Stood 5.4

AADL Tutorial

Pierre Dissaux
Pam Flood
Ellidiss

Contents

1 Introduction

The purpose of this document is to describe the standard modelling process to be used to build a new **AADL** project with **Stood**. It is not a complete tutorial or user manual for **Stood** and knowledge about **AADL** is a prerequisite for using this document. Please refer to the official **AADL** web site (`www.aadl.info)` to learn more about this standard.

In addition to the various modelling concepts defined by the **AADL** standard, we need to introduce the following additional ones that are more development process oriented:

A **Stood Design** is the main modelling entry point. It represents the root of a hierarchy of components or packages. A **Design** may be used to define the overall system, a particular set of concrete components describing an executable software application, or a library of abstract components. In the first case, the **Design** is directly associated to an **AADL** system instance. In the second one, it will represent an **AADL** process instance, whereas in the last case, it will represent an **AADL** package. A fourth case, still subject to deeper investigations, concerns the software to hardware binding activity, for which the current entry point is an **AADL** processor.

A **Stood Project** refers to a set of **Designs** that collaborate in a common realization. The **Project** specifies the scope of the realization, and restricts the access to other non-referenced **Designs**.

A **Stood Session** begins when a user launches the tool and ends when **Stood** is closed. **Stood** is a multi-user environment, so several **Sessions** may be opened on the same **Project**, or even on the same **Design**.

A **Stood Workspace** is a user-defined disk area where **Projects** and **Designs** can be stored.

This tutorial includes the following sections:

1. Define a **Workspace**, launch **Stood** and create a **Project**
2. Create a **Design** representing an **AADL** system
3. Create a **Design** representing an **AADL** package
4. Create a **Design** representing an **AADL** process

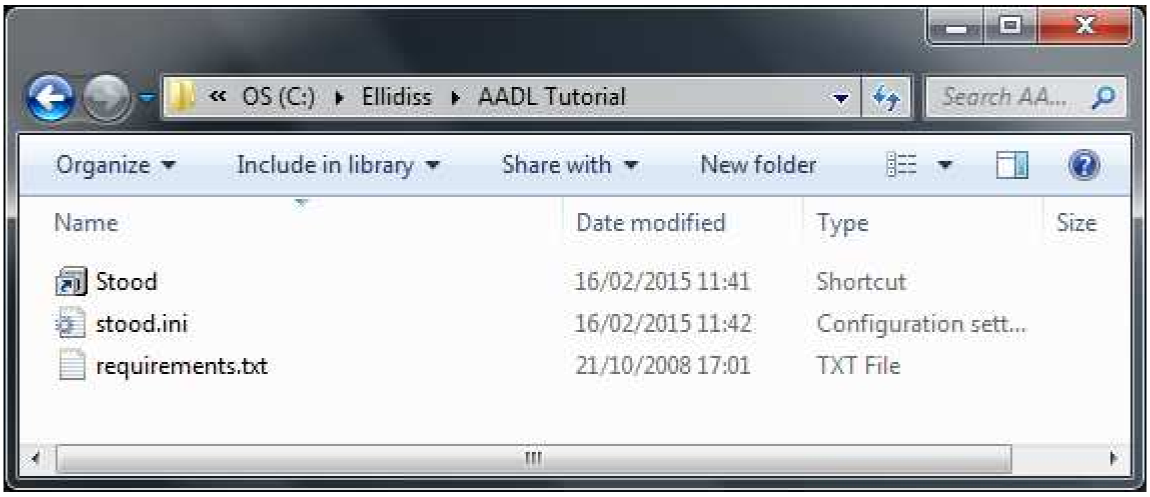
2 Define a workspace

It is possible to launch **Stood** from the start-up menu or from the desktop icon that is created by the **Windows** installer program. However, any customization of the tool requires altering the original distribution, which may be an issue if several users or several **Projects** require different customizations.

This is the reason why it is recommended that you launch **Stood** from a **Workspace** that can be dedicated to each user or **Project**. Such a **Workspace** consists of a simple directory that must contain at least two elements:

- a shortcut to the **Stood** executable file (under **Windows**) or a redirecting shell script (under **Unix**)
- a local initialization file (`stood.ini` under **Windows** or `.stoodrc` under **Unix**)

In addition, this **Workspace** may contain other **Project** related data such as lists of textual requirements or subdirectories with legacy source code to be reused in the **Project**.

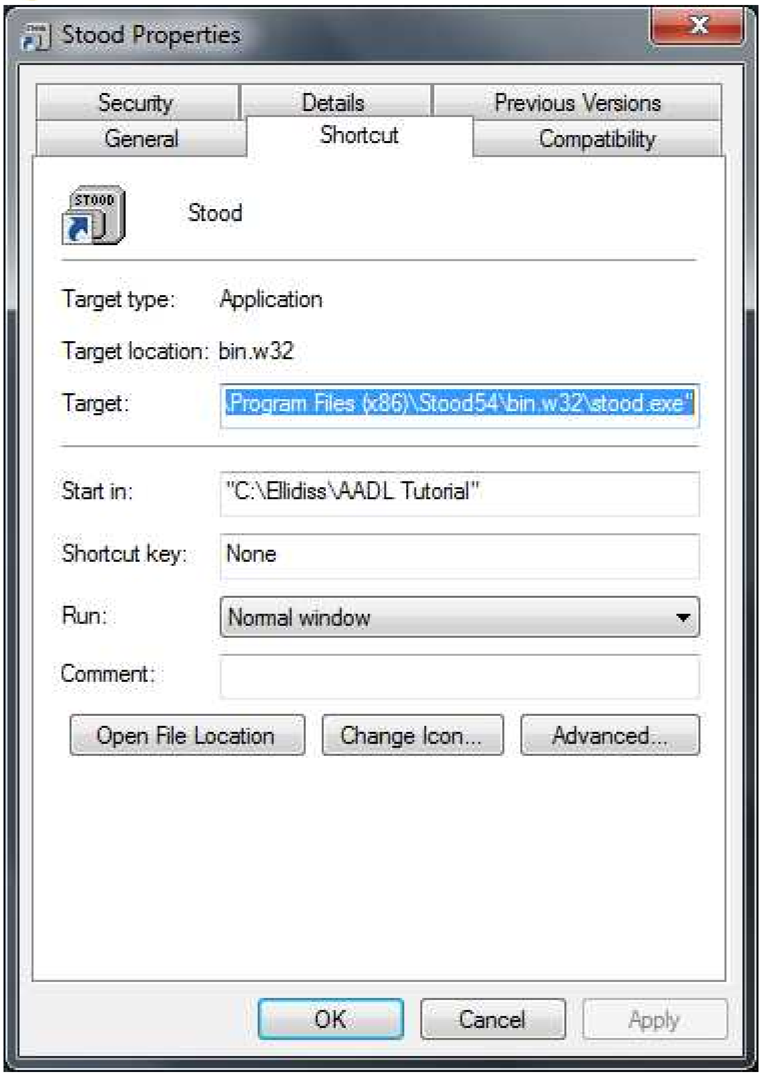


Name	Date modified	Type	Size
Stood	16/02/2015 11:41	Shortcut	
stood.ini	16/02/2015 11:42	Configuration sett...	
requirements.txt	21/10/2008 17:01	TXT File	

2.1 Stood shortcut

A **Stood** shortcut must be created by one of the standard **Windows** procedures. However, it is necessary to customize it so that the **Workspace** can be used as the default storage area for the newly created **Projects** when **Stood** is launched from there.

Open the *properties* box of the shortcut, select the *shortcut* tab, and check that the *target* field points to the correct installation file and that the *start in* field points to the current **Workspace** directory instead of the installation one, as shown below:


Stood Properties

Security | Details | Previous Versions
General | Shortcut | Compatibility

Stood

Target type: Application

Target location: bin.w32

Target: Program Files (x86)\Stood54\bin.w32\stood.exe"

Start in: "C:\Ellidiss\AADL Tutorial"

Shortcut key: None

Run: Normal window

Comment:

2.2 *Stood initialization file*

The second customization that may be required consists of modifying one or more of the properties that are specified in the **Stood** initialization file (`stood.ini` under **Windows** or `.stoodrc` under **Unix**). A complete specification of these properties is provided in the *Stood Administrator Manual*.

Only the properties that differ from the default initialization file located in the installation directory need to be specified locally. All the other properties will be automatically inherited. Typical properties that may be customized locally are the various environment variables that are used by **Stood** to interact with external tools, such as compilers or verification tools.

```
[Environment]
ADA_PATH=C:\cygwin\bin
C_PATH=C:\cygwin\bin
CPP_PATH=C:\cygwin\bin
REQTIFY_PATH=C:\reqtify\bin.w32
OSATE_PATH=/cygdrive/C/osate2/osate.exe
```

2.3 *Requirements file*

The **Workspace** directory may also contain one or more **Project** specific textual requirements files that may be imported into **Stood**. Once imported, it is possible to specify requirements coverage for each modelling entity.

The internal format for **Stood** compliant requirements files is quite simple as it consists of plain **ASCII** text declaring one requirement per line. Each requirement is expressed by a unique identifier and a free comment, separated by a `tab` character.

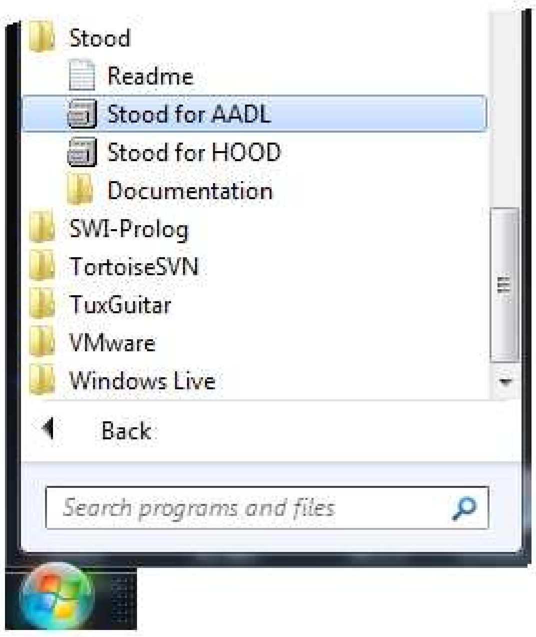
```
CALC101     interact with the Keyboard
CALC102     interact with the Screen
CALC111     define integer type
CALC112     define real type
CALC121     add integers
CALC122     add reals
CALC123     sub integers
CALC124     sub reals
CALC131     scan the Keyboard
CALC132     perform the operation
CALC133     display on Screen
```

Such a requirements file may be easily generated by a requirements management tool such as **Doors**™. Note that thanks to a specific connection feature, it is also possible to directly import requirements and export coverage information from and to a traceability graph defined in the **Reqtify**™ tool.

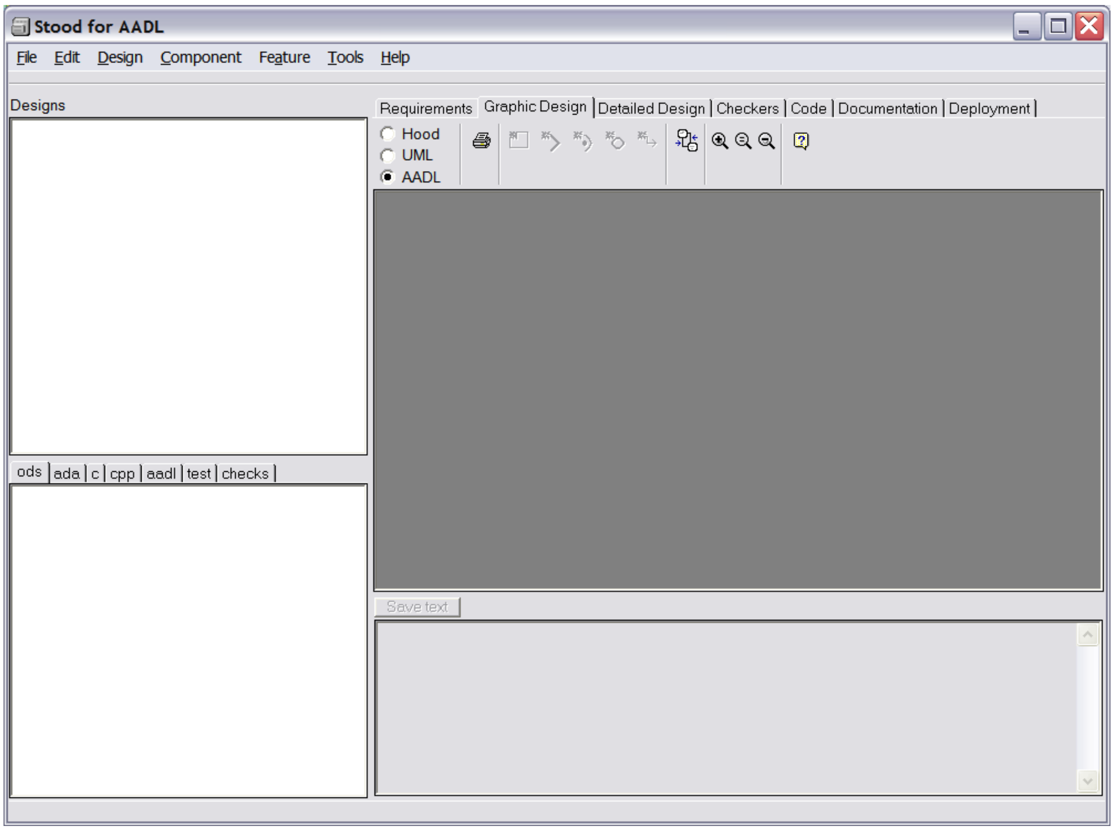
2.4 *Launching Stood*

If not launched from the **Workspace** shortcut, **Stood** can be launched either from the desktop shortcut or from the standard **Windows** start up menu.

Note that these two options are not available if the corresponding set up has been deactivated during the installation process.



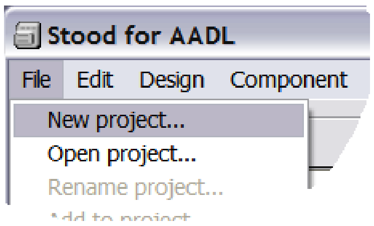
When launched, **Stood** shows a start up screen during its initialization phase, and then opens its main windows, as shown is the diagram below:

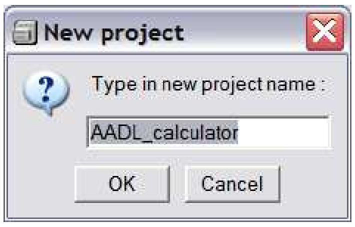

Stood for AADL
File Edit Design Component Feature Tools Help
Designs
Requirements Graphic Design Detailed Design Checkers Code Documentation Deployment
Hood
UML
AADL
ods ada c cpp aadl test checks
Save text

2.5 *Create a Stood project*

First step of the modelling process consists of either opening an existing **Project** or creating a new one. **Projects** can be managed from the *File* menu.

For the purpose of this tutorial, we create a new **Project** and specify its name in the dialog box.





We can now import existing **Designs** within the scope of our **Project** using the *File/Add to project* menu, create new **Designs** from existing source code using the *Design/New design from…* menus or create new local **Designs** using the *Design/New design* menu.

This last choice offers several options. The new created **Design** may be profiled as a **HOOD Design**, in which case one of the options *design*, *generic* or *virtual node* must be selected, or as an **AADL Design**, in which case one of the options *aadl package*, *aadl system*, *aadl process* or *aadl processor* must be selected.

This tutorial explains how to create in the context of a consistent **Project**:

- **AADL Systems** to define concrete system wide architectures composed of hardware and software components.
- **AADL Packages** to specify libraries of abstract components, and especially abstract **Data** components to be used as classifiers for ports and parameters.
- **AADL Processes** to perform complete software design activities including architectural design, detailed design and coding, model analysis, source code and design documentation generation.

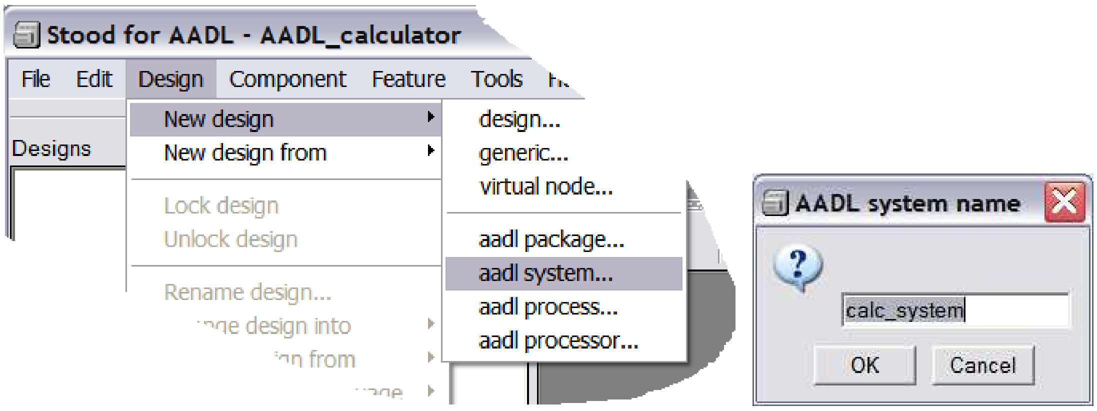
These three modelling processes are described in the next three sections. The last section explains how to quit **Stood**.

3 Create an AADL system

Specifying a **System** with **Stood** consists of building a **System Instance**. Instead of creating abstract component types, component implementations and then instantiating them as subcomponents, the designer can directly define subcomponents in a hierarchical way, and then specify whether they correspond to instances of already defined abstract components or of anonymous abstract components that will have to be automatically created while producing the textual **AADL** specification.

3.1 *Create a new design of type "aadl system"*

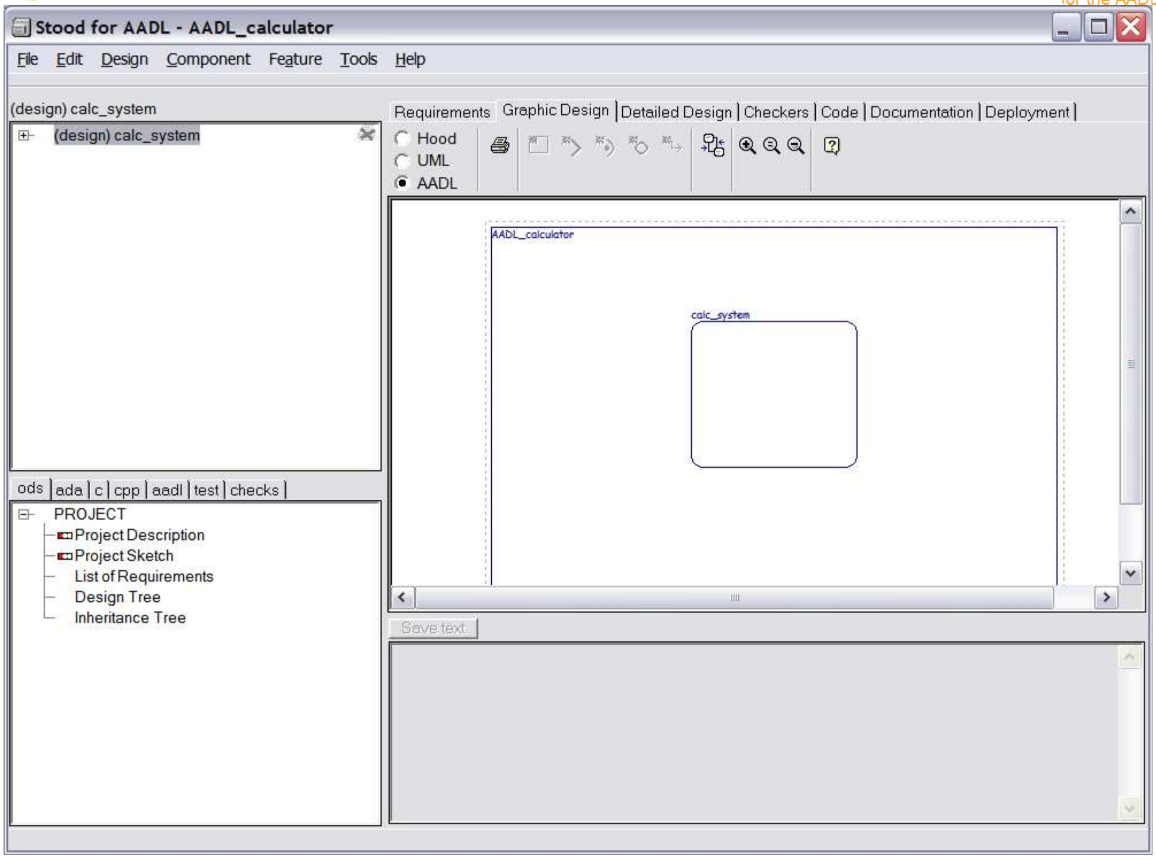
To create a new **System** inside the current **Project** (cf.§2.5), use the menu *Design/New design/aadl system…* and then specify its name in the dialog box.



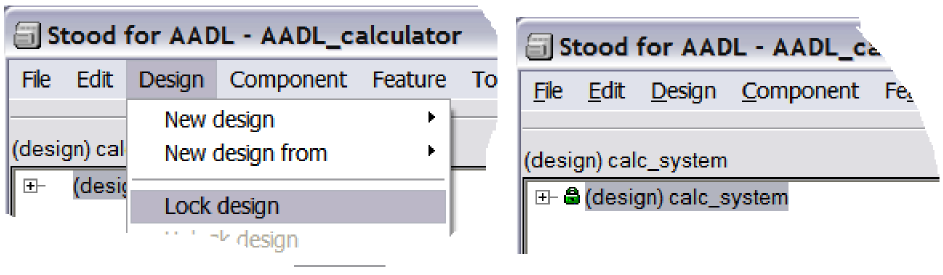
Note that a **Design** name in **Stood** must be alphanumeric (i.e. only contain characters '`a`' to '`z`', '`A`' to '`Z`', '`0`' to '`9`' or the underscore character '`_`').

3.2 *Lock the system to enter edit mode*

When it has just been created, the new **System** design is automatically loaded and it is shown in the **AADL** graphical editor as an empty box at the middle of a larger one representing the **Project**.


Stood for AADL - AADL_calculator
File Edit Design Component Feature Tools Help
(design) calc_system
Requirements Graphic Design Detailed Design Checkers Code Documentation Deployment
Hood
UML
AADL
AADL_calculator
calc_system
ods ada c cpp aadl test checks
PROJECT
Project Description
Project Sketch
List of Requirements
Design Tree
Inheritance Tree
Save text

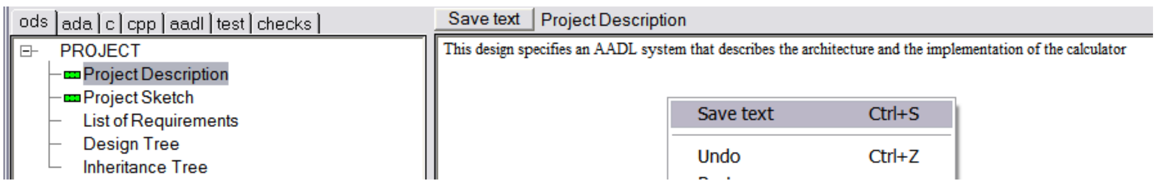
However, this **System** design is set to read-only mode by default. To enable modification of this **System**, it is necessary to “lock” it so that no other user is allowed to get a concurrent write access to the model.


Stood for AADL - AADL_calculator
File Edit Design Component Feature To
New design
New design from
Lock design
(design) calc_system

When a **System** design is locked (may be modified in the current **Session**), a green padlock is shown to the left of its name.

3.3 *Document the project*

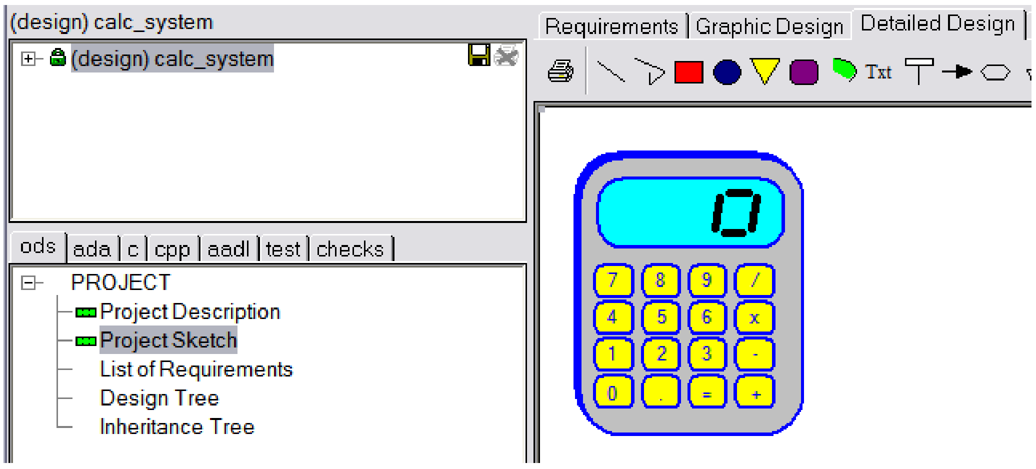
When the **System** design is selected in the upper left list of **Stood** window, the lower left list shows description sections for the local view of the **Project**. Default sections are textual descriptions and a graphical sketch that will be included inside the design documentation.



This design specifies an AADL system that describes the architecture and the implementation of the calculator

Note that each time some text is entered in the text input area, it must be saved by pressing the

Save text button, or using the corresponding contextual menu or the *Ctrl-S* keyboard shortcut.



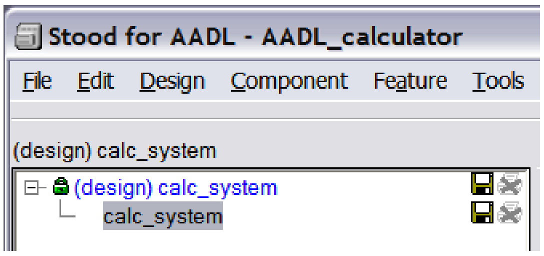
(Note that shown items may vary depending on the tool configuration)

Note that sketches are provided for documentation purpose only and that they do not carry any semantic information.

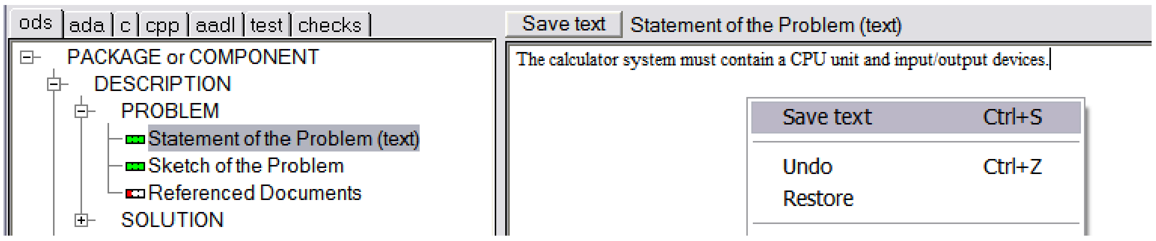
Note also that the other sections *List of Requirements*, *Design Tree* and *Inheritance Tree* are automatically filled in by **Stood**.

3.4 Document the system

When the **System** design tree is deployed in the upper left list, it shows another line with the same name. This corresponds to the **System** component instance to be edited.


