



# AADL Workshop 2005

October 17th-18th, 2005

## Overview of AADL Syntax

J-P. Rosen, Adalog  
J-F. Tilman, Axlog

# Component categories

## software categories



process



subprogram



data



thread



thread group

## composite category



system

## platform categories



processor



memory



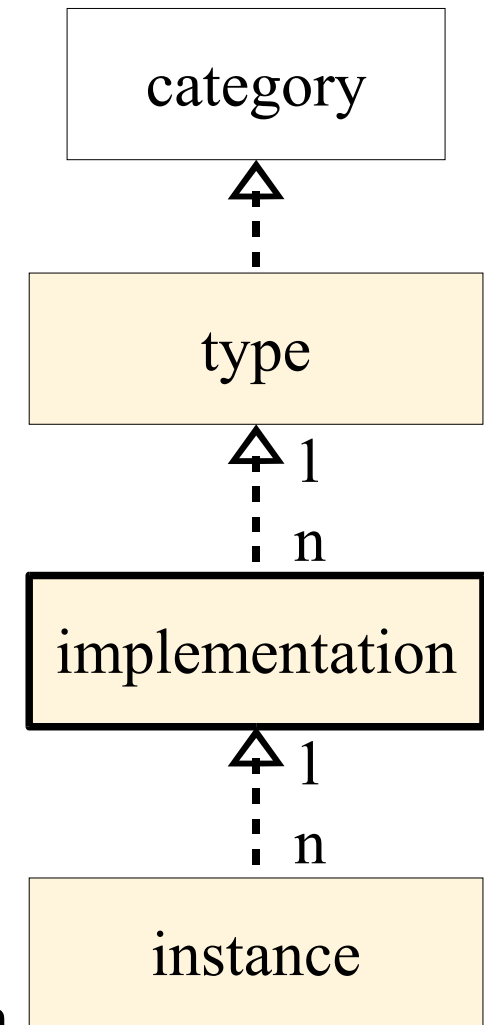
device



bus

# Three levels of description

- Category (predefined)
- Type
  - specification of the external interface
- Implementation
  - specification of the content
- Instance
  - instantiation of a type or an implementation



# Component type

- A component type contains:
  - features;
  - flow specifications;
  - property associations.
- Example

```
process application
  features
    -- an out data port
    result: out data port App::result_type;

    -- access to a data component
    home: requires data access Directory hashed;
end application;
```

## Component implementation (1)

- The description of an implementation must conform to the description of the corresponding type<sub>AADL</sub>.
- Contents of a component implementation:
  - **refines type** : refines the properties of features declared in the component type<sub>AADL</sub>.
  - **subcomponents** : declaration of subcomponents
  - **calls** : calls of subprograms
  - **connections** : declaration of connections between features
  - **flows** : declaration of flow implementations
  - **modes** : declaration of operational modes
  - **properties** : property associations for the component

## Component implementation (2)

- Example

```

thread DriverModeLogic
  features
    CruiseActive : out data port Bool_Type ;
end DriverModeLogic ;

-- Two implementations with different source ports.
thread implementation DriverModeLogic.Simulink
  refines type
    CruiseActive: refined to out data port Bool_Type
      { Source_Name => "CruiseControlActive" ; } ;
  properties Period => 10 ms;
end DriverModeLogic.Simulink;

thread implementation DriverModeLogic.C
  refines type
    CruiseActive: refined to out data port Bool_Type
      { Source_Name => "CCActive" ; };
  properties Period => 10 ms ;
end DriverModeLogic.C ;

