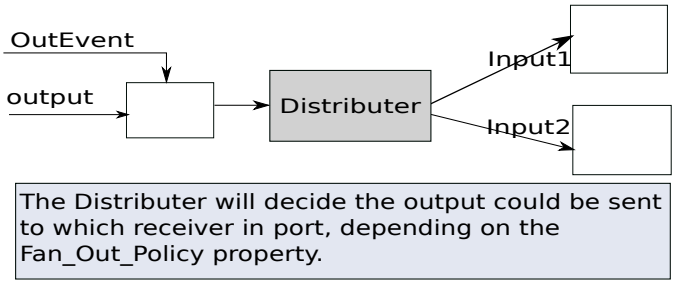


Figure 13: Fan_Out_Policy

11.1.3 In out (common) port behavior

Rate properties (29) The Input_Rate and Output_Rate properties specify the rate at which input and output is expected to occur at the port with the associated property. By default the input and output rate of ports is the rate at which the thread executes. The rate can be fixed (periodic) or according to a distribution. An input or output rate higher than the dispatch rate of a thread indicates that multiple inputs or multiple outputs are expected during a single dispatch. An input or output rate lower than the dispatch rate of a thread indicates that inputs or outputs are not expected at every dispatch. If an Input_Time or Output_Time property is specified, then the number of values must be consistent with the rate. An input or output rate lower than the period indicates that input is not expected at every dispatch and that output is not expected to be transmitted at every dispatch.

Those rate properties will not generate anything but comments in Signal. The consistence between a (Input/Output)_Time list statically defined and a (Input/Output)_Rate given by a distribution is not obvious to understand as such. Moreover, a rate lower than the rate given by the size of the (Input/Output)_Time results in a non-deterministic behavior. Thus, we should assume that (Input/Output)_Time list gives the (maximal) number of (Input/Output) values between 2 dispatches in the current Mode. And we consider Rates as information for verification tools.

Input ports (13) Data, events, and event data arriving through incoming ports is made available to the receiving thread, processor, or device at a specified input time. From that point on any newly arriving data, event, or event data is not available to the receiving component until the next dispatch (PLG: Input_Time, not really dispatch), i.e., the input is frozen.

(17) The Input_Time property can have a list of values. In this case it indicates that input is frozen multiple times for the execution of a dispatch.

1. **Actual input**

   (15) The Input_Time property can be used to explicitly specify an input time for ports. This can be done for all ports by specifying the property value for the thread, or it can be specified separately for each port. (PLG: may some ports inheriting thread property while others have their own specification?) Yue: The property definition declares that this property could only apply to a port not to a thread. Then how to specify it for a thread?

   (40) A Receive_Input runtime service allows the source text of a thread to explicitly request port input on its incoming ports to be frozen and made accessible through the port variables....The Receive_Input service takes a mask parameter that specifies for which ports the input is frozen. (PLG: links with Input_Time ???)

   (42) A Get_Value runtime service shall be provided that allows the source text of a thread to access the current value of a port variable. The service call returns the data value. Repeated calls to Get_Value result in the same value to be returned, unless the current value is updated through a Receive_Input call or a Next_Value call.

   PLG: as far as I understand these rules allow several freezing between two dispatches, not only at dispatch time as indicated in (13).

   There are some questions concerning the consistency:

   - Is it possible to freeze input after completion ? The reasonable answer is probably that a thread cannot emit complete before all Input_Time occurrences. But in AADL an Input_Time may follow the Completion_Time !!! How is this possible if the thread is not running (see (40) below) ??? Does this means that input freezing is done by some AADL implicit action ??? How this policy can be made consistent with hidden Receive_Input calls.
   - Negative time associated with Completion_Time is generally not causal !!!

(9.1.4(28)) Arrival of events on event ports can also trigger a mode switch if the event port is named in a mode transition originating in the current mode (see Section 12). Events that trigger mode transitions are not queued at event ports.

2. **Input ports and Polychrony**

   The input port behavior induces an event Signal-signal InEvent for a port. InEvent has as many occurrences as given by Input_Time list. This occurrences may be dynamically generated according to queue size, TimeOffset, ...

Output ports (27) The Output_Time property can have a list of values. In this case it indicates that output is transmitted multiple times as part of the execution of a dispatch.

1. **Actual output**

   (38) A Send_Output runtime service allows the source text of a thread to explicitly cause events, event data, or data to be transmitted through outgoing ports to receiver ports. The Send_Output service takes a mask parameter that specifies for which ports the transmission is initiated. Send_Output is a non-blocking service. (PLG: links with Output_Time ???) Yue: for event (event data) ports, the output could occur anytime during the execution through a Send_Output service call.

   (39) A Put_Value runtime service allows the source text of a thread to supply a data value to a port variable. This data value will be transmitted at the next Send_Output call in the source text or by the runtime system at completion time or deadline.

   PLG: These rules allow several sending between two dispatches.

   There are some questions concerning the consistency:

   - Is it possible to send after completion ? The reasonable answer is probably that a thread cannot emit complete before all Output_Time occurrences. But in AADL an Output_Time may follow the Completion_Time !!! How is this possible if the thread is not running (see (40) below) ??? Does this means that output is achieved by some AADL implicit action ??? How this policy can be made consistent with hidden Send_Output calls.
   - Negative time associated with Completion_Time is generally not causal !!!

2. **Output ports and Polychrony**

   The output port behavior induces an event Signal-signal OutEvent for a port. OutEvent has as many occurrences as given by Output_Time list. This occurrences may be dynamically generated according to queue size, TimeOffset, ... and Fan_Out_Policy.

   A Fan_Out_Policy that is not the standard Broadcast policy, will generate a Signal-process in charge of this policy

   An AADL in out port is translated into a front-end Signal-process depending upon the port properties to manage directed connections.

11.1.4 Data port

(9) Data ports are intended for transmission of state data such as signals. Therefore, no queuing is supported for data ports. A thread can determine whether the input buffer of an in data port has new data at this dispatch by checking the port status trough a Get_Count service call, which is accessible through the port variable through a Get_Value service call. If no new data value has been received the old value is made available.

(9.1(L10))A data port cannot be the destination of more than one semantic port connection unless each semantic port connection is contained in a different mode.

(5.4.6(71))...data port connections across synchronization domains are sampled connections.

1. **Aggregate data port**

   (8.3.1(15))The role of an aggregate data port is to make a collection of data from multiple outgoing data ports available in a time-consistent manner. Time consistency in this context means that if a set of periodic threads is dispatched at the same time to operate on data, then the recipients of their data see either all old values or all new values. This is accomplished by declaring a data port, whose data classifier has an implementation with data components corresponding to the data of the individual data ports.

   The functionality of an aggregate data port can be viewed as a thread whose only role is to collect the data values from several in data ports and make them available as an aggregate data record; on the receiving side an equivalent thread takes passes on the elements of the aggregate data record on to the respective out data ports of receiving threads....

2. **Behavior**

It seems that data ports can have multiple output and multiple input during an AADL thread dispatch

(24) By default, the output time, i.e., the time output is transmitted to connected components, is the completion time for data ports.

3. **Data ports and Polychrony**

   Data port can be represented using cell Signal-process. A data port is close to a Signal-signal (several data ports can be synchronous).

   An aggregate data port can be implemented as indicated in AADL-8.1(7), as a Signal-process that builds a struct before sending values, and counterpart one that breaks the struct before delivering individual flows. Yue: no aggregate data port exists any more in AADLv2.

   So we need a Signal-process model for input data port and a Signal-process model for output data port. These models may be a unique common model..

   A data port supports only one value. If no new data value has been received, the old value is made available. A data port could be represented by a buffer, where a data written to the buffer remains there, until it is overwritten by a new one.

```
process buffer=(? i; ! o)
(| interleave(i, o)
 | o := current{false}(i, ^o)
 |)

process current =
{boolean v0;}
(? wx; event c; ! rx;)
(| rx := (wx cell c init v0) when c
 |)

process interleave =
(? x, sx;)
(| b:= not (b$1 init false)
 | x ^= when b
 | sx ^= when (not b)
 |) where boolean b; end;
```

4. **In data port and Polychrony**

The latest value of in data port buffer is frozen at InEvent time, and this value is memorized in M.

```
process M = (? i; ! o)
(| o := (i cell ^o) when ^o
 |)
```

The *Frozen_data_ port* copies the latest value of the *buffer* at specified time instant (*InEvent*). It can be represented by a cell and when operation.

```
process Frozen_data_port =
(? ii; event InEvent; ! oo;)
(| oo := ii cell InEvent when InEvent |)
```

A notation **DataPortTranslation**()is used to represent the syntax translation schema for *data port*. Only the **Timing** and **Input_Time** property are considered.

$$\begin{aligned} \text{Let } x &= (x_1, x_2, \{x_3, x_4\}) \in In\_data\_port \\ \text{where: } x_1 &\in portID \\ x_2 &\in Data\_reference \\ x_3 &= Input\_Time \in Port\_property \\ x_4 &= Timing \in Port\_property \end{aligned}$$

A notation $V(p)$ is used to represent the value of a property p, and $Count(v)$ represents the number of a value v (if v is a list).

(a) $V(x_4)$ = Sampled.
The **Timing** property is specified as *sampled* or not specified.

i. $Count(V(x_3)) = 1$
Input_Time contains only one value, or **Input_Time** not specified. The *InEvent* (the actual input time) is under time constraint (*Between*) (Figure 14).
The *InEvent* (*Controlled_Time*) must be in the time range of [*ReferenceTime + min_offset , ReferenceTime+max_offset*]. *ReferenceTime* is the reference time (Specified by **Input_Time** property, which could be *Dispatch*, *Start* ...). *timeunit* is the unit of two time offsets: *min_offset, max_offset*.

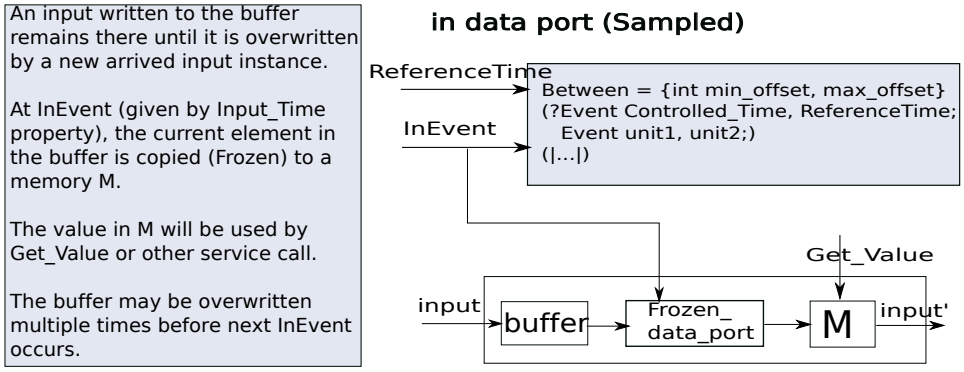


Figure 14: In data port (sampled)

```
process Between=
{ integer min_offset, max_offset }
(? event Controlled_Time, ReferenceTime, unit1, unit2;)
(|...|)
```

DataPortTranslation(x) =

process $InDataPort\_x_1'$ =
$(?\ x_2'\ i;$ **event** $InEvent,\ ReferenceTime,\ unit1,\ unit2;$
$!\ x_2'\ o;)$
$(|\ o1\ :=\ buffer(i)$
$|\ o2\ :=\ Frozen\_data\_port(o1,\ InEvent)$
$|\ Between\{\}(InEvent,\ ReferenceTime,\ unit1,\ unit2)$
$|\ o\ :=\ M(o2)$
$|)$

x_1' = **IDTranslation**(x_1) = x_1
x_2' = **DataReferenceTranslation**(x_2) = $Signal-type\ x_2$
x_3' = **PropertyTranslation**(x_3) = **process** $Between$

The *buffer(), Frozen_data_port()* and *M()* processes will be defined in a library. The translation of ID, Data_reference and Port_property will be introduced in later sections in detail. (The detailed **Input_Time** property translation can be found in Section 14.4.)

ii. $Count(V(x_3))$ = $n\ > 1$

Input_Time contains a list of values.
For example:
Input_Time: **list of** (Dispatch, 0.0ns .. 1.0ns) (Start, 0.0ns .. 1.0ns)
applies to portA;
Each value will have a corresponding *min_offset, max_offset* and *ReferenceTime* (Figure 15). The *InEvent* must be in the time range constraint *SeveralBetween*.

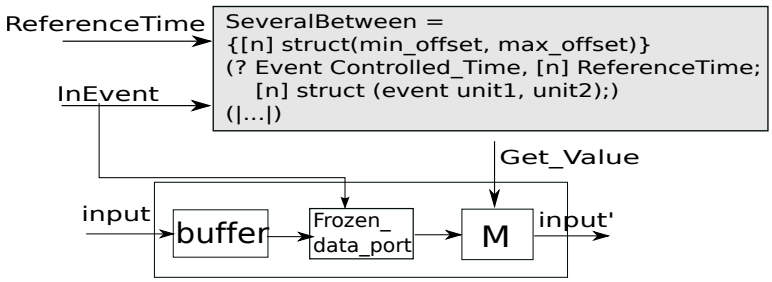


Figure 15: In data port (sampled): a list of values

$\mathbf{DataPortTranslation}(x)$ = **process** $InDataPort\_x_1'$ =
($?\ x_2'\ i$; **event** $InEvent$, $[n]$**event** $ReferenceTime$;
$[n]$ **struct** (**event** $unit1, unit2$);
$!\ x_2'\ o$;)
($|\ o1$:= $buffer(i)$
$|\ o2$:= $Frozen\_data\_port(o1,\ InEvent)$
$|\ SeveralBetween\{\}(InEvent,\ ReferenceTime,\ [n](unit1, unit2))$
$|\ o$:= $M(o2)$
$|$)
x_1' = $\mathbf{IDTranslation}(x_1)$ = x_1
x_2' = $\mathbf{DataReferenceTranslation}(x_2)$ = $Signal - type\ x_2$;
x_3' = $\mathbf{PropertyTranslation}(x_3)$ = **process** $SeveralBetween$
n = $Count(V(x_3))$

The translation of **Input_Time** property could be found in Section 14.4.

(b) $V(x_4)$ = $Immediate$. If the **Timing** property is declared as *immediate*, then the *InEvent* is *Start* and zero *Offset* (the **Input_Time** value is

ignored if it is defined). (Figure 16)

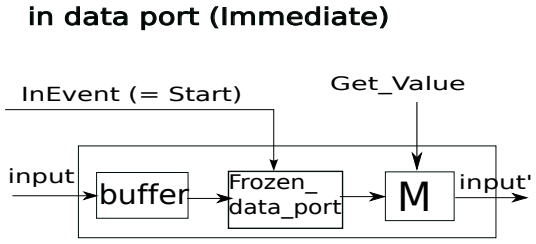


Figure 16: In data port (immediate)

In this case:

$$
\begin{aligned}
&\mathbf{DataPortTranslation}(x) \ = \ \mathbf{process}\ InDataPort\_x_1' \ = \\
&\qquad (?\ x_2'\ i;\ \mathbf{event}\ Start;\ !\ x_2'\ o;) \\
&\qquad (|\ o1 \ := \ buffer(i) \\
&\qquad |\ o2 \ := \ Frozen\_data\_port(o1,\ Start) \\
&\qquad |\ o \ := \ M(o2) \\
&\qquad |) \\
&x_1' \ = \ \mathbf{IDTranslation}(x1) \\
&x_2' \ = \ \mathbf{DataReferenceTranslation}(x2)
\end{aligned}
$$

(c) $V(x_4)$ = *Delayed*. **Delayed**. If the **Timing** property is declared as *delayed*, then the *InEvent* is *Dispatch* and zero *Offset* (the defined **Input_Time** value is ignored). (Figure 17.)