Stood for AADL - AADL_calculator
File Edit Design Component Feature Tools
(design) calc_system
(design) calc_system
calc_system

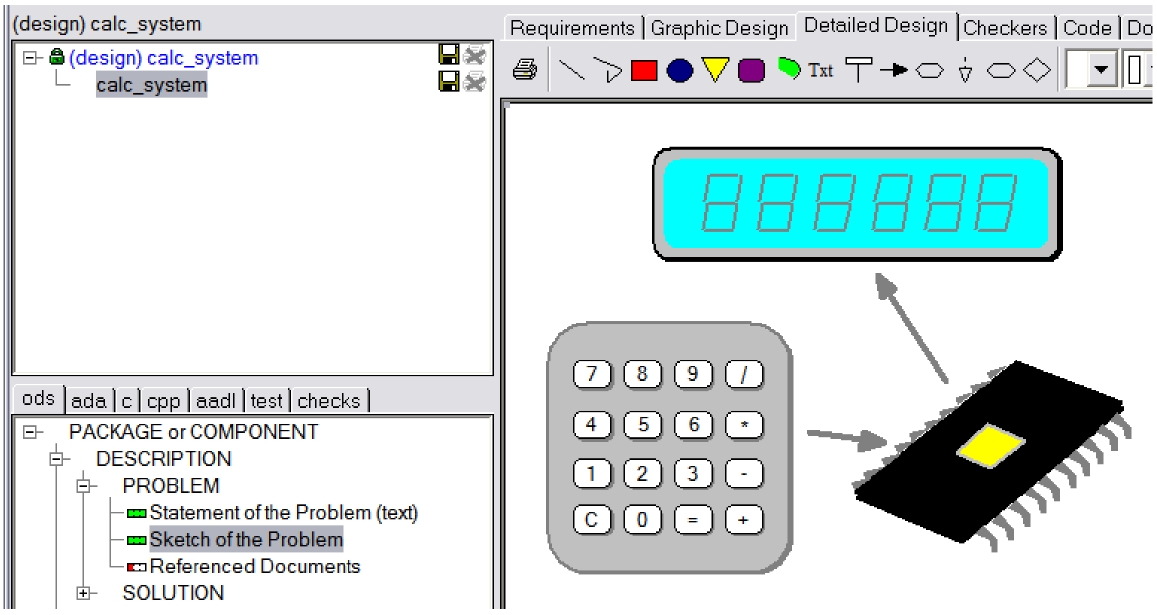
Note that when a component is selected in the upper left list, the current selection of the graphical editor and contents of the lower left list are automatically updated. It is thus possible to complete the textual sections and sketches that are available to describe each component individually.



The calculator system must contain a CPU unit and input/output devices.

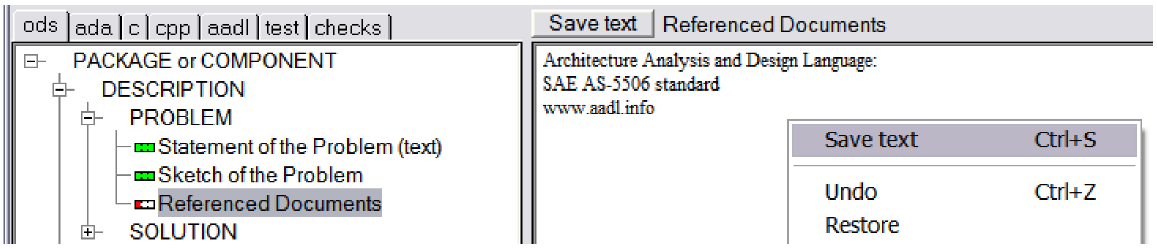
The *Statement of the Problem* section (see figure above) must be used to provide textual details about how the currently selected component contributes to solving a particular modelling problem.

The *Sketch of the Problem* section (see figure below) can be used to complete this information by an informal drawing that is included in the design documentation.


(design) calc_system
(design) calc_system
calc_system
Requirements | Graphic Design | Detailed Design | Checkers | Code | Do
Txt
ods | ada | c | cpp | aadl | test | checks
PACKAGE or COMPONENT
DESCRIPTION
PROBLEM
Statement of the Problem (text)
Sketch of the Problem
Referenced Documents
SOLUTION

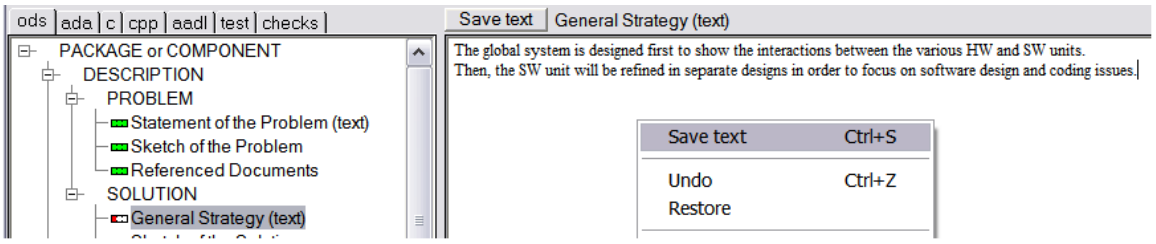
(Note that shown items may vary depending on the tool configuration)

The following figures show other examples of documentation sections that can be filled in for each component.



Architecture Analysis and Design Language:
SAE AS-5506 standard
www.aadl.info

(Note that shown items may vary depending on the tool configuration)

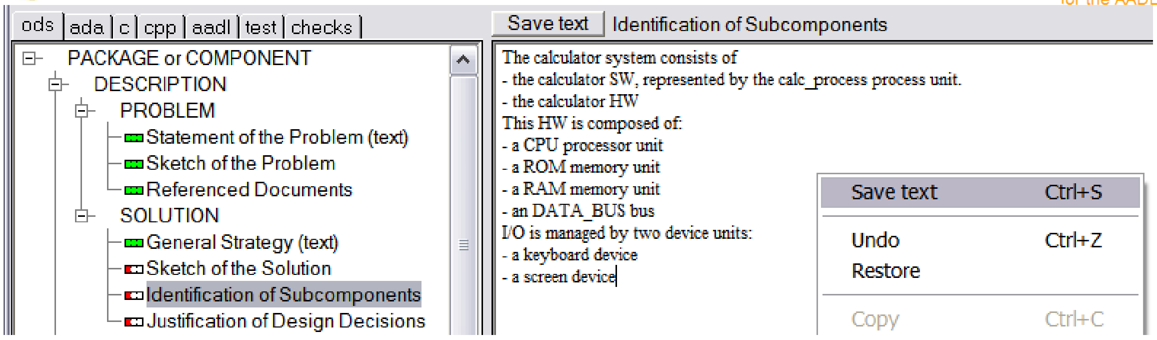


The global system is designed first to show the interactions between the various HW and SW units.
Then, the SW unit will be refined in separate designs in order to focus on software design and coding issues.

(Note that shown items may vary depending on the tool configuration)

Stood promotes the concept of “incremental documentation” that consists of asking the designer to document each modelling element independently at the time they are performing the modelling actions. The final design documentation compiles all these elementary sections to build a complete report.

This modelling process also recommends documenting each component before going deeper into the architecture hierarchy. For instance, the *Identification of Subcomponents* documentation section can be used as a guideline for actually creating the subcomponents (see next chapter).



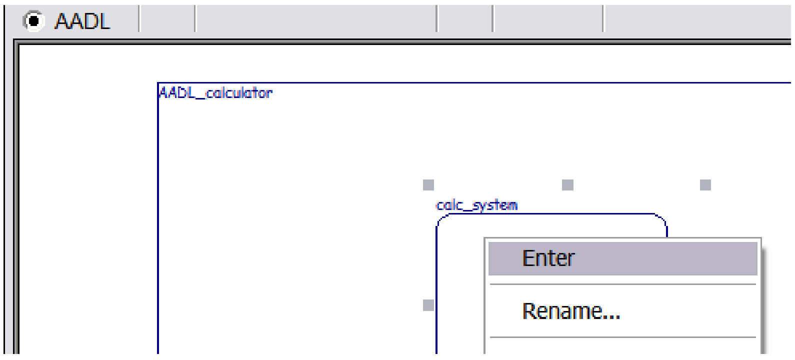
The calculator system consists of
- the calculator SW, represented by the calc_process process unit.
- the calculator HW
This HW is composed of:
- a CPU processor unit
- a ROM memory unit
- a RAM memory unit
- an DATA_BUS bus
I/O is managed by two device units:
- a keyboard device
- a screen device

(Note that shown items may vary depending on the tool configuration)

3.5 Create subcomponents

In the graphical editor, a component can be represented either by its “black box” view which shows the contents of the corresponding component type, or by its “white box” view which shows the contents of the corresponding component implementation.

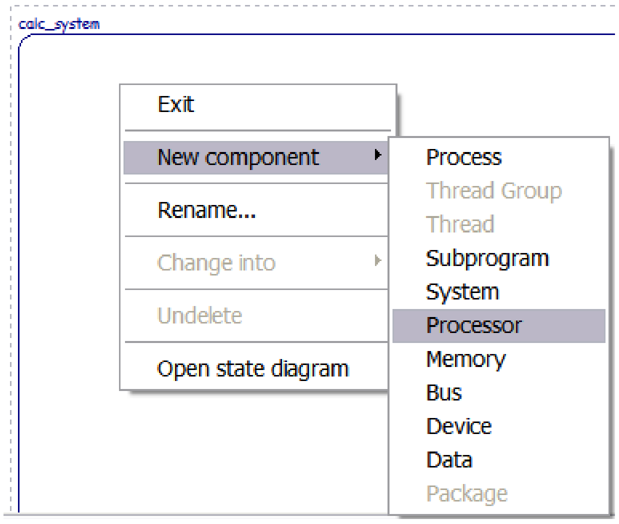
In the picture below, the “black box” view of the **System** component is shown. To show its “white box” view, it is necessary to go one step down the hierarchy. This action can be performed by double-clicking inside the component box or using the *enter* contextual menu.

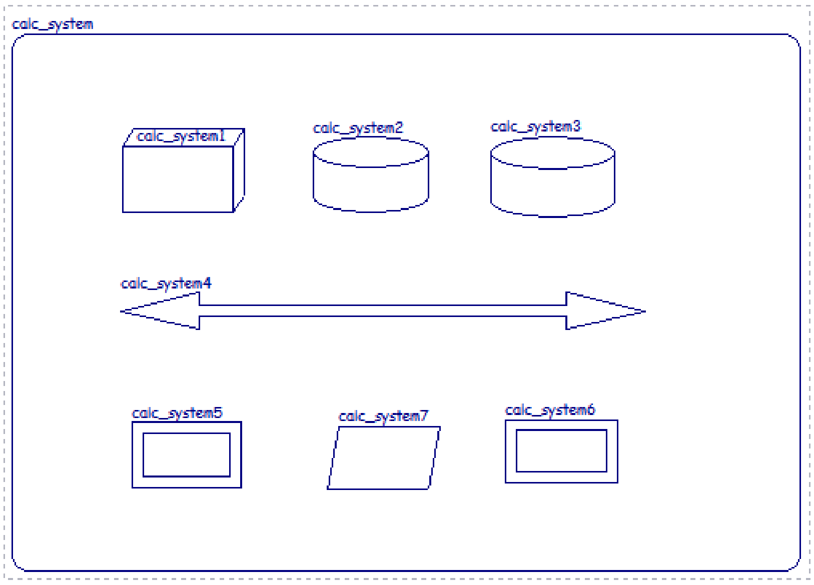


After using the enter menu, the “white box” view of the **System** is shown. It is now possible to add subcomponents to its implementation. To add a new subcomponent, use the *new AADL component* button, or the *New component* contextual menu. The button will create a component of a pre-defined category, which can be modified later using the *Change into* contextual menu. The *New component* contextual menu offers the full choice of valid categories for creating subcomponents within the current component implementation.

We can for instance create seven subcomponents inside the **System** implementation:

- One component whose category is **Processor**
- Two components whose category is **Memory**
- One component whose category is **Bus**
- Two components whose category is **Device**
- One component whose category is **Process**



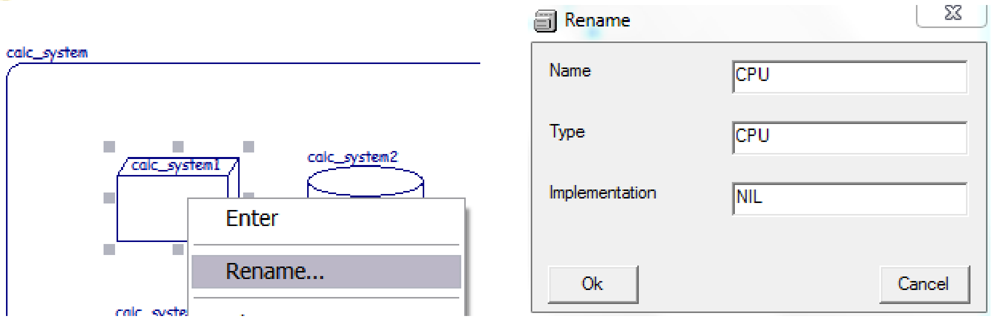


Note that a default name is given to newly created components, and that this name is the name of the concrete subcomponent within the enclosing **System**, which may differ from the corresponding component type classifier and component implementation names.

3.6 Rename and give a type to subcomponents

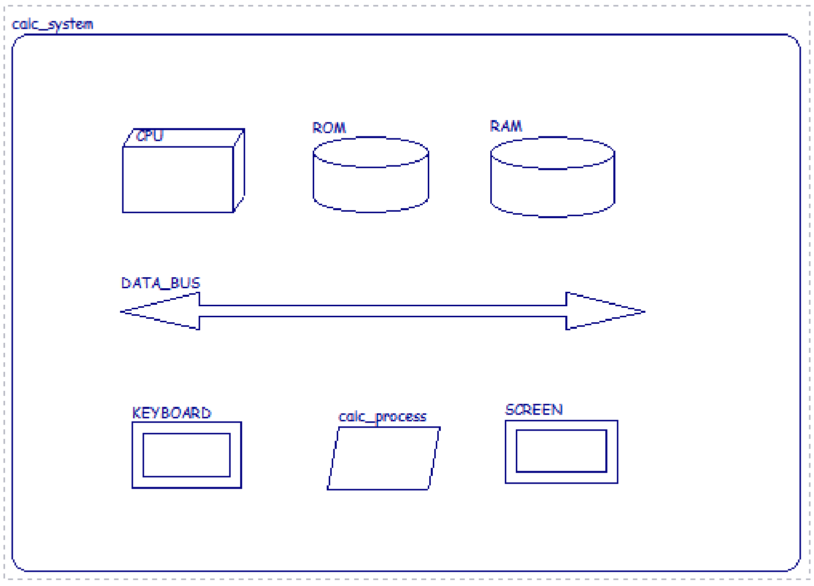
When a component is selected in the graphical editor, the rename contextual menu can be used to perform the following actions:

- Change the name of the selected concrete subcomponent.
- Change the corresponding abstract component type (by default, it is assumed that the subcomponent and component type name are identical)
- Change the corresponding abstract component implementation name (by default, it is assumed that this name is `NIL`). If the implementation name is left to `NIL` and that a component implementation becomes required while generating the **AADL** code, then the default name `others` will be used.



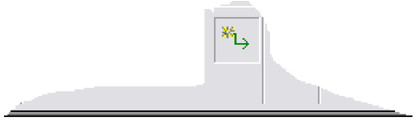
Name	CPU
Type	CPU
Implementation	NIL

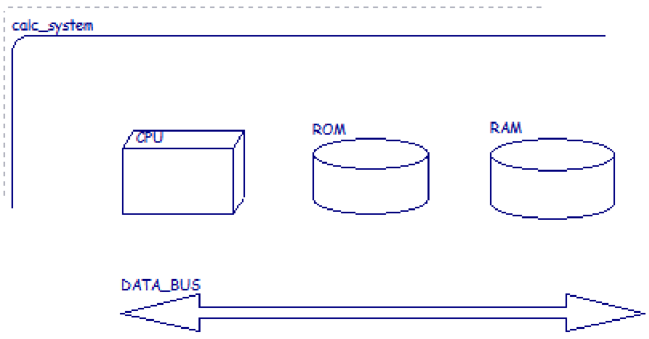
This is used to give the following names to the seven subcomponents of the **System**



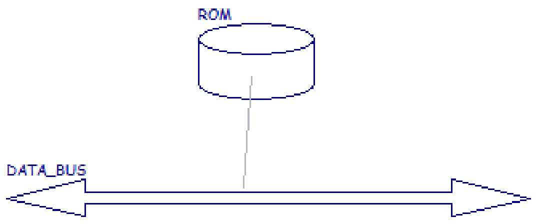
3.7 *Create bus access connections*

To create bus access connections, use the *new connection* button of the graphical editor.

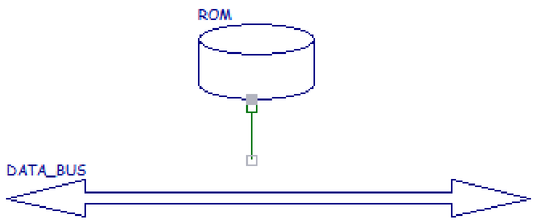




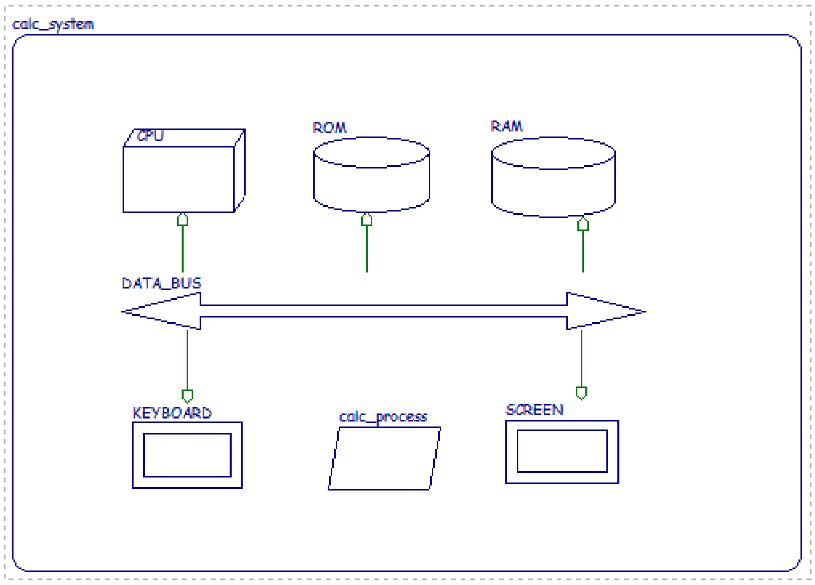
Then click on the accessed bus component before clicking on the accessing component.



This results in a graphical bus access connection between the **Bus** and the **Memory** components.

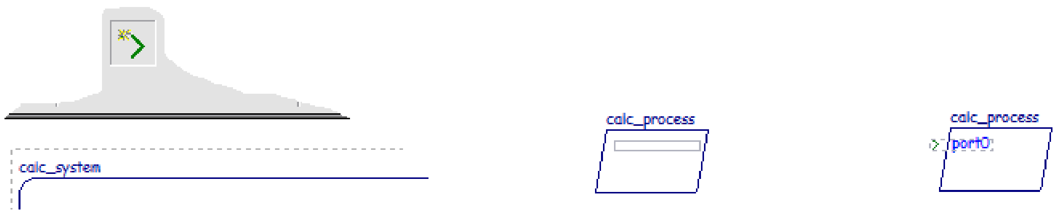


Additional connections may now be added inside our **System** implementation.



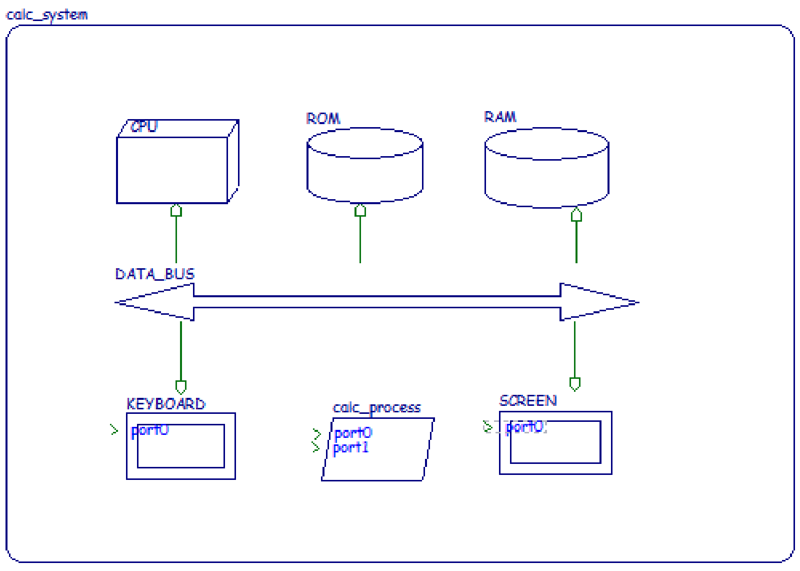
3.8 *Create ports*

To express for instance that the **Process** can exchange data with the **Device** components it is necessary to add ports. To create a port in a component type, use the *new port* button of the graphical editor. Click on the button (a), drag the mouse on the target component (b) and then click to create the port (c).



(a) (b) (c)

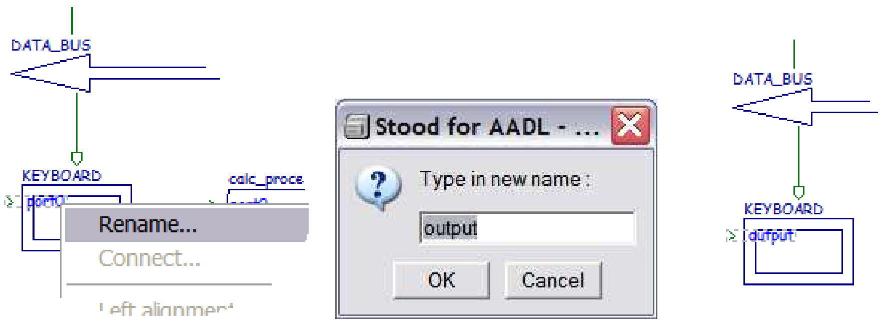
We can now create four ports as shown in the figure below:



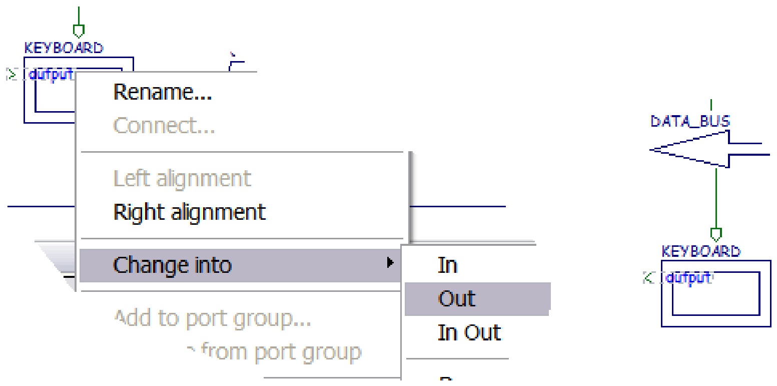
Note that these ports have been created with a default name and as **In Event Ports** by default. The next section explains how to modify the name, the direction and the type of the ports.

3.9 Rename and customize ports

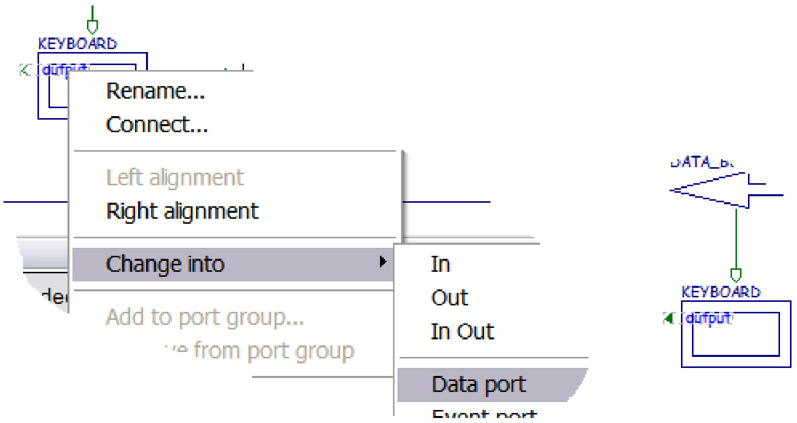
To rename a port, select it and use the *Rename* contextual menu.



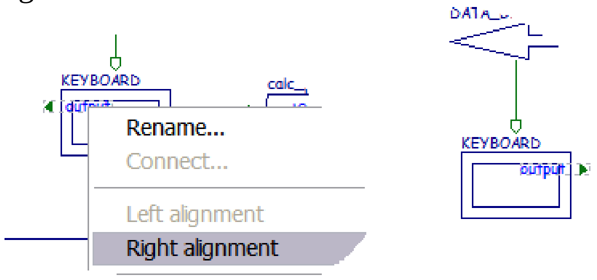
To change the direction of a port, select it and use the *Change into* contextual menu.



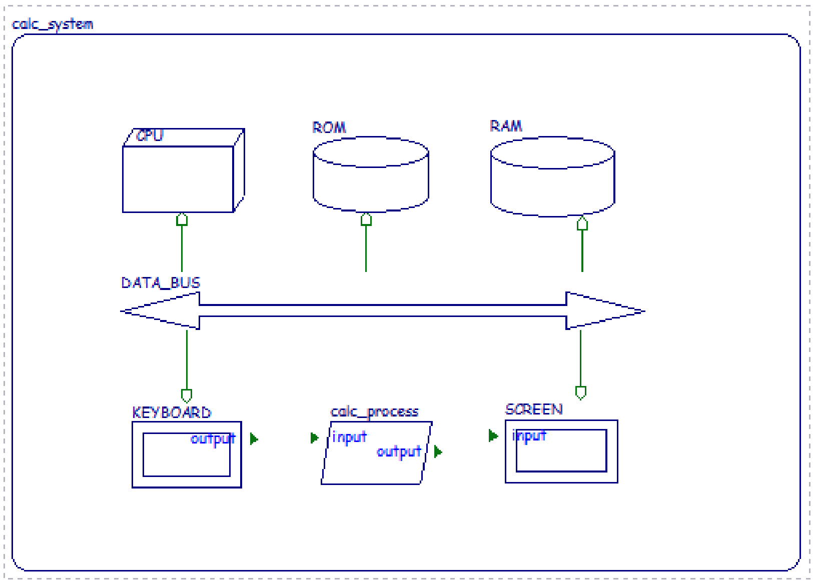
To change the type of a port, select it and use the *Change into* contextual menu.



By default, ports are attached to the left border of the component. It is possible to move them to the right border, using the *Right alignment* contextual menu. In a similar way, a port attached to the right border can be moved back to the left border using the *Left alignment* contextual menu.



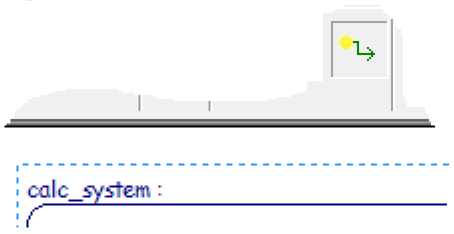
We can now customize the four ports as follow:

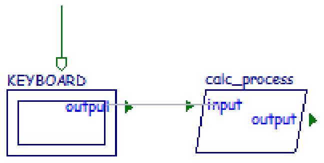


3.10 Create port connections

To create connections between ports, use the new connection button of the graphical editor. First click on the button (a), click one of the connected ports (b) and then the other port (c).

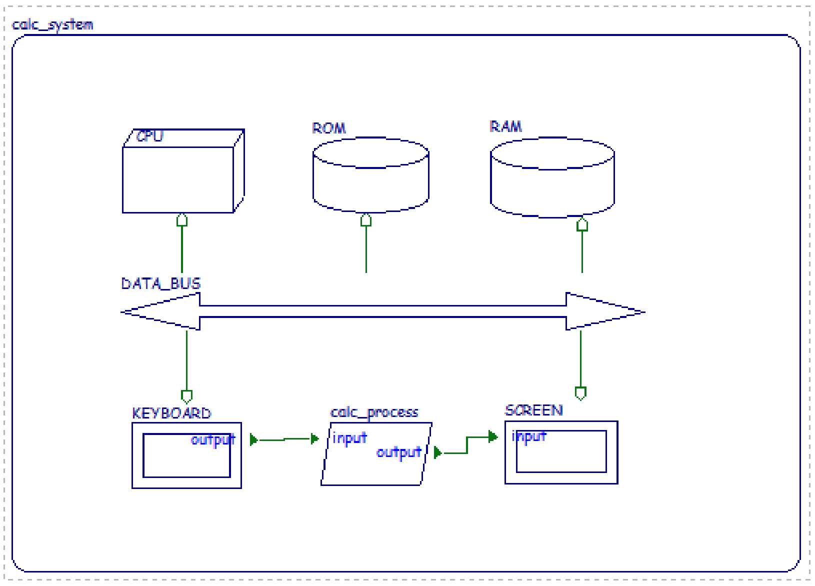
Note that port compatibility is verified before creating the connection. These verifications check, port kind, port direction and port type for **Data Ports** and **Event Data Ports**.