```

# Subcomponent

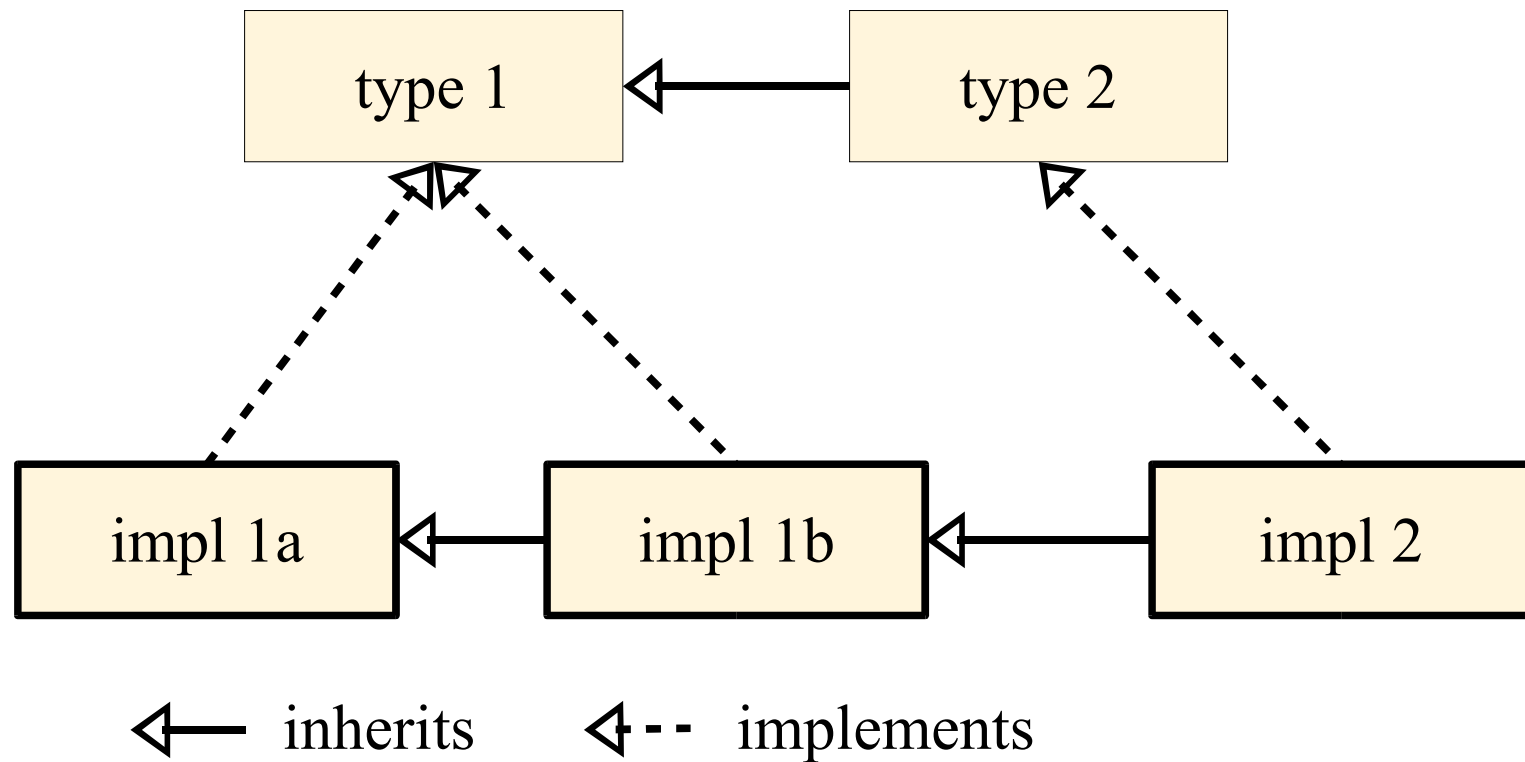
- The specification of a subcomponent is defined by:
  - its category (mandatory) ;
  - its type (optional) ;
  - its implementation (optional).
- Example

```
processor sparc end sparc ;
processor implementation sparc.leon end sparc.leon ;
system computer end computer ;

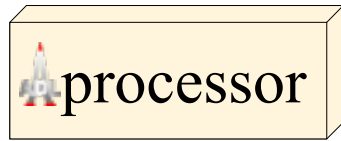
system implementation computer.prototype
  subcomponents
    main_processor : processor sparc.leon ;
    RAM : memory
      { Word_Count => 10000; };
    scheduler : process ;
  end computer.prototype;
```

# Inheritance

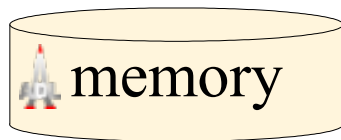
- A type<sub>AADL</sub> may inherit from another type<sub>AADL</sub>.
- An implementation may inherit from another implementation.



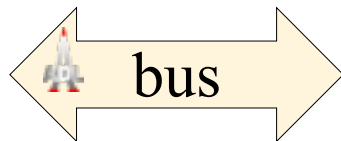
# Platform categories



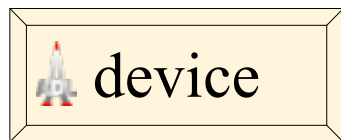
- A *processor* is the execution platform component that is capable of scheduling and executing threads.



- A *memory* component represents an execution platform component that stores binary images.

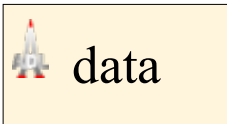


- A *bus* component represents an execution platform component that can exchange control and data between memories, processors, and devices.



- A *device* component represents an execution platform component that provides an interface with the external environment.

# Data component



- The *data* component type represents a data type in the source text that defines a representation and interpretation for instances of data in the source text:
  - component type<sub>AADL</sub>  $\Rightarrow$  data type;
  - component implementation<sub>AADL</sub>  $\Rightarrow$  internal structure of the data type;
  - instance  $\Rightarrow$  static data in the source code.
- Components may have a shared access to a data component.

# Subprogram component

## subprogram

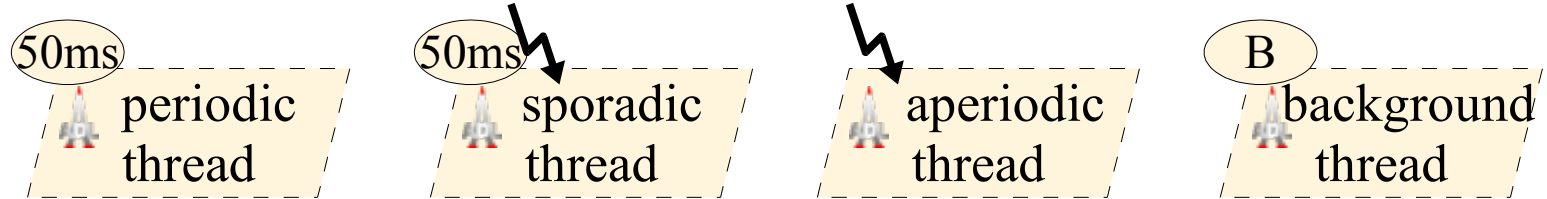
- A *subprogram* component represents an execution entrypoint in source text.
- No component can contain subprogram subcomponents. *Instances of subprogram don't exist.*
- A *subprogram call* in the implementation of a thread or another subprogram may be “seen as” the inclusion of a subprogram subcomponent.

```
thread implementation t1.impl
calls
  sequence : { action1 : subprogram function1 ;
               action2 : subprogram function2 ; } ;
end t1.impl ;
```

