AADL to Signal/SSME Transformation

Yue Ma, ESPRESSO Team, INRIA
Dec 05, 2012

Foreword

The purpose of this document is to provide a global view of our implementation of transformation from AADLv2 [4, 5] to SSME [8] / Signal [6, 7]. It gives a list of components that we have implemented, and also some technical details of the implementation. The implementation Java code has been uploaded to Polychrony gforge [1]. A validated example, **SDSCS** (**Simplified Doors and Slides Control System** [12, 11]), has been applied to this implementation. This example contains a subset of AADL components: system, process, thread, data, device, bus, processor, data port, port connection and some properties. These components have been (partly) represented in Signal.

version	date	notes
0.1	13/07/2011	components modeling in Signal
0.2	22/07/2011	process is passed as parameter in the processor Signal process
0.3	29/07/2011	the SDSCS example is referenced as an example
0.4	08/08/2011	bundle is used to simplify the Signal code
0.5	18/08/2011	io signals of process instance are omitted to simplify the Signal code
0.6	18/10/2011	event (event data) port modeling in Signal event (event data) port connection modeling in Signal
0.7	26/10/2011	port group and port group connection modeling in Signal program architecture
0.8	15/12/2011	a brief introduction of behavior annex and the behavior modeling in Signal update the modeling of event (event data) port
0.9	16/01/2012	shared data component modeling in Signal
1.0	03/02/2012	bundle for shared data
1.1	28/06/2012	subprogram and subprogram call, requires subprogram, subprogram remote call
1.2	03/10/2012	integration with BA, use .aadl as source model instead of .aaxl model connect to schedule generator
1.3	29/10/2012	test cases
1.4	05/12/2012	refine and update port modeling

Table 1. Document history

Contents

1 Introduction

Architecture Analysis and Design Language (AADL) is gradually adopted in the design of safety-critical systems. It is based on the component-based paradigm that enables reusability and compositionality. However, its formal verification and validation is always a challenge. In the framework of European ITEA2 OPEES project [2], we propose to adopt a polychronous model of computation of the Signal language, to bridge between AADL and its formal temporal analysis.

In our approach, concretely, the AADL components specified with temporal properties are first modeled in Polychrony. Non-functional timing properties of a system are then used for formal timing analysis based on both logical clock calculus and affine clock relation analysis. Considering durability and interoperability, we have developed an AADL2SSME tool chain, with the experimentation on case studies, to support OPEES processes related to tools maturation, verification and qualification. Our tool chain is integrated into the framework of Polarsys, an OPEES industry working group.

A rough description of AADL and a brief description of its modeling in Polychrony [10] have been given in the ESPRESSO AADL Digest Report [9]. In the present document, we mainly focus on the technical implementation details.

1.1 Global architecture

The global architecture, in the context of Polychrony, is represented in Figure 1.

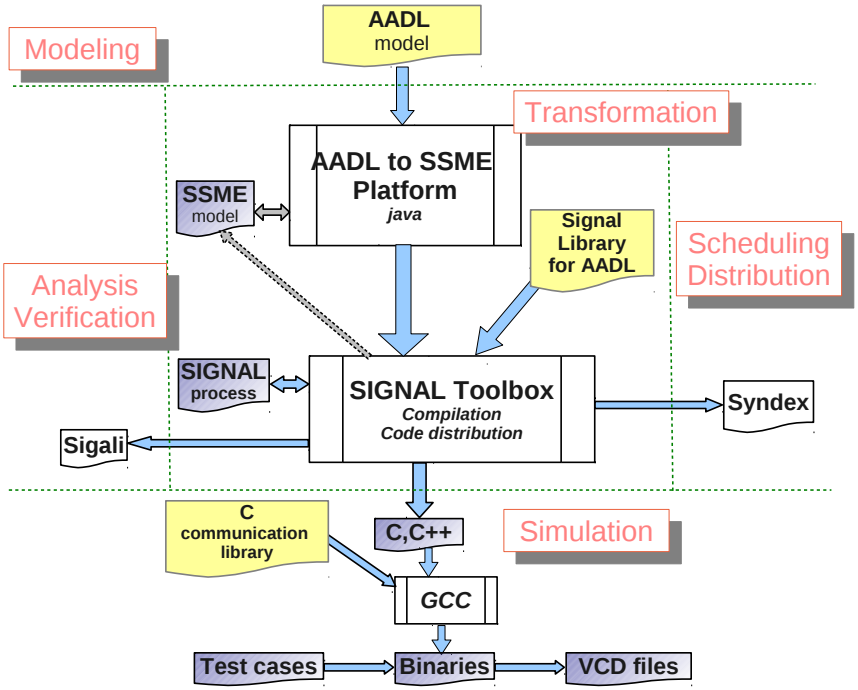

Figure 1. Global architecture

1.2 Functional architecture

The functional architecture is described in Figure 2.

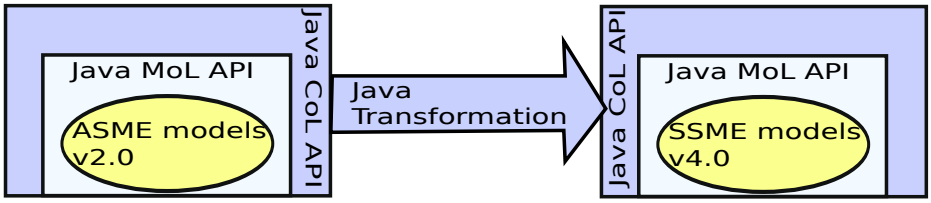

Figure 2. Functional architecture

- **MoL** stands for **Model (low) Level**. The corresponding Java Mol API is generated by Eclipse as a set of classes providing access to attributes.
- **CoL** stands for **Concept (high) Level**. The corresponding Java CoL API is designed to make the transformation independent of some specific meta-model version.
- **SSME**: **Signal Syntax Model under Eclipse** is a full syntax oriented meta-model of Signal. The SSME Java CoL API is implemented as a partial implementation of the API specified by the corresponding API in Signal tools, mostly the generative methods (signalTreeAPI).
- **ASME**: **AADL Syntax Model under Eclipse** is a full syntax oriented meta-model of AADL provided by OSATE [3]. The Java Mol API for AADL is developped by OSATE. The Java CoL API for ASME is built in parallel with the transformation such that all model oriented accesses are rejected in this API, and this API is designed following the principles of the Java CoL API for SSME (mostly the “get” methods).

1.3 Meta architecture

The meta architecture is described in Figure 3.

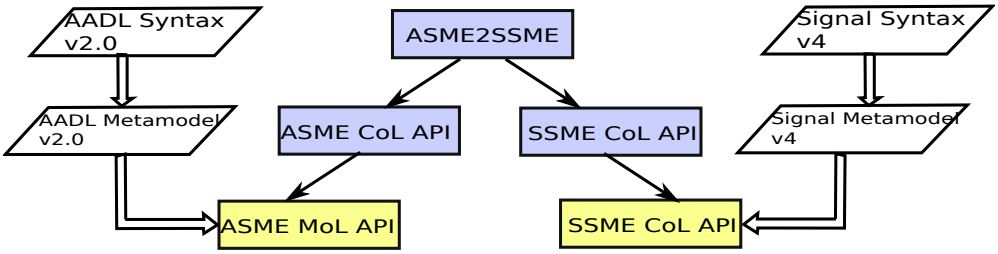

Figure 3. Meta architecture

1. **AADL2SSME** is the Java hand written translator. It uses high level APIs to models. This program should only be sensible to bug corrections, Java deprecation, major AADL or Signal language evolution.

2. **XX CoL/MoL API** are the Java high/low level APIs to models (high level APIs are hand written). They are sensible to bug corrections, Java deprecation, major AADL or Signal language evolution. They are also sensible to pure syntactic changes in the interfaced language, and to changes in the model representation.

3. The ASME meta-model and ASME MoL API are provided by OSATE. An AADL Col API is developed.

2 Signal library

We define a Signal library containing the Signal process models representing some AADL concepts.

The following modules are developed and tested. A short comment is given for the processes in the Signal program:

1. **ASME2SSME_LIB**: uses the following defined modules, so that in the transformation, we can have a simple $use\ ASME2SSME\_LIB$ expression to use all the defined libraries.

```
module ASME2SSME_LIB =
   use AADL_TYPE;
   use AADL_PROPERTY;
   use AADL_CONNECTION;
   use AADL_DATAPORT;
   ...
end;
```

2. **SIGNALLIB**: two memory processes: $AT()$ and $Typed\_AT()$ (the type is passed as a parameter).

3. **AADL_TYPE**: some predefined types and constants, e.g.:

```
type Time_Units = enum (ps, ns, us, ms, sec, min, hr);
type Supported_Dispatch_Protocols = enum
     (Periodic, Sporadic, Aperiodic, Timed, Hybrid, Background);
```

4. **AADL_PROPERTY**: predefined property processes. The body of these processes is empty (they are considered as external processes).

5. **AADL_CONNECTION**: defines a $Connection()$ process that represents the connection behavior (connection between two processes that are bound to different processors).

6. **AADL_DATAPORT**: defines $InDataPort\_behavior()$ and $OutDataPort\_behavior()$ processes, modeling the in and out data port behaviors.

7. **AADL_EVENTPORT**: $InEventPort\_behavior()$, $OutEventPort\_behavior()$.

8. **AADL_EVENTDATAPORT**: $InEventDataPort\_behavior()$, $OutEventDataPort\_behavior()$.

9. **fifoLib**: defines two types of fifo (with or without reset): $fifo\_reset()$, $fifo()$.

3 ASME2SSME Transformation

The structural translation of AADL (ASME) systems to SSME has been (partly) implemented, according to the functional architecture and using the Signal library. In this section, we give a global view of the components and features that have currently been implemented in our ASME2SSME transformation. **The implementation details of each component (type and implementation) will be given in the following subsections.**

3.1 A global view of the implementation

Fig 4 shows the complete tool chain for modeling, timing analysis and verification of AADL in the polychronous MoC. The AADL model, which conforms to the AADL metamodel, is captured textually as AADL textual code in the OSATE toolkit. The timing properties provide detailed timing specifications related to the AADL components. A model transformation ASME2SSME tool allows to perform analysis on the ASME models (AADL Syntax Model under Eclipse) and generate Signal models in SSME (Signal Syntax Model under Eclipse) models. This tool was implemented in Java, as an Eclipse plugin, and takes as input an AADL model (.aadl2) and generates an SSME model (.ssme), which can be transformed to Signal text within Polychrony.

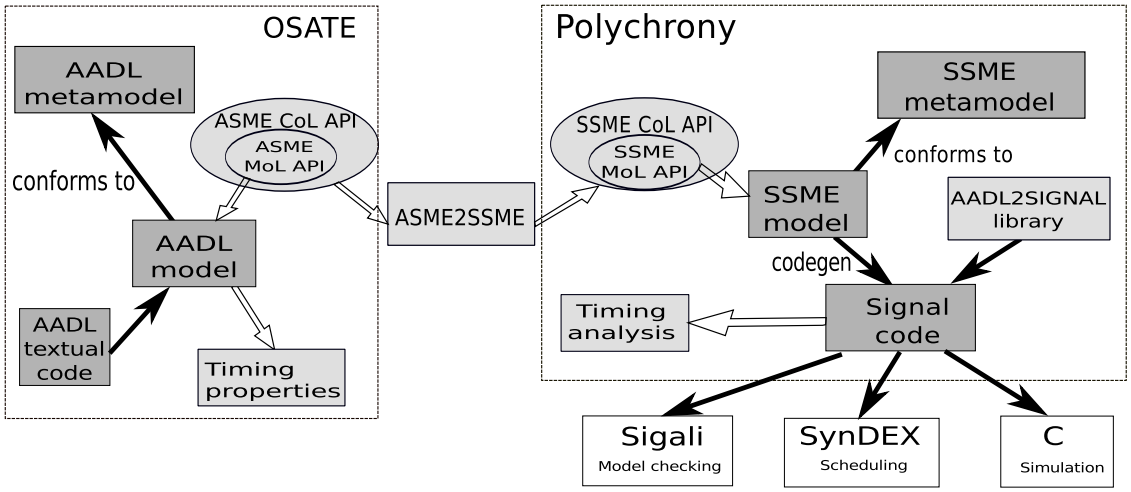

Figure 4. AADL to Signal transformation tool chain

An AADL2SIGNAL library provides common Signal processes reducing significantly the transformation cost. The timing properties represented as Signal clocks are calculated and analyzed in the compilation of Signal programs. After that, the executable code is generated for simulation. Associated tools, such as Sigali or SynDEX, can be used for further verification and validation.

3.1.1 The implemented components

Note: the implementation status of each component (or feature) is marked as:

- "fully" (for totally implemented),

- "partly" (for partly implemented),
- "expected" (for not implemented yet but to be done),
- "open" (for not decided),
- "none" (for not taken into account).

The date of last modification is given below.

The status of ASME2SSME transformation of AADL components are shown in Table 2. For the reason of recursion (one component may be a sub-component of another component), most of the components are marked as "partly".

composite component	System	partly	2011-07-22
software component	Process	partly	2011-07-22
	Thread	partly	2011-07-22
	Thread group	none	
	Subprogram	partly	2012-03-28
	Subprogram group	none	
	Data	partly	2012-02-03
hardware component	Processor	partly	2012-02-03
	Virtual processor	none	
	Memory	partly	2012-02-03
	Bus	partly	2011-07-22
	Virtual bus	none	
	Device	partly	2011-07-22
feature	port	partly	2012-02-03
	parameter	partly	2011-07-22
	subprogram access	partly	2012-03-29
	data access	partly	2012-02-02
	bus access	open	
	feature group	partly	2012-02-03
connection	port connection	partly	2011-07-22
	parameter connection	partly	2011-07-22
	access connection	none	
	feature group connection	partly	2012-02-03
others	flow	none	
	mode	expected	
	property	partly	2012-03-28

Table 2. A global view of ASME2SSME transformation status

Since there are a number of properties predefined in the AADL standard, and also the users can define their own properties, we will not list all the properties here. We will give the implemented properties in the corresponding subsection for each considered component.

3.1.2 Transformation principles

The **ASME2SSME** transformation is recursive. (The notation $\mathcal{T}()$ is used to represent the translation.)

A package $k \in Pkg =< id_k,\ p\_Pkg,\ v\_Pkg,\ P >$ which represents the root of an AADL specification, is identified by a name id_k, and may consist of a public package content p_Pkg, a private package content v_Pkg and properties P. (The other possible AADL root, a property set, is not considered in this document.) It is transformed into one SIGNAL module, which is a root of a SIGNAL program allowing to describe an application in a modular way, with the help of predefined AADL2SIGNAL libraries.

$$\mathcal{T}(k) ::= \textbf{module}\ id_k\ =\ (\mathcal{T}(p\_Pkg)|\ \mathcal{T}(v\_Pkg)|\ \mathcal{T}(P)|)$$

The translated SIGNAL module contains the processes that are translated from the AADL components by the following recursive translation rules.

1. A public package content $p\_Pkg =< c\_Pkg >$ declares a package content c_Pkg that is visible outside the package. A private package content $v\_Pkg =< c\_Pkg >$ declares a package content c_Pkg that is visible only within the package. In the current implementation, they are translated the same as the declared package content c_Pkg:

$$\mathcal{T}(p\_Pkg) ::= (\mathcal{T}(c\_Pkg))$$
$$\mathcal{T}(v\_Pkg) ::= (\mathcal{T}(c\_Pkg))$$

2. A package content $c\_Pkg =< TX, X >$ is composed of component types TX and component implementations X. In our transformation, we only translate the component implementations X. The corresponding component type is referenced during the transformation of component implementation, but not translated directly. A package content c_Pkg is represented by a composition of SIGNAL processes that are translated from the component implementations.

$$\mathcal{T}(c\_Pkg) ::= (\mathcal{T}(x_1)|\ \mathcal{T}(x_2)|\ \ldots)$$
where $\forall x_i \in X$, apply the transformation rule $\mathcal{T}(x_i)$ described below

3. Generally, a AADL component implementation $x \in X$ is a tuple $< id_x, F, Y, Cal, C, P, M >$, (the features F of its type is considered as one section of component implementation, and the flows and modes are not considered yet in our transformation. A component may not include certain sections depending on the component category.), where:

   - id_x is the identifier of the component implementation,
   - F is the features (mainly the ports) of the corresponding component type that define a functional interface,

- Y is the subcomponents of x, where $\forall y_i \in Y$ is a subcomponent, $y_i =< id_{y_i}, y_i' >$, which is composed of an identifier id_{y_i} and its component classifier $y_i' \in X$ which is a component implementation (in our current version),
- Cal is the subprogram call sequences,
- $C = F \times F$ is the connections (mainly the port connections in the cuurent implementation), which is an explicit relation between features (ports) that enables the directional exchange of data or event among its features (ports) and subcomponents,
- P is the set of properties,
- M is a transition system that provides the functional behavior.

