Stood

AADL Import/Export User Manual

Pierre Dissaux

Contents

The **A**rchitecture **A**nalysis and **D**esign **L**anguage (**AADL**) standards document was prepared by the **SAE AS-2C** Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division. The **AADL** standard is based on **MetaH**, an architecture description language developed at Honeywell Laboratories under the sponsorship of the US Defense Advanced Research Projects Agency (**DARPA**) and US Army Aviation and Missile Command (**AMCOM**). Release 1.0 of the **AADL** standard (**SAE** AS5506) has been issued in November 2004.

The **AADL** is a language used to describe the software and hardware **Components** of a system and the interfaces between those **Components**. The language can describe functional interfaces to **Components**, such as data inputs and outputs, and non-functional aspects of **Components**, such as timing. The language can describe how **Components** are combined, such as how data inputs and outputs are connected or how software **Components** are allocated to hardware **Components**. More detailed information about this language may be found at: `www.aadl.info`.

Stood is a software design tool that is used for the same kinds of mission critical systems as those for which the **AADL** has been developed. This manual describes the import and export features that have been included into **Stood**, in order to let software designers benefit from the **AADL**. Some of the most important advantages of **Stood** is that it offers a very good support of the modeling process, and brings a large set of development features that have already been in use on many large scale industrial projects.

With **Stood** it is thus possible to import specifications written in **AADL** to set up a preliminary software architecture, or to generate standard **AADL** descriptors of existing software **Components** for future reuse.

1. AADL mapping

Unlike for some other toolsets that are fully dedicated to the **AADL**, the support of the language in **Stood** is based on a point to point mapping between the **Stood** internal meta-model, which is a implementation of the **HOOD**, **HRT-HOOD and UML 2.0** standards, and the closest definitions in the **AADL** standard. Please, refer to the corresponding **AADL**, **HOOD** and **UML 2.0** standards definition for a more complete and precise description of these concepts. The support of the **AADL** into **Stood** thus mainly consists in a set of transformation rules, that are packaged inside external plugins for better flexibility. This chapter presents this mapping for **AADL Packages**, **Components**, **Features**, **Connections**, **Properties**, and other concepts. Variants in this mapping may be defined to comply with specific requirements. For instance the default transformation rules that have been defined for the import and export features may differ in a few points to best fit the actual need of each use case.

1.1. AADL components

A **HOOD System_Configuration** represents a software project. It refers to one or several cooperative programs (**Design**), libraries and software interface for hardware (**Environment**), and generic components (**Generic**). Each **Design** becomes an **Environment** within the scope of another **Design** of the same **System_Configuration**. This is the *logical view* of the project. In addition, it may contain one ore several **Virtual_Nodes** describing a distributed architecture of processors and an allocation table for the elements of the *logical view*. This is the *physical view* of the project which is only needed for distributed systems.

The **System_Configuration** is mapped to an **AADL System** instance, when the main **Design** aims to the production of an executable program, and thus represents a **Process**. **Environments** may represent other collaborative **Processes** within the same **System**, or **Packages** if they only aim to the production of libraries of types or classes. Further investigations will be performed to extend this mapping in order to support some other kinds of **Environments** as **Devices**, **Generics** as **Packages** and **Virtual_Nodes** as **Processors**.

A **HOOD Design** is the root of a hierarchy of **Modules**. Each **Module** is a well identified subset of the **Design** that is built in an iterative way, following a top-down decomposition process. This process must comply with the *information hiding*, *low-coupling* and *high consistency* rules promoted by the **HOOD** methodology.

The most natural mapping for a **HOOD Module** is an **AADL** software **Component**, although it will represent either a **Component Type** and **Implementation** only, if defined within a **Package**, or a **Component Type**, **Implementation** and a **Subcomponent**, if defined within a **Process**.

A **HOOD Module** may be **Passive** or **Active**. If it is **Active**, it contains its own thread of control. Two specialized kinds of **Active Modules** have been defined by **HRT-HOOD** to support **Hard Real-Time** architectures: **Cyclic Objects** to represent periodic activity, and **Sporadic Objects** that are triggered by an applicative event. A specialized kind of **Passive Module** has been defined to encapsulate shared data and their appropriate access procedures, they are called **Protected Objects**. There are other kinds of **Modules** like **Classes**, that represent object oriented data structures associated with their member functions, **Op_Controls**, that represent purely functional entities, and **Instances_Of** generic, that represent parameterized entities.

The table below summarizes the mapping between **HOOD Module** kinds and **AADL Component** categories:

AADL	**STOOD**
System instance	System_Configuration
Processor	Virtual_Node
Device	Environment
Process	Design
Package	Class or Environment
Thread (aperiodic)	Active Terminal
Thread (periodic)	Cyclic Terminal
Thread (sporadic)	Sporadic Terminal
Thread Group	Active Non_Terminal
Data	Protected or Passive Terminal
Subprogram	Op_Control

In **HOOD**, each **Module** (including the **Design**), has an interface and a body. The interface consists of the **Provided_Interface** that lists the declaration of the software elements that are implemented by the **Module** and made visible to be used by other **Modules**, and the **Required_Interface** that lists the references to all the remote software elements that are required to implement the **Module**. The body is called the **Internals** of the **Module** and contains either child **Modules** (for a **Non_Terminal Module**) or a list of declaration of private software elements and all the implementation (for a **Terminal Module**). Mapping with the corresponding **AADL** concepts is shown in the table below:

AADL	STOOD
Component Type	Module Interfaces
Features	Provided_Interface
Required Subcomponents	Required_Interface
Component Implementation	Module Internals
Subcomponents	Child Modules
Calls	OPeration Control Structure
Extends	Class Inheritance

Unlike the **AADL**, **HOOD** only distinguishes between **Modules** *type* and **Modules** *instances* in two particular cases only, the general case being that a **Module** is handled as an **Object**, that is, the unique instance of an anonymous *type*. The first particular case is when the **Module** *type* is described by a **Class** (like in **UML**), but then, instances of this **Class** (which are in effect only instances of the main data **Type** provided by the **Class**) becomes **Data**, **Constants**, etc... embedded somewhere inside another **Module**. The other particular case is when the **Module** *type* is described by a **Generic**, then, instances of this **Generic** (which are in effect only instances of the formal parameters) become other **Modules** called **Instance_Of**. **HOOD** also supports **Generic Classes** (similar to **C++** templates) that need to be instanciated twice. The other extension mecanism offered by **HOOD** is the **Class** Inheritance.

Practically, when a **HOOD Module** is defined as a child of a **Passive** parent, then it will be considered as being abstract, and it will be translated into a **Component Type** and **Implementation**. On the contrary, if it is a child of an **Active** parent, it will be translated into a **Subcomponent** and the corresponding **Component Type** and **Implementation**. **Subcomponents** which **Type** is defined within a remote **Package** should be modeled in **HOOD** by an **Instance_Of** a **Generic Module**. However this mapping is not implemented yet. **Component Type Extension** is currently available for **Data Components** only, and represented in **HOOD** by the **Class Inheritance** relationship.

1.2. AADL features

A **HOOD Module** may contain references and declarations to, and/or implementation of, software elements. These elements are: **Operations**, **Exceptions**, (data) **Types** and **Constants** in a **Provided_Interface**, and **Operations**, (data) **Types**, **Constants** and **Data** in the **Internals**. **Operations** declaration may specify **Parameters** and **Types** declaration may specify **Attributes**. **Constants**, **Data**, **Parameters** and **Attributes** are instances of a data **Type**. **Operations** may raise and handle **Exceptions**. As **Provided Data** are not allowed in **HOOD**, data flows can only be propagated between two **Modules** along client/server functional calls.

AADL	**STOOD**
Data Port	"getter" or "setter" Operation
Event Port	async. Oper. or Exception
Event Data Port	async. Oper. with parameters
Server Subprogram	synchronous Opereration
Data Subprogram	class member Operation
Data Access	Required or Used Object

By default, the execution request for an **Operation** is said to be unconstrained. However, it is possible to specify one or several **Operation Constraints** (trigger events) to describe more precisely the interaction between a client and a server. **Protocol Constraints** are used to specify the synchronization protocol between two concurrent threads of execution. These constraints are: **ASER** (asynchronous), **LSER** (synchronous, acknowledge), **HSER** (synchronous, wait-reply). The additional constraint **TO** specifies a time-out for **ASER** and **LSER**, and hardware interrupts are identified by the additionnal constraint **BY_IT** attached to an **ASER** constraint. **State Constraints** specify the receptivity of a service regarding the current **State** of the server and **Concurrency Constraints** can be used to manage mutual exclusion.

1.3. AADL connections

At architectural level, the interactions between **HOOD Modules** are described by **Use** relationships on both functional and structural views. A functional **Use** relationship defines a client/server interaction between **Operations** of the two **Modules**. They are the support for control flows, **DataFlows** (related to the **Parameters** of the called **Operations**) and **Exception_Flows** (related to the **Exceptions** that may be raised by the called **Operations**). A structural relationship defines a **Type** dependency (instanciation, aggregation and inheritance for **Classes**) between the two **Modules**. They are the support for the definition of **Attributes** (for any structured **Type**) and super-**Classes** (for **Classes**).

In **HOOD**, **Non Terminal Modules** are empty shells. That means that any element (**Type**, **Constant**, **Operation** or **Exception**) declared in the **Provided_Interface** of a **Non Terminal Module** must be **Implemented_By** a element of the same kind in the **Provided_Interface** of one of the child **Modules**.

AADL	**STOOD**
between sibling Components	Op_Use relationship
Component to Subcomponent	Implemented_By relationship
dot notation for Data Types	Type_Use relationship

1.4. AADL properties

HOOD offers a standard information structure for each **Module**. This structure is called the **ODS** (Object Description Skeleton) and provides low level details about the **Module** and its elements. The **ODS** contains:

- Textual sections to justify the design choices and the requirements traceability
- Real-Time attributes (period, priority, deadline, worst case execution time, etc...)
- The list of required remote elements (**Required_Interface**)
- Textual comment for each **Operation**, **Type**, etc...
- Source code for the declaration of each **Type**, **Constant**, or **Data** element
- Source code for the **OPCS** (procedural code of an **Operation**) and the **OBCS** (behavioral code of a **Module**)
- etc...

Many of these elements can represent **AADL Properties**. Currently, just a few of them are supported by **Stood**:

AADL	**STOOD**
Source_Text	naming rules
Source_Language	pragma Target_Language
Source_Name	naming rules
Dispatch_Protocol	kind of Module (cyclic, ...)
Period	HRT Attribute: Period
Compute_Deadline	HRT Attribute: WCET
Deadline	HRT Attribute: Deadline
Compute_Entrypoint	naming rules

1.5. AADL modes

The **Internals** of a **Terminal Module** contain the procedural code associated to each **Provided** or **Internal Operation**, within a structure called the **OPCS** (**OP**eration **C**ontrol **S**tructure). If there is at least one **Operation** that has a **Protocol** or **State Constraint**, the **Internals** of a **Terminal Module** will also contain the behavioral code associated to the corresponding thread and/or states-transitions model, within another structure called **OBCS** (**O**Bject **C**ontrol **S**tructure).

Internal Data may be shared by all the elements contained by the **Internals** of the **Module**. They can be used as **State** variables for the states-transitions model implemented in the **OBCS** of a **Terminal Module**. The only states that need to be specified in a **HOOD** states-transitions model are those defining areas of receptivity for the provided **Operations**. The execution request for a provided **Operation** becomes an event that triggers a transition (in the **OBCS**), before executing the appropriate code (in the **OPCS**). In **HOOD**, a states-transitions model can also be used to specify the execution modes for a whole **Design**. In that case, all the real-time attributes must have a known value for each mode.