# Thread and thread group components

thread

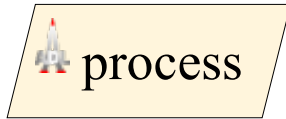
- A *thread* represents a sequential flow of control that executes instructions within a binary image produced from source text.
- A thread always executes within the virtual address space of a process;
- Several types of thread exist



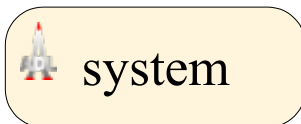
thread group

- A *thread group* represents an organizational component to logically group threads contained in processes.
- The thread groups can be hierarchically nested

## Process and system components

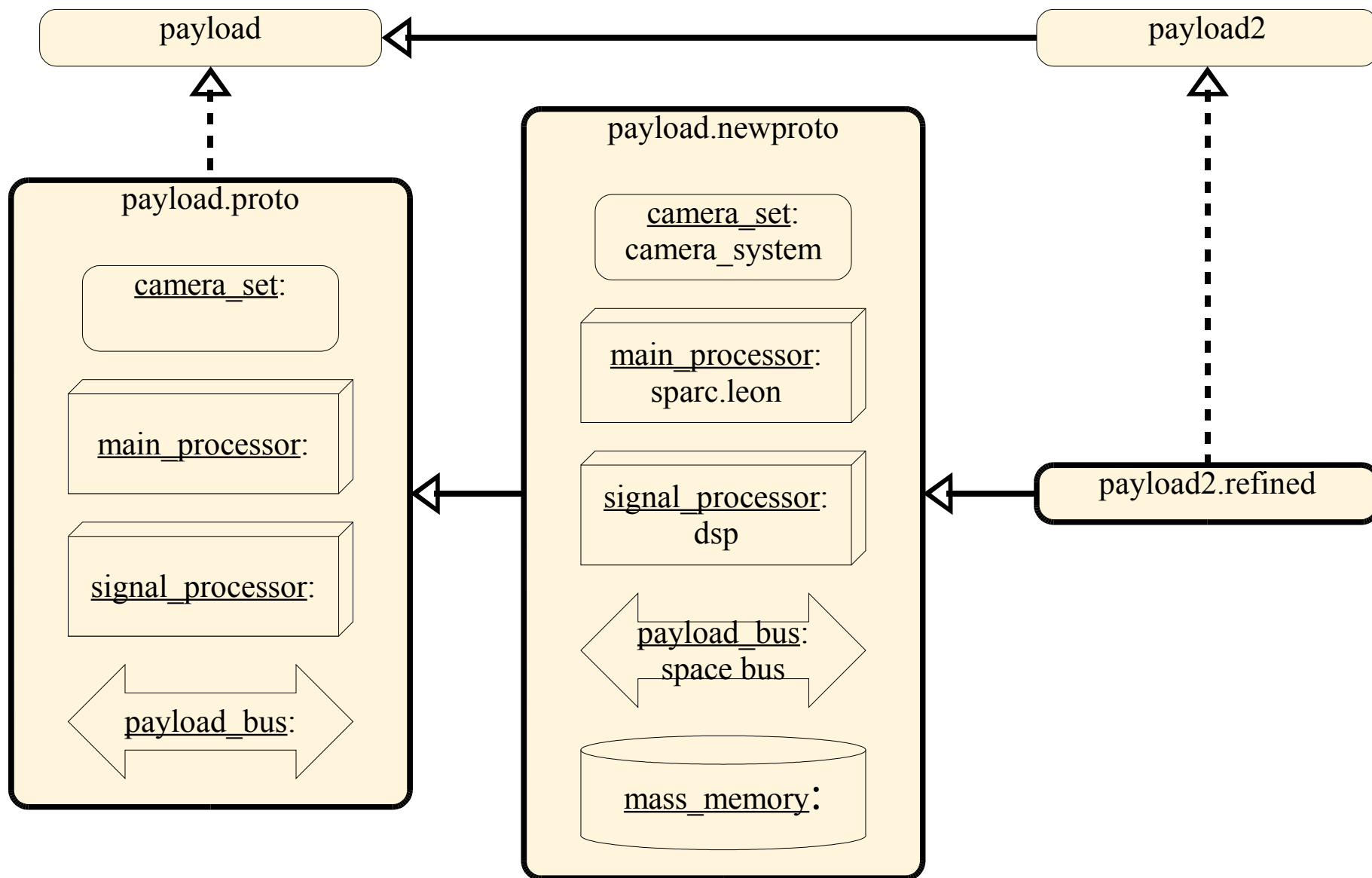


- A *process* represents a virtual address space.
- To be complete, the implementation of a process must contain at least one thread or thread group subcomponent.



- A system component represents an assembly of software and execution platform components.
- It is the only composite category.