(a) (b) (c)

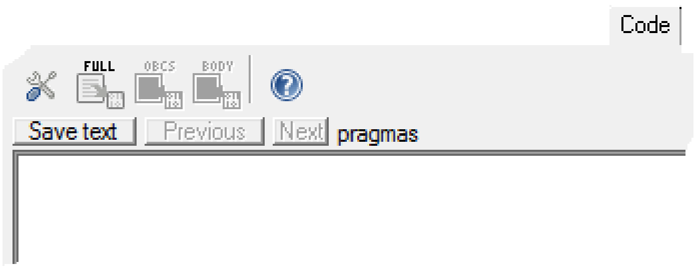
We can create port connections between the **Process** and each of the two **Devices**.



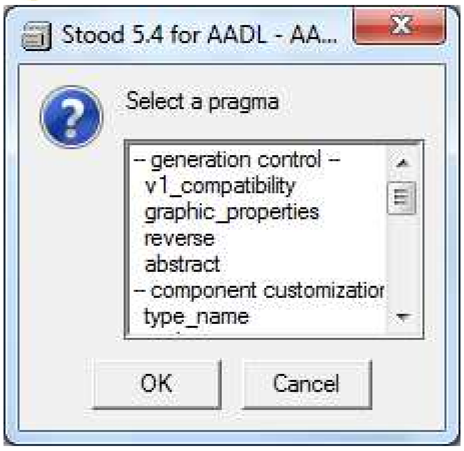
3.11 *Generate the AADL code for the system*

Our model has been created with just a few mouse clicks in the graphical editor. Even so you can still generate a full **AADL** specification from it.

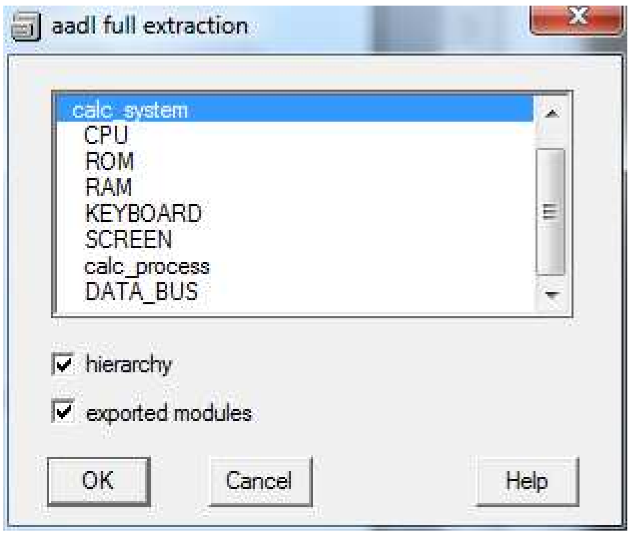
To generate textual **AADL** code from a graphical model in **Stood**, the *Code* tab must be selected.



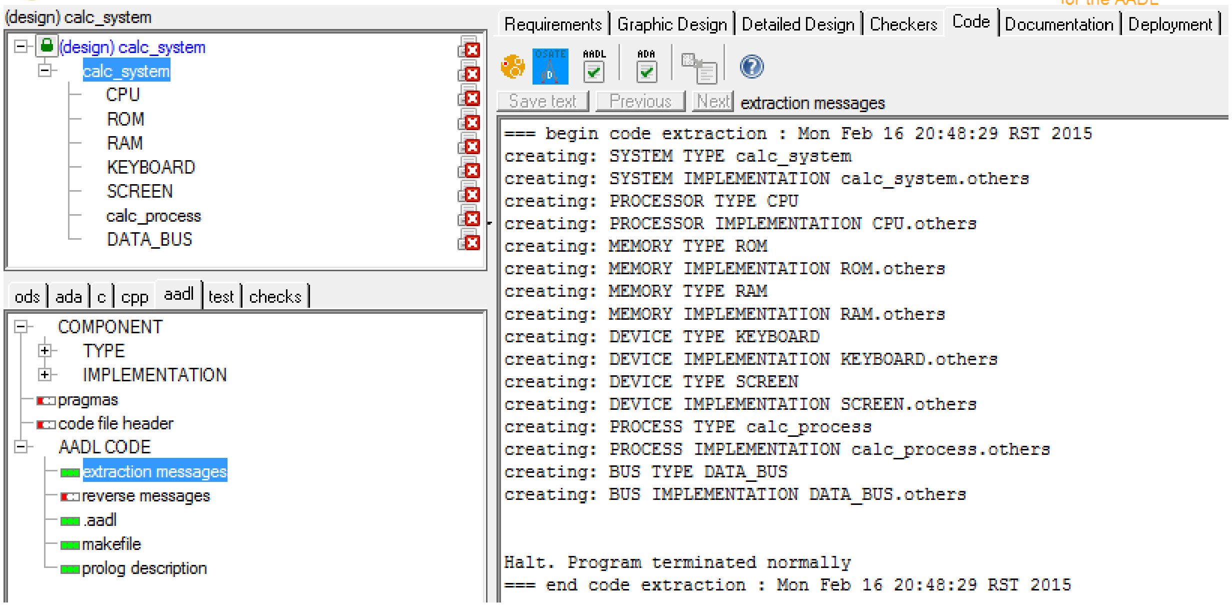
The new button bar shows two buttons. The first button on the left is called *add pragma* and may be used to customize the code generation. Pressing this button opens a dialog box showing the list of the possible options that can be selected.



In most cases, it is not required to activate these options. The **AADL** code generation is started by pressing the *full extraction* button. This opens a dialog box which is used to specify the part of the **Design** to be generated. Mostly, we need the whole **Design** to be generated, which is the default.



When the *Ok* button is pressed, the *extraction messages* are displayed and the **AADL Inspector** is opened in a separate window to view and analyse the code generated. Please refer to the **AADL Inspector** User manual to learn about the provided model analysis tools, such as scheduling analysis and run-time simulation.

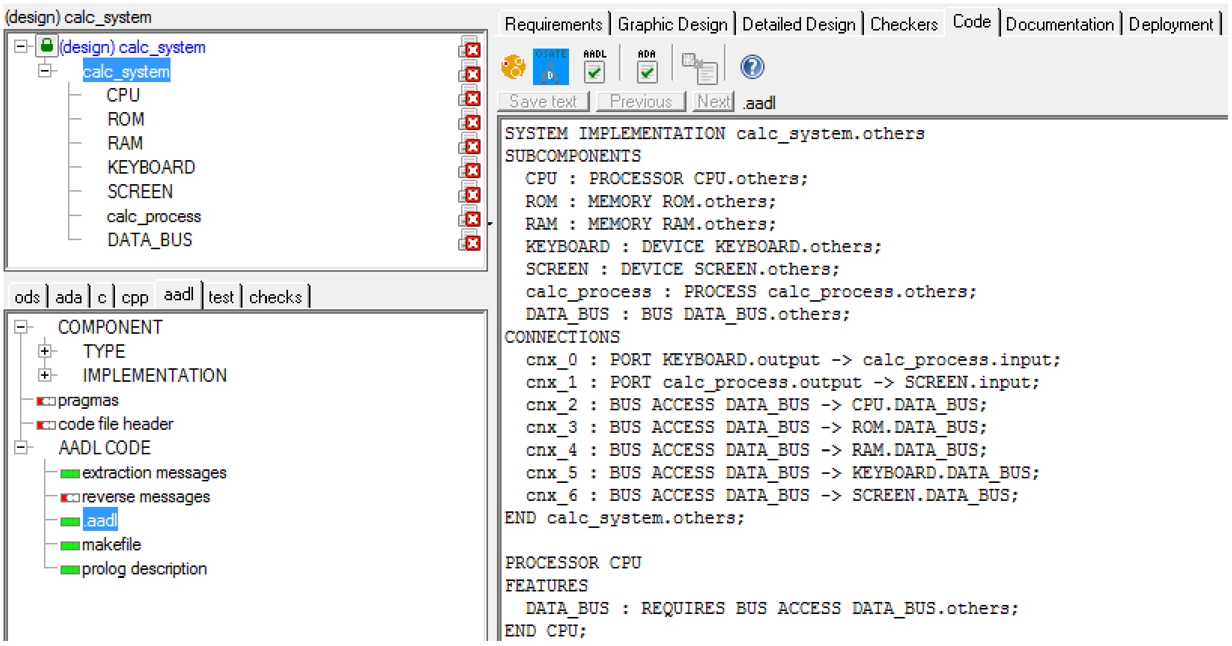


```
=== begin code extraction : Mon Feb 16 20:48:29 RST 2015
creating: SYSTEM TYPE calc_system
creating: SYSTEM IMPLEMENTATION calc_system.others
creating: PROCESSOR TYPE CPU
creating: PROCESSOR IMPLEMENTATION CPU.others
creating: MEMORY TYPE ROM
creating: MEMORY IMPLEMENTATION ROM.others
creating: MEMORY TYPE RAM
creating: MEMORY IMPLEMENTATION RAM.others
creating: DEVICE TYPE KEYBOARD
creating: DEVICE IMPLEMENTATION KEYBOARD.others
creating: DEVICE TYPE SCREEN
creating: DEVICE IMPLEMENTATION SCREEN.others
creating: PROCESS TYPE calc_process
creating: PROCESS IMPLEMENTATION calc_process.others
creating: BUS TYPE DATA_BUS
creating: BUS IMPLEMENTATION DATA_BUS.others


Halt. Program terminated normally
=== end code extraction : Mon Feb 16 20:48:29 RST 2015
```

The display lists the component types and implementations that were created to fully describe our **System** instance.

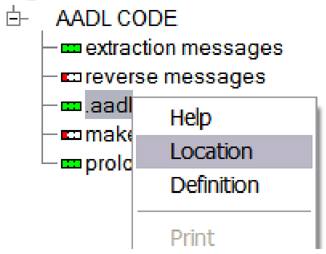
3.12 *Show generated AADL code*

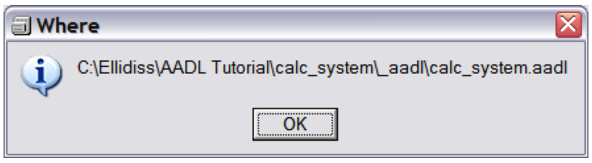
The **AADL** generated code is shown by changing the selection in the lower left list of the **Stood** window from *extraction messages* to *aadl*.



```
SYSTEM IMPLEMENTATION calc_system.others
SUBCOMPONENTS
  CPU : PROCESSOR CPU.others;
  ROM : MEMORY ROM.others;
  RAM : MEMORY RAM.others;
  KEYBOARD : DEVICE KEYBOARD.others;
  SCREEN : DEVICE SCREEN.others;
  calc_process : PROCESS calc_process.others;
  DATA_BUS : BUS DATA_BUS.others;
CONNECTIONS
  cnx_0 : PORT KEYBOARD.output -> calc_process.input;
  cnx_1 : PORT calc_process.output -> SCREEN.input;
  cnx_2 : BUS ACCESS DATA_BUS -> CPU.DATA_BUS;
  cnx_3 : BUS ACCESS DATA_BUS -> ROM.DATA_BUS;
  cnx_4 : BUS ACCESS DATA_BUS -> RAM.DATA_BUS;
  cnx_5 : BUS ACCESS DATA_BUS -> KEYBOARD.DATA_BUS;
  cnx_6 : BUS ACCESS DATA_BUS -> SCREEN.DATA_BUS;
END calc_system.others;

PROCESSOR CPU
FEATURES
  DATA_BUS : REQUIRES BUS ACCESS DATA_BUS.others;
END CPU;
```

The **AADL** code can be edited with **Stood**, although the corresponding file in the repository is easily located for remote access. To locate a particular file, select the corresponding entry in the lower left list and use the *Location* contextual menu.

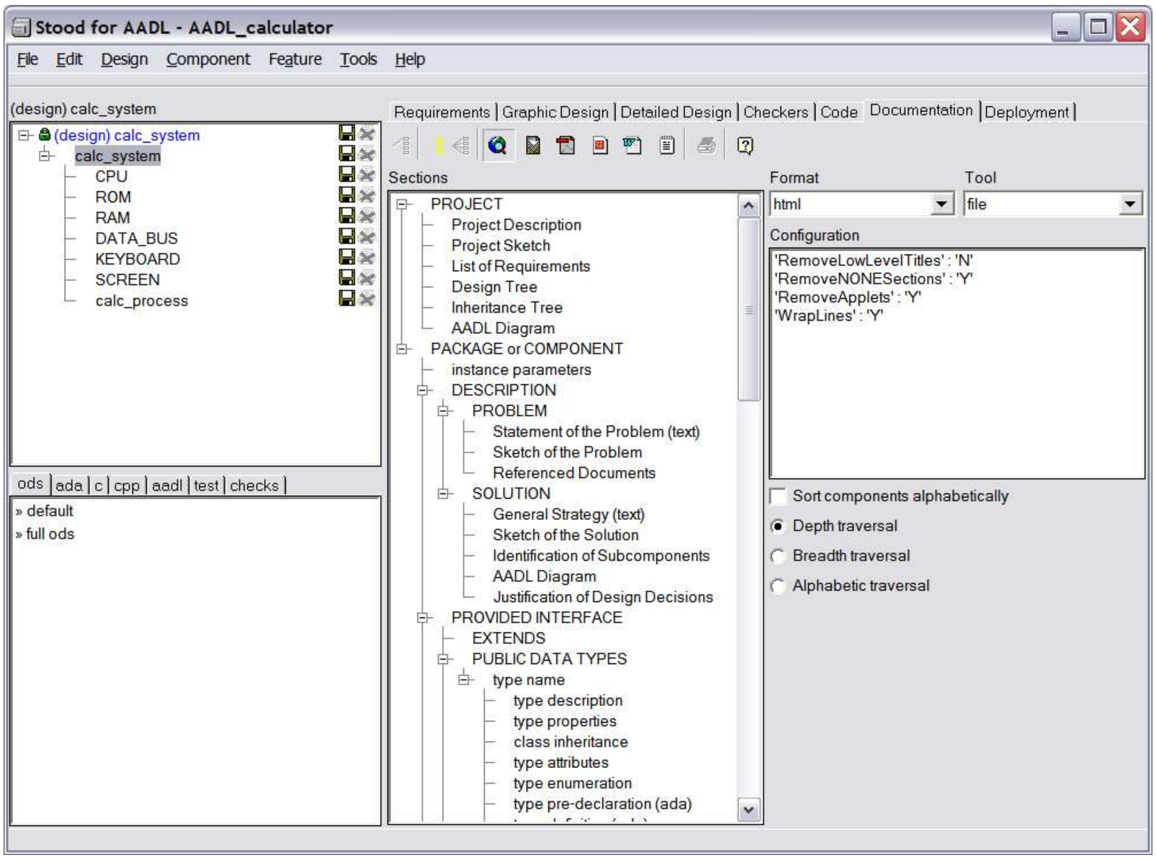




C:\Ellidiss\AADL Tutorial\calc_system_aadl\calc_system.aadl

3.13 Create a design report

Stood also offers a way to create design documentation reports. Such reports compile all the appropriate information that was entered while building the model. This includes textual and graphical entries. To switch on the report settings tool, select the *documentation* tab.


Stood for AADL - AADL_calculator
File Edit Design Component Feature Tools Help
Requirements | Graphic Design | Detailed Design | Checkers | Code | Documentation | Deployment |
ods | ada | c | cpp | aadl | test | checks |

(design) calc_system
- (design) calc_system
  - calc_system
    - CPU
    - ROM
    - RAM
    - DATA_BUS
    - KEYBOARD
    - SCREEN
    - calc_process

» default
» full ods

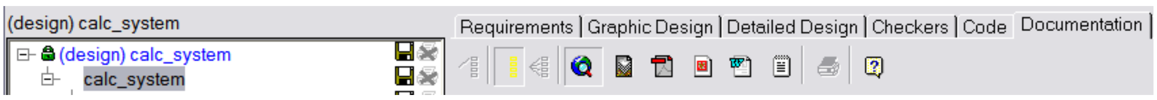
Sections
- PROJECT
  - Project Description
  - Project Sketch
  - List of Requirements
  - Design Tree
  - Inheritance Tree
  - AADL Diagram
- PACKAGE or COMPONENT
  - instance parameters
  - DESCRIPTION
    - PROBLEM
      - Statement of the Problem (text)
      - Sketch of the Problem
      - Referenced Documents
    - SOLUTION
      - General Strategy (text)
      - Sketch of the Solution
      - Identification of Subcomponents
      - AADL Diagram
      - Justification of Design Decisions
  - PROVIDED INTERFACE
    - EXTENDS
    - PUBLIC DATA TYPES
      - type name
        - type description
        - type properties
        - class inheritance
        - type attributes
        - type enumeration
        - type pre-declaration (ada)

Format: html
Tool: file

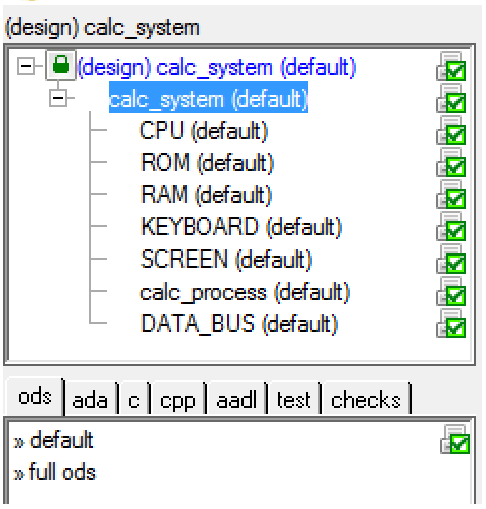
Configuration
```
'RemoveLowLevelTitles' : 'N'
'RemoveNONESections' : 'Y'
'RemoveApplets' : 'Y'
'WrapLines' : 'Y'
```

☐ Sort components alphabetically
● Depth traversal
○ Breadth traversal
○ Alphabetic traversal

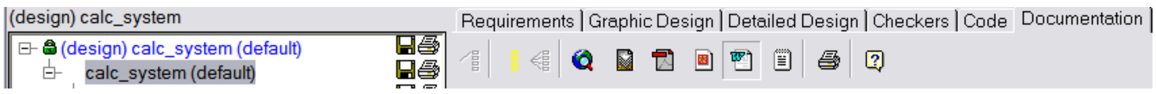
First define the components to be included in the report. To select all components, use the *select all* button.


(design) calc_system
(design) calc_system
calc_system
Requirements | Graphic Design | Detailed Design | Checkers | Code | Documentation |

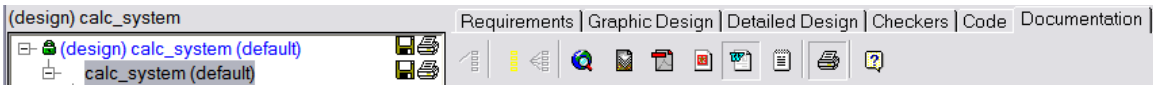
The effect of this action is to display a green tick over the small printer icon to the left of the component showing that it is included in the report (the small floppy disk icon means that the component has not been saved yet).



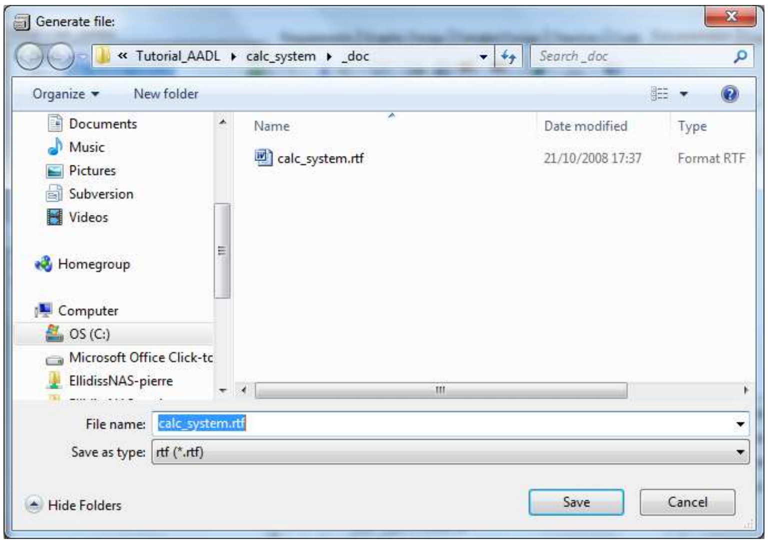
Next select the output format for the report. This can be done by selecting the appropriate button among *html* (default), *mif*, *pdf*, *ps*, *rtf* and *odt*. Select *rtf*.



Next start the document generation by pressing the *print* button.



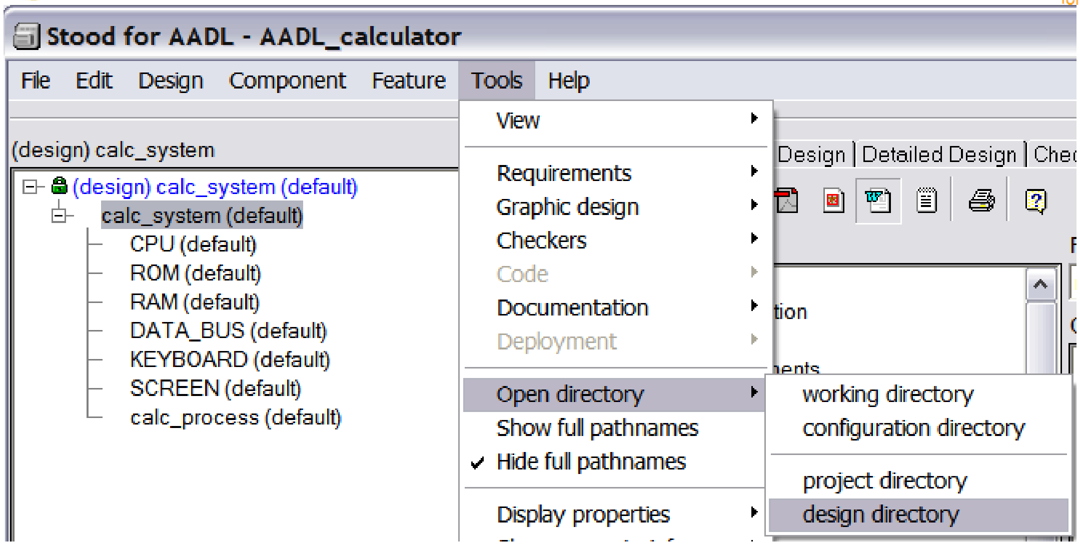
A standard Windows file navigator is shown to select the output file. A default filename and directory is provided.



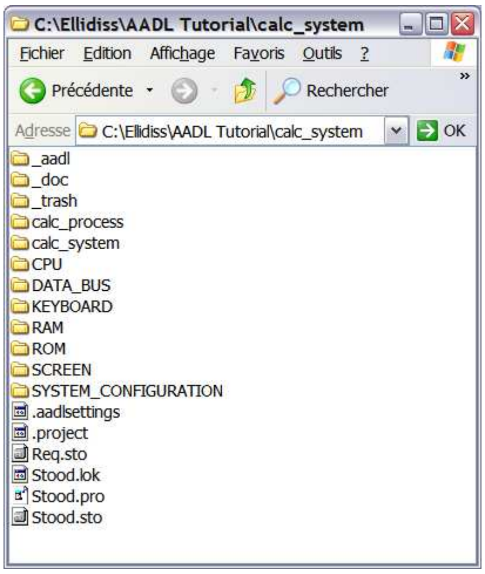
Name	Date modified	Type
calc_system.rtf	21/10/2008 17:37	Format RTF

File name: calc_system.rtf
Save as type: rtf (*.rtf)

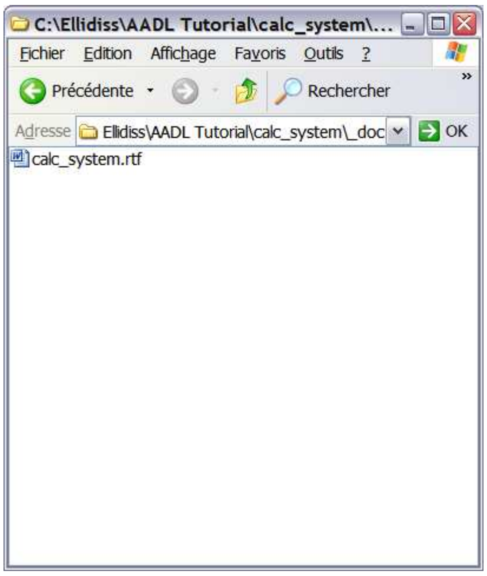
Once the document generator has finished, the report can be viewed by opening the file from its saved location. If the default output file location is used, it can be found by using the menu *Tools/Open directory/design directory*.



The design directory contains all the information related to the **Design**. Generated documentation is found under the `_doc` subdirectory.


C:\Ellidiss\AADL Tutorial\calc_system
Fichier Edition Affichage Favoris Outils ?
Précédente Rechercher
Adresse C:\Ellidiss\AADL Tutorial\calc_system OK

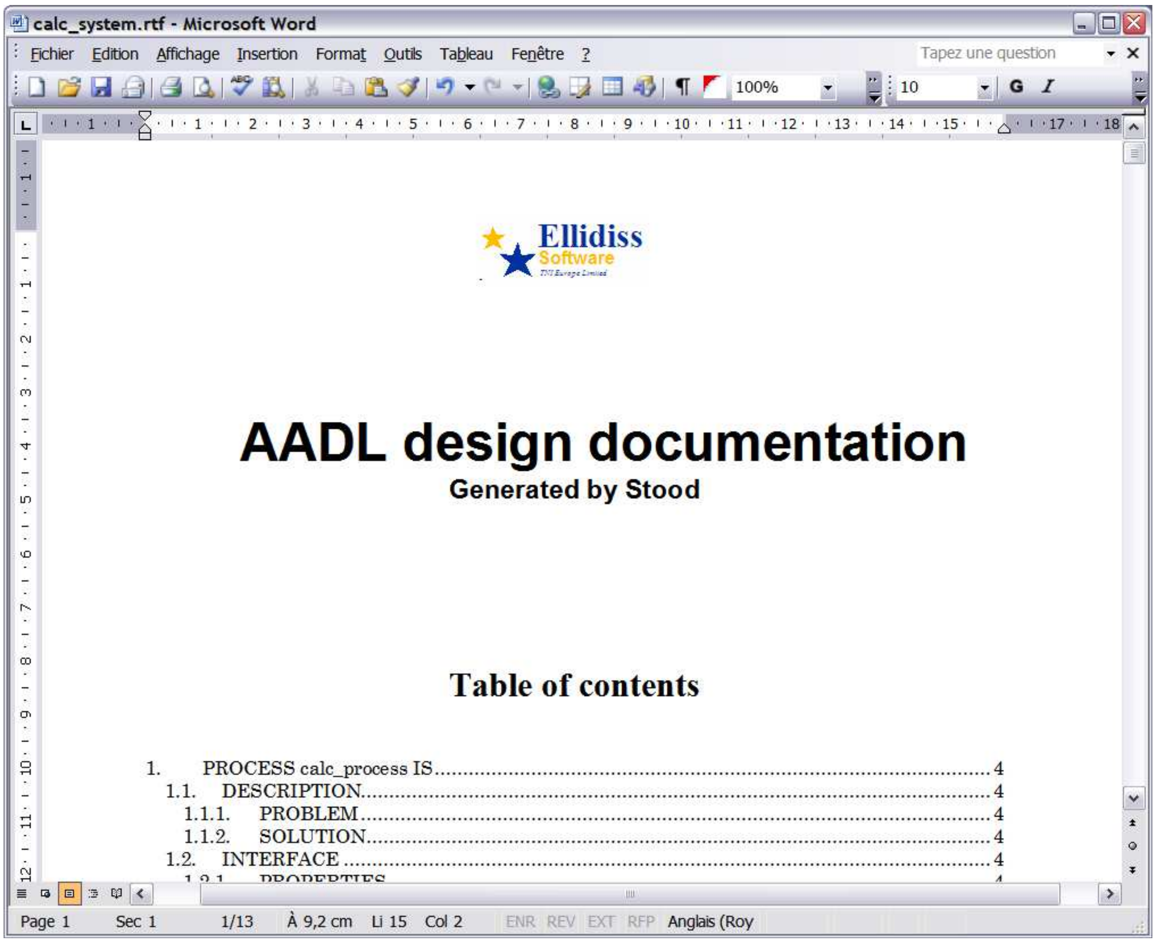
_aadl
_doc
_trash
calc_process
calc_system
CPU
DATA_BUS
KEYBOARD
RAM
ROM
SCREEN
SYSTEM_CONFIGURATION
.aadlsettings
.project
Req.sto
Stood.lok
Stood.pro
Stood.sto


C:\Ellidiss\AADL Tutorial\calc_system\...
Fichier Edition Affichage Favoris Outils ?
Précédente Rechercher
Adresse Ellidiss\AADL Tutorial\calc_system_doc OK

calc_system.rtf

Note that the **AADL** code that was generated previously is found under the `_aadl` subdirectory.