AADL	**STOOD**
Mode	State at Root level
Mode Mode transition	Transition at Root level

2. AADL export

The process to export an **AADL** specification from an existing **HOOD Design** is similar to a target language code generation. This process consists in the following steps:

- Set **AADL** as the default language
- Load an existing **Design**, or create a new one.
- Open the **AADL** extractor window.
- Set **AADL** code generation options.
- Launch the generation, and view the generated code.

2.1. Set the default language

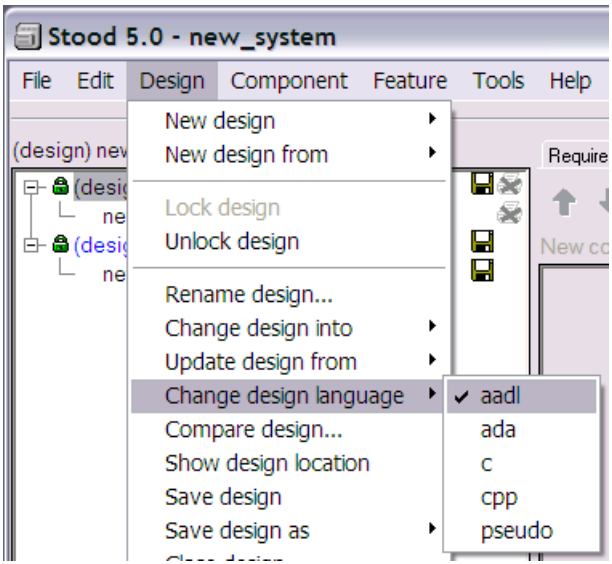
Stood is a muti-language modeling tool. It is thus possible to perform detailed design and coding activities for several target languages concurrently. Dedicated coding sections are available for that purpose for each **Component** design framework (**ODS**). In the case of the **AADL**, there are no dedicated coding sections, but *pseudo code* sections can be used if required.

However, there is generally one main target and it is possible to configure the tool in such a way that the corresponding functions are selected by default. For instance, open the right code generator while selecting the *code* tab.

There are two ways to select the default target language. First way is at a tool configuration level, by editing the initialization file that is loaded each time **Stood** is launched. The following lines show the properties that must be set for **Windows** (`bin.w32\stood.ini`) and **Unix** (`bin.*/.stoodrc`).

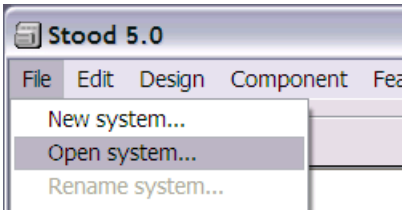
```
[General]
DefaultLanguage=aadl

General.DefaultLanguage:aadl
```

The second way consists in changing the main target language for the current **Design**. This can be done from the **Stood** menus, as shown below. This option will then be stored and saved at the same time as the **Design**.

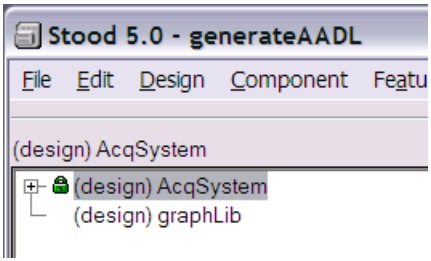


2.2. Load a design

It is always necessary to open a **System** before loading a **Design**. A given **Design** may however be loaded from several **Systems**. Use the *File* menu to create a new **System** or open an existing one. The list of **Designs** that are available for the **System** is then updated and it becomes possible to load one or several of them.



When just selected (single mouse click on the **Design** name), the **Design** is loaded but no change can be made (read-only mode). To enable changes, the **Design** must be locked by yourself (read-write mode). This can be done either by a double mouse click on the **Design** name, or by using the *Design/lock Design* menu. In that case, a green locker can be seen at the left of the **Design** name.



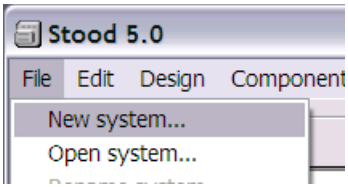
2.3. Create a new design

This section explains how to create most of the **AADL** constructs from the **Stood** graphical user interface.

2.3.1. Components and packages

2.3.1.1. Systems

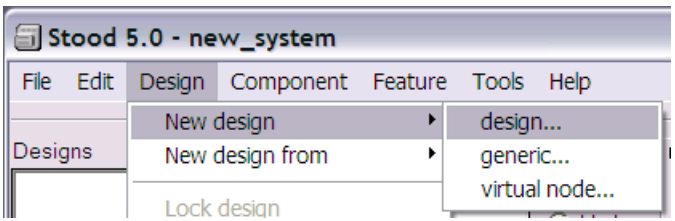
Any modeling activity must be done within a **System**. Just after having launched **Stood** or closed the currently opened **System**, a new one can be created from the *File* menu.



There is no direct **AADL** code generation for a **System**. The corresponding **AADL System** will be generated on top of a **Process** while generating code for a **Design**.

2.3.1.2. Packages

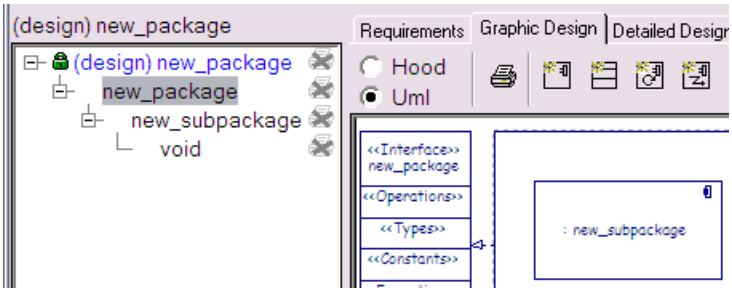
A new **Design** will be translated into a **Package** if it is **Passive** and if no **pragma** *main* has been defined before generating the code. A **Package** represents a library of reusable **Components**. Unlike within a **Process**, these **Components** are not instanciated.



If no **Component** has been defined inside the **Package**, then a dummy **Data Component Type** will be generated to comply with the **AADL** syntax:

```
PACKAGE new_package
PUBLIC

  DATA void
  END void;

END new_package;
```

In addition, if **Passive Objects** are created inside this **Design**, and are not empty, they will be translated into **AADL** sub **Packages**.

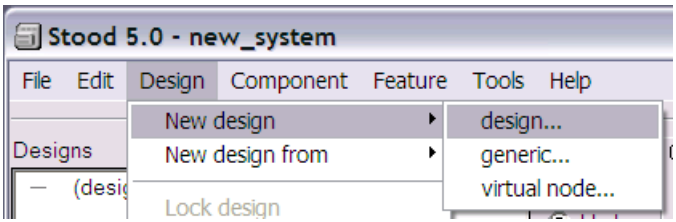


From the **Stood** model shown above, the corresponding **AADL** code will be generated as follow if the **pragma** *compact* has been set:

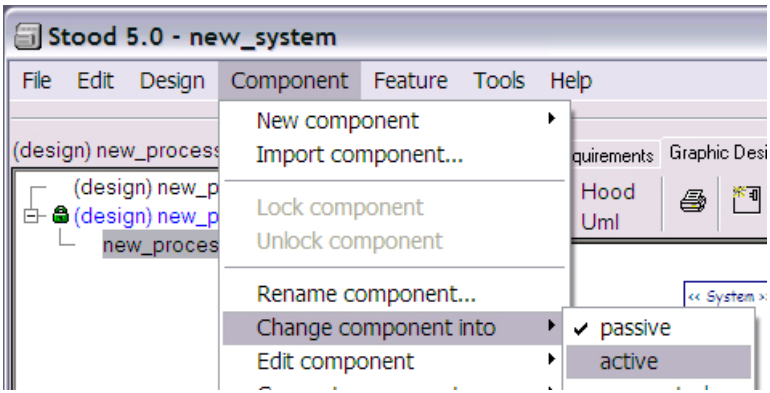
```
PACKAGE new_package
PUBLIC

  DATA void
  END void;

END new_package;

PACKAGE new_package::new_subpackage
PUBLIC

  DATA void
  END void;

END new_package::new_subpackage;
```

2.3.1.3. Processes

A new **Design** will be translated into a **Process** if it has been set to **Active** or if a **pragma** *main* has been defined for it.



To change the **Passive/Active** attribute of a **Component**, use the *Component/Change component into* menu:



The **AADL** code that is generated for an empty **Process** is as follow:

```
SYSTEM new_system
END new_system;

SYSTEM IMPLEMENTATION new_system.others
SUBCOMPONENTS
  new_process : PROCESS new_process;
END new_system.others;

PROCESS new_process
END new_process;
```

If a **pragma** *main* has been defined, where the *event* parameter must refer to an existing **in event port** of the **Process**,

```
PRAGMA main
    (event => --|start|--)
```

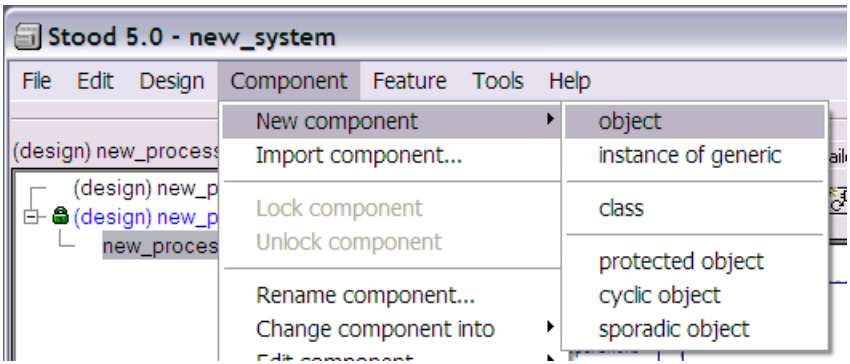
then the generated **AADL** code is modified as follow:

```
SYSTEM new_system
FEATURES
  start : IN EVENT PORT;
END new_system;

SYSTEM IMPLEMENTATION new_system.others
SUBCOMPONENTS
  new_process : PROCESS new_process;
CONNECTIONS
  EVENT PORT start -> new_process.start;
END new_system.others;
```

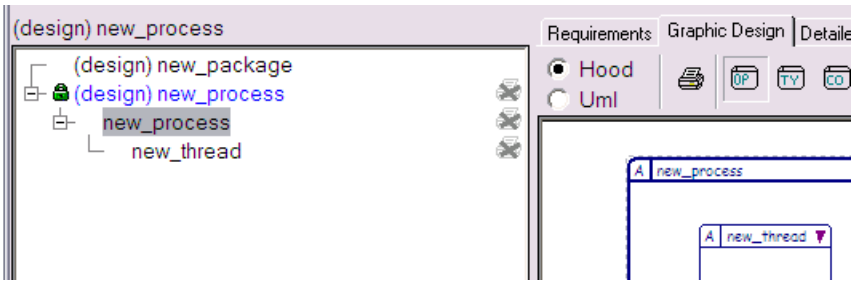
2.3.1.4. Threads

An **Active Object** will be translated into a **Thread Component Type** and **Implementation**, and a **Subcomponent** if it is located within a **Process**. It will be translated into a **Component Type** and **Implementation** only if it is created within a **Package**.

To create a new **Thread**, use the *Component/New component* menu and select either *object* (for an aperiodic **Thread**), *cyclic object* (for a periodic **Thread**) or *sporadic object* (for a sporadic **Thread**). If the container is **Active**, then the new **Object** will be created **Active** by default, else it is always possible to let it become **Active**, thanks to the *Change component into* menu.



The following diagram can then be displayed in the editing area. The **HOOD** view of the graphical architecture is shown here because it offers more details, but the corresponding **UML** representation can be obtained at any time just by switching the *HOOD/UML* radio button.