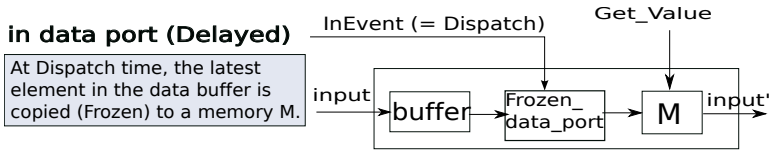


Figure 17: In data port (delayed)

In this case:

$$
\begin{aligned}
&\textbf{DataPortTranslation}(x) \ = \ \textbf{process}\ InDataPort\_x_1' \ = \\
&\qquad (?\ x_2'\ i;\ \textbf{event}\ Dispatch;\ !\ x_2'\ o;\ ) \\
&\qquad (|\ o1 \ := \ buffer(i) \\
&\qquad |\ o2 \ := \ Frozen\_data\_port(o1,\ Dispatch) \\
&\qquad |\ o \ := \ M(o2) \\
&\qquad |) \\
&x_1' \ = \ \textbf{IDTranslation}(x1) \\
&x_2' \ = \ \textbf{DataReferenceTranslation}(x2)
\end{aligned}
$$

5. **Out data port and Polychrony**

   Only the **Timing** and **Output_Time** property are considered.

$$
\begin{aligned}
&\text{Let } x = (x_1, x_2, \{x_3, x_4\}) \in Out\_data\_port \\
&\text{where: } x_1 \in portID \\
&\qquad x_2 \in Data\_reference \\
&\qquad x_3 = Output\_Time \in Port\_property \\
&\qquad x_4 = Timing \in Port\_property
\end{aligned}
$$

   The Output is sent out (Send) at OutEvent time. A Distributer will select the recipients depending on Fan_Out_Policy.

```
process Send = (? i, OutEvent; ! o;)
  (| o := i cell OutEvent when OutEvent | )

process Distributer = (? i; ! o;)
  (|...|)
```

   (a) **Immediate**. $V(x_4) = Immediate$. If the **Timing** property is declared as *immediate*, the *OutEvent* is *Completion* (Figure 18). The value of **Output_Time** is ignored.

   In this case:

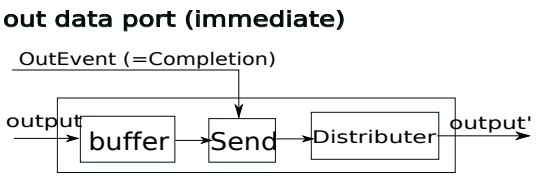


Figure 18: Out data port (immediate)

$$\begin{aligned}
&\mathbf{DataPortTranslation}(x) \ = \ \mathbf{process}\ OutDataPort_x_1' \\
&\qquad (?\ x_2'\ i;\ \mathbf{event}\ Completion;\ !\ x_2'\ o;\ ) \\
&\qquad (|\ o1\ :=\ buffer(i) \\
&\qquad |\ o2\ :=\ Send(o1,\ Completion) \\
&\qquad |\ o\ :=\ Distributer(o2) \\
&\qquad |) \\
&x_1' \ = \ \mathbf{IDTranslation}(x1) \\
&x_2' \ = \ \mathbf{DataReferenceTranslation}(x2)
\end{aligned}$$

(b) **Delayed**. $V(x_4)$ = $Delayed$. Similar as *immediate*. **Output_Time** is ignored. The *OutEvent* is *Deadline*.
In this case:

$$\begin{aligned}
&\mathbf{DataPortTranslation}(x) \ = \ \mathbf{process}\ OutDataPort_x_1' \ = \\
&\qquad (?\ x_2'\ i;\ \mathbf{event}\ Deadline;\ !\ x_2'\ o;\ ) \\
&\qquad (|\ o1\ :=\ buffer(i) \\
&\qquad |\ o2\ :=\ Send(o1,\ Deadline) \\
&\qquad |\ o\ :=\ Distributer(o2) \\
&\qquad |) \\
&x_1' \ = \ \mathbf{IDTranslation}(x1) \\
&x_2' \ = \ \mathbf{DataReferenceTranslation}(x2)
\end{aligned}$$

(c) **Sampled**. $V(x_4)$ = $Sampled$.
The *OutEvent* is restricted by a constraint *Between* (or *SeveralBetween*, depending on the number of values the *Output_Time* property specifies). (Figure 19)

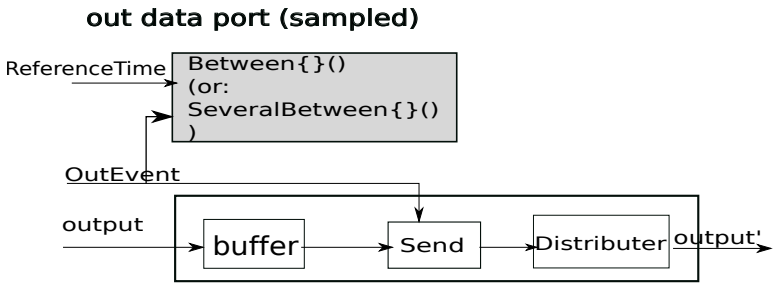


Figure 19: Out data port (sampled)

i. In case of one **Output_Time** value, $Count(V(x_3) = 1)$:

$$
\begin{aligned}
&\textbf{DataPortTranslation}(x) \ = \ \textbf{process}\ OutDataPort\_x'_1 = \\
&\qquad (?\ x'_2\ i;\ \textbf{event}\ OutEvent,\ ReferenceTime,\ unit1,\ unit2; \\
&\qquad !\ x'_2\ o;) \\
&\qquad (|\ o1\ :=\ buffer(i) \\
&\qquad |\ o2\ :=\ Send(o1,\ OutEvent) \\
&\qquad |\ Between\{\}(OutEvent,\ ReferenceTime,\ unit1,\ unit2) \\
&\qquad |\ o\ :=\ Distributer(o2) \\
&\qquad |) \\
&x'_1\ =\ \textbf{IDTranslation}(x_1) \\
&x'_2\ =\ \textbf{DataReferenceTranslation}(x_2) \\
&x'_3\ =\ \textbf{PropertyTranslation}(x_3)
\end{aligned}
$$

ii. In case of a list of **Output_Time** value, $n = Count(V(x_3)\ > 1)$:

$$
\begin{aligned}
&\mathbf{DataPortTranslation}(x) \ = \ \mathbf{process}\ OutDataPort\_x_1' = \\
&\qquad (?\ x_2'\ i;\ \mathbf{event}\ OutEvent,\ [n]\mathbf{event}\ ReferenceTime; \\
&\qquad [n]\mathbf{struct}\ (\mathbf{event}\ unit1, unit2); \\
&\qquad !\ x_2'\ o;) \\
&\qquad (|\ o1\ :=\ buffer(i) \\
&\qquad |\ o2\ :=\ Send(o1,\ OutEvent) \\
&\qquad |\ SeveralBetween\{\}(OutEvent,\ ReferenceTime,\ [n](unit1,\ unit2)) \\
&\qquad |\ o\ :=\ Distributer(o2) \\
&\qquad |) \\
x_1' \ &= \ \mathbf{IDTranslation}(x_1) \\
x_2' \ &= \ \mathbf{DataReferenceTranslation}(x_2) \\
x_3' \ &= \ \mathbf{PropertyTranslation}(x_3)
\end{aligned}
$$

6. **In out data port and Polychrony**

   In out data port is separated into in and out two ports?

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. Standard properties

   - Port specific compute entrypoint properties for event and event data ports:

     Compute_Entrypoint: **classifier** (Subprogram Classifier) **applies to** (thread, device, provides subprogram access, event port, event data port);
     Compute_Execution_Time: Time_Range **applies to** (thread, device, subprogram, event port, event data port);
     Compute_Deadline: Time **applies to** (thread, device, subprogram, subprogram access, event port, event data port);

     (4) Event (-data) ports may dispatch a port specific Compute_Entrypoint. This permits threads with multiple event or event data ports to execute different source text sequences for events arriving at different event ports (PLG: such an entry is a black box in a gray box) If specified, the port specific Compute_Execution_Time and Compute_Deadline takes precedence over those of the containing thread.

   - **Queue_Processing_Protocol**. Queues will be serviced according to this property, by default in a FIFO order. An event (event data) port could be represented by a Signal FIFO.

     Queue_Processing_Protocol: Supported_Queue_Processing_Protocols ⇒ FIFO **applies to** (event port, event data port, subprogram access);

   - **Queue_Size**. The default port queue size is 1.

     Queue_Size: **aadlinteger** 0 .. Max_Queue_Size ⇒ 1 **applies to** (event port, event data port, subprogram access);

   - **Dequeue_Protocol**. This property specifies the dequeuing option to the receiving application.

     Dequeue_Protocol: **enumeration** (OneItem, MultipleItems, AllItems) ⇒ OneItem **applies to** (event port, event data port);

     - **OneItem**: a single frozen item is dequeued and made available to the source text unless the queue is empty.
     - **AllItems**: all items that are frozen at input time are dequeued and mad available to the source text via the port variable.
     - **MultipleItems**: multiple items can be dequeued one at a time from the frozen queue.

   - **Dequeued_Items**. This property specifies the maximum number of items that ate made available to the application when the input is frozen at input time.

     Dequeued_Items: **aadlinteger applies to** (event port, event data port);

- **Overflow_Handling_Protocol**. This property determine the action, when an event (event data) arrives and the number of queued events is equal to the specified queue size.

  Overflow_Handling_Protocol: **enumeration** (DropOldest, DropNewest, Error) ⇒ DropOldest **applies to** (event port, event data port, subprogram access);

2. **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).

3. **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.

4. **Port queue and Polychrony**

   A Signal-process is dedicated to manage the port queue. It is made of a FIFO 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,...)

```
process FIFO =
{ integer n;}
( ?  x_in;
  !  x_out; integer nbmsg;)
(| prev_nbmsg := nbmsg$1 init 0
 | OK_write := prev_nbmsg < n
 | OK_read := prev_nbmsg > 0
 | nbmsg := ((prev_nbmsg+1) when (^x_in) when OK_write)
```

```
        default ((prev_nbmsg-1) when (^x_out) when OK_read)
        default prev_nbmsg
| access_clk ^= ^x_in default ^x_out
| nbmsg ^= access_clk
| queue := (x_in window n) cell (^access_clk)
| x_out := prev_msg_out when (not OK_read) when (^x_out)
       default queue[n - prev_nbmsg] when (^x_out)
| prev_msg_out := x_out $ 1
|)
where
  integer prev_nbmsg; [n]boolean queue;
   prev_msg_out; event access_clk;
  boolean OK_write, OK_read;
end;
```

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

   An event (event data) port could be represented by a pair of FIFOs (*Ex-FIFO* and *In-FIFO*) and a container of constraints. (Figure 20). *Ex-FIFO* receives inputs from other threads. At *InEvent* (constraint by **Input_Time** in *Between*), move (*Frozen*) a number of elements (the number depends on the **Dequeue_Protocol**) from *Ex-FIFO* to *In-FIFO*. The inputs arrived after the *InEvent* will be available at the next *InEvent*. The elements in *In-FIFO* will be used through *Next_Value* service call.

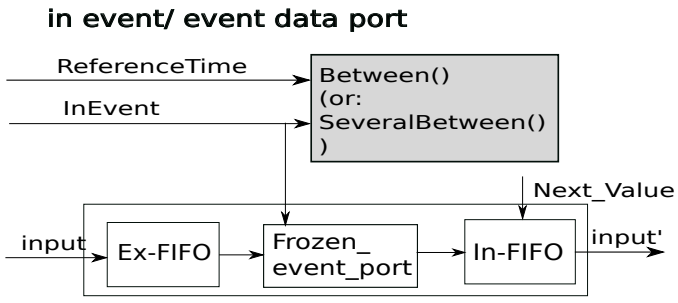


Figure 20: In event port

At **InEvent**, frozen the inputs: copy k elements of *Ex-FIFO* into internal FIFO (*In-FIFO*).

```
process Frozen_event_port =
{integer n,k;}
```

```
(? event InEvent;  x_in;
 !  x_out;)
(| EX_FIFO :: (y, nbmsg1) := FIFO{n}(x_in)
 | y ^= frozen_tick
 | frozen_tick := m_Overspl{10000,k}(InEvent)
 | IN_FIFO :: (x_out, nbmsg2) := FIFO{n}(y)
 |)
where
    y;
   integer nbmsg1, nbmsg2;
   event frozen_tick;
   label EX_FIFO, IN_FIFO;
end;

process m_Overspl = {integer m,k; }
  (? event tick;
   ! event tick_out;)
  (| cnt := (m-1 when tick) default (pre_cnt-1)
   | pre_cnt := cnt $ 1 init 1
   | tick ^= when (pre_cnt <= 1)
   | tick_out ^= when ( cnt >= (m-k) )
   |)
   where
     integer cnt, pre_cnt;
end;
```

The *Dequeue_number* k is decided by **Dequeue_Protocol** and **Dequeued_Items** property.

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

(a) **In_event_port**

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

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

$\mathbf{EventPortTranslation}(x) =$
$\mathbf{process}\ InEventPort\_x'_1 =$
$\{\mathbf{integer}\ n,\ k;\}$
$(?\ \mathbf{event}\ i,\ InEvent;\ [m]\mathbf{event}\ ReferenceTime;$
$!\ \mathbf{event}\ o;)$
$(|\ o := Frozen\_event\_port\{n,\ k\}(InEvent,\ i)$
$|\ x'_2$
$|)$
$x'_1 = \mathbf{IDTranslation}(x_1)$
$x'_2 = \mathbf{PropertyTranslation}(x_2)$
$n = \mathbf{PropertyTranslation}(x_3)$
$k = \begin{cases} n, & \text{if } V(x_4) = \text{AllItems} \\ \mathbf{PropertyTranslation}(x_5), & \text{if } V(x_4) = \text{MultiItems} \\ 1, & \text{if } V(x_4) = \text{OneItem} \end{cases}$
$m = Count(V(x_2))$

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

(b) **In_event_data_port**

Let $x = (x_1, \{x_2, x_3, x_4, x_5\}, x_6) \in In\_event\_data\_port$,
where : $x_1 \in portID$
$x_2 = Input\_Time \in Port\_property$
$x_3 = Queue\_Size \in Port\_property$
$x_4 = Dequeue\_Protocol \in Port\_property$
$x_5 = Dequeue\_Items \in Port\_property$
$x_6 \in Data\_reference$

$\mathbf{EventDataPortTranslation}(x)$ =
$\mathbf{process}\ InEventDataPort\_x'_1$ =
$\{\mathbf{integer}\ n,\ k;\}$
$(?\ x'_6\ i,\ \mathbf{event}\ InEvent;\ [m]\mathbf{event}\ ReferenceTime;$
$!\ x'_6\ o;)$
$(|\ o\ :=\ Frozen\_event\_port\{n,\ k\}(InEvent,\ i)$
$|\ x'_2$
$|)$
x'_1 = $\mathbf{IDTranslation}(x_1)$
x'_2 = $\mathbf{PropertyTranslation}(x_2)$
n = $\mathbf{PropertyTranslation}(x_3)$
$$k = \begin{cases} n, & \text{if } V(x_4) = \text{AllItems} \\ \mathbf{PropertyTranslation}(x_5), & \text{if } V(x_4) = \text{MultiItems} \\ 1, & \text{if } V(x_4) = \text{OneItem} \end{cases}$$
m = $Count(V(x_2))$
x'_6 = $\mathbf{DataReferenceTranslation}(x_6)$

6. Out event (event data) port and Polychrony

   The Output is stored in a FIFO, and sent out at *OutEvent* time (Figure 21). The *OutEvent* is restricted by a constraint *Between* (or *SeveralBetween*). A *Distributer* selects the recipients depending on the **Fan_Out_Policy**.