# Example (1/2)





## Example (2/2)

```
system payload end payload ;

system implementation payload.proto
  subcomponents
    camera_set : system ;
    main_processor : processor ;
    signal_processor : processor ;
    payload_bus : bus ;
end payload.proto ;

system implementation payload.newproto
  extends payload.proto
  subcomponents
    camera_set : refined to
      system camera_system ;
    main_processor : refined to
      processor sparc.leon ;
    signal_processor : refined to
      processor dsp ;
    payload_bus : refined to
      bus spacebus ;
    mass_memory : memory ;
end payload.newproto ;

system camera_system end camera_system;
bus spacebus end spacebus ;
processor dsp end dsp ;
```

```
processor sparc end sparc ;
processor implementation sparc.leon
end sparc.leon ;

property set space_properties is
  mass_t: type aadreal units mass_u ;
  mass_u: type units
    (g, kg => g*1000, t => kg*1000);
  mass_range_t: type range of mass_t;
  allowed_mass: mass_range_t applies to
    (memory, processor, bus, device, system);
  actual_mass: mass_t applies to
    (memory, processor, bus, device, system);
end space_properties ;

system payload2 extends payload
  properties
    space_properties::allowed_mass
      => constant 200.0kg .. 500.0kg ;
end payload2 ;

system implementation payload2.actual
  extends payload.newproto
  properties
    space_properties::actual_mass
      => 300.0 kg ;
end payload2.actual ;
```

# Properties

- A property provides information about a component, a feature, a mode or a subprogram call.
- A property has a *name* + a *type* + a *value*. It can be constant or variable.
- Property types and property names are declared in property sets.
  - A property set may be predeclared or defined by the user.

```
property set my_properties is
  { property_type | property_name | property_constant }+
end set_name ;
```

- Two property sets are predeclared:
  - *AADL\_Properties* defines properties which are common to all AADL specifications;
  - *AADL\_Project* defines types and enumerated constants, whose values may differ from one project to another. They can be modified by the user.



# Property type and name declarations

- The type of a property defines the set of allowed values.

```
-- simple types declarations
type_name : type other_type_name ;
type_bool : type aadlboolean ;
type_string : type aadlstring ;

-- enumeration type declaration
Access_Type : type enumeration (read_only, write_only, read_write);

-- unit type declaration
Length_Unit : type units (mm, cm=> mm*10, m=> cm*100, km=> m*1000 );

-- number type declaration
Car_length: type aadlreal 1.5 m .. 4.5 m units Length_Unit;

-- range type declaration
Speed_range : type range of aadlreal 0 .. 250 ;

-- constant property name
Max_Aadlinteger: constant aadlinteger => 2#1#e32;
Max_Memory_Size: constant aadlinteger Size_Units => 2#1#e32 B;
Max_Queue_Size : constant aadlinteger => 512;

-- variable property
Wheel_speed : aadlinteger 0 rpm .. 5000 rpm units ( rpm )
               applies to ( system );
```

# Property association

- A property *association* gives values to properties of components.

```
system implementation Software.Basic
subcomponents
  Sampler_A : process Collect_Samples {
    Source_Text => ("collect_samples.ads", "collect_samples.adb") ;
    Period => 50 ms ;
  } ;
end Software.Basic ;
```

- The association may depend on the mode or the binding.

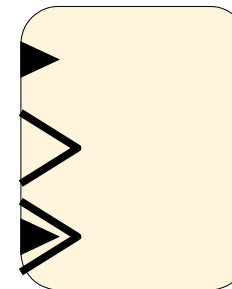
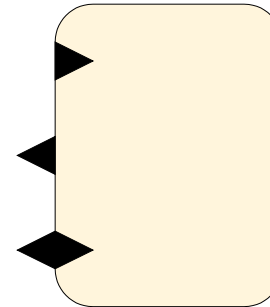
```
thread implementation Producer.Basic
properties
  Compute_Execution_Time => 0 ms .. 10 ms in binding (powerpc.speed_350MHz) ;
  Compute_Execution_Time => 0 ms .. 8 ms in binding (powerpc.speed_450MHz) ;
end Producer.Basic ;
```

## Feature concept

- A *feature* is a part of a component type definition that specifies how that component interfaces with other components in the system.
- Four categories of features:
  - port;
  - subprogram;
  - parameter;
  - subcomponent access.

# Port

- A *port* is a logical connection point between components that can be used for the transfer of control and data.
- Three directions:
  - input port (*in*)
  - output port (*out*),
  - bidirectional port (*in out*).
- Three types of port:
  - data port,
  - event port,
  - event data port.



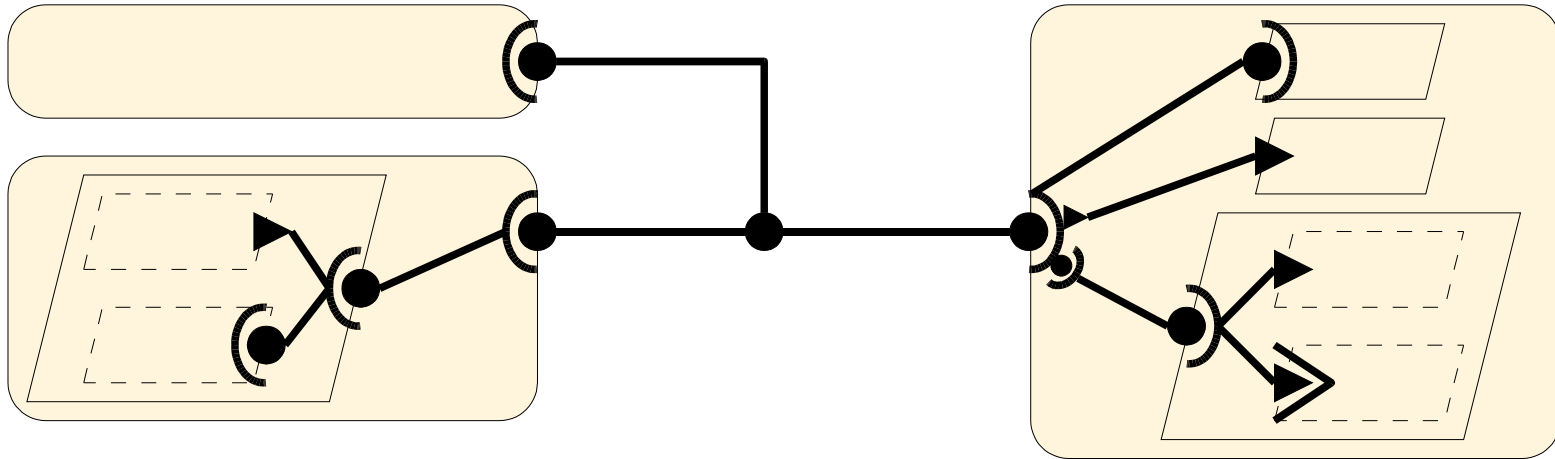
```

process thermostat
  features
    measure : in data port temperature_type ;
    setting : out event data port ;
  end thermostat ;

