



USER MANUAL

Contents

Project manager - - - - -	7
Project	7
Files and directories	8
Packages and folders	8
Supported file types	9
Rearranging the project tree	11
Adding components to the system	12
Adding a single component	12
Importing a directory	13
Sharing project parts	14
Interface with traceability tools	15
Importing a PR/CIF file	16
Source & destination panel	17
Basic options panel	18
Advanced options panels	20
Summary panel	22
PR/CIF import progress and result	22
Importing a MSC-PR file	22
Exporting the project as a SDL/PR file	23
Exporting a project as a Verimag IF file	25
Exporting elements as HTML files	26
Exporting a single element	26
Exporting the whole project	26
Export all the publications in a whole project	27
Preferences	27
Project manager preferences	27
Diagram preferences	28
Text editor preferences	32
Debugger preferences	33
General preferences	33
PR import & export preferences	34
Advanced options	35
User defined external tools	35
Tool menus definition	36
Tool commands	37
Hooks addition and removal	39
Traceability information	40
Scope	40
Traceability editor	40
Integration with Reqtify	40
Editor windows - - - - -	52
Tab management	52
Windows menu	53
Diagram editor- - - - -	55
Common features	55

Editor modes	55
Frame concept	56
Symbol and link properties	56
Moving symbols	58
Modifying links	58
Button and tool bars	59
Partitions	60
Page setup	61
Publications	62
SDL editor features	67
Creating and opening components	67
Automatic insertion	68
Automatic transition selection in legacy mode	70
"View" / "Go to" menu and state / message browser	72
State and connector usage	72
Diagram diff	73
MSC editor	77
Specific tools	77
Manipulating components in lifelines	77
Message parameters display	79
Compare diagrams	81
Filtering	88
MSC PR import	89
MSC PR export	91
UML diagrams	92
Symbol properties	92
Link properties	96
Access to generated C++ files	96
Text Editor - - - - -	97
MSC generation from TTCN-3 source file.	97
SDL generation for comments in a C source file.	98
Document editor - - - - -	99
General presentation	99
Full documentation generation	102
Documentation display options	104
Documentation export options	106
Styled text editor	111
Table editor	112
Exporting document	113
Exporting as RTF	115
Exporting as OpenDocument format	115
Exporting as HTML	115
Using exported documents	117
In Microsoft Word via RTF	117
In OpenOffice.org via OpenDocument format	118
Questions and answers	118
Requirements Table Editor - - - - -	120

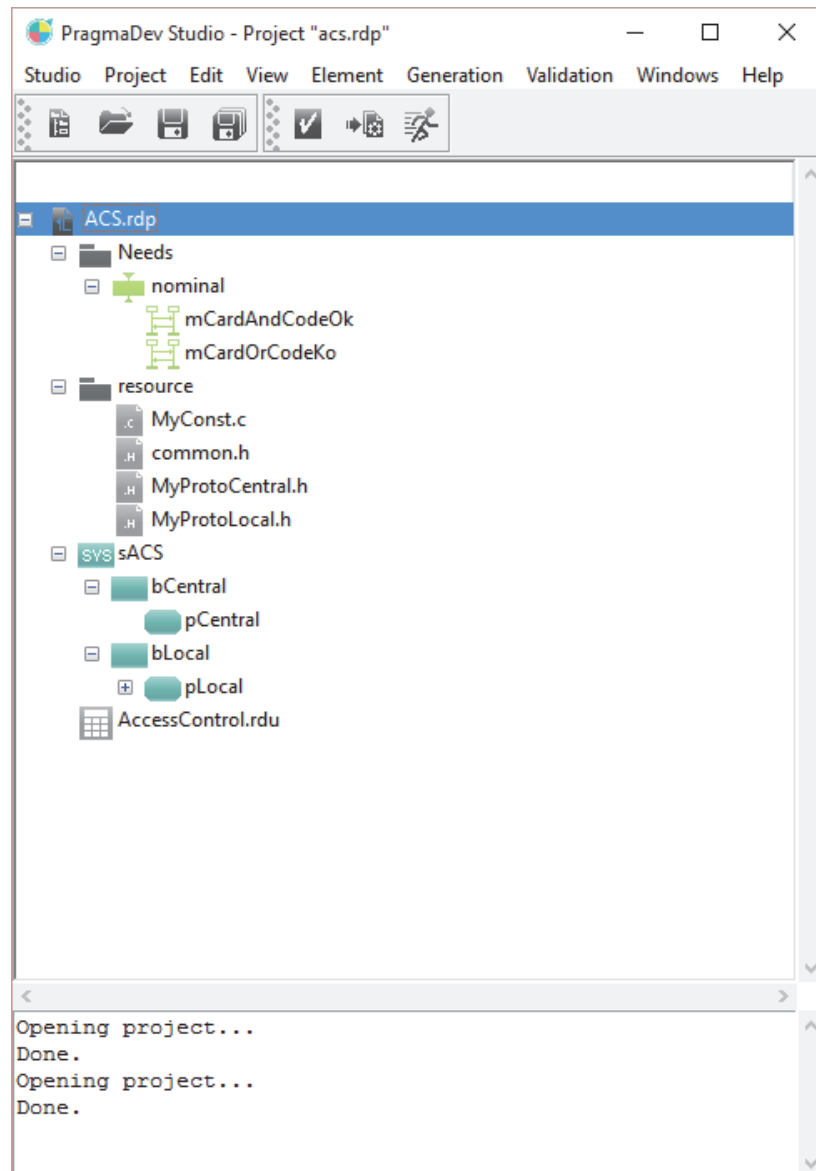
Principles	120
Covering symbols	123
Covering testcases	126
Co-simulation code coverage analysis	127
Importing testcase coverage information in requirements	130
Prototyping GUI - - - - -	132
Prototyping GUI editor	132
Prototyping GUI runner	139
SDL-RT project - - - - -	140
Data and SDL types declarations	140
C types declarations	140
SDL messages and message lists declaration	140
SDL timer declaration	140
Semaphore declaration	141
Process declaration	141
Procedure declaration	142
Class description	142
SDL-RT symbols syntax	143
Task block	143
Next state	143
Continuous signals	144
Message input	144
Message output	145
Saved message	149
Semaphore take	149
Semaphore give	149
Timer start	149
Timer stop	150
Process	150
Object initialization	151
Connectors	152
Decision	152
SDL keywords	153
Code generation	155
Concerned elements	155
Profiles	157
UML options	203
Generated C++ code	204
Built in scheduler	209
Deployment simulator	215
Good coding practise	227
Memory allocation	227
Shared memory	227
C macros and functions	227
PragmaDev Debugger	229
Debugger architecture	229
Launching the PragmaDev Debugger	230

Stepping levels	232
MSC trace	233
Displayed information	234
Shell	240
Status bar	250
Breakpoints	250
Sending SDL messages to the running system	252
Testing	254
Code coverage	254
Connecting an external tool	254
Debugger tree view	255
Command line debug	256
SDL Z.100 project - - - - -	257
SDL types and data declarations	257
General restrictions	257
Pre-defined sorts	257
NEWTPE declarations	258
SYNTYPE declarations	259
SYNONYM declarations	260
FPAR & RETURNS declarations	260
TIMER declarations	260
SIGNAL & SIGNALLIST declarations	260
SIGNALSET declarations	260
USE declarations	261
INHERITS declaration	261
Data declarations (DCL)	261
Structural element declarations	261
SDL symbols syntax	263
PragmaDev Simulator	265
Simulator architecture	266
Simulator options	267
Launching the PragmaDev Simulator	268
Stepping levels	269
MSC trace	270
Displayed information	271
Shell	278
Status bar	287
Breakpoints	288
Sending SDL messages to the running system	289
Code coverage	290
Connecting an external tool	291
Command line simulation	291
Communication with an external XML-RPC server	291
SDL Z.100 Code generation	294
Performance Analyzer - - - - -	295
Objectives	295
Time and payload information	295

The model	295
The architecture	296
The test cases	298
Semantics	298
Table and graphical analysis	299
Architectures and test cases	301
Running the analysis	301
Saving the results	302
SDL Z.100 performance simulation	303
Validation and Model-Based Testing	305
Scope	305
IF Toolbox	305
Diversity	315
Code coverage results - - - - -	325
Generating code coverage results	325
Code coverage results viewer window	325
Merging code coverage results	326
TTCN-3 support- - - - -	328
Levels of support	328
TTCN-3 core language file editor	328
TTCN-3 co-simulation	329
Conventions	329
Restrictions	331
Simulation	333
C++ code generation	335
Stand alone	336
Combined with SDL	336
Combined with SDL-RT	336
Generate the main function	336
RTOS integration	337
Conventions.	337
TTCN-3 automatic generation	337
From an SDL/SDL-RT architecture	337
From MSCs and/or HMSCs	338
From a complete SDL system via model checking technology	342
Index - - - - -	343

1 - Project manager

The project manager window is the main window of the application. It's the first window that appears when you run PragmaDev Studio, and closing this window quits the application.



Project manager window

1.1 - Project

A *project* is the set of all needed files necessary to build your system. A project may include files of many types, as described in “Supported file types” on page 9.

A project is arranged in a tree. For SDL-RT or SDL diagrams, this tree represents the hierarchy from top-level blocks to leaf processes or procedures. The tree may also include packages to group files of different types (see “Packages and folders” on page 8).

All diagrams and files referenced by the project tree can be opened from the project manager window by double-clicking on the tree node representing it.

1.2 - Files and directories

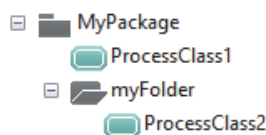
The files associated to nodes in a project tree are stored as relative paths from the directory containing the project file. This mode allows the project to be on a shared disk so that it can be used by several users on different machines. This mode is also suited for sharing a project across different platforms (Windows client and Unix file server, for example).

1.3 - Packages and folders

Within a project tree, files and diagrams may be grouped in two different types of containers:

- *Folders* are just general-purpose containers, not implying any semantics in the files or diagrams it contains;
- *Packages* are also containers for files and diagrams, but also for the elements they contain. A package represents a namespace for all elements described by or within system, block, processe, block class, process class, MSC, HMSC, class and deployment diagrams.

All elements described by or within a diagram which is in a folder are actually in the parent package for this folder. For example:

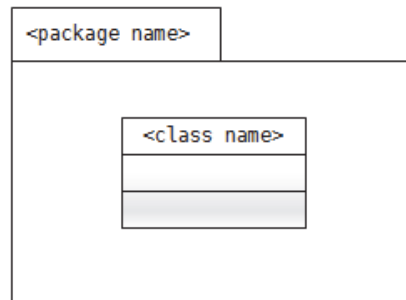


ProcessClass1 and ProcessClass2 are both in the same package: MyPackage. The folder MyFolder is just used for grouping and does not imply any organisation on the elements it contains. So any diagram including a "USE MyPackage;" in its declarations will be able to instantiate both ProcessClass1 and ProcessClass2.

In a diagram which is in a package, any element is supposed to be in that package, unless stated otherwise. References to elements outside the diagram's package is always explicit: the element is then preceded by the name of its container package followed by ' : '. There are two other ways of referencing elements outside the current package in a diagram:

- In system, block, process, block class or process class diagrams, a declaration text symbol may be used. Its text must have a line "USE <package name>;". All elements in this package are then known in the current diagram, without prefixing by the package name.

- In class diagrams, a class symbol may be enclosed in a package symbol:



Packages and folders may contain other packages or folders. There is no limitation on the number of nested containers within a project.

NB:

- PragmaDev Studio currently only supports a single "USE" in a diagram. All other references must be completely specified using the "<package> : : <name>" notation.
- Deployment diagrams cannot reference elements in other packages. They must be put in the same package as the system they describe.

1.4 - Supported file types

By default, the following file types may be included in a project:

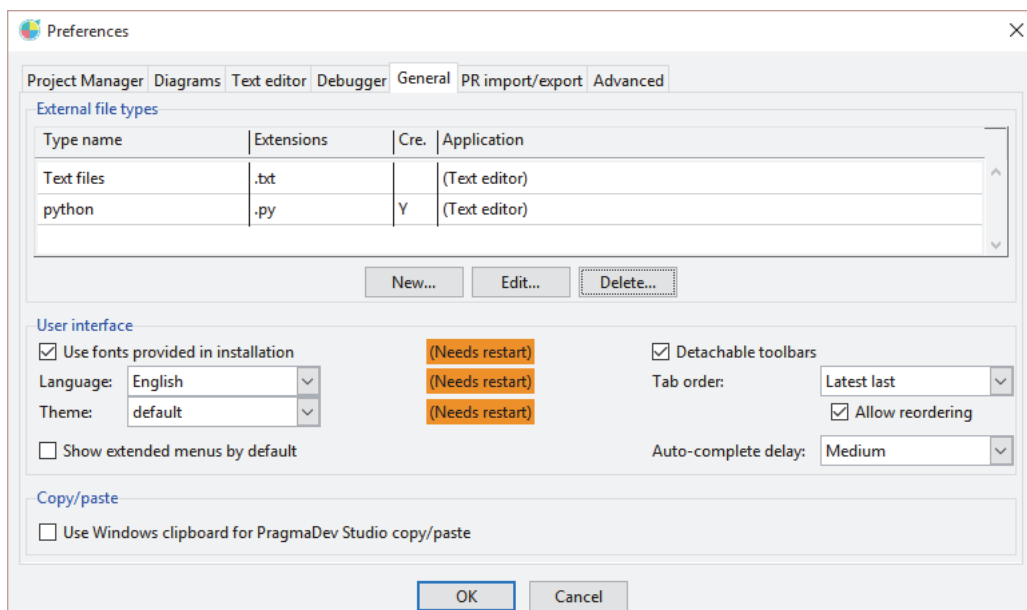
- Diagrams:
 - SDL system,
 - SDL block,
 - SDL block class,
 - SDL process,
 - SDL process class,
 - SDL procedure,
 - SDL composite state (cf. notes 1. & 2. below),
 - SDL service (cf. note 1. below),
 - SDL macro (cf. note 1. below),
 - MSC,
 - High-level MSC,
 - UML use case diagram,
 - UML class diagram,
 - UML deployment diagram,
 - IF observer diagrams.
- SDL/SDL-RT declarations files. The standard extensions for these files is .rdm.
- C/C++ files: Source and header files are supported (.c / .cpp and .h respectively).
- TTCN-3 core language files (.ttn3).
- ASN-1 declaration files (.asn1 or .asn).