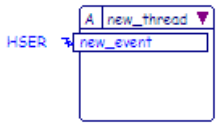


The generated **AADL** code for this architecture will be:

```
SYSTEM new_system
END new_system;

SYSTEM IMPLEMENTATION new_system.others
SUBCOMPONENTS
  new_process : PROCESS new_process;
END new_system.others;

PROCESS new_process
END new_process;

PROCESS IMPLEMENTATION new_process.others
SUBCOMPONENTS
  new_thread : THREAD new_thread;
END new_process.others;

THREAD new_thread
END new_thread;
```

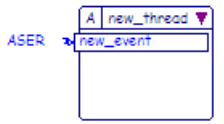
However, it is possible to get a more precise **AADL** code, using the predefined **HOOD** patterns:

If a **HSER** or **LSER Constrained Operation** is defined in the interface of an **Active Terminal Object**, then a **Server Subprogram** will be created for the corresponding **AADL Thread**, and the **Property** `Dispatch_Protocol => aperiodic` will be set.




```
THREAD new_thread
FEATURES
  new_event : SERVER SUBPROGRAM new_event
              { Compute_Entrypoint => new_event; };
PROPERTIES
  Dispatch_Protocol => aperiodic;
END new_thread;
```

If an **ASER Constrained Operation** is defined in the interface of an **Active Terminal Object**, then an **Event Port** will be created for the corresponding **AADL Thread**, and the **Property** `Dispatch_Protocol => sporadic` will be set.

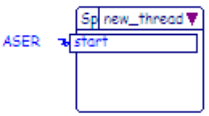



```
THREAD new_thread
FEATURES
  new_event : IN EVENT PORT;
PROPERTIES
  Dispatch_Protocol => sporadic;
END new_thread;
```

If a **Cyclic Object** is created, then an **Internal Operation** called `thread` is automatically defined, and the **Property** `Dispatch_Protocol => cyclic` will be set for the corresponding **AADL Thread**.


```
THREAD new_thread
PROPERTIES
  Dispatch_Protocol => periodic;
  Compute_Entrypoint => thread;
END new_thread;
```

If a **Sporadic Object** is created, then an **Internal Operation** called `thread` and a **Provided ASER Constrained Operation** called `start` are automatically defined, and the **Property** `Dispatch_Protocol => sporadic` will be set for the corresponding **AADL Thread**.

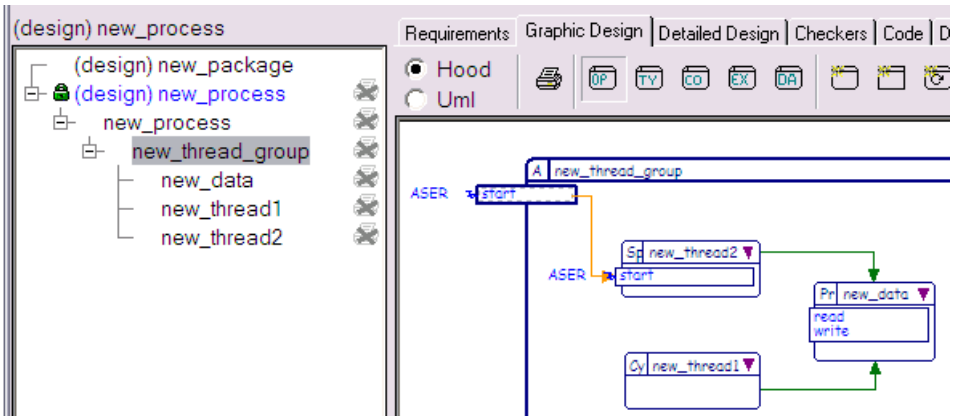



```
THREAD new_thread
FEATURES
  start : IN EVENT PORT;
PROPERTIES
  Dispatch_Protocol => sporadic;
  Compute_Entrypoint => thread;
END new_thread;
```

2.3.1.5. Thread groups

Thread Groups can be used to introduce intermediate levels of hierarchy in the architecture between a **Process** and the executing **Threads**. As it has been shown, a **Process** is mapped to a **Root Object** in the **Stood Component** hierarchy, whereas **Threads** are mapped to **Terminal Active**, **Cyclic** or **Sporadic Objects**. If **Non Terminal Active Objects** are defined, then they will be translated into **Thread Groups** in the **AADL** generated code.

A **Thread Group** must represent a logical subset of the real time software architecture. In compliancy with the **HOOD** modeling rules, a **Non Terminal Object** must encompass a set of subcomponents which are highly coupled, whereas external dependencies are minimized. For instance, a **Thread Goup** can be created to isolate a set of **Threads** communicating with a same **Protected Object**.



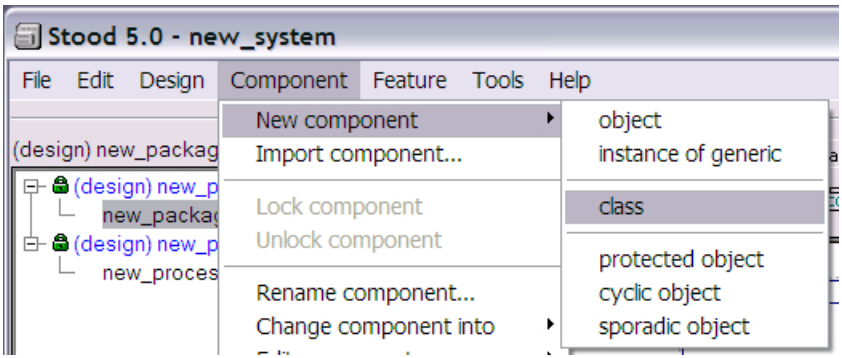
The corresponding generated **AADL** code will be as follow:

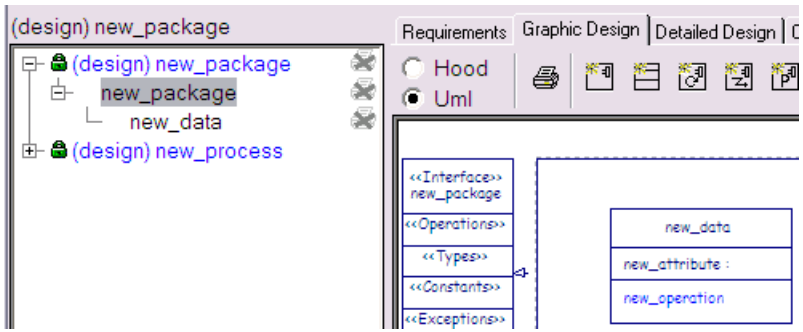
```
PROCESS new_process
FEATURES
  start : IN EVENT PORT;
END new_process;

PROCESS IMPLEMENTATION new_process.others
SUBCOMPONENTS
  new_thread_group : THREAD GROUP new_thread_group;
CONNECTIONS
  EVENT PORT start -> new_thread_group.start;
END new_process.others;

THREAD GROUP new_thread_group
FEATURES
  start : IN EVENT PORT;
END new_thread_group;

THREAD GROUP IMPLEMENTATION new_thread_group.others
SUBCOMPONENTS
  new_data : DATA new_data;
  new_thread1 : THREAD new_thread1;
  new_thread2 : THREAD new_thread2;
CONNECTIONS
  EVENT PORT start -> new_thread2.start;
  DATA ACCESS new_data -> new_thread1.new_data;
  DATA ACCESS new_data -> new_thread2.new_data;
END new_thread_group.others;
```

2.3.1.6. Data

Data Components can be handled by **Stood** in two different ways, depending on whether they represent abstract descriptions of data structures (**Classes**), or instances of such data structures (**Objects**). The former will be generated as **Data Components** within an **AADL Package**, wheareas the latter will be used to describe shared **Data Subcomponents** inside a **Process**.

When a **Class** is created within a **Passive Design**, it will be translated into a **Data Component** definition inside a **Package**. This **Class** can contain **Attributes** and **Operations** that will be displayed graphically with the **UML** or **HOOD** notations.





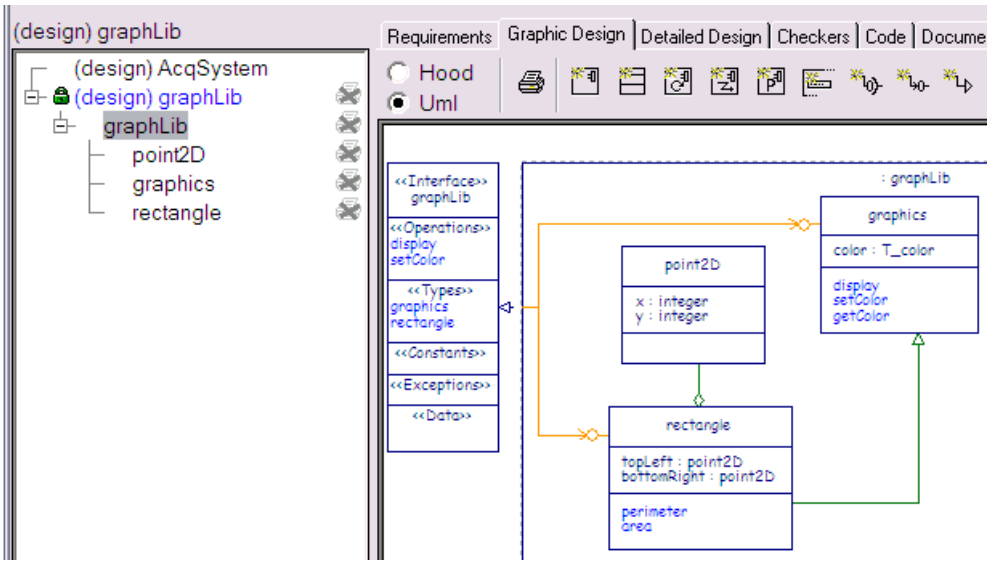
The corresponding generated **AADL** code is as follow:

```
PACKAGE new_package
PUBLIC

  DATA new_data
  FEATURES
    new_operation : SUBPROGRAM new_operation;
  END new_data;

  DATA IMPLEMENTATION new_data.others
  SUBCOMPONENTS
    new_attribute : DATA ;
  END new_data.others;

  SUBPROGRAM new_operation
  FEATURES
    me : IN OUT PARAMETER new_data;
  END new_operation;

END new_package;
```

Note that in case of several **Classes** defined in the same **Package**, they will all be generated as a flat structure within the same namespace. This may be an issue as several **Classes** may define **Operations** with the same name, because the **AADL** doesn't support **Subprogram** overloading. To solve this problem, it is possible to add the **pragma** *more_packages* to tell the code generator to create a separate sub **Package** for each **Class**:

```
PACKAGE new_package::new_data
...
END new_package::new_data;
```

Next example shows how a typical **UML** class diagram, that has been edited in **Stood**, can be translated into **AADL** code. The **pragma** *more_packages* has been used.