```

# Port group

- A *port group* can group component ports or other port groups



```

port group thermostat_ports
  features
    measure : in data port temperature_type ;
    setting : out event data port ;
  end thermostat_ports ;

system thermostat
  features
    port_set : port group thermostat_ports ;
  end thermostat ;

```

## Subprogram feature

- A *subprogram* feature represents:
  - either an execution entrypoint in source text that operates on a data component of the associated data type<sub>AADL</sub> (*data subprogram*);
  - or an entrypoint for remote procedure calls (*server subprogram*).
- A subprogram **feature** refers to a subprogram **component**.

```
subprog_name : [ refined to ] [ server ] subprogram  
<subprog_component_name>  
  [ { <property association> } ] ;
```

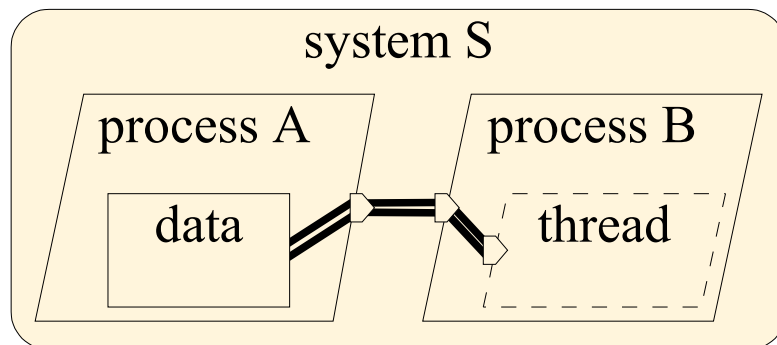
## Subcomponent access

- A subcomponent (data or bus) can be made accessible outside its containment hierarchy.
- A component can declare that it requires access to externally declared subcomponents.

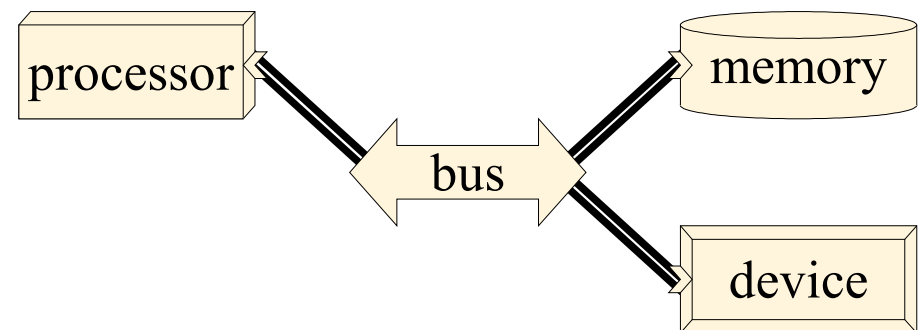
```

<access_name> : [ refined to ]
  ( provides | requires ) ( data | bus ) access
  [ <component typeAADL> [ . <component implementation> ] ]
  [ { <property association> } ] ;
  
```

### Access to a data



### Access to a bus

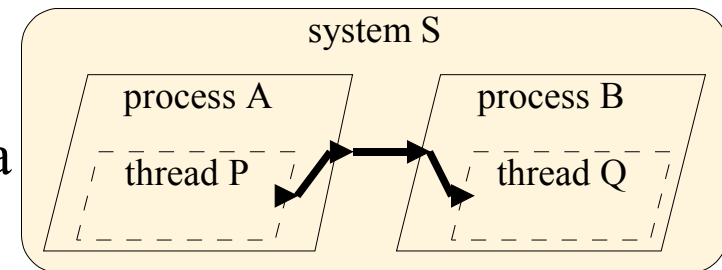


# Connections

- A *connection* is a linkage that represents communication of data and control between components.
- Three types of connections:

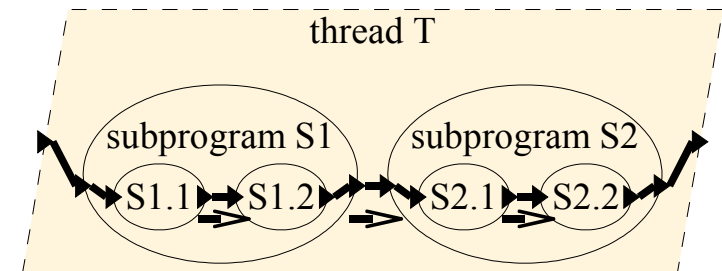
- port connection:

- transfer of data and control between two threads, or between a thread and a processor or a device.