A SIGNAL process (id_x_TT where TT is a category of AADL component, e.g., *thProducer_Thread* for thread *thProducer*) corresponding to an AADL component implementation *x* composed of Signal subprocesses:.

$$\begin{aligned}\mathcal{T}(x) ::= \ & \textbf{process}\ id_x\_TT\ =\ (\mathcal{T}(F)) \\ & (|\mathcal{T}(Y)\ |\ \mathcal{T}(Cal)\ |\ \mathcal{T}(C)\ |\ \mathcal{T}(P)\ |\ \mathcal{T}(M)|) \\ \texttt{where}\ & TT\ =\ \textbf{enumeration}\ \{\texttt{system, process, thread, subprogram,} \\ & \texttt{processor, device, memory, device, bus}\}\end{aligned}$$

An example of Signal process that represents the *Door.impl* subsystem of the SDSCS system is given below (for space limits, only part of the code is shown):

```
process Door_imp_System = ( ! DataA closed1;)
(| Door_imp_System_behavior()%system behavior sub process %
 | Door_imp_System_property()%system property sub process %
 | L_4 := ClosedSensor_imp_Device{}()
 | closed1 := L_4
 |)
where
  DataA L_4;
  process Door_imp_System_behavior =( )%Door_imp_System_behavior%;
  process Door_imp_System_property =( )%Door_imp_System_property%;
  process ClosedSensor_imp_Device = (! DataA closed;) (| ...|);
end      %Door_imp_System%;
```

4. The **interface** of process id_x_TT contains the input/output signals translated from the features (ports) $F\ =< fi_1, fi_2, \ldots, fo_1, fo_2, \cdots >$ provided by the component type, where an in port $fi_m \in F$ is modeled as an input signal, and an out port $fo_n \in F$ is modeled as an output signal, (Some additional control signals may also be added depends on the component category, e.g., Dispatch and Deadline for a *thread*.):

$$\mathcal{T}(F) ::= \ ?\,\mathcal{T}(fi_1);\mathcal{T}(fi_2);\ldots;$$
$$!\,\mathcal{T}(fo_1);\mathcal{T}(fo_2);\ldots;$$
where $\forall fi_m \in F (resp.\ fo_n \in F)$, apply the transformation rule $\mathcal{T}(fi_m)(resp.\ \mathcal{T}(fo_n))$ in Section 3.15

5. The **body** is composed of SIGNAL processes that represent:

- subcomponents Y:

$$\mathcal{T}(Y) ::= (|id_{y_1} :: \mathcal{T}(y'_1) \mid id_{y_2} :: \mathcal{T}(y'_2) \mid \ldots |)$$
where $\forall y'_i \in Y$, apply the recursive translation rule $\mathcal{T}(y'_i)$

- subprogram call sequences Cal:

$$\mathcal{T}(Cal) ::= (|\mathcal{T}(cal_1) \mid \mathcal{T}(cal_2) \mid \ldots |)$$
$\forall cal_i \in Cal$, apply the translation rule $\mathcal{T}(cal_i)$ for subprogram call sequence

- connections $C = F \times F$:

$$\mathcal{T}(C) ::= (|\mathcal{T}(c_1) \mid \mathcal{T}(c_2) \mid \ldots |)$$
$\forall c_i \in C$, apply the translation rule $\mathcal{T}(c_i)$ in Section 3.16

- associated properties P:

$$\mathcal{T}(P) ::= \mathbf{process}\ id_x\_TT\_property = ()$$
$$(|\mathcal{T}(p_1) \mid \mathcal{T}(p_2) \mid \ldots |)$$
$$\forall p_i \in P,\ \mathcal{T}(p_i) ::= \mathbf{process}\ p_i\_property = ()()$$

Each property is represented as a Signal process predefined in the library (only the timing properties are represented in the current implementation). For example, the property of $Input\_Time$ is implemented as a Signal process, where the actual input time is to be verified whether it satisfies the timing constraints by providing external C code (to be implemented).

- transition system M. We define a transition system M to be a tuple $< S', T', s_0, C', A >$, where S' is a set of states, $T' \subseteq S' \times C' \times S' \times A$ is the set of transitions between states S', triggered by the condition C' with functional behavior A, and s_0 an initial state (the transformation details of transition system is to be presented in Section 4.):

$$\mathcal{T}(M) ::= \textbf{process}\ x\_TT\_behavior = ()$$
$$(|\mathcal{T}(S' \times C' \times S' \times A)|)$$

We have given the general principles of the transformation in SIGNAL. Certain component may have some specific transformation. The interpretation details of each component are presented in sections below.

3.2 Data

A data implementation component, noted $x \in D$, is a tuple $< id_x, P, Y >$ (the features and connections are not considered yet) where:

- id_x is the identifier of the data implementation,
- P is the set of properties,
- Y is the subcomponents of x, where $\forall y_i =< id_{y_i}, y_i' >\in Y$ is a subcomponent, which is composed of an identifier id_{y_i} and its classifier $y_i' \in D \cup Sp$ which is a data implementation D or a subprogram implementation Sp.

Two cases of context are considered:

- **Case 1**: a **data component type** represents a data type in the source text.
- **Case 2**: a **data subcomponent** (not declared in subprograms) represents (shared) static data in the source text.

Case1: data x represents a type

- If x is a predeclared type in Signal, e.g., **integer**, **boolean**, the Signal predefined type is used directly:

$$\mathcal{T}(x) ::= id_x$$
$$\texttt{where}\ id_x = \textbf{enumeration}\ \{\texttt{integer, boolean, string, ...}\}$$

- If x is not predeclared and no subcomponents is contained, a new type declaration is created:

$$\mathcal{T}(x) ::= \mathbf{type}\ id_x\ =\ \mathbf{external};$$

- If x is not predeclared and x contains data subcomponents, a **struct** type (— or a **bundle** type for more precisely since they are not synchronized —) declaration is created:

$$\mathcal{T}(x) ::= \mathbf{type}\ id_x\ =\ \mathbf{struct}\ (\mathcal{T}(y_1);\ \mathcal{T}(y_2);\ \ldots)$$
$$\texttt{where}\ \forall y_i =< id_{y_i}, y_i' >\in Y \wedge y_i' \in D, \mathcal{T}(y_i) ::= \mathcal{T}(y_i')\ id_{y_i}$$

The data implementation status in Signal are given in Table 3.

data type	features	feature group subprogram access subprogram group access	none expected none	
data implementation	subcomponents connection	data subprogram	partly expected expected	2012-02-20
predefined properties		Type_Source_Name Source_Name Source_Text Source_Language Source_Data_Size Allowed_Memory_Binding_Class Allowed_Memory_Binding Actual_Memory_Binding Base_Address Source_Code_Size Access_Right Concurrency_Control_Protocol	expected expected expected expected expected expected expected expected expected expected expected expected	

Table 3. The data implementation status

For instance, we have two data types, DataA and DataB.imp, in the SDSCS example:

```
data DataA
end DataA;

data DataB
end DataB;

data implementation DataB.imp
  subcomponents
```

```
    x: data integer;
    y: data DataA;
end DataB.imp;
```

They are represented as follows in Signal:

```
type DataA = external;
type DataB = external;
type DataB_imp = struct (integer x; DataA y;);
```

Case2: data x represents static data shared by the components Data subcomponent (of a process or a system) represents shared resources . (Data subcomponent declared in a subprogram or a thread represents a **local variable**.)

The status of implementation of data subcomponent and data access are given in Table 4.

data subcomponent of					
process	thread	system	thread group	subprogram	data
partly	partly	none	none	none	partly
2011-12-15	2011-12-15				2012-02-03

Table 4. Data subcomponent

Components can have shared access to data subcomponents. A **requires data access** declaration indicates that a component requires access to a component declared external to the component. A **provides data access** declaration indicates that a subcomponent provides access to a data component contained in the component. Each required reference to shared data may have its own **Access_Right** property value (**read_write** by default).

1. **Case 2.1: data subcomponent of a process, or a system**

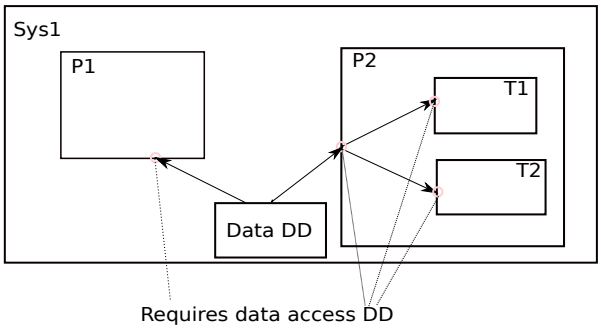

Figure 5. AADL requires data access

In Figure 5, data *DD* is a subcomponent of system *Sys1*, and two processes *P1* and *P2* **requires data access** of *DD*.

In Figure 6, Data *DD* is made accessible outside process *P1* through a **provides data access**, and it is accessed by process *P2* (thread *T1* and *T2*) as expressed by **requires data access**.

Implementation in Signal:

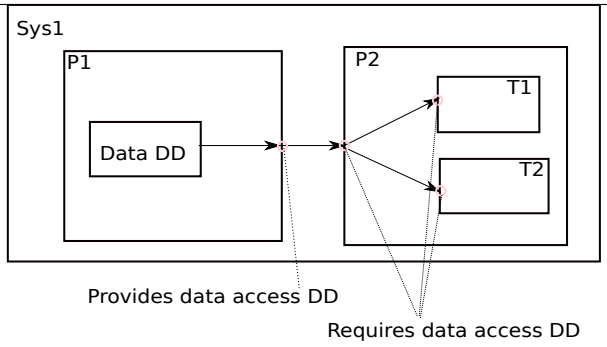

Figure 6. AADL data access

- The shared data is represented as a Signal process *fifo_reset()* (Figure 7). (*fifo_reset()* is declared in the library **fifoLib**.)
  $(Queue\_r,\ Queue\_cnt)\ :=\ fifo\{integer,\ 3,\ 0\}\ (Queue\_w,\ Queue\_reset)$
  A Signal process call is made in the body of its upper level component (system *Sys1* in the first example, and process *p1* in second example). The fifo $size$, $type$ and $initial\ values$ are passed as parameters.

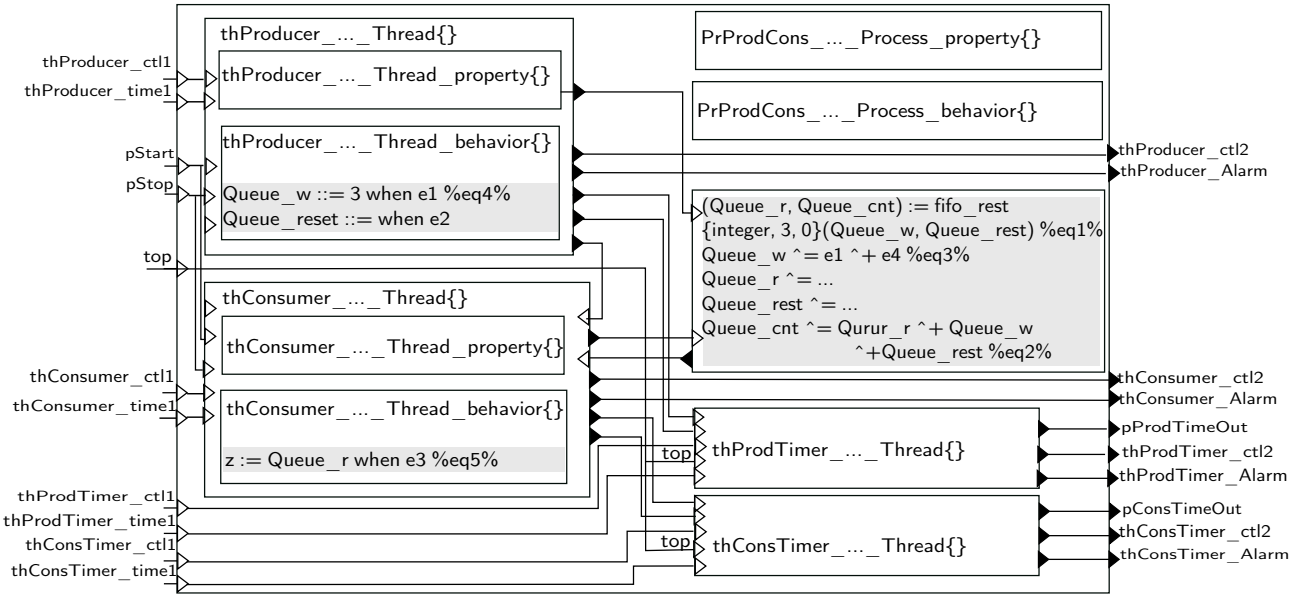

Figure 7. AADL data access in Signal

- Shared variables $Queue\_w$, $Queue\_reset$ and local variables $Queue\_r, Queue\_cnt$ (where $Queue$ is the name of the data) are declared as local signals of Signal process *p*, where *p* is the Signal process which represents Queue's upper component.

```
shared event Queue_reset;
shared integer Queue_w;
integer Queue_r, Queue_cnt;
```

For example, in Figure 6, $Queue$ is declared as a subcomponent of process *P1*, and *P1* declares a **provides data access Queue**, so $Queue\_w, Queue\_reset, Queue\_r, Queue\_cnt$ are declared as local variables of upper level Signal process which is composed of *P1, P2* and others. If no **provides data access Queue** is declared in *P1*, then the variables are declared in *P1*, and the data is shared inside *P1*.

- The data is accessed by different Signal processes in different time instances. To write a data into the fifo, a partial definition of $Queue\_w$ is provided (in the behavior part): $Queue\_w ::= xx\ when\ e1$. To read a data, one can direct use: $z := Queue\_r\ when\ e3$.
- The interface of **provides** and **requires** data access is not translated explicitly. For each component that requires access data $Queue$, the event that represent the read, write and reset clock are added in the interface of its Signal process: $event\ x\_ReadTime$, $event\ x\_WriteTime$, $eventx\_ResetTime$. The read (write, reset) clock of data $Queue$ is the union of all such events, and the clock of count cnt is at least the union of read clock, write and reset clock.

```
Queue_r ^= e3
Queue_w ^= e1 ^+ e4
Queue_reset ^= e2
Queue_cnt ^= Queue_r ^+ Queue_w ^+ Queue_reset
```

- The **data access connection** is not translated explicitly. The access of read/write data implicitly indicates their connections.
- A bundle **SharedData_TIME** is used to represent the read/write/reset clock of a shared data.

```
type SharedData_TIME = bundle (event TRead; event TWrite; event TReset;);
```

A possible Signal implementation of the first example is given below:

```
process s1 = (? ... ! ...)
(| (..., DD1) := p1(...)
 | (..., DD2) := p2(...)
 | (DD_r, DD_cnt) := fifo{integer, 10, 0}(DD_w, DD_reset)
 | DD_cnt ^= DD_w ^+ DD_r ^+ DD_reset
 | DD_r ^= DD1.TRead ^+ DD2.TRead
 | DD_w ^= DD1.TWrite ^+ DD2.TWrite
 | DD_reset ^= DD1.TReset ^+ DD2.TReset
 | ...
 |)
where
    shared integer DD_w;
    integer DD_r, DD_cnt;
    SharedData_TIME DD1, DD2;
    ...
    process p1 = (? ... ! ..., SharedData_TIME DD1;)
      (| DD_w ::= 1 when ...
       | DD1.TRead := ...
       | DD1.TWrite := ...
       | DD1.TReset :=
```

```
        | ... |);
    process p2 = (? ... ! ..., SharedData_TIME DD2;)
      (|(..., l3) := T1(...)
       |(..., l4) := T2(...)
       | DD2.TRead := l3.TRead ^+ l4.TRead
       | DD2.TWrite := l3.TWrite ^+ l4.TWrite
       | DD2.TReset := l3.TReset ^+ l4.TReset
       | ...|)
      where
          ...
          process T1 = (?...!..., SharedData_TIME DD3;)
          (| DD_w ::= 2 when ...
           | DD3.TRead := ...
           | DD3.TWrite := ...
           | DD3.TReset := ...
           |... |);
          process T2 = (?...!..., SharedData_TIME DD4;)
          (| y:= DD_r when ...
           | DD4.TRead := ...
           | DD4.TWrite := ...
           | DD4.TReset :=
           |...|);
      end;
end;
```

2. **Case 2.2: data subcomponent of a thread or a subprogram**

   In this case, the data is implemented as a local Signal variable.

3. **Case 2.3: data subcomponent of a thread group** (expected)

4. **Case 2.4: data subcomponent of a data** (expected)

3.3 Subprogram and subprogram call

An AADL subprogram sp component represents sequentially executed source text that is called with parameters. A subprogram implementation $sp \in Sp$ is a tuple $< id_{sp}, F, Y, Cal, C, P, M >$ where:

- id_{sp} is the identifier of the subprogram implementation,
- F is the features (ports) of the corresponding subprogram type,
- Y is the subcomponents of sp, where $\forall y_i =< id_{y_i}, y_i' >\in Y$ is a data subcomponent, which is composed of an identifier id_{y_i} and its component classifier $y_i' \in D$ which is component of data D,
- Cal is the subprogram call sequences,
- C is the connections,
- P is the set of properties,
- M is a transition system that provides the functional behavior.

A subprogram can be a subcomponent of a thread or a subprogram. It can be called by threads or subprograms **locally** or **remotely**. If a thread *t1* calls a subprogram *sp* that is executed in the context of other thread *t2*, e.g., *sp* is a subcomponent of *t2*, the call is **remote**, and *t1* is **suspended** (by default).

Subprogram transformation In general, a subprogram *xx* is implemented as a Signal process. A Signal struct type *xx_IN* is created for the in parameters (these parameters are supposed to be synchronized). A struct type *xx_OUT* is created for the out parameters and out event (event data) ports. (In this moment, we suppose that a subprogram's inputs (outputs) is not null.) An out signal that represents the component id of which the results will be returned to is added in the interface (type *TID* predefined in the library).