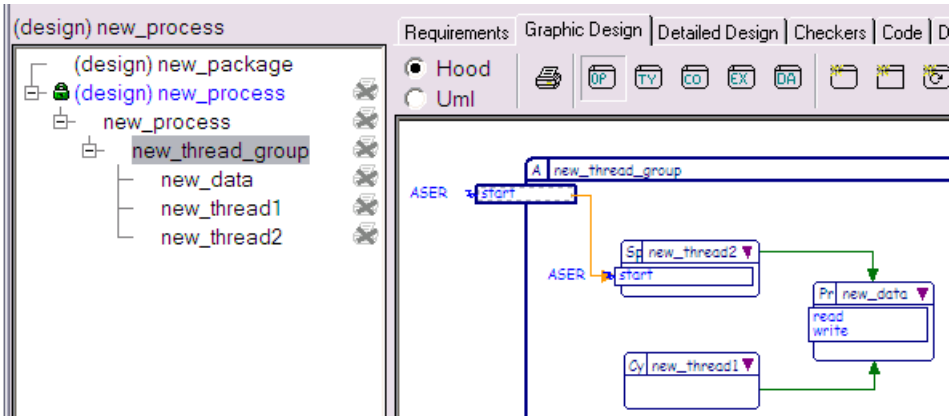



```
PACKAGE graphLib::point2D
PUBLIC

  DATA point2D
  END point2D;

  DATA IMPLEMENTATION point2D.others
  SUBCOMPONENTS
    x : DATA integer;
    y : DATA integer;
  END point2D.others;

END graphLib::point2D;
```

```
PACKAGE graphLib::graphics
PUBLIC

  DATA graphics
  FEATURES
    display : SUBPROGRAM display;
    setColor : SUBPROGRAM setColor;
    getColor : SUBPROGRAM getColor;
  END graphics;

  DATA IMPLEMENTATION graphics.others
  SUBCOMPONENTS
    color : DATA T_color;
  END graphics.others;

  SUBPROGRAM display
  FEATURES
    me : IN OUT PARAMETER graphLib::graphics;
  END display;

  SUBPROGRAM setColor
  FEATURES
    me : IN OUT PARAMETER graphLib::graphics;
  END setColor;

  SUBPROGRAM getColor
  FEATURES
    me : IN OUT PARAMETER graphLib::graphics;
  END getColor;

END graphLib::graphics;
```

```
PACKAGE graphLib::rectangle
PUBLIC

  DATA rectangle EXTENDS graphLib::graphics
  FEATURES
    perimeter : SUBPROGRAM perimeter;
    area : SUBPROGRAM area;
  END rectangle;

  DATA IMPLEMENTATION rectangle.others
  SUBCOMPONENTS
    topLeft : DATA point2D::point2D;
    bottomRight : DATA point2D::point2D;
  END rectangle.others;

  SUBPROGRAM perimeter
  FEATURES
    me : IN OUT PARAMETER graphLib::rectangle;
  END perimeter;

  SUBPROGRAM area
  FEATURES
    me : IN OUT PARAMETER graphLib::rectangle;
  END area;

END graphLib::rectangle;
```

On the contrary, when a **Passive** or **Protected Object** is created within an **Active Design**, it represents a shared **Data** instance. This **Object** is supposed to encompass the actual **Internal Data** and the corresponding **Provided** access **Operations**.



The corresponding **AADL** code will be:

```
THREAD GROUP IMPLEMENTATION new_thread_group.others
SUBCOMPONENTS
  new_data : DATA new_data;
  new_thread1 : THREAD new_thread1;
  new_thread2 : THREAD new_thread2;
CONNECTIONS
  EVENT PORT start -> new_thread2.start;
  DATA ACCESS new_data -> new_thread1.new_data;
  DATA ACCESS new_data -> new_thread2.new_data;
END new_thread_group.others;
```

```
DATA new_data
FEATURES
  read : SUBPROGRAM read;
  write : SUBPROGRAM write;
END new_data;

SUBPROGRAM read
END read;

SUBPROGRAM write
END write;
```

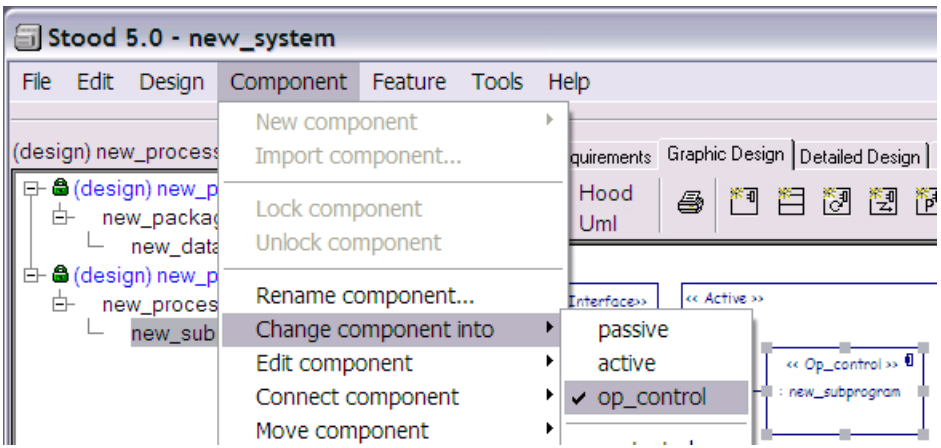
Note that when the **Object** also provides a **Type** that can be used to define the **Parameters** of the **Operations**, then the name of the corresponding **AADL Data Component Type** will be the name of this **Type** instead of the name of the **Object**.

```
DATA new_shared
FEATURES
  read : SUBPROGRAM read;
  write : SUBPROGRAM write;
END new_shared;

SUBPROGRAM read
FEATURES
  item : OUT PARAMETER new_shared;
END read;

SUBPROGRAM write
FEATURES
  item : IN PARAMETER new_shared;
END write;
```

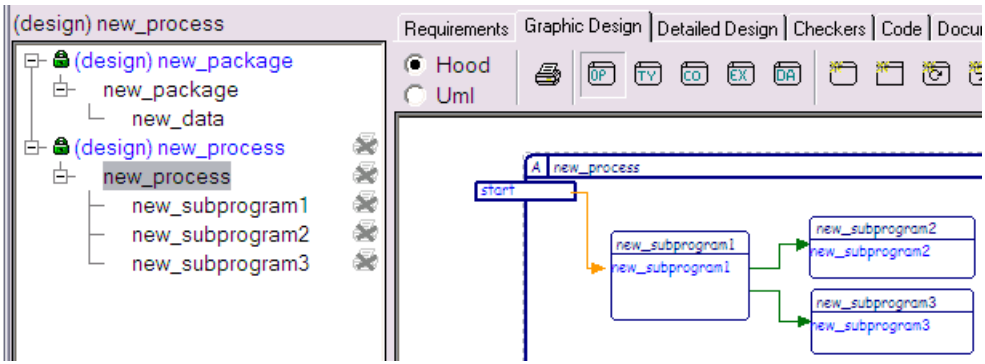
2.3.1.7. Subprograms as components

In most cases, **Subprogram Components** will be automatically generated from the corresponding **Subprogram Features** definition. However it is possible to explicitely create a **Subprogram Component Type** by adding an **Op_Control Object** inside a **Process** or a **Thread Group**.

To create an **Op_Control Object**, first create a plain **Object** and then change it into **Op_Control** with the relevant menu:



By creating several **Op_Controls** and **Use** links between them, it is possible to describe functional call sequences that will be translated into a corresponding **AADL Call** subsection.



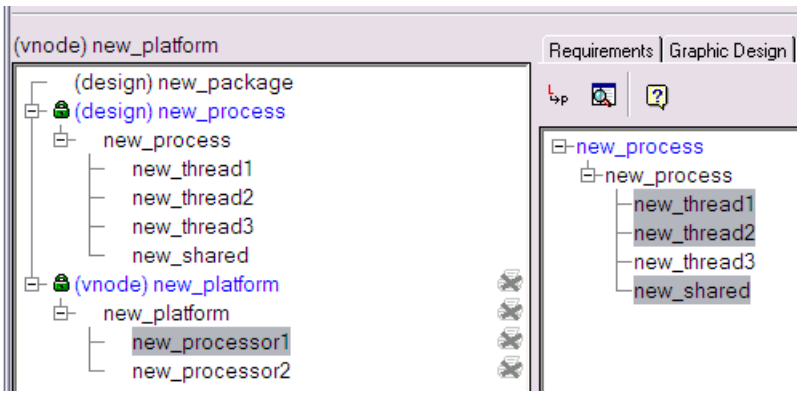

```
PROCESS new_process
FEATURES
  start : SUBPROGRAM new_subprogram1;
END new_process;

SUBPROGRAM new_subprogram1
END new_subprogram1;

SUBPROGRAM IMPLEMENTATION new_subprogram1.others
CALLS {
  new_subprogram2 : SUBPROGRAM new_subprogram2;
  new_subprogram3 : SUBPROGRAM new_subprogram3;
};
END new_subprogram1.others;

SUBPROGRAM new_subprogram2
END new_subprogram2;

SUBPROGRAM new_subprogram3
END new_subprogram3;
```

2.3.1.8. Execution platform components

There is currently no direct support of the **AADL** execution platform **Components** in **Stood**. However, the two following particular cases may be efficiently represented in the context of a pure software development process.

Firstly, the hardware environment of the software being developed with **Stood** can be shown as **Environment Objects**. These **Objects** are located within the **System**, at the same level as the **Design**, and can be used to represent **AADL Device Components**, which interface are required by the **Process**. However, there will be no corresponding specific code generation for now.

Secondly, it will be possible in the future to describe the deployment of the software on a multi-processors architecture thanks to the concept of **Virtual_Nodes** that is supported by **Stood**.

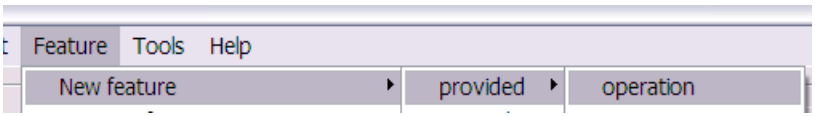


The result of the allocation of **Threads** onto **Processors** could be used to generate the appropriate **AADL** binding **Properties**.

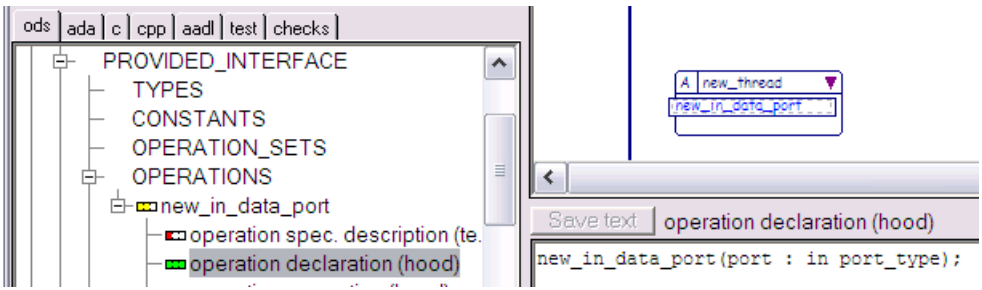
2.3.2. Features

2.3.2.1. Ports

AADL Ports represent variables that can be used for the transfer of control and data between **Threads**. Like all the other **AADL Features**, they are defined within the public interface of a **Component**. This implementation is not compatible with the strong information hiding principle that is promoted by **Stood**. In order to comply with these rules, **AADL Ports** must be represented in **Stood** by an **Internal Data** item and a **Provided** getter or setter **Operation**. Practically, the only definition of the **Operation** in the **Provided Interface** is sufficient to generate the proper **AADL** output. To create a new **Operation**, first select the target **Object** in the diagram, and then use the menu *Feature/New Feature/Provided/Operation.*



It is then possible to edit the **Operation** signature within the text area that is shown below the diagram. Note that an **Ada**-like syntax is used by **Stood** to define **Operation** signatures:



To create an **In Data Port**, specify an **In Parameter** with the corresponding data **Type**, for instance:

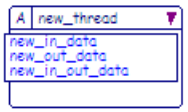
```
new_in_data(port : in port_type);
```

To create an **Out Data Port**, specify an **Out Parameter** with the corresponding data **Type**, for instance:

```
new_out_data(port : out port_type);
```

To create an **In Out Data Port**, specify an **In Out Parameter** with the corresponding data **Type**, for instance:

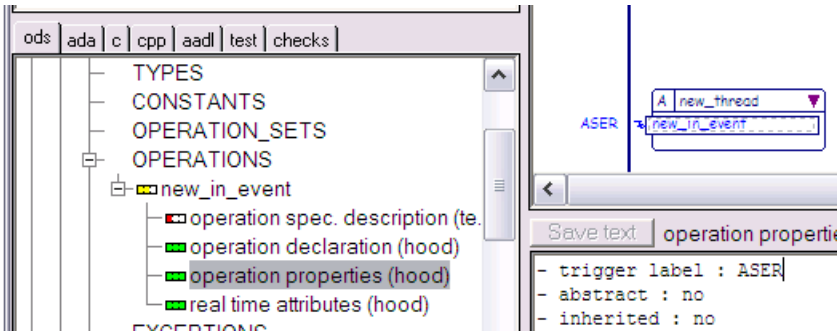
```
new_in_out_data(port : in out port_type);
```