The figure below shows the result of the documentation generation for the **RTF** format.


calc_system.rtf - Microsoft Word
Fichier Edition Affichage Insertion Format Outils Tableau Fenêtre ?
Tapez une question
100%
10
Ellidiss Software

AADL design documentation

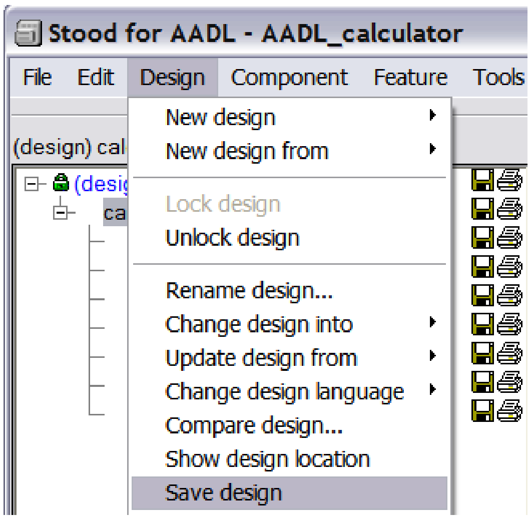
Generated by Stood

Table of contents

Page 1 Sec 1 1/13 À 9,2 cm Li 15 Col 2 ENR REV EXT RFP Anglais (Roy

3.14 Save the design

It is recommended that the design is saved to the design directory regularly. This is done by selecting *Design/Save design*.



Note that the save icons that are shown at the right of the component names will disappear once they are saved.

4 Create an AADL package

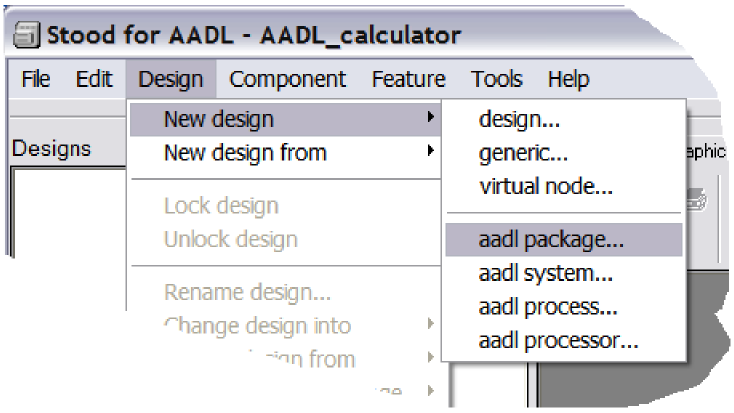
In the previous section, we did not explain how existing component classifiers could be referenced in other **AADL** models. This is however mandatory to enable proper component reuse.

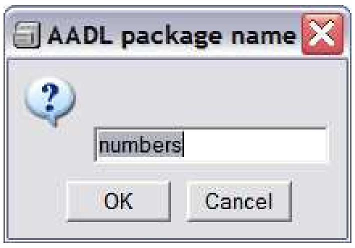
In order to be properly referenced with a **Project** wide scope, it is a good practice to group component classifier definitions within **AADL Packages**. In theory, this should be done for any category of component, however the particular case of **Data** component classifiers is especially important as they are not only instantiated to create subcomponents, but also to specify the actual data type of **Data Ports**, **Event Data Ports** or **Subprogram Parameters**.

This section explains how to create an **AADL Package** that provides a set of **Data** component classifiers.

4.1 Create a new design of type "aadl package"

To create a new **Package** inside the current **Project** (cf.§2.5), use the menu *Design/New design/aadl package…* and then specify its name in the dialog box.

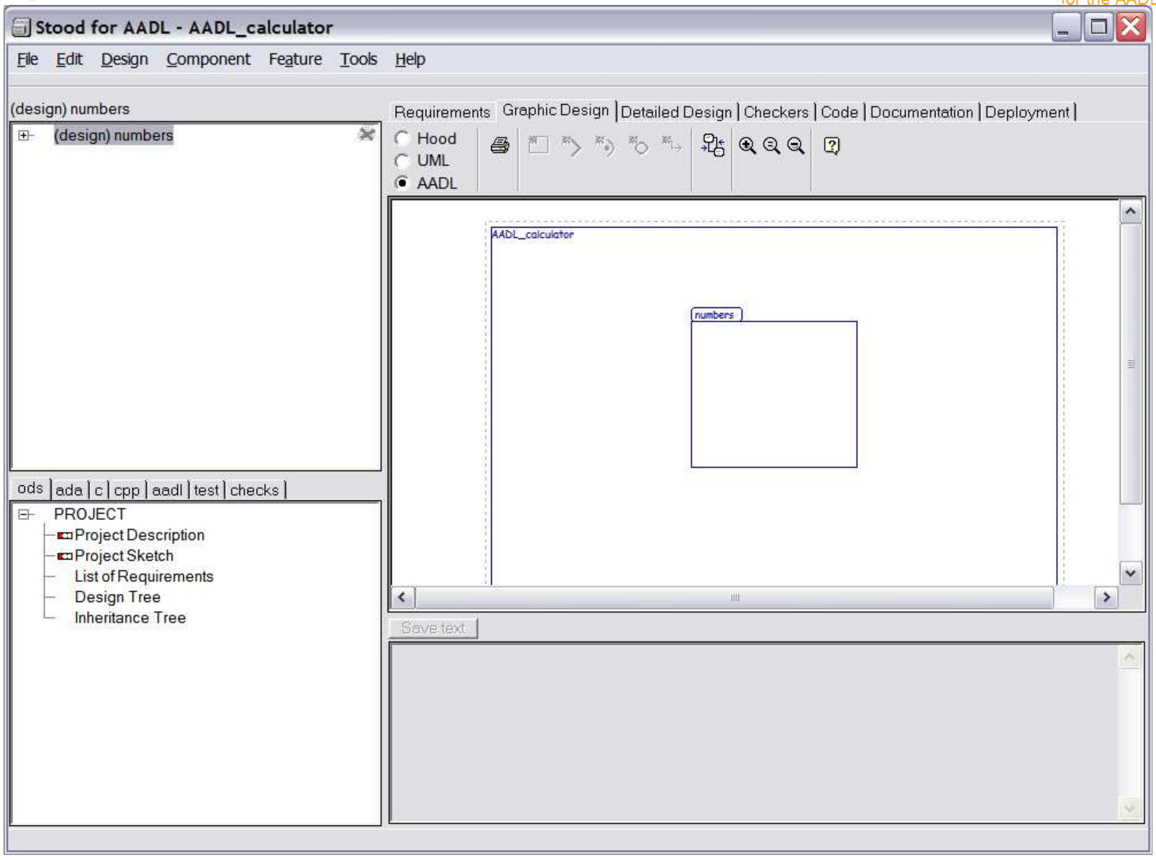




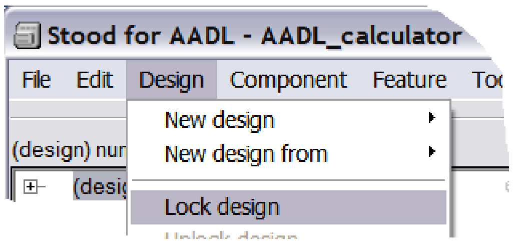
Note that a **Design** name in **Stood** must be alphanumeric (i.e. only contain characters '`a`' to '`z`', '`A`' to '`Z`', '`0`' to '`9`' or the underscore character '`_`').

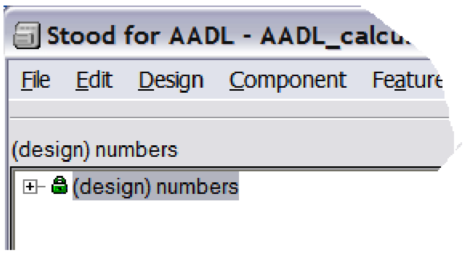
4.2 Lock the package to enter edit mode

Once created, the new **Package** design is automatically loaded and shown in the **AADL** graphical editor as an empty box in the middle of a larger one representing the **Project**.


Stood for AADL - AADL_calculator
File Edit Design Component Feature Tools Help
(design) numbers
Requirements Graphic Design Detailed Design Checkers Code Documentation Deployment
Hood
UML
AADL
AADL_calculator
numbers
ods ada c cpp aadl test checks
PROJECT
Project Description
Project Sketch
List of Requirements
Design Tree
Inheritance Tree
Save text

This **Package** design is set to read-only mode by default. To enable modifications on this **Package**, you need to "lock" it to prevent other users getting concurrent write access to the model.

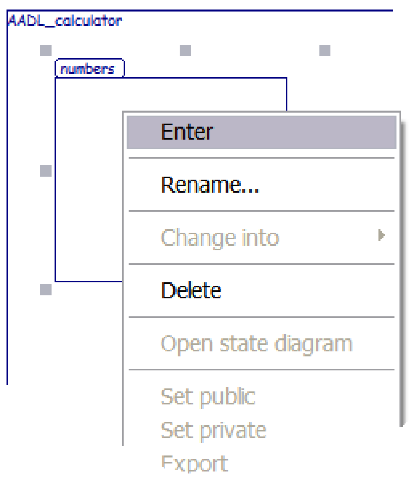

Stood for AADL - AADL_calculator
File Edit Design Component Feature
New design
New design from
Lock design
(design) numbers

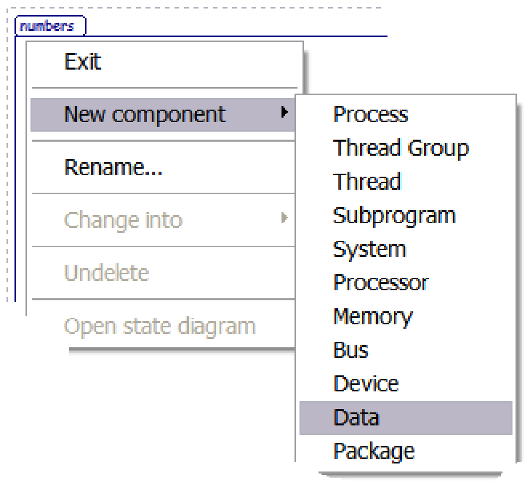

Stood for AADL - AADL_calcu
File Edit Design Component Feature
(design) numbers
(design) numbers

When a **Package** design is locked (may be modified in the current **Session**), a green padlock is shown to the left of its name.

4.3 *Create Data component classifiers inside the package*

To create components in the **Package**, it must be opened first. To open a **Package**, it must be selected using the contextual menu *enter* option. An alternate way of doing this is to perform a mouse double click inside the boundaries of the **Package**.

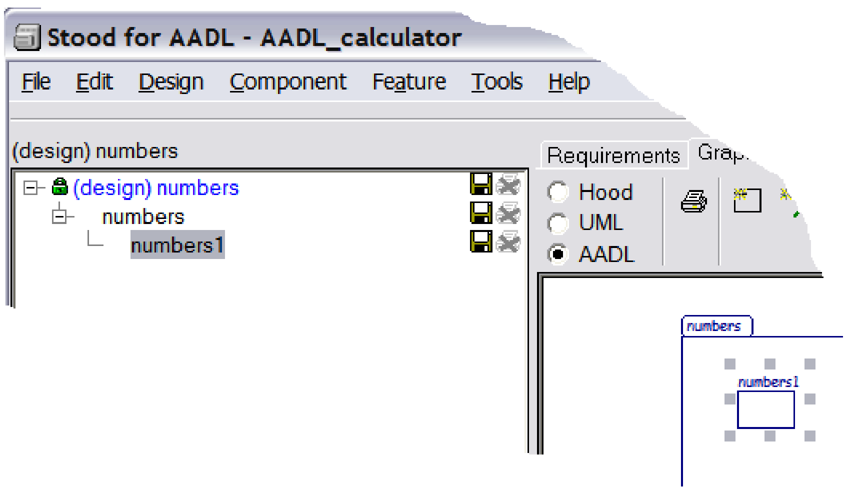


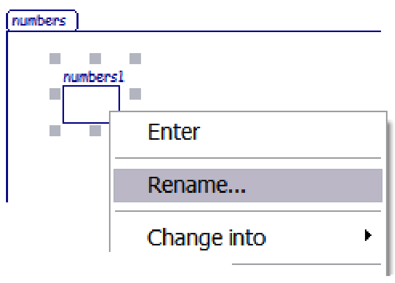


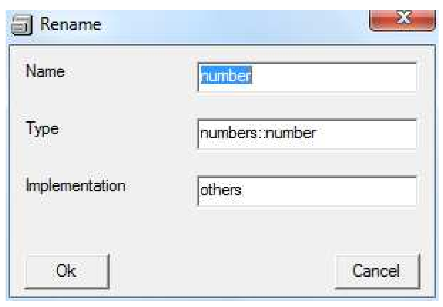
Once the **Package** is opened and selected, you can create components, either with the *new AADL component* button (in which case a **Data** component classifier is created by default), or with the *new component* contextual menu. After the button or the menu has been used, a new box is shown on the diagram and a new component is added to the top left list. The default name given to new components is the name of the container box followed by an integer value.

4.4 Rename the Data component

To rename the new component, select it (on the diagram or in the list), and select the contextual menu *rename* option. This opens a dialog box where the actual name of the component can be entered.







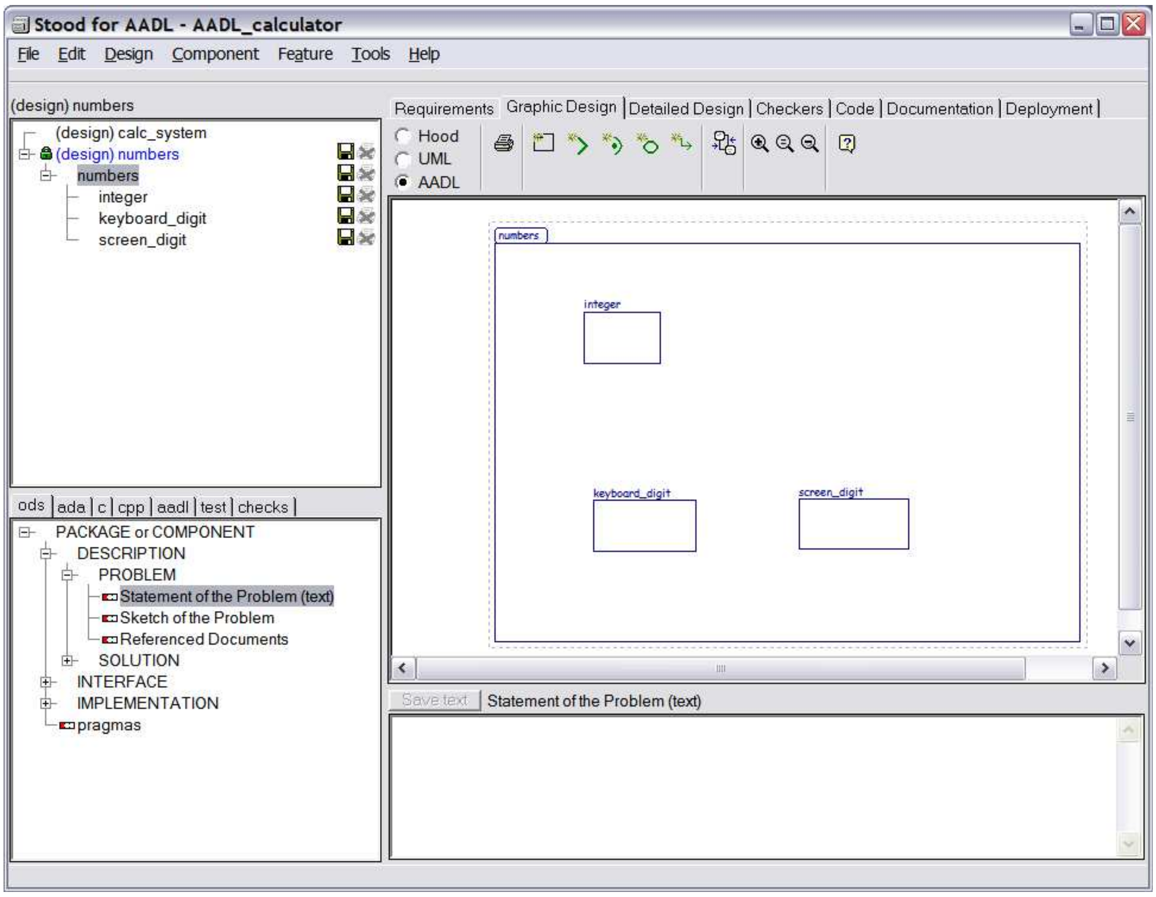
Name	number
Type	numbers::number
Implementation	others

Note that a default value for the *Type* field is automatically updated when the *Name* field is edited

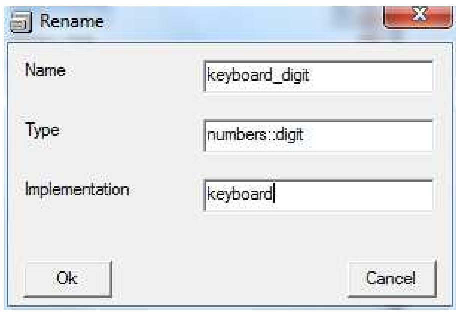
Also note that within a **Package**, only component classifiers are described, so the **Stood** component name generally matches the **AADL** component type name. However, in case where several components are the same type, but different implementations, the box name is used to distinguish them, as explained in next chapter.

4.5 Specify component type and implementation

Within a **Package**, a unique **Stood** component is used to represent both the **AADL** component type and implementation. If two components have the same type and different implementations, two **Stood** components must be created. The three fields of the *rename* dialog box are then used to specify the unique **Stood** component name, the **AADL** type name and the **AADL** implementation name.

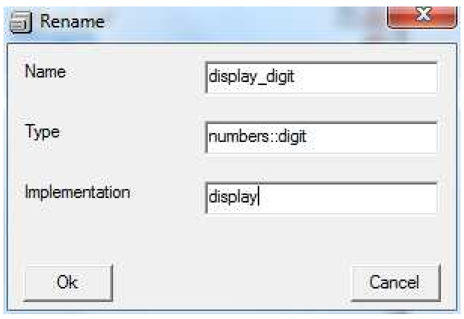


In the example below, two **Stood** components `keyboard_digit` and `screen_digit` have been created. They have the same **AADL** type name `digit` and different implementation name `keyboard` and `screen`.



Rename

Name	keyboard_digit
Type	numbers::digit
Implementation	keyboard



Rename

Name	display_digit
Type	numbers::digit
Implementation	display

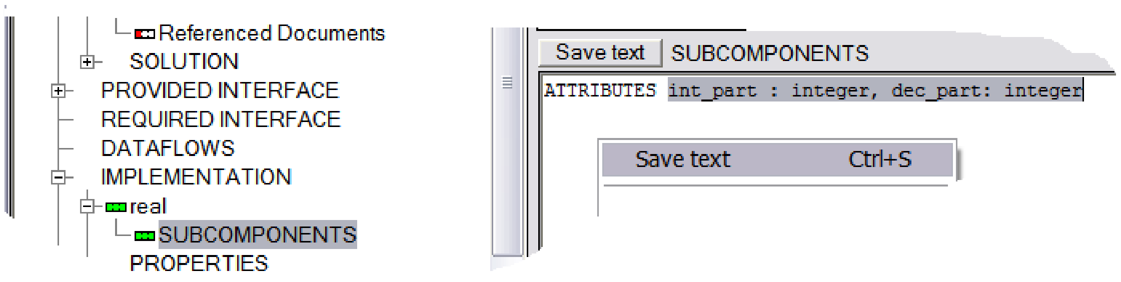
Note that if no implementation has to be produced for the component type, the value `NIL` must be entered in the *Implementation* field. However, if an implementation becomes required to generate a correct **AADL** specification (for Instance, if the component has subcomponents), the default value `others` is used.

4.6 Define subcomponents of a Data component

According to the **AADL** standard, a **Data** component classifier can accept subcomponents that must also be instances of **Data** components. As a special case, subcomponents of **Data** component classifiers are managed by **Stood** as a list of typed **Attributes**. This is in fact compliant with the way software structured data types (**Ada** record, **C** struct, **C++** class) are handled by the tool.

When a **Data** component classifier is selected on the diagram or in the top-left list, a dedicated *SUBCOMPONENTS* section is available inside the component descriptor in the bottom-left list. This section contains a formal declaration of the list of the **Data** subcomponent names associated with their corresponding classifier reference (name of a **Data** component classifier). Note that keyword `ATTRIBUTES` must not be removed and that the list separator is a comma.

To illustrate this, we can create a new **Data** component classifier called `real` and specify that it has two subcomponents `int_part` and `dec_part` that are both instances of `integer`.



```
ATTRIBUTES int_part : integer, dec_part: integer
```

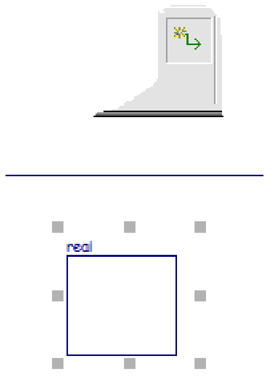
Note: do not forget to use the *save text* button after any change in the text input area.

4.7 Define Data component extension

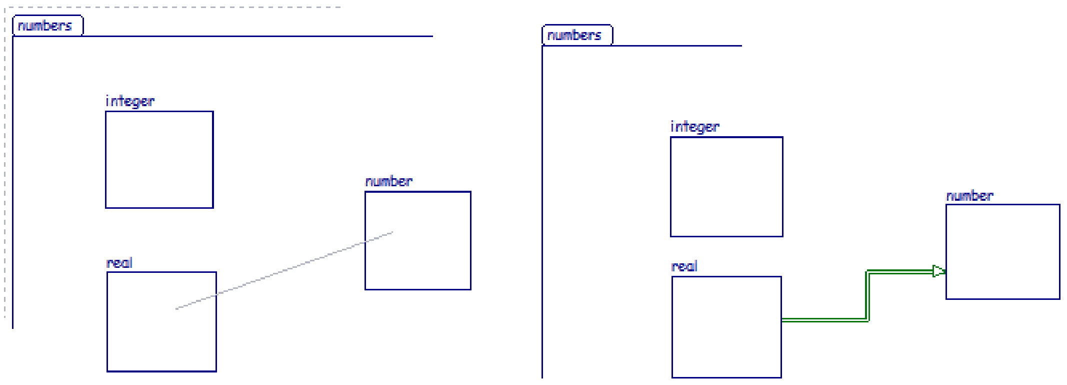
According to the **AADL** standard, a **Data** component classifier may be specified as an extension of another **Data** component classifier. Such an extension mechanism can be compared to software class inheritance in object-oriented languages. The fact that a **Data** component classifier extends another **Data** component classifier can be expressed on the diagram using a graphical connection. To create this connection, for example to specify that integers and floats are types of numbers, perform the following:

(a) create a new **Data** component classifier called `number`

(b) click on the new connection button of the **AADL** graphical editor

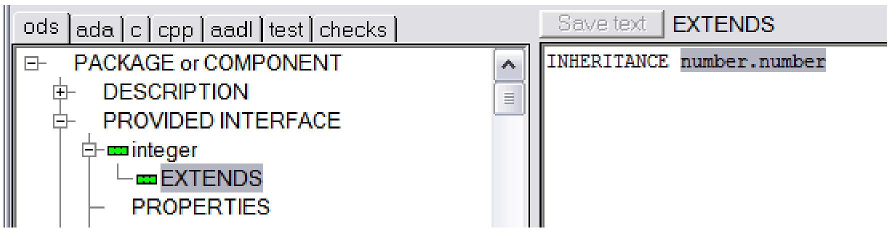


(c) select the descendent **Data** component classifier



(d) select the ancestor **Data** component classifier.
Note: the graphical representation of the extend connection does not comply with the recommendation of the annex A of the standard.

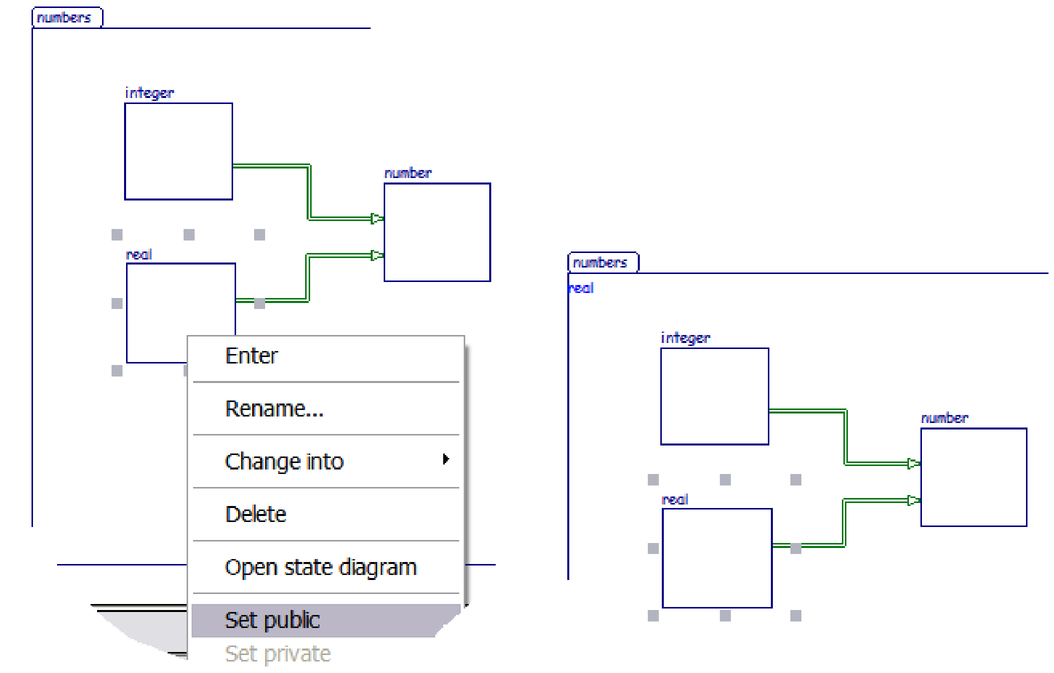
This component extension information can also be edited textually. It can be accessed by selecting the *EXTENDS* section of the component descriptor while the component is selected on the diagram or in the top-left list.



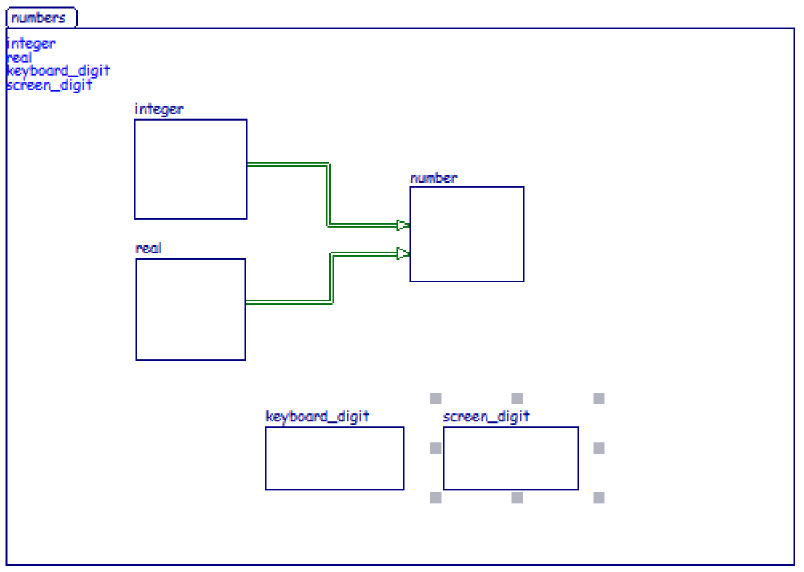
```
INHERITANCE number.number
```

4.8 Define the public section of the package

When adding **Data** component classifiers in a **Package** with **Stood**, they are put in its private section by default. To make a **Data** component classifier public, select it and use the contextual menu *Set public* option.