   (a) **Out_event_port**

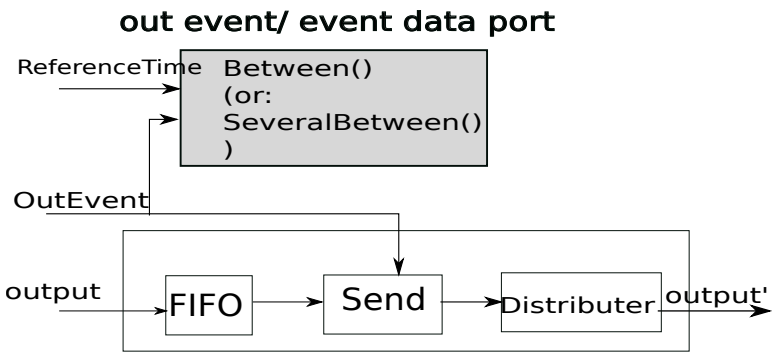


Figure 21: Out event port

Let $x = (x_1, \{x_2, x_3\}) \in Out\_event\_port$
where: $x_1 \in portID$
$x_2 = Output\_Time \in Port\_property$
$x_3 = Queue\_Size \in Port\_property$

$$
\begin{aligned}
&\textbf{EventPortTranslation}(x) = \\
&\quad \textbf{process}\ OutEventPort\_x_1' = \\
&\quad \{\textbf{integer}\ n;\} \\
&\quad (?\ \textbf{event}\ i,\ InEvent;\ [m]\textbf{event}\ ReferenceTime; \\
&\quad !\ \textbf{event}\ o;) \\
&\quad (|\ (o1, nbmsg) := FIFO\{n\}(i) \\
&\quad |\ o2 := Send(o1, OutEvent) \\
&\quad |\ x_2' \\
&\quad |\ o := Distributer(o2) \\
&\quad |) \\
&n = \textbf{PropertyTranslation}(x_3) \\
&m = Count(V(x_2)) \\
&x_1' = \textbf{IDTranslation}(x_1) \\
&x_2' = \textbf{PropertyTranslation}(x_2)
\end{aligned}
$$

The *Dequeue_Items* ($F(x_3)$) and *Queue_Size* ($F(x_4)$) property associations will be interpreted in the property section.

(b) **Out_event_data_port**

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

$\mathbf{EventDataPortTranslation}(x) =$
process $OutEventDataPort\_x'_1 =$
$\{\mathbf{integer}\ n;\}$
$(?\ x'_4\ i;\ \mathbf{event}\ InEvent;\ [m]\mathbf{event}\ ReferenceTime;$
$!\ x'_4\ o;)$
$(|\ (o1, nbmsg) := FIFO\{n\}(i)$
$|\ o2 := Send(o1, OutEvent)$
$|\ x'_2$
$|\ o := Distributer(o2)$
$|)$
$n = \mathbf{PropertyTranslation}(x_3)$
$m = Count(V(x_2))$
$x'_1 = \mathbf{IDTranslation}(x_1)$
$x'_2 = \mathbf{PropertyTranslation}(x_2)$
$x'_4 = \mathbf{DataReferenceTranslation}(x_4)$

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

11.1.6 Port and Polychrony

An event (data) port differs from a Signal-signal in that a single Event (data) port is transmitted at each dispatch.

Event data ports can be represented by FIFOs (or FIFO pairs, the last FIFO contains the frozen values) or cell Signal-processes (extended "at" of Yue) following the port queue property.

Event ports can be represented by counters (?)
The meaning of frozen is not fully clear in my mind.
Persistence of queued events through mode transitions ?

11.2 Parameter

(1) Subprogram parameter declarations represent data values that can be passed into and out of subprograms. Parameters are typed with a data classifier reference representing the data type.

11.2.1 Abstract syntax of *Parameter*

$$Parameter ::= parameterID \times Parameter\_direction \times \mathbf{opt}(Data\_reference) \times \mathbf{opt}(\mathbf{list}(Parameter\_property))$$
$$Parameter\_direction ::= \{in,\ out,\ \{in\ out\}\}$$

The **in out** parameter declaration represents a parameter whose value is passed in and returned by value. Parameters passed by reference are modeled using **requires data access**.

11.2.2 Standard properties

Property	In Parameter	Out Parameter
Allowed_Connection_Binding_Class	X	X
Allowed_Connection_Binding	X	X
Actual_Connection_Binding	X	X
Required_Connection	X	X
Acceptable_Array_Size	X	X

11.2.3 Parameter and Polychrony

A parameter could be modeled as a Signal signal?

$$\text{Let } x = (x_1, x_2, x_3) \in Parameter$$
$$\text{where } x_1 \in parameterID$$
$$x_2 \in Data\_reference$$
$$x_3 \in Parameter\_property$$

$$\mathbf{ParameterTranslation}(x) \;=\; x'_2\; x'_1$$
$$x'_1 \;=\; \mathbf{IDTranslation}(x_2) \;=\; x_1$$
$$x'_2 \;=\; \mathbf{DataReferenceTranslation}(x_2) \;=\; signal\_type\; x_2$$

11.3 Subprogram and subprogram group access

11.3.1 Subprogram access

(8.3(1)) ... Subprogram access is used to model binding of a subprogram call (local or remote) to the subprogram instance being called.

1. **Abstract syntax of *Subprogram access***

$$Subprogram\_access ::= subprogram\_access\_ID \times Access\_status \times \mathbf{opt}(Subprogram\_reference) \times \mathbf{opt}(\mathbf{list}(Subprogram\_access\_property))$$
$$Access\_status ::= \{provides, requires\}$$
$$Subprogram\_reference ::= subprogramID$$

2. **Standard properties**

Input_Rate: Rate_Spec
Output_Rate: Rate_Spec

(8.3-(7)) Input_Rate and Output_Rate specify the rate at which a subprogram is called. (PLG: As rate in ports) Yue: in the property definition, Input_Rate and Output_Rate could only apply to ports. Maybe the property Subprogram_Call_Rate could be used instead.

Property	Provides	Requires
Allowed_Connection_Binding_Class	X	X
Allowed_Connection_Binding	X	X
Actual_Connection_Binding	X	X
Allowed_Subprogram_Call	X	X
Actual_Subprogram_Call	X	X
Overflow_Handling_Protocol		X
Queue_Processing_Protocol	X	X
Queue_Size	X	X
Required_Connection	X	X
Subprogram_Call_Rate	X	X
Compute_Entrypoint	X	
Compute_Entrypoint_Call_Sequence	X	
Compute_Entrypoint_Source_Text	X	
Acceptable_Array_Size	X	X

3. **Subprogram access and Polychrony**

11.3.2 Subprogram group access

1. **Abstract syntax of *Subprogram_group_access***

$$Subprogram\_group\_access ::= subprogram\_group\_access\_ID \times Access\_status \times \mathbf{opt}(Subprogram\_group\_reference) \times \mathbf{opt}(\mathbf{list}(Subprogram\_group\_access\_property))$$

$$Subprogram\_group\_reference ::= subprogram\_group\_ID$$

2. **Standard properties**

Property	Provides	Requires
Allowed_Connection_Binding_Class	X	X
Allowed_Connection_Binding	X	X
Actual_Connection_Binding	X	X
Required_Connection	X	X
Acceptable_Array_Size	X	X

3. **Subprogram group access and Polychrony**

11.4 Data access

Data access is used to model shared data.

Components can declare that they require access to externally declared data components. Components may provide access to their data components.

11.4.1 Abstract syntax of *Data access*

A requires data access declaration indicates that a component requires access to a component declared external to the component. For data components, different forms of required access, such as read-only access, are specified by a Access_Right property.

A provides data access declaration indicates that a subcomponent provides access to a data component contained in the component.

Shared data may be accessed by multiple threads. Such potential concurrent access is controlled according to the Concurrency_Control_Protocol property. This property applies to data.

$$Data\_access ::= data\_access\_ID \times Access\_status \times \mathbf{opt}(Data\_reference) \times \mathbf{opt}(\mathbf{list}(Data\_access\_property))$$
$$Data\_reference ::= dataID$$

Figure 22 shows two types of data access. *Data1* is made accessible outside *Thread1* through a *provides data access* feature declaration of *Thread1*. It is being accessed by *Thread2* as expressed by a *requires data access* feature declaration in the thread type of *Thread2*.

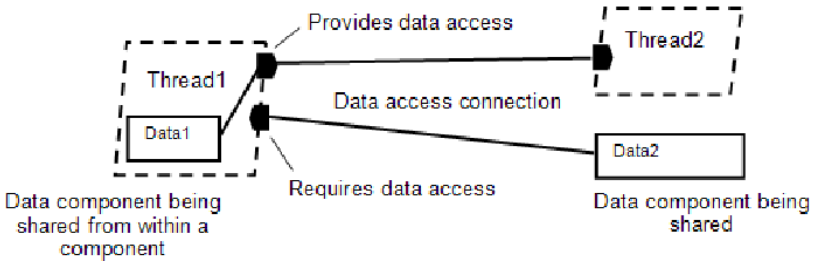


Figure 22: Data access

```
thread Thread1
  features
```

```
    Dataset: provides data access Data1;
end Thread1;

thread Thread2
  features
    Reqdataset: requires data access Data1;
end Thread2;
```

11.4.2 Standard properties

Property	Provides	Requires
Access_Right	X	X
Access_Time	X	X
Base_Address	X	X
Acceptable_Array_Size	X	X

The **Access_Time** property specifies the time range over which a component has access to a shared data component. By default, access is required for the duration of the component execution. The value of a shared data component is read or written through the use of a data variable that represents the shared data component, or through Get_Value and Put_Value service calls. Write access immediately updates the shared data component.

Access_Time : **record** (First: IO_Time_Spec; Last: IO_Time_Spec;)
⇒ (First ⇒ (Time ⇒ Start; Offset ⇒ 0.0ns .. 0.0 ns;);
Last ⇒ (Time ⇒ Completion; Offset ⇒ 0.0ns .. 0.0ns;);)
applies to (data access);

Between First time and Last time, the data component is being accessed. First (Last) is specified by a reference time and a time offset range. It is modeled as a constraint.

11.4.3 Data access and Polychrony

1. **Requires data access**

   In Figure 23, two constraints are added. They represent the *First* and *Last* access time specified by **Access_Time** property. Each access time is represented by a renference time (*Time*) and an *Offset*.

   Get_Value, Get_Resource and *Release_Resource* are three predefined service calls. At *FirstTime*, a *Get_Resource* is performed to lock the data resource. At *LastTime*, a *Release_Resource* is sent out to unlock the resource.

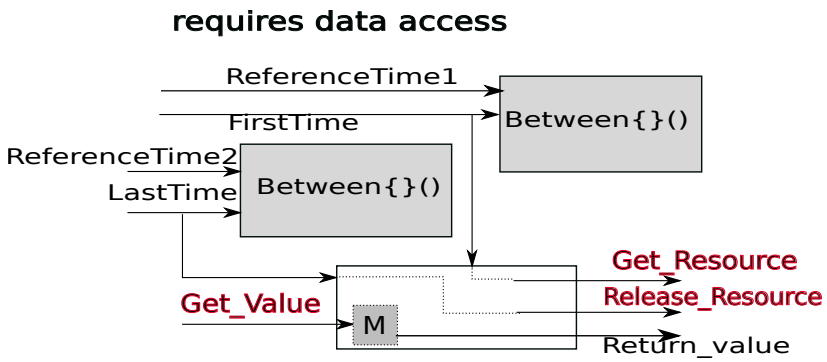


Figure 23: Requires data access

A *Get_Value* may be sent to the corresponding data resource during the execution depending on the detailed implementation. The required value is returned (*Return_value*) and stored in a memory *M*, and it will be updated when a new *Get_Value* is performed (a new value is returned).

$$\begin{aligned} \text{Let } x &= (x_1, x_2, x_3) \in Requires\_data\_access \\ \text{where } x_1 &\in data\_access\_ID \\ x_2 &\in Data\_reference \\ x_3 &= Access\_Time \in Data\_access\_property \end{aligned}$$

RequiresDataAccessTranslation(x) =
process $DataAccess\_x'_1$ =
(? **event** $FirstTime$, $ReferenceTime1$, $LastTime$, $ReferenceTime2$;
!$Return\_value$)
(| x'_{31}
| x'_{32}
| $Get\_Resource(FirstTime,\ x'2)$
| $Return\_value$:= $Get\_Value(x'2)$
| $Release\_Resource(LastTime,\ x'2)$
|)
x'_1 = **IDTranslation**(x_1)
x'_2 = **DataReferenceTranslation**($x2$)
(x'_{31}, x'_{32}) = **PropertyTranslation**(x_3)

Get_Resource(), *Release_Resource*() and *Get_Value*() are defined in a library. **Access_Time** property translation is explained in Section 14.4.

```
process Get_Resource =
(? event time; Resource;)

process Release_Resource =
(? event time; Resource;)

process Get_Value =
(? Resource; ! ReturnValue;)
```

2. **Provides data access**

   At *FirstTime*, a *Get_Resource* service call is performed. At *LastTime*, a *Release_Resource* service call is performed. A *Put_Value* service call is used to write a value. (Figure 24)

Let $x = (x_1, x_2, x_3) \in Provides\_data\_access$
where$x_1 \in data_access_ID$
$x_2 \in Data\_reference$
$x_3 = Access\_Time \in Data\_access\_property$

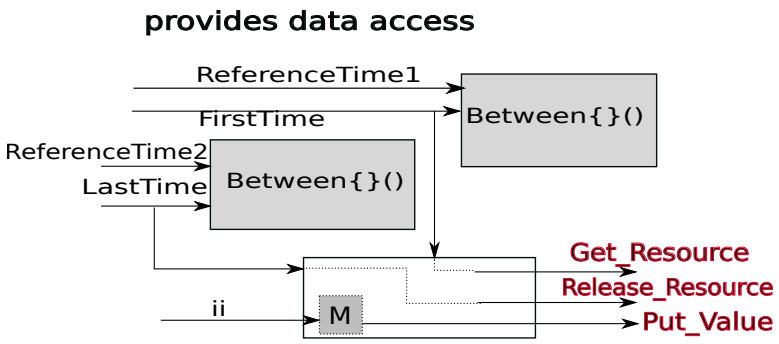


Figure 24: Provides data access

ProvidesDataAccessTranslation(x) =
process $DataAccess\_x'_1$ =
(? **event** $FirstTime$, $ReferenceTime1$, $LastTime$, $ReferenceTime2$; ii)
(| x'_{31}
| x'_{32}
| $Get\_Resource(FirstTime,\ x'2)$
| $Put\_Value(x'2, ii)$
| $Release\_Resource(LastTime,\ x'2)$
|)
x'_1 = **IDTranslation**(x_1)
x'_2 = **DataReferenceTranslation**$(x2)$
(x'_{31}, x'_{32}) = **PropertyTranslation**(x_3)

```
process Put_Value =
(? Resource; Value;)
```

11.5 Bus access

Bus components can be made accessible to other components. Components can declare that they require access to externally declared buses. Components may provide access to their buses. Bus access is used to model connectivity of execution platform components through buses.