The **AADL** code that is generated by **Stood** for such a **Thread** is shown below:

```
THREAD new_thread
FEATURES
  new_in_data_port : IN DATA PORT port_type;
  new_out_data_port : OUT DATA PORT port_type;
  new_in_out_data_port : IN OUT DATA PORT port_type;
END new_thread;
```

In order to represent **Event** or **Event Data Ports**, it is necessary to specify that the **Operation** execution is triggered by an **ASER** (**AS**ynchronous **E**xecution **R**equest) event. This can be done inside the *operation properties* section as shown below. Note that, as a special case, **Out Event Ports** are represented in **Stood** by **Exceptions** instead of **Operations**:

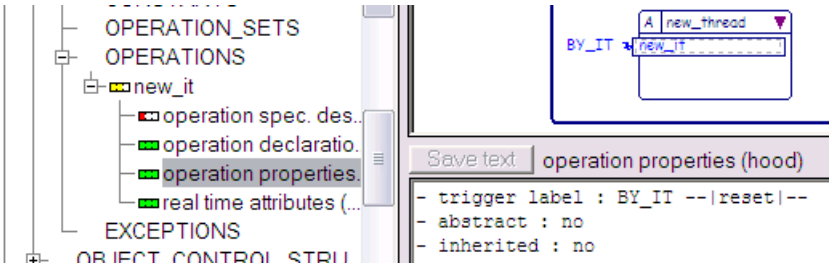


It is of course possible to combine a signature and a trigger to represent **Event Data Ports**. The **Out Event Port** can be created just by inserting an **Exception**:

operation view *exception view*

```
THREAD new_thread
FEATURES
  new_in_event : IN EVENT PORT;
  new_in_event_data_port : IN EVENT DATA PORT
                                        port_type;
  new_out_event_data_port : OUT EVENT DATA PORT
                                        port_type;
  new_in_out_event_data_port : IN OUT EVENT DATA PORT
                                        port_type;
  new_out_event : OUT EVENT PORT;

PROPERTIES
  Dispatch_Protocol => sporadic;
END new_thread;
```

Note that **Stood** manages **System** interrupts as a special case. In fact, the supported methodology recommends not to propagate **IT** events along the composition hierarchy, as opposed to the applicative events. A specific **Operation Constraint** (**BY_IT**) has been defined to denote that the corresponding event is triggered by a **System** interrupt. This trigger label must be associated to a parameter representing the interrupt vector or identifier.

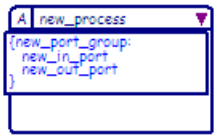


In that case, the **AADL** code generator will automatically insert all the higher level **Ports** and **Connections** along the containment hierarchy, so that the **IT** can be actually seen as being transmitted by an **Out Event Port** of an execution platform **Component** and received by an **In Event Port** of the **Process**.

```
PROCESS new_process
FEATURES
  reset : IN EVENT PORT;
END new_process;

PROCESS IMPLEMENTATION new_process.others
SUBCOMPONENTS
  new_thread : THREAD new_thread;
CONNECTIONS
  EVENT PORT reset -> new_thread.new_it;
END new_process.others;
```

2.3.2.2. Port groups

Port Groups are currently not supported by the **AADL** code generator of **Stood**. However, due to the mapping between **Ports** and **Operations**, the appropriate representation of **Port Groups** will be **Operation Sets** that can be used in **Stood** to group a set of **Operations**.



Note that this mapping between **Port Groups** and **Operation Sets** is implemented in the **AADL** import function of **Stood**.

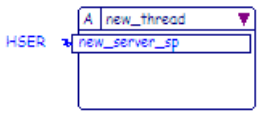
2.3.2.3. Subprograms as features

As opposed to **Processes**, **Thread Groups**, **Threads** and **Data**, **Subprograms** cannot be instanciated as **Subcomponents** within a **Component Implementation**. They must be declared as **Features** in a **Component Type**. There are two kinds of **Subprogram Features**: **Data Subprograms** that can be declared in the interface of **Data Components** and **Server Subprograms** that can be declared especially in the interface of **Processes**, **Thread Groups** and **Threads**.

Operations declared in the interface of a **Class** or a **Passive Terminal Object** or a **Protected Object**, will be translated into **Data Subprogram Features**. In addition, if the specified **Subprogram Component Type** doesn't exist, it will be also created.


```
DATA new_class
FEATURES
  new_method : SUBPROGRAM new_method;
END new_class;

DATA IMPLEMENTATION new_class.others
SUBCOMPONENTS
  new_attribute : DATA ;
END new_class.others;

SUBPROGRAM new_method
FEATURES
  me : IN OUT PARAMETER new_class;
END new_method;
```

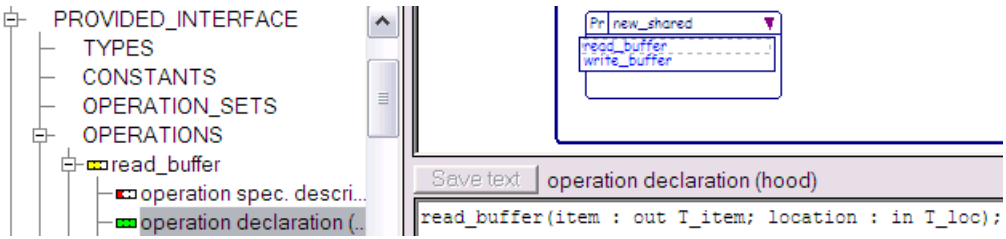
Operations declared in the interface of an **Active Object** and to which a **S**ynchronous **E**xecution **R**equest trigger event has been set (**HSER** or **LSER**), will be translated into **Server Subprogram Features**. In order to specify that the **Subprogram** code must be executed instead of the default **Thread** execution code, a `Compute_Entrypoint` **Property** is automatically added to the **AADL** specification. In addition, if the specified **Subprogram Component Type** doesn't exist, it will be also created.



```
THREAD new_thread
FEATURES
  new_server_sp : SERVER SUBPROGRAM new_server_sp
         { Compute_Entrypoint => new_server_sp; };
PROPERTIES
  Dispatch_Protocol => aperiodic;
END new_thread;

SUBPROGRAM new_server_sp
END new_server_sp;
```

2.3.2.4. Subprogram parameters

In **Stood**, **Operation Parameters** must be declared using an **Ada** like syntax. This may be done within the *operation declaration* section:

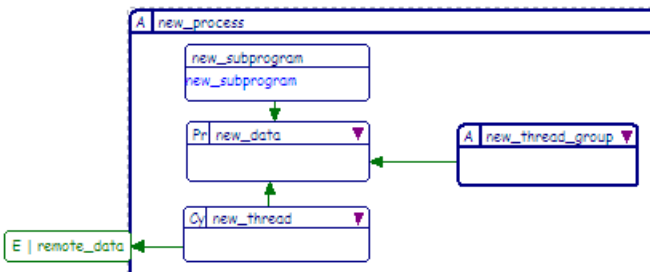


The corresponding **AADL** code will be generated as follow:

```
SUBPROGRAM read_buffer
FEATURES
  item : OUT PARAMETER T_item;
  location : IN PARAMETER T_loc;
END read_buffer;
```

2.3.2.5. Subcomponent access

The **AADL** code generator of **Stood** will produce a **Requires Data Access Feature** in a **Process**, **Thread Group**, **Thread** or **Subprogram Component** to denote a reference to a remote shared **Data Component**. Note that **Provides Data Access** and **Bus Access Features** are not currently supported by **Stood**. To produce a **Requires Data Access Feature**, **Use** dependencies must be drawn on the diagram, as shown below:



The **AADL** code that is generated for this set of **Components** is as follow:

```
PROCESS new_process
FEATURES
  remote_data : REQUIRES DATA ACCESS remote_data;
END new_process;

THREAD new_thread
FEATURES
  new_data : REQUIRES DATA ACCESS new_data;
  remote_data : REQUIRES DATA ACCESS remote_data;
PROPERTIES
  Dispatch_Protocol => periodic;
END new_thread;
```

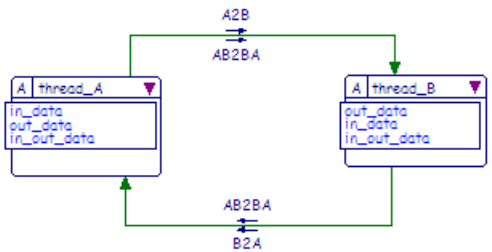
```
THREAD GROUP new_thread_group
FEATURES
  new_data : REQUIRES DATA ACCESS new_data;
END new_thread_group;

THREAD inner_thread
FEATURES
  new_data : REQUIRES DATA ACCESS new_data;
END inner_thread;

SUBPROGRAM new_subprogram
FEATURES
  new_data : REQUIRES DATA ACCESS new_data;
END new_subprogram;

DATA new_data
END new_data;
```

2.3.3. Connections

2.3.3.1. Between sibling components

AADL Connections between sibling **Components** are represented by graphical relationships between the corresponding **Objects** in the diagram. Like for all the other graphical edition functions, most constructs can be shown either with the **UML** or the **HOOD** notation. However, the **HOOD** view often shows more details for **Ports** and **Connections**.

Ports Connections can be defined with **Use** relationships on the **Operation** view. **Use** relationships are directional and will only connect **Out** and **In Out Ports** from the origin to **In** and **In Out Ports** of the destination. Two opposed links are thus necessary to fully connect two **Components** with compatible composite interfaces. **Dataflow** labels may be added to the graphical notation to better highlight the **Data** items that are propagated by the **Ports**.

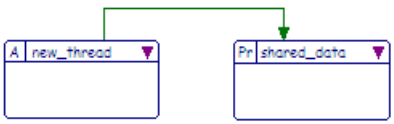


When such links are defined, point to point mapping between **Ports** are established in regards to the **Port** names and/or the propagated **Data** names. When several **Ports** propagate **Data** of the same name, a concatenation of the corresponding **Port** and **Data** names will be done to avoid ambiguities.

```
PROCESS IMPLEMENTATION new_process.others
SUBCOMPONENTS
  thread_A : THREAD thread_A;
  thread_B : THREAD thread_B;
CONNECTIONS
  DATA PORT thread_B.B2A -> thread_A.B2A;
  DATA PORT thread_B.AB2BA -> thread_A.AB2BA;
  DATA PORT thread_A.A2B -> thread_B.A2B;
  DATA PORT thread_A.AB2BA -> thread_B.AB2BA;
END new_process.others;

THREAD thread_A
FEATURES
  B2A : IN DATA PORT T_port;
  A2B : OUT DATA PORT T_port;
  AB2BA : IN OUT DATA PORT T_port;
END thread_A;

THREAD thread_B
FEATURES
  B2A : OUT DATA PORT T_port;
  A2B : IN DATA PORT T_port;
  AB2BA : IN OUT DATA PORT T_port;
END thread_B;
```

Note that the current version of the **AADL** code generator doesn't support **Parameter Connections**.