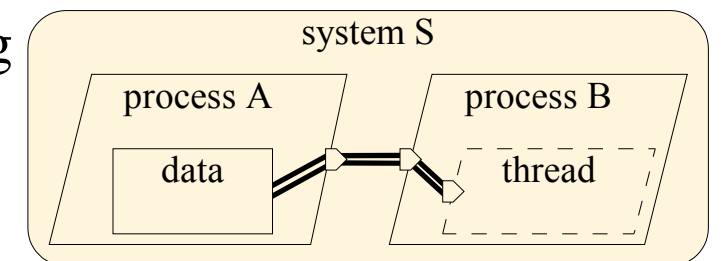
- parameter connection:

- flow of data between parameters of a sequence of subprogram calls.



- access connection:

- access to a shared data component by a thread or by a subprogram executing within a thread.
- communication between processors, memories, or devices through a bus.



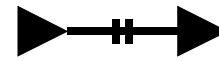
# Port connection

- Immediate connection



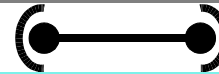
```
[ connection_name : ] ( data | event | event data ) port
<source_port_name> -> <dest_port_name>
  [ { <property association> } ]
  [ in modes ( <mode or transition> { , <mode or transition> }* ) ];
```

- Delayed connection



```
[ connection_name : ] data port
<source_port_name> ->> <dest_port_name>
  [ { <property association> } ]
  [ in modes ( <mode or transition> { , <mode or transition> }* ) ];
```

- Port group connection



```
[ connection_name : ] port group
<source_group_name> -> <dest_group_name>
  [ { <property association> } ]
  [ in modes ( <mode or transition> { , <mode or transition> }* ) ];
```

- Port connection refinement

```
connection_name : refined to
( data port | event port | event data port | port group )
  [ { <property association> } ]
  [ in modes ( <mode or transition> { , <mode or transition> }* ) ];
```

# Parameter and access connection

- Parameter connection

```
[ connection_name : ] parameter <source_parameter> -> <dest_parameter>  
  [ { <property association> } ]  
  [ in modes ( <mode_name> { , <mode_name> }* ) ] ;
```

- Parameter connection refinement

```
connection_name : refined to parameter  
  [ { <property association> } ]  
  [ in modes ( <mode_name> { , <mode_name> }* ) ] ;
```

- Access connection

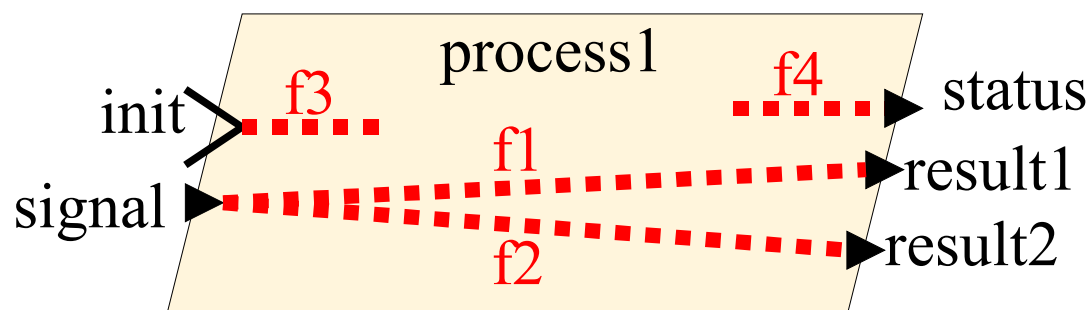
```
[ connection_name : ] ( bus | data ) access <provider> -> <requirer>  
  [ { <property association> } ]  
  [ in modes ( <mode_name> { , <mode_name> }* ) ] ;
```

- Access connection refinement

```
connection_name : refined to ( bus | data ) access  
  [ { <property association> } ]  
  [ in modes ( <mode_name> { , <mode_name> }* ) ] ;
```

# Flows

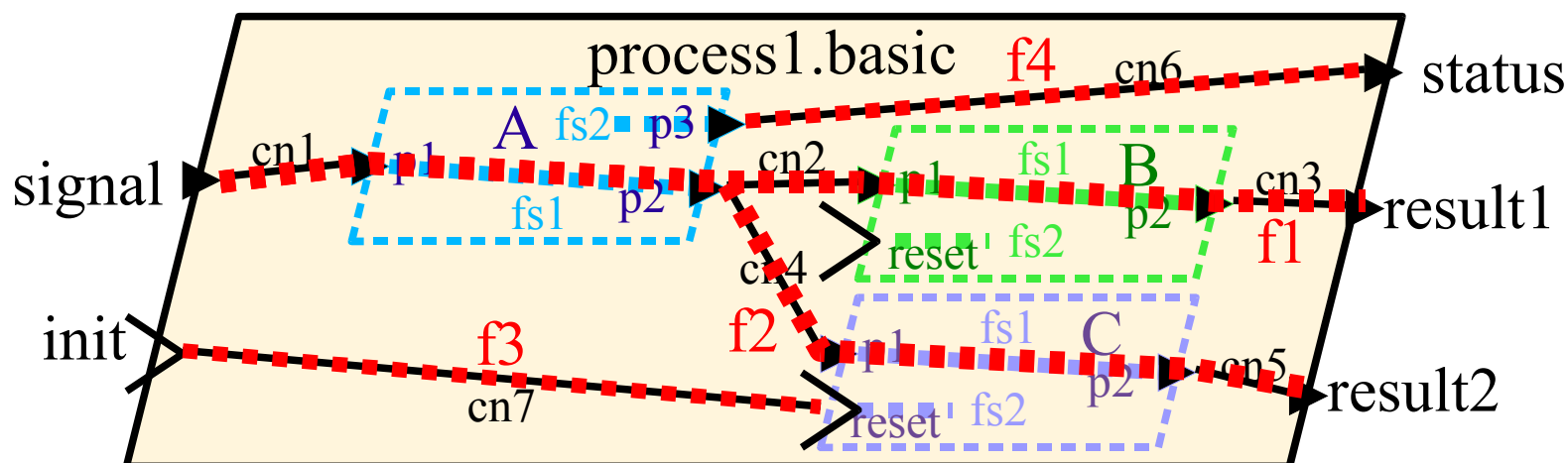
- Goal: enable flow analyses, such as timing, latency, reliability, quality of service, etc.
- Example of flow specification:



```

process process1
  features
    init: in event port;
    signal: in data port signal_data;
    result1: out data port position.radial;
    result2: out data port position.cartesian;
    status: out data port;
  flows
    f1: flow path signal -> result1;
    f2: flow path signal -> result2;
    f3: flow sink init;
    f4: flow source status;
end process1;
  