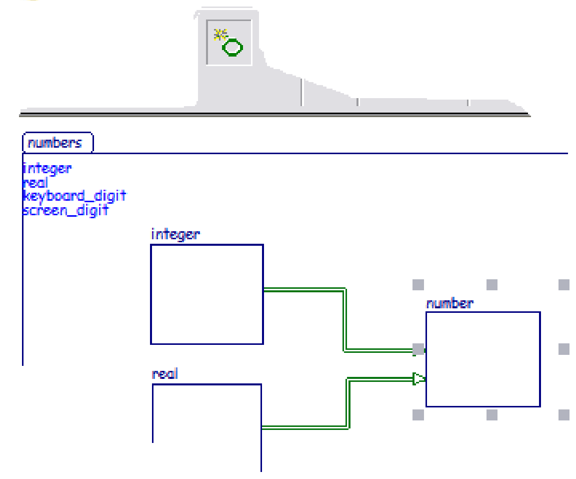
To distinguish between public and private **Data** component classifiers, the former are listed on the left border of the **Package** box in the diagram. Note this graphical notation is specific to **Stood**. For the purpose of our example, we can specify that **Data** component classifiers `integer`, `float`, `keyboard_digit` and `screen_digit` are public, whereas `number` remains private.



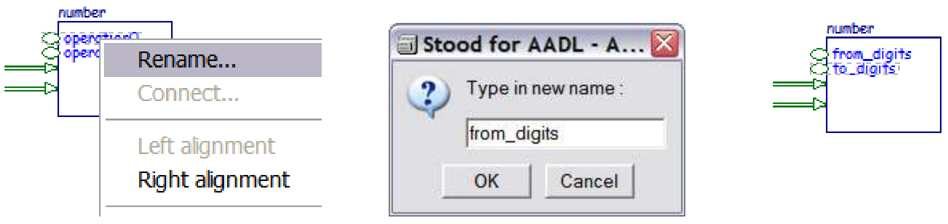
4.9 *Define Data Subprograms*

Similar to other object oriented languages, it is possible to define the methods or member functions that are associated with a **Data** component classifier. In **AADL** they are called data **Subprogram** features.

To create **Subprograms**, click the *new subprogram* button in the **AADL** graphical editor, and select a Data component classifier.



The new **Subprogram** is given a default name. The rename contextual menu is used to change it.



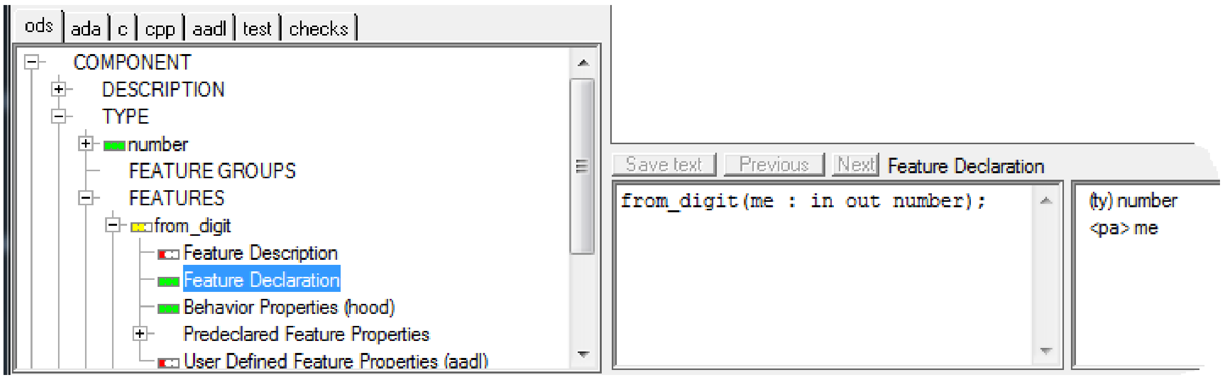
4.10 *Specify Subprogram Parameters*

Whereas **Ports** can carry a single event or data message, **Subprograms** can express more complex dataflows that are defined by a list of directional typed **Parameters**. In **Stood**, this parameter list must be entered in the *Feature Declaration* section as shown in the sequence of screenshots below.

The syntax used by **Stood** to specify a list of **Parameters** for a **Subprogram** is the one recommended by **HOOD** which is very similar to the one defined by the **Ada** standard. The syntax for a single **Parameter** is (list separator is a semicolon):

```
<parameter_name> : <direction> <parameter_type>
```

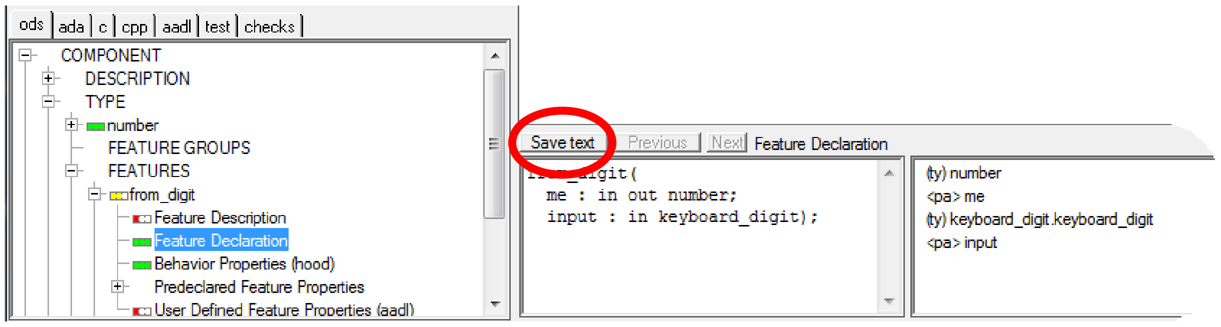
(a) Select the **Subprogram** in the diagram or in the list. The default list of **Parameters** is shown in the text input area. In case of a data **Subprogram** defined in a **Data** component classifier, the default **Parameter** is the receiver whose default name is “me” and default type is the **Data** component classifier.



```
from_digit(me : in out number);
```

(ty) number
<pa> me

(b) Modify the **Parameters** in the text input area. Do not forget to save the changes with the button, contextual menu or keyboard shortcut.

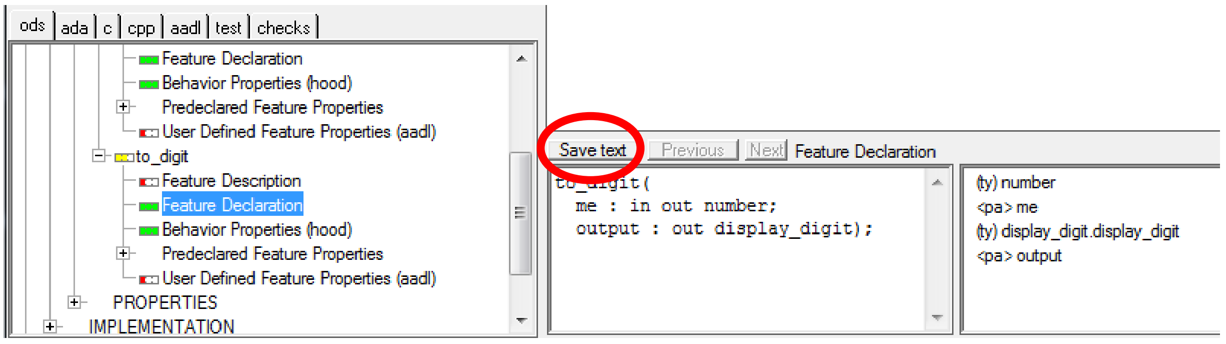


```
from_digit(
  me : in out number;
  input : in keyboard_digit);
```

(ty) number
<pa> me
(ty) keyboard_digit.keyboard_digit
<pa> input

(c) Note that the specified **Parameter** type is checked against a **Stood** cross reference table and the result of this analysis is shown in the right hand side of the text input area.

By specifying proper **Parameters**, we can express that the two data **Subprogram** features of **Data** component classifier `number` are `from_digits` with an input **Parameter** of type `keyboard_digit`, and `to_digits` with an output **Parameter** of type `screen_digit`.


ods | ada | c | cpp | aadl | test | checks
Feature Declaration
Behavior Properties (hood)
Predeclared Feature Properties
User Defined Feature Properties (aadl)
to_digit
Feature Description
Feature Declaration
Behavior Properties (hood)
Predeclared Feature Properties
User Defined Feature Properties (aadl)
PROPERTIES
IMPLEMENTATION
Save text
Previous
Next
Feature Declaration

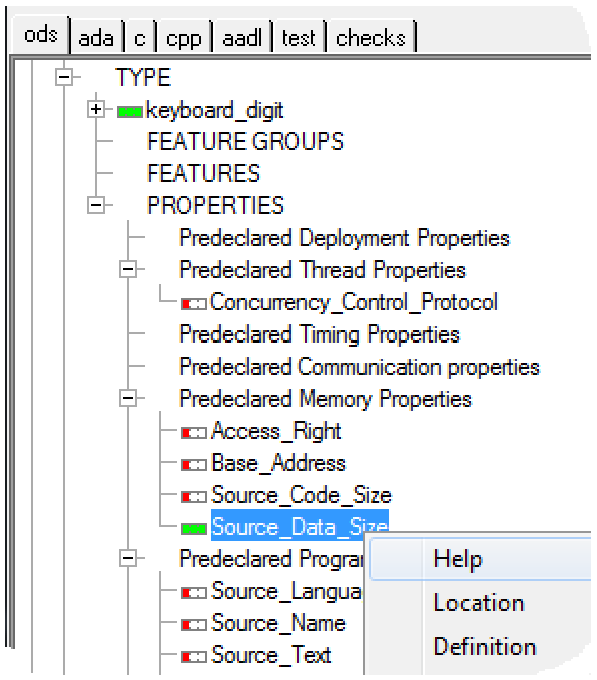
```
to_digit(
  me : in out number;
  output : out display_digit);
```

(ty) number
<pa> me
(ty) display_digit.display_digit
<pa> output

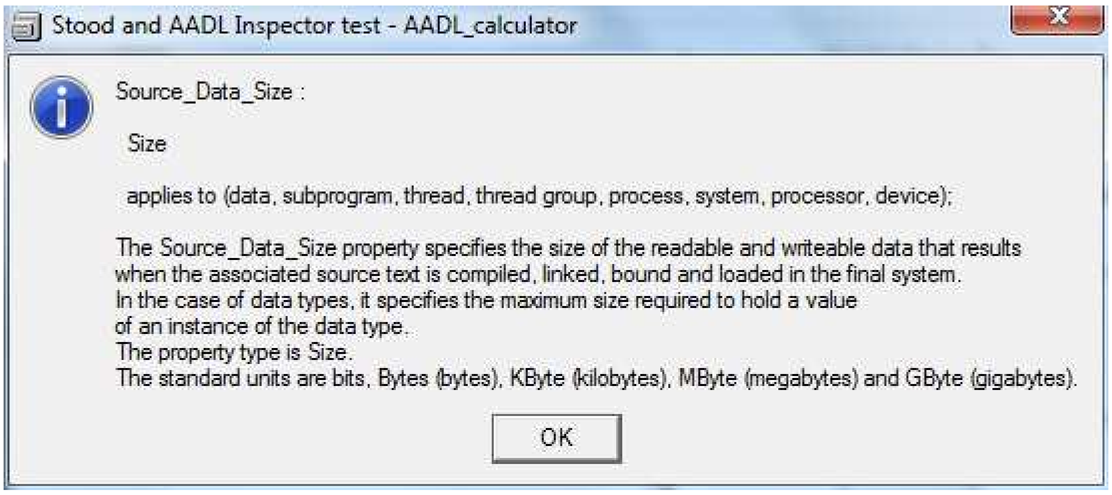
4.11 Add AADL Properties

AADL entities specification can be refined by a set of predefined or project-specific **Properties**. In **Stood 5.4**, all the predefined **Properties** have been included into the default configuration, which simplifies their use. Note that it is possible to customize this list in the tool configuration files, to hide the **Properties** that are not relevant for the current **Project**.

When a component or a feature is selected, the list of possible valid **Properties** is shown in the *TYPE* section. A contextual help is available for each individual **Property**.



To add a **Property** association, select the appropriate **AADL** entity (here, `keyboard_digit` **Data** component classifier), and the chosen **Property** in the list (here, `Source_Data_Size`).
Note that for each predefined **Property**, a help text can be displayed thanks to the *Help* contextual menu


Stood and AADL Inspector test - AADL_calculator

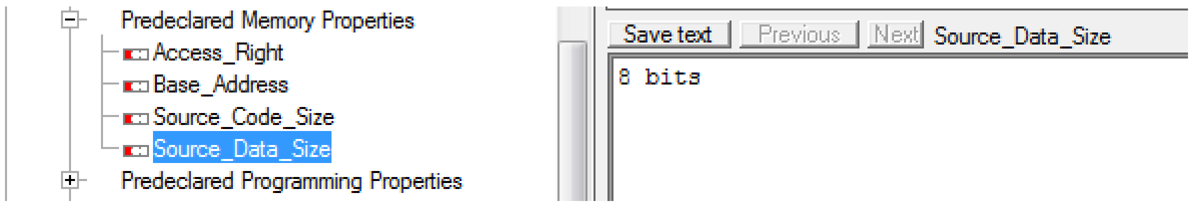
Source_Data_Size :

Size

applies to (data, subprogram, thread, thread group, process, system, processor, device);

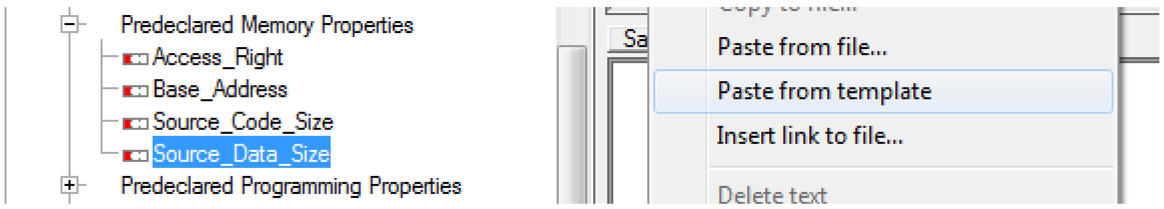
The Source_Data_Size property specifies the size of the readable and writeable data that results
when the associated source text is compiled, linked, bound and loaded in the final system.
In the case of data types, it specifies the maximum size required to hold a value
of an instance of the data type.
The property type is Size.
The standard units are bits, Bytes (bytes), KByte (kilobytes), MByte (megabytes) and GByte (gigabytes).

Then enter the **Property** value in the text input area and save with the button, the contextual menu or the keyboard shortcut.



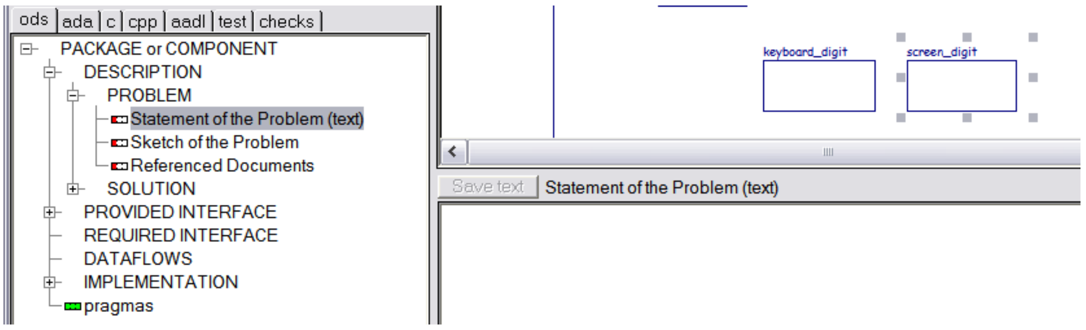
```
8 bits
```

Note that a default value is proposed for each **Property**. To display this default value, use *paste from template* contextual menu of the text input area.

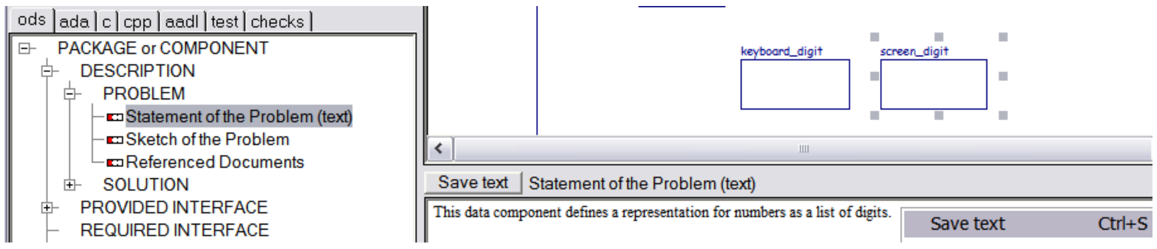


4.12 Add textual comments

It is recommended that you insert comments and other textual information inside the design data structure while creating new entities. These comments may be used to provide explanations about the "why", "what" and "how" of each component or feature. The standard configuration of the tool presents a structured list of comment sections that can however be customized to better fit any other documentation strategy. To fill in one of the proposed documentation sections, select the appropriate entity and one of the proposed (text) sections.



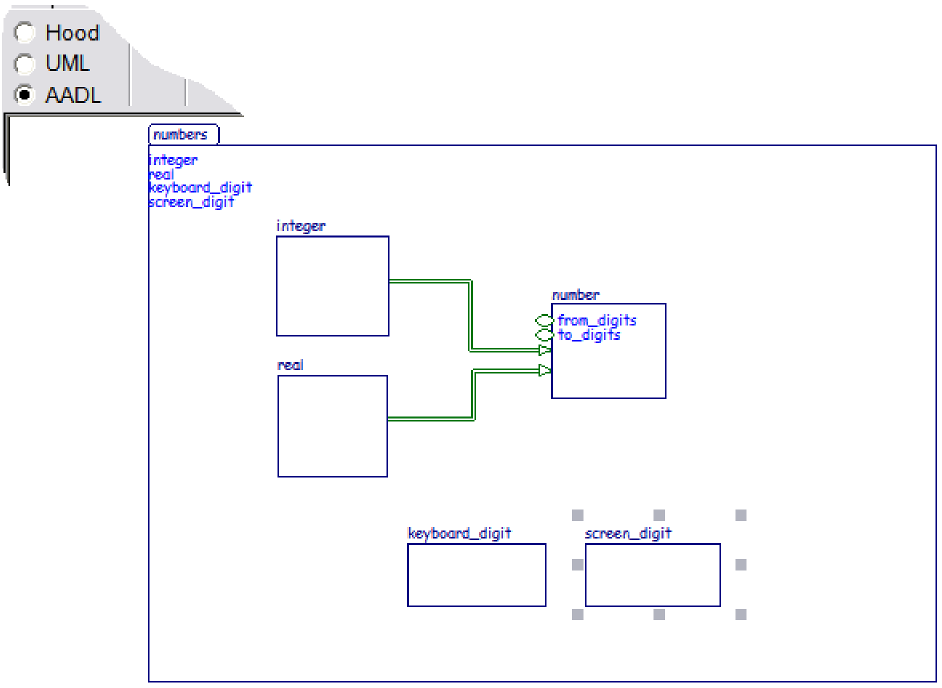
Text can be entered inside the text input area and must be saved using the button, contextual menu or keyboard shortcut.



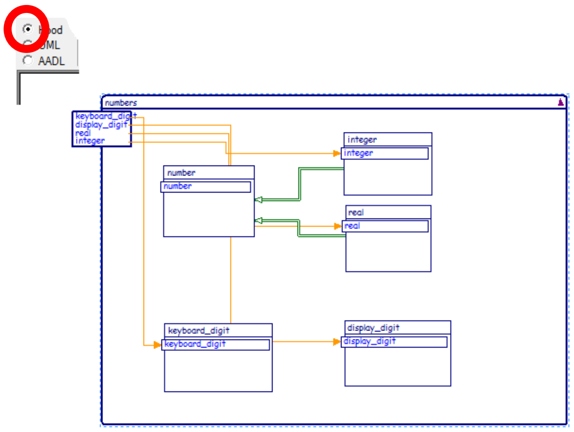
This data component defines a representation for numbers as a list of digits.

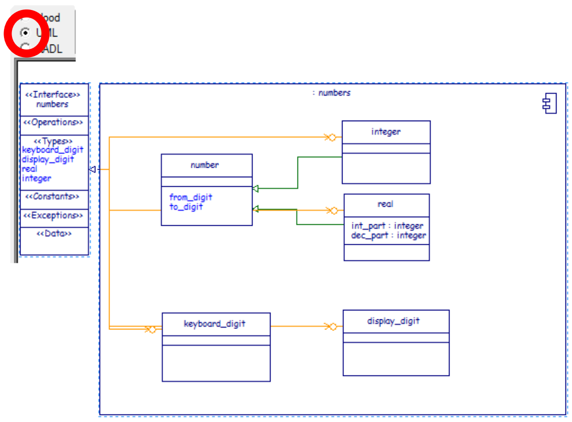
4.13 Show full AADL diagram of the package

The complete **AADL** diagram for our **Package** is now as shown below:



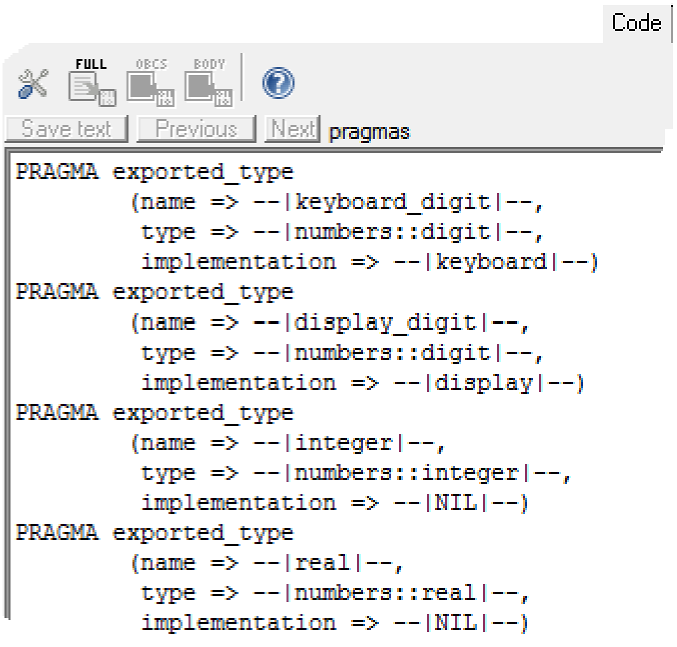
Note that the corresponding graphical representation of the same model in **UML** and **HOOD** is also available, simply by switching the notation selector. If the switch is inactive, please use the main menu *Design/Change design model* and select *ANY* to enable the multi-view display.



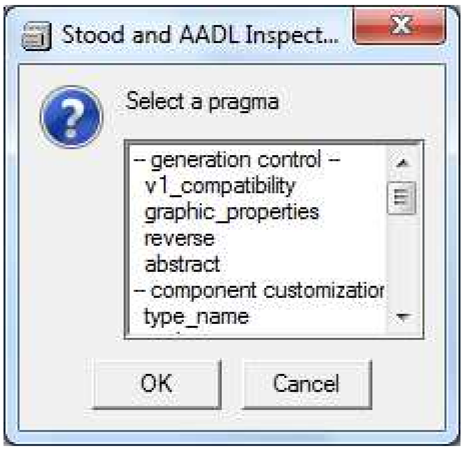


4.14 Generate the AADL code for the package

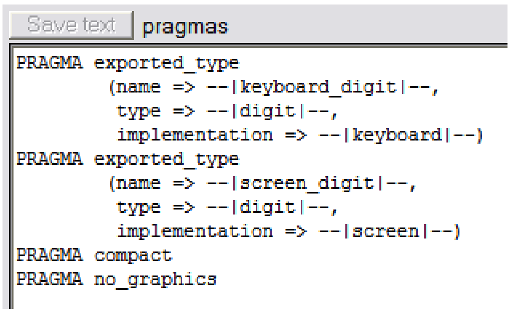
To generate textual **AADL** code from a graphical model in **Stood**, the *Code* tab is selected instead of the *Graphic Design* one. The new button bar shows two buttons. The first button on the left is called *add pragma* and may be used to customize the code generation. Pressing this button opens a dialog box showing the list of possible pragmas that can be selected. Note that some pragmas may have already been automatically inserted by **Stood**, as it is the case here.



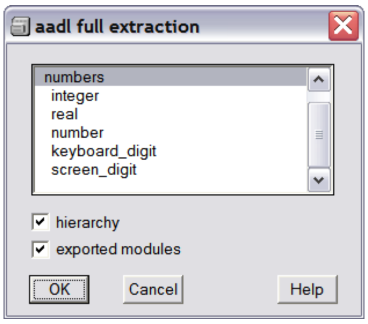
```
PRAGMA exported_type
         (name => --|keyboard_digit|--,
          type => --|numbers::digit|--,
          implementation => --|keyboard|--)
PRAGMA exported_type
         (name => --|display_digit|--,
          type => --|numbers::digit|--,
          implementation => --|display|--)
PRAGMA exported_type
         (name => --|integer|--,
          type => --|numbers::integer|--,
          implementation => --|NIL|--)
PRAGMA exported_type
         (name => --|real|--,
          type => --|numbers::real|--,
          implementation => --|NIL|--)
```



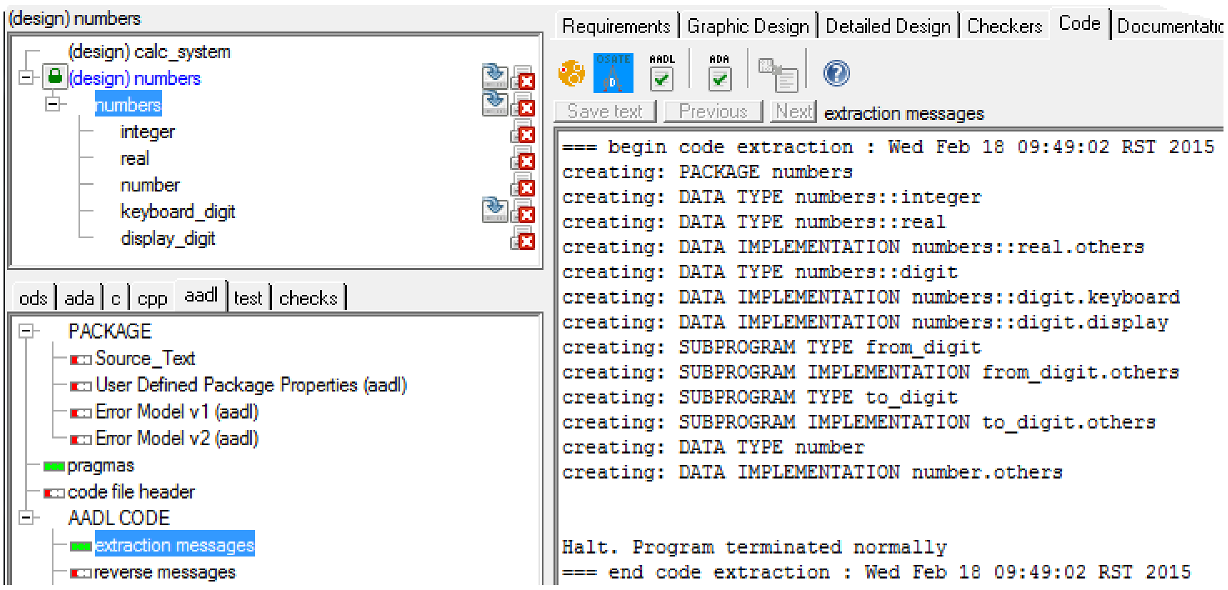
The **AADL** code generation is started by pressing the *full extraction* button. This opens a dialog box that can be used to specify which part of the **Design** has to be generated. More often, we need the whole **Design** generated, which is the default.



```
PRAGMA exported_type
         (name => --|keyboard_digit|--,
          type => --|digit|--,
          implementation => --|keyboard|--)
PRAGMA exported_type
         (name => --|screen_digit|--,
          type => --|digit|--,
          implementation => --|screen|--)
PRAGMA compact
PRAGMA no_graphics
```