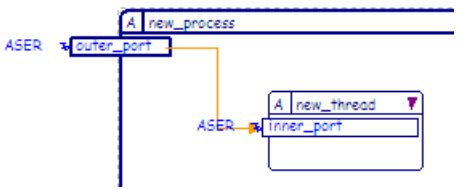
Access Connections can also be represented by simple **Use** relationships between **Processes**, **Thread Groups**, **Threads** or **Subprograms** and shared **Data Components**.




```
PROCESS new_process
END new_process;

PROCESS IMPLEMENTATION new_process.others
SUBCOMPONENTS
  new_thread : THREAD new_thread;
  shared_data : DATA shared_data;
CONNECTIONS
  DATA ACCESS shared_data -> new_thread.shared_data;
END new_process.others;

THREAD new_thread
FEATURES
  shared_data : REQUIRES DATA ACCESS shared_data;
END new_thread;

DATA shared_data
END shared_data;
```

2.3.3.2. Up and down the containment hierarchy

Ports Connections along the containment hierarchy can be defined by the **Implemented_By** links (also called **Delegate** in **UML** 2.0) between the **Provided Interface** of a container **Component** and the **Provided Interface** of contained **Components**.




```
PROCESS new_process
FEATURES
  outer_port : IN EVENT PORT;
END new_process;

PROCESS IMPLEMENTATION new_process.others
SUBCOMPONENTS
  new_thread : THREAD new_thread;
CONNECTIONS
  EVENT PORT outer_port -> new_thread.inner_port;
END new_process.others;

THREAD new_thread
FEATURES
  inner_port : IN EVENT PORT;
PROPERTIES
  Dispatch_Protocol => sporadic;
END new_thread;
```

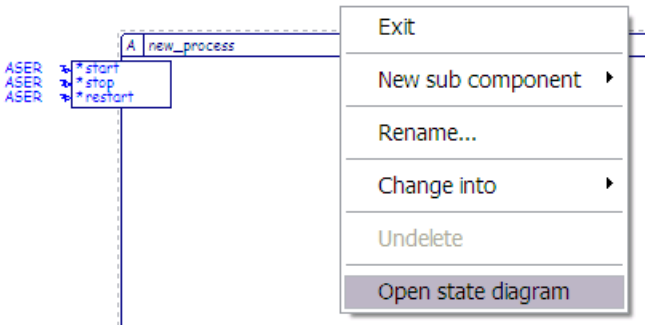
Required Access Connections may also be defined between the **Required Interface** of a container **Component** and the **Required Interface** of contained **Components**. This especially occurs when an inner **Thread** requires access to a share **Data Component** that is also used by the **Process** at a higher level. This pattern can be described with **Stood** thanks to the **Use Uncle** relationship:

at higher level *at lower level*

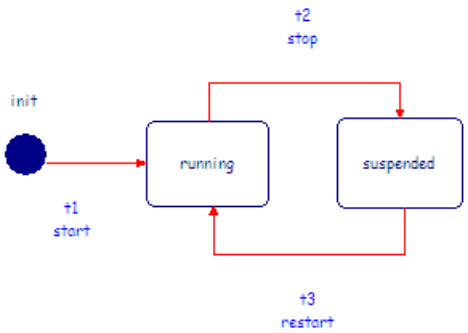
```
PROCESS new_process
FEATURES
  remote_data : REQUIRES DATA ACCESS remote_data;
END new_process;

PROCESS IMPLEMENTATION new_process.others
SUBCOMPONENTS
  new_thread : THREAD new_thread;
CONNECTIONS
  DATA ACCESS remote_data -> new_thread.remote_data;
END new_process.others;

THREAD new_thread
FEATURES
  remote_data : REQUIRES DATA ACCESS remote_data;
END new_thread;
```

2.3.4. Operational modes

Operational **Modes** may be defined by creating a **State Transition Diagram** (**STD**) for the current **Process**. **States** will be used to represent **Modes** and **In Event Ports** defined within the **Provided Interface** of the **Process** will act as **Transition** events. To create a **STD**, select the **Process** and use the contextual menu *Open state diagram*.



It is then possible to create **States** and **Transitions** and to allocate one of the **Event Ports** to each **Transition**.



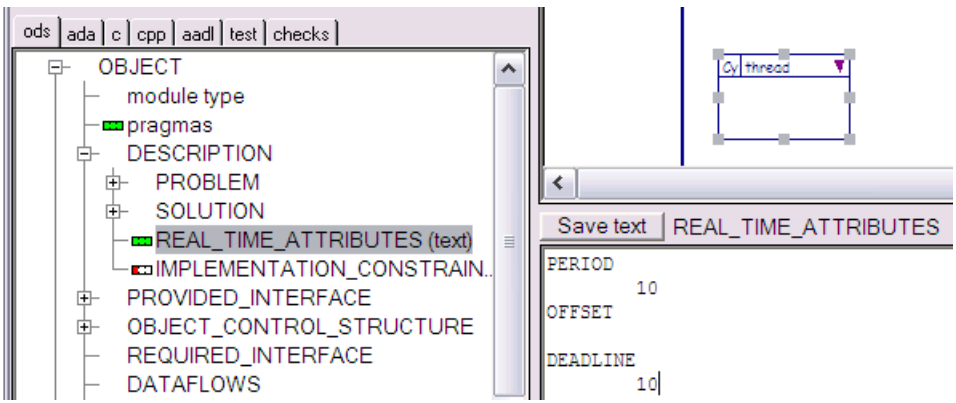
This information will be used by the **AADL** code generator to produce operational modes:

```
SYSTEM new_system
END new_system;

SYSTEM IMPLEMENTATION new_system.others
SUBCOMPONENTS
  new_process : PROCESS new_process;
END new_system.others;

PROCESS new_process
FEATURES
  start : IN EVENT PORT;
  stop : IN EVENT PORT;
  restart : IN EVENT PORT;
END new_process;

PROCESS IMPLEMENTATION new_process.others
MODES
  init : INITIAL MODE;
  running : MODE;
  suspended : MODE;
  init -[ start ]-> running;
  running -[ stop ]-> suspended;
  suspended -[ restart ]-> running;
END new_process.others;
```

2.3.5. Properties

Real Time Attributes that have been specified in the **Stood** model will be translated into closest corresponding **AADL Properties**.

example:



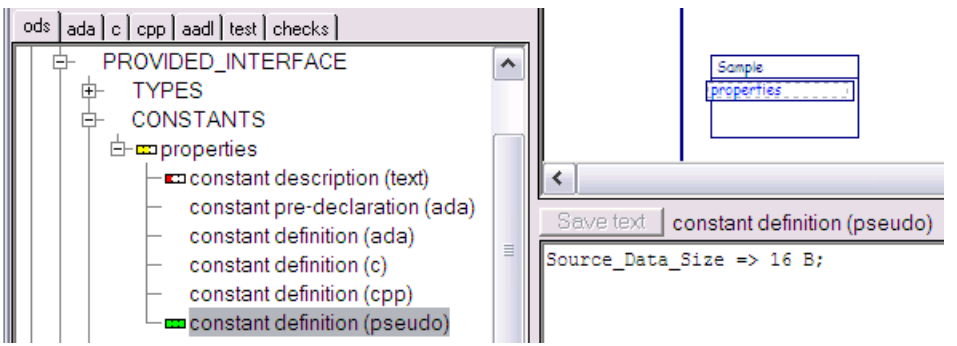
The **AADL** code generated by **Stood** will be:

```
THREAD task
PROPERTIES
  Dispatch_Protocol => periodic;
  Period => 10;
  Deadline => 10;
  Compute_Entrypoint => thread;
END task;
```

In a similar way, the **WCET** attribute that can be set for each **Operation** will be translated into a `Compute_Deadline` **Property**.

When other **Properties** are required, they must be included into the **Stood** model as a specific **Constant** which name is **Properties**. The **pseudo code** section can be used to insert **AADL Property** associations that will be inserted within the generated code.

example:

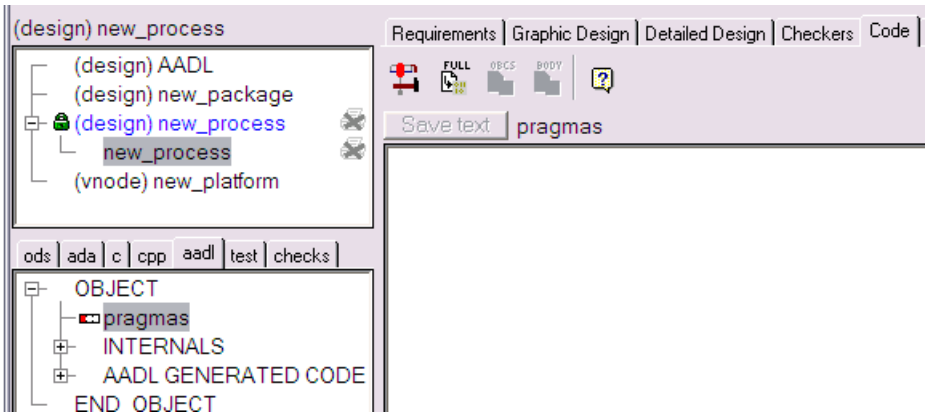


The **AADL** code generated by **Stood** will be:

```
DATA Sample
PROPERTIES
  Source_Data_Size => 16 B;
END Sample;
```

2.4. Set generation options

When ready to generate **AADL** code from the current **Design**, switch **Stood** to the *code* tab as shown below. If *aadl* has been specified as the default target language, then this will automatically open the **AADL** code generator. Else, it may be necessary to reselect the *aadl* tab in the bottom right area of **Stood.**



The *code* window gives access to a few generation options (**pragmas**), that are described below. To set an option, first select the **Module** on which it should apply, then click on the *add pragma* button  , and select the appropriate **pragma** in the list. When a **pragma** is set, its name is preceded by a >> tick. The list of all the currently set **pragmas** is shown in the editing area where it is possible to remove or dupplicate them, and change the value of their arguments. Supported **pragmas** for the **AADL** code generator are listed in the next sections.

2.4.1. Pragma os

This **pragma** can be set to specify the name of the Operating System that is used in the **System**. Default name is `OS`. It must be set for the **Root Module** only.

example:

If the **pragma** `os(operating_system => ARINC653)` is set, then the following **AADL** code will be generated:

```
SYSTEM new_system
END new_system;

SYSTEM IMPLEMENTATION new_system.others
SUBCOMPONENTS
  ARINC653 : SYSTEM OS.ARINC653;
  new_process : PROCESS new_process;
END new_system.others;
```

2.4.2. Pragma main

This **pragma** can be used to specify the main functional entry point for the current **System**. The parameter value must refer to the name of one of the **Provided Operations** of the main **Process**. This **pragma** must be set for the **Root Module** only.

example:

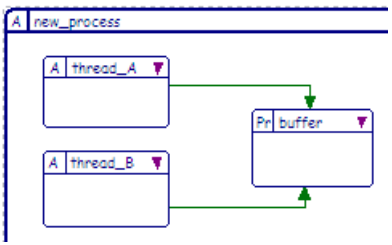
If the **pragma** `main(event => start)` is set, then the following **AADL** code will be generated:

```
SYSTEM new_system
FEATURES
  start : IN EVENT PORT;
END new_system;

SYSTEM IMPLEMENTATION new_system.others
SUBCOMPONENTS
  new_process : PROCESS new_process;
CONNECTIONS
  EVENT PORT start -> new_process.start;
END new_system.others;

PROCESS new_process
FEATURES
  start : IN EVENT PORT;
  stop : IN EVENT PORT;
  restart : IN EVENT PORT;
END new_process;
```

2.4.3. Pragma compact

This **pragma** can be set to ask the generator to create only one **AADL** source code file for the whole **Design** (**Package** or **Process**). The default option, when the **pragma** is reset, is to generate one **AADL** file for each **Component** or **Package**. This **pragma** must be set for the **Root Module** only.

example:



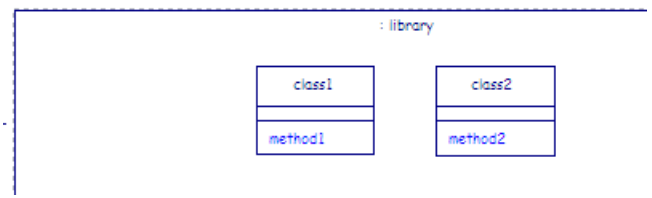
With such an architecture, default code generation would produce four files:
- `new_process.aadl`
- `thread_A.aadl`
- `thread_B.aadl`
- `buffer.aadl`

However, if the **pragma** *compact* is set, only the file `new_process.aadl` will be produced and will contain all the **AADL** statements for the four **Components**.