The body is composed of $xx\_Subprogram\_behavior()$ and $xx\_Subprogram\_property()$ process calls if xx is a subprogram implementation.

```
type xx_IN = struct(a;b;);
type xx_OUT = struct(c;d;)
process xx_Subprogram = (? xx_IN x; ! xx_OUT y; TID id; )
  (| xx_Subprogram_behavior()
   | xx_Subprogram_property()
   |)
 where
   process xx_Subprogram_behavior = (? xx_IN x; ! xx_OUT y; TID id; ) external;
   process xx_Subprogram_property =( )(...);
 end;
```

- The status of subprogram representation in Signal is given in Table 5.

subprogram type	features	out event (data) port	partly	2012-03-27
		feature group	none	
		data access	partly	2012-03-28
		subprogram access	partly	2012-03-28
		subprogram group access	none	
		parameter	partly	2011-07-22
	flow		none	
	mode		expected	
subprogram implementation	subcomponents	data	expected	
	subprogram call		partly	2012-03-28
	connection		partly	2012-03-29
	flow		none	
	mode		expected	

Table 5. The subprogram implementation status

- The standard subprogram properties could be found in the ESPRESSO AADL Digest Report [9], Section 4.1.3. For the current implementation, they are implemented as process calls in the $xx\_Subprogram\_property()$ process.

```
process acquire_impl_Subprogram_property = ()(| Source_Name_property() |);
```

Subprogram call We mainly consider the case that a thread calls subprograms. The case that a subprogram calls another subprogram is similar, to be implemented later.

1. **Local subprogram call**

   If the called or required subprogram *sp* is not a subcomponent of any other threads, then it is considered as a **local subprogram call** (Figure 8).

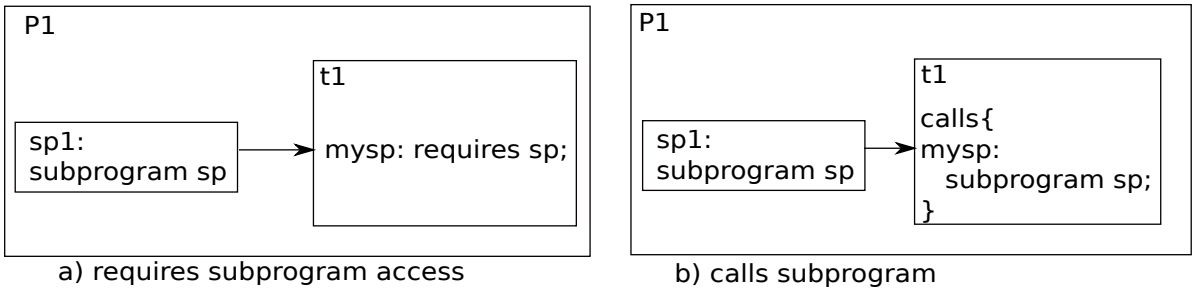

Figure 8. local subprogram access and subprogram call (in the same process)

The subprogram *sp* is shared by the threads. It is implemented similar as the shared data: only one instance of *sp* process is created in the process *p1*, and the subprogram inputs *l_x* is declared as shared signal, so that it could be partially defined by different threads. In Figure 9, thread *t1* sends out its clock of subprogram calling *mysp*. The subprogram input *l_x* is shared by the threads, and its clock is the merge of subprogram calling clocks: *l_x := mysp* $\hat{+}$ .... In thread behavior body *t1_Thread_behavior()*, the clock of *mysp* is defined: *mysp := when Start*, and the input of *l_x* is partially defined: *l_x.a ::= aa when mysp*, *l_x.b ::= bb when mysp*.

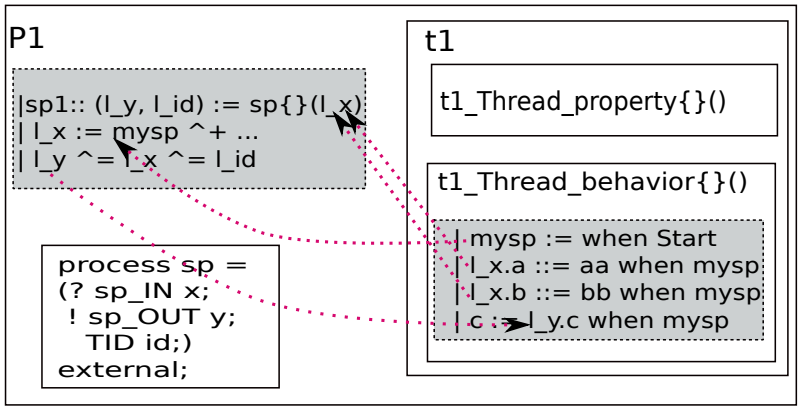

Figure 9. local subprogram call (access) implemented in Signal

The called subprogram *sp* in a different process is not implemented yet.

2. **Remote subprogram call**

   If the called subprogram *sp* is executed in the context of another thread, e.g., *sp* is a subcomponent of thread *t2*, or *sp* is a subprogram provided by thread *t2*, then it is a **remote call**. Figure 10 shows a remote subprogram call example: thread *t1* calls subprogram *sp* which is provided by thread *t2*.

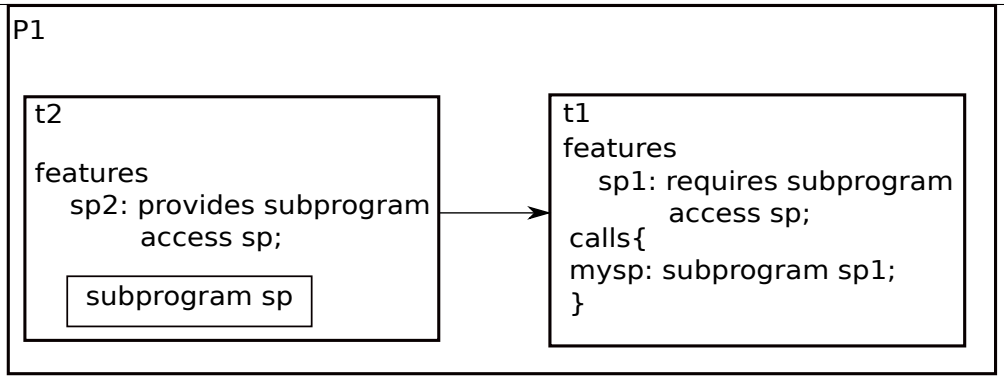

Figure 10. remote subprogram call

The caller thread is by default **suspended** until the execution of the subprogram completes (**synchronous call**). The caller thread may issue multiple concurrently executing subprogram calls and wait for their result when needed (**semi-synchronous call**). This is indicated by property **Subprogram_Call_Type**. The called subprogram acts as a **critical region** for all calls.

```
Subprogram_Call_Type: enumeration (Synchronous, SemiSynchronous) => Synchronous;
```

The behavior annex introduces more precise communication protocols for remote call. This is provided by propety **Subprogram_Call_Protocol**:

```
Subprogram_Call_Protocol: enumeration (HSER, LSER, ASER)
                  => HSER applies to (subprogram access);
```

- **HSER**: Highly Synchronous Execution Request. The caller thread is suspended until the completion of the corresponding behavior action in the server thread. (The **Subprogram_Call_Type** property is set to *synchronous*.)
- **LSER**: Loosely Synchronous Execution Request. The caller thread is suspended until the beginning of the server thread is ready to serve this request. (The **Subprogram_Call_Type** property is set to *semi-synchronous*. The subprogram cannot have out or in out parameters.)
- **ASER**: ASynchronous Execution Request. The caller thread is never suspended by the corresponding remote call. (The **Subprogram_Call_Type** property is set to *semi-synchronous*. The subprogram cannot have out or in out parameters.)

Here we only consider the case of synchronous call: the caller thread is suspended until the execution of the subprogram completes.

Figure 11 shows a possible implementation of the example in Figure 10: a subprogram subcomponent is instantiated in the body of thread *t2*. The subprogram input *l_x* (declared as a shared signal in the process *p1* body) is shared by other threads. Thread *t1* gives out the clock (*mysp*) of calling subprogram *sp*. In the behavior body of *t1*, the shared subprogram input *l_x* is partially defined.

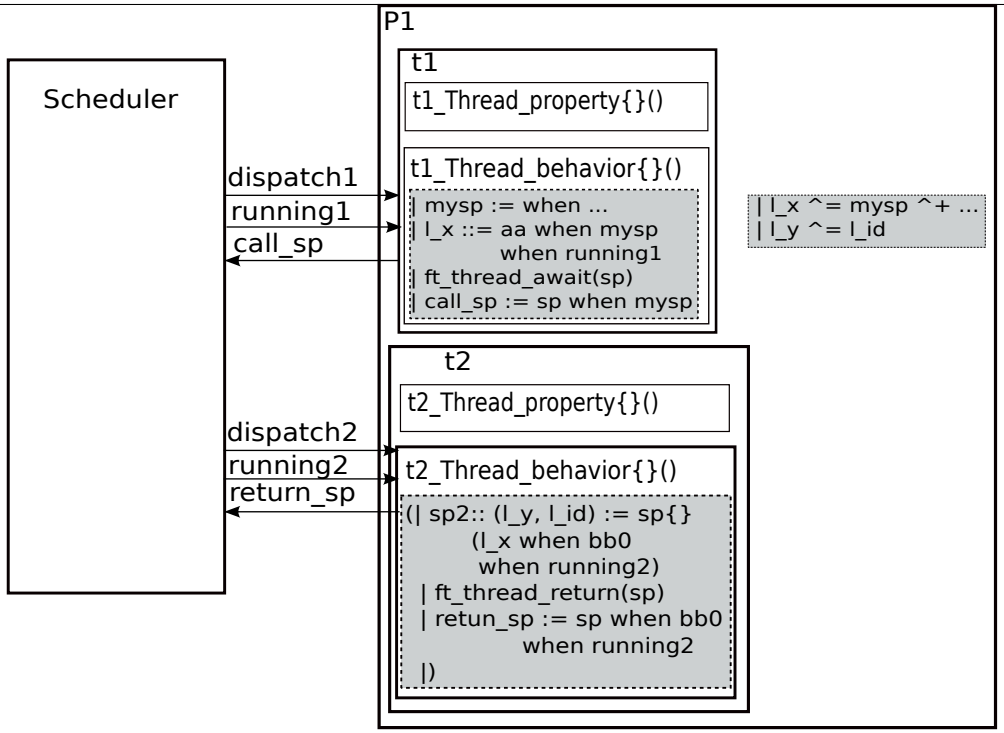

Figure 11. remote subprogram call implemented in Signal

To model the suspend/resume of threads, some synchronisation primitives, (considered as external function calls, similar as the commands from the FairThreads library) are placed into the behaviors, and some control signals, e.g., *call_sp, rentun_sp*, that communicate with the scheduler are presented. For example, a synchronisation primitive command *ft_thread_await()* is added in *t1*. It represents that *t1* requires call *sp* and waits for the result of *sp*. A control signal *call_sp* is sent to the scheduler to inform the scheduler the time that *t1* requires remote call of subprogram *sp*, and it is suspended then. The scheduler selects thread *t2* for execution. When the correspongding behavior action (that executes the subprogram) completes, a synchronisation primitive operation *ft_thread_return()* is performed, and a *return_sp* signal is sent to to scheduler, so that thread *t1* that requires access of *sp* is reset to be in the **ready** state.

Implementation details For the current implementation, we only consider the case that a subprogram *sp* is called (or required) by threads that are in a same process *p* (*sp* is not called by other processes or threads in other processes).

1. case 1: *sp* is a subcomponent of *p*; (local call)
2. case 2: *sp* is not a subcomponent of any process and thread; (local call)
3. case 3: *sp* is a subcomponent of a sub thread *t1* (remote call)

The cases that call subprogram across processes are not yet implemented.

For a process *p* add into the body of *p*:

1. case1:

```
| sp1 : (l_y, l_id) := sp{}(l_x)
| l_x ^= e1 ^+ ...
| l_y ^= l_id ^= l_x

where
  ...
  shared sp_IN l_x;
  sp_OUT l_y;
  TID l_id;
  label sp1;
end;
```

   where *ei* is a local clock that represents a subthread *ti* that requires or calls subprogram *sp*.

   The related methods:

   ASME2SSME_ProcessImpl_SubSubprograms()

   ASME2SSME_Create_SharedSubprogram()

   ASME2SSME_SharedSubprogram_Call_Sync_x()

   ASME2SSME_SharedSubprogram_Sync_y()

2. case 2:

```
| (l_y, l_id) := sp{}(l_x)
| l_x ^= e1 ^+ ...
| l_y ^= l_id ^= l_x

where
  ...
  shared sp_IN l_x;
  sp_OUT l_y;
  TID l_id;
end;
```

   ASME2SSME_ProcessImpl_Instantiate_Subprograms()

   ASME2SSME_Create_SharedSubprogram()

   ASME2SSME_SharedSubprogram_Call_Sync_x()

   ASME2SSME_SharedSubprogram_Sync_y()

3. case3:

```
| l_x ^= e1 ^+ ...
| l_y ^= l_id ^= l_x
where
  ...
  shared sp_IN l_x;
  sp_OUT l_y;
  TID l_id;
end;
```

ASME2SSME_ProcessImpl_SubThread_SubSubprograms_Clocks()

ASME2SSME_SubThread_SubSubprogram_Call_Sync_x()

ASME2SSME_SubThread_SubSubprogram_Sync_y()

For a sub thread *t1* *t1* has a subcomponent *sp* or requires/calls subprogram *sp*

1) *t1* interface:

- *t1* requires or calls subprogram in case 1 or 2: *t1(? . . . ; ! . . . ; event x;)*

  ASME2SSME_ComponentInterface_SubprogramAccesses()

  ASME2SSME_ComponentInterface_CallSubprograms()

- case 3: null

2) *t1* body:

- *t1* requires or calls subprogram in case 1 or 2: *(. . . , x) := t_behavior(. . .)*
- case 3:

```
| sp1: (l_y,l_id) := sp{}(l_x)

where
   ...
   process sp = ();
end;
```

ASME2SSME_ThreadImpl_SubSubprograms()

3) *t1_behavior* interface:

- *t1* requires or calls subprogram in case 1 or 2: *t1_behavior(? . . . ; ! . . . ; event x;)*
- case 3: null

4) *t1_behavior* body:

- *t1* requires or calls subprogram in case 1 or 2: *x := when Start*

  ASME2SSME_RequiresSubprogramAccesses_Clock_Def()

  ASME2SSME_SubprogramCalls_Clock_Def()

  Connection 1: a parameter connection from thread t1 to a called subprogram sp1 (if sp is not a subcomponent of t, case 1 or case 2): a $\rightarrow$ sp1.b: *L_x ::= a when sp1*

  ASME2SSME_ParameterConnection_Thread_to_CalledSubprogram()

  Connection 2: a parameter connection from a called subprogram sp1 to thread t1 (if sp is not a subcomponent of t, case 1 or case 2): sp1.a $\rightarrow$ b: *b := L_y.a when sp1*

  and remove b from interface of *behavior_external* process.

  ASME2SSME_ParameterConnection_CalledSubprogram_to_Thread()

- case 3: null

Note: if two (or more) threads share a same subprogram, pay attention to the calling clocks: they must be exclusive, otherwise, clock constraints will occur.

For a subprogram *sp* : A subprogram *sp* can call or require other subprogram *sp'*.
sp requires or calls subprogram *sp'* in case 1 or 2:
1) *sp* interface: *sp(? . . . ; ! . . . ; event x;)*
ASME2SSME_ComponentInterface_SubprogramAccesses()
ASME2SSME_ComponentInterface_CallSubprograms()
2) *sp* body: *(. . . , x) := sp_behavior(. . .)*
3) *sp_behavior* interface: *sp_behavior(? . . . ; ! . . . ; event x;)*

3.4 Subprogram group

Not implemented.

3.5 Thread

A thread implementation $t \in T$ is a tuple $< id_t, F, Y, Cal, C, P, M >$ where:

- id_t is the identifier of the thread implementation,
- F is the features (ports) of the corresponding thread type,
- Y is the subcomponents of t, where $\forall y_i \in Y$ is a subcomponent, $y_i =< id_{y_i}, y_i' >$, which is composed of an identifier id_{y_i} and its component classifier $y_i' \in D \cup Sp \cup Spg$ which is component of data D or subprogram Sp or subprogram group Spg,
- Cal is the subprogram call sequences,
- C is the port connections,

- P is the set of properties,

- M is a transition system that provides the functional behavior.