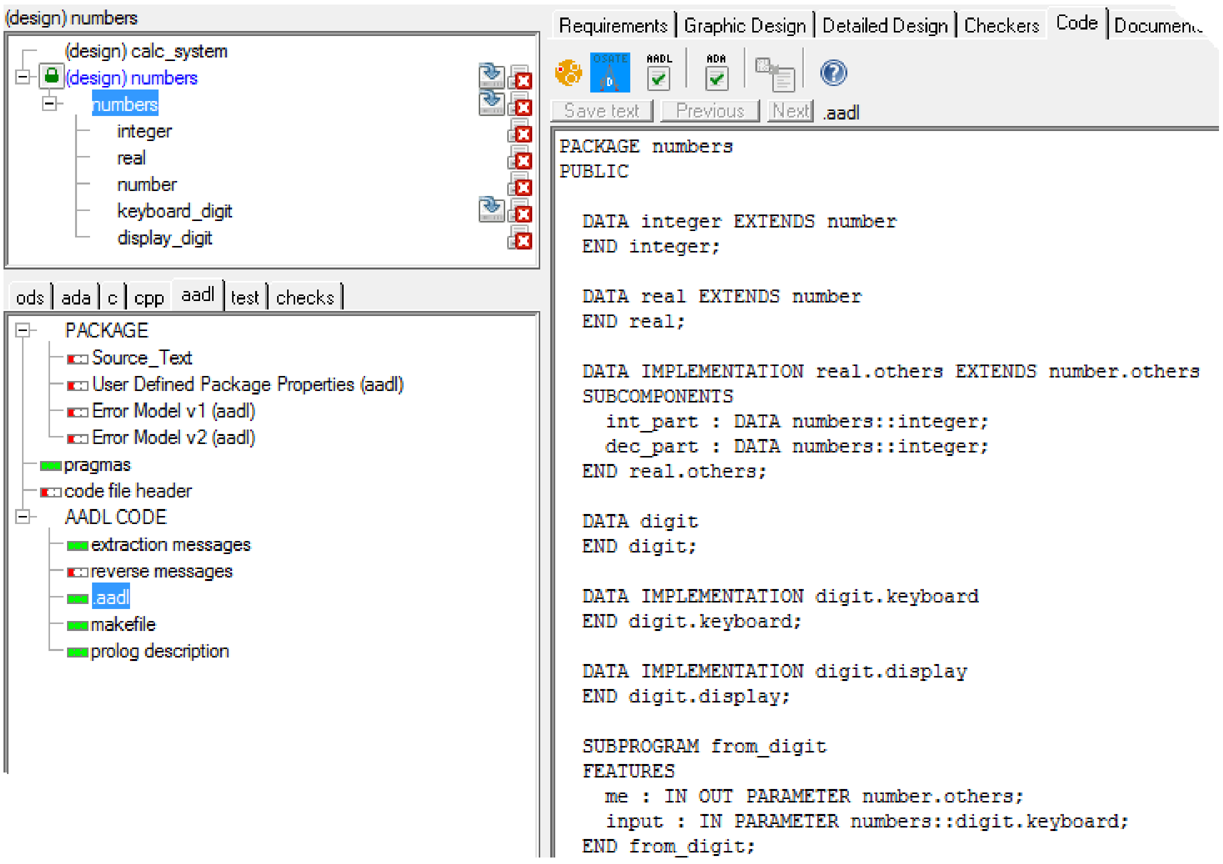
When the *Ok* button is pressed, the *extraction messages* are displayed and the **AADL Inspector** is opened in a separate window to view the code generated.



```
=== begin code extraction : Wed Feb 18 09:49:02 RST 2015
creating: PACKAGE numbers
creating: DATA TYPE numbers::integer
creating: DATA TYPE numbers::real
creating: DATA IMPLEMENTATION numbers::real.others
creating: DATA TYPE numbers::digit
creating: DATA IMPLEMENTATION numbers::digit.keyboard
creating: DATA IMPLEMENTATION numbers::digit.display
creating: SUBPROGRAM TYPE from_digit
creating: SUBPROGRAM IMPLEMENTATION from_digit.others
creating: SUBPROGRAM TYPE to_digit
creating: SUBPROGRAM IMPLEMENTATION to_digit.others
creating: DATA TYPE number
creating: DATA IMPLEMENTATION number.others


Halt. Program terminated normally
=== end code extraction : Wed Feb 18 09:49:02 RST 2015
```

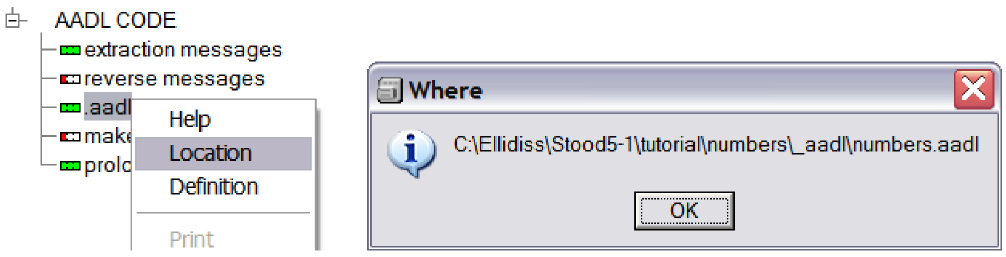
4.15 Show generated AADL code

The **AADL** generated code is shown by changing the selection in the lower left list of the **Stood** window from *extraction messages* to *aadl*.



```
PACKAGE numbers
PUBLIC

  DATA integer EXTENDS number
  END integer;

  DATA real EXTENDS number
  END real;

  DATA IMPLEMENTATION real.others EXTENDS number.others
  SUBCOMPONENTS
    int_part : DATA numbers::integer;
    dec_part : DATA numbers::integer;
  END real.others;

  DATA digit
  END digit;

  DATA IMPLEMENTATION digit.keyboard
  END digit.keyboard;

  DATA IMPLEMENTATION digit.display
  END digit.display;

  SUBPROGRAM from_digit
  FEATURES
    me : IN OUT PARAMETER number.others;
    input : IN PARAMETER numbers::digit.keyboard;
  END from_digit;
```

The **AADL** code can be edited with **Stood**, although the corresponding file in the repository is easily located for remote access. To locate a particular file, select the corresponding entry in the lower left list and use the *Location* contextual menu.

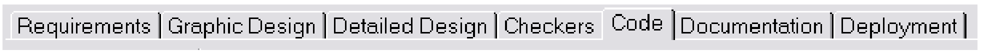


C:\Ellidiss\Stood5-1\tutorial\numbers_aadl\numbers.aadl

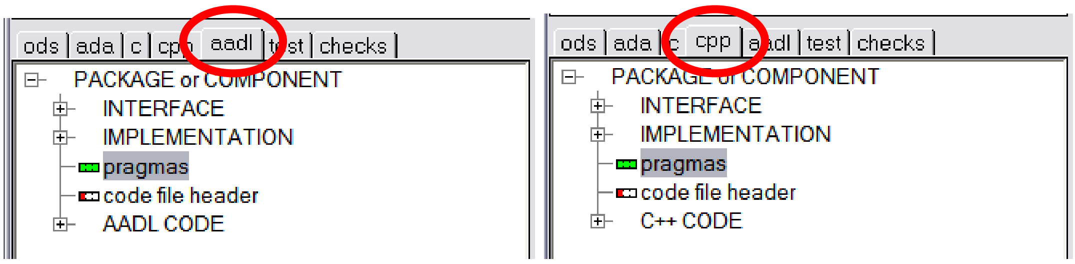
4.16 Generate C++ code

Without changing anything in the design model, you can generate **Ada**, **C** or **C++** source code. The process for generating **C++** code (for example) is very similar to generating the **AADL** code, i.e.:

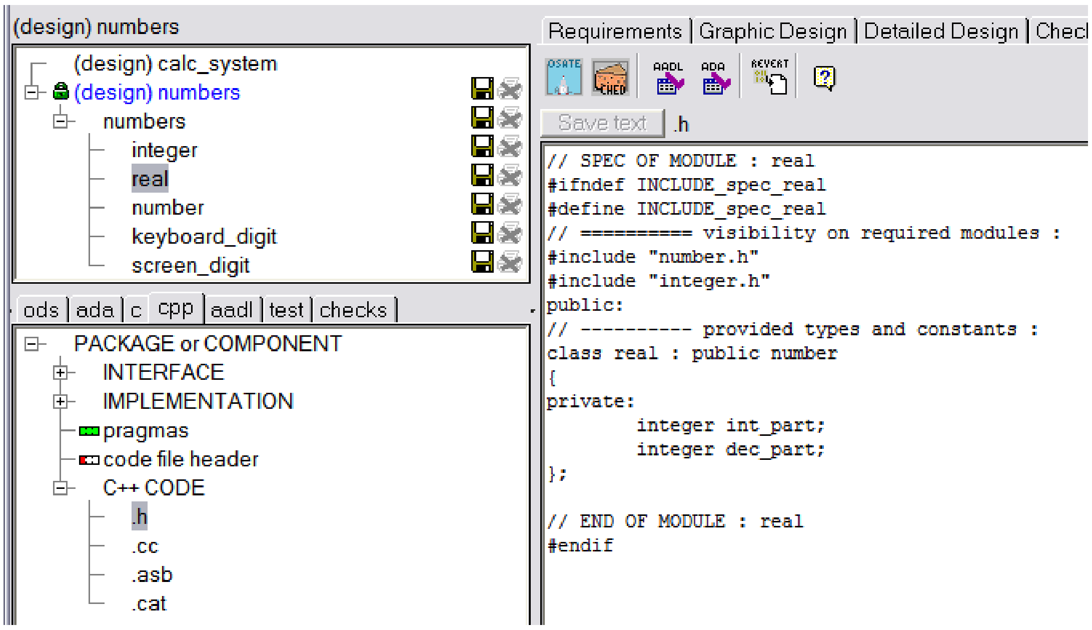
(a) Select the *Code* tab:



(b) Change the source language tab from *aadl* to *cpp*:



(c) Press the *full extraction* button, followed by the *OK* button in the dialog box. The generated files are shown by selecting the appropriate items in the selection list:



```
// SPEC OF MODULE : real
#ifndef INCLUDE_spec_real
#define INCLUDE_spec_real
// ========== visibility on required modules :
#include "number.h"
#include "integer.h"
public:
// ---------- provided types and constants :
class real : public number
{
private:
        integer int_part;
        integer dec_part;
};

// END OF MODULE : real
#endif
```

4.17 Save the design

It is recommended that the design is saved to the design directory regularly. This is done by selecting *Design/Save design*.

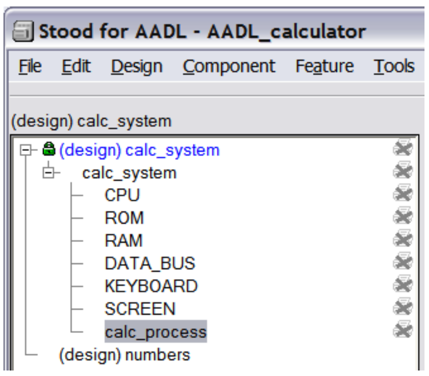
5 Create an AADL process

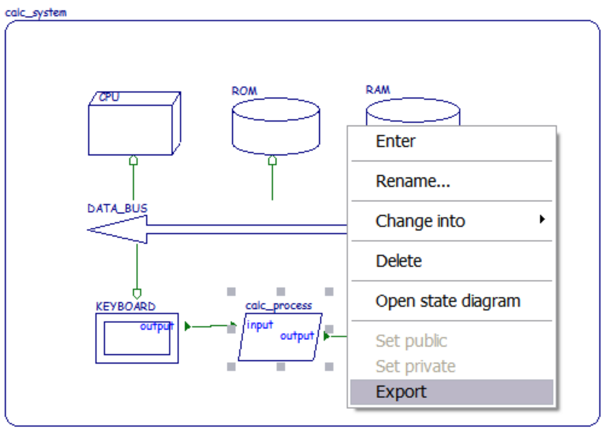
The third kind of **Design** that can be managed with **Stood** is associated with an **AADL Process**. Such a **Design** represents a software program for which you can generate a complete set of target language source files that can be compiled and linked to produce an executable file. To create a **Design** of type **AADL** Process use the *Design/New design* menu. Another option is to create a new **Design** by exporting a **Process** subcomponent defined in an **AADL System**.

5.1 Create a new design by export

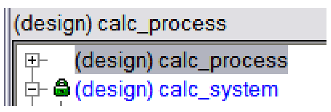
In the **Design** `calc_system`, we have defined a **Process** subcomponent called `calc_process`. It is possible to refine the contents of this **Process** within this **Design**. However, it may be interesting to isolate the pure software part of the **System** in a separate **Design** to complete software design and coding activities until the end.

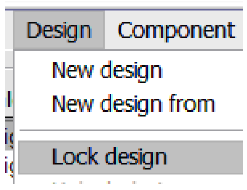
To do so, load the `calc_system` Design, select the `calc_process` component and use the export contextual menu option, as shown below:

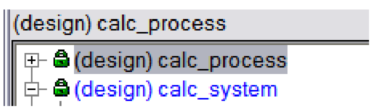




A new **Design** is added to the list of the **Project** `AADL_calculator`. This new **Design** now needs to be selected (loaded) and locked (opened in read-write mode) as shown below.





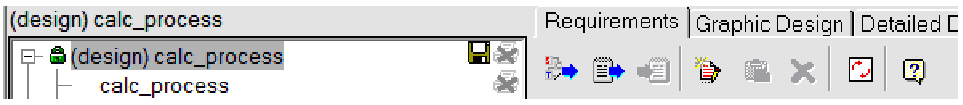


5.2 Import a list of design requirements

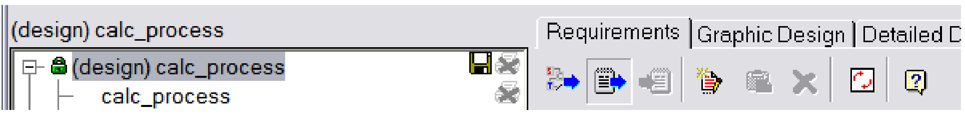
It is possible to import the list of software requirements that must be covered by the current design and coding activities. The simplest way to import such a list of requirements consists of reading a tabulated **ASCII** text file that is produced using any requirements management tool. Such a file must be formatted as follows for import into **Stood**:

- one requirement per line
- two fields separated by a tab character per requirement: the unique requirement ID and a comment.

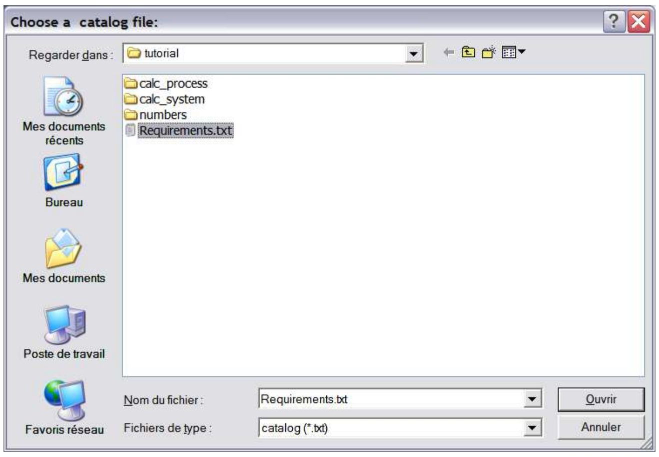
When such a file is available, switch the **Stood** lifecycle tab to *Requirements*:



Then click the button *load requirements from text*:

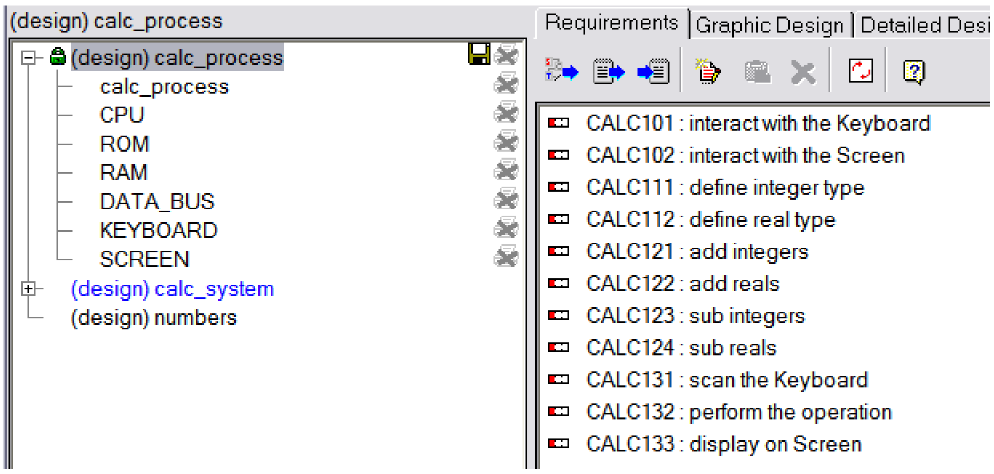


A file navigator dialog is shown so that the appropriate file can be selected. Our **Workspace** contains a file called `Requirements.txt` whose contents were described in chapter 2.3.



(Note that shown items may vary depending on the actual directory contents)

Once the file is loaded, the corresponding list of requirements appears in **Stood**. Note that a small red gauge is shown at the left of each individual item, which means that the requirement is not covered yet by any design entity.

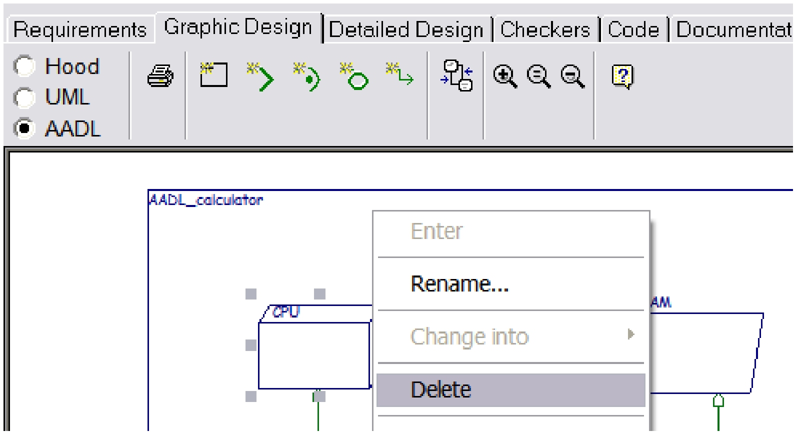


CALC101 : interact with the Keyboard
CALC102 : interact with the Screen
CALC111 : define integer type
CALC112 : define real type
CALC121 : add integers
CALC122 : add reals
CALC123 : sub integers
CALC124 : sub reals
CALC131 : scan the Keyboard
CALC132 : perform the operation
CALC133 : display on Screen

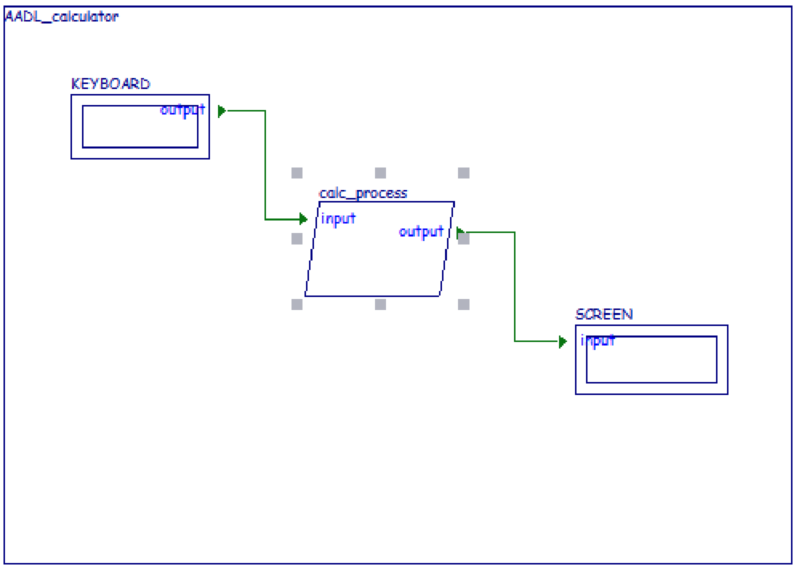
5.3 *Clean up the environment*

The current **Design** has been created by exporting a **Process** subcomponent of an **AADL System** in another **Design**. This export function also propagates information about the environment of the exported component, and in particular, all the sibling subcomponents. However, they are not all relevant in the context of our new **Design**.

To clean this environment up, switch to the *Graphic Design* tab to show the **AADL** diagram, select each component you wish to remove and use the *Delete* contextual menu as shown below:



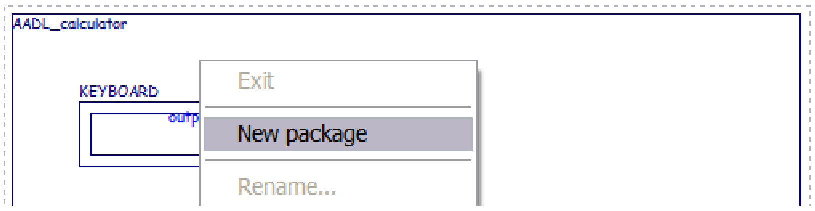
Once all the components that are not required in `calc_process` are deleted, the cleaned model now looks as follows:



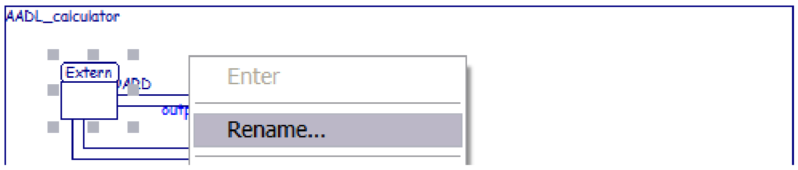
5.4 *Import a package*

In the model we have imported from `calc_system`, we did not specify the data types associated with the **Data Ports**. We now need to reference the **Data** component classifiers that were defined in the `numbers` **Package** to give a type to the **Data Ports**. The easiest way to do this in **Stood** requires the **Package** to be imported within the **Design** scope.

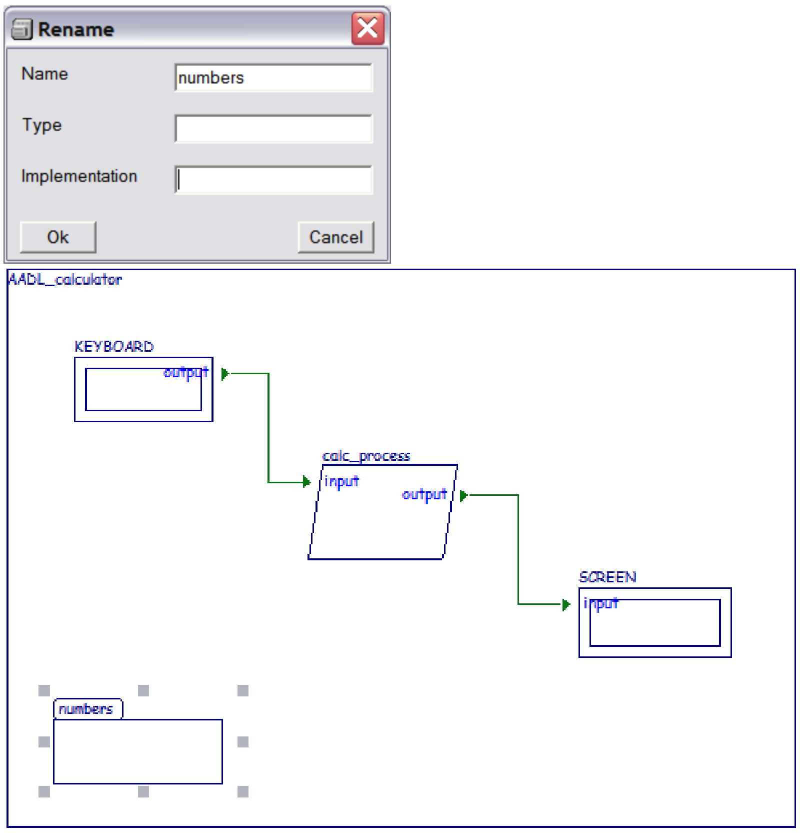
In the **AADL** diagram, with the outer box selected (`aadl_calculator`), use the *New package* contextual menu to create a local representation of a remote **Package**:



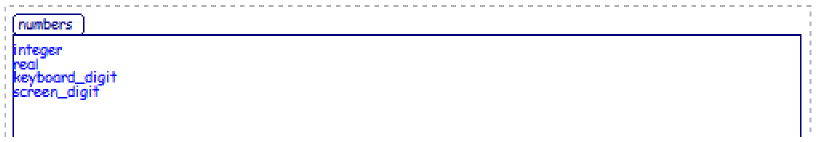
A new box is added to the diagram, with the default name `Extern`. To change this name, use the *Rename* contextual menu (the `Extern` **Package** must be selected).



Now give this local **Package** the name of the actual **Package** we created in the **Project**, i.e. `numbers`.



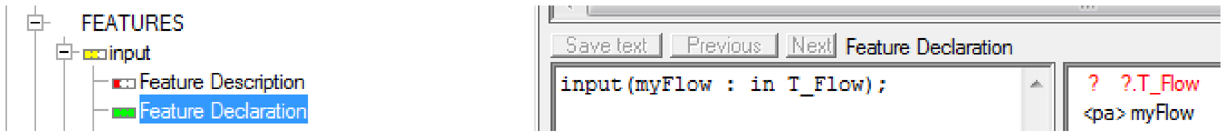
Stood has now made the association between our local **Package** and the actual remote **Design** of the same name. This will allow us to use the various **Data** component classifiers defined in the **Package** `numbers` in the current **Design**. To check the available **Data** component classifiers, double-click on the local **Package** numbers:



Note that if changes are made to the remote **Package**, they will be propagated to the local copy only during a **Design** load.

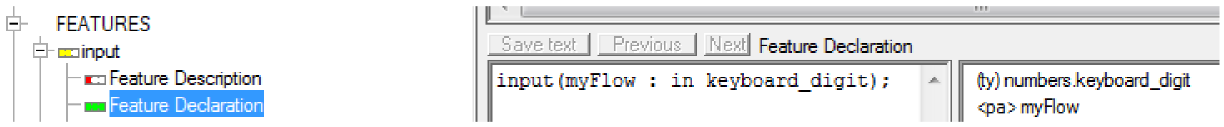
5.5 *Change data ports type*

When we added **Data Ports** to subcomponents in section 3.8, we did not care about the associated data type. A default data type `T_Flow` was used. It is now possible to reference the **Data** component classifiers provided by the imported **Package** to specify the actual type of the **Data Ports**. To illustrate how a port definition can be modified, select the *Feature declaration* section for port `input` in component `calc_process`.



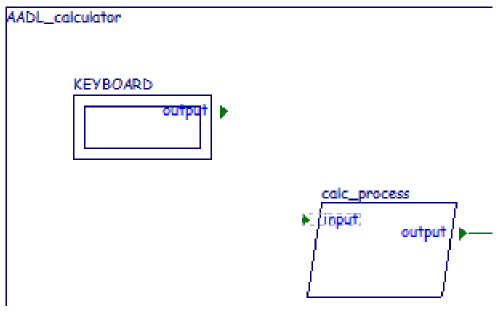
```
input(myFlow : in T_Flow);
```

The right hand list, called *symbol table*, shows that type `T_Flow` is unknown. Let us modify the port declaration to use type `keyboard_digit` instead. Do not forget to save the changes using the button, the contextual menu or the keyboard shortcut. The new port declaration will look as follows:

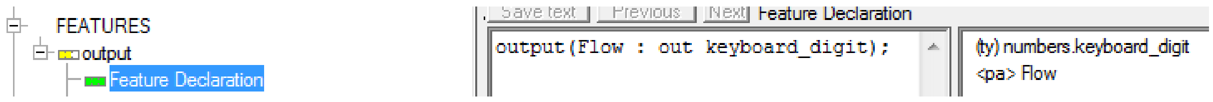


```
input(myFlow : in keyboard_digit);
```

Note that the data type is now recognized in the *symbol table*. However, the **Data Port** connection between **Device** `KEYBOARD` and **Process** `calc_process` has suddenly disappeared from the diagram.



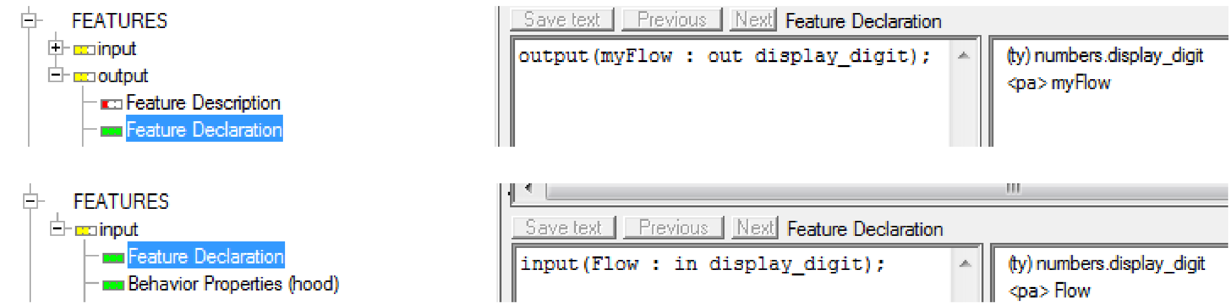
The reason is that the two ends of the connection are no longer type compatible. The declaration of port `output` in component `KEYBOARD` must also be changed in the same way:



```
output(Flow : out keyboard_digit);
```

Note that the **Data Port** connection becomes visible again in the diagram as soon as both ends become type compatible again.