A requires bus component access declaration indicates that a component requires access to a component declared external to the component. Required bus

accesses are resolved to actual bus subcomponents through access connection declarations.

A provides bus access declaration indicates that a subcomponent provides access to a bus contained in the component. Provided bus accesses can be used to resolve required bus access.

A bus that is accessed by more than one component is shared. The shared bus is a common resource through which processor, memory and device components communicate.

11.5.1 Abstract syntax of *Bus access*

$$Bus\_access ::= bus\_access\_ID \times Access\_status \times \mathbf{opt}(Bus\_reference) \times \mathbf{opt}(\mathbf{list}(Bus\_access\_property))$$
$$Bus\_reference ::= busID$$

11.5.2 Standard properties

Property	Provides	Requires
Access_Right	X	X
Acceptable_Array_Size	X	X

11.5.3 Bus access and Polychrony

11.6 Feature group

Feature groups represent groups of component features of feature groups.

The content of a feature group is declared through a feature group type declaration. This declaration is then referenced when feature groups are declared as component features.

Within a component, the features of a feature group can be connected to individually. Outside a component, feature groups can be connected as a single unit.

A feature group can be declared to be the inverse of another feature group type.

A feature group of a component can be connected to another component through a single connection declaration. It represents a connection for each of the feature inside the feature group.

11.6.1 Abstract syntax of *Feature group*

$$Feature\_group ::= featuregroupID \times \mathbf{opt}(\mathbf{list}(Feature)) \times \mathbf{opt}(Inverse\_featuregroup\_reference) \times \mathbf{opt}(\mathbf{list}(Feature\_group\_property))$$
$$Inverse\_featuregroup\_reference ::= featuregroupID$$

11.6.2 Standard properties

Property	Provides	Requires
Device_Register_Address	X	X
Source_Text	X	X
Acceptable_Array_Size	X	X

11.6.3 Feature group and Polychrony

12 Connection

A connection is a linkage between features of two components that represents communication of data and control between components.

AADL supports connections between abstract features, feature groups connections, port connections, parameter connections and access connections.

Abstract syntax of *Connection*

$$Connection ::= Port\_connection + Parameter\_connection + Access\_connection + Feature\_group\_connection \quad (16)$$

12.1 Port connection

(9.1(1) Port connections represent transfer of data and control between two concurrently executing components.... These connections are semantic port connections. A semantic port connection is determined by a sequence of one or more individual port connection declarations that follow the component containment hierarchy in a fully instantiated system from an ultimate source to an ultimate destination.

(9.1(2) ... The ultimate source of a semantic port connection is ... an out or in out port of a thread, processor, or device component. The ultimate destination of a semantic port connection is .... an in or in out port of a thread, a processor, or a device component. (4) ... the ultimate source or the ultimate destination of a semantic port connection, but not both, can be a data component.

(9.1(4) Semantic port connections also represent the sampling of a data component content by a data or event data port, and updating a data component with the output of a data or event data port. In other words, the ultimate source or the ultimate destination of a semantic port connection, but not both, can be a data component.

(9.1(5) Semantic port connections may also route a raised event to a modal component through a sequence of connection declarations. A mode transition in such a component is the ultimate destination of the connection, if the mode transition names an in or in out event port in the enclosing component, or an out or in out event port of one of the subcomponents.

(9.1(3)) ... An individual port connection declaration links a (source) of one subcomponent to a (destination) of another subcomponent, i.e., it connects sibling components at the highest level in the component hierarchy required for the connection. Alternatively, a port connection declaration maps a (source) of a subcomponent to an outgoing port of a containing component or an incoming port of a containing component to a (destination) of a subcomponent. PLG: names them filiation connections, and sibling connections.

(9.1(6) Semantic port connections may exist between arrays of component instances...

Semantic port connection A semantic port connection is determined by a sequence of one or more individual port connection declaration that follow the component containment hierarchy in a fully instantiated system from an *ultimate source* to an *ultimate destination*.

An example:

```
connections
  C1:  data port port1 -> port2;
  C2:  event data port port3 -> port4;
  C3:  event port port5 -> port6;
```

12.1.1 Port connection categories

(10) A port connection declared with the optional in_modes_and_transitions subclause specifies whether the connection is part of specific modes or is part of the transition between two specific modes.

(L11) A semantic (data) connection cannot contain both immediate and delayed connection declarations.

(13) Event port connections may refer to an event source or event destination specification (self.eventname) (PLG ???). An event source specification indicates

that the component itself is the source of an event. In case of a thread this may be due to a Send_Output or Raise_Event system call or due to an event raised by the underlying runtime system, i.e., the processor. In case of incomplete system models it may also represent the fact that a subcomponent to be specified is the source of an event. An event destination specification indicates that the event may be destined for an event port in the execution platform component(s) the component is bound to, or for a subcomponent yet to be declared in an incomplete system model. (PLG: To be clarified)

12.1.2 Legal port connection

(L1) ...The sources and destinations must be features of an AADL-thread, AADL-thread group, AADL-process, processor, device, or system component as indicated in the following.

Legal port connections			**Destination**			
			1 Data, Data access	2 Data port	3 Event data port	4 Event port
Source	1	Data, Data access		X	X	X
	2	Data port	X	X	X	X
	3	Event data port	X	X	X	X
	4	Event port				X

- 1 → 2: Content of data component is sampled by data port at the specified input time
- 1 → 3: Content of data component is copied to the event data port when the data component is written to; the connection destination monitors write operations to data components (may not be supported by all runtime systems).
- 1 → 4 ??
- (2,3) → 1: Event data (3), data(2) port output is written into data component at the specified output time.
- 2 → 2: Data port output is transferred and available upon receipt as most recent value.
- 2 → (3,4): Data port output is transferred and received as event data (3), event (4), i.e., queued and may result in a dispatch. (PLG: 2 → 4 is not listed as acceptable in (L5))
- 3 → 2: Event data port output is transferred and available upon receipt as most recent value.

The ultimate source ... must be a feature of a thread, processor, or device.

The ultimate destination ... must be a port of a thread, a processor, a device, or a mode transition.

(L2) If the ultimate destination ... is a (PLG event port in a ?) mode transition, then the ultimate source must be an out event port. (L1) ...This mode transition must be declared in the mode subclause of a thread, thread group, process, system, device, bus, memory, or processor naming an in event port in one of its mode transitions.

(L3) If a semantic port connection may be active in a particular mode, then the ultimate source and ultimate destination components must be part of that mode.

(L4) If a semantic port connection may be active in a particular mode transition, then the ultimate source component must be part of a system mode that includes the old mode identifier and the ultimate destination component must be part of a system mode that includes the new mode identifier.

(from L7) $sibling\ connection \in \{out,\ in\ out\} \times \{in,\ in\ out\}$

(from L8) $filiation\ connections \in \{out,\ in\ out\}^2 \cup \{in,\ in\ out\}^2$

(from L9) connection between a data component and a port, then the data component must have the (correct) access right

(21)... Bi-directional flow between two components is represented by two connections between the in out ports of two components.

(9.1(L10))A data port cannot be the destination of more than one semantic port connection unless each semantic port connection is contained in a different mode.

N-to-n connectivity is supported for event and event data ports (9.1.2(16))

PLG: What about other connections (see table above)

(L17) A processor port specification must only be used in event connections within threads and subprograms. (PLG ???)

(C2) The processor port identifier of a processor port specification (processor.processor_port_identifier) must name a port of the processor that the thread is bound to.

(L12) The ultimate source (and destination) of an immediate or delayed port connection must be a periodic thread or periodic device.

Data type matching(see L13...L15)

(C1) There cannot be cycles of immediate connections between threads, devices, and processors.

(PLG: strong static rule that does not take mode into account?)

1. **Abstract syntax of *Port_connection***

$$\begin{aligned} Port\_conncetion ::= {} & Event\_event\_port\_connection + Data\_data\_port\_connection \\ & + Eventdata\_Eventdata\_port\_connection + Data\_eventdata\_port\_connection \\ & + Data\_event\_port\_connection + Eventdata\_data\_port\_connection \\ & + Eventdata\_event\_port\_connection + DATA\_Port\_connection \\ & + DATA\_access\_Port\_connection + Data\_DATA\_connection \\ & + Data\_DATA\_access\_connection + Eventdata\_DATA\_connection \\ & + Eventdata\_DATA\_access\_connection \end{aligned} \tag{17}$$

A *Port_connection* can either be a *Event_event_port_connection*, or a *Data_data_port_connection* or others (17).

- Event port, data port, event data port, data, data access → event port: port output or written data is recognized as event and quequed in the event port.
- Event data port, data port, data, data access → event data port: data output or written data is transferred and received as event data in a queuqed port.
- Data port, event data port, data, data access → data port: data output or writted data is transferred and available upon receipt as most recent value of a data port variable: the data port samples data.

2. **Abstract syntax of *Event_event_port_connection***

$$\begin{aligned} Basic\_connection ::= {} & \mathbf{opt}(connectionID) \times Source\_reference \\ & \times Connection\_direction \times Destination\_reference \\ & \times \mathbf{opt}(\mathbf{list}(Connection\_property)) \\ & \times \mathbf{opt}(In\_modes\_and\_transitions) \end{aligned} \tag{18}$$

$$\begin{aligned} Event\_event\_port\_connection ::= {} & Basic\_connection \\ & ([Source\_reference = Event\_port\_reference] \\ & [Destination\_reference = Event\_port\_reference]) \end{aligned} \tag{19}$$

$$Event\_port\_reference ::= portID \tag{20}$$

$$Connection\_direction ::= \{directional, bidirectional\} \tag{21}$$

(a) *Basic_connection* is a basic form for all categories of (port) connection (18).

(b) *Event_event_port_connection* is a *Basic_port_connection* whose *Source_reference* and *Destination_reference* are both *Event_port_reference* (19).

(c) *Event_port_reference* is a reference of event port identifier (20).

(d) *Connection_direction* could be *directional* or *bidirectional* (21). In case of a *bidirectional* port connection, both ports must be **in out** ports or a data component with *read_write* access.

(e) A *Connection_property* is a property related to port connections. The legal port connection properties are listed in next section.

(f) *In_modes_and_transitions* is defined in Mode section.

3. **Abstract syntax of other port connections** The other categories of port connection take the same form as *Basic_connection*, but with different *Source_reference* and *Destination_reference*. The following table gives acceptable source and destination reference for each category.

Connection	**Source_reference**	**Destination_reference**
Event_event_port_connection	Event_port_reference	Event_port_reference
Data_data_port_connection	Data_port_reference	Data_port_reference
Eventdata_eventdata_port_connection	Eventdata_port_reference	Eventdata_port_reference
Data_eventdata_port_connection	Data_port_reference	Eventdata_port_reference
Data_event_port_connection	Data_port_reference	Event_port_reference
Eventdata_data_port_connection	Eventdata_port_reference	Data_port_reference
Eventdata_event_port_connection	Eventdata_port_reference	Event_port_reference
DATA_Port_connection	Data_reference	Port_reference
DATA_access_Port_connection	Provides_data_access_reference	Port_reference
Data_DATA_connection	Data_port_reference	Data_reference
Data_DATA_access_connection	Data_port_reference	Requires_data_access_referenc
Eventdata_DATA_connection	Eventdata_port_reference	Data_reference
Eventdata_DATA_access_connection	Eventdata_port_reference	Requires_data_access_referenc

$Event\_port\_reference$ ::= $event\_port\_ID$

$Data\_port\_reference$ ::= $data\_port\_ID$

$Eventdata\_port\_reference$ ::= $eventdata\_port\_ID$

$Port\_reference$ ::= $port\_ID$

$Data\_reference$ ::= $dataID$

$Provides\_data\_access\_reference$ ::= $provides\_data\_access\_ID$

$Requires\_data\_access\_reference$ ::= $requires\_data\_access\_ID$

- *Data_port_reference* (*Eventdata_port_reference*) is a reference of data (event data) port identifier.
- *Port_reference* refers to a port identifier.
- *provides_data_access_ID* (*requires_data_access_ID*) is a *data_access_ID* whose related *Access_status* is *provides* (*requires*).
- The data component must have the following access right: as source, the access right must be *read-only* or *read-write*; as destination, the access right must be *write-only* or *read-write*.

12.1.3 Standard properties

This section gives some standard properties that could be applied to port connections. All the listed properties (except Transmission_Type, which is only applied to port connections) are acceptable for other connections (parameter connection, access connection, . . .).

Property	Port connection	Other categories of connection
Allowed_Connection_Binding_Class	X	X
Allowed_Connection_Binding	X	X
Actual_Connection_Binding	X	X
Required_Virtual_Bus_Class	X	X
Required_Connection_Quality_Of_Service	X	X
Connection_Pattern	X	X
Connection_Set	X	X
Transmission_Type	X	
Latency	X	X
Classifier_Matching_Rule	X	X

1. **Actual_Connection_Binding**. Connections and virtual buses are bound to the bus, virtual bus, processor, virtual processor, device and memory.

   Actual_Connection_Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, memory) **applies to** (feature, connection, thread, thread group, process, system, virtual bus);

   This property specifies through which bus a connection is transmitted.

2. **Transmission_Type**. The Transmission_Type property specifies whether the transmission across a connection is initiated by the sender (push) or by the receiver (pull). By default the transmission is initiated by the sender. A pull transmission type results in data being transmitted at the rate of the receiver. In the case of event data port or event ports, a pull transmission type

results in events or event data queued with the sender to be transmitted upon receiver request.

When associated with a connection, the property represents the transmission type the connection expects. When associated with a port, the property represents the transmission type expected by the port. When associated with a bus (or virtual bus), the property represents the transmission type that is provided by the bus or protocol.

Transmission_Type: **enumeration** (push, pull) **applies to** (data port, port connection, bus, virtual bus);

3. **Latency**. This property specifies the minimum and maximum amount of elapsed time allowed between the time the data or events enter the connection or flow and the time it exits. Its numeric value must be positive.

   Latency is a time range of transmission time?

   Latency: Time_Range **applies to** (flow, connection, bus);

   The time that data (or event) exits the connection is some time (this time must be in a time range of Latency) delayed than it enters. A time constraint is needed to model this property.

4. **Classifier_Matching_Rule**. This property specifies the rule to be applied to mach the data classifier of a connection source to the data classifier of a connection destination. Allowed rules: *Classifier_Match*, *Equivalence*, *Subset* and *Conversion*.

   Classifier_Matching_Rule: **inherit enumeration** (Classifier_Match, Equivalence, Subset, Conversion, Complement) **applies to** (connection, component implementation);

12.1.4 Standard behavior

(11) While in a given mode, transmission over a port connection only occurs if the connection is part of the current mode.