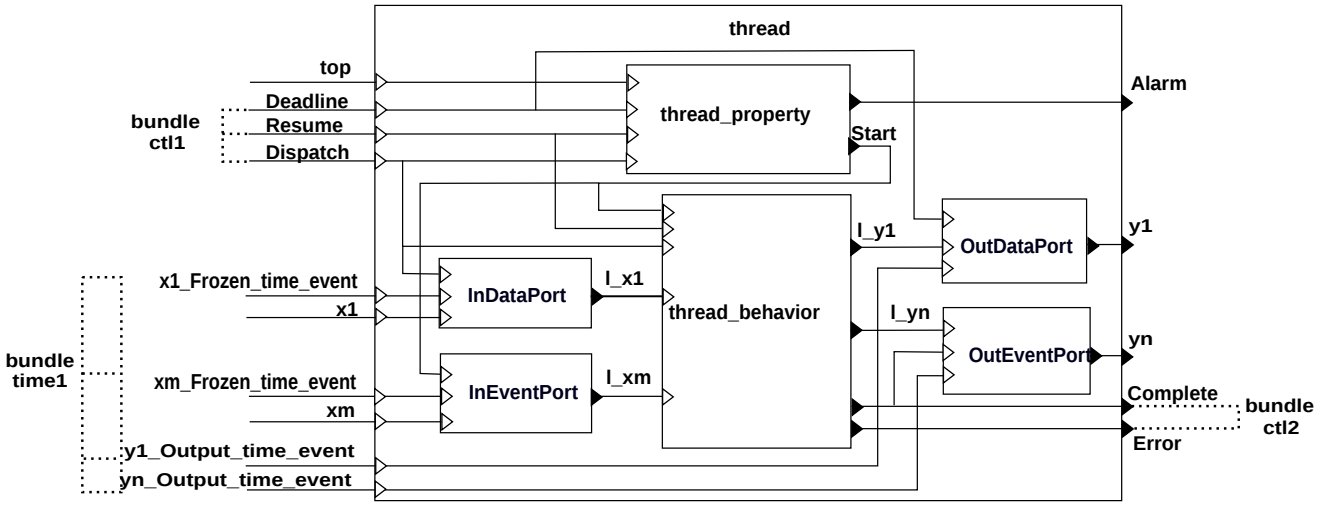

Figure 12. Thread transformation

An AADL thread component is implemented as a Signal process (Figure 12): it is composed of processes that represent its behavior, property, ports and subcomponents. **Dispatch**, **Complete** and **Error** are predeclared ports in AADL. They are represented as input/output signals ($Dispatch$, $Complete$, $Error$). According to the AADL semantics, the signals $Resume$ and $Deadline$ are added as inputs, which are generated by the scheduler. $Start$ is represented as the first $Resume$ after $Dispatch$ signal. It is computed in the $xx\_Thread\_property$ subprocess. The event signals ($x1\_Frozen\_time\_event$, $y1\_Output\_time\_event\ldots$) are represented as input signals, which are produced by the scheduler.

```
process xx_thread =
  (? x1, ... ;
    event Dispatch, Resume, Deadline;
    event x1_Frozen_time_event, ..., y1_Output_time_event, ...;
  ! y1, ...; event Complete, Error;)
     (| (...) := xx_thread_behavior(..., Dispatch, Start, Resume)
      | Start := xx_thread_property(Dispatch, Resume, Deadline)
      | x1_InDataPort{...}(...)
      | ...
      | y1_OutDataPort{...}(...)
      | ...
      | sxx1()
      | ...
      |)
  where
    event Start;
    process xx_thread_behavior(...);
    process xx_thread_property(?event Dispatch,Resume,Deadline; !Start;);
    process x1_InDataPort{...}(...);
    process y1_OutDataPort{...}(...);
```

```
    process sxx1();
    ...
  end;
```

An AADL thread $t \in T$ is translated to a Signal process id_t_Thread, ($\mathcal{T}^1(F)$ and $\mathcal{T}^2(F)$ represent the transformation of features in different cases):

$$
\begin{aligned}
\mathcal{T}(t) ::= \ & \textbf{process}\ id_t\_Thread\ = \\
& (\mathcal{T}^1(F); \\
& ?\ \textbf{event}\ top;\ CTL1\ ctl1;\ id_t\_TIME\_EVENT\ time1; \\
& !\ CTL2\ ctl2;\ \textbf{event}\ Alarm;) \\
& (|\mathcal{T}^2(F) \mid \mathcal{T}(Y) \mid \mathcal{T}(Cal) \mid \mathcal{T}(C) \mid \mathcal{T}(P) \mid \mathcal{T}(M)|)
\end{aligned}
$$

1. The **interface** contains the input/output signals that represent the features F provided by the component type, and also some added control signals: top, $ctl1$, $time1$, $ctl2$ and $Alarm$.

   The features $F = < f_1, f_2, \cdots >$ are firstly represented as input/output signals:

$$
\begin{aligned}
& \mathcal{T}^1(F) ::= \mathcal{T}^1(f_1); \mathcal{T}^1(f_2); \ldots; \\
& \forall f_i \in F, \texttt{apply the translation rule } \mathcal{T}(f_i) \texttt{ in Section 3.15}
\end{aligned}
$$

2. The **body** is composed of SIGNAL processes that represent:

   - ports F with timing semantics:

$$
\begin{aligned}
& \mathcal{T}^2(F) ::= (|\mathcal{T}^2(f_1) \mid \mathcal{T}^2(f_2) \mid \ldots|) \\
& \forall f_i \in F, \texttt{apply the translation rule } \mathcal{T}(f_i) \texttt{ in Section 3.15}
\end{aligned}
$$

   - subcomponents Y:

$$
\begin{aligned}
& \mathcal{T}(Y) ::= (|id_{y_1} :: \mathcal{T}(y_1') \mid id_{y_2} :: \mathcal{T}(y_2') \mid \ldots|) \\
& \forall y_i' \in D, \texttt{apply the translation rule } \mathcal{T}(y_i') \texttt{ in Section 3.2} \\
& \forall y_j' \in Sp, \texttt{apply the translation rule } \mathcal{T}(y_j') \texttt{ in Section 3.3} \\
& \forall y_k' \in Spg, \texttt{apply the translation rule } \mathcal{T}(y_k') \texttt{ in Section 3.4}
\end{aligned}
$$

- subprogram call sequences Cal:

$$\mathcal{T}(Cal) ::= (|\mathcal{T}(cal_1) \mid \mathcal{T}(cal_2) \mid \ldots|)$$
$$\forall cal_i \in Cal, \texttt{ apply the translation rule } \mathcal{T}(cal_i) \texttt{ in Section 3.3}$$

- connections C:

$$\mathcal{T}(C) ::= (|\mathcal{T}(c_1) \mid \mathcal{T}(c_2) \mid \ldots|)$$
$$\forall c_i \in C, \texttt{ apply the translation rule } \mathcal{T}(c_i) \texttt{ in Section 3.16}$$

- associated properties P:

$$\mathcal{T}(P) ::= \textbf{process } id_x\_Thread\_property = ()$$
$$(|\mathcal{T}(p_1) \mid \mathcal{T}(p_2) \mid \ldots|)$$
$$\forall p_i \in P, \texttt{ apply the translation rule } \mathcal{T}(p_i) \texttt{ in Section 3.19}$$

- transition system M. We define a transition system M to be a tuple $< S', T', s_0, C', A >$, where S' is a set of states, $T' \subseteq S' \times C' \times S' \times A$ is the set of transitions between states S', triggered by the condition C' with functional behavior A, and s_0 an initial state:

$$\mathcal{T}(M) ::= \textbf{process } id_x\_Thread\_behavior = ()$$
$$(|\mathcal{T}(S' \times C' \times S' \times A)|)$$
$$\texttt{apply the translation rule } \mathcal{T}(S' \times C' \times S' \times A) \texttt{ in Section 4}$$

Referring to the SDSCS example, the thread *doors_mix* is as follows:

```
thread doors_mix
  features
    cll1: in data port CESAR_behavior::integer;
    cll2: in data port CESAR_behavior::integer;
    cll: out data port CESAR_behavior::integer;
end doors_mix;

thread implementation doors_mix.imp
  properties
    Dispatch_Protocol => Periodic;
    Period => 50 Ms;
end doors_mix.imp;
```

We get the following Signal code from our transformation (the details for the data port subprocesses are not given here):

```
process doors_mix_imp_Thread =
( ? integer cll1, cll2; CTL1 ctl1; doors_mix_TIME_EVENT time1; event top;
  ! integer cll; CTL2 ctl2; boolean Alarm;)
(| l_cll1 := cll1_InDataPort{integer}(time1.cll1_Frozen_time_event,cll1,Start)
 | l_cll2 := cll2_InDataPort{integer}(time1.cll2_Frozen_time_event,cll2,Dispatch)
 | cll := cll_OutDataPort{integer}(time1.cll_Output_time_event,l_cll,Deadline)
 | (l_cll,ctl2) := doors_mix_imp_Thread_behavior{}
                   (l_cll1,l_cll2,time1.Dispatch,Start,time1.Resume)
 | (Start, Alarm) := doors_mix_imp_Thread_property{}(ctl1, top)
 |)
 where
   event Start;
   integer l_cll, l_cll2, l_cll1;
   process cll1_InDataPort = {...} ( ? ... ! ...) (|...|)
   process cll2_InDataPort = {...} ( ? ... ! ...) (|...|)
   process cll_OutDataPort = {...} ( ? ... ! ...) (|...|)
   process doors_mix_imp_Thread_behavior =
   (? integer cll1, cll2; CTL1 ctl1; ! integer cll; CTL2 ctl2;)(|...|);
   process doors_mix_imp_Thread_property =
   ( ? CTL1 ctl1; event top; ! event Start; boolean Alarm;)(| ... |);
 end;
```

- The status of thread transformation in Signal is given in Table 6.

thread type	features	port	partly	2011-07-22
		feature group	partly	2012-02-03
		data access	partly	2012-02-03
		subprogram access	partly	2012-03-29
		subprogram group access	none	
	flow		none	
	mode		expected	
thread implementation	subcomponents	data	partly	2012-02-03
		subprogram	partly	2011-07-22
		subprogram group	none	
	subprogram call		partly	2012-03-29
	connection		partly	2011-07-22
	flow		none	
	mode		expected	

Table 6. The thread transformation

- The thread properties are listed in the ESPRESSO AADL Digest Report [9], Section 6.3. They are implemented as process calls in the $xx\_Thread\_property()$ subprocess. The values of **Dispatch_Protocol**, **Period** and **Deadline** properties are fetched, and they appear in the process calls as parameters. Getting values for other properties is left to do.

Some properties, e.g., **Dispatch_Protocol**, are chosen as fixed properties. The default value is used if it is not specified explicitly in the AADL specification. For example, the properties of the thread *doors_mix.impl* are implemented as a Signal process. The values of properties **Dispatch_Protocol** and **Period** are listed as parameters of the corresponding process call:

```
process doors_mix_imp_Thread_property =
( ? CTL1 ctl1;%control signal of bundle type CTL1 (Dispatch, Resume, Deadline)%
    event top;
  ! event Start; boolean Alarm;)
(| Start := when ((ctl1.Resume from ctl1.Dispatch)=1)
 | Alarm := PeriodicDispatch_constraint{50}(top, ctl1.Dispatch)
 | Dispatch_Protocol_property{Supported_Dispatch_Protocols#Periodic}()
 | Period_property{Time_Units#ms,50}()
 |) %doors_mix_imp_Thread_property%;
```

The six types of threads (**periodic, aperiodic, sporadic, timed, hybrid, background**) are discriminated by the dispatch: a dispatch request is periodicly, or is triggered by an event (event data) arriving, etc.. Therefore, the different threads are implemented in the same way, except the "dispatch" signal is generated differently.

3.6 Thread group

Not implemented.

3.7 Process

A process implementation $x \in Ps$ is a tuple $< id_x, F, C, P, Y, M >$ where:

- id_x is the identifier of the process implementation,
- F is the features (ports) of the corresponding process type that define a functional interface,
- $C = F \times F$ is the port connection,
- P is the set of properties,
- Y is the subcomponents of x, where $\forall y_i \in Y$ is a subcomponent, $y_i =< id_i, y_i' >$, which is composed of an identifier id_i and its component classifier $y_i' \in D \cup T \cup Tg \cup Sp$ which can be type of data D, thread T, thread group Tg or subprogram Sp.
- M is a transition system that provides the functional behavior.

An AADL process component x is implemented as a Signal process model. It is represented by a process type $TT\_id_x\_Process()$, and an implementation in $id_x\_Process()$:

$$\mathcal{T}(x) ::= (|TT\_id_x\_Process()|\ id_x\_Process()|)$$

For a process model type $TT\_id_x\_Process()$, beside the in/out ports declared in the original AADL process type, the Signal process interface should also include the clock signals for each of its threads t_i:

$$\begin{aligned}&\textbf{process } TT\_id_x\_Process() = \\ &\quad (\mathcal{T}^1(F); \\ &\quad ?\ event\ top;\ CTL1\ t1\_ctl1;\ TIME1\ time1;\ \ldots \\ &\quad !\ CTL2\ t1\_ctl2;\ event\ t1\_Alarm;\ \ldots)\end{aligned}$$

The process implementation $id_x\_Process()$ is composed of sub-processes, i.e., $id_x\_Process\_behavior()$, $id_x\_Process\_property()$:

$$\begin{aligned}&\textbf{process } id_x\_Process() = \ TT\_id_x\_Process() \\ &\qquad (|\mathcal{T}(P) \mid \mathcal{T}(Y) \mid \mathcal{T}(C) \mid \mathcal{T}(M)|)\end{aligned}$$

- associated properties P:

$$\begin{aligned}&\mathcal{T}(P) ::= \textbf{process } id_x\_Process\_property = () \\ &\qquad (|\mathcal{T}(p_1) \mid \mathcal{T}(p_2) \mid \ldots|) \\ &\forall p_i \in P,\ \mathcal{T}(p_i) ::= \textbf{process } p_i\_property = ()()\end{aligned}$$

  Each property $\forall p_i \in P$ is represented as a Signal process.

- subcomponents Y:

$$\mathcal{T}(Y) ::= (|id_{y_1} :: \mathcal{T}(y_1') \mid id_{y_2} :: \mathcal{T}(y_2') \mid \ldots|)$$

  For $\forall y_i \in Y$, apply the recursive translation rules for its classifier (component implementation) $\mathcal{T}(y_i')$.

- connections $C = F \times F$:

$$\begin{aligned}&\mathcal{T}(C) ::= (|\mathcal{T}(c_1) \mid \mathcal{T}(c_2) \mid \ldots|) \\ &\texttt{where } \forall c_i =< f_m, f_n >\in C, \mathcal{T}(c_i) ::= \mathcal{T}(f_n) := \mathcal{T}(f_m)\end{aligned}$$

  For $\forall c_i \in C$, connects the two signals that represent the source and destination ports.

- transition system M:

$$\mathcal{T}(M) ::= \textbf{process}\ id_x\_Process\_behavior = (...)(...)$$

Referring to the SDSCS example, we have a *doors_process* process which has three threads (*door_handler1*, *door_handler2* and *door_mix*) as subcomponents:

```
process doors_process
  features
    closed1_1: in data port CESAR_behavior::integer;
    closed1_2: in data port CESAR_behavior::integer;
    in_flight: in data port CESAR_behavior::integer;
    cll: out data port CESAR_behavior::integer;
end doors_process;

process implementation doors_process.imp
  subcomponents
    door_handler1: thread door_handler.imp;
    door_handler2: thread door_handler.imp;
    door_mix: thread doors_mix.imp;
  connections
    conn3: port in_flight -> door_handler1.in_flight;
    conn10: port closed1_1 -> door_handler1.closed1;
    conn20: port door_handler1.cll -> door_mix.cll1;
    conn23: port in_flight -> door_handler2.in_flight;
    conn30: port closed1_2 -> door_handler2.closed1;
    conn40: port door_handler2.cll -> door_mix.cll2;
    conn41: port door_mix.cll -> cll;
end doors_process.imp;
```

The implemented transformation provides the following Signal code, that contains a process type (*TT_doors_process_imp_Process*) and a process definition (*doors_process_imp_Process*):

```
type process TT_doors_process_imp_Process =
  ( ? event top;
      integer closed1_1, closed1_2, in_flight;
      CTL1 door_handler1_ctl1;
      door_handler_TIME_EVENT door_handler1_time;
      CTL1 door_handler2_ctl1;
      door_handler_TIME_EVENT door_handler2_time;
      CTL1 door_mix_ctl1;
      doors_mix_TIME_EVENT door_mix_time;
    ! integer cll;
      CTL2 door_handler1_ctl2;
      boolean door_handler1_Alarm;
      CTL2 door_handler2_ctl2;
      boolean door_handler2_Alarm;
      CTL2 door_mix_ctl2;
      boolean door_mix_Alarm;) ;
```

```
process doors_process_imp_Process = TT_doors_process_imp_Process
  (| doors_process_imp_Process_behavior()
   | doors_process_imp_Process_property()
   | (L_56,door_handler1_ctl2, door_handler1_Alarm) := door_handler_imp_Thread{}
               (L_57, L_58, door_handler1_ctl1, door_handler1_time1, top)
   | (L_59,door_handler2_ctl2, door_handler2_Alarm) := door_handler_imp_Thread{}
               (L_60, L_61, door_handler2_ctl1, door_handler2_time1, top)
   | (L_62,door_mix_ctl2, door_mix_Alarm) := doors_mix_imp_Thread{}
               (L_63, L_64, door_mix_ctl1, door_mix_time1, top)
   | L_58 := in_flight
   | L_57 := closed1_1
   | L_63 := L_56
   | L_61 := in_flight
   | L_60 := closed1_2
   | L_64 := L_59
   | cll := L_62
   |)
 where
   integer L_62, L_64, L_63, L_59, L_56, L_61, L_60, L_58, L_57;
   process doors_process_imp_Process_behavior =( );
   process doors_process_imp_Process_property =( );
   process door_handler_imp_Thread = ...
   process doors_mix_imp_Thread = ...
 end;
```

- The implementation status of process type features is given in Table 7.

process type	features	port	partly	2011-07-22
		feature group	partly	2012-02-03
		data access	expected	
		subprogram access	expected	
		subprogram group access	none	
	flow		none	
	mode		expected	
process implementation	subcomponents	data	partly	2012-02-03
		subprogram	partly	2012-03-29
		thread	partly	2011-07-22
		thread group	none	
	connection		partly	2011-07-22
	flow		none	
	mode		expected	

Table 7. The process implementation status

- All the standard process properties are listed in the ESPRESSO AADL Digest Report [9], Section 8.3. They are implemented as process calls in the $xx\_Process\_property()$ process. Only the name of the property is given. The values are to be fetched.