- Code coverage analysis result files. The standard extensions for these files is .rdc.
- PragmaDev Studio documents. The standard extensions for these files is .rdo.
- PragmaDev Studio prototyping GUIs. The standard extensions for these files is .rdu.
- Requirements tables. These tables are stored in regular text files as tab-separated cells, or Excl-compatible CSV files (semicolon-separated with double-quoted strings).

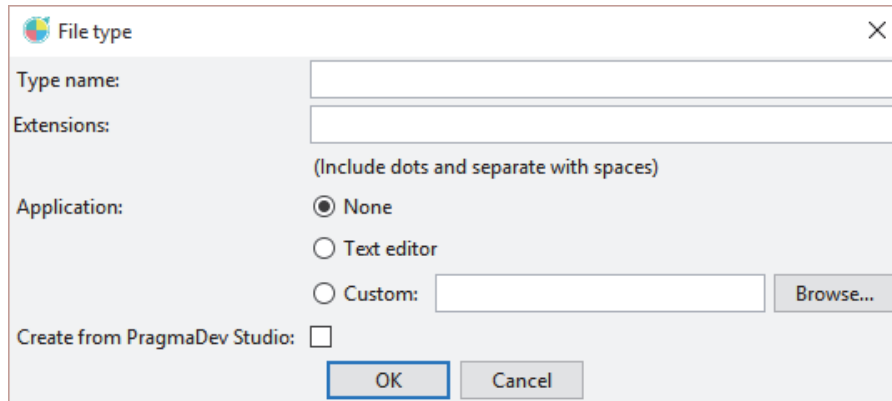
Notes:

1. These diagram types are only available in SDL projects.
2. Composite state diagrams are different from those found in SDL 2000. Please refer to PragmaDev Studio Reference Manual for a detailed description of the differences.

It is also possible to define new custom file types, allowing to include any file in the project. The definition of custom file types is made in the "General" tab in the preferences dialog, opened via the "Preferences..." item in the "Studio" menu:



Creating a new custom file type opens the following window:



The dialog box is titled "File type" and contains the following fields and controls:

- Type name:** A text input field.
- Extensions:** A text input field.
- Application:** A group box containing three radio buttons: "None" (selected), "Text editor", and "Custom:". The "Custom:" radio button is followed by a text input field and a "Browse..." button.
- Create from PragmaDev Studio:** A checkbox.
- Buttons:** "OK" and "Cancel" buttons at the bottom.

Below the "Extensions" field, there is a note: "(Include dots and separate with spaces)".

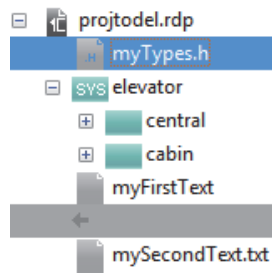
Notes:

- If the "Application" field is "None", files of this type may be included in the project, but not opened from it
- Setting the application to "Text editor" allows to do text searches and replacements in the file. Otherwise, when doing a text search/replace in the whole project, the file is skipped.
- If "Application" is set to "Custom", a full path to the application executable must be provided in the appropriate field. On Unix, you may only enter the command name if it is in your PATH.
- If the "Create from PragmaDev Studio" box is checked, it means that files with this type can be created from PragmaDev Studio project manager. If it is not, existing files with this type may be included in a project, but not created from it. The check-box is on by default when the application is "Text editor" and off in other cases.
- The application used to view the on-line documentation is the one defined for the file type with the extension ".pdf" or ".PDF". A type with one of these extensions must be defined to be able to view the manuals.
- If a custom application is defined for files with the extension .c, .cpp and/or .h, it will be used to open the files instead of the built-in text editor. *Please note* that the second note above still applies: if a custom application is associated with C files, when doing text search and replace in the whole project, these files will be skipped, which is probably not what you want.

1.5 - Rearranging the project tree

The nodes within the project tree may be re-arranged after first creation, including order changing and nesting. These functions can be done either via drag and drop, or using the copy, cut and paste operations. Several nodes can be moved or copied at a time.

When dragging nodes or during pasting, the cursor changes to a horizontal arrow pointing left. Moving this arrow along the project tree will display red horizontal lines at places where the copied node may be pasted:



The green arrow indicates where the node will be pasted. In the above example, the node will be moved or pasted between myFirstText and mySecondText node.

The actual move operation is done by dropping the moved node at the desired position. To cancel a move operation, just press the "Esc" key.

Please note you cannot move or copy/paste block or process diagrams, since what appears in the project manager must be consistent with what appears in the diagrams.

1.6 - Adding components to the system

There are three ways of adding components to your project:

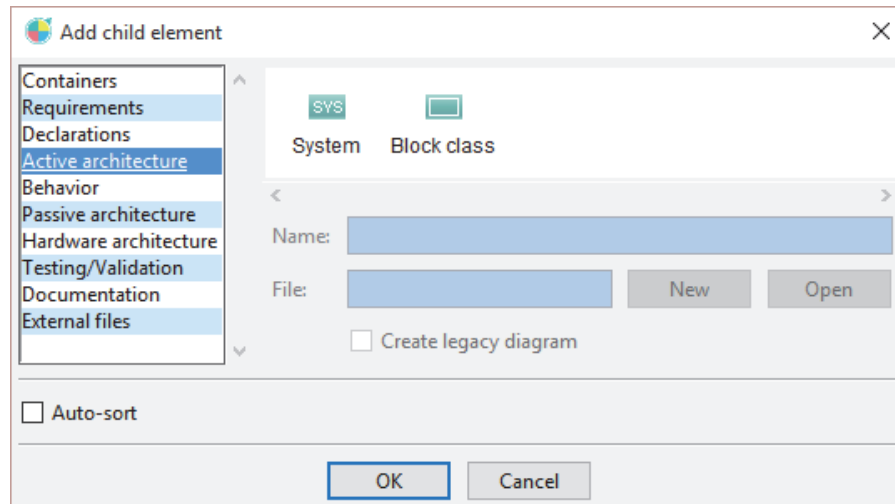
- You can directly add components one by one via the project manager in the project tree. This operation is described in “Adding a single component” on page 12.
- You can import a whole directory with all its sub-directories in the project tree. This operation is described in “Importing a directory” on page 13.
- Operations on diagrams may also automatically add a node in the project tree. This operation is described in paragraph “Creating and opening components” on page 67. This way is used to create the part of the project tree that is mapped to the block/process hierarchy

1.6.1 Adding a single component

This is the preferred way when dealing with packages, source files or top-level nodes for diagrams (such as systems, block or process classes, UML diagrams, MSC diagrams, etc...).

To add a component to a node via the project manager, just select the parent node for the new node, drop down its contextual menu with the right mouse button and select the

"Add child element..." item (or menu "Element", item "Add child..."). The following dialog appears:



On the left are listed all categories of element that can be added in a project. Once a category has been chosen, available node type are listed on the upper right of the window.

The other fields in this dialog are the following:

- **Name:** the name appearing for the node in the project tree. If not set, this field will be automatically set if you choose a file in the "File" field.
- **File:** the name of the file associated to the node. This field cannot be set if the node's type is "Package", since packages have no associated file. Otherwise, the field can only be modified via the "New" and/or "Open" buttons.
- **Auto-sort:** allows to ensure a consistent order in the parent's child nodes. If this option is checked, folders are placed first in alphabetical order, then packages, then all other children. Note this option will only work if the parent's existing children are already sorted. Otherwise, the behavior is undefined.

Create legacy diagram option allows to create a diagram as it was done with older version of PragmaDev Studio (Real Time Developer Studio).

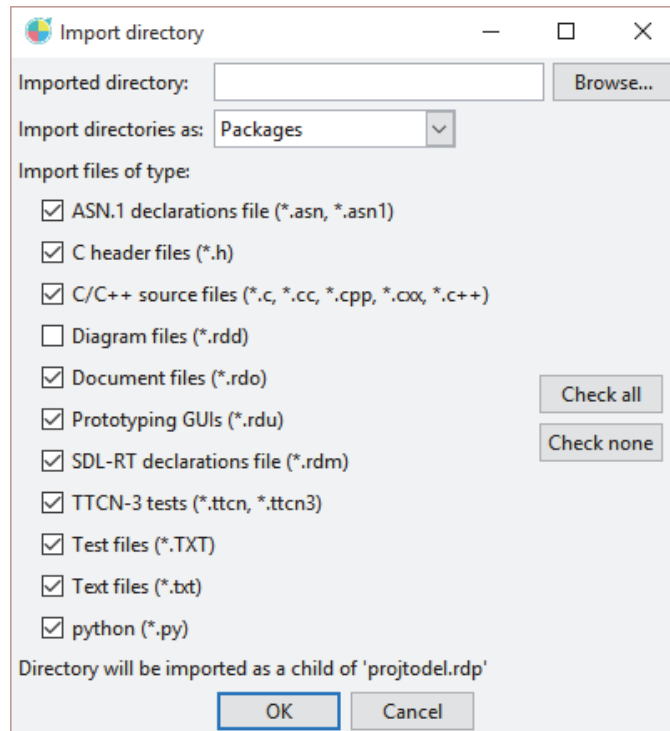
To add the child node, fill in all active fields and validate the dialog. The new node then appears in the project tree.

For the "File" field, selecting the file via the "New..." button allows to create a new empty component and selecting it via the "Open..." button allows to attach the component to an already existing file. Please note that if you select the file using "New..." and if you choose an already existing file, the file will be erased before the component is created.

1.6.2 Importing a directory

When a project is created, some files that should be included in it may already exist. For example, if a project uses an existing code library, but that may evolve with the project, all the files in this library should be included in the project for convenience. This can be achieved easily in PragmaDev Studio by using the directory import function. First select the package that will be the parent of the imported file, then select the "Import direc-

tory..." item in the "Element" menu of the project manager. The import directory options dialog appears:



This window allows to select the directory to import and the type of the files that will actually be imported in the project tree. The file types that appear are standard PragmaDev Studio types (C source and header files, diagram files) and all the external file types (see "Supported file types" on page 9). The actual import will create one node for each file with one of the selected types, mapping directories to packages.

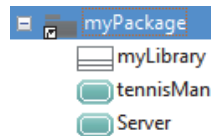
Important note: checking the "PragmaDev Studio diagram files" type in the previous dialog will probably not have the result you expect, because the diagram node hierarchy will *not* be re-created by this function. The directory import function treats the diagram files exactly like other files, which means it just creates the corresponding node without analyzing the contents of the file. Since the diagram hierarchy is described in the diagrams themselves, it cannot be re-created. If the checkbox for diagrams is checked in the directory import options dialog, a message will warn you about this issue.

1.7 - Sharing project parts

It is possible to export a part of a project and to dynamically import it in another project. All changes made to the shared part in any project will be automatically seen in all other projects.

Any sub-tree in a project can be shared. To export a sub-tree, select its root node and select "Export element sub-tree" in the "Element" menu. The sub-tree is exported to a file

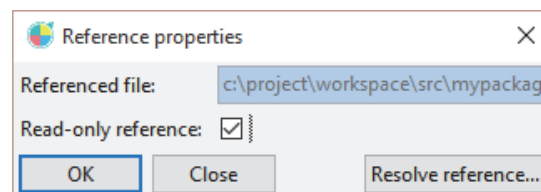
with a rdx extension. Once exported, the icon for the sub-tree root will be displayed in the project manager as follows:



The small arrow in the icon's lower left part indicates that the sub-tree is shared.

To import the sub-tree in another project, select the node under which the sub-tree should be inserted and select "Import element sub-tree" in the "Element" menu, then select the rdx file for the sub-tree to import. The whole sub-tree appears under the selected node. The icon for the imported sub-tree will be displayed with the same small arrow as the the project from which it was exported.

It is also possible to make a read-only reference. In this case, PragmaDev Studio forbids the modification of any element in the imported sub-tree. To make a reference read-only, open its properties via the menu 'Element', then 'Reference properties...', and check the 'Read-only reference' checkbox:



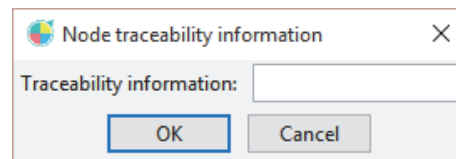
NB:

- There is no difference in the representation for shared sub-trees in the project from which it was exported and in the project into which it was imported. This is intentional, since after the sub-tree has been exported and imported, there is no difference between both projects: a change on any of them will be seen in both.
- Make sur the rdx file is not deleted or moved, or the sub-tree will disappear from both projects.
- To cancel the sharing of a sub-tree in a project, select its root node and display its reference properties via the corresponding item in the "Element" menu. In the dialog, click on the "Resolve reference" button. The sharing will then be cancelled and the icon will no more be displayed with the small arrow.

1.8 - Interface with traceability tools

Traceability information can be added to PragmaDev Studio projects, enabling to connect it to requirement management tools such as Reqtify. This information is added in the project itself to individual nodes in the tree: diagrams, files, packages, ...

The traceability information for a node can be viewed or changed by selecting the node and choosing the item "Traceability information..." in the "Element" menu. The following dialog is then displayed:



If some information has already been entered for the node, it will be displayed in the "Traceability information" field. Changing this field, validating the dialog and saving the project will change the information for the node.

Note PragmaDev Studio doesn't interpret the traceability information at all. It is just saved in the project XML file as an attribute for the node. The interpretation of the information is done by the requirement management tool.

For more details and the description of the integration with the Reqtify tool delivered with PragmaDev Studio, see the corresponding section "Traceability information" on page 40.