(12) During a mode switch, transmission over a data port connection only occurs at the actual time of mode switch if the port connection is declared to apply to the transition between two specific modes. The actual mode switch initiates transmission. This allows data state to be transferred between threads active in different modes. Similarly, for event or event data ports it allows for transfer of queue content.

(31) Within a synchronized system, an event arrives logically simultaneously at all ultimate connection destinations (see also Section 13.3).

12.1.5 Data port connection and Polychrony

(32) A data port connection is declared to be sampling, immediate or delayed. Yue: By Timing property. In a sampling semantic connection the recipient samples the output of the sender at dispatch time or as specified by the Input_Time property of the recipient port. In an immediate semantic connection the sender always communicates with the receiver mid-frame, i.e., in the same dispatch frame. In a delayed semantic connection the sender always communicates with the recipient phase-delayed, i.e., in the next dispatch frame of the recipient.

(33) Immediate and delayed connections only apply to semantic data connections whose end-points are both periodic. They ensure that over- and under-sampling of periodic data streams occurs deterministically. The alignment of transmission start and end times between the sending and receiving component is statically known and is not affected by preemption of thread execution and variation in actual execution time. (PLG: check consistency of this claim)

(34) A semantic data port connection is considered to be delayed if at least one of the connection declarations is declared to be delayed. A semantic data port connection is considered to be immediate if at least one of the connection declarations is declared to be immediate. Otherwise, the semantic data port connection is considered to be sampling. Typically, an immediate or delayed data connection is specified through the sibling connection declaration, i.e., the declaration at the top of the containment hierarchy of a semantic connection.

(35) For immediate data port connections data transfer only occurs when the periods of the sending and receiving component align, i.e., their dispatch occurs logically simultaneous ($T_{source} = 0 \wedge T_{destination} = 0$). The data transmission is initiated when the source component completes and enters the suspended state ($C_{source} = C_{complete,source}$). The actual execution of the receiving component is delayed until the sending thread completes execution ($C_{destinationt} = 0 \wedge C_{source} \leq C_{complete,source}$). The input is received at that time, i.e., the output time of the source data port is Completion_Time with zero range, and the input time of the receiving port is Start_Time with zero range. Note that both the source and destination must complete their execution by the deadline of the destination, i.e.,

($C_{source} = C_{complete,source} \wedge C_{source} = C_{complete,source} \wedge T_{destination} \geq Deadlinedestination$).

This rule is transitive for sequences of immediate semantic connections.

(36) For delayed data port connections data transmission is initiated at the deadline of the source component ($T_{source} = Deadlinesource$, i.e., the output time of the source data port is Deadline_Time). The input time of the receiving component port is the Dispatch_Time, i.e., the data is received at the next dispatch of the receiving component following or equal to the source deadline.

(37) For immediate and delayed connections the input time and output time cannot be explicitly declared by Input_Time and Output_Time properties.

(39) For delayed data port connections, the data transmission is initiated at the deadline of the source thread. The data is available at the destination port at the next dispatch of the destination thread that occurs at or after the source thread deadline. If the source deadline and the destination dispatch occur at the same logical time instant, the transmission is considered to occur within the same time instant.

(41) If multiple transmissions occur for a data port connection from the source thread before the dispatch of the destination thread, then only the most recently transmitted data is available in the destination port.

(42) If no transmission occurs on an in data port between two dispatches of the destination thread, then the thread receives the same data again, resulting in oversampling of the transmitted data. A status indicator is accessible to the source text of the thread as part of the port variable to determine whether the data is fresh.

(46) Deterministic communication expressed by immediate and delayed connections must be guaranteed by the method of implementation. Even if the transmission is initiated and completed by explicit send and receive service calls in the source text of the sending and receiving thread, the send and receive order of the two communicating threads must be assured.

Data port connection are restricted to 1-n.

The transmission time of a connection is specified by **Latency** property. In Figure 25, the output of a connection is sent out at *LatencyTime*, which is constraint by a time range (*Between()*). The *LatencyTime* should occur between min and max time units after the input.

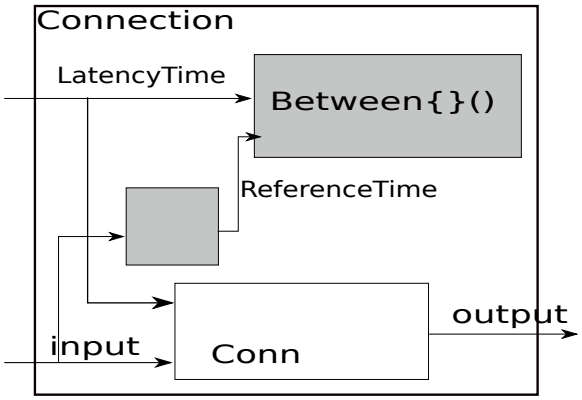


Figure 25: Connection

```
ReferenceTime := when ^input
```

```
process Conn
 (? input, LatencyTime;
  ! output; )
 (| output := input cell LatencyTime when LatencyTime |)
```

The **Output_Time** of the sender and the **Input_Time** of the the receiver are constraint by *OutEvent* of the out port and *InEvent* of the in port.

Let $x = (x_1, x_2, x_3, x_4, x_5, x_6) \in Data\_data\_port\_connection$
where $x_1 \in connectionID$
$x_2, x_4 \in Data\_port\_reference$
$x_3 = directional \in Connection\_direction$
$x_5 = Latency \in Connection\_property$
$x_6 \in In\_modes\_and\_transitions$

ConnectionTranslation(x) =
process $Connection\_x'_1$ =
$(?\ x'_2\ input;$ **event** $LatencyTime;!\ x'_4\ output)$
$(|\ output := Conn(input, LatencyTime)$
$|\ ReferenceTime :=$ **when** $\hat{}input$
$|\ x'_5$ %LatencyTime is under constraint of Latency property%
$|)$
where
event $ReferenceTime;$
$end;$
x'_1 = **IDTranslation**(x_1)
x'_2 = **DataPortReferenceTranslation**(x_2)
x'_4 = **DataPortReferenceTranslation**(x_4)
x'_5 = **PropertyTranslation**(x_5)

1. **Sampling data port connection**. The source and destination thread or device must be periodic. The output of the sender is sent out at its **Output_Time** (*OutEvent*). Only the most recently transmitted data is available

in the destination port. The received data will be sampled (*InEvent*) on the receiver side (dispatch time by default) (Figure 26.) The source *Distributer* determines the output should be sent to which receiver.

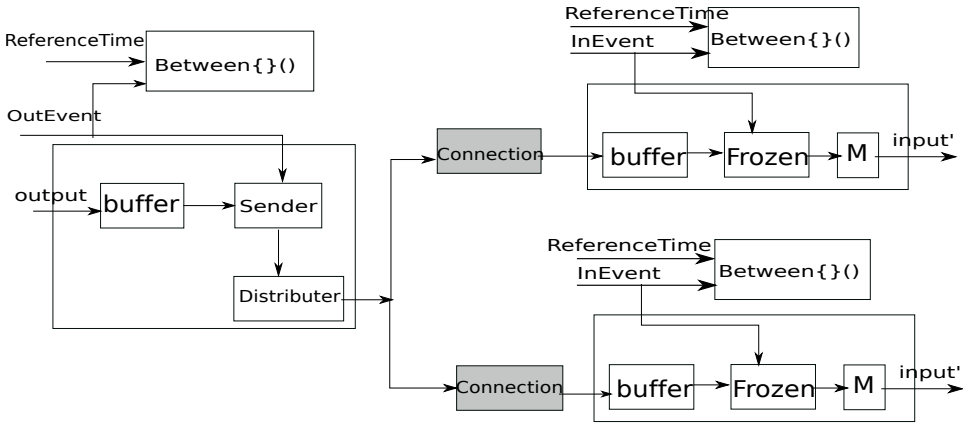


Figure 26: Sampling data port connection

2. **Immediate data port connection**. Deterministic. The sender and receiver must be both periodic. The actual execution of the receiver is delayed until the sender completes execution. The **Output_Time** of the source data port is assumed to be Completion. The **Input_Time** of the receiver port is assumed to be Start with zero offset, and any other specified time is ignored.

   $OutEvent := Completion$; $InEvent := Start$;

   The scheduler must ensure that the execution of the receiver is aligned with the completion of the sender.

3. **Delayed data port connection**. Deterministic. The sender and receiver are both periodic. The data transmission is initiated at the Deadline of the sender. The input time of the receiver is the Dispatch time (next dispatch of the receiver following the sender's deadline).

   $OutEvent := Deadline$; $InEvent := Dispatch$;

12.1.6 Event (event data) port connection and Polychrony

Event (event data) ports support n-n connectivity. The event (event data) is sent out at *OutEvent* which is under constraint of **Output_Time**. The received event (event data) is available at *InEvent*. The connection is under constraint of **Latency** time delay. (Figure 27)

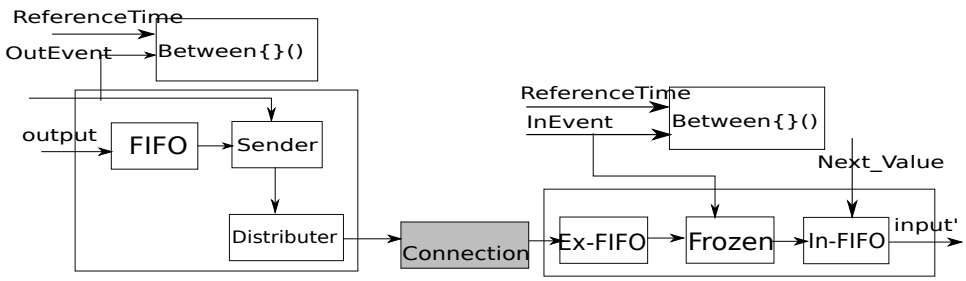


Figure 27: Event (event data) port connection

Let $x = (x_1, x_2, x_3, x_4, x_5, x_6) \in Event\_event\_port\_connection$
where $x_1 \in connectionID$
$x_2, x_4 \in Event\_port\_reference$
$x_3 = directional \in Connection\_direction$
$x_5 = Latency \in Connection\_property$
$x_6 \in In\_modes\_and\_transitions$

ConnectionTranslation(x) =
process $Connection\_x'_1$ =
$(?\ x'_2\ input;\ \textbf{event}\ LatencyTime;!\ x'_4\ output)$
$(|\ output := Conn(input, LatencyTime)$
$|\ ReferenceTime := \textbf{when}\ \hat{}input$
$|\ x'_5$ %LatencyTime is under constraint of Latency property%
$|)$
where
event $ReferenceTime$;
end;
x'_1 = **IDTranslation**(x_1)
x'_2 = **EventPortReferenceTranslation**(x_2)
x'_4 = **EventPortReferenceTranslation**(x_4)
x'_5 = **PropertyTranslation**(x_5)

bidirectional connection?

12.2 Parameter connection

(9.2-(1)) Parameter connections represent flow of data between the parameters of a sequence of subprogram calls in a thread.

Acceptable parameter connections include:

Source	Destination
call.parameter	thread port thread feature group port thread in complete feature group requires data access feature group requires data access call.parameter data subcomponent
thread port thread feature group port requires data access feature group requires data access data subcomponent	call.parameter
enclosingcall.parameter	containedcall.parameter
containedcall.parameter	enclosingcall.parameter

(9.2-(1)) ... Parameter connections may be declared from an in data or event data port or in out data or event data port of the containing thread to a subprogram call in or in out parameter. Parameter connections also ... follow the containment hierarchy of subprogram calls nested in other subprograms.

PLG: it seems that a data component (access) cannot be connected to a parameter; the syntax does not allow it.

(L3) If the parameter connection declaration represents a parameter connection between sibling components, then the source must be an out or an in out parameter and the destination must be an in or an in out parameter (PLG: parameter ↔ port forbidden). Furthermore, the source must be a parameter of a preceding subprogram call in the call sequence, and the destination must be a parameter of a succeeding subprogram call in the call sequence.

(PLG: when a subprogram call is in a subprogram are parameter ↔ port connections forbidden ?)

12.2.1 Abstract syntax of *Parameter_connection*

$$
\begin{aligned}
Parameter\_connection \ ::= \ & \mathbf{opt}(connectionID) \times Source\_parameter\_reference \\
& \times Destination\_parameter\_reference \\
& \times \mathbf{opt}(\mathbf{list}(Parameter\_connection\_property)) \\
& \times \mathbf{opt}(In\_modes\_and\_transitions)
\end{aligned}
$$

$$
Source\_parameter\_reference ::= parameterID + dataID + requires\_data\_access\_ID + portID
$$

$$
Dest\_parameter\_reference ::= parameterID + dataID + requires\_data\_access\_ID + portID
$$

12.2.2 Parameter connection and Polychrony

The parameter connection is the same as the port connection. The output is LatencyTime delayed after input, and the LatencyTime is constraint by min and max time range.

$$
\begin{aligned}
\text{Let } x \ &= \ (x_1, x_2, x_3, x_4, x_5) \in Parameter\_connection \\
\text{where } x_1 \ &\in connectionID \\
x_2, x_3 \ &\in Parameter\_reference \\
x_4 = Latency \ &\in Connection\_property \\
x_5 \ &\in In\_modes\_and\_transitions
\end{aligned}
$$

$\mathbf{ConnectionTranslation}(x)$ =
  process $Connection\_x'_1$ =
  $(?\ x'_2\ input;$ **event** $LatencyTime;!\ x'_3\ output)$
  $(|\ output$:= $Conn(input, LatencyTime)$
  $|\ ReferenceTime$:= **when** ^$input$
  $|\ x'_4$ % LatencyTime is under constraint of Latency property%
  $|)$
  where
  event $ReferenceTime;$
  $end;$
x'_1 = $\mathbf{IDTranslation}(x_1)$
x'_2 = $\mathbf{ParameterReferenceTranslation}(x_2)$
x'_3 = $\mathbf{ParameterReferenceTranslation}(x_3)$
x'_4 = $\mathbf{PropertyTranslation}(x_4)$

12.3 Feature group connection

Feature group connections represent a collection of connections between groups of features of different components.

A feature group may have a direction declared, otherwise, it is considered bidirectional.

A feature group connection must be bidirectional or be consistent with the direction of the source and destination feature.

12.3.1 Abstract syntax of *Feature_group_connection*

$Feature\_group\_connection$ ::= $\mathbf{opt}(connectionID) \times Source\_feature\_group\_reference$
  $\times\ bidirectional \times Destination\_feature\_group\_reference$
  $\times\ \mathbf{opt}(\mathbf{list}(Feature\_group\_connection\_property)) \times \mathbf{opt}(In\_modes\_and\_transitions)$
$Source\_feature\_group\_reference$ ::= $feature\_group\_ID$
$Dest\_feature\_group\_reference$ ::= $feature\_group\_ID$

12.4 Access connection

Access connections represent access to shared data components by concurrently executing threads or by subprograms executing withing thread.

9.4(2) A semantic access connection is defined by a sequence of one or more individual access connection declarations that follow the component containment hierarchy from an ultimate source to an ultimate destination.

9.4(3) In the case of bidirectional connections, either the subcomponent being accessed or the feature that requires access can be the ultimate source or destination. In the case of directional data access connections, the ultimate source is the source of the data flow .... In the case of partial AADL models, the ultimate source or destination may be a provides access feature of a component without subcomponents.

(9.3(L1)) The category of the source and the destination of a access connection declaration must be the same...