3.8 Processor

An AADL processor is represented as a Signal process (Figure 13): the AADL processes bound to a processor are provided as parameters of the **interface** of the processor, e.g., {*process TT_p_Process p_Process;*}, where *TT_p_Process* is the type of the process *p_Process*.

The **interface** contains the original inputs/outputs of these AADL processes except for the internal inputs/outputs between these processes of which are defined as internal local signals.

The **body** is composed of the Signal processes that represent the (AADL) processes and $xx\_Processor\_behavior()$, $xx\_Processor\_property()$ subprocesses. The $xx\_Processor\_behavior()$ subprocess is implemented as a scheduler. It gives all the scheduling signals for each thread, e.g., $p1\_t1\_Dispatch$ ($p1$: subprocess name, $t1$: thread name), $p2\_t2\_y2\_Output\_time\_event\ldots$ In the current implementation, it is **connected to a static scheduler**: the information of timing properties of each thread is passed to the scheduler, and the scheduling result is automatically generated by the connected scheduler.

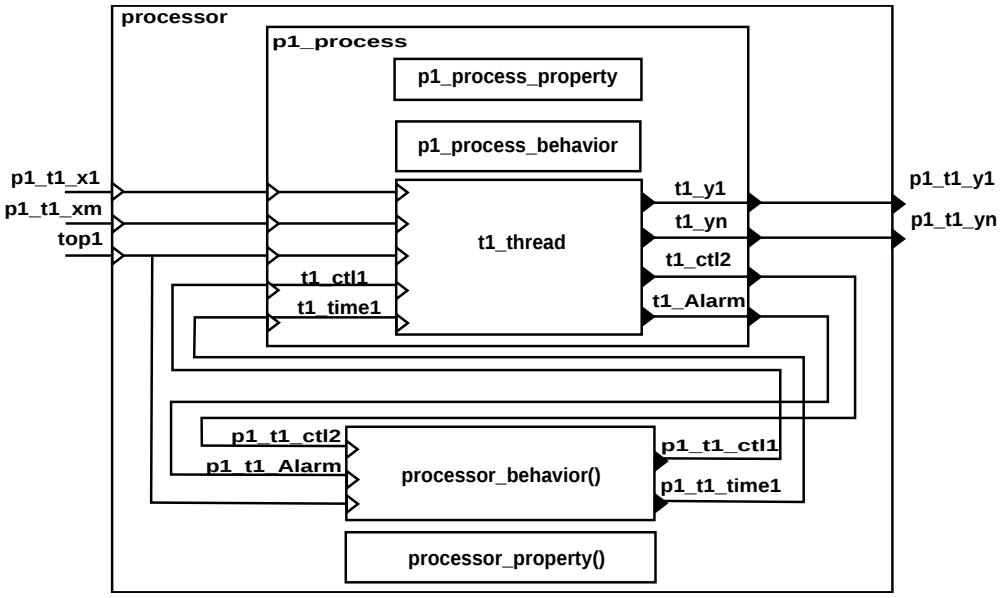

Figure 13. Processsor transformation

For example: a process subcomponent *p1* (whose type is *p*, implementation is *p.imp*) is bound to a processor subcomponent *cpiom1*—whose implementation is *CPIOM* (this information is specified as a property in the system level). The Signal representation is as follows:

```
process cpiom1_Processor = {process TT_p_Process P;}
    (? event top; p1_t1_x1; p1_t1_xm; ! p1_t1_y1; p1_t1_yn;)
    (| (...) := P{}(...)
     | cpiom1_behavior{}
     | cpiom1_property{}
    |)
    where
        ...
        process  cpiom1_Processor_behavior();
```

```
        process  cpiom1_Processor_property();
     end;
```

The process call should be in the following form:

```
(p1_t1_y1, p1_t1_yn) := cpiom1_Processor{p_imp_Process}(top, p1_t1_x1, p1_t1_xm)
```

Referring to the SDSCS system example, there are two processors: *cpiom1* and *cpiom2*. Each processor executes a process. We give part of the generated Signal code for processor *cpiom1*.

```
process cpiom1_Processor ={ process TT_doors_process_imp_Process doors_process1; }
(? event top;
   integer doors_process1_closed1_1, doors_process1_closed1_2, doors_process1_in_flight;
 ! integer doors_process1_cll;)
(|(L_114,
   doors_process1_door_handler1_ctl2, doors_process1_door_handler1_Alarm,
   doors_process1_door_handler2_ctl2, doors_process1_door_handler2_Alarm,
   doors_process1_door_mix_ctl2, doors_process1_door_mix_Alarm)
   := doors_process1{}(top, L_91, L_92, L_93,
   doors_process1_door_handler1_ctl1, doors_process1_door_handler1_time1,
   doors_process1_door_handler2_ctl1, doors_process1_door_handler2_time1,
   doors_process1_door_mix_ctl1, doors_process1_door_mix_time1)
 | cpiom1_Processor_behavior{}
 | cpiom1_Processor_property{}
 |)
 where
   ...
   process cpiom1_Processor_behavior =
   ( ? event top;
       CTL2 doors_process1_door_handler1_ctl2;
       boolean doors_process1_door_handler1_Alarm;
       CTL2 doors_process1_door_handler2_ctl2;
       boolean doors_process1_door_handler2_Alarm;
       CTL2 doors_process1_door_mix_ctl2;
       boolean doors_process1_door_mix_Alarm;
     ! CTL1 doors_process1_door_handler1_Dispatch;
       door_handler_TIME_EVENT doors_process1_door_handler1_time1;
       CTL1 doors_process1_door_handler2_Dispatch;
       door_handler_TIME_EVENT doors_process1_door_handler2_time1;
       CTL1 doors_process1_door_mix_Dispatch;
       door_mix_TIME_EVENT doors_process1_door_mix_time1;)
     (|...|);
     process cpiom1_Processor_property =()(|...|);
 end;
```

- The implementation status of processor is given in Table 8.
- The standard processor properties are listed in the ESPRESSO AADL Digest Report [9], Section 9.1.2. They are implemented as process calls in the $xx\_Processor\_property()$ subprocess. The values are to be fetched.

processor type	features	port	partly	2011-07-22
		feature group	partly	2012-02-03
		bus access	none	
		subprogram access	none	
		subprogram group access	none	
	flow		none	
	mode		expected	
processor implementation	subcomponents	memory	partly	2012-02-03
		bus	partly	2011-07-22
		virtual processor	none	
		virtual bus	none	
	connection		open	
	flow		none	
	mode		expected	

Table 8. The processor implementation status

3.9 Virtual processor

Not implemented.

- The status of virtual processor is given in Table 9.

virtual processor type	features	port	none	
		feature group	none	
		subprogram access	none	
		subprogram group access	none	
	flow		none	
	mode		expected	
virtual processor implementation	subcomponents	virtual processor	none	
		virtual bus	none	
	flow		none	
	mode		expected	

Table 9. The virtual processor status

- The standard virtual processor properties are listed in the ESPRESSO AADL Digest Report [9], Section 9.2.2.

3.10 Memory

Implemented date 2012-02-03
A Signal process skeleton represents a AADL memory.

memory type	features mode	feature group bus access	none none expected	
memory implementation	subcomponents connection mode	memory bus	none none open expected	

Table 10. The memory status

- The status of memory implementation is given in Table 10.
- Standard memory properties are listed in Table 11.

Provided_Virtual_Bus_Class	none
Provided_Connection_QUality_Of_Service	none
Memory_Protocol	none
Resumption_Policy	none
Byte_Count	none
Word_Size	none
Word_Space	none
Write_Time	none
Source_Text	none
Hardware_Description_Source_Text	none
Hardware_Source_Language	none
Implemented_As	none

Table 11. The standard memory properties

3.11 Bus

An AADL bus component is implemented as a Signal process. A skeleton is provided. It is composed of $xx\_Bus\_behavior()$, $xx\_Bus\_property()$.

- The status of bus implementation is given in Table 12.
- Standard bus properties (Table 13) are represented as process calls in $xx\_Bus\_property()$ subprocess (an API is needed to get the property values).

3.12 Virtual bus

Not implemented.

bus type	features mode	feature group requires bus access	none none expected	
bus implementation	subcomponents connection mode	virtual bus	none open expected	

Table 12. The bus status

Provided_Connection_Quality_Of_Service	partly
Allowed_Connection_Type	partly
Allowed_Physical_Access_Class	partly
Allowed_Physical_Access	partly
Resumption_Policy	partly
Transmission_Type	partly
Transmission_Time	partly
Latency	partly
Access_Right	partly
Allowed_Message_Size	partly
Source_Language	partly
Source_Text	part
Hardware_Description_Source_Text	partly
Hardware_Source_Language	partly
Implemented_As	partly

Table 13. The standard bus properties

3.13 Device

An AADL device component is represented as a Signal process. A process skeleton is provided. The interface contains the signals which represent the port features (the other features are not implemented yet). The body is composed of $xx\_Device\_behavior()$ and $xx\_Device\_property()$ subprocesses.

- The status of device is given in Table 14.
- The standard device properties are listed in the ESPRESSO AADL Digest Report [9], Section 9.6.2. They are implemented as process calls in the $xx\_Device\_property()$ subprocess. The **Device_Dispatch_Protocol** and **Period** property values are fetched, and represented as parameters in the corresponding process call.

3.14 System

A system $x \in S =< id_x, F, C, P, Y, M >$ specifies the runtime architecture of an operational physical system, where:

device type	features	port	partly	2011-07-22
		feature group	none	
		bus access	none	
		provides subprogram access	none	
		provides subprogram group access	none	
	flow		none	
	mode		expected	
device implementation	subcomponents	bus	none	
		virtual bus	none	
	connection		none	
	flow		none	
	mode		expected	

Table 14. The device status

- id_x is the identifier of the system implementation,
- F is the features (ports) of the corresponding system type that define a functional interface,
- $C = F \times F$ is the port connection,
- P is the set of properties,
- Y is the subcomponents of x, where $\forall y_i \in Y$ is a subcomponent, $y_i =< id_i, y_i' >$, which is composed of an identifier id_i and its component classifier $y_i' \in D \cup Ps \cup Pr \cup Vpr \cup Mo \cup B \cup Vb \cup Dv \cup S$ which is component of data D, process Ps, processor Pr, virtual processor Vpr, memory Mo, bus B, virtual bus Vb, device Dv or system S,
- M is a transition system that provides the functional behavior.

A system x is represented as a Signal process:

$$\begin{aligned}\mathcal{T}(x) ::= \ & \textbf{process}\ id_x\_System\ = \\ & (\mathcal{T}^1(F);\ ?\ event\ top;) \\ & (|\mathcal{T}(P) \mid \mathcal{T}(Y) \mid \mathcal{T}(C) \mid \mathcal{T}(M)|)\end{aligned}$$

The **body** is composed of SIGNAL processes that represent:

- associated properties P:

$$\begin{aligned}\mathcal{T}(P) ::= & \textbf{process}\ id_x\_System\_property = () \\ & (|\mathcal{T}(p_1) \mid \mathcal{T}(p_2) \mid \ \ldots\ |)\end{aligned}$$

For each property $\forall p_i \in P$, apply the transformation rule in Section 3.19.

- subcomponents Y:

$$\mathcal{T}(Y) ::= (|id_{y_1} :: \mathcal{T}(y_1') \mid id_{y_2} :: \mathcal{T}(y_2') \mid \ldots |)$$

For $\forall y_i \in Y$, apply the recursive translation rules for its classifier (component implementation) $\mathcal{T}(y_i')$.

- connections $C = F \times F$:

$$\mathcal{T}(C) ::= (|\mathcal{T}(c_1) \mid \mathcal{T}(c_2) \mid \ldots |)$$

For $\forall c_i \in C$, apply the transformation rule in Section 3.16.

- transition system M: apply the transformation rule in Section 4.

A system is represented in the following way:

```
process xx_System = (? ... ; event top1; ! ...)
  (| xx_System_behavior()
   | xx_System_property()
   | (...) := cpiom1_Processor{p_imp_Process}(top1, ...)
   | d1_Device()
   | ...
   | cxx1_Connection()
   | ...
   |)
 where
      process  xx_Process_behavior();
      process  xx_Process_property();
      type process TT_p_Process = (? ...; ! ...;);
      process p_imp_Process = TT_p_Process();
      process cpiom1_Processor = {process TT_p_imp_Process P;}(? ...; !...;)();
      process d1_Device();
      process cxx1_Connection();
      ...
 end;
```

Referring to the SDSCS example, the top level system component is represented as follows in Signal (details of subprocesses are not shown here):

```
process simple_door_management_imp_System =
( ? integer in_flight; event top1, top2; ! integer cll1, cll2;)
 (| simple_door_management_imp_System_behavior()
  | simple_door_management_imp_System_property()
  | L_19 := Door_imp_System{}()
  | L_20 := Door_imp_System{}()
```

```
  | L_21 := cpiom1_Processor{doors_process_imp_Process}(top1, L_22, L_23, L_24)
  | L_25 := cpiom2_Processor{doors_process_imp_Process}(top2, L_26, L_27, L_28)
  | RDC_imp_Device{}()
  | RDC_imp_Device{}()
  | AFDX_imp_Bus{}()
  | L_24 := in_flight
  | cll1 := L_21
  | L_22 := L_19
  | L_23 := L_20
  | L_28 := in_flight
  | cll2 := L_25
  | L_26 := L_19
  | L_27 := L_20
  |)
where
  ...
end;
```

- The status of system is given in Table 15.

system type	features	port	partly	2012-02-03
		feature group	partly	2012-02-03
		bus access	none	
		data access	none	
		subprogram access	none	
		subprogram group access	none	
	flow		none	
	mode		expected	
system implementation	subcomponents	data	expected	
		process	partly	2011-07-22
		processor	partly	2011-07-22
		virtual processor	none	
		memory	partly	2012-02-03
		bus	partly	2011-07-22
		virtual bus	none	
		device	partly	2011-07-22
		system	partly	2011-07-22
	connection		partly	2011-07-22
	flow		none	
	mode		expected	

Table 15. The system status

- The standard system properties are listed in the ESPRESSO AADL Digest Report [9], Section 10.2. They are implemented as process calls in the $xx\_System\_property()$ subprocess. The values are to be fetched. The API functions to get values of **Actual_Processor_Processor_Binding** and **Actual_Connection_Binding** properties have already been implemented.

3.15 Feature

3.15.1 Port

In the current implementation, the (in out) direction port has not been implemented. (Table 16).

Data port	Event port	Event data port
partly	partly	partly
2011-07-22	2011-10-20	2011-10-21

Table 16. The ports

A port $f =< id_f,\ direction_f,\ category_f,\ classifier_f,\ P >\in F$ is composed of:

- id_f is the identifier of the port,
- $direction_f \in$ **enumeration** {`in, out, inout`} is the direction of the port,
- $category_f \in$ **enumeration** {`event, data, event data`} is the port category,
- $classifier_f \in D$ is the classifier (type) of data (event data) port, which is a data component,
- P is the set of properties.

In the current implementation, we mainly consider two cases of context: port an an interface, and port of thread of which timing semantics are considered.

Case 1: port of component as an interface Port is a communication interface for the directional exchange of data, events, or both between components. In this case, neither timing semantics nor properties are considered. It is represented directly as an input or output signal:

- **Case 1.1: if** $category_f = event$:

$$\mathcal{T}(f) ::= \begin{cases} ?\ \textbf{event}\ id_f; & \texttt{if}\ \ direction_f = in \\ !\ \textbf{event}\ id_f; & \texttt{if}\ \ direction_f = out \end{cases}$$

- **Case 1.2: if** $category_f = data$ **or** $category_f = event\ data$:

$$\mathcal{T}(f) ::= \begin{cases} ?\ \mathcal{T}(classifier_f)\ id_f; & \texttt{if}\ \ direction_f = in \\ !\ \mathcal{T}(classifier_f)\ id_f; & \texttt{if}\ \ direction_f = out \end{cases}$$

Case 2: port of thread with timing semantics

- **Case 2.1: if** $category_f = data$

1. **Case 2.1.1: if** $direction_f = in$
   An AADL in data port f is implemented as a Signal process model, composed of $id_f\_InDataPort\_behavior()$ and $id_f\_InDataPort\_property()$ subprocesses:

$\mathcal{T}(f)$ ::= **process** $id_f\_InDataPort = \{$**type** $item\_type;\}$
(? **event** $Frozen\_time\_event$; $item\_type\ write\_flow$; **event** $Reference\_time\_event$;
! $item\_type\ read\_flow$;)
(| $read\_flow := id_f\_InDataPort\_behavior\{item\_type\}$
$(Frozen\_time\_event,\ write\_flow)$
| $id_f\_InDataPort\_Property\{\}(Frozen\_time\_event,\ Reference\_time\_event)$
|)
where
process $id_f\_InDataPort\_behavior = \{$**type** $item\_type;\}$
(? **event** $Frozen\_time\_event$; $item\_type\ write\_flow$; ! $item\_type\ read\_flow$;)
(| $read\_flow\ :=\ InDataPort\_behavior\{item\_type\}$
$(Frozen\_time\_event,\ write\_flow)$|);
process $id_f\_InDataPort\_Property =$
(? **event** $Frozen\_time\_event,\ Reference\_time\_event$;)
(|$\mathcal{T}(p_1)$ | $\mathcal{T}(p_2)$ | ...|);
end;

   - The translation of port classifer $\mathcal{T}(classifier_f)$ is passed as parameter when a process instance is made in the body of process that represents the thread.
   - Process $id_f\_InDataPort\_behavior()$ calls the $InDataPort\_behavior()$ process which is defined in the **AADL_DATAPORT** Signal library.
   - Process $id_f\_InDataPort\_property()$ is composed of processes that represent the properties $\forall p_i \in P$ applying to this port, i.e., **Input_Time**. In the current implementation, the **Input_Time** property (**Output_Time** for the out data port) is partly implemented: **the property values are completely fetched and provided as parameters, and a Signal process is defined in the library to verify whether the actual Input_Time (Output_Time) signal satifies the constraints** (external C code should be provided to complete this process for the verification), e.g.:
     $Input\_Time\_property\{Time\_Units\#ns,\ Time\_Units\#ns,\ 0.0, 0.0\}(\ldots)$
     If this property is not explicitly specified in the AADL model, default values are used for the process call.