```

# Example of flow implementation



```
process implementation process1.basic
```

```
subcomponents
```

```
A: thread t1.basic; B: thread t2.basic; C: thread t2.basic;
```

```
connections
```

```
cn1: data port signal -> A.p1;
```

```
cn2: data port A.p2 -> B.p1;
```

```
cn3: data port B.p2 -> result1;
```

```
cn4: data port A.p2 -> C.p1;
```

```
cn5: data port C.p2 -> result2;
```

```
cn6: data port A.p3 -> status;
```

```
cn7: event port init -> C.reset;
```

```
flows
```

```
f1: flow path signal->cn1->A.fs1->cn2->B.fs1->cn3->result1;
```

```
f2: flow path signal->cn1->A.fs1->cn4->C.fs1->cn5->result2;
```

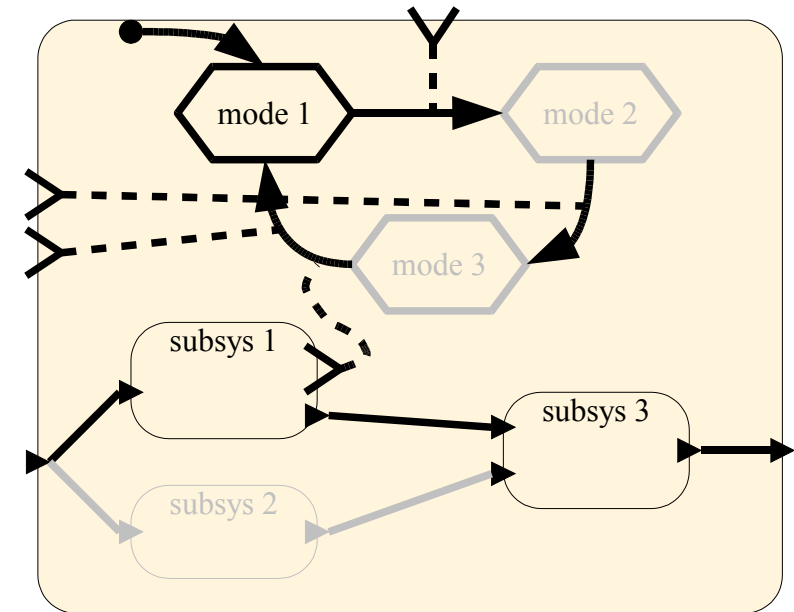
```
f3: flow sink init->cn7->C.fs2;
```

```
f4: flow source A.fs2->cn6->status;
```

```
end process1.basic;
```

# Mode

- A mode represents an operational state of a system.
- A component can have:
  - property values depending on the mode;
  - configuration of subcomponents and connections depending on the mode.



The mode transitions model the dynamic operational behaviour that represents switching between configurations.  
At each time, exactly one mode is the current mode.