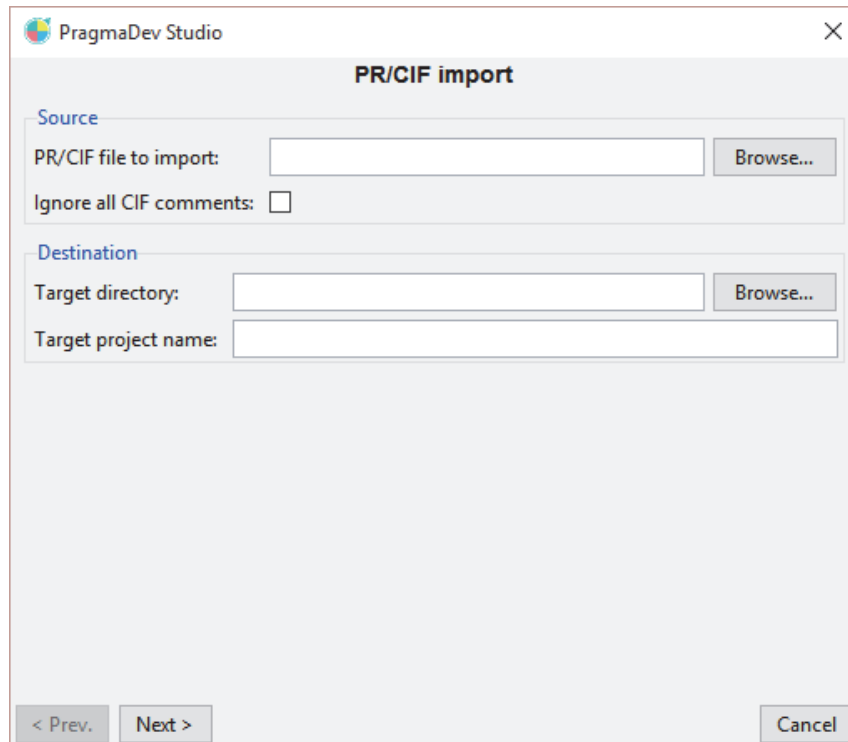
1.9 - Importing a PR/CIF file

PR (Phrasal Representation) and CIF (Common Interchange Format) files are ITU-T standards to exchange SDL models from one tool to another. PragmaDev Studio allows to import a file in SDL PR/CIF format to a PragmaDev Studio SDL project file. This is done via the "*Import SDL-PR/CIF file...*" item in the "*Project*" menu. The import config-

uration is made via a wizard; The panels in this wizard are described in the following paragraphs.

1.9.1 Source & destination panel

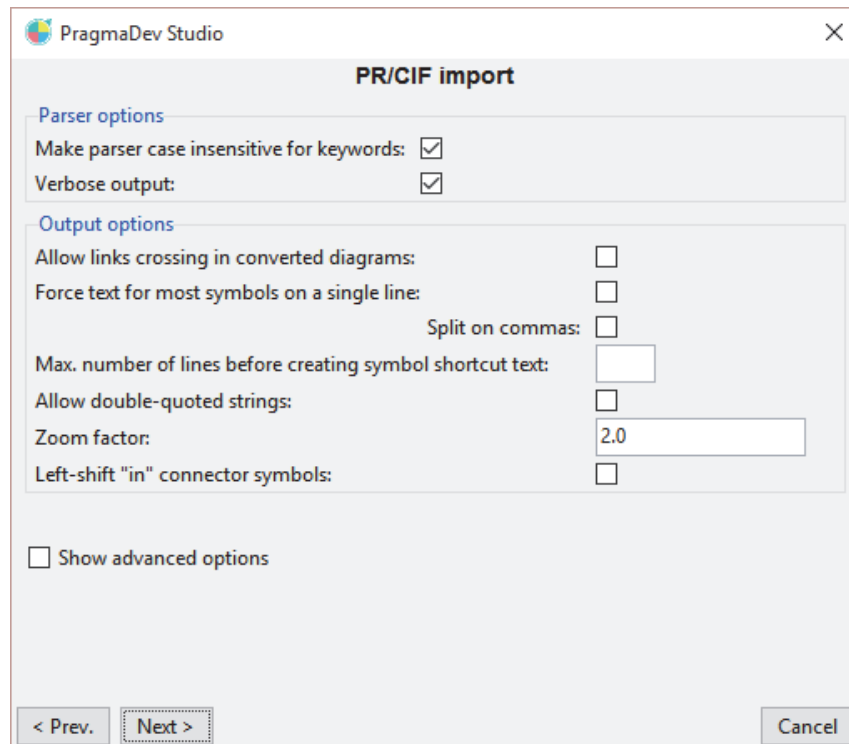
The first panel in the PR/CIF import wizard is the following:



The "*PR/CIF file to import*", "*Target directory*" and "*Target project name*" should be set to the name of the file to import, the destination directory for all created files and the name to given to the project file respectively. If the "*Ignore all CIF comments*" option is not checked, PragmaDev Studio will expect to find valid CIF comments in the imported file and will use them to set the positions and sizes for all symbols. If the comments are not present or wrong, the import may fail. If the comments are not present, or wrong, or if they should be ignored for any reason, the option must be checked. PragmaDev Studio will then automatically place and size the symbols.

1.9.2 Basic options panel

The next panel are the basic import options:



The options in this panel are:

- *Make parser case insensitive for keywords:* by default, only keywords all in uppercase or all in lowercase are considered. Checking this option allows to import a file with keywords in any case.
- *Verbose output:* By default, only error or warnings messages are displayed. This option allows to also display messages about the conversion progress.
- *Allow link crossing in converted diagrams:* This option allows links to cross other links in all converted diagrams, including systems and blocks.
- *Force text for most symbols on a single line:* By default, the text for symbols is taken as it is in the imported file. This option allows to put these texts on a single line, except for "naturally" multi-line symbols such as declarations or task blocks.
- *Split on commas:* Used with the previous one, this option will insert a newline after each comma in the symbol text.
- *Max. number of lines before creating symbol shortcut text:* Automatically creates a shortcut text for a symbol when the number of text lines is above the given threshold. If this option is blank, no shortcut text will ever be created.
- *Allow double-quoted strings:* Some SDL tools allow strings to start and end with double-quotes instead of single quotes as specified in the standard. Check this option if the tool used to create the files to import allowed this.

- *Zoom factor*: Only available when considering CIF comments. All positions and sizes in these comments will be multiplied by this factor.
- *Left-shift "in" connector symbols*: Only available when considering CIF comments. This option controls how "in" connector symbols (labels) are placed in the bounding box specified in the CIF comments for the symbol:
 - With the option unchecked, the symbol will be placed as follows:



- With the option checked, the symbol will be placed as follows:



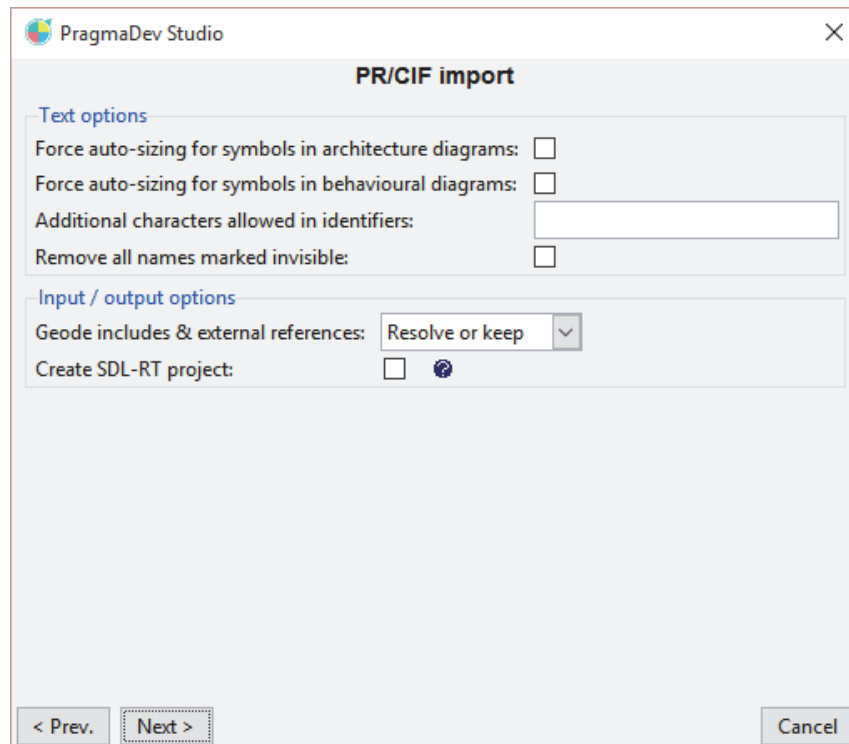
This option should be checked when importing files from Geode / ObjectGeode.

- *Create one partition for each state*: Only available when CIF comments are ignored. This option will automatically create a new partition for each STATE encountered in the imported PR file.

The next two panels are only displayed if the "*Show advanced options*" checkbox is checked in the basic options panel. Otherwise, the wizard goes directly to the summary panel (cf. "Summary panel" on page 22).

1.9.3 Advanced options panels

The first advanced options panel is the following:

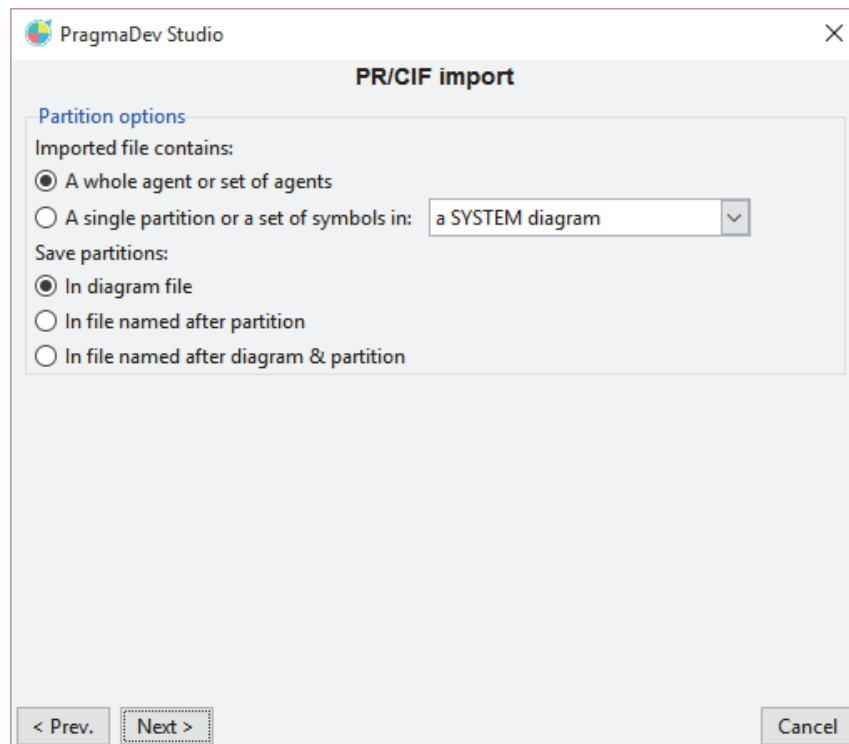


The options are:

- *Force auto-sizing for symbols in architecture diagrams*: Will automatically adapt the symbol dimensions to their text for all symbols in architecture diagrams.
- *Force auto-sizing for symbols in behavioural diagrams*: Same as the previous option, but for behavioural diagrams (processes, procedures, services and macros).
- *Additional characters allowed in identifiers*: By default, PragmaDev Studio only allows letters, digits, underscores and dots in identifiers. This field can be used to add characters that will be considered as valid in identifiers. Please note that specifying characters meaningful in the SDL syntax may have unpredictable results. Using special characters such as @ or % should however not cause any problem.
- *Remove all names marked invisible*: Only available when CIF comments are considered. With this option checked, all names marked as invisible in the CIF comments in the imported file will not be used in the created diagrams.
- *Geode includes & external references*: Specify how Geode "CIF includes" and "COMMENT '#ref ...'" will be handled in the imported project:
 - If this option is set to "Resolve or keep", the included or referenced file will be imported if it exists, or the reference on it will be kept, either as text or as a symbol PR code suffix (cf. "Symbol and link properties" on page 56).

- If this option is set to "*Resolve or discard*", the included or referenced file will be imported if it exists, or the reference will be discarded. So the name of the referenced file will not appear anywhere in the converted diagrams.
- If this option is set to "*Always keep*", the included or referenced files will never be imported and the reference will be kept as text or a symbol PR code suffix.
- *Create SDL-RT project*: In case there is C code in the input file.

The second advanced options panel is:

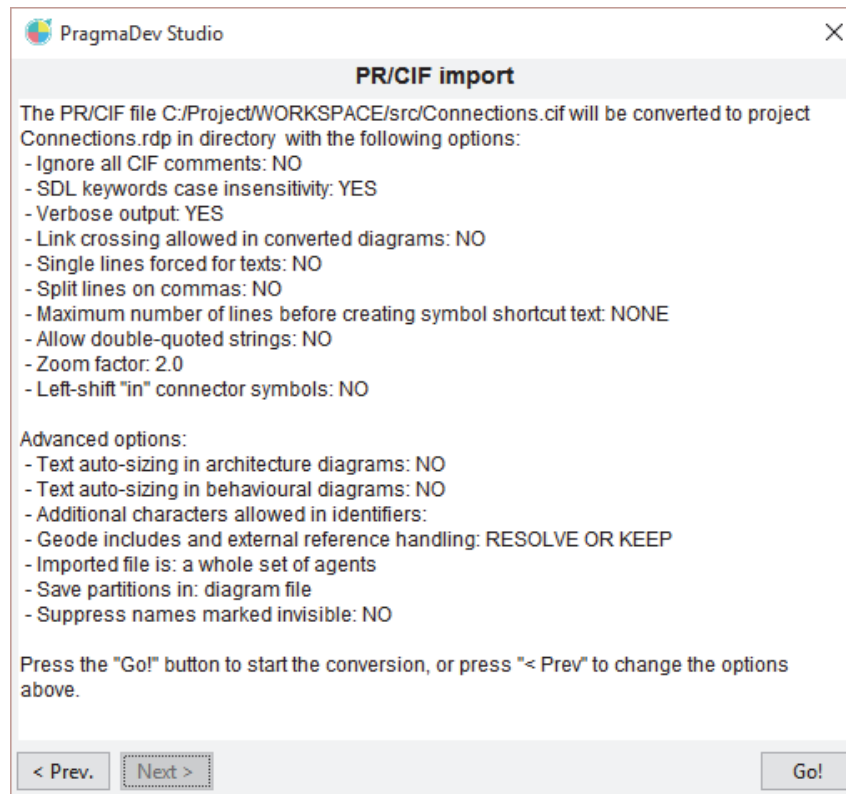


The option "*Imported file contains*" allows to import PR/CIF files containing only a part of a diagram. If this option is set to a diagram type, the imported file will be considered as containing only a part of a diagram of this type. The imported project will then include a diagram with this type marked as a diagram extract. If this diagram is exported back to a PR file, no heading or end marker will be created in the file.

The "*Save partitions*" options allows to control if partitions created in the converted diagrams will be saved in the diagram file or in an external file. If they are saved in an external file, the computed name for the partition file may include either the partition name alone or the diagram name and the partition name.

1.9.4 Summary panel

The last panel in the PR/CIF import wizard is the summary panel:



This is a summary of all options chosen in the previous panels. Pressing the "Go!" button in the panel will actually start the import.