(9.3(L2)) The ultimate source of a semantic access connection must be data, subprogram, subprogram group, or bus subcomponent (or their respective access feature.)

(9.3(L3))The ultimate destination of a semantic data access connection must be a requires data access feature of a thread or a subprogram call that requires the same data access.

(9.3(7)) Access connections are restricted to 1-n connectivity...

12.4.1 Abstract syntax of *Access_connection*

9.4(L10) The category of the access connection source and destination must be identical. If the component category is specified as part of the connection declaration, then it must be identical to that of the source and destination. Bus, data, subprogram and subprogram group are acceptable categories.

$$Access\_connection ::= Bus\_access\_connection + Subprogram\_access\_connection + Subprogram\_group\_access\_connection + Data\_access\_connection$$

1. **Bus_access_connection**

   A bus access connection represents communication between processors, memory and devices by accessing a shared bus.

   (1) ... Bus access is used to model connectivity of execution platform components through buses.

$Bus\_access\_connection$ ::= $\mathbf{opt}(connectionID) \times Bus\_access\_provider\_reference$
$\times Connection\_direction \times Bus\_access\_requirer\_reference$
$\times \mathbf{opt}(\mathbf{list}(Bus\_access\_connection\_property)) \times \mathbf{opt}(In\_modes\_and\_transitions)$
$Bus\_access\_provider\_reference$ ::= $provides\_bus\_access\_ID + busID$
$Bus\_access\_requirer\_reference$ ::= $requires\_bus\_access\_ID + busID$

2. **Subprogram_access_connection**

A subprogram access or subprogram group access connection represents access to subprogram code or a library that calls can be made to.

$Subprogram\_access\_connection$ ::= $\mathbf{opt}(connectionID)$
$\times Subprogram\_access\_provider\_reference \times Connection\_direction$
$\times Subprogram\_access\_requirer\_reference$
$\times \mathbf{opt}(\mathbf{list}(Subprogram\_access\_connection\_property))$
$\times \mathbf{opt}(In\_modes\_and\_transitions)$
$Subprogram\_access\_provider\_reference$ ::= $subprogramID$
$+ provides\_subprogram\_access\_ID$
$Subprogram\_access\_requirer\_reference$ ::= $subprogramID$
$+ requires\_subprogram\_access\_ID$

3. **Subprogram_group_access_connection**

$Subprogram\_group\_access\_connection$ ::= $\mathbf{opt}(connectionID)$
$\times Subprogram\_group\_access\_provider\_reference \times Connection\_direction$
$\times Subprogram\_group\_access\_requirer\_reference$
$\times \mathbf{opt}(\mathbf{list}(Subprogram\_group\_access\_connection\_property))$
$\times \mathbf{opt}(In\_modes\_and\_transitions)$
$Subprogram\_group\_access\_provider\_reference$ ::= $subprogram\_group\_ID$
$+ provides\_subprogram\_group\_access\_ID$
$Subprogram\_group\_access\_requirer\_reference$ ::= $subprogram\_group\_ID$
$+ requires\_subprogram\_group\_access\_ID$

4. **Data_access_connection**

   9.4(6) A data access connection represents access to a shared data component by concurrently executing threads or by subprograms executing within thread.

$$Data\_access\_connection \ ::= \ \mathbf{opt}(connectionID) \times Data\_access\_provider\_reference \times Connection\_direction \times Data\_access\_requirer\_reference \times \mathbf{opt}(\mathbf{list}(Data\_access\_connection\_property)) \times \mathbf{opt}(In\_modes\_and\_transitions)$$

$$Data\_access\_provider\_reference \ ::= \ provides\_data\_access\_ID + dataID$$

$$Data\_access\_requirer\_reference \ ::= \ requires\_data\_access\_ID + dataID$$

13 Flows

(1) A flow is a logical flow of data and control through a sequence of threads, processors, devices, and port connections or data access connections. A component can have a flow specification, which specifies whether a component is a flow source, i.e., the flow starts within the component, a flow sink, i.e., the flow ends within the component, or there exists a flow path through the component, i.e., from one of its incoming ports to one of its outgoing ports.

A flow is a logical flow of data and control through a sequence of threads, processors, devices and port connections or data access connections.

13.1 Abstract syntax

Flows are represented by flow specification, flow implementation and end-to-end flow declarations.

$$Flow \ ::= \ Flow\_spec + Flow\_implementation + End\_to\_end\_flow$$

Flow_spec

$$Flow\_spec ::= Flow\_source + Flow\_sink + Flow\_path \tag{22}$$

$$Flow\_end ::= Terminal \times flowID \times flow\_feature\_ID \times \mathbf{opt}(\mathbf{list}(Flow\_property)) \times \mathbf{opt}(In\_modes) \tag{23}$$

$$Terminal ::= \{source, sink\} \tag{24}$$

$$Flow\_source ::= Flow\_end([Terminal = source]) \tag{25}$$

$$Flow\_sink ::= Flow\_end([Terminal = sink]) \tag{26}$$

$$Flow\_path ::= flowID \times in\_flow\_feature\_ID \times out\_flow\_feature\_ID \times \mathbf{opt}(\mathbf{list}(Flow\_property)) \times \mathbf{opt}(In\_modes) \tag{27}$$

1. *Flow_source (Flow_sink)* is a *Flow_end* with its *Terminal* equals to *source (sink)*.
2. *Terminal* defines whether a flow specification is a flow source or a flow sink.
3. *flow_feature_ID* could be a *feature_ID*.

Flow_implementation

$$Flow\_implementation ::= Flow\_source\_implementation + Flow\_sink\_implementation + Flow\_path\_implementation \tag{28}$$

$$Flow\_source\_implementation ::= flowID \times \mathbf{opt}(\mathbf{list}(flowID \times connectionID)) \times out\_flow\_feature\_ID \times \mathbf{opt}(\mathbf{list}(Flow\_property)) \times \mathbf{opt}(In\_modes\_and\_transitions) \tag{29}$$

$$Flow\_sink\_implementation ::= flowID \times in\_flow\_feature\_ID \times \mathbf{opt}(\mathbf{list}(connectionID \times flowID)) \times \mathbf{opt}(\mathbf{list}(Flow\_property)) \times \mathbf{opt}(In\_modes\_and\_transitions) \tag{30}$$

$$Flow\_path\_implementation ::= flowID \times in\_flow\_feature\_ID \times \mathbf{opt}(\mathbf{list}(connectionID \times flowID)) \times out\_flow\_feature\_ID \times \mathbf{opt}(\mathbf{list}(Flow\_property)) \times \mathbf{opt}(In\_modes\_and\_transitions) \tag{31}$$

End_to_end_flow

$$End\_to\_end\_flow \ ::= \ flowID \times start\_flow\_ID \times \mathbf{opt}(\mathbf{list}(connectionID \times flowID)) \times connectionID \times end\_flow\_ID \times \mathbf{opt}(\mathbf{list}(Flow\_property)) \times \mathbf{opt}(In\_modes\_and\_transitions) \quad (32)$$

$$start\_flow\_ID \ ::= \ flowID \quad (33)$$

$$end\_flow\_ID \ ::= \ flowID \quad (34)$$

13.2 Standard properties

Actual_Latency Latency: Time_Range

13.3 Flows and Polychrony

(2) The purpose of providing the capability of specifying end-to-end flows is to support various forms of flow analysis, such as end-to-end timing and latency, reliability, numerical error propagation, Quality of Service (QoS) and resource management based on operational flows.

This purpose does not require specific Signal features. Flows properties can be represented in comments if necessary.

14 Properties

14.1 Abstract syntax

(1) A property provides information about component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls. A property has a name, a type, and a value. The property definition declares a name for a given property along with the AADL components and functionality to which the property applies. The property type specifies the set of acceptable values for a property. Each property has a value or list of values that is associated with the named property in a given specification.

(2) A property set contains declarations of property types and property definitions that may appear in an AADL specification. The two predeclared property sets in this standard define properties and property types that are applicable to all AADL specifications. Users may define property sets that are unique to their model, project or toolset. The properties and property types that are declared in user-defined property sets are accessed using their qualified name. A property definition declaration within a property set indicates the component types, component

implementations, subcomponents, features, connections, flows, modes, and subprogram calls, for which this property applies.

(3) Properties can have associated expressions that are statically typed, and evaluate to a specific value. The time at which a property expression is evaluated may depend on the property and on how a specification is processed. For example, some expressions may be evaluated immediately, some after binding decisions have been made, and some reflect runtime state information, e.g., the current mode. During analysis, all property expressions can be evaluated to known values, if necessary, by considering all possible runtime states. A given property definition may have a default expression.

PLG: look deeper to clearly understand inheritance of time properties.

1. **Property_set**

$$Property\_set \ ::= \ property\_set\_ID \times \mathbf{opt}(\mathbf{list}(Property\_type\_declaration)) \times \mathbf{opt}(\mathbf{list}(Property\_definition\_declaration)) \times \mathbf{opt}(\mathbf{list}(Property\_constant))$$

2. **Property_type_declatarion**

$Property\_type\_declaration$ ::= $property\_type\_ID \times Property\_type$
$Property\_type$ ::= $aadlboolean + aadlstring + Enumeration\_type$
$+ Units\_type + Number\_type + Range\_type$
$+ Classifier\_type + Reference\_type + Record\_type$
$Enumeration\_type$ ::= $\mathbf{list}(enumeration\_literal\_ID)$
$Units\_type$ ::= $Units\_list$
$Units\_list$ ::= $unitID \times \mathbf{opt}(\mathbf{list}(unitID \times numeric\_literal))$
$Number\_type$ ::= $Real + Integer$
$Real$ ::= $aadlreal \times \mathbf{opt}(Real\_range) \times \mathbf{opt}(Units\_designator)$
$Integer$ ::= $aadlinteger \times \mathbf{opt}(Integer\_range) \times \mathbf{opt}(Units\_designator)$
$Units\_designator$ ::= $units\_property\_type\_ID + Units\_list$
$Real\_range$ ::= $Real\_bound \times Real\_bound$
$Real\_bound$ ::= $real\_literal + constant$
$Integer\_range$ ::= $Integer\_bound \times Integer\_bound$
$Integer\_bound$ ::= $integer\_literalORconstant$
$Range\_type$ ::= $Number\_type + number\_property\_type\_ID$
$Classifier\_type$ ::= $\mathbf{list}(Classifier\_category\_reference)$
$Reference\_type$ ::= $\mathbf{list}(Reference\_category)$
$Record\_type$ ::= $\mathbf{list}(Record\_field)$
$Record\_field$ ::= $fieldID \times Property\_type\_designator$
$Valued\_property$ ::= $Single\_valued\_property + Multi\_valued\_property$

3. **Property_definition_declaration**

$Property\_definition\_declaration$ ::= $property\_name \times Valued\_property$
$\times \mathbf{list}(Property\_owner)$

4. **Property_constant**

$$
\begin{aligned}
&Property\_constant ::= Single\_valued\_property\_constant \\
&\quad + Multi\_valued\_property\_constant \\
&Single\_valued\_property ::= Property\_type\_designator \\
&\quad \times \mathbf{opt}(Default\_property\_expression) \\
&Multi\_valued\_property ::= \mathbf{list}(Property\_type\_designator) \\
&\quad \times \mathbf{list}(Default\_property\_expression) \\
&Single\_valued\_property\_constant ::= property\_constant\_ID \\
&\quad \times Property\_type\_designator \times Constant\_property\_expression \\
&Multi\_valued\_property\_constant ::= property\_constant\_ID \\
&\quad \times Property\_type\_designator \times \mathbf{list}(Constant\_property\_expression)
\end{aligned}
$$

5. **Property_expression**

$Property\_expression$ ::= $Boolean\_term + Real\_term + Integer\_term$
$+ String\_term + Enumeration\_term + Unit\_term + Real\_range\_term$
$+ Integer\_range\_term + Property\_term + Component\_classifier\_term$
$+ Reference\_term + Record\_term + Computed\_term$
$Boolean\_term$ ::= $boolean\_value + NOT\_boolean\_term + AND\_boolean\_term$
$+ OR\_boolean\_term$
$boolean\_value$ ::= $\{true, false\}$
$NOT\_boolean\_term$ ::= $NOT \times Boolean\_term$
NOT ::= $\{not\}$
$AND\_boolean\_term$ ::= $Boolean\_term \times AND \times Boolean\_term$
AND ::= $\{and\}$
$OR\_boolean\_term$ ::= $Boolean\_term \times OR \times Boolean\_term$
OR ::= $\{or\}$
$Real\_term$ ::= $real\_literal + constant$
$Integer\_term$ ::= $integer\_literal + constant$
$String\_term$ ::= $string\_literal + string\_property\_constant\_term$
$Enumeration\_term$ ::= $enumerationID$
$+ enumeration\_property\_constant\_term$
$Unit\_term$ ::= $unitID + unit\_property\_constant\_term$
$Real\_range\_term$ ::= $Real\_term \times Real\_term \times \mathbf{opt}(Real\_term)$
$Integer\_range\_term$ ::= $Integer\_term \times Integer\_term \times \mathbf{opt}(Integer\_term)$
$Property\_term$ ::= $property\_name$
$Component\_classifier\_term$ ::= $Component\_type\_reference$
$+ Component\_implementation\_reference$
$Reference\_term$ ::= $contained\_model\_element\_path$
$Record\_term$ ::= $\mathbf{list}(record\_field\_ID \times property\_value)$
$Computed\_term$ ::= $functionID$

14.2 Build in property types

1. **Property types**

- aadlboolean,

- aadlstring
- enumeration_type
- units_type
- number_type
- range_type
- classifier_type
- reference_type
- record_type

2. **Number types**
   - aadlinteger [integer_range] [units units_designator]
   - aadlreal [real_range] [units units_designator]

14.3 Scheduling features

- The Data_Volume property type specifies a property type for the volume of data per time unit. The predeclared unit literals are expressed in terms of seconds as time unit. The numeric value of the property must be positive.