2. **Case 2.1.2: if** $direction_f = out$

   An AADL out data port f is implemented as a Signal process model, composed of $id_f\_OutDataPort\_behavior()$ and $id_f\_OutDataPort\_property()$ subprocesses:

$\mathcal{T}(f)$ ::= **process** $id_f\_OutDataPort = \{$**type** $item\_type;\}$
(? **event** $Output\_time\_event$; $item\_type\ write\_flow$; **event** $Reference\_time\_event$;
! $item\_type\ sent\_flow$;)
(| $sent\_flow := id_f\_OutDataPort\_behavior\{item\_type\}$
$(Output\_time\_event,\ write\_flow)$
| $id_f\_OutDataPort\_Property\{\}(Output\_time\_event,\ Reference\_time\_event)$
|)
where
process $id_f\_OutDataPort\_behavior = \{$**type** $item\_type;\}$
(? **event** $Output\_time\_event$; $item\_type\ write\_flow$; ! $item\_type\ sent\_flow$;)
(| $sent\_flow := OutDataPort\_behavior\{item\_type\}$
$(Output\_time\_event,\ write\_flow)$|);
process $id_f\_OutDataPort\_Property =$
(? **event** $Output\_time\_event,\ Reference\_time\_event$;)
(|$\mathcal{T}(p_1)$ | $\mathcal{T}(p_2)$ | ...|);
end;

The **properties** associated with a data port have been listed in the ESPRESSO AADL Digest Report [9]. Since we mainly take into account the properties related to **timing aspects**, here we give a list of timing properties that have been implemented in the current version (Table 17).

Property	In data port	Out data port
Input_Time	partly (2011-07-22)	-
Input_Rate	none	-
Output_Time	-	partly (2011-07-22)
Output_Rate	-	none
Timing	partly (2011-07-22)	partly (2011-07-22)
Fan_Out_Policy	-	partly (2011-07-22)

Table 17. The implemented data port properties

The **Input_Rate** and **Output_Rate** are associated with **Input_Time** and **Output_Time**, respectively. They must conform to the rate of the input (or output) time. Since we have translated the Input_Time and Output_Time properties, the two rate properties should be useless. The external C code should be provided for the verification of Input_Time (Output_Time) constraint.

- **Case 2.2: if** $category_f = event\ data$ **or** $category_f = event$**:**

  Event data ports are intended for message transmission, and event ports are intended for event and alarm transmission. In this document, we give the transformation details of event data port. The implementation of event port is similar to event data port.

  An event data (or event) port can have a queue associated with it. The default port queue size is 1 and can be changed by explicitly declaring a *Queue_Size* property association for the port. Queues will be serviced according to the *Queue_Processing_Protocol*, by default in a first in first out order (FIFO).

  The properties that should be taken into account are listed in Table 18.

Property	In event (event data) port	Out event (event data) port
Input_Time	partly (2011-10-20)	-
Input_Rate	none	-
Output_Time	-	partly (2011-10-20)
Output_Rate	-	none
Fan_Out_Policy	-	expected
Overflow_Handling_Protocol	total (2011-10-20)	-
Queue_Size	total (2011-10-20)	total (2011-10-20)
Queue_Processing_Protocol	total (2011-10-20)	total (2011-10-20)
Dequeued_Items	total (2011-10-20)	-
Dequeue_Protocol	total (2011-10-20)	-

Table 18. The properties to be implemented for an event (event data) port

  For an in event data (or event) port, the items are frozen and *Input_Time*, and a number of items (depend on the *Dequeue_Protocol*) are dequeued and made available to the receiving application through the port variable (implemented as constraint in the property part).

  For an out event data (or event) port, the items are stored in a queue, and sent out at *Output_Time*.

  - **Case 2.2.1: if** $direction_f = in$

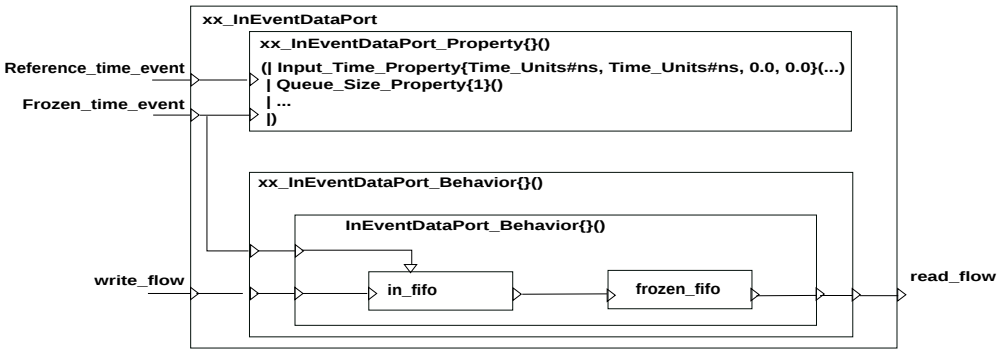

Figure 14. In event data port transformation

An AADL in event data (or event) port f can be implemented as a Signal process model, composed of $xx\_InEventDataPort\_Behavior()$ and $xx\_InEventDataPort\_Property()$ subprocesses (Figure 14).

$xx\_InEventDataPort\_Behavior()$ calls the $InEventDataPort\_Behavior()$ process which is defined in the **AADL_EVENTDATAPORT** Signal library. (In the current version, only FIFO is implemented. The other queue processing order is expected.) Two FIFOs are provided: **in_fifo** that storing the receiving in event data and **frozen_fifo** (only one element for the current implementation) that is accessible by the thread. At **Frozen_time_event** (**Input_Time**), freeze the actual items of the **in_fifo**: move certain items (move one item for the current version) to a **frozen_fifo**. The inputs arrived after the **Frozen_time_event** will be available at the next **Frozen_time_event**.

The properties are implemented as Signal processes composed in the body of $xx\_InEventDataPort\_Property()$. External C code should be provided to verify whether the constraints are satisfied.

– **Case 2.2.2: if** $direction_f = out$

An AADL out data port *xx* is implemented as a Signal process model, composed of $xx\_OutEventDataPort\_Behavior()$ and $xx\_OutEventDataPort\_Property()$ subprocesses (Figure 15).

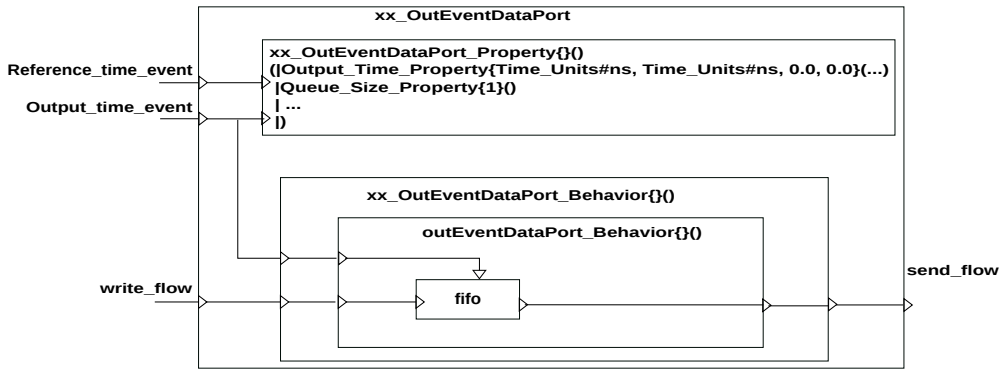

Figure 15. Out event data port transformation

$OutEventDataPort\_Behavior()$ is defined in the **AADL_EVENTDATAPORT** Signal library: the output data is stored into a **fifo** and one element is sent out at **Output_time_event** (**Output_Time**).

3.15.2 Parameter

A subprogram parameter declaration represents data value that can be passed into or out of a subprogram. A parameter is typed with a data classifier reference representing the data type.

A parameter $f =< id_f,\ direction_f,\ classifier_f,\ P >\in F$ is composed of:

- id_f is the identifier of the parameter,

- $direction_f \in$ **enumeration** {`in, out, inout`} is the direction of the parameter,

- $classifier_f \in D$ is the classifier (type) of data,

- P is the set of properties.

In the Signal translation, a parameter declaration is represented as a signal declaration. This signal is typed with the data classifier. The properties are not considered.

$$\mathcal{T}(f) ::= \begin{cases} ?\,\mathcal{T}(classifier_f)\,id_f; & \texttt{if}\ \ direction_f = in \\ !\,\mathcal{T}(classifier_f)\,id_f; & \texttt{if}\ \ direction_f = out \end{cases}$$

In the following example, the subprogram *acquire* contains an in parameter *p1* whose data classifier is *DataA*, and an out parameter *p2* whose data classifier is *DataB.imp*:

```
subprogram acquire
  features
    p1: in parameter DataA;
    p2: out parameter DataB.imp;
end acquire;
```

These two parameters are represented as input (resp., output) in the interface of the *acquire_Subprogram()* process. Their type is the corresponding data type of the data classifier.

```
process acquire_Subprogram =(? DataA p1; ! DataB_imp p2;)(| ... |);
```

3.15.3 Feature group

Port group In AADLv1, a port group represents a collection of ports or other port groups. The content and structure of a port group are declared through a **port group type** declaration. A port group can bundle different port types and directions. A port group type can be declared as the **inverse of** another port group type. The ports of the inverted port group must be in the same order as in the referenced port group, but with the opposite directions.

Referring to the **session2_cockpit_display_exercise** example, a port group type declaration and its inverse port group is given:

```
port group Page_Processing_Socket
  features
    New_Page_Content: in data port Page_Content;
    New_Page_Request: out data port Page_Request;
end Page_Processing_Socket;

port group Page_Processing_Plug
  inverse of Page_Processing_Socket
end Page_Processing_Plug;
```

For the current state, we take a port group that it represents a collection of ports in consideration. A port group that consists of other port groups is left for further implementation. Since the ports of a port group may be in all directions: some in ports and some out ports. We can not simply use a Signal bundle to represent it. Instead, we may use **a pair of bundles**: one for the in ports and the other for the out ports, named as **xx_BW** and **xx_WB**, where xx is the name of the port group, and BW stands for Black White. (The properties are not considered yet.)

We can have a Signal representation of the port groups declarations of the previous example:

```
type Page_Processing_Socket_BW = bundle (Page_Content New_Page_Content;);
type Page_Processing_Socket_WB= bundle (Page_Request New_Page_Request);
type Page_Processing_Plug_BW = Page_Processing_Socket_WB;
type Page_Processing_Plug_WB = Page_Processing_Socket_BW;
```

The port group declaration is referenced when it is declared as component features. An example:

```
device Display
  features
    ReqOut_ResultIn: port group Page_Processing_Socket;
  ....
end Display;
```

Signal input/output that represents the feature group should be added into the interface of Signal process that represents the AADL process.

```
process Display_Device =
(? Page_Processing_Socket_BW ReqOut_ResultIn_bw;
 ! Page_Processing_Socket_WB ReqOut_ResultIn_wb;)
(|...|)
```

For a port group of a thread, maybe it is more complicated (the ports of a thread). For the moment, we do not take this case into account.

Feature group Implemented data 2012-02-03

In AADLv2, a feature group represents a group of component features or feature groups. It can be declared for any kind of feature, for ports, and for access features. For the current implementation, only the ports are considered. It is implemented the same as the port group (in AADLv1).

3.16 Connection

Table 19 gives the implementation status of connections.

Port connection	Parameter connection	Access connection	Feature group connection
partly	partly	none	partly
2011-07-22	2011-07-22		2012-02-03

Table 19. The implemented connections

The connection properties are listed in Table 20. They are implemented as process calls in the $xx\_connection\_property()$ subprocess. The **Latency** property is taken into account, and implemented as a memory for a latency time.

Property	port conn.	param. conn.	access conn.	feature group conn.
Allowed_Connection_Binding_Class	partly (2011-07-22)	partly (2011-07-22)	none none	none none
Allowed_Connection_Binding	partly (2011-07-22)	partly (2011-07-22)	none none	none none
Actual_Connection_Binding	partly (2011-07-22)	partly (2011-07-22)	none none	none none
Required_Virtual_Bus_Class	partly (2011-07-22)	partly (2011-07-22)	none none	none none
Required_Connection_Quality_Of_Service	partly (2011-07-22)	partly (2011-07-22)	none none	none none
Connection_Pattern	partly (2011-07-22)	partly (2011-07-22)	none none	none none
Connection_Set	partly (2011-07-22)	partly (2011-07-22)	none none	none none
Transmission_Type	partly (2011-07-22)	- -	- -	- -
Latency	totally (2011-07-22)	totally (2011-07-22)	none none	none none
Classifier_Matching_Rule	partly (2011-07-22)	partly (2011-07-22)	none none	none none

Table 20. The standard connection properties

3.16.1 Port connection

A semantic port connection represents a directed flow of data and control between threads, processors and devices. A port connection $x \in C =< id_x, p_s, p_d, P >$ (identified by id_x) specifies a directed flow of data between source port p_s and destination port p_d. The properties P can be associated to x.

1. **Case 1: the connected entities belong to the same processor.**

   One considers that latencies associated with AADL connections are meaningless when connected entities necessarily belong to the same processor: they are ignored in our implementation. These connections are represented as **direct links** between the components.

   $$\mathcal{T}(x) ::= \mathcal{T}(p_d) := \mathcal{T}(p_s)$$

   For example, t1 and t2 are two thread subcomponents of a same process p1:

   port connection: t1.porta $\rightarrow$ t2.portb;

   This port connection is represented as follows in the body of the Signal process corresponding to p1:

```
process p1_Process = (? ...; ! ...;)
  (| (..., t1_porta) := t1_Thread(...)
   | t2_portb := t1_porta
   | (...) := t2_Thread(..., t2_portb)
   | ...|)
where ... end;
```

2. **Case 2: the connected entities belong to different processors.**

   Latencies of process connections (of different processors) must be taken into account. Such a connection is translated as a $id_x\_Connection()$ Signal process, the behavior of which ($id_x\_Connection\_behavior()$) is a memorization of the transmitted value to mimic the latency; this connection is composed as usual with a $id_x\_Connection\_property()$.

   For example, p1 and p2 are two processes, and they are executed on two different processors:

```
data port connection: p1.portc -> p2.portd;
```

   This port connection is represented in the upper level system (sys1) process body as:

```
process sys1_System = (? ...; ! ...;)
(| p2_portd := cxx_Connection{integer}(p1_portc, latency_time_event)
 | ...|)
where
  ...
  process cxx_Connection = {type item_type;}
    (? item_type write_data; event Latency_time_event; ! item_type read_data;)
    (| read_data := xx_Connection_behavior
                {item_type}(write_data, Latency_time_event)
     | xx_Connection_property(Latency_time_event, Reference_time_event)
     | Reference_time_event := when (^ write_data)
     |)
  where
    event Reference_time_event;
    process xx_Connection_behavior= {type item_type;}
      (? item_type write_data; event Latency_time_event; ! item_type read_data;)
      (| read_data := Connection{item_type}(write_data,Latency_time_event)
       | );
    process xx_Connection_property=
      (? event Latency_time_event, Reference_time_event;)
      (| Latency_property{Time_Units#ms, Time_Units#ms, 2, 5}
            (Latency_time_event, Reference_time_event)
       | ...|);
  end;
  ...
end;
```

   - The $id_x\_Connection\_behavior()$ subprocess invokes the $Connection()$ process defined in the **AADL_CONNECTION** Signal library.
   - The $id_x\_Connection\_property()$ subprocess invokes the processes that represent the properties that are specified for the connection.

3.16.2 Parameter connection

A parameter connection represents a flow of data between the parameters of a sequence of subprogram calls in a thread. A parameter connection may be declared between a thread and its subprogram or between subprograms of a thread. We assume that a thread and its subprograms belong to a same processor, so that the latencies associated with parameter connections are ignored.

A parameter connection definition is represented as a Signal process ($xxx\_Parameter\_Connection()$). For simple, the output equals directly to the input. A subprocess ($xxx\_Parameter\_Connection\_property()$) for specifying the properties is composed.

```
process xxx_Parameter_Connection = { type item_type; }
  ( ? item_type write_data; ! item_type read_data;)
  (| xxx_Parameter_Connection_property()
   | read_data := write_data |)
where
   process xxx_Parameter_Connection_property =( );
end;
```

This process representing the connection is instantiated as a process call in the Signal process body of the thread implementation.