1.9.5 PR/CIF import progress and result

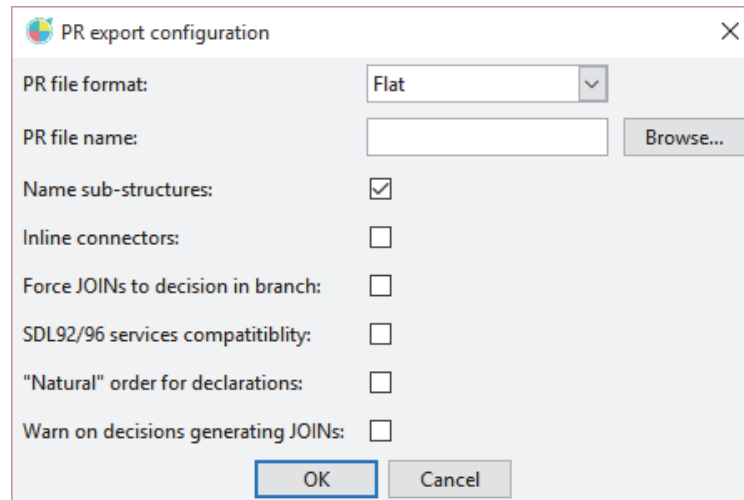
Once the import has started, a dialog will appear displaying all messages returned by the conversion, including warning and error messages if any and progress messages if the "Verbose" option was checked. All warnings and errors will also be saved in the converted project and displayed each time the project is loaded. Double-clicking on a message will automatically open the concerned diagram, allowing to check the conversion result or to correct the problem if any.

1.10 - Importing a MSC-PR file

PragmaDev Studio allows to convert a MSC PR file as defined in the ITU-T Z120 standard to a PragmaDev Studio MSC diagram file and to include it in an existing project. This feature is described in detail in paragraph "MSC PR import" on page 89. Note that the conversion does not produce a project file, but only a diagram file. So a project must be opened in the project manager to be able to import MSC-PR files.

1.11 - Exporting the project as a SDL/PR file

The item "Export as PR..." in the "Project" menu displays the following dialog:



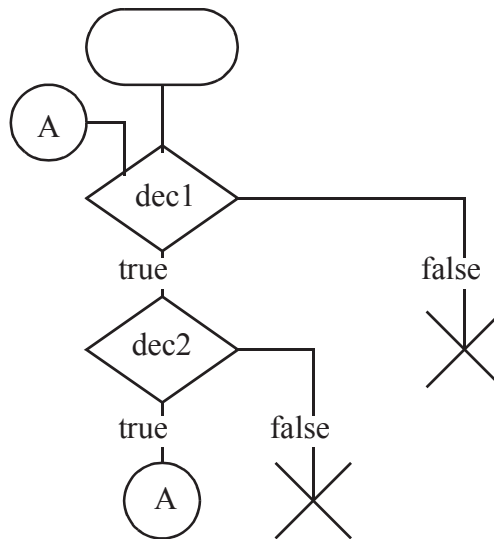
The PR file format indicates how agent definitions will be inserted in the exported file:

- In a "*Flat*" PR file, an agent defined in another one will be defined as REFERENCED in its parent and its definition will appear later in the file. The definition will be fully qualified to avoid ambiguities.
- In a "*Hierarchical*" PR file, an agent in another one will have its whole definition included in its parent's. No qualifier will be inserted as no ambiguity can occur.

The options are:

- *Name sub-structures*: By default, the SUBSTRUCTURE declarations created for blocks containing other blocks are not named. If this option is checked, a SUBSTRUCTURE name will also be inserted, which will be the same as the parent block's name.
- *Inline connectors*: By default, all connectors are put in CONNECTION blocks outside the transitions. If this option is checked, no CONNECTION blocks will be created and a label will be inserted for each connector in the middle of the first transition that uses it.

- *Force JOINS to a decision branch:* By default, a JOIN to a label in front of a decision can be put anywhere. This means for example that the following diagram:



may be exported as:

```

START;
A:
  DECISION dec1;
    (true):
      DECISION dec2;
        (true):
          JOIN A;
        (false):
          STOP;
      ENDDECISION;
    (false):
      STOP;
  ENDDECISION;
  
```

This is sometimes not desirable, since the JOIN to the label before the first DECISION is generated in a branch of the second DECISION. This makes it difficult for external code generators to figure out that the JOIN may actually be converted to a while-style loop.

Checking the "*Force JOINS to a decision branch*" option will always try to put JOIN statements to a DECISION in one of this DECISION's branches. So the diagram above would be exported as:

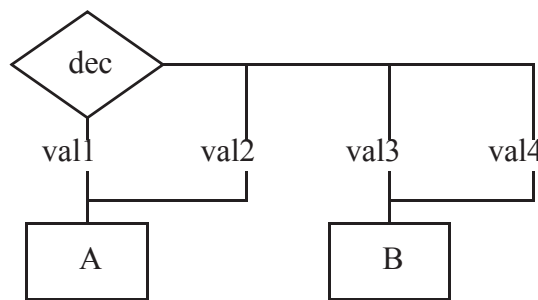
```

START;
A:
  DECISION dec1;
    (true):
      DECISION dec2;
        (true):
          (false):
            STOP;
      ENDDECISION;
      JOIN A;
    (false):
      STOP;
  ENDDECISION;
  
```

- *SDL 92/96 services compatibility:* By default, processes defining composite states are exported as such in the PR file, using the SDL 2000 syntax for composite states. Checking this option will try to export them as SDL 92/96 processes containing services. To be able to do this, the processes must:
 - Define at most one composite state;
 - If they do, have only a start transition containing a single NEXTSTATE symbol to this composite state.

In this case, the services defined at composite state level will be put directly into the parent process using a SDL 92/96 syntax for services. If any process does not meet these constraints, the export will fail.

- *"Natural" order for declaration:* By default, all declarations and decision branches are exported with no specific order. The order may even change from one export to another, even if the diagrams have not changed. Checking this option will force a deterministic order on exporting. This order is the "natural" one when available, i.e the order of the symbols in the diagrams.
- *Warn on decisions generating JOINS:* There are some cases where a decision in a diagram cannot be exported without generating an additional label. Here is such a decision:



One of the task blocks A or B can be put after the ENDDECISION, but for the other one, a label must be generated or the symbol's code will have to be duplicated in the exported PR.

By default, this additional label is silently generated. This option forces a warning to be displayed if such a label is generated.

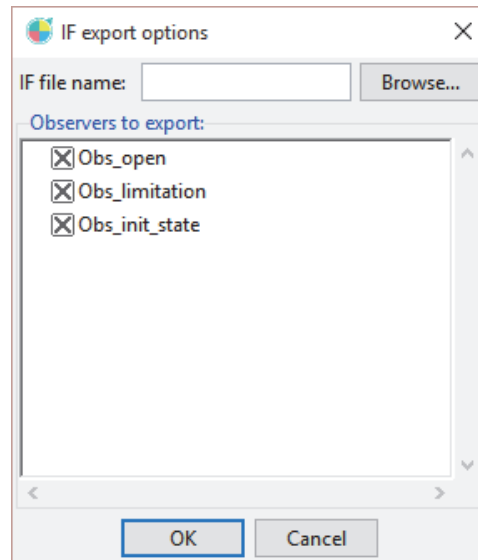
Note: The PR export is available in all project types, but is likely to produce an interesting result only for Z100 SDL projects. A warning will be displayed if it is attempted on another type of project.

1.12 - Exporting a project as a Verimag IF file

PragmaDev Studio allows to export a whole project in the IF language as defined by Verimag. For the current possibilities and limitations, refer to the corresponding chapter in the Reference Manual.

It is also possible to define an external model checker based on IF via the *Validation/Options...* menu described in "Validation and Model-Based Testing" on page 305.

For both features, if IF observer diagrams exist in the current project, PragmaDev Studio will ask which ones should be exported with the system via the following dialog:



1.13 - Exporting elements as HTML files

There are two ways of exporting elements as HTML files:

- Exporting a single element
- Exporting the whole project

These two ways are described in the paragraphs below.

1.13.1 Exporting a single element

Exporting an element is done via the "Save element as HTML..." item in the "Element" menu. An element must be selected in the project tree.

When exporting, all elements referenced by the exported one are also recursively exported, and links are created for each referenced element. For example, if the exported element is a block diagram and includes a process symbol, the element describing this process is also exported. In the HTML file for the block, a link will be created on the process symbol which will open the HTML file for the process.

The destination directory for all the HTML files is asked to the user.

1.13.2 Exporting the whole project

Exporting the whole project is done via the "Export as HTML..." item in the "File" menu. This kind of export does the same thing than exporting all elements in the project, but also generates an HTML file for the whole project. This file includes two frames: the one at left hand contains a representation of the project tree and the one at right hand may contain any HTML file for any element in the project. Clicking on a node in the left frame opens the corresponding element in the right one.

The file name for the HTML file for the project is asked to the user. All other HTML files will be generated in the same directory than this file. If a file in this directory has the name of a generated file, it will be silently replaced.

1.14 - Export all the publications in a whole project

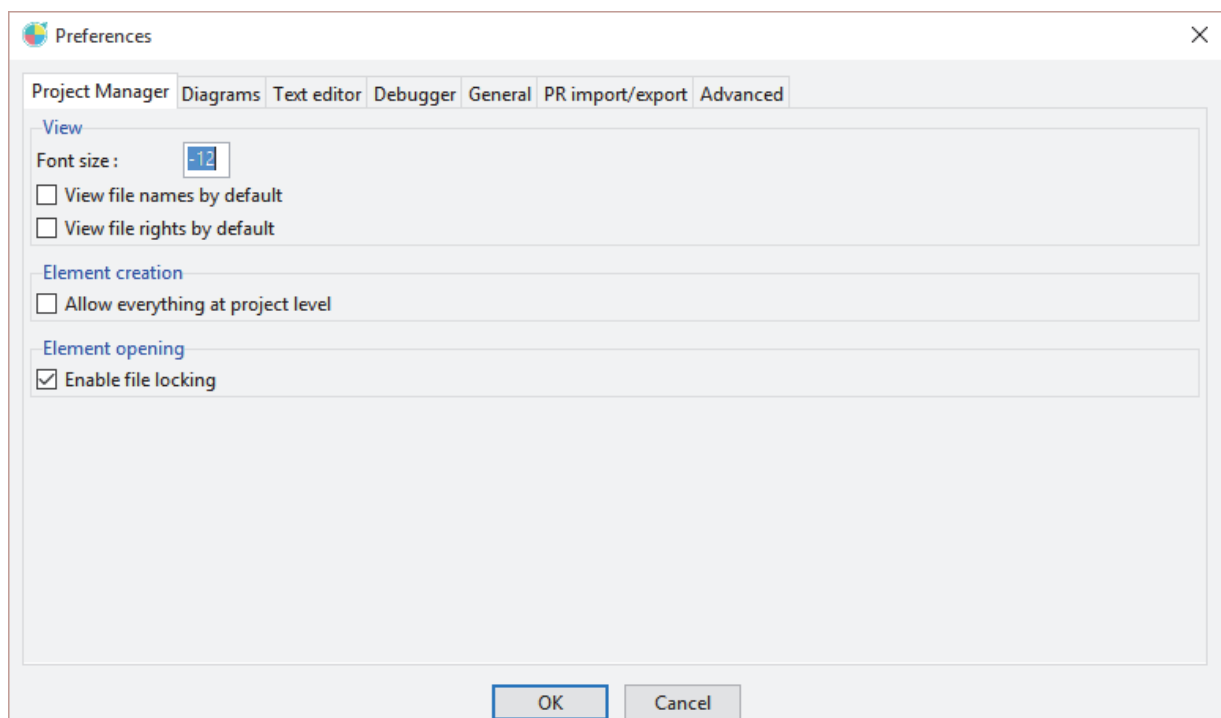
PragmaDev Studio allows to export diagrams or parts of diagrams as publications to be able to include them in external documentation. Publications are described in “Publications” on page 62.

It is possible to export all publications in all diagrams in a whole project. This allows to be sure that everything is up to date in the external documentation. Please note that this function will open all diagrams in the project, even if they have no publications at all. So it may be quite long for large projects.

1.15 - Preferences

The preferences for the application are opened via the item "Preferences..." in the "Studio" menu. The dialog that appears is divided in tabs, that are described in the following paragraphs.

1.15.1 Project manager preferences

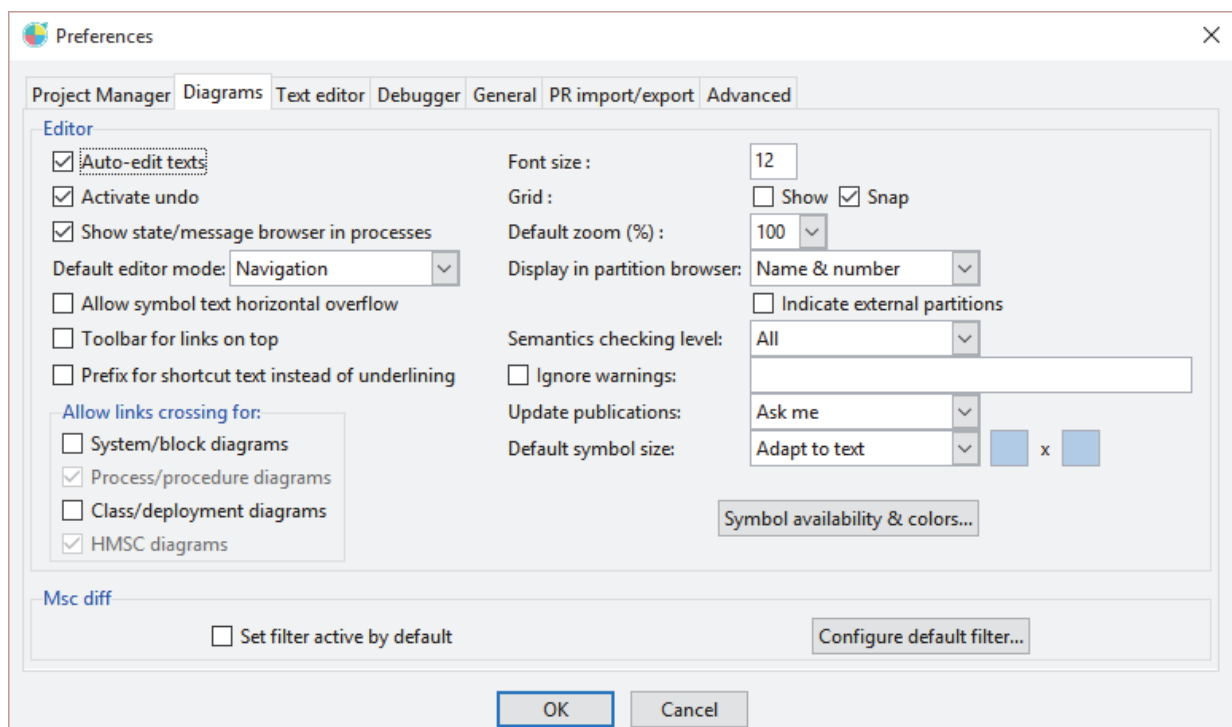


The "View file names by default" and "View file rights by default" options control whether the file names and/or access rights will be automatically displayed in the project manager window when a project is opened.