  Note: Conversion factor of 1000 consistent with ISO.

  ```
  Data_Volume: type aadlinteger 0 bitsps .. value(Max_Aadlinteger)
      units ( bitsps, Bytesps ⇒ bitsps * 8,
      Kbytesps ⇒ Bytesps ∗ 1000,
      Mbytesps ⇒ Kbytesps ∗ 1000,
      Gbytesps ⇒ Mbytesps ∗ 1000 );
  ```

- The Throughput property specifies the maximum volume of data transferred per time unit. Its numeric value must be positive.

  ```
  Throughput: Data_Volume applies to (flow, connections);
  ```

- The Time property type specifies a property type for time that is expressed as numbers with predefined time units. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

  ```
  Time: type aadlinteger 0 ps .. value(Max_Time) units Time_Units;
  ```

- The Tim_Range property type specifies a property type for a closed range of time, i.e., a time span including the lower and upper bound. The property type is Time.

  ```
  Time_Range: type range of Time;
  ```

14.4 Property and Polychrony

1. **Input_Time**

$$
\begin{aligned}
Input\_Time &::= \mathbf{list}(IO\_Time\_Spec) \\
IO\_Time\_Spec &::= TimeRange \times IO\_Reference\_Time \\
TimeRange &::= Time \times Time
\end{aligned}
$$

(a) If **Input_Time** contains only one value of **IO_Time_Spec**:

$$
\begin{aligned}
\text{Let } x &= (\{x_1\}) \in Input\_Time \\
\text{where } x_1 &\in IO\_Reference\_Time
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{PropertyTranslation}(x) &= x_1' \\
x_1' &= \mathbf{PropertyTranslation}(x_1)
\end{aligned}
$$

(b) If **Input_Time** contains more than one values (two for example).

$$
\begin{aligned}
\text{Let } x &= (\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}) \in Input\_Time \\
\text{where } x_1,\ x_2,\ x_4,\ x_5 &\in Time \\
x_3,\ x_6 &\in IO\_Reference\_Time
\end{aligned}
$$

PropertyTranslation(x) =
process $SeveralBetween$ =
{[n]**struct** (**integer** min_offset; **integer** max_offset);}
(? **event** $InEvent$;
[n]**event** $ReferenceTime$;
[n]**struct** (**event** $unit1$; **event** $unit2$);)
(| $event1\_1$:= **when** (($unit1$[1] **after** $ReferenceTime$[1]) = min_offset[1])
| $event1\_2$:= **when** (($unit2$[1] **after** $ReferenceTime$[1]) = max_offset[1])
| $event2\_1$:= **when** (($unit1$[2] **after** $ReferenceTime$[2]) = min_offset[2])
| $event2\_2$:= **when** (($unit2$[2] **after** $ReferenceTime$[2]) = max_offset[2])
| ...%InEvent is between event1_1 and event1_2%
| ...%or between event2_1 and event2_2%
|)
where
event $event1\_1$, $event1\_2$, $event2\_1$, $event2\_2$;
end;
n = $Count(V(x))$
(min_offset[1], $unit1$[1]) = **PropertyTranslation**(x_1)
(max_offset[1], $unit2$[1]) = **PropertyTranslation**(x_2)
(min_offset[2], $unit1$[2]) = **PropertyTranslation**(x_4)
(max_offset[2], $unit2$[2]) = **PropertyTranslation**(x_5)
$ReferenceTime$[1] = **PropertyTranslation**(x_3)
$ReferenceTime$[2] = **PropertyTranslation**(x_6)

2. **IO_Time_Spec**

IO_Time_Spec ::= $TimeRange \times IO\_Reference\_Time$
$TimeRange$ ::= $Time \times Time$

Let x = $(\{x_1, x_2, x_3\}) \in Input\_Time$
where $x_1,\ x_2 \in Time$
$x_3 \in IO\_Reference\_Time$

PropertyTranslation(x) =
process $Between$ =
$\{$**integer** $min\_offset, max\_offset;\}$
$(?$ **event** $Controlled\_Time,\ ReferenceTime,\ unit1,\ unit2;)$
$(|\ event1$:= **when** $((unit1$ **after** $ReferenceTime)$ = $min\_offset)$
$|\ event2$:= **when** $((unit2$ **after** $ReferenceTime)$ = $max\_offset)$
$|$ ...%Controlled_Time is between event1 and event2%
$|)$
where
event $event1,\ event2;$
end;
$(min\_offset,\ unit1)$ = **PropertyTranslation**(x_1)
$(max\_offset,\ unit2)$ = **PropertyTranslation**(x_2)
$ReferenceTime$ = **PropertyTranslation**(x_3)

3. **Output_Time**

$Output\_Time$ ::= **list**(IO_Time_Spec)
IO_Time_Spec ::= $TimeRange \times IO\_Reference\_Time$

(a) If **Output_Time** contains only one value of **IO_Time_Spec**:

Let x = $(\{x_1\}) \in Output\_Time$
where $x_1 \in IO\_Reference\_Time$

PropertyTranslation(x) = x_1'
x_1' = **PropertyTranslation**(x_1)

(b) If **Output_Time** contains more than one values (two for example).

$$\begin{aligned} &\text{Let } x = (\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}) \in Output\_Time \\ &\text{where } x_1,\ x_2,\ x_4,\ x_5 \in Time \\ &\qquad x_3,\ x_6 \in IO\_Reference\_Time \end{aligned}$$

$\mathbf{PropertyTranslation}(x) =$
  process $SeveralBetween =$
  $\{[n]$**struct** (**integer** min_offset; **integer** max_offset); $\}$
  (? **event** $OutEvent$;
  $[n]$**event** $ReferenceTime$;
  $[n]$**struct** (**event** $unit1$; **event** $unit2$);)
  (| $event1\_1$:= **when** ($(unit1[1]$ **after** $ReferenceTime[1])$ = $min\_offset[1]$)
  | $event1\_2$:= **when** ($(unit2[1]$ **after** $ReferenceTime[1])$ = $max\_offset[1]$)
  | $event2\_1$:= **when** ($(unit1[2]$ **after** $ReferenceTime[2])$ = $min\_offset[2]$)
  | $event2\_2$:= **when** ($(unit2[2]$ **after** $ReferenceTime[2])$ = $max\_offset[2]$)
  | ...%OutEvent is between event1_1 and event1_2%
  | ...%or between event2_1 and event2_2%
  |)
  where
  event $event1\_1$, $event1\_2$, $event2\_1$, $event2\_2$;
  end;
$n = Count(V(x))$
$(min\_offset[1],\ unit1[1]) =$ **PropertyTranslation**(x_1)
$(max\_offset[1],\ unit2[1]) =$ **PropertyTranslation**(x_2)
$(min\_offset[2],\ unit1[2]) =$ **PropertyTranslation**(x_4)
$(max\_offset[2],\ unit2[2]) =$ **PropertyTranslation**(x_5)
$ReferenceTime[1] =$ **PropertyTranslation**(x_3)
$ReferenceTime[2] =$ **PropertyTranslation**(x_6)

4. **Access_Time**

$$Access\_Time ::= First \times Last$$
$$First ::= IO\_Time\_Spec$$
$$Last ::= IO\_Time\_Spec$$

$$\text{Let } x = (x_1, x_2) \in Access\_Time$$
$$\text{where } x_1, x_2 \in IO\_Time\_Spec$$

$$\mathbf{PropertyTranslation}(x) = (x_1', x_2')$$
$$x_1' = \mathbf{PropertyTranslation}(x_1)$$
$$x_2' = \mathbf{PropertyTranslation}(x_2)$$

5. **Time**

$$Time ::= aadlinteger \times Time\_Unit$$

$$\text{Let } x = (x_1, x_2) \in Time$$
$$\text{where } x_1 \in integer$$
$$x_2 \in Time\_Unit$$

$$\mathbf{PropertyTranslation}(x) = (x_1', x_2')$$
$$x_1' = \mathbf{PropertyTranslation}(x_1) = V(x_1)$$
$$x_2' = \mathbf{PropertyTranslation}(x_2)$$

6. **Time_Unit**

$$Time\_Unit ::= \{ps, ns, us, ms, sec, min, hr\}$$

Let $x \in Time\_Unit$

$$\textbf{PropertyTranslation}(x) \ = \ \textbf{event}\ V(x)$$
$$\begin{aligned}\text{where } ns\,\hat{}\, &= \textbf{when}\ ((ps\ \textbf{after}\ ns) = 1000)\\ us\,\hat{}\, &= \textbf{when}\ ((ps\ \textbf{after}\ us) = 1000)\\ ms\,\hat{}\, &= \textbf{when}\ ((us\ \textbf{after}\ ms) = 1000)\\ sec\,\hat{}\, &= \textbf{when}\ ((ms\ \textbf{after}\ sec) = 60)\\ min\,\hat{}\, &= \textbf{when}\ ((sec\ \textbf{after}\ min) = 60)\\ hr\,\hat{}\, &= \textbf{when}\ ((min\ \textbf{after}\ hr) = 60)\end{aligned}$$

7. **IO_Reference_Time**

   $IO\_Reference\_Time$ ::= $\{Dispatch, Start, Completion, Deadline, NoIO\}$

   Let $x \in IO\_Reference\_Time$

$$\textbf{PropertyTranslation}(x) \ = \ \textbf{event}\ V(x)$$

8. **Queue_Size**

   $Queue\_Size$:= $aadlinteger$

   Let $x \in Queue\_Size$, where $x \in aadlinteger$

   $\textbf{PropertyTranslation}(x) \ = \ V(x)$

9. **Dequeue_Items**

   $Dequeue\_Items$:= $aadlinteger$

   Let $x \in Dequeue\_Items$, where $x \in aadlinteger$

   $\textbf{PropertyTranslation}(x) \ = \ V(x)$

15 Modes

(13)The modes subclause declares a state machine describing the dynamic mode switching behavior of modes. The states of the state machine represent the different modes and the transitions specify the event(s) that can trigger a mode switch to the destination mode. Only one mode alternative represents the current mode at any one time.

(1) A mode represents an operational mode state, which manifests itself as a configuration of contained components, connections, and mode-specific property value associations . . .

(2) Mode transitions . . . are triggered by events . . .

15.1 Mode declaration

(L1) A mode or mode transition can be declared in any of the component categories.

(L3) The set of transitions declared within a single component implementation must define a deterministic transition function. For each mode, there must exist exactly (PLG: at most ??? see item13) one transition, which can cause transition to another mode. Unless logical conditions are defined for mode switches, an event port can only be named in one outgoing transition from the same mode.

A mode represents an operational mode state. Mode transitions model dynamic operational behavior that represents switching between configurations and changes in components internal characteristics.

15.1.1 Abstract syntax

$$Modes ::= Mode + Mode\_transition \tag{35}$$

$$Mode ::= ModeID \times \mathbf{opt}(\mathbf{list}(Mode\_property)) \tag{36}$$

$$\begin{aligned} Mode\_transition ::= {} & \mathbf{opt}(Mode\_transition\_ID) \times source\_mode\_ID \\ & \times \mathbf{list}(Mode\_transition\_trigger) \times destination\_mode\_ID \\ & \times \mathbf{opt}(\mathbf{list}(Mode\_transition\_property)) \end{aligned} \tag{37}$$

$$Mode\_transition\_trigger ::= portID \tag{38}$$

$$In\_modes ::= \mathbf{list}(modeID) \tag{39}$$

$$In\_modes\_and\_transitions ::= \mathbf{list}(Mode\_or\_transition) \tag{40}$$

$$Mode\_or\_transition ::= modeID + Mode\_transition\_ID \tag{41}$$

(L2) If a component classifier contains mode declarations, one of those modes must be declared with the reserved word initial. If the component classifier extends another component classifier, the initial mode may have been declared in one of the ancestor component classifier.

(L4) The unique port identifier must be either an in or in out event port identifier in the namespace of the associated component type or an out or in out event port in the namespace of the component type associated with the named subcomponent.

15.1.2 Standard properties

1. Mode properties:
2. Mode transition properties:

   Mode_Transition_Response

15.2 Model life

(10) The in modes statement is declared as part of subcomponent declarations, subprogram call sequences, flow implementations, and property associations. It specifies the modes for which these declarations and property values hold. The mode identifiers refer to mode declarations in the modes subclause of the component classifier.

(11) The in modes statement declared as part of connection declarations specify the modes or mode transitions for which these connection declarations hold. The mode identifiers refer to mode declarations in the modes subclause of the component implementation. If a connection is declared to be part of a mode transition, then the content of the ultimate source port is transferred to the ultimate destination port at the actual mode switch time. If the in modes statement contains only mode transitions, then the connection is part of the specified mode transitions, but not part of any particular mode....

(from 10-11) If the in modes statement is not present, then the subcomponent, subprogram call sequence, flow implementation, property association or connection is part of all modes.

(from 10-11) If a property association (a connection) has both mode-specific declarations and a declaration without an in modes statement, then the declaration without the in modes statement applies to those modes not covered by the mode-specific declarations.

15.3 Mode behavior

(3) The mode semantics described here focus on a single mode subclause. A system instance that represents the runtime architecture of an operational system can contain multiple components with their own mode transitions. The semantics of system-wide mode switching are discussed in Section 13.3

(5) A mode may represent a runtime configuration of systems, processes, thread groups and threads and their connections for a given operational state. In this case the modes are declared in thread groups, processes and systems, and in modes clauses indicate which subcomponents and connections are active in a given

mode. In this case, only the threads that are part of the current mode are in the suspended awaiting dispatch state responding to dispatch requests. All other threads are in the suspended awaiting mode state or thread terminated state.

(9) A component type or component implementation may contain several declared modes. Exactly one of those modes is the current mode. Initially, the initial mode is the current mode. On mode activation the Activation_Mode property (PLG: p.251: applies to thread) determines whether the initial mode is entered or the mode from the last deactivation is resumed.

(13) ... A mode switch is triggered when an event arrives at an event port that is named in one of the transitions out of the state representing the current mode. If an event is raised and there is no transition out of the current mode naming the event port through which the event arrives, the event is ignored. If several events occur logically simultaneously and affect different mode transitions out of the current mode, the order of arrival for the purpose of determining the mode transition is implementation dependent. If an Urgency property is associated with each port named in mode transitions, then the mode transition with the highest port urgency takes precedence. If several ports have the same urgency then the mode transition is chosen non-deterministically. (PLG: why not implementation dependent as above ?)

15.3.1 Mode switch within a thread

(15) A mode switch within a thread may logically occur at dispatch time. An external event through an incoming event port, or an event raised within the thread or will cause the thread to enter the new mode at the next dispatch. Such an event raised within a thread is declared as self.eventname, or by a subprogram call with an outgoing event port including a call to the Send (deprecated Raise_Event) service call, and implemented as a service call to Send (deprecated Raise_Event) in the application source text or runtime system.

(16) A mode switch within a thread results in a change of its current mode. The effect is a change in the subprogram call sequence and mode-specific property values to reflect a change in source text internal execution behavior...

(17) Similarly, mode switches within an execution platform component occur as a result of external or internal events. A mode switch within a thread or execution platform component does not affect the set of active threads, processors, devices, buses, or memories, nor does it affect the set of active connections.

15.3.2 Mode switch within set of threads

(18) A mode switch within a system, process, or thread group implementation

has the effect of deactivating and activating threads to respond to dispatches, and changing the pattern of connections between components. Deactivated threads transition to the suspended awaiting mode state. Background threads that are not part of the new mode suspend performing their execution. Activated threads transition to the suspended awaiting dispatch state and start responding to dispatches. Suspended background threads that are part of the new mode resume performing execution once the transition into the new mode is complete. Threads that are part of both the old and new mode of a mode transition continue to respond to dispatches and perform execution. Ports that were connected in the old mode, may not be connected in the new mode and vice versa.

(19) When a mode switch is requested through the arrival of an event on a mode transition it may result in activation or deactivation of threads and connections, or in the change of a threads period, deadline, dispatch protocol, or execution time. In this case the actual mode switch occurs immediately if no periodic threads are part of the old mode, otherwise it occurs once these periodic threads in the old mode are synchronized at their hyperperiod. Only those threads with a Synchronzied_Component property value of true are considered in the determination of the hyperperiod.

(20) Starting with the actual time of mode switch, the component is in a mode transition in progress state for a limited amount of time. During this time some threads are deactivated, other threads are activated, connections are adjusted, and the active threads in the new mode start to execute. This time period takes the Synchronized_Component property into account and is determined at the level of the whole system instance (see Section 13.3). After that period of time, the component is considered to operate in the new mode.

(21) At the time of the actual mode switch, the deactivate entrypoint is invoked for the following threads that must be deactivated: periodic threads that are synchronized with the mode switch; aperiodic or sporadic threads that are in the suspended awaiting dispatch state...