2.4.4. Pragma more_packages

Stood can use two different coding patterns to produce **AADL** code from a set of **Classes**. Default rule consists in generating a flat list of **Data** and **Subprogram Components** that are located within a container **Package**. However it may be better in some cases to create a specific sub **Package** for each **Class**. This is the purpose of the **pragma** *more_packages* that must be set for the **Root Module** only.

example:



The default code generation rules will produve the following **AADL** code:

```
PACKAGE library
PUBLIC

  DATA class1
  FEATURES
    method1 : SUBPROGRAM method1;
  END class1;

  DATA class2
  FEATURES
    method2 : SUBPROGRAM method2;
  END class2;
```

```
  SUBPROGRAM method1
  FEATURES
    me : IN OUT PARAMETER class1;
  END method1;

  SUBPROGRAM method2
  FEATURES
    me : IN OUT PARAMETER class2;
  END method2;

END library;
```

As the **AADL** doesn't support **Operation** overloading, there could have been a problem if the two methods had the same name. On the contrary, with a **pragma** *more_packages* properly set, the generated code will become:

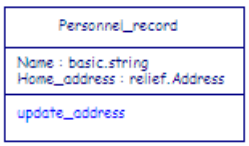
```
PACKAGE library
PUBLIC

  DATA void
  END void;

END library;
```

```
PACKAGE library::class1
PUBLIC

  DATA class1
  FEATURES
    method1 : SUBPROGRAM method1;
  END class1;

  SUBPROGRAM method1
  FEATURES
    me : IN OUT PARAMETER class1;
  END method1;

END library::class1;

PACKAGE library::class2
PUBLIC

  DATA class2
  FEATURES
    method2 : SUBPROGRAM method2;
  END class2;

  SUBPROGRAM method2
  FEATURES
    me : IN OUT PARAMETER class2;
  END method2;

END library::class2;
```

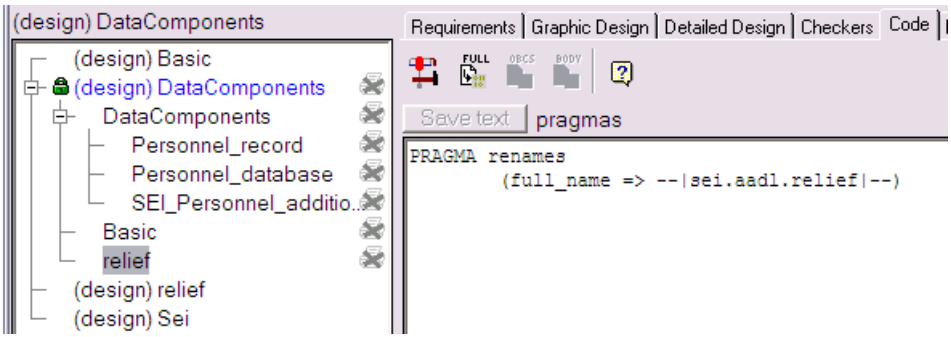
2.4.5. Pragma renames

When referencing an existing **Component Type** or **Implementation** to instanciate a **Subcomponent**, it may be necessary to specify the **Package** which contain the referenced **Component**. In **Stood**, this will be generally specified by using a dot notation. However, the internal naming rules of the tool only allow a simple name for the remote **Package**. In order to get the appropriate full name for the **Package** in the **AADL** code, a **pragma** *renames* can be used. This **pragma** must be set for each **Environment Object** that needs it, and takes as parameter the actual name to be used in the **AADL** code: **pragma** *renames (full_name => A.B.C.P)*

example*(taken from SAE AS5506, section 5.1)*:



Data Component `Personnel_Record` has a **Data SubComponent** `Home_address` which **Data Type** is provided by the **Package** relief. However, `relief` is in effect the sub **Package** `sei::aadl::relief`. A **pragma** *renames (full_name => sei::aadl::relief)* has thus been set for the local **Environment Module** `relief`, as shown below:



The **AADL** generated code thus becomes:

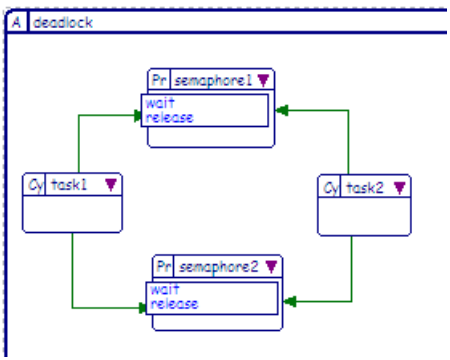
```
DATA Personnel_record
FEATURES
  update_address : SUBPROGRAM update_address;
END Personnel_record;

DATA IMPLEMENTATION Personnel_record.others
SUBCOMPONENTS
  Name : DATA basic::string;
  Home_address : DATA sei::aadl::relief::Address;
END Personnel_record.others;
```

2.4.6. Pragma behavior

One of the current lacks in the **AADL** specification is the ability to describe finely the actual behavior of a **Thread** or a **Subprogram**, in order to be able to perform various kinds of verifications. However, it is possible to extend the language thanks to **Property Sets** and **Annexes**. As a result of the French RNTL project **COTRE**, a behavioral **Annex** for the **AADL** has been defined. Although there is still some ongoing work on that subject, the current version of this **Annex** has been included into **Stood**.

To activate the generation of this **Annex**, the **pragma** *behavior (annex => COTRE)* must be set for the **Root Module**. The effect will be to copy the contents of the **OBCS** and **OPCS** *pseudo code sections* into the corresponding **Component Implementation**.

example*(taken from COTRE)*:



If the appropriate *pseudo code sections* have been properly completed, the following code will be generated for the **Component** `task1`, if the **pragma** *behavior (annex => COTRE)* has been set:

```
THREAD task
FEATURES
  semaphore1 : REQUIRES DATA ACCESS semaphore;
  semaphore2 : REQUIRES DATA ACCESS semaphore;
END task;

THREAD IMPLEMENTATION task.T1
PROPERTIES
  Dispatch_Protocol => periodic;
  Period => 13;
  Compute_Entrypoint => thread;
ANNEX COTRE {**
  STATES
    s0, s1, s2, s3, s4, s5, s6, s7, s8 : STATE;
    s0 : INITIAL STATE;

  TRANSITIONS
    s0 -[ ]-> s1 { PERIODIC_WAIT };
    s1 -[ ]-> s2 { COMPUTATION(1.9ms, 1.9ms) };
    s2 -[ semaphore1.wait ! (-1.0ms) ]-> s3;
    s3 -[ ]-> s4 { COMPUTATION(0.1ms, 0.1ms) };
    s4 -[ semaphore2.wait ! (-1.0ms) ]-> s5;
    s5 -[ ]-> s6 { COMPUTATION(2.5ms, 2.5ms) };
    s6 -[ semaphore2.release ! ]-> s7;
    s7 -[ ]-> s8 { COMPUTATION(1.5ms, 1.5ms) };
    s8 -[ semaphore1.release !]-> s0;
**};
END task.T1;
```

2.4.7. Pragma reverse

This **pragma** is currently useful only in collaboration with the **pragma** *behavior*. It offers the ability to perform round trip engineering between the **AADL** source code and the **Stood** design model for the behavioral code located into the corresponding **Annex**. The **pragma** *reverse* has no parameter and must be set on the **Root Module** only.

example:

If the two **pragmas** *behavior (annex=>COTRE)* and *reverse* have been set, then the following code will be generated for an empty **Thread Implementation**:

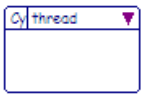
```
THREAD IMPLEMENTATION new_thread.others
ANNEX COTRE {**
  -- <begin pseudo::obcs new_thread>
  -- <end>
  -- <begin pseudo::OpDef new_thread thread>
  -- <end>
**};
END new_thread.others;
```

Annex specific code may then be added between the `begin` and `end` tags and will be fed back to the **Stood** model, when pressing the round trip button:

2.4.8. Pragma type_name

By default, an **AADL Component Type** takes the same name as the corresponding **Module** in **Stood**. It is however possible to change this name by specifying it in a **pragma** *type_name (name => reused_type_name)*. This **pragma** must be set for each **Module** that needs it.

example:



If the **pragma** *type_name (name => task)* has been set for the **Module** `thread`, then the following code will be generated:

```
THREAD task
PROPERTIES
  Dispatch_Protocol => periodic;
  Compute_Entrypoint => thread;
END task;
```

This **pragma** may be especially useful in collaboration with the **pragma** *implementation_name*, to declare several **Implementations** for the same **Type**.

2.4.9. Pragma implementation_name

Default name for an **AADL Component Implementation** is `others`. It is however possible to change this name by using a **pragma** i*mplementation_name (name => other_name)*. This **pragma** must be set for each **Module** that needs it. This **pragma** may be especially useful in collaboration with the **pragma** *type_name*, to declare several **Implementations** for the same **Type**.

example:

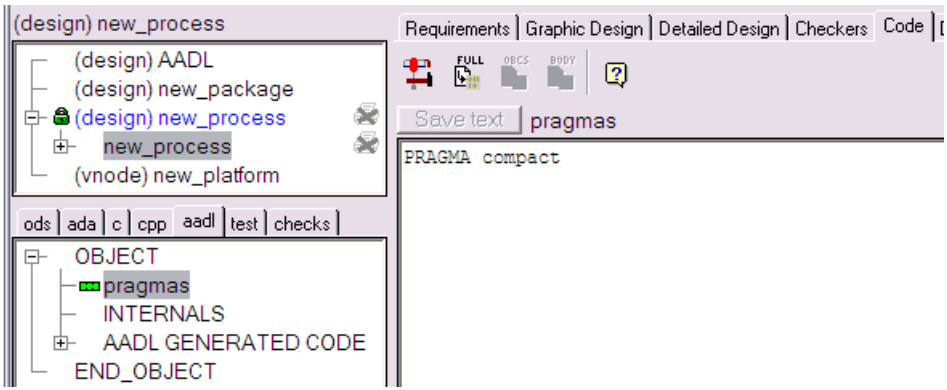
If the following **pragmas** have been set:
- For **Module** `new_process`:
  pragma *implementation_name (name => demo)*
- For **Module** `task1`:
  pragma *type_name (name => task)*
  pragma *implementation_name (name => T1)*
- For **Module** `task2`:
  pragma *type_name (name => task)*
  pragma *implementation_name (name => T2)*

Then the following code will be generated:

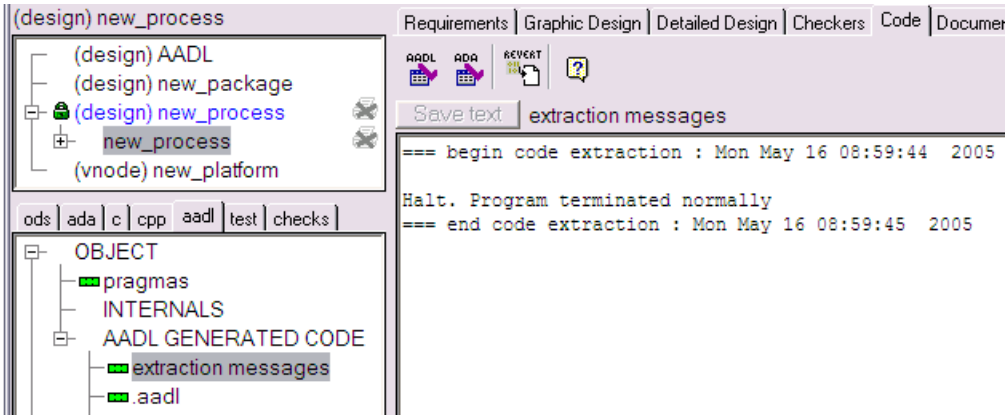
```
PROCESS IMPLEMENTATION new_process.demo
SUBCOMPONENTS
  task1 : THREAD task.T1;
  task2 : THREAD task.T2;
END new_process.demo;
```