The option "*Allow everything at project level*" allows to create any kind of diagram directly in the project. By default, no other SDL diagram than systems can be created at project level.

The option "*Enable file locking*" controls whether the files you open for modification are locked for your personal use or not. If this option is set, two users cannot modify the same file at the same time.

1.15.2 Diagram preferences



The "*Auto-edit text*" option controls whether the text for symbols or links will be automatically opened for modification when the symbol or link is created.

The "*Activate Undo*" option activates the undo operation in the diagram editor. Deactivating the undo can speed up editing on very large diagrams.

The "*Show state/message browser in processes*" option controls whether the browser allowing direct access to transitions for each state and input message is displayed when opening a process. Refer to paragraph "'View' / 'Go to' menu and state / message browser" on page 72 for further details on this browser.

The "*Default editor mode*" option controls the default mode for the editor, which can be "*Edition*" or "*Navigation*". In both modes, a button appears while hovering over a symbol. In edition mode, the button allows to open the diagram associated to the symbol if any, and double-clicking on the symbol edits its text; in navigation mode, the button allows to edit the symbol's text, and double-clicking on the symbol opens the associated diagram.

The "*Allow symbol text horizontal overflow*" option controls how the text is displayed if it is too large for the symbol. With this option unchecked, the text will be as wide as the

symbol and will overflow only above and below the symbol. With this option text, no width will be defined for the text and it may overflow on all sides of the symbol, including left and right sides.

The *"Toolbar for links on top"* option controls the placement of the toolbar for link insertion in diagram editors; by default, this toolbar is placed below the toolbar for symbol insertion. If this option is checked, the link toolbar will be put above the symbol toolbar.

The *"Prefix for shortcut text instead of underlining"* option controls the display for symbols with a shortcut text. The default is to underline the symbol text; if for any reason, another representation is preferred, checking this box will display a prefix looking like ">>" instead.

The options in the *"Allow links crossing"* group set on which diagram the crossing of links will be allowed. This option is today always set for process, procedure and HMSC diagrams. It can be activated for system, block, class and deployment diagrams by checking the corresponding check-box. Please note once a diagram is saved when the corresponding option is on, the crossing of links will be definitely allowed for the diagram. There is no way to forbid link crossing for a diagram that allows them.

The font size is the font size for diagrams. It does not apply in the source file editor.

The *"Grid"* options control the default behavior of the grid in diagram editors: If the *"Show"* check-box is checked, the grid will appear; if the *"Snap"* check-box is checked, all objects will align automatically to the grid.

The *"Default zoom"* is the initial value for the zoom level for all diagram windows.

The *"Display in partition browser"* option controls what is displayed in the partition browser in diagram editors. The display may include the partition order number, its name or both. If the *"Indicate external partitions"* checkbox is checked, partitions stored in external files will be prefixed with a '*' in the browser.

The *"Semantics checking level"* option can be set to:

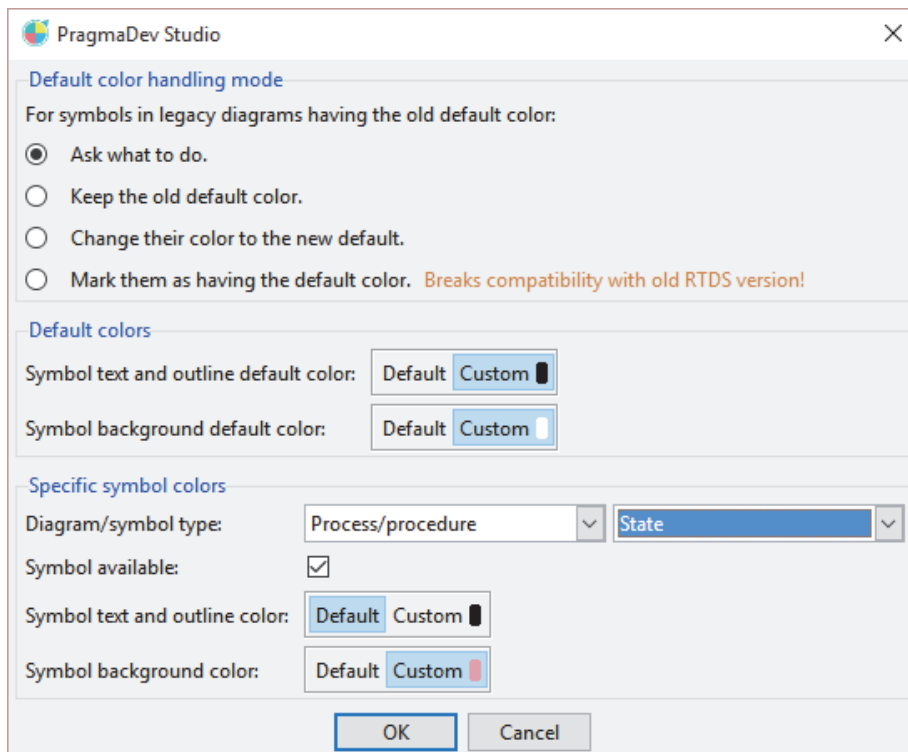
- "All" to report errors for any problem
- "Critical only" to report errors only for problems preventing the system to work and only warnings for other problems

The *"Ignore warnings"* checkbox and entry allow to specify which warnings won't be displayed during a syntax / semantics check. If the box is not checked, all warnings are displayed. If the box is checked, warnings having the identifiers specified in the entry will not be displayed. The warning identifiers should be separated with spaces. Warning identifiers are documented in PragmaDev Studio Reference Manual.

The *"Update publications"* option controls whether the publications of a diagram will be automatically saved when the diagram is saved. See paragraph "Publications" on page 62.

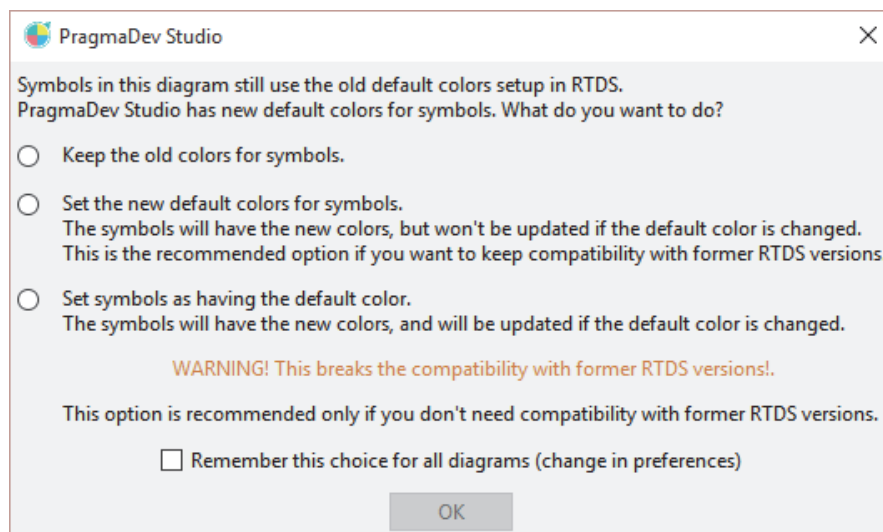
The *"Default symbol size"* option controls how symbol are sized when created. If this option is set to *"Adapt to text"*, the symbol size will not be fixed and will adapt to whatever text is entered in it. The other option is *"Set to:"* and requires a default size to be entered. If these are set, all symbols will have the specified size when created. The unit for the size are printer points (same unit as font sizes).

The "*Symbol availability and colors...*" button allows to configure which symbols will appear in the toolbars in diagram editors, as well as the default colors for all symbols in diagrams depending on their type. Pressing this button opens the following dialog:



The upper part of the dialog allows to choose what PragmaDev Studio will do in case of a legacy diagram (diagram coming from older version of PragmaDev Studio tool, called Real Time Developer Studio):

- *Ask what to do*: Each time a legacy diagram will be opened, the following window will be opened asking what to do:



- *Keep the old default color*: will keep default color from legacy diagram.

- *Change their color to the new default:* will change color of symbol to the new default colors.
- *Mark them as having the default color:* symbols will have the new colors and will be updated if default color is changed. Doing this break the compatibility with former Real Time Developer Studio.

The middle part of the dialog allows to change the default color for all symbols. Two colors may be selected: the one for outline and text and the one for background.

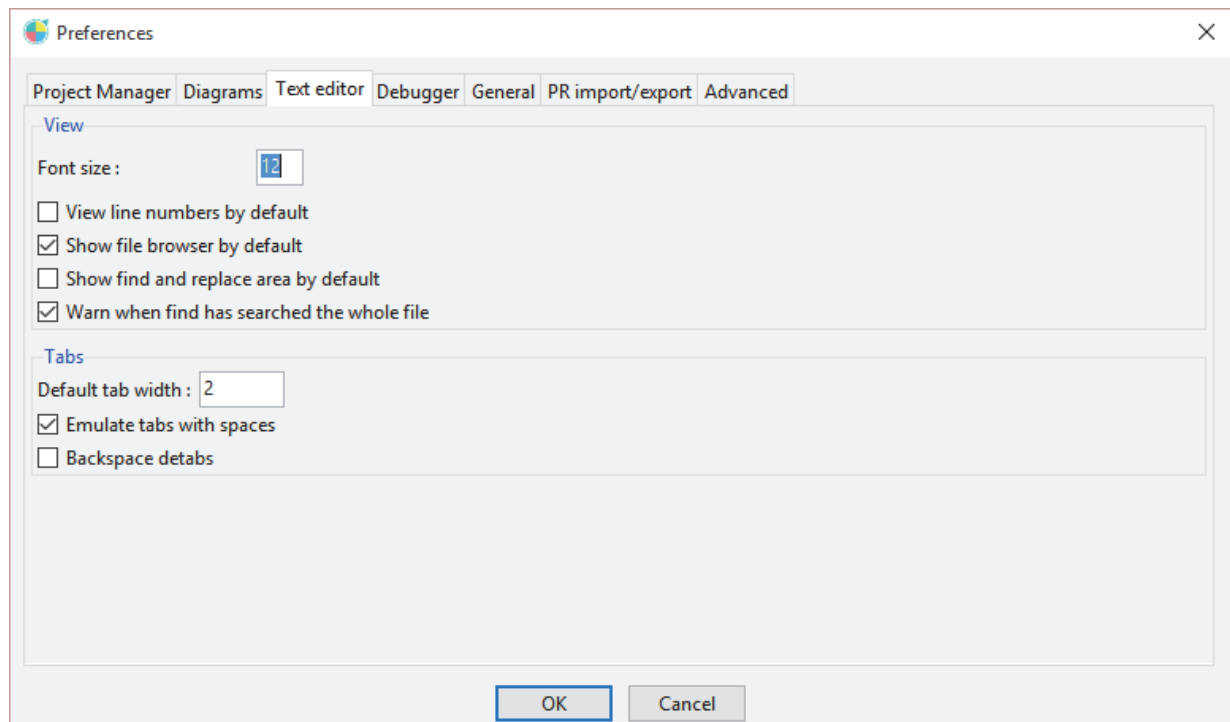
The lower part allows to configure the availability and colors for individual symbol types: select the parent diagram type for the symbol type and the symbol type itself in the corresponding menus: Its current availability, outline color and background color are then displayed in the lower part of the dialog. Checking or unchecking the availability box will make the symbol be present or absent from the toolbars in the diagram editors. Specifying new colors will override the default ones for all symbols with the selected type.

Note that making a symbol type unavailable only prevents it from appearing in the toolbars, it doesn't make it invalid. If a diagram containing a symbol with this type is opened, it will still be displayed correctly, and all operations will work on the symbol.

Also note that the default colors only applies when the symbol is created. All symbols keep the colors set when they were created, so setting colors in this dialog will not change symbols in existing diagrams. Colors for specific symbols may be changed via the symbol properties dialog in the diagram editor (see "Symbol and link properties" on page 56).

The options in the "MSC diff" group are the default values for the filter when doing a MSC diff / merge. A detailed description of these options can be found in paragraph "Compare diagrams" on page 81.

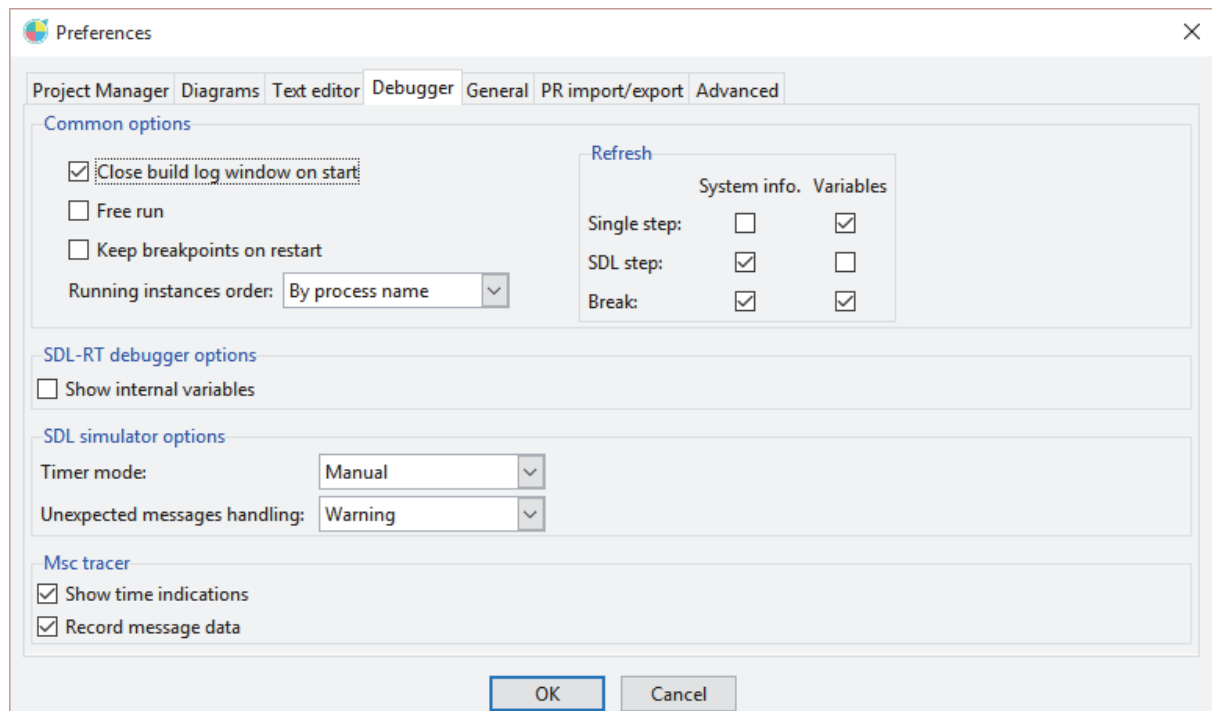
1.15.3 Text editor preferences



These options are the default values for those found in the "Preferences" menu in the text editor. They control:

- The font size for texts ("*Font size*");
- The visibility of the line numbers ("*View line numbers by default*");
- The visibility of the class/method/function browser ("*Show file browser by default*");
- The visibility of the find / replace bar in the editor ("*Show find and replace area by default*");
- Whether the search function should warn when the whole current file has been searched ("*Warn when find has searched the whole file*");
- The width for tab characters ("*Default tab width*");
- Whether the "Tab" key should insert hardware tabs or spaces ("*Emulate tabs with spaces*");
- Whether the backspace key should act as if it deleted a tab ("*Backspace detabs*"). This behavior is only active when there is only whitespace between the beginning of the line and the current position.