For example, subp1 is a subprogram subcomponent of thread t, and they have a connection:

```
   pconn1: parameter connection: in1 -> subp1.x1;
```

This connection is represented as follows in the process body of thread t:

```
process t_Thread = (? ...; ! ...;)
  (| subp1_x1 := pconn1_Parameter_Connection{integer}(l_in1)
   | ...|)
where
  process pconn1_Parameter_Connection ={}();
  ...
end;
```

3.16.3 Access connection

Not implemented. The data access connection is not implemented explicitly, but implicit connection is provided when accessing shared data.

3.16.4 Feature group connection (port group connection)

- **Port group connection**

  Within a component, elements of a port group in its component type can be individually connected to ports of subcomponents. But elements of a port group of a subcomponent can not be individually connected to other subcomponents.

  An port group connection of **session2_cockpit_display_exercise** example is given below:

```
system implementation Flight_System.impl
    subcomponents
        D: device Display.MFD;
        DM: system Display_Manager.impl;
        ...
    connections
        D_and_DM_conn: port group D.ReqOut_ResultIn -> DM.ReqIn_ResultOut;
        ...
end Flight_System.impl;
```

It can be represented in Signal as two equations that connect the pair of bundles.

```
process Flight_System_impl_system = (? ... ! ... )
( | l1 := D_device(l0)
  | (..., l3) := DM_system(..., l2)
  | l2 := l1
  | l0 := l3
  | ...|)
where
  ...
  Page_Processing_Socket_BW l0;
  Page_Processing_Socket_WB l1;
  Page_Processing_Plug_BW l2;
  Page_Processing_Plug_WB l3;
end;
```

- **Feature group connection**

  Implemented data 2012-02-03

  In the current implementation, only the ports are considered when translating the feature group. Hence the feature group connection is implemented similar as port group connection.

3.17 Flow

Not yet implemented. (open)

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 (type) can have a **flow specification**, which specifies whether a component is a **flow source**, a **flow sink** or there exists a **flow path** through the component. Within a component implementation, a **flow implementation** specifies how a flow specification is realized, and an **end-to-end flow** is a logical flow through a sequence of system components. The differences between a **flow implementation** and an **end-to-end flow** is that, an **end-to-end flow** involves the declaration of a path from a **flow source** or a **flow path** to a **flow sink** or a **flow path** within the component.

An example of flow declaration (flow source, flow sink and flow path) is given:

```
device brake_pedal
    features
        brake_event: out event data port float_type;
```

```
    flows
        Flow1: flow source brake_event;
end brake_pedal;

device throttle_actuator
    features
        throttle_setting: in data port float_type;
    flows
        Flow1: flow sink throttle_setting;
end throttle_actuator;

system cruise_control
    features
        brake_event: in event data port;
        throttle_setting: out data port float_type;
    flows
        brake_flow: flow path brake_event -> throttle_setting;
end cruise_control;
```

A flow (path) implementation example is given:

```
system implementation cruise_control.impl
    subcomponents
        data_in: process interface;
        control_laws: process control;
    connections
        C1: event data port brake_event -> data_in.brake_event;
        C3: data port data_in.out_port -> control_laws.in_port;
        C5: data port control_laws.out_port -> throttle_setting;
    flows
        brake_flow: flow path brake_event -> C1 -> data_in.interface_flow1
            -> C3 -> control_laws.control_flow1 -> C5 -> throttle_setting;
end cruise_control.impl;

process interface
    features
        brake_event: in event data port;
        out_port: out data port float_type;
    flows
        interface_flow1: flow path brake_event -> out_port;
end interface;

process control
    features
        in_port: in data port float_type;
        out_port: out data port float_type;
    flows
        control_flow1: flow path in_port -> out_port;
end control;
```

3.18 Mode

Not yet implemented. (open)

flow specification	flow source flow sink flow path	none none none
flow implementation	flow source flow sink flow implementation	none none none
end to end flow		none
predefined flow property	Latency Actual_Latency	none none

Table 21. The status of flow

3.19 Property

A property $p =< id_p,\ assignment,\ in\_binding >\in P$ is composed of :

- id_p is the identifier of the property,
- $assignment ::= property\_value$ is the property value,
- $in\_binding$ is a set of binding components.

A property $p \in P$ is represented as a SIGNAL process:

$$\mathcal{T}(p) \ ::= \ \mathbf{process}\ id_p\_property = ()()$$

In the current implementation, most of the property processes are empty. For example, the process that represents the **Input_Time** property, the property values are fetched as process parameters, the actual $input\_time$ signal is intended to be verified (external C code should be provided) whether it satisfies the timing constraints as specified in the property.

4 Behavior Annex transformation

This behavior annex contains the following scopes:

- internal behavior of component implementations.
- extended run-time execution semantics, such as thread dispatch protocols.
- subprogram calls synchronization protocols.

In this document, we mainly focus on the behavior description part. Behavior specifications annex can be attached to AADL component implementations. (When defined within component type, it represents behavior common to all the associated implementations.)

The behavioral annex describes a state transition system by three sections (note: $[x]$ represents x is optional, $\{x\}+$ represents x repeats at least once, $\{x\}*$ represents x repeats zero or several times):

```
behavior_annex ::=
  [variables {behavior_variable}+]
  [states {behavior_state}+]
  [transitions {behavior_transition}+]
```

1. The **variables** section declares local typed identifiers. Types are data classifiers of the AADL model.

   ```
   behavior_varialbe ::=
       declarator {, declarator}* : data_unique_component_classifier_reference;
   declarator ::= id {array_size}*
   ```

2. **states**: declares automaton states. The states can be qualified as **initial, final, complete**. A component starts from an **initial** state, and ends with a **final** state. It will resume from that state at next dispatch.

   A **complete** state represents temporary suspension of execution and resumption based on external trigger conditions. A **complete** state acts as a suspend/resume state out of which threads and devices are dispatched. (The **complete** state is not used in subprograms.)

   A state that is qualified as **final**, and is not at the same time **initial** or **complete**, cannot accept outgoing transitions.

   ```
   behavior_state ::= id {, id}* : [initial] [complete] [final] state;
   ```

   No composite state ??

3. **transitions**: this section could define system transitions from a source state to a destination state, (the action is related to the transition and not to the states).

   A transition out of the **initial** state is triggered by the **initialize** action.

   A **dispatch trigger** will result in a transition out of a **complete** state to zero or one execution state.

   A component is suspended if it performs a transition to a **complete** state, after having executed the action associated to the transition. The next dispatch will restart the thread from that state.

```
behavior_transition ::=
    [id [ [priority] ] :] source_state_id {, source_state_id}*
      -[behavior_condition]-> dest_state_id [behavior_action_block];
```

(a) **priority**

If transitions have been assigned a **priority number**, then the priority determines the transition to be taken. If more than one transition out of a state evaluates its condition to true, and no priority is specified, then one transition is chosen **non-deterministically**. For multiple transitions with the same priority value, the selection is also **non-deterministic**. (**No example has been found yet**)

(b) **condition**

Transitions can be guarded by **dispatch conditions** or **execute conditions**.

```
behavior_condition ::= dispatch_condition | execute_condition
```

i. **Dispatch conditions** explicitly specify dispatch conditions out of a **complete** state. A **dispatch trigger condition** can be the arrival of events or event data on ports, calls on provides subprogram access features, the **stop** event, and occurrence of **dispatch** related and completion related **time outs**. If not specified, the **default dispatch conditions** provided by the core AADL standard apply.

```
dispatch_condition ::=
    on dispatch [dispatch_trigger_condition] [frozen frozen_ports]
dispatch_trigger_condition ::=
    dispatch_trigger_logical_exp
  | provides_subprogram_access_id
  | stop
  | timeout behavior_time
  | timeout
dispatch_trigger_logical_exp ::=
    dispatch_conjuction {or dispatch_conjunction}*
dispatch_conjuction ::= in_port_id {and in_port_id}*
frozen_ports ::= in_port_id {, in_port_id}*
```

Periodic dispatches are always considered to be implicit unconditional dispatch triggers.
Timeout is a dispatch trigger condition that is raised after the specified amount of time since the last dispatch or the last completion is expired. For the **timed** thread, the **Timeout** property specifies the timeout value.
Stop events are dispatch triggers to model initiation of finalization and transition from a complete state to the final state, possibly via one or more execution states.
If the freezing of an input port is specified with the **Input_Time** property, then no **freeze input action** must be specified in the corresponding dispatch condition (**frozen** section) or behavior actions (>> operator) of the behavior subclause, or the two statements must be equivalent.

ii. **Execute conditions** specify transition conditions out of an execution state to another state.

```
execute_condition ::= [logical_exp | timeout | otherwise]
```

For the **timed** or **hybrid** thread, the value of the **timeout** dispatch condition is given by the **Period** property.

An **otherwise** execute condition becomes true when all the other execute conditions associated with transitions from the same state are false.

An empty execute condition is equivalent to a condition that is always true.

(c) **expression**

The behavior expression **exp** includes:

i. **variable expression**: incoming ports and parameters, local variables, referenced data subcomponents, and port count, port fresh and port dequeue.

```
p       %returns the data value stored in the port variable. Multiple
         reference to p return the same value unless a dequeue
         or an input freeze is performed.%
p'count %returns the number of elements available through the port
         variable p%
p'fresh %returns whether the port variable p contains a new value%
p?      %dequeues an event (event data) on an event (event data)
         port variable%
```

ii. **constant expression**: $true$, $false$, $1, 2, ...$

iii. **logical expression (operator)**: or, xor, and, not

iv. **relational expression (operator)**: $<$, $<=$, $=$, $>$, $>=$, $!=$

v. **mathematical expression (operator)**: $+$, $-$, $*$, $/$, mod, rem, $**$, abs

vi. **other expression**:

```
integer_value .. integer_value %integer range%
integer_value unit %time%
```

(d) **action**: Actions are built from **basic actions**, and a minimal set of control structures allowing **action sequences**, **action sets**, **conditional** and **finite loops**. **Action sequences** are executed in order, while **action sets** can be executed in any order. **Finite loop** allow iterations over finite integer ranges.

```
behavior_action_block ::= {behavior_actions} [timeout behavior_time]
behavior_actions ::=
    behavior_action | behavior_action_sequence | behavior_action_set
behavior_action_sequence ::= behavior_action {; behavior_action}+
behavior_action_set ::= behavior_action {& behavior_action}+
```

The **behavior_action** includes:

i. **assignment**:

```
exp1 := exp2
exp := any  ??
```

ii. **communication**:

```
s!(subprogram_parameter_list)
          %calls the subprogram s. In action sequences, it represents
           synchronous subprogram call, while in action sets, it
           represents semi-synchronous call%
p>>       %freeze input port p%
```

```
p?          %dequeues an event on an event port variable%
p?(x)       %dequeues an element from port variable p, and assigns it
             to the local variable x%
p!          %calls Send_Output on an event (event data) port p,
             transmission of the message is initiated immediately%
p!(x)       %writes data x to the event data port p,
             and calls the Send_Output service%
data_id!<   %explicit call Get_Resource of shared data%
data_id!>   %explicit call Release_Resource of shared data%
*!<         %starting time of a time range%
*!>         %ending time of a time range%
```

Note:
If the sending time of an output port is specified with the **Output_Time** property, then no **send_output** action must be specified in the corresponding behavior actions (! operator) of the behavior subclause.

iii. **timing**:

```
computation(min, max) %use of CPU for a possibly non-deterministic
                       period of time between min and max%
```

iv. **loop**:

```
for (id : data_unique_component_classifier_reference in element_values)
    {behavior_actions;}

forall (id : data_unique_component_classifier_reference in element_values)
    {behavior_actions;}

while (exp)
    {behavior_actions}

do behavior_actions
    until (exp)
```

v. **condition**:

```
if (exp1) behavior_actions
    {elseif (exp2) behavior_actions}*
    [else behavior_actions]
end if;
```

vi. **action sequence**: $behavior\_action$; $behavior\_action$

vii. **action set**: $behavior\_action$ & $behavior\_action$

Synchronization protocols This annex introduces more precise communication protocols:

- HSER for Highly Synchronous Execution Request: the caller thread remains blocked until the completion of the corresponding behavior action in the server thread.
- LSER for Loosely Synchronous Execution Request: the caller thread remains blocked until the beginning of the server thread is ready to serve this request.
- ASER for ASynchronous Execution Request:the caller thread thread is never blocked by the corresponding remote call.

...

4.1 Transition system transformation

(expected)
The transition system can be represented using Signal automata.
...

4.2 Expression transformation

(expected)
The basic expressions (operators), such as: x, or, and, not, $<$, $<=$, $=$, $>$, $>=$, $!=$, $+$, $-$, $*$, $/$, can be translated to corresponding Signal expressions (operators), listed in Table 22.
The translation of **other expressions** to be discussed.
Note:
We use $S(E)$ (or $S(A)$) to represent a AADL expression E (or action A) in Singal.

	BA expression	Signal expression
identifier	x	x
constant	true, false, 1, 2, ...	true, false, 1, 2, ...
logical	not E E1 or E2 E1 and E2 E1 xor E2	not S(E) S(E1) or S(E2) S(E1) and S(E2) S(E1) xor S(E2)
relational	E1 $<$ E2 E1 $<=$ E2 E1 $=$ E2 E1 $>$ E2 E1 $>=$ E2 E1 $!=$ E2	S(E1) $<$ S(E2) S(E1) $<=$ S(E2) S(E1) $==$ S(E2) S(E1) $>$ S(E2) S(E1) $>=$ S(E2) S(E1) $/=$ S(E2)
mathematics	E1 $+$ E2 E1 $-$ E2 E1 $*$ E2 E1 $/$ E2 E1 mod E2 E1 rem E2 E1 $**$ E2 abs E	S(E1) $+$ S(E2) S(E1) $-$ S(E2) S(E1) $*$ S(E2) S(E1) $/$ S(E2) S(E1) $modulo$ S(E2) S(E1) $**$ S(E2) abs(S(E))
Others	p'count p'fresh p?	read the value of counter read one element from p

Table 22. Expression translation

4.3 Action transformation

(expected)

Part of the actions possible transformation is listed in Table 23. Actions, such as loop, sequence, set, are left to be discussed.

	BA action	Signal action
assignment	E1 := E2	S(E1) := S(E2) or S(E1) ::= S(E2) ??
	E1 := any	??
communication	$s!(p1, p2, ...)$	subprogram process instance:s(p1,p2,...)
		?? depends on the protocol
	$p >>$	??
	$p?$	p when ...
		get one element of p at certain clock
	$p?(x)$	x:= p when ...
	$p!$	??
	$p!(x)$	??
	$d! <$	??
	$d! >$	??
	$*! <$	??
	$*! >$	??
timing	computation(E1, E2)	??
loop	for (x: data_ref in element_values) {A}	??
	forall (x: data_ref in element_values) {A}	??
	while (E) {A}	??
	do A until (E)	??
condition	if (E1) A1 elseif (E2) A2 else A3 end if;	S(A1) when S(E1)
		| S(A2) when S(E2) when not S(E1)
		| S(A3) when not S(E2) when not S(E1)
sequence	A1;A2	??
set	A1;A2	??

Table 23. Actions translation

4.4 Synchronization protocols transformation

(expected)

5 Implementation technical notes

This section gives some technical points on the implementation.

5.1 Program architecture

Implemented date: Oct 21, 2011

The program of ASME2SSME translation is organized as follows. In general, each AADL component is separated into a java class: **ASME2SSME_Translate_xx**. The hierarchy of classes and sub classes is added.

- **ASME2SSME**: translations from AADL to SSME.
  - **ASME2SSME_Translate_AADLPackage**
  - **ASME2SSME_Translate_AADLSpec**
  - **ASME2SSME_Translate_AnnexLibrary**
  - **ASME2SSME_Translate_Thread**
    1. **ASME2SSME_Translate_AperiodicThread**
    2. **ASME2SSME_Translate_BackgroundThread**
    3. **ASME2SSME_Translate_HybridThread**
    4. **ASME2SSME_Translate_PeriodicThread**
    5. **ASME2SSME_Translate_SporadicThread**
    6. **ASME2SSME_Translate_TimedThread**
  - **ASME2SSME_Translate_Bus**
  - **ASME2SSME_Translate_Common**
  - **ASME2SSME_Translate_Connection**
    1. **ASME2SSME_Translate_ParameterConnection**
    2. **ASME2SSME_Translate_PortConnection**
    3. **ASME2SSME_Translate_PortGroupConnection**
  - **ASME2SSME_Translate_Data**
  - **ASME2SSME_Translate_Device**
  - **ASME2SSME_Translate_Feature**
    1. **ASME2SSME_Translate_Port**
       (a) **ASME2SSME_Translate_DataPort**
       (b) **ASME2SSME_Translate_EventDataPort**
       (c) **ASME2SSME_Translate_EventPort**
    2. **ASME2SSME_Translate_PortGroup**
  - **ASME2SSME_Translate_Flow**

  - **ASME2SSME_Translate_Memory**
  - **ASME2SSME_Translate_Mode**
  - **ASME2SSME_Translate_Name**
  - **ASME2SSME_Translate_Process**
  - **ASME2SSME_Translate_Processor**
  - **ASME2SSME_Translate_Property**
  - **ASME2SSME_Translate_Subprogram**
  - **ASME2SSME_Translate_System**
  - **ASME2SSME_Translate_ThreadGroup**
  - **ASME2SSME_Simulate_Dispatch**

- **signalTreeAPI**: API for SSME setting operations.
- **AADLCoLAPI**: API for AADL getting operations.