2.5. Generate and view AADL code

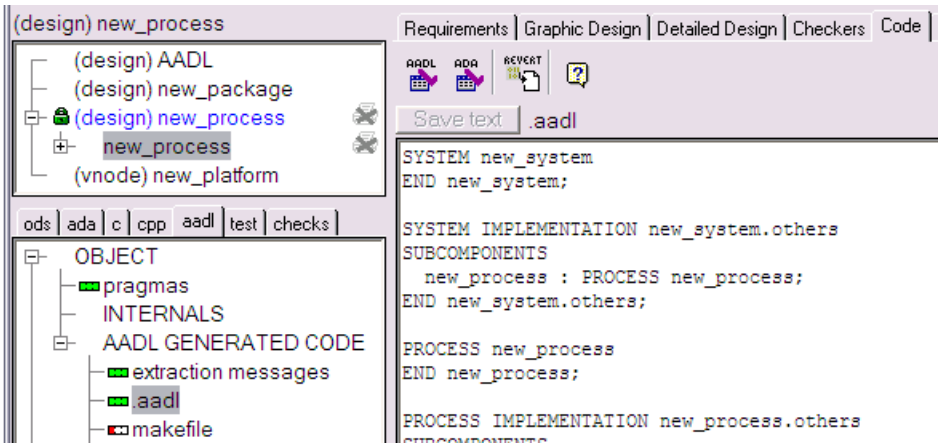
To start the generation of the **AADL** code, select the *Tools/Code/Full extraction* menu or simply press the  button, then select *OK* in the dialog box.



When completed, the result of the code generation process is shown as follow:



For each **Module** of the **Design**, select the *.aadl* section to edit the corresponding generated **AADL** code. If the **pragma** *compact* has been set, then the whole **AADL** specification will be located inside the **Root Module**.

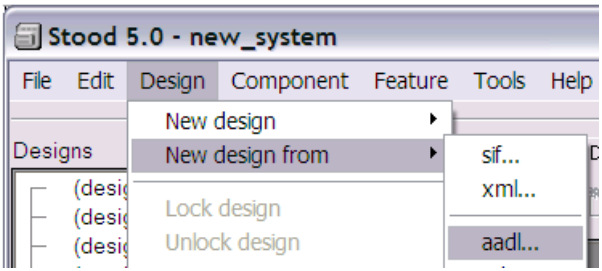


It is possible to make changes in theses source files. Changes can be saved with the Save text button to the **AADL** syntax can be checked with the button. If the **pragma** *reverse* has been set, the changes done between the round trip engineering tags will be fed back to the **Design** model by pressing the button.

The **AADL** source file can also be edited directly from the **Design** database. Default location is the `_aadl` subdirectory in the **Design** directory. To open it from **Stood**, use the *Tools/Open directory/Design directory* menu.

3. AADL import

To import an existing **AADL** specification into a new **HOOD Design** or to update an existing one, open an existing **System** or create a new one, then use the *Design/New design from/aadl* or *Design/Update design from/aadl* menu.



It is then necessary to select a `.aadl` file from the dialog box and press the *Open* button.

Important Notes:

- All the **AADL** source files to be parsed to create or update the **Design** must be located within the same directory.
- Avoid any code dupplication within the parsed directory.
- The selected file name must be the name of the root **AADL Package** or **Component** to be parsed. It must include the **Component Type** name and the **Component Implementation** name, separated by an underscore character.
- To get proper results, all **Subcomponents** must refer to an existing **Component Type** and **Implementation**.
- It may be necessary to perform the import in several steps, one for each root **Package** or **Component**. In that case, it may be necessary to rename the corresponding **AADL** source file if it contains several roots.

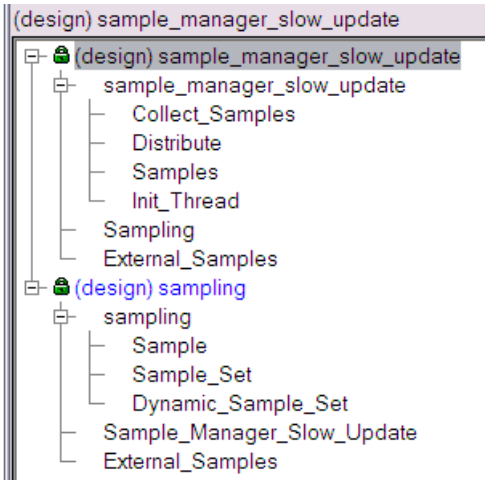
Example:

Let's consider the following **AADL** source file (taken from SAE AS5506, section 4.5). It is composed of two parts: the **Package** `Sampling` and its contents, as well as the **Process Implementation** `Sample_Manager.Slow_Update` and its contents.

```
package Sampling
public
data Sample
  properties
    Source_Data_Size             => 16 B;
end Sample;

data Sample_Set
features
  read : subprogram;
  write : subprogram;
properties
    Source_Data_Size             => 1 Mb;
end Sample_Set;

data Dynamic_Sample_Set extends Sample_Set
end Dynamic_Sample_Set;
end Sampling;
-----------------------------------------------------

thread Init_Samples
features
  OrigSet : requires data access Sampling::Sample_Set ;
  SampleSet : requires data access Sampling::Sample_Set ;
end Init_Samples ;

thread Collect_Samples
features
  Input_Sample : in event data port Sampling::Sample;
  SampleSet : requires data access Sampling::Sample_Set ;
end Collect_Samples ;
```

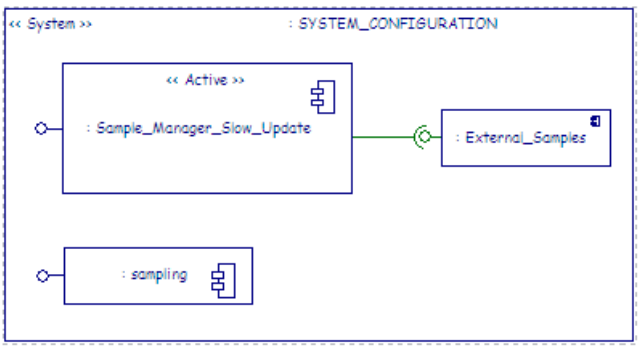
```
thread implementation Collect_Samples.Batch_Update
refines type
  Input_Sample: refined to in event data port
      Sampling::Sample_Set;
end Collect_Samples.Batch_Update;

thread Distribute_Samples
features
  SampleSet : requires data access Sampling::Sample_Set ;
  UpdatedSamples : out event data port Sampling::Sample;
end Distribute_Samples ;

process Sample_Manager
features
  Input_Sample: in event data port Sampling::Sample;
  External_Samples: requires data access Sampling::Sample_Set;
  Result_Sample: out event data port Sampling::Sample;
end Sample_Manager ;

process implementation Sample_Manager.Slow_Update
subcomponents
  Samples: data Sampling::Sample_Set;
  Init_Thread : thread Init_Samples;
  -- the required access is resolved to a subcomponent decl.
  Collect_Samples: thread Collect_Samples.Batch_Update;
  Distribute: thread Distribute_Samples ;

connections
  data access Samples -> Init_Thread.SampleSet;
  data access External_Samples -> Init_Thread.OrigSet;
  data access Samples -> Collect_Samples.SampleSet;
  event data port Input_Sample -> Collect_Samples.Input_Sample;
  data access Samples -> Distribute.SampleSet;
  event data port Distribute.UpdatedSamples -> Result_Sample;
end Sample_Manager.Slow_Update ;
```

To properly import this **AADL** code into **Stood**, it is necessary to create two separate directories, each of them containing a copy of the original file. Within the first directory, the source file must be renamed into `sampling.aadl`, and within the second directory, the same file must be renamed into `sample_manager_slow_update.aadl`. The import process will be done in two steps, one to import the **Package**, and one to import the instanciated **Process**. Note that the two files could have been let in the same directory if the had contained no dupplicated code.

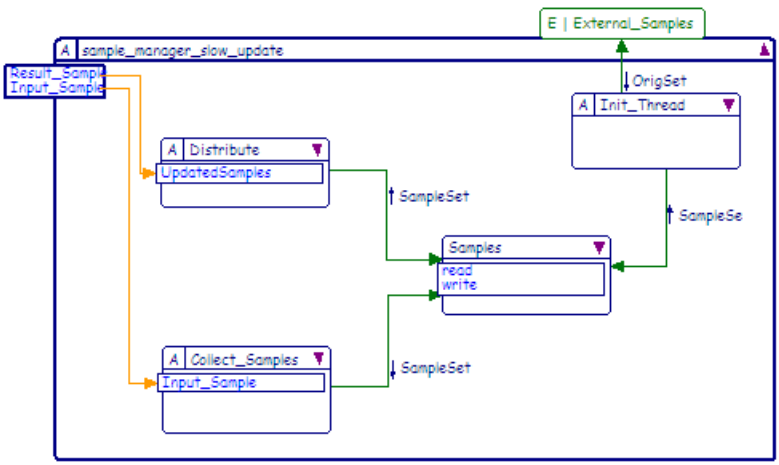


After some simple changes in the diagrams layout to improve the readability, the result is as follow. The diagrams below just show the architecture. Detailed design information is also imported into the database, such as some **Properties** and comments. On the contrary some **AADL** constructs are not supported by the import function yet. Each diagram can be displayed either with the **HOOD** or the **UML 2.0** graphical notation.

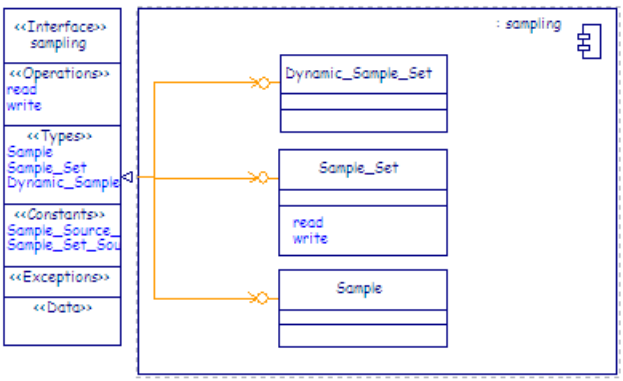
The high level view of the architecture shows an **Active Component** that represents the **Process** instance, a **Passive Component** that represents the **Package** of **Data Types** (note that a better representation would have been an **UML Package** there), and an external shared **Component** instance that is required by the **Process**.



The diagram showing the contents of the **Process** has been provided here with the **HOOD** notation that offers more details. It is easy to recognize the three **Thread Subcomponents** and the local shared **Data** instance. **Data Ports** are mapped to **Operations** like for the **AADL** export function.



Finally, the contents of the **Package** shows three **Classes** that are displayed with the **UML** notation which is more appropriate here. Note that the inheritance link between the two **Classes** `Dynamic_Sample_Set` and `Sample_Set` is not shown because the it is currently managed at a **Component Implementation Extension** level only. `Source_Data_Size` **Properties** are represented by **Constants** in the **Stood Design** model.



After this import operation, the software design process may continue by adding detailed design information, such as design documentation items and target language code sections. It becomes then possible to use the existing verification and automatic code and documentation generation tools, to complete the full **AADL** to code industrial process that is supported by **Stood**.

www.tni-world.com
stood@tni-world.com

TNI Europe
Triad House
Mountbatten Court
Worall Street
Congleton
Cheshire
CW12 1AG
UK

+44 1260 291 449

Ellidiss Technologies
Technopôle Brest-Iroise
115 rue Claude Chappe
29280 Plouzané
Brittany
France

+33 298 451 870