1.15.4 Debugger preferences



The options in "Options" are the default values for those found in the "Options" menu in the SDL-RT debugger and SDL simulator windows. They're described in chapters "PragmaDev Debugger" on page 229 and "PragmaDev Simulator" on page 265.

In the "Msc tracer" options, "*Show time indications*" controls whether timing information for the executed system are included in the trace as absolute time constraints symbols or not. "*Message data*" controls how the data associated to the exchanged messages will be displayed (completely, only n bytes or not at all).

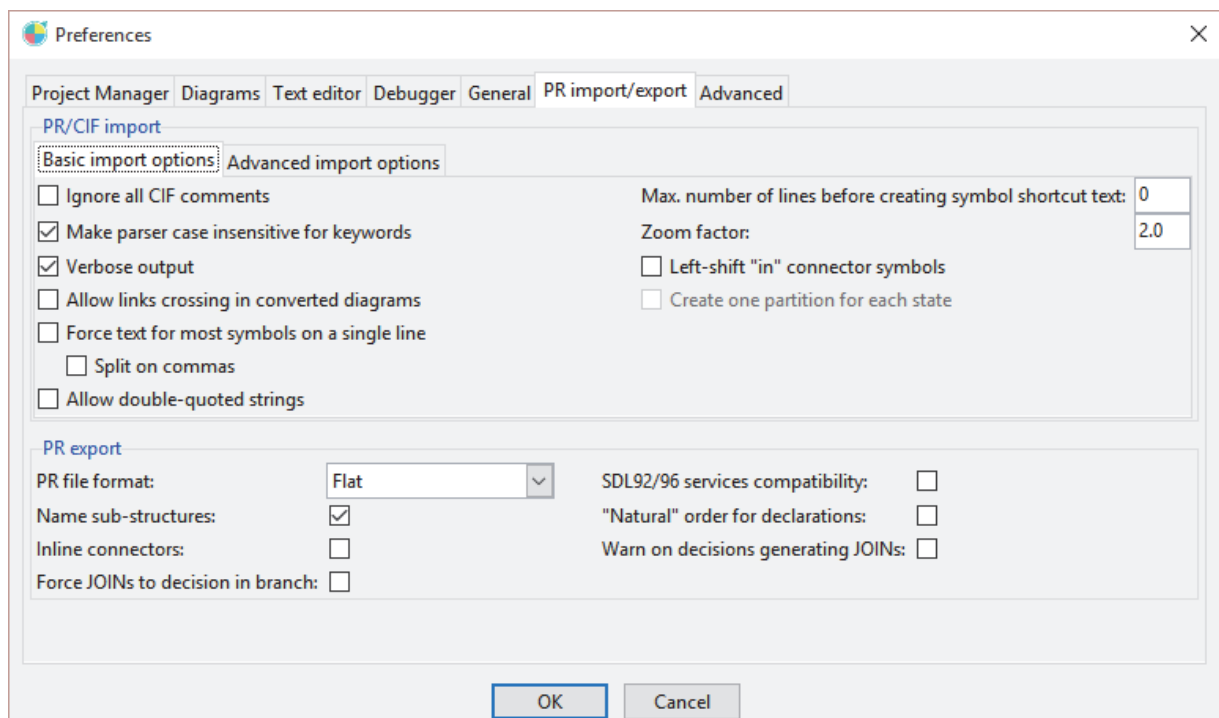
1.15.5 General preferences

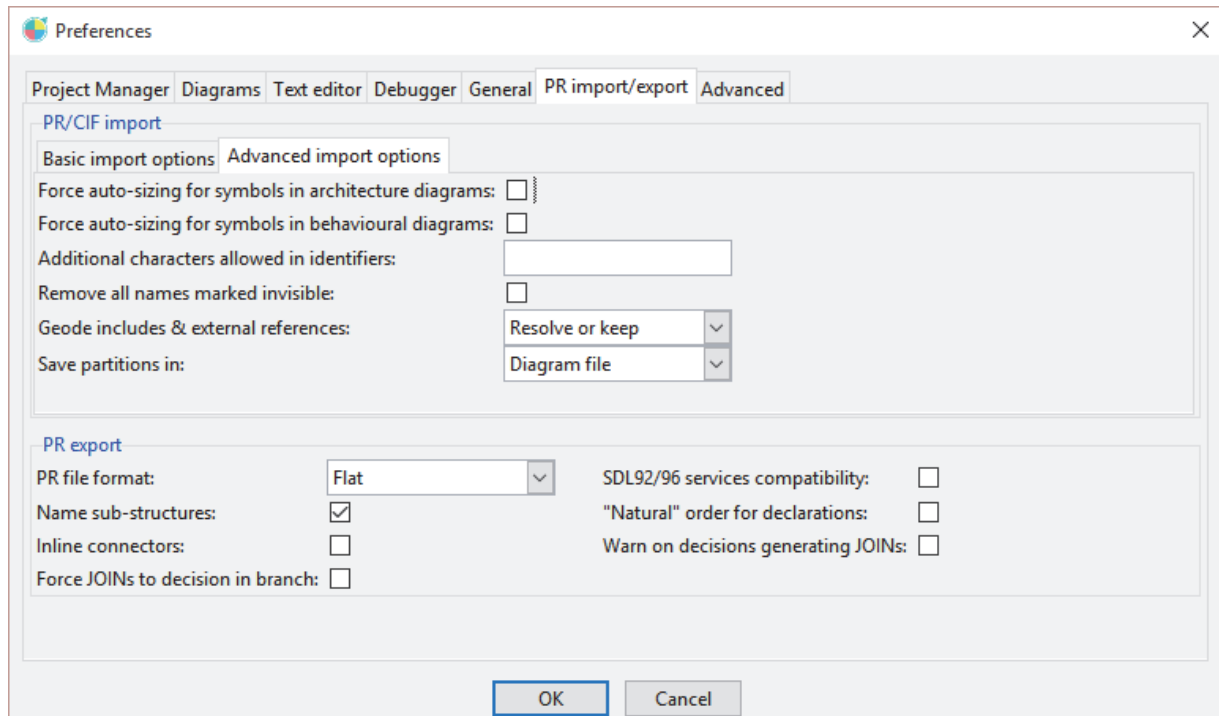
This tab is shown in paragraph "Supported file types" on page 9. It contains:

- The external file types defined for the application.
- The option setting the font for diagrams and source files to the font delivered with PragmaDev Studio. This option will be automatically set when PragmaDev Studio is launched for the first time.
- The option allowing to emulate a double-click with a middle click or a control-left click. This can be used as a workaround for a bug in some versions of Windows where the double-click does not work with PragmaDev Studio.
- The language setting for PragmaDev Studio user interface; when this setting is modified, PragmaDev Studio will have to be restarted for the changes to appear.
- The GUI theme to use for PragmaDev Studio.
- The options "*Detachable toolbars*" makes the toolbars in the various editors detachable from their parent window.

- The option “*Tab order*” determines how tabs will be ordered in the editor windows. Values are “*Latest last*”, “*Latest first*” or “*Alphabetical*”. If the “*Allow reordering*” option is checked, it will be possible to reorder tabs in their parent window. Note that detaching tabs is always possible.
- On Windows, whether the copy / paste operation for diagrams should use the regular Windows clipboard or PragmaDev Studio own internal one. In the first case, copying symbols from one PragmaDev Studio instance to another will be possible, but it will delete the former clipboard contents and have unexpected results if symbols are pasted in anything else than PragmaDev Studio. In the second case, it will be impossible to paste copied symbols in anything else than the current PragmaDev Studio instance, including other PragmaDev Studio instances.
NB: This option is not available on Unix as copied symbols can only be pasted in another PragmaDev Studio.
- On Unix, the default print command to use for all print operations.

1.15.6 PR import & export preferences





These preferences are the default values for most of the PR import & export options. The description of these options can be found at paragraphs “Importing a PR/CIF file” on page 16 and “Exporting the project as a SDL/PR file” on page 23 respectively.

1.15.7 Advanced options

As their name implies, these options are for advanced users only. They should be left to their default values except on explicit advice from the PragmaDev support team.

1.16 - User defined external tools

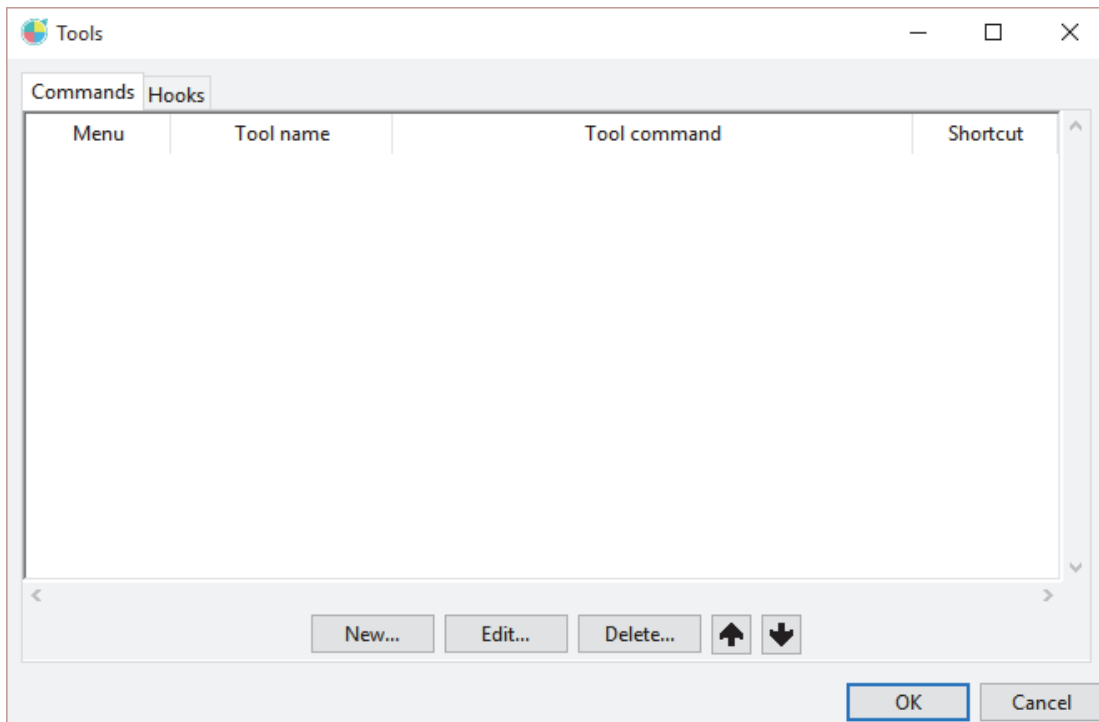
The project manager allows to define a set of external tools that may be used on the files in the project or on the project itself. A typical use of these tools is to interface Real Time Developer Studio with a configuration management system, but their use is much larger than that.

External tools appear in new dynamic menus in the project manager. They are defined via the "Tools" sub-menu in the "Studio" menu of the project manager. The three items in this sub-menu are:

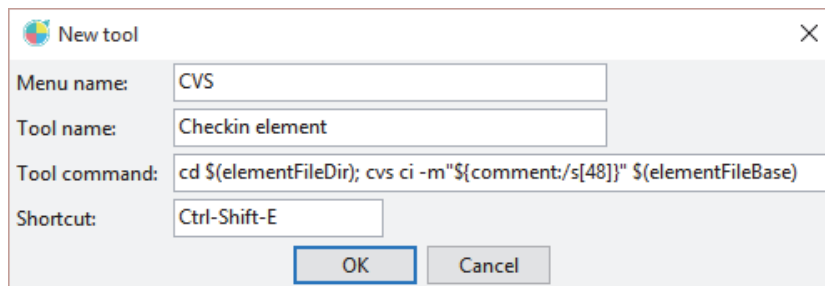
- "Configure...", allowing to manage external tools (see below),
- "Import..." and "Export...", allowing to share tool definitions between different users using a representation of these definitions in a text file.