5.2 Use of bundles

Implementation date: Aug 03, 2011

Signal "bundles" (groups of signals) are used to simplify the generated code. The following types of bundles are added:

1. A bundle type for representing the *Dispatch, Resume* and *Deadline* signals. This bundle type is predefined in the **AADL_TYPE.SIG** Signal library.

   ```
   type CTL1 = bundle (event Dispatch; event Resume; event Deadline;);
   ```

2. A bundle type for representing the *Complete* and *Error* signals. This bundle type is predefined in the library.

   ```
   type CTL2 = bundle (event Complete; event Error;);
   ```

3. A bundle type for representing the time event signals (*Frozen_time_event* and *Output_time_event*) of the inputs/outputs. For example, a thread *t1* contains an in port *in1* and an out port *out1*. Then in our implementation, a bundle type named *t1_TIME_EVENT* is created, which is composed of two event fields: *in1_Frozen_time_event* and *out1_Output_time_event*.

   ```
   type t1_TIME_EVENT = bundle (event in1_Frozen_time_event;
                                event out1_Output_time_event;);
   ```

   This category of bundle type is generated dynamically during the transformation. Each thread will have one such bundle type, in which each field is an event signal that represents the input (or output) time event for an input (or output).

5.3 Use of implicit signals

Implementation date: Aug 16, 2011

To make the Signal code easier to read, we omit the input/output signals of a process instance if the effective inputs/outputs have the same name as the ones declared in the process definition.

Since the interface of a generated Signal process (e.g., a Signal process that represents an AADL processor) may contain many inputs and outputs, it makes the reader a bit difficult to search the definition (for an output) or usage (for an input) in the process body. In order to explicitly inform the inputs/outputs in the body, we add equations to indicate the usage, $xx := l\_xx$ for an output xx or $l\_yy := yy$ for an input yy, where l_xx and l_yy are local signals that are used as inputs/outputs of the subprocesses.

5.4 Addition of comments

Implementation date: Aug 17, 2011

A new Signal API function, $STree\_addcomments()$, is defined to add comments to a Signal element. The following comments are added:

- At the very beginning of the generated Signal code, we add comments to identify the Signal version, the copyright information, the date, etc. These information are generated in the method $Generate\_FileComments()$. The following messages are given:

```
%This file has been generated by SSME translator version xx.
Copyright: IRISA / INRIA Rennes - developed by ESPRESSO team.
Source simple_door_management in file simple_door_management.aaxl.
Date 2011/Aug/17 15:00:00
%
```

- Comments for data type.

```
type DataB_imp_Data = struct ( DataA x; DataA y; );
  %data type for AADL data (implementation) DataB.imp %
```

- Comments for process instance. For example, the comment for a subprocess instance that represents a device *ClosedSensor.imp* is added in the *Door_impl_System* process body.

```
process Door_imp_System = ( ! DataA closed1;)
(| Door_imp_System_behavior()%system behavior sub process %
 | Door_imp_System_property()%system property sub process%
 | closedsensor1_closed := ClosedSensor_imp_Device{}()
       %process instance of AADL device ClosedSensor.imp%
 | closed1 := closedsensor1_closed
 |)
```

Some other comments are added. They are not all listed here.

5.5 Java documentation

Implementation date: Feb 03, 2012.

The implementation of the transformation mainly contains the following Java files (each Java class contains a set of methods).

1. ASME2SSME_Translate_xx.java: a Java class for translating AADL elements xx to Signal elements.
2. AADLCoLAPI.java: a high level Java API for getting methods from AADL components.
3. signalTreeAPI.java: a high level Java API for setting methods of Signal abstract tree.

Java documentation is generated for these Java files. Each method is given a brief comment on its parameters, return value and function.

5.6 Use of .aadl text file

Implementation date: June, 2012

From OSATEv2, Xtext is used to access the aadl resource. With the help of Xtext, we can directly access the objects from .aadl text file instead of .aaxl object file. The method $xtextEditor.getDocument().readOnly()$ is used (details can be found in method $startTransformation()$ in file $ASME2SSME.java$).

Important: To access the objects of a text aadl file, this file must be opened in a current view.

5.7 Connect to BA plug-in

Implementation date: June, 2012

The BA plug-in (RAMSES) is installed from souce code. The class $BehaviorAnnex$ of BA extends the $AnnexSubclause$ class of OSATEv2.

The test code can be found in method $ASME2SSME\_AADLPackage()$ in file $ASME2SSME\_Translate\_AADLPackage.java$.

5.8 Connect to schedule generator

Implementation date: September, 2012

An automatic schedule generator based on affine clock is implemented. The code is uploaded to gforge $svn + ssh://username@scm.gforge.inria.fr/svnroot/polychrony/ASME2SSME$.

A class *ASME2SSME_Scheduling* is added to connect to the schedule generator. A list of time information of each thread is construceted. The schedule generator reads this list, and performs scheduling.

In method *ASME2SSME_Processor_Behavior()* in file *ASME2SSME_Translate_Processor.java*, two versions of scheduler are connected: 1) version 1: simulate the dispatch, deadline schedule signals separately; 2) version 2: connect to schedule generator.

References

[1] ASME2SSME Gforge. svn+ssh://username@scm.gforge.inria.fr/svn/polychrony/ASME2SSME.

[2] OPEES Project. http://www.opees.org/.

[3] OSATE. http://gforge.enseeiht.fr/projects/osate/.

[4] SAE Aerospace. Architecture Analysis and Design Language (AADL). *SAE AS5506*, 2004.

[5] SAE Aerospace. Aerospace Standard AS5506A: Architecture Analysis and Design Language (AADL) . *SAE AS5506A*, 2009.

[6] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, pages 16:103–149, 1991.

[7] Loïc Besnard, Thierry Gautier, and Paul Le Guernic. SIGNAL V4-Inria Version: Reference manual, 2011.

[8] ESPRESSO, INRIA. SME, 2011. http://www.irisa.fr/espresso/Polychrony/.

[9] ESPRESSO Team. ESPRESSO AADL Digest Report. *Report*, April 2011.

[10] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12:261–304, 2002.

[11] Yue Ma. *Compositional modeling of globally asynchronous locally synchronous (GALS) architectures in a polychronous model of computation*. PhD thesis, University of Rennes1, France, 2010.

[12] Huafeng Yu, Yue Ma, Yann Glouche, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Paul Le Guernic, Andres Toom, and Odile Laurent. System-level Co-simulation of Integrated Avionics Using Polychrony. In *SAC'11*, Taiwan, March 2011.

A Test cases

Table 24 gives a list of cases that are tested in ASME2SSME (based on the version implemented on **18/05/2012**) translation tool. Since there are not many AADLv2 examples published, the cases we used are mainly modified from their v1 version by hand.

Name	Size (lines)	Main features	Not yet implemented
ProducerConsumer	170	system, processor, process, thread, shared data, behavior annex	behavior annex (implemented by hand)
APOTA	760	system, processor, process thread, shared data, behavior annex	behavior annex
test_sp	240	system, processor, process, thread, subprogram, subprogram call	call of external simulink source code
case1	110	system, process, processor, thread, subprogram, subprogram call	
case2	125	system, processor, process, thread, subprogram, subprogram call, requires subprogram access	
case3	100	system, processor, process, thread, subprogram	remote subprogram call
SDSCS	500	system, processor, process, thread	bus access
Flight_Mgmt	160	system, processor, process, thread	call of external c code
Flight_Mgmt2*	160	system, processor, process, sporadic thread	call of external c code, (clock constraint)
safetyexample	75	system, processor, process, thread	
safetyexample2	70	system, processor, process, thread (an exception is thrown when thread type not specified)	
securityexample	57	system, processor, process, periodic thread, aperiodic thread	
errormodelexample*	80	system	modes, error annex
avionics_system _flow_latency	140	system, device	flow
Avionics_System	125	system, device	flow

Table 24. Test examples list

All these examples can be translated to Signal programs (maybe with some components not implemented yet) and the generated Signal code can be complied (except for the two marked with *). For further code execution and simulation, some processes need to be modified or replaced, e.g., the Signal process that are declared as *external* should be provided, the behavioral automaton should be added, etc..

The following subsections will give a more precise description for some of the examples.

A.1 ProducerConsumer

This case study is from C-S Communication and Systems. Four threads share one data *Queue* in a process.

The example implements the following components (Table 25):

Components	Implemented	Not yet implemented
system	subcomponents, port connections	
processor	subcomponents	
memory		
process	subcomponents, port connections	
thread	event port, share data requires data access, Dispatch_Protocol, Period, Deadline	behavior annex (hand written)
data	data shared by threads	

Table 25. ProducerConsumer implementation

The associated code (Table 26) is attached in folder *./Examples/ProducerConsumer*:

File	Description
ProducerConsumer.aadl	original AADLv1 text file
ProducerConsumer.aadl2	confirm to AADLv2 version
ProducerConsumer.aaxl2	AADLv2 object file
ProducerConsumer.ssme	generated SSME file
CS.SIG	generated Signal program
ProducerConsumer.SIG	refined Signal program for invoking automata Signal process
automata.SIG	hand written automata Signal process
ProducerConsumerSim.SIG	simulation code
simuLib.SIG	simulation library code

Table 26. ProducerConsumer file

A.2 APOTA

This case study is from C-S Communication and Systems. Twelve data are shared by five threads in a same process.

The example implements the following components (Table 27):

The associated code (Table 28) is attached in folder *./Examples/APOTA*:

A.3 Subprogram case study

This case study is from TELECOM ParisTech (*http://eve.enst.fr/aadl/wiki/CaseStudySimulink*).

Components	Implemented	Not yet implemented
system	subcomponents	
processor	subcomponents, Clock_Period	
memory	Read_Time	
process	subcomponents, port connections, data access	
thread	event port, subcomponents, requires data access, Dispatch_Protocol, Period, Deadline	behavior annex
data	data shared by threads	

Table 27. APOTA implementation

File	Description
ProducerConsumer.aadl	original AADLv1 text file
CS_APOTA.aadl2	confirm to AADLv2 version
CS_APOTA.aaxl2	AADLv2 object file
CS_APOTA.ssme	generated SSME file
CS_APOTA.SIG	generated Signal program

Table 28. APOTA files

A graphical view of AADL model is shown in figure 16. Three processes are given: *pilot*, *controller* and *aircraft*. Each process has one thread, and the thread calls a subprogram that is described by Simulink source code.

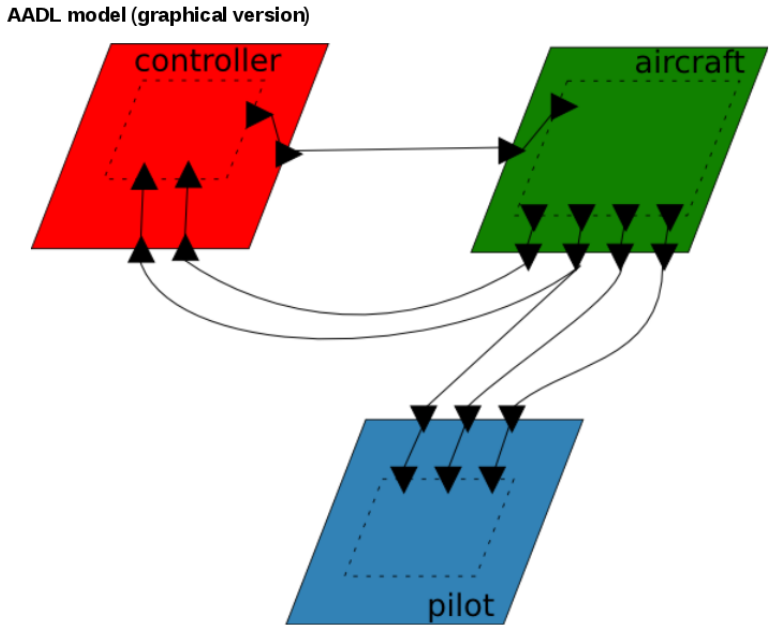

Figure 16. AADL model (graphical view)

test_sp.SIG is the Signal program generated by our tool. In order to make the program executable, we add some behavior instead of external process in *test_sp2.SIG* for test and simulation. The code is attached in folder *./Examples/Subprogram*.

Table 29 shows the implementation in this case.

Components	Implemented	Not yet implemented
system	subcomponents, port connections, actual_processor_binding	
processor		requires bus access
bus		
process	subcomponents, port connections	
thread	data port, parameter connections, subprogram call, Dispatch_Protocol, Period	
subprogram	parameter	call of external Simulink code
subprogram call	parameter connections	

Table 29. Case of subprogram call implementation

A.4 Doors management

Simplified doors and slides control system (SDSCS) is a generic simplified version of the system that allows managing doors on Airbus series aircrafts. Each passenger door has a software handler, which achieves the management tasks, such as monitoring door status via doors sensors, controlling flight lock actuators, etc.. These tasks are implemented with simple logic that determines the status of monitors and actuators according to the sensor readings. Figure 17 is an overview of SDSCS modeled in AADL.

The implementation status of this example is given in Table 30. The code is attached in folder *./Examples/Doors*.

Components	Implemented	Not yet implemented
system	data ports, port connections, subcomponents, Actual_Processor_Binding	bus access
processor		requires bus access
device	data port, Dispatch_Protocol, Period	requires/provides bus access
bus		
process	data ports, port connections, subcomponents	
thread	data port, Dispatch_Protocol, Period	

Table 30. SDSCS implementation

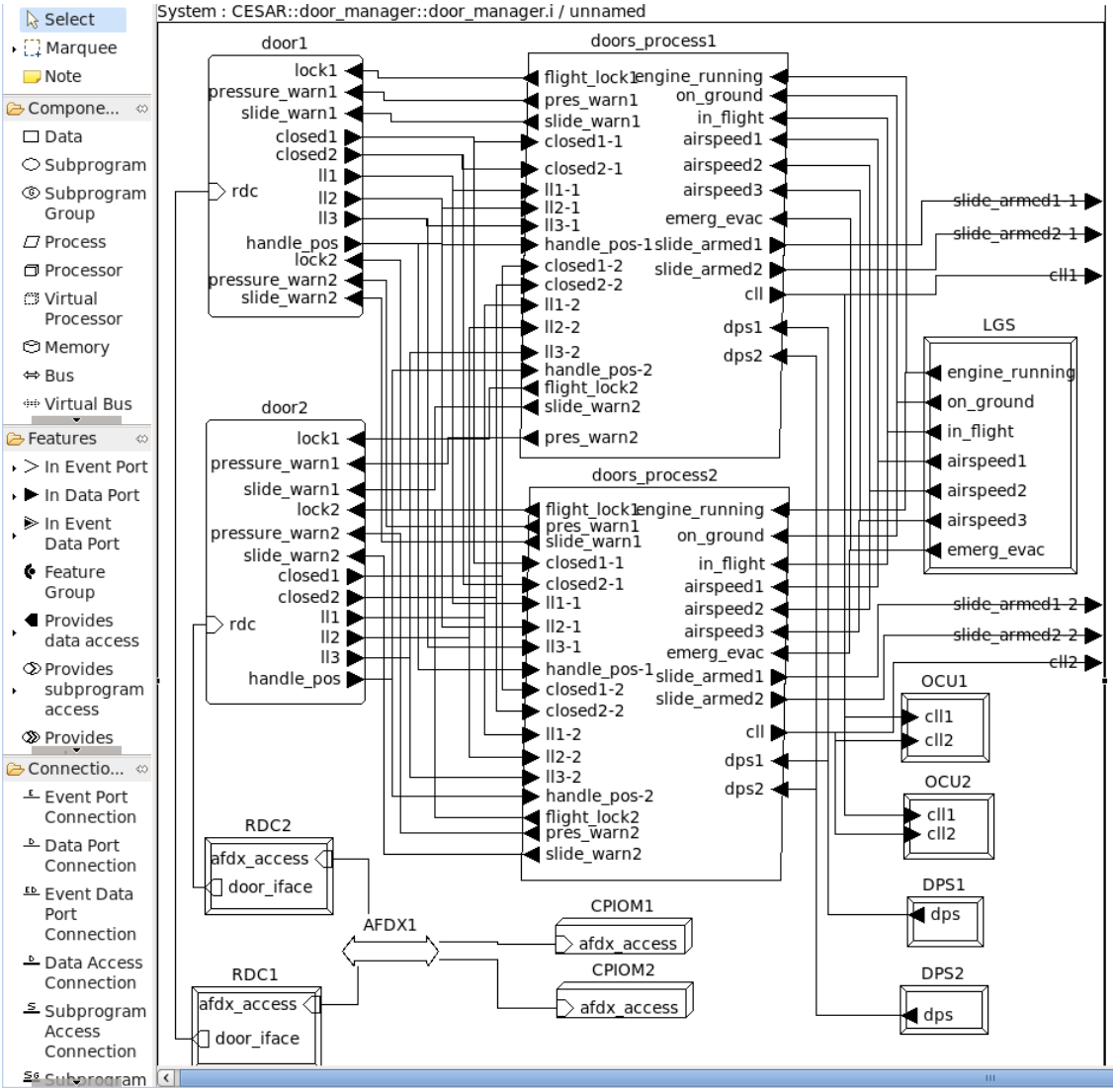

Figure 17. AADL model of SDSCS