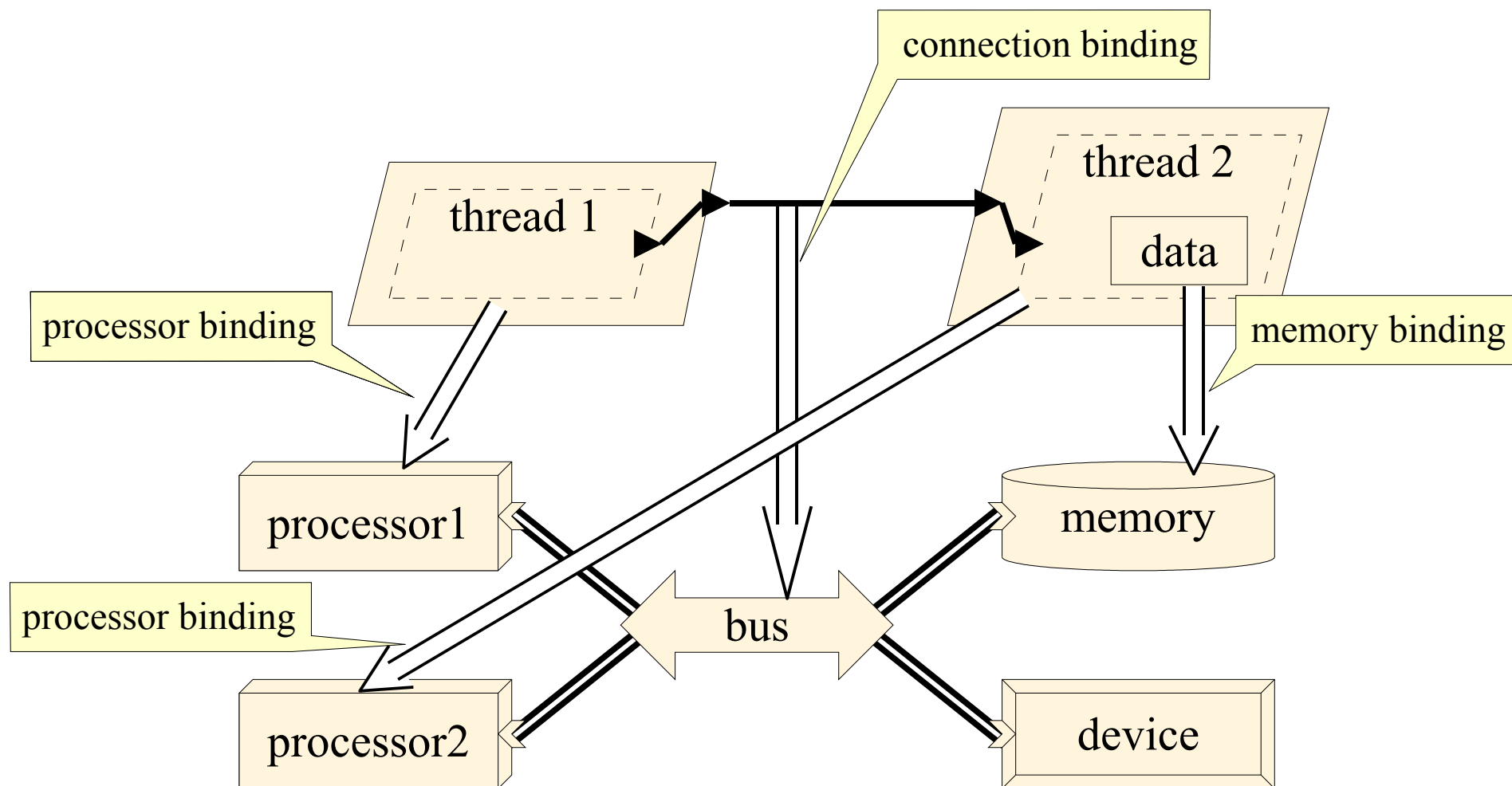
# Mode example

```
system implementation Gps.Dual
subcomponents
  Main_Gps : process Gps_Sender.Basic
    in modes ( Dualmode, Mainmode );
  Backup_Gps : process Gps_Sender.Basic
    in modes ( Dualmode, Backupmode );
  Monitor : process GPS_Health_Monitor;

connections
  data port Main_Gps.Position -> Position
    in modes ( Dualmode, Mainmode );
  data port Backup_Gps.Position -> Position
    in modes ( Backupmode );
  data port Backup_Gps.Position -> Main_Gps.SecondaryPosition
    in modes ( Dualmode );

modes
  Initialize : initial mode ;
  Dualmode   : mode ;
  Mainmode   : mode ;
  Backupmode : mode ;
  Initialize -[ Init_Done ]-> Dualmode ;
  Dualmode -[ Monitor.Backup_Stopped ]-> Mainmode ;
  Dualmode -[ Monitor.Main_Stopped ]-> Backupmode ;
  Mainmode, Backupmode -[ Monitor.All_Ok ]-> Dualmode ;
end Gps.Dual ;
```

# Binding





# Example of binding

```
system smp end smp;  
  
-- multiprocessor system  
system implementation smp.s1  
subcomponents  
  p1: processor cpu.u1;  
  p2: processor cpu.u1;  
  p3: processor cpu.u1;  
end smp.s1;  
  
process p1 end p1;  
  
process implementation p1.i1  
subcomponents  
  ta: thread t1.i1;  
  tb: thread t1.i1;  
end p1.i1;  
  
thread t1 end t1;  
  
thread implementation t1.i1  
end t1.i1;  
  
processor cpu end cpu;  
  
processor implementation cpu.u1  
end cpu.u1;
```

```
system S end S;  
  
-- system containing application  
-- components with execution platform  
-- components  
system implementation S.I  
subcomponents  
  p_a: process p1.i1;  
  p_b: process p1.i1;  
  up1: processor cpu.u1;  
  up2: processor cpu.u1;  
  ss1: system smp.s1;  
  
properties  
  Allowed_Processor_Binding =>  
    ( reference up1, reference up2 )  
    applies to p_a.ta;  
  Allowed_Processor_Binding =>  
    ( reference up1, reference up2 )  
    applies to p_a.tb;  
  Allowed_Processor_Binding =>  
    reference ss1.p3  
    applies to p_b.ta;  
  Allowed_Processor_Binding =>  
    reference ss1  
    applies to p_b.tb;  
end S.I;
```

# Package

- A *package* provides a means to organize the descriptions by the use of namespaces.
- A package can contain:
  - component types;
  - component implementations;
  - port group types;
  - annex libraries.

```
package my_package
  public
    ...
  private
    ...
end my_package ;
```

```
system my_package::my_component
```

# Annex

- An annex enables the use of declarations written in another language.
- An annex subclause can be used only in a component type and a component implementation.
- An annex library can be declared into a package. It can be referred to by an annex subclause if the annex language enables it.

```
thread Collect_Samples
features
  Input_Sample    : in data port Sampling::Sample;
  Output_Average : out data port Sampling::Sample;
annex
  OCL {**
    pre: 0 < Input_Sample < maxValue;
    post: 0 < Output_Sample < maxValue;
  **} ;
end Collect_Samples;
```