1.16.1 Tool menus definition

The "Tools / Configure..." sub-menu opens the following window:

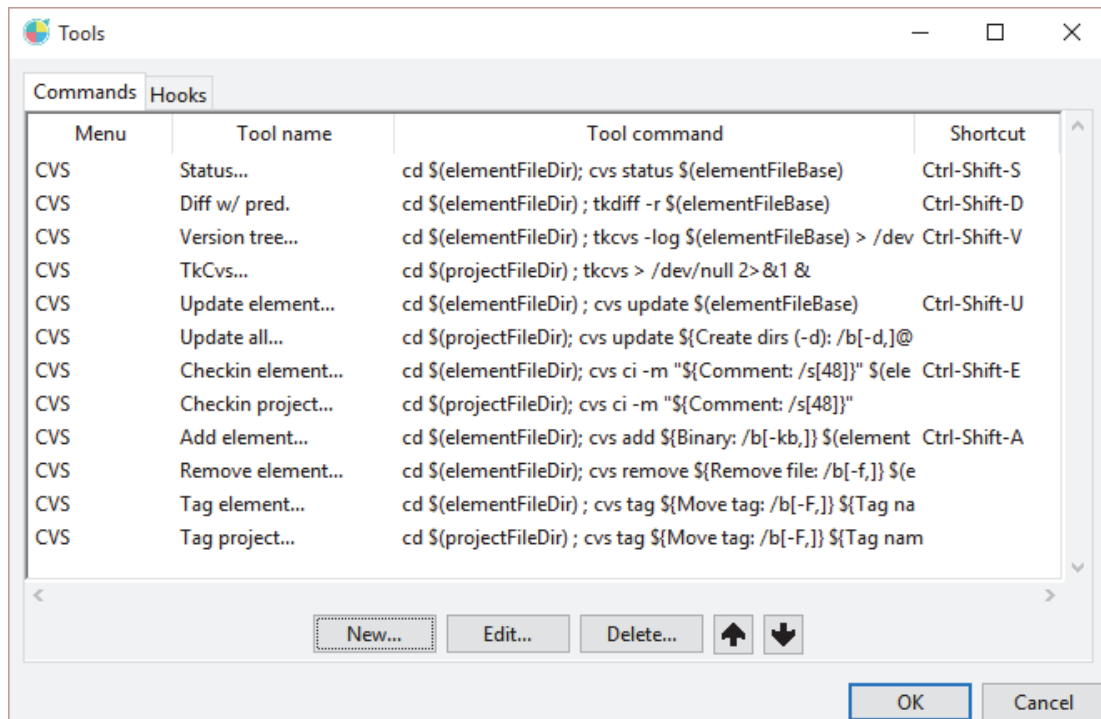


The columns are the name for the menu, the name of the menu item, the command to be run for the tool, which may include special markers (see "Tool commands" on page 37) and the shortcut for the item. Defining a new tool or editing an existing one opens the tool definition window:

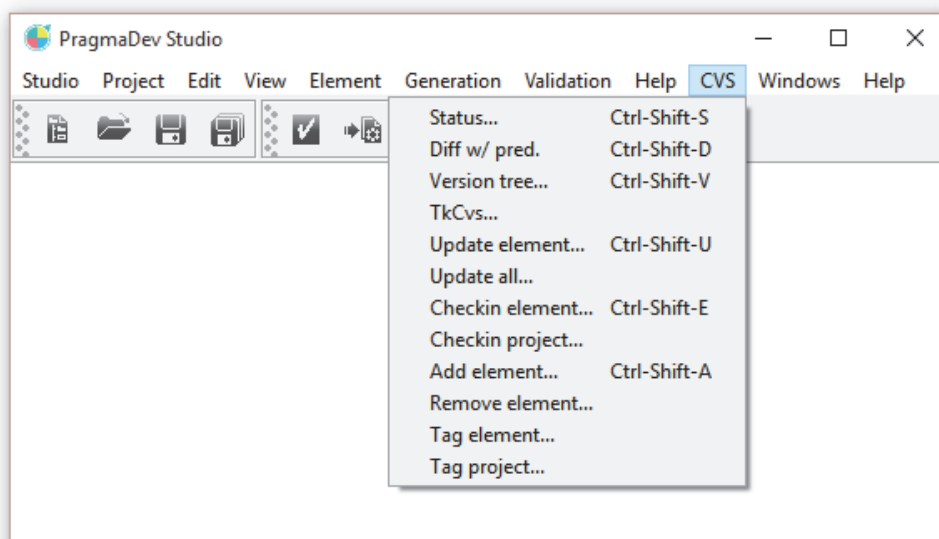


where these 4 fields may be entered or modified. The arrow buttons in the main tools window allow to reorder the tools in their menu.

Below is an example of a typical CVS menu:



Once defined, the menus appear in the project manager window:



There is no way to control the order of the tool menus in the current version of PragmaDev Studio.

1.16.2 Tool commands

The tool commands are regular OS-dependent commands that will be executed via a shell or equivalent (sh on Unix, command.com or cmd.exe on Windows). These commands may however include special markers, allowing to get information from the current project, the currently selected element in the project manager, or from the user via a dialog box. These markers are:

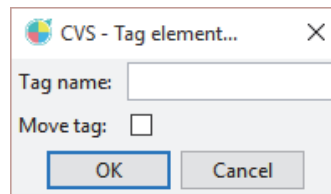
- `"$(projectFile)"`: this marker will be replaced by the opened project's full file name. If no project is opened, a message will be displayed and the tool command won't be executed.
- `"$(projectFileDir)"`: same as `$(projectFile)`, but for the project directory.
- `"$(projectFileBase)"`: same as `$(projectFile)`, but for the project file name without its directory.
- `"$(elementFile)"`: this marker will be replaced by the full file name for the currently selected element in the project manager. If no element is selected, a message will be displayed and the tool command won't be executed.
- `"$(elementFileDir)"`: same as `$(elementFile)`, but for the selected element directory.
- `"$(elementFileBase)"`: same as `$(elementFile)`, but for the selected element file name without its directory.
- `"$(descendantElementFiles)"`: this marker will be replaced by the list of all file names for the currently selected element and all its descendants. The file names will be separated by the standard path separator for the current platform (':' for Unix; ';' for Windows).
- `"${<label>/<type>[<options>]@<order>}"`: this type of marker is used to ask information to the user before executing the command. All markers of this type will be used to build a dialog where the user may enter information before executing the command. The dialog will include one field per marker, built from the information between the braces in the marker:
 - `"<label>"` will be the text written before the field. It defaults to the empty string.
 - `"<type>"` is the field type. Today, two types are recognized: "s" for strings and "b" for booleans. The corresponding field type are a text entry and a checkbox respectively. The default type is string.
 - `"<options>"` are options for the chosen type. For strings, the only option is its length (default: 20). For booleans, options are the value when checked and the value when unchecked, separated by a comma. For example, a field with type "b[-r,]" will be replaced in the command by "-r" if the user checks the corresponding checkbox, and by the empty string otherwise. The defaults are "1" for checked and "0" for unchecked.
 - `"<order>"` is the order of the field in the dialog. If not set, the order will be chosen randomly. If set only for a subset of the fields, the ordered fields will always appear before the unordered ones.

Example:

If a tool command includes the following markers:

- `${Tag name: /s@1}`
- `${Move tag: /b[-F,]}`

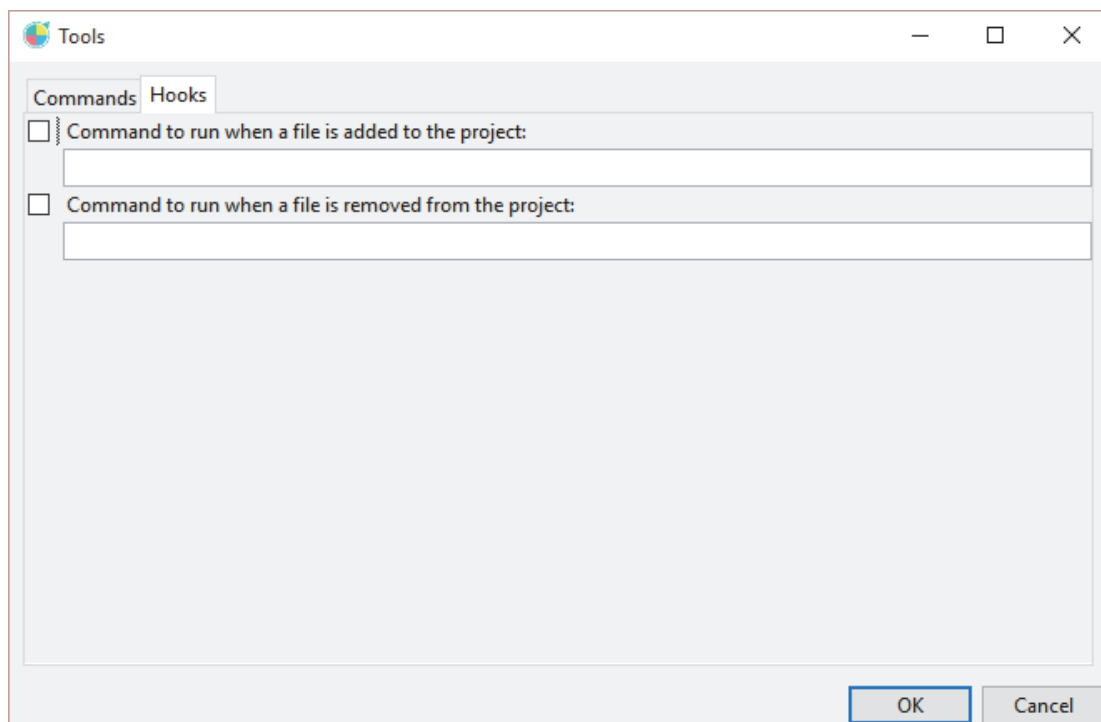
the following dialog will be built:



The "\${Tag name...}" marker will be replaced by the contents of the "Tag name:" entry field in the dialog and the "\${Move tag...}" marker will be replaced by "- F" if the checkbox is checked and by the empty string otherwise.

1.16.3 Hooks addition and removal

PragmaDev Studio allows to automatically call a command when an element is added or removed from the opened project. These "hooks" are configured in the external tools management dialog in *Studio / Tools / Configure...* and *Hooks* tab:



The syntax for the hook commands is the same as the one for the regular tool commands.

Please note that if the command contains user-defined variables (\${...}) and if several files are added or removed in a single operation, the value for the variables will only be asked once and applied to all added or removed files.

1.17 - Traceability information

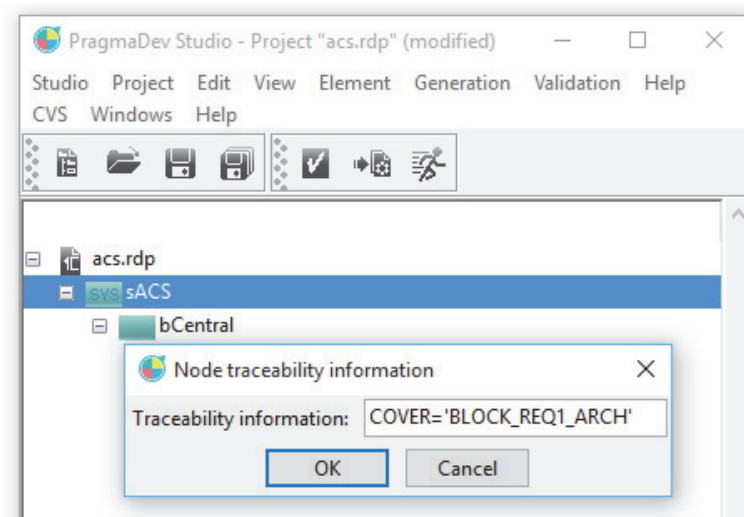
1.17.1 Scope

Traceability information can be defined on any element of the *Project manager*. These informations can be used in a traceability informations tool in order :

- to manage requirements,
- to analyze the impact of modifications.

1.17.2 Traceability editor

To add traceability information, select a node in the *Project manager* and go to the menu *Element / Traceability information* :



The traceability information is stored in the XML PragmaDev Studio project file (.rdp) as the attribute `traceabilityInfo` of the node tag.

1.17.3 Integration with Reqtify

1.17.3.1 Organisation

An integration with Reqtify traceability tool is delivered with PragmaDev Studio. This integration is composed of the following files :

- `rtds.br` : PragmaDev Studio behavior file,
- `rtds.types` : PragmaDev Studio type file,
- `pragma.bmp` : PragmaDev logo,
- `rtds_block.bmp` : Block diagram logo,
- `rtds_block_type.bmp` : Block class diagram logo,
- `rtds_class.bmp` : Class diagram logo,

- `rtds_codecov.bmp` : Code coverage logo,
- `rtds_compstate.bmp` : Composite state diagram logo,
- `rtds_depl.bmp` : Deployment diagram logo,
- `rtds_document.bmp` : Document logo,
- `rtds_file.bmp` : File logo,
- `rtds_folder.bmp` : Folder logo,
- `rtds_hmsc.bmp` : HMSC logo,
- `rtds_macro.bmp` : Macro logo,
- `rtds_msc.bmp` : MSC logo,
- `rtds_package.bmp` : Package logo,
- `rtds_procedure.bmp` : Procedure diagram logo,
- `rtds_process.bmp` : Process diagram logo,
- `rtds_process_type.bmp` : Process class diagram logo,
- `rtds_project.bmp` : PragmaDev Studio Project logo,
- `rtds_protogui.bmp` : Prototyping logo,
- `rtds_service.bmp` : Service diagram logo,
- `rtds_system.bmp` : System diagram logo,
- `rtds_use_case.bmp` : Use case diagram logo,
- `rtdsRDP2Text4Reqtify.exe` : Stand alone program that extracts information from PragmaDev Studio project file and generates an information file readable by Reqtify.

1.17.3.2 Installation for Reqtify before version 4.0

This paragraph make the assumption that the Reqtify toolset is installed in `%REQTIFY_HOME%` directory. The procedure is to copy the files:

- `rtds.br`
from `%RTDS_HOME%\share\3rdparty\Reqtify\`
to `%REQTIFY_HOME%\config\otscript`
- `rtds.types`
from `%RTDS_HOME%\share\3rdparty\Reqtify\`
to `%REQTIFY_HOME%\config\types\design`
- All `bmp` files
from `%RTDS_HOME%\share\3rdparty\Reqtify\`
to `%REQTIFY_HOME%\config\images`

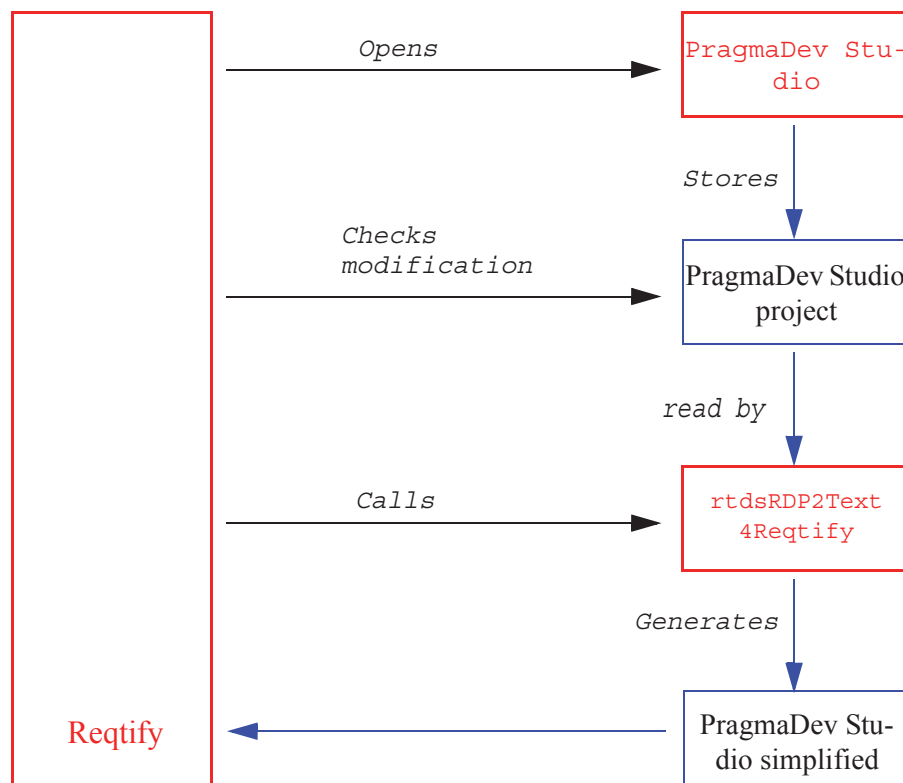
`rtds.types` defines

- the new *PragmaDev RTDS* type of analysis,

- how to handle an PragmaDev Studio project : in that integration Reqtify is told via the `rtds.br` behavior file to call the `rtdsRDP2Text4Reqtify.exe` program that will generate an information file understandable by Reqtify,
- how to analyse the generated file with coverage and requirement information.