We can now do similar changes to port `output` of component `calc_process` and port `input` of component `SCREEN`. The data type these two ports should reference is screen_digit.

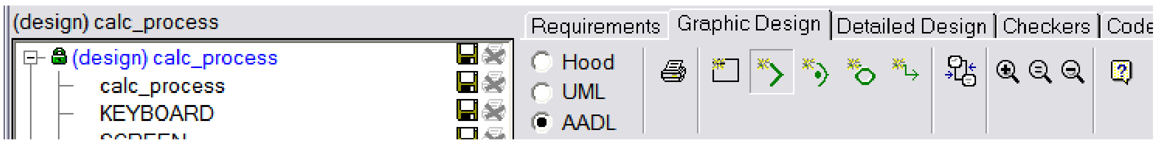


```
output(myFlow : out display_digit);
```

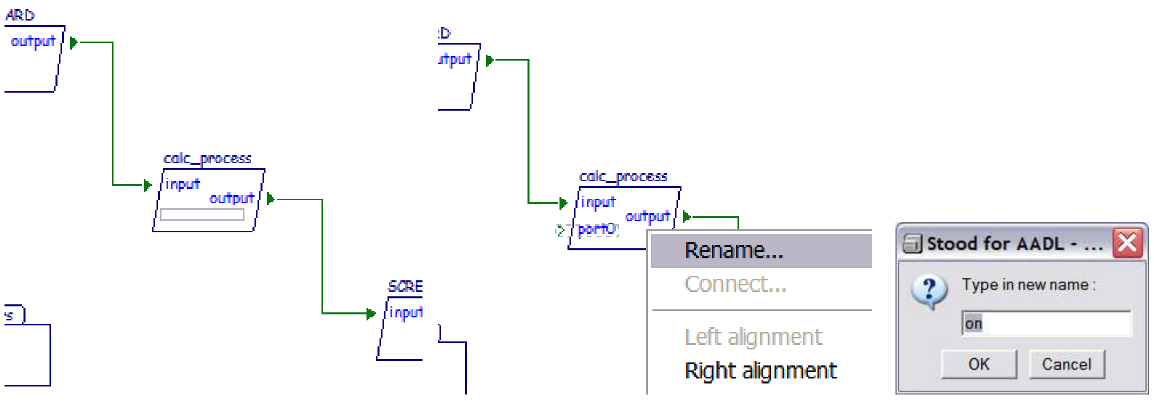
```
input(Flow : in display_digit);
```

5.6 *Add ports to the process*

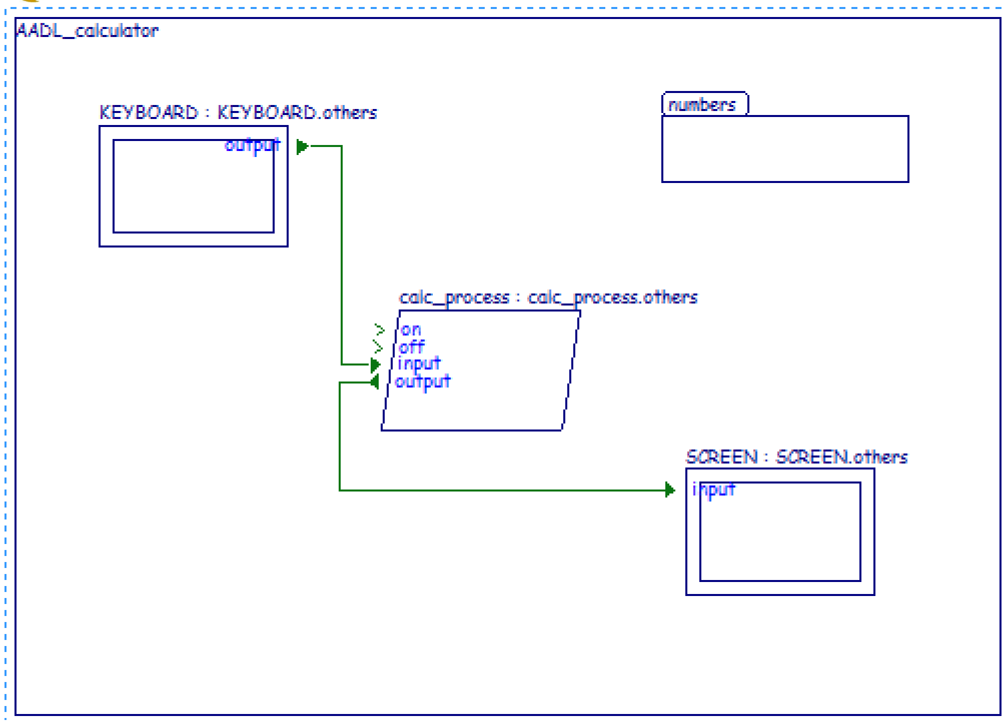
Currently, the **Process** only shows **Data Ports** in its interface. **Data Ports** can be used to describe data flows between components. We are now going to add **Event Ports** to specify control flow entry points for the **Process**.



To create a new **Event Port**, use the *new port* button of the button bar in the **AADL** diagram editor then click inside the `calc_process` box. The default name `port0` can be changed using the *Rename* contextual menu, as shown below:

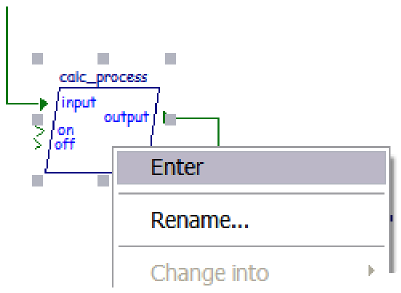


After adding a second **Event Port** called `off`, the diagram will look as follows:

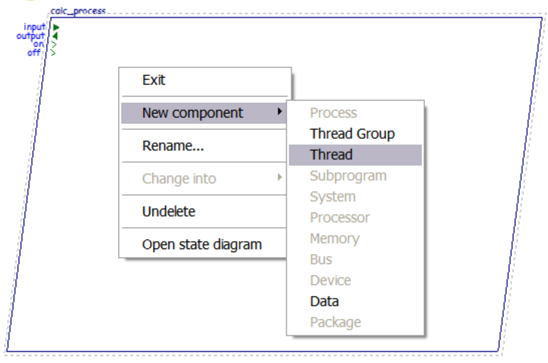


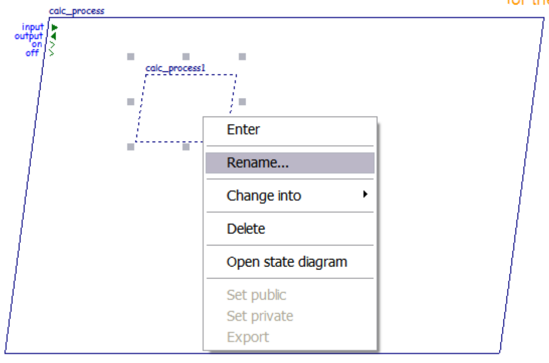
5.7 *Add subcomponents to the process*

We now need to provide some details about the internals of our **Process**. The current graphical representation of `calc_process` only shows its interfaces. It is its *black box view*. In order to edit its internal details, we must enter the component to show its *white box view*. The *Enter* contextual menu, or a double-click, is used to do this:

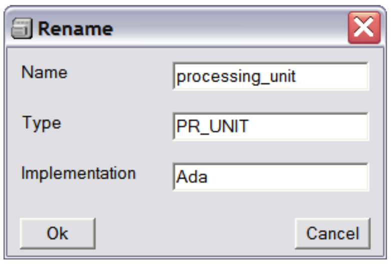


According to the **AADL** rules, a **Process** component implementation can contain **Thread Group**, **Thread** or **Data** subcomponents. To create a subcomponent, use the *new AADL component* button of the button bar in the AADL diagram editor, or the *New component* contextual menu. The newly created component is renamed using the *Rename* contextual menu as shown below:

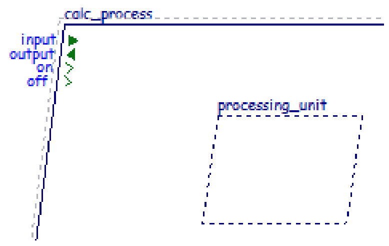




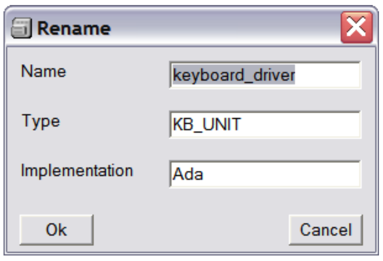
As already explained in section 3.6, the *Rename* dialog box is also used to specify the **AADL** component type and component implementation of the subcomponents.



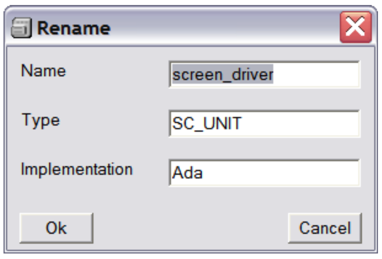
Name	processing_unit
Type	PR_UNIT
Implementation	Ada



Our model can now be enriched by two other **Thread** subcomponents representing local **Device** interface software.

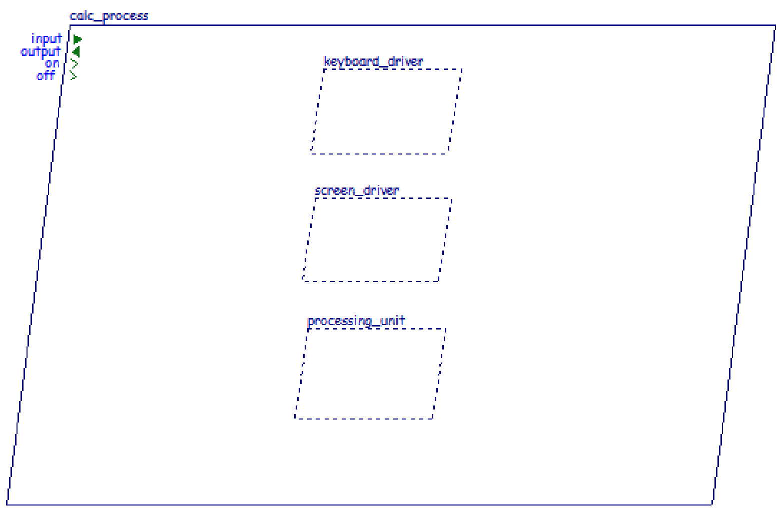


Name	keyboard_driver
Type	KB_UNIT
Implementation	Ada



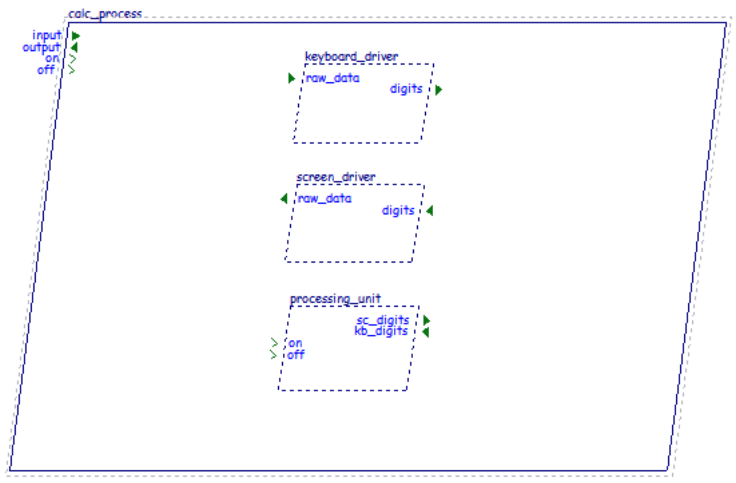
Name	screen_driver
Type	SC_UNIT
Implementation	Ada

The graphical representation of the internals of our **Process** as shown in the diagram below:

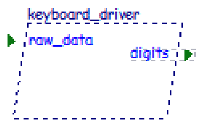


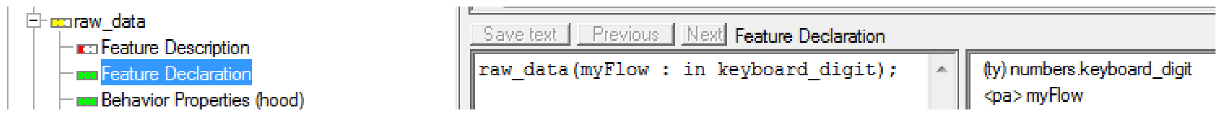
5.8 *Create and customize ports in subcomponents*

Following a top-down modelling process, we must now specify the interface of each subcomponent. Let us add ports as shown below:



The details for new port declarations are given below for each subcomponent:



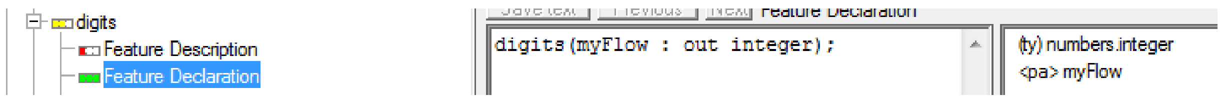


- raw_data
  - Feature Description
  - Feature Declaration
  - Behavior Properties (hood)

Feature Declaration

```
raw_data(myFlow : in keyboard_digit);
```

(ty) numbers.keyboard_digit
<pa> myFlow

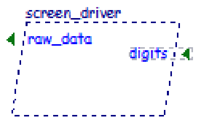


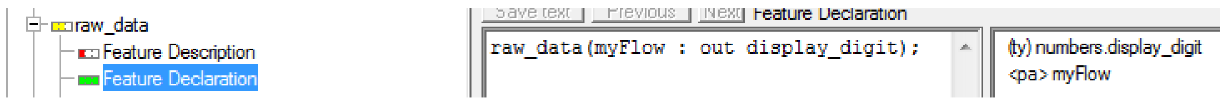
- digits
  - Feature Description
  - Feature Declaration

Feature Declaration

```
digits(myFlow : out integer);
```

(ty) numbers.integer
<pa> myFlow



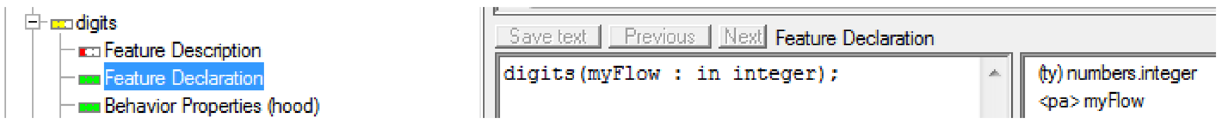


- raw_data
  - Feature Description
  - Feature Declaration

Feature Declaration

```
raw_data(myFlow : out display_digit);
```

(ty) numbers.display_digit
<pa> myFlow

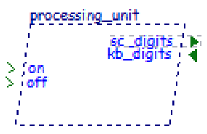


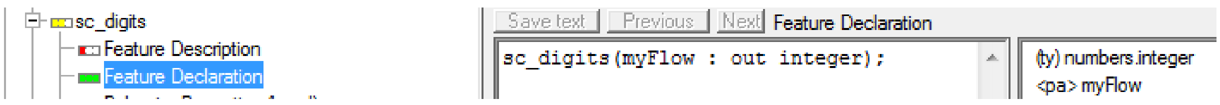
- digits
  - Feature Description
  - Feature Declaration
  - Behavior Properties (hood)

Feature Declaration

```
digits(myFlow : in integer);
```

(ty) numbers.integer
<pa> myFlow



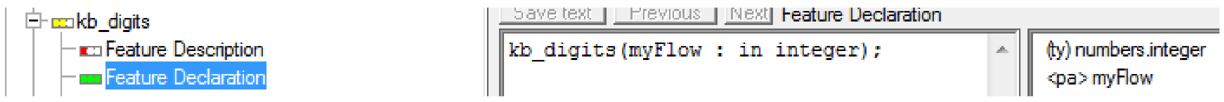


- sc_digits
  - Feature Description
  - Feature Declaration

Feature Declaration

```
sc_digits(myFlow : out integer);
```

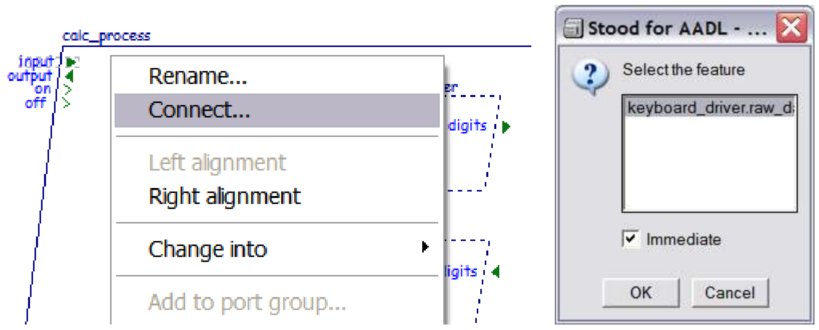
(ty) numbers.integer
<pa> myFlow



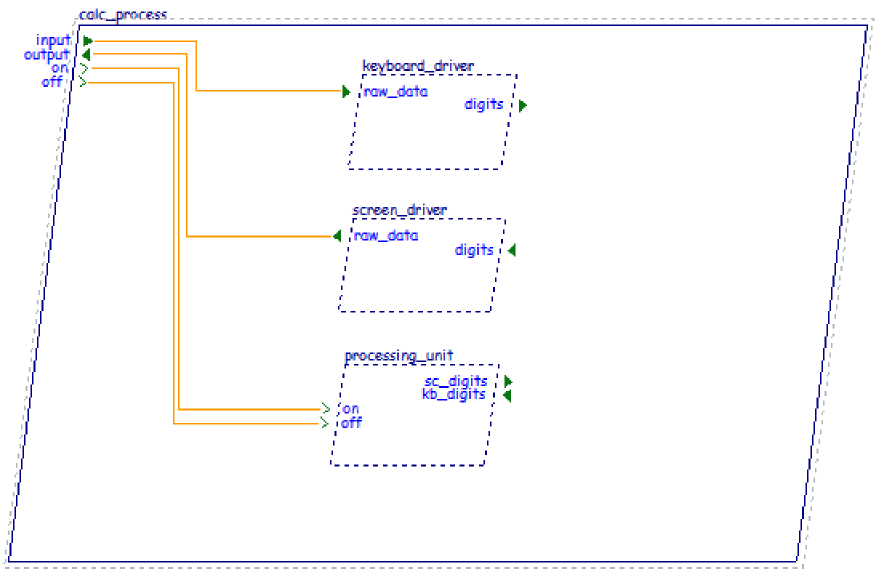
```
kb_digits(myFlow : in integer);
```

5.9 Connect ports between a component and its subcomponents

Connections can now be established between the interface of outer component and the interface of its inner subcomponents. To create such connections, use the *new connection* button in the button bar of the **AADL** diagram editor and click, in sequence, the two ports to be connected. Another way to do this is to select a port in the outer component interfaces and use the *Connect* contextual menu option.

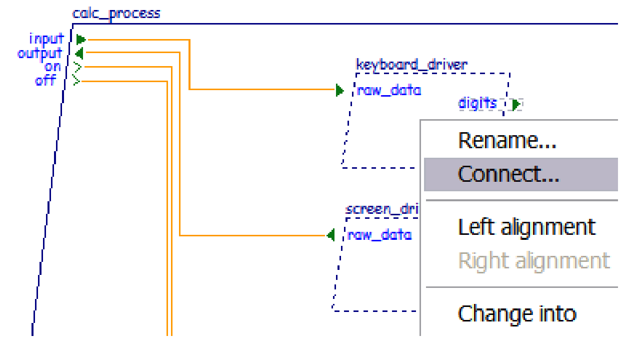


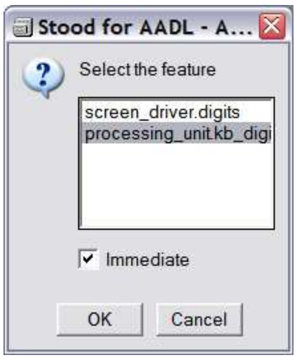
Note that only the direction and type compatible ports are given in the dialog box. The new diagram will look as follows after connecting the four ports of the **Process** interface is:



5.10 Connect ports between subcomponents

To connect ports between subcomponents of a same component, use the *new connection* button in the button bar of the **AADL** diagram editor and click, in sequence, the two ports to be connected. Another way to do this is to select a port in one of the subcomponent interfaces and use the *Connect* contextual menu option.



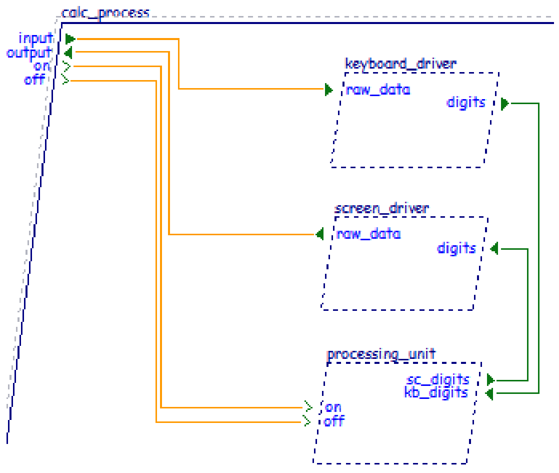


Select the feature

screen_driver.digits
processing_unit.kb_digi

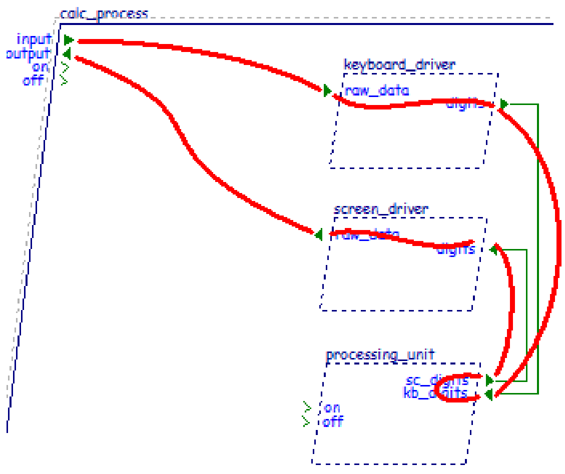
☑ Immediate

Note that only the direction and type compatible ports are given in the dialog box. The new diagram will look as follows after connecting all the remaining subcomponents:



5.11 Specify flows

AADL connections represent point to point interaction between two ports of the same type and having compatible directions. Even if they do not refer to the same data type, several connections may participate in the same more global data flow. Stood has a particular way to express such flow specifications.

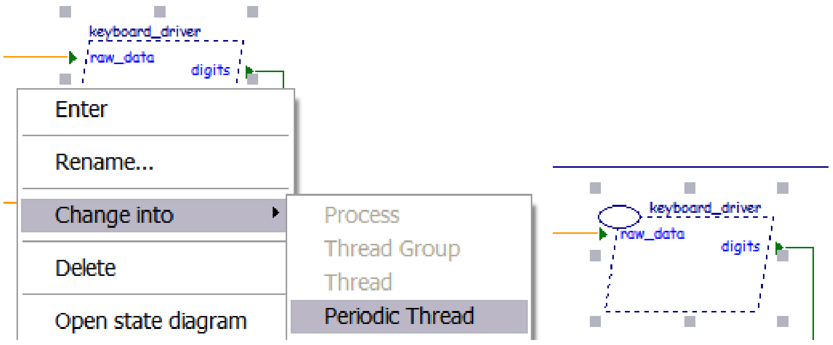


For each port involved in a given flow, the flow name must be inserted into the *Feature declaration* section to replace the default name `Flow`. In section 5.8, we already changed these names into `myFlow`, so that the **AADL** code generator can produce the corresponding flow specification.

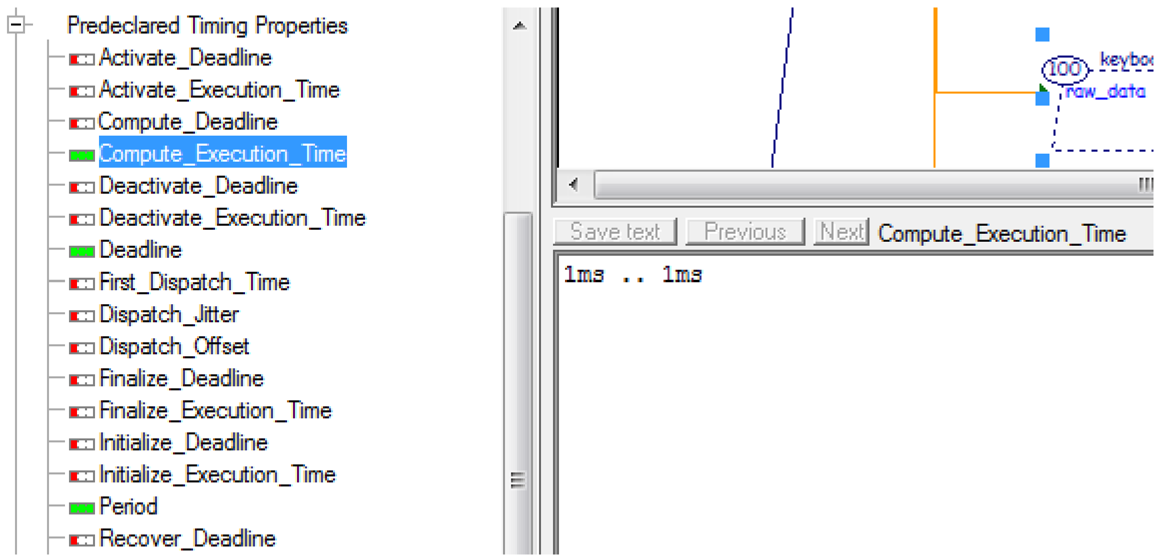
```
THREAD KB_UNIT
FEATURES
  raw_data : IN DATA PORT numbers::digit.keyboard;
  digits : OUT DATA PORT numbers::integer;
FLOWS
  myFlow_0 : FLOW PATH raw_data -> digits;
END KB_UNIT;
```

5.12 Specify real-time properties

Threads that have been created are *aperiodic*. When **Threads** are *periodic* or *sporadic*, more details about their real time behaviour can be managed by the tool. This subcomponent sub-category can be modified using the contextual menu *Change into*. To illustrate this feature, let us change the two drivers from *aperiodic* **Threads** into *periodic* **Threads**:



Note that the graphical notation has changed to comply with the **AADL** rules. Real-time properties must be entered into the model as standard **AADL** Properties. We can specify for instance that `keyboard_driver` has a *period* of `100 ms` a *deadline* of `50 ms` and a *compute_execution_time* of `1ms..1ms`:



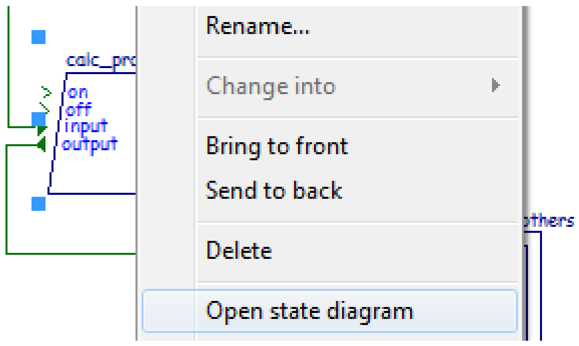
```
1ms .. 1ms
```

Similar modifications can now be done for the **Thread** `screen_driver` to change it into a periodic **Thread** and to add appropriate real-time **Properties**. This information is included by the code generator at the most appropriate location in the **AADL** source text:

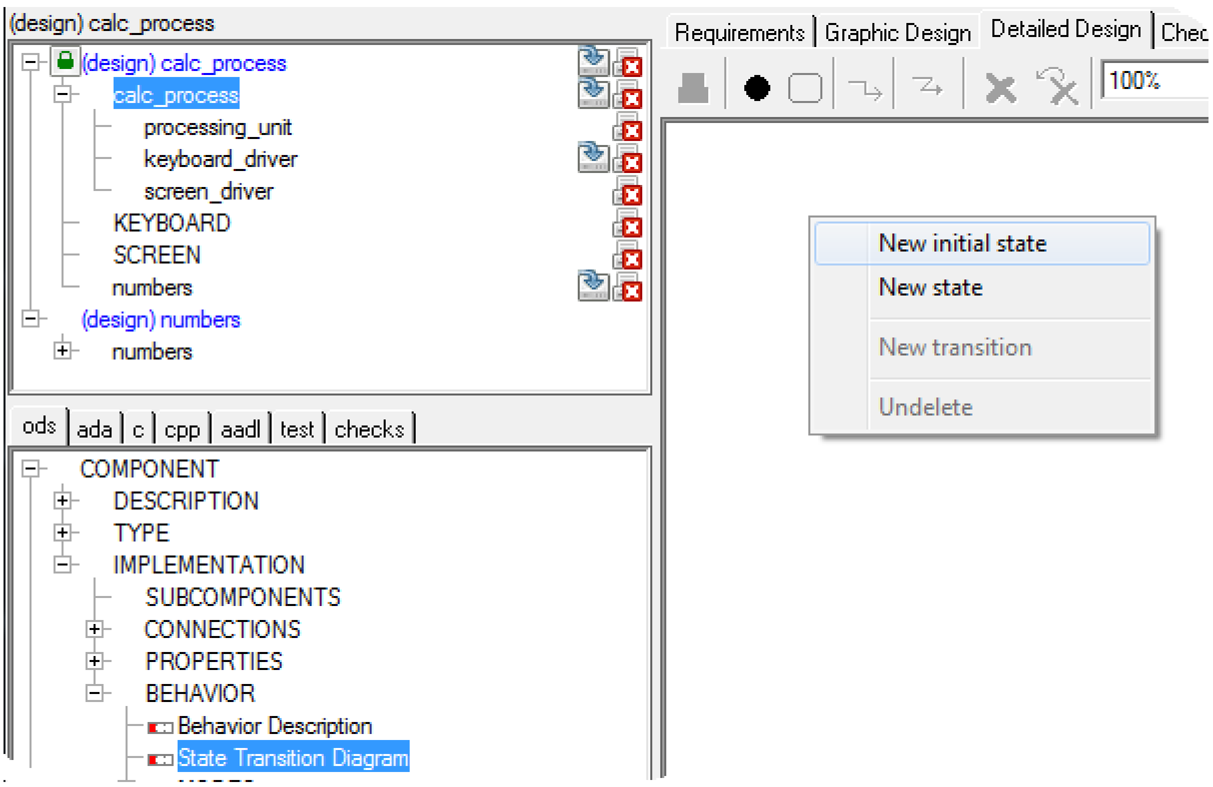
```
THREAD IMPLEMENTATION KB_UNIT.Ada
PROPERTIES
  Dispatch_Protocol => Periodic;
  Compute_Execution_Time => 1ms .. 1ms;
  Deadline => 50 ms;
  Period => 100 ms;
END KB_UNIT.Ada;
```

5.13 Specify modes

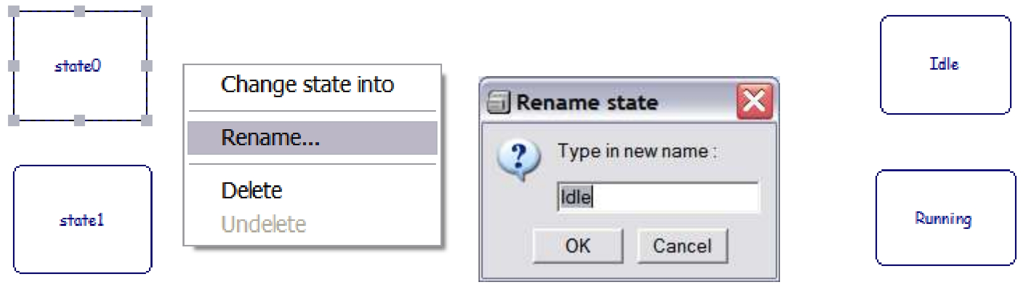
AADL Modes can be defined to represent the operational states of a component, using a state diagram editor to specify them. We are going to illustrate this on the **Process** `calc_Process`. The state diagram editor is launched using the *Open state diagram* contextual menu:



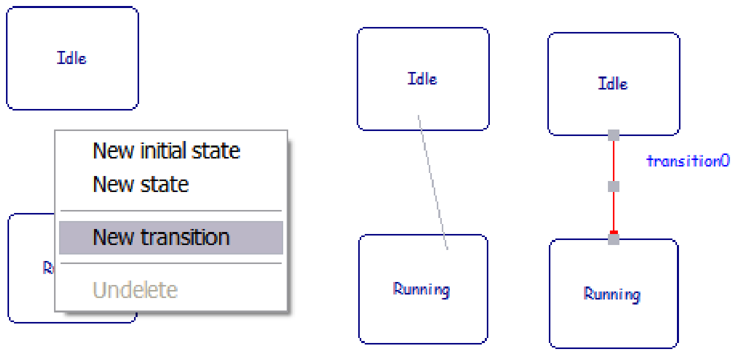
A new graphical editing area is representing **AADL Modes** and transitions representing **AADL Mode Transitions**. To create a new Mode, use the buttons or contextual menu:



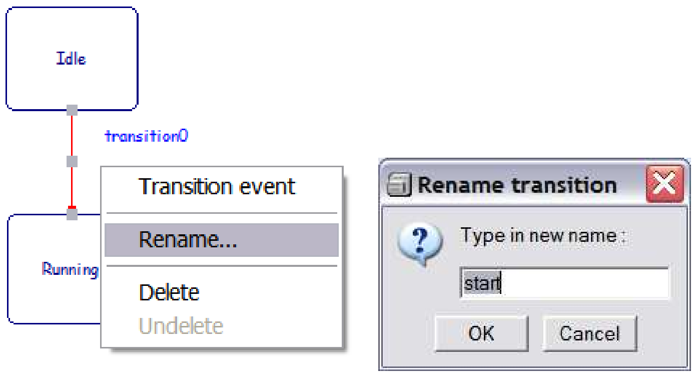
States are given a default name that can be modified using the *Rename* contextual menu. Let us create two **Modes**: `idle` and `running`:



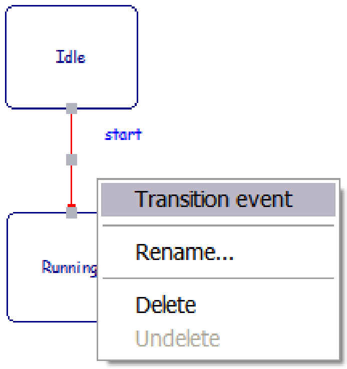
Transitions can be created using the *new transition* button or contextual menu. First click the origin **Mode** and then the destination **Mode**:

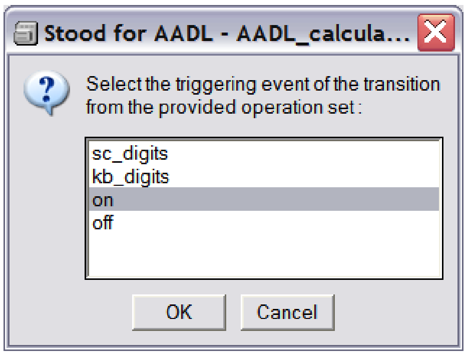


Transitions are given a default name that can be changed using the *Rename* contextual menu:



Transitions must also be attached to a triggering **Event**. Selection of the **Event** is done using the *select transition event* button or the *Transition event* contextual menu. This action opens a dialog box showing *all* the **Ports** and **Subprograms** defined in the interface of the current component. Note that only **In** and **In Out Event Ports** can be used to trigger **Mode Transitions**.

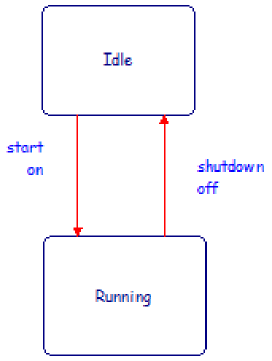




Select the triggering event of the transition from the provided operation set :

sc_digits
kb_digits
on
off

The complete graphical **Modes** definitions as well as the corresponding generated **AADL** source text fragment are shown below:

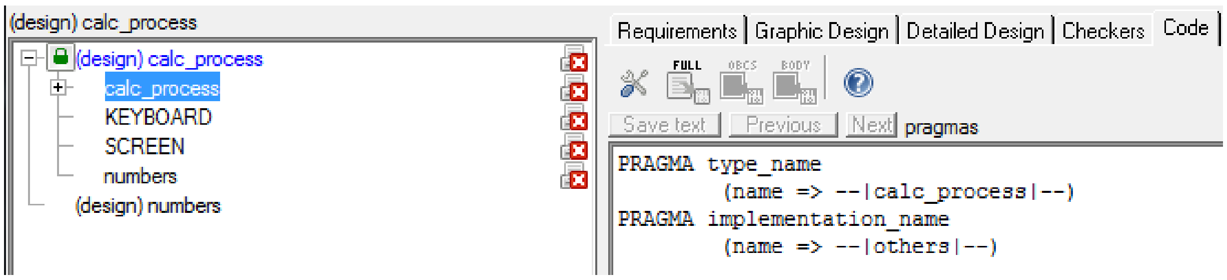


```
MODES
  Idle : MODE;
  Running : MODE;
  start : Idle -[ on ]-> Running;
  shutdown : Running -[ off ]-> Idle;
END calc_process.others;
```

Note that using the *state diagram editor* with **AADL Threads** or **Subprograms** generates a **Behavior Annex** subclause instead of a **Modes** subclause.

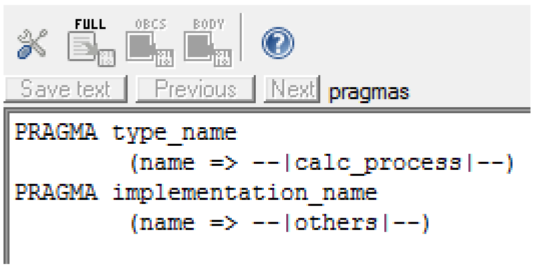
5.14 Generate the AADL code for the process

To generate textual **AADL** code from a graphical model in **Stood**, the *Code* tab is selected. The new button bar shows two buttons. The first button on the left is called *add pragma* and may be used to customize the code generation. Pressing this button opens a dialog box showing the list of possible pragmas that can be selected. Note that some pragmas may have already been automatically inserted by **Stood**, as it is the case here.

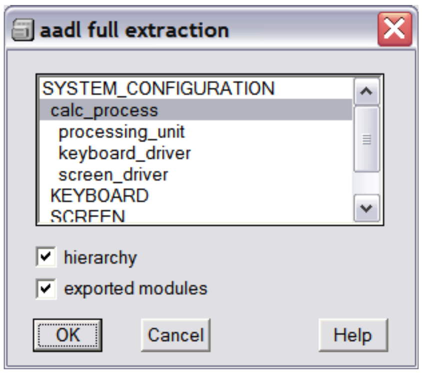


```
PRAGMA type_name
        (name => --|calc_process|--)
PRAGMA implementation_name
        (name => --|others|--)
```

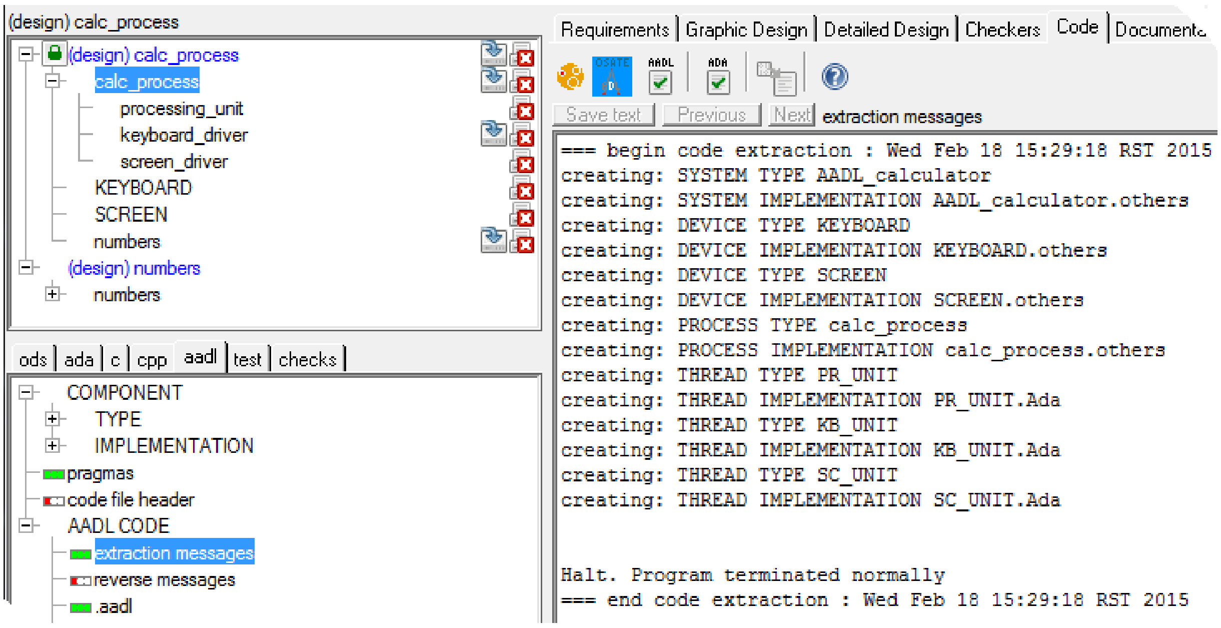
The **AADL** code generation is started by pressing the *full extraction* button. This opens a dialog box which is used to specify the part of the **Design** to be generated. Mostly, we need the whole **Design** to be generated, which is the default.



```
PRAGMA type_name
        (name => --|calc_process|--)
PRAGMA implementation_name
        (name => --|others|--)
```



When the *Ok* button is pressed, the *extraction messages* are displayed and the AADL Inspector is opened in a separate window to view the code generated. The messages file lists the abstract component types and implementations that were created to fully describe our **Process**. Note that the context of the **Process** is also generated as a **System** having the same name as the current **Project** and which also contains the two **Device** components.

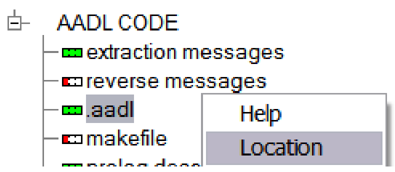


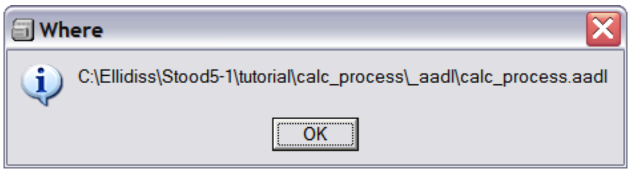
```
=== begin code extraction : Wed Feb 18 15:29:18 RST 2015
creating: SYSTEM TYPE AADL_calculator
creating: SYSTEM IMPLEMENTATION AADL_calculator.others
creating: DEVICE TYPE KEYBOARD
creating: DEVICE IMPLEMENTATION KEYBOARD.others
creating: DEVICE TYPE SCREEN
creating: DEVICE IMPLEMENTATION SCREEN.others
creating: PROCESS TYPE calc_process
creating: PROCESS IMPLEMENTATION calc_process.others
creating: THREAD TYPE PR_UNIT
creating: THREAD IMPLEMENTATION PR_UNIT.Ada
creating: THREAD TYPE KB_UNIT
creating: THREAD IMPLEMENTATION KB_UNIT.Ada
creating: THREAD TYPE SC_UNIT
creating: THREAD IMPLEMENTATION SC_UNIT.Ada


Halt. Program terminated normally
=== end code extraction : Wed Feb 18 15:29:18 RST 2015
```

5.15 *Show generated AADL code*

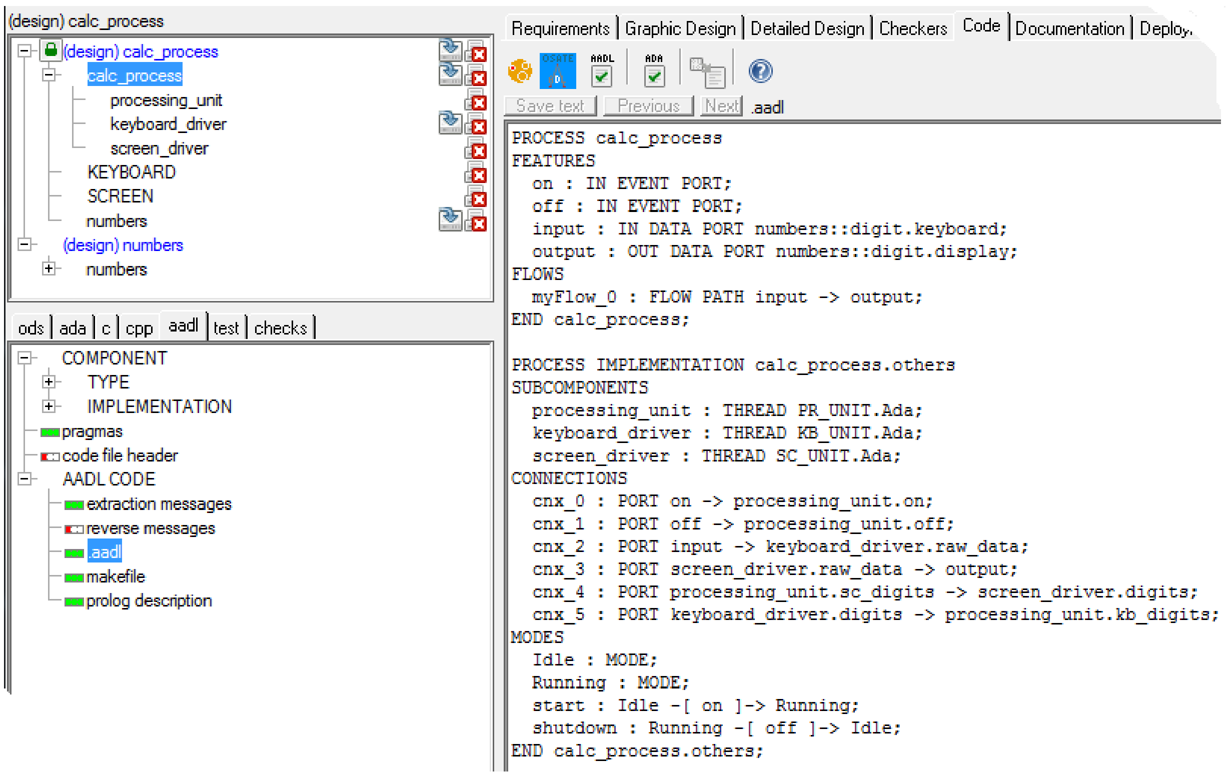
The **AADL** generated code can be shown by changing the selection in the lower left list of the **Stood** window from *extraction messages* to *aadl*.

The **AADL** code can be edited with **Stood**, although the corresponding file in the repository is easily located for remote access. To locate a particular file, select the corresponding entry in the lower left list and use the *Location* contextual menu.




Where
C:\Ellidiss\Stood5-1\tutorial\calc_process_aadl\calc_process.aadl
OK

The screenshot below shows the result of an **AADL** code generation that only uses the information that has been inserted during the previous modelling steps:



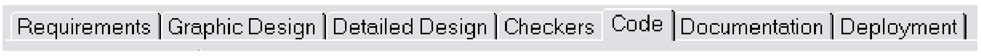
```
PROCESS calc_process
FEATURES
  on : IN EVENT PORT;
  off : IN EVENT PORT;
  input : IN DATA PORT numbers::digit.keyboard;
  output : OUT DATA PORT numbers::digit.display;
FLOWS
  myFlow_0 : FLOW PATH input -> output;
END calc_process;

PROCESS IMPLEMENTATION calc_process.others
SUBCOMPONENTS
  processing_unit : THREAD PR_UNIT.Ada;
  keyboard_driver : THREAD KB_UNIT.Ada;
  screen_driver : THREAD SC_UNIT.Ada;
CONNECTIONS
  cnx_0 : PORT on -> processing_unit.on;
  cnx_1 : PORT off -> processing_unit.off;
  cnx_2 : PORT input -> keyboard_driver.raw_data;
  cnx_3 : PORT screen_driver.raw_data -> output;
  cnx_4 : PORT processing_unit.sc_digits -> screen_driver.digits;
  cnx_5 : PORT keyboard_driver.digits -> processing_unit.kb_digits;
MODES
  Idle : MODE;
  Running : MODE;
  start : Idle -[ on ]-> Running;
  shutdown : Running -[ off ]-> Idle;
END calc_process.others;
```

Note that this view has new buttons that can be used to activate **AADL** compliant tools such as **AADL Inspector** or **Osate.** To user the later, the environment variable `OSATE_PATH` must have been properly configured in the initialisation file (`stood.ini` or `.stoodrc`).

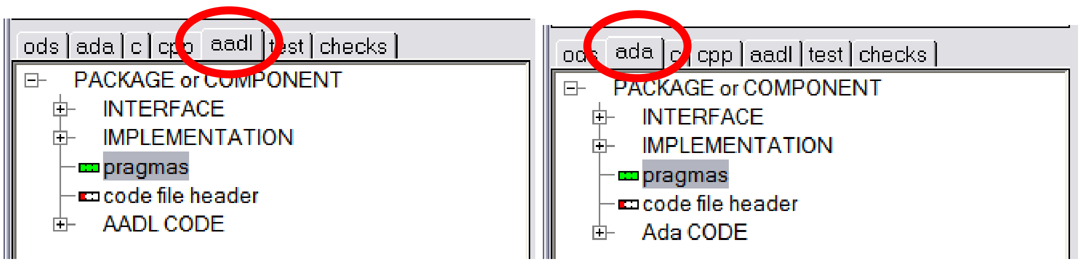
5.16 *Generate Ada code*

Without changing anything in the design model, you can generate **Ada**, **C** or **C++** source code. The process for generating **Ada** code (for example) is very similar to generating the **AADL** code, i.e.:

(a) Select the *Code* tab again:

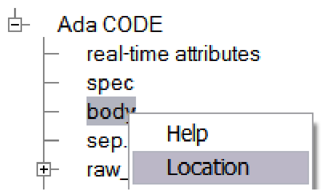

Requirements | Graphic Design | Detailed Design | Checkers | Code | Documentation | Deployment

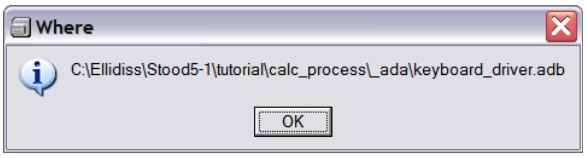
(b) Change the source language tab from *aadl* to *Ada*:



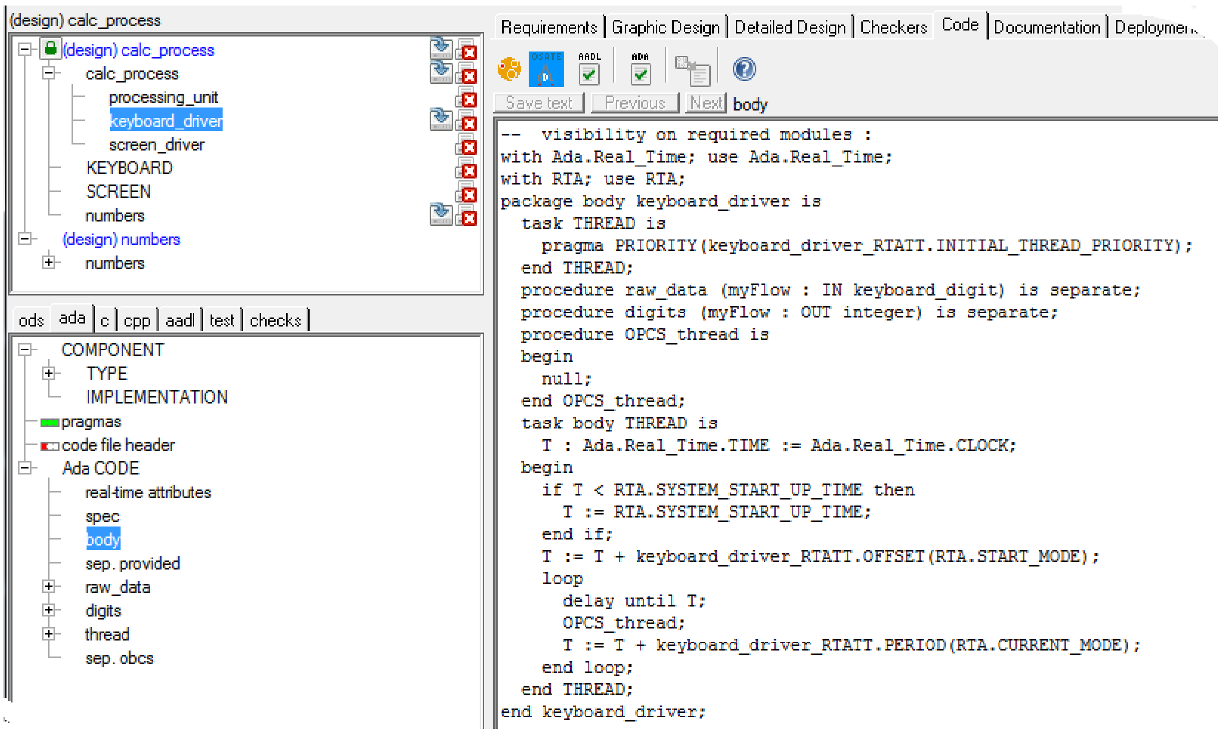
(c) Press the *full extraction* button, followed by the *OK* button in the dialog box. The generated files are shown by selecting the appropriate items in the selection list:

Note that the generated **Ada** source files are stored in a default location in the **Workspace**. The location of each file is found using the *Location* contextual menu:





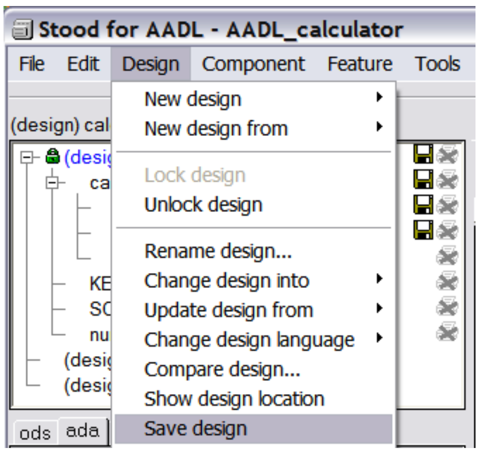
C:\Ellidiss\Stood5-1\tutorial\calc_process_ada\keyboard_driver.adb



```
-- visibility on required modules :
with Ada.Real_Time; use Ada.Real_Time;
with RTA; use RTA;
package body keyboard_driver is
  task THREAD is
    pragma PRIORITY(keyboard_driver_RTATT.INITIAL_THREAD_PRIORITY);
  end THREAD;
  procedure raw_data (myFlow : IN keyboard_digit) is separate;
  procedure digits (myFlow : OUT integer) is separate;
  procedure OPCS_thread is
  begin
    null;
  end OPCS_thread;
  task body THREAD is
    T : Ada.Real_Time.TIME := Ada.Real_Time.CLOCK;
  begin
    if T < RTA.SYSTEM_START_UP_TIME then
      T := RTA.SYSTEM_START_UP_TIME;
    end if;
    T := T + keyboard_driver_RTATT.OFFSET(RTA.START_MODE);
    loop
      delay until T;
      OPCS_thread;
      T := T + keyboard_driver_RTATT.PERIOD(RTA.CURRENT_MODE);
    end loop;
  end THREAD;
end keyboard_driver;
```

5.17 Save the design

It is recommended that the design is saved to the design directory regularly. This is done by selecting *Design/Save design*.



6 Conclusion

This tutorial does not provide information about all the possible modelling and model processing features of **Stood**. In particular, this first version of the document does not give explanations about the following important topics that are nevertheless already supported by **Stood**:

- Create a new **Design** from a remote **AADL** textual specification.
- Update an existing **Design** from a remote **AADL** textual specification.
- Create an **AADL** model from legacy **Ada** or **C** source files.
- Use the integrated **Design** verification tools.
- Perform software to hardware binding.

Moreover, the **AADL Inspector** companion tool may be included with your distribution of **Stood**. In that case, this tool is automatically opened after the **AADL** code has been generated and gives the opportunity to perform advanced model verifications such as scheduling analysis and virtual execution at the **AADL** level. Please refer to the **AADL Inspector** user manual for further details.

Ellidiss Technologies

www.ellidiss.com

Sales office:
TNI Europe Limited
Triad House
Mountbatten Court
Worall Street
Congleton
Cheshire
CW12 1AG
UK
info@ellidiss.com
+44 1260 291 449

Technical support :
Ellidiss Technologies
24 quai de la douane
29200 Brest
Brittany
France

aadl@ellidiss.fr
+33 298 451 870