(25) At the time of the actual mode switch, any threads that were inactive in the old mode and are active in the new mode execute their activate entrypoint. In the case of periodic threads, this is immediately followed by their first dispatch of the compute entrypoint. (TG: does it mean they don't go through the suspended awaiting dispatch state?)

In the case of background threads, the thread resumes execution from where it was suspended at the last deactivation. (TG: if it is at the time of the actual mode switch, is it compatible with (18) where it is said once the transition into the new mode is complete?)

(24) Background processes that are only part of the old mode are suspended when the actual mode switch occurs.

(27) Some property values for a component or its subcomponents may be mode-specific, for example the period of a periodically dispatched thread may be different in different modes of operation. It changes at the time of actual mode switch.

15.3.3 Mode switch for thread that are not synchronized

(22) At the time instant of actual mode switch, aperiodic and sporadic threads as well as periodic threads not synchronized with the mode switch may still be in the perform computation state. The Active_Thread_Handling_Protocol property specifies for each such thread what action is to be taken at mode switch. Possible actions are:

- Abort the execution of the thread and permit the thread to recover any state through execution of its recover entrypoint. This permits the thread to recover to a consistent state for future activation and dispatch. Upon completion of the recover entrypoint, execution the thread enters the suspended awaiting mode state; event and event data port queues of the thread are flushed by default or remain in the queue until the thread is activated again as specified by the Active_Thread_Queue_Handling_Protocol property. If the thread was executing a remotely called subprogram, the current dispatch execution of the calling thread of a call in progress or queued call is also aborted.
- Permit the thread to complete the execution of its current dispatch. Any remaining queued events, or event data may be flushed by default, or remain in the queue until the thread is activated again as specified by the Active_Thread_Queue_Handling_Protocol property.
- Permit the thread to finish processing all events or event data in its queues.

  (TG: does it possibly include new ones?)

16 An AADL abstract syntax

This section is a copy/paste from MyDigest.

We describe the AADL language using the abstract syntax trees defined in this section.

16.1 Notations

16.1.1 General AST

1. **Tree set** Given a set of labels Λ that contains a special empty label ε, the set of trees labeled in Λ is the smallest set that satisfies the following rules :
   - $\bullet$ is in ; it denotes the tree that only contains an unlabeled root;ts label is ε; by definition $\bigcirc = \{\bullet\}$;
   - if $t_1, \ldots, t_n$ are trees then for all λ in Λ $(\lambda, (t_1, \ldots, t_n))$ is a tree labeled λ
2. **Tree sets** For all subsets of trees $SS, SS_1, \ldots, SS_n$,
   - $SS_1 + SS_2$ denotes the set of trees $SS_1 \cup SS_2$
   - $[SS_1] = SS_1 + \bigcirc$
   - $SS_1 \times SS_2 \times \cdots \times SS_n$ is the set of n_tuples of trees; $\times$ is defined as associative (ie $(t_1, (t_2, t_3))$, $((t_1, t_2), t_3))$, (t_1, t_2, t_3) are not distinguished)
   - SS_{1+} is the smallest set of non empty sequences of trees defined by: $SS_{1+} = SS_1 + (SS_1 \times SS_{1+})$
   - $SS_{1*} = [SS_{1+}]$
   - for all λ in Λ, $\lambda : SS$ is the set of trees $(\lambda, (t_1, \ldots, t_n))$ such that $(t_1, \ldots, t_n)$ is a n-tuple in SS, completed by $(\lambda, \bullet)$ when $\bigcirc$ is in SS.
3. **Tree set variables** For an identifier X
   - $X = SS_1$ is the definition of X that associates to X the set of trees SS_1
   - each occurrence of a variable in a tree set expression denotes the set of trees that is associated to X by its definition.

16.1.2 AADL AST

A_SSS is a set of AADL abstract syntax trees;
The set of labels contain the component categories.

16.2 Lexical elements

$$none\_statement ::= \textbf{none}$$

16.2.1 Word characters

$letter\_or\_digit$ ::= $identifier\_letter$ + $digit$
$identifier\_letter$ ::= $upper\_case\_identifier\_letter$ + $lower\_case\_identifier\_letter$

upper_case_identifier_letter: Any character of Row 00 of ISO 10646 BMP whose name begins Latin Capital Letter.
lower_case_identifier_letter: Any character of Row 00 of ISO 10646 BMP whose name begins Latin Small Letter.
Digit: One of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

16.2.2 Other characters

space_character: The character of ISO 10646 BMP named Space".
special_character: Any character of the ISO 10646 BMP that is not reserved for a control function, and is not the space_character, an identifier_letter, or a digit.
format_effector: The control functions of ISO 6429 called character tabulation (HT), line tabulation (VT), carriage return (CR), line feed (LF), and form feed (FF).
other_control_function: Any control function, other than a format_effector, that is allowed in a comment; the set of other_control_functions allowed in comments is implementation defined.

16.2.3 Decimal literals

$$decimal\_integer\_literal ::= numeral[positive\_exponent]$$
$$decimal\_real\_literal ::= numeral.numeral[exponent]$$
$$numeral ::= digit\{[underline]digit\}^*$$
$$exponent ::= E[+]numeral + E - numeral$$
$$positive\_exponent ::= E[+]numeral$$

16.2.4 Based literals

16.2.5 String literals

16.2.6 Comments

16.2.7 Identifiers

A_IDENT is the set of identifiers defined by

$$identifier ::= identifier\_letter\{[underline]letter\_or\_digit\}^*$$
$$A\_NAME ::= A\_IDENT \times A\_IDENT$$
$$A\_LNAME ::= \mathbf{list}(A\_IDENT) \times A\_NAME$$

$package\_name ::= \{package\_identifier ::\}^* package\_identifier - -A\_IDENT+$
$component\_implementation\_name ::=$
$component\_type\_identifier.component\_implementation\_identifier$
$unique\_component\_type\_reference ::= [package\_name ::]component\_type\_identifier$
$unique\_component\_implementation\_reference ::=$
$[package\_name ::]component\_implementation\_name$
$unique\_component\_classifier\_reference ::=$
$unique\_component\_type\_reference + unique\_component\_implementation\_reference$
$unique\_feature\_group\_type\_reference ::= [package\_name ::]feature\_group\_type\_identifier$

16.3 Non extensible AADL

16.3.1 Component type

$$A\_COMP\_TYPE ::= A\_IDENT \times A\_PROTOTYPE \times A\_FEATURE$$
$$\times A\_FLOW\_SPEC \times [A\_MODALITY] \times A\_PROPERTY \times A_ANNEX$$
$$A\_MODALITY ::= A\_MODE + \times A\_MODE\_TRANS$$
$$A\_PROPERTY ::= A\_PROP\_ASSO + A\_CONT\_PROP\_ASSO$$

$component\_type$ ::= $component\_category defining\_component\_type\_identifier$
$--A\_IDENT$
[**prototypes**($prototype+|none\_statement$)]
[**features**($feature+|none\_statement$)]
[**flows**($flow_spec+|none\_statement$)]
[**modes**($mode+mode\_transition*|none\_statement$)]
[**properties**(
$component\_type\_property\_association|contained\_property\_association+$
$|none\_statement$)]
$annex\_subclause^*$
end$defining\_component\_type\_identifier$;

A_PROP_ASSO = $add : (id : A\_IDENT+) \times A\_PROP\_VALUE+$
$set : (id : A\_IDENT+) \times A\_PROP\_VALUE\times$
$A\_IN\_BINDING \times A\_IN\_MODES$

$property\_association$ ::= $--newvalueforaproperty$
[$property\_set\_identifier$ ::]$property\_name\_identifier \Rightarrow property\_value$
[$in\_binding$] $--A\_IN\_BINDINGTOBEDEFINED$
[in_modes]; $--A\_IN\_MODES$

16.3.2 Component implementation

A_COMP_IMPL = $A\_IDENT \times A\_PROTOTYPE \times A\_FEATURE$
$\times A\_FLOW\_SPEC \times A\_MODALITY \times A\_PROPERTY \times A\_ANNEX$

$component\_implementation$ ::= $component\_category$**implementation**
$defining\_component\_implementation\_name$ – –A_NAME
[**prototypes**($prototype$ + |$none\_statement$)]
[**subcomponents**($subcomponent$ + |$none\_statement$)]
[**calls**($subprogram\_call\_sequence$ + |$none\_statement$)]
[**connections**($connection$ + |$none\_statement$)]
[**flows**($flow\_implementation$|$end\_to\_end\_flow\_spec$ + |$none\_statement$)]
[**modes**($mode$ + $mode\_transition$ * |$none\_statement$)]
[**properties**($property\_association$|$contained\_property\_association$+
|$none\_statement$)]
$annex\_subclause$*
end$defining\_component\_implementation\_name$;

$subcomponent$ ::= $defining\_subcomponent\_identifier$: – – A_IDENT
(($component\_category$
[$unique\_component\_classifier\_reference$ – –A_IDENT
[$prototype\_bindings$]]
[$array\_dimensions$])
|$prototype\_reference$)
[$subcomponent\_property\_association$|$contained\_property\_association$+]
[in_modes]; – – A_IN_MODES

$data\_subcomponent$ ::=
$defining_subcomponent_identifier$: – – A_IDENT
(($data$[$unique\_component\_classifier\_reference$ – –A_IDENT
[$prototype\_binding$]])
|$prototype\_reference$)

NOTE: The above syntax rule is a variation of the subcomponent syntax rule. The above syntax rule also applies to the subcomponent_refinement syntax.

16.4 Annex

$annex\_subclause$ ::= **annex** $annex\_identifier$ $--A\_IDENT$
(({ ∗ ∗ $annex\_specific\_language\_constructs$ ∗ ∗ })|$none$);
$annex\_library$ ::= **annex** $annex\_identifier$ $--A\_IDENT$
(({ ∗ ∗ $annex\_specific\_reusable\_constructs$ ∗ ∗ })|$none$);

16.5 Prototypes

$prototype$ ::= $defining\_prototype\_identifier$: $--A\_IDENT$
$component\_category$[$unique\_component\_classifier\_reference$] $--A\_IDENT$
{$property\_association$}+;
$prototype\_refinement$ ::= $defining\_prototype\_identifier$: $refinedto$ $--A\_IDENT$
$component\_category$[$unique\_component\_classifier\_reference$] $--A\_IDENT$
{$property\_association$}+;
$prototype\_reference$ ::= **prototype** $prototype\_identifier$ $--A\_IDENT$
$prototype\_bindings$ ::= ($prototype\_binding$(, $prototype\_binding$)∗)
$prototype\_binding$ ::=
$prototype\_identifier$ ⇒ $--A\_IDENT$
($component\_categoryunique\_component\_classifier\_reference$
$--A\_IDENT$
[$array\_dimensions$])
|($prototypeprototype\_identifier$) $--A\_IDENT$

16.6 Extensible AADL

$A\_EXT\_COMP\_TYPE$ = A_IDENT × $A\_REF\_PROTOTYPE$ ∗ ×$A\_REF\_FEATURE$
× $A\_REF\_FLOW\_SPEC$ ∗ ×$A\_MODALITY$ × $A\_PROPERTY$ ∗ ×A_ANNEX∗

component_type_extension ::=
    component_categorydefining_component_type_identifier
extends*unique_component_type_reference*[*prototype_bindings*] −−*A_IDENT*
[**prototypes**(*{prototype|prototype_refinement}* + *|none_statement*)]
[**features**(*feature|feature_refinement* + *|none_statement*)]
[**flows**(*flow_spec|flow_spec_refinement* + *|none_statement*)]
[**modes**(*mode|mode_refinement|mode_transition* + *|none_statement*)]
[**properties**(
    component_type_property_association|contained_property_association+
    |none_statement)]
annex_subclause∗
end*defining_component_type_identifier*;

component_implementation_extension ::=
component_category **implementation**
defining_component_implementation_name − −*A_NAME*
extends *unique_component_implementation_reference* − −*A_LNAME*
[*prototype_bindings*]
[**prototypes**({*prototype*|*prototype_refinement*}+|*none_statement*)]
[**subcomponents**
(*subcomponent*|*subcomponent_refinement*+|*none_statement*)]
[**calls**(*subprogram_call_sequence*+|*none_statement*)]
[**connections**
(*connection*|*connection_refinement*+|*none_statement*)]
[**flows**({*flow_implementation*|*flow_implementation_refinement*|
end_to_end_flow_spec|*end_to_end_flow_spec_refinement*}+
|*none_statement*)]
[**modes**({*mode*|*mode_refinement*|*mode_transition*}+|*none_statement*)]
[**properties**({*property_association*|*contained_property_association*}+
|*none_statement*)]
{*annex_subclause*}*
end *defining_component_implementation_name*;

subcomponent_refinement ::=
defining_subcomponent_identifier : **refinedto**
((*component_category*
[*unique_component_classifier_reference*] − −[*A_IDENT*]
[*prototype_bindings*]
[*array_dimensions*])
|*prototype_reference*)
[{{*subcomponent_property_association*
|*contained_property_association*}+}]
[*in_modes*]; − −*A_IN_MODES*

$array\_dimensions$ ::= $\{[[array\_dimension\_size]]\}*$
$array\_dimension\_size$ ::= $numeral$
$array\_selection\_identifier$ ::= $identifierarray\_selection$
$array\_selection$ ::= $\{[range\_selection]\}*$
$range\_selection$ ::= $numeral[..numeral]$

$feature\_refinement$ ::=
$port\_refinement|feature\_group\_refinement|subprogram\_refinement|$
$subcomponent\_access\_refinement|parameter\_refinement$

$feature\_group\_type$ ::=
featuregroup $defining\_identifier$
[**prototypes**($\{prototype\}+|none\_statement$)]
(**features** $\{feature|feature\_group\_spec\}*$
[**inverseof** $unique\_feature\_group\_type$]
|**inverseof** $unique\_feature\_group\_type$)
[**properties**($\{featuregroup\_property\_association\}+|none\_statement$)]
$\{annex\_subclause\}*$
end $defining\_identifier$;

$feature\_group\_type\_extension$ ::=
featuregroup $defining\_identifier$
extends $unique\_feature\_group\_type\_reference[prototype\_bindings]$
$--A\_IDENT$
[**prototypes**($\{prototype\}+|none\_statement$)]
features
$\{feature|feature\_refinement|$
$feature\_group\_spec|feature\_group\_refinement\}*$
[**inverseof** $unique\_feature\_group\_type$]
[**properties**($\{featuregroup\_property\_association\}+|none\_statement$)]
$\{annex\_subclause\}*$
end $defining\_identifier$;

$feature\_group\_spec$ ::=
$defining\_feature\_group\_identifier$: **featuregroup**
[[**inverseof**]$unique\_feature\_group\_type\_reference[prototype\_bindings]$]
$--A\_IDENT$
[{{$featuregroup\_property\_association\}+$}];

$feature\_group\_refinement$ ::=
$defining\_feature\_group\_identifier$: **refinedto**
featuregroup
[[**inverseof**]$unique\_feature\_group\_type\_reference[prototype\_bindings]$]
$--A\_IDENT$
[{{$featuregroup\_property\_association\}+$}];

References

[1] Erwan Jahier, Nicolas Halbwachs, and P. Raymond. Synchronous Modeling and Validation of Priority Inheritance Schedulers. In *Fundamental Approaches*

to Software Engineering Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, pages 140–154, York Royaume-Uni, 03 2009. Springer Verlag.

[2] SAE Aerospace. Architecture Analysis and Design Language (AADL). *SAE AS5506A*, 2009.