1.17.3.3 General architecture

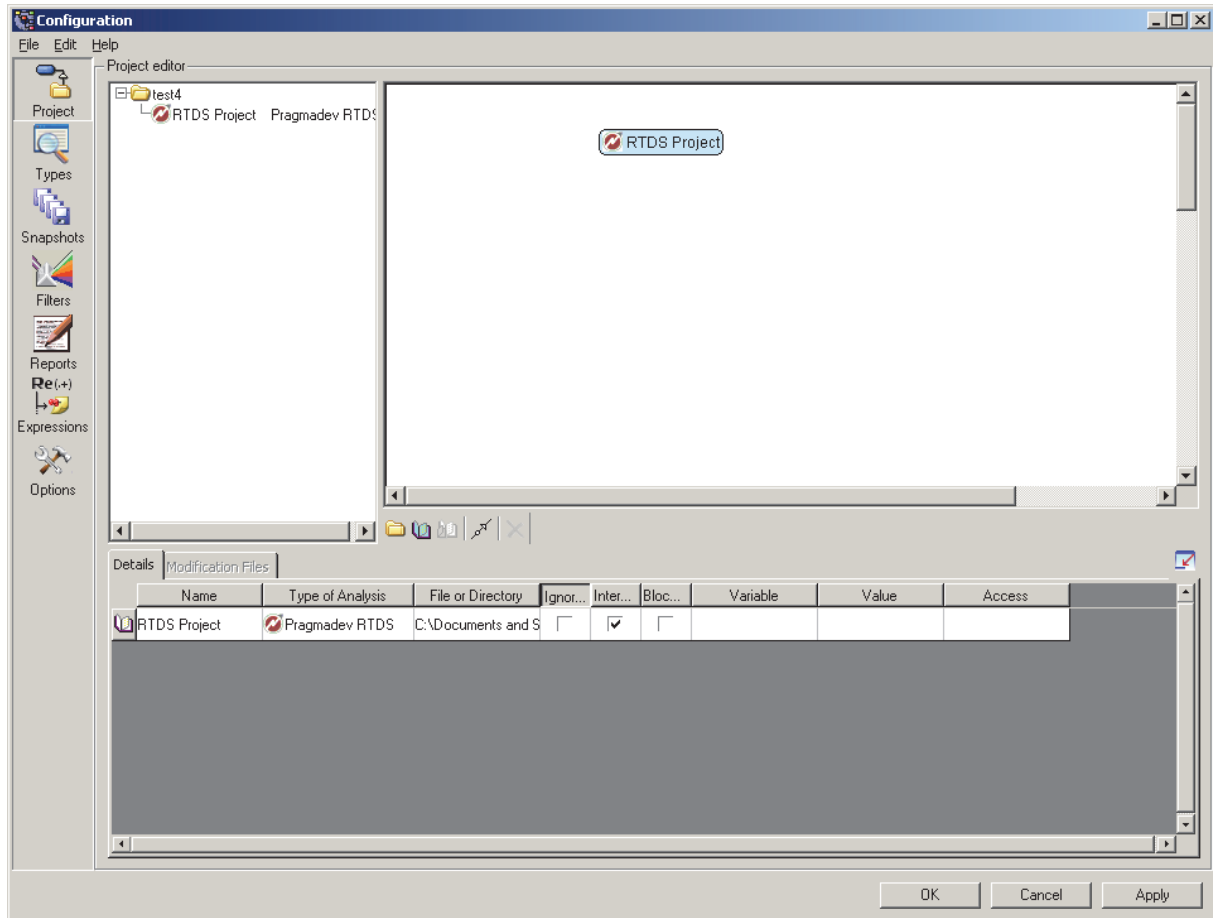
Reqtify relies on these previously installed files to work as described in the following diagram:



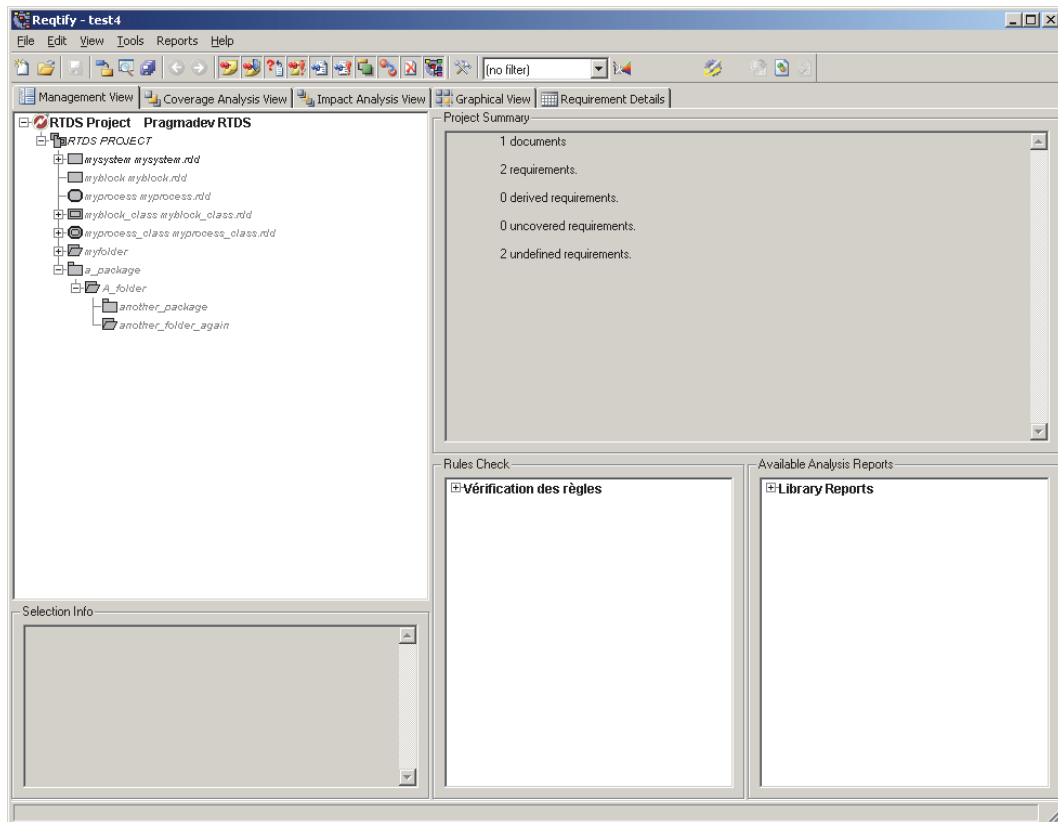
An PragmaDev Studio diagram can be opened directly from Reqtify. Reqtify also checks if the PragmaDev Studio project file has been modified. If the file has been modified, it calls `rtdsRDP2Text4Reqtify` utility that generates a project file understandable by Reqtify.

1.17.3.4 Usage

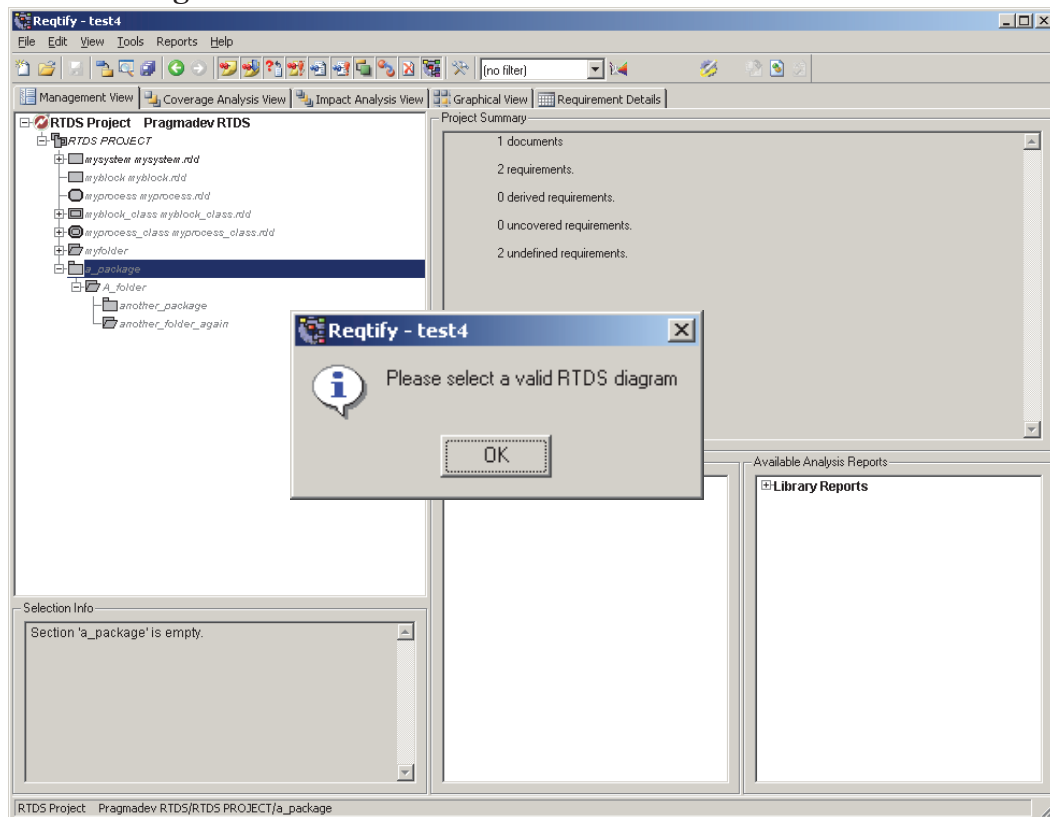
It is now possible to add an PragmaDev Studio project into Reqtify and to set its type to *PragmaDev RTDS* as shown below :



Click Ok and Reqtify will display the PragmaDev Studio project architecture as shown in the example below :



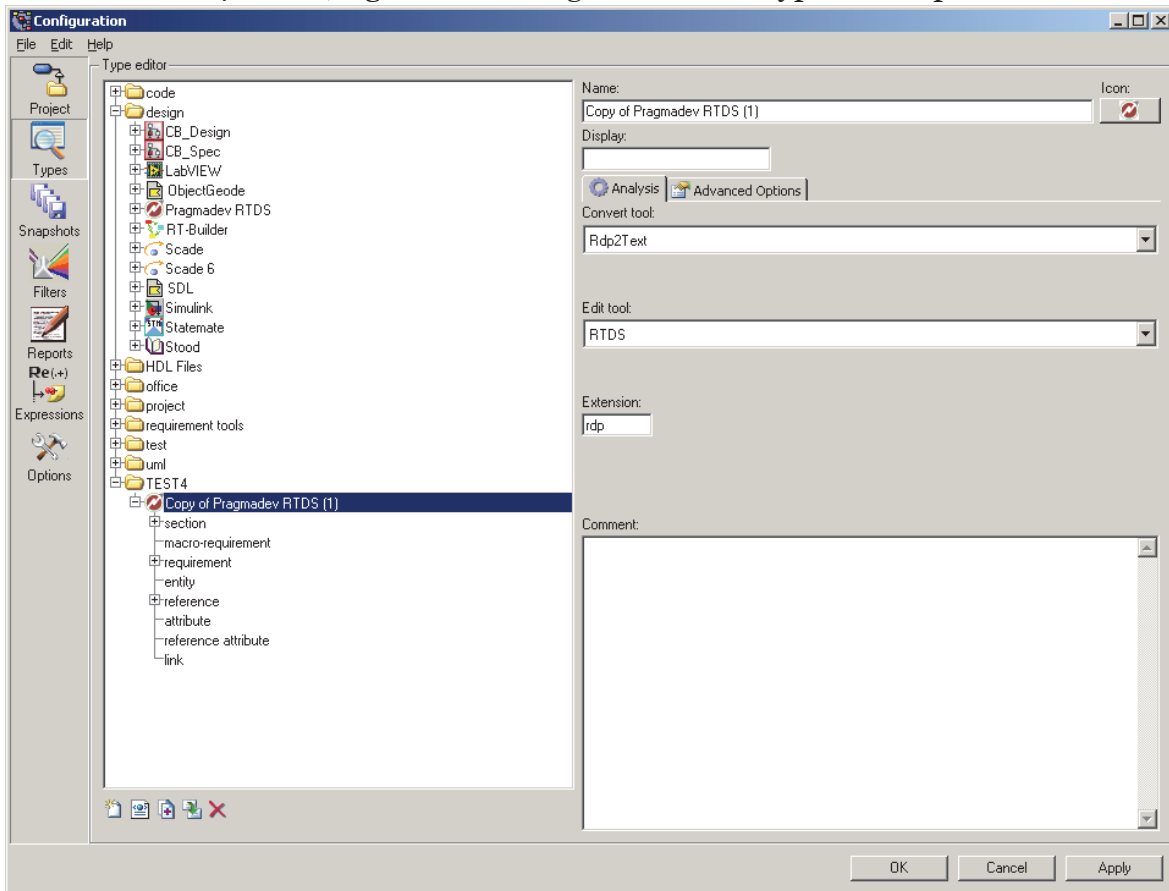
Clicking on an element of the PragmaDev Studio project will open that element in PragmaDev Studio. If not a valid component (package or folder) a warning message will ask to select a valid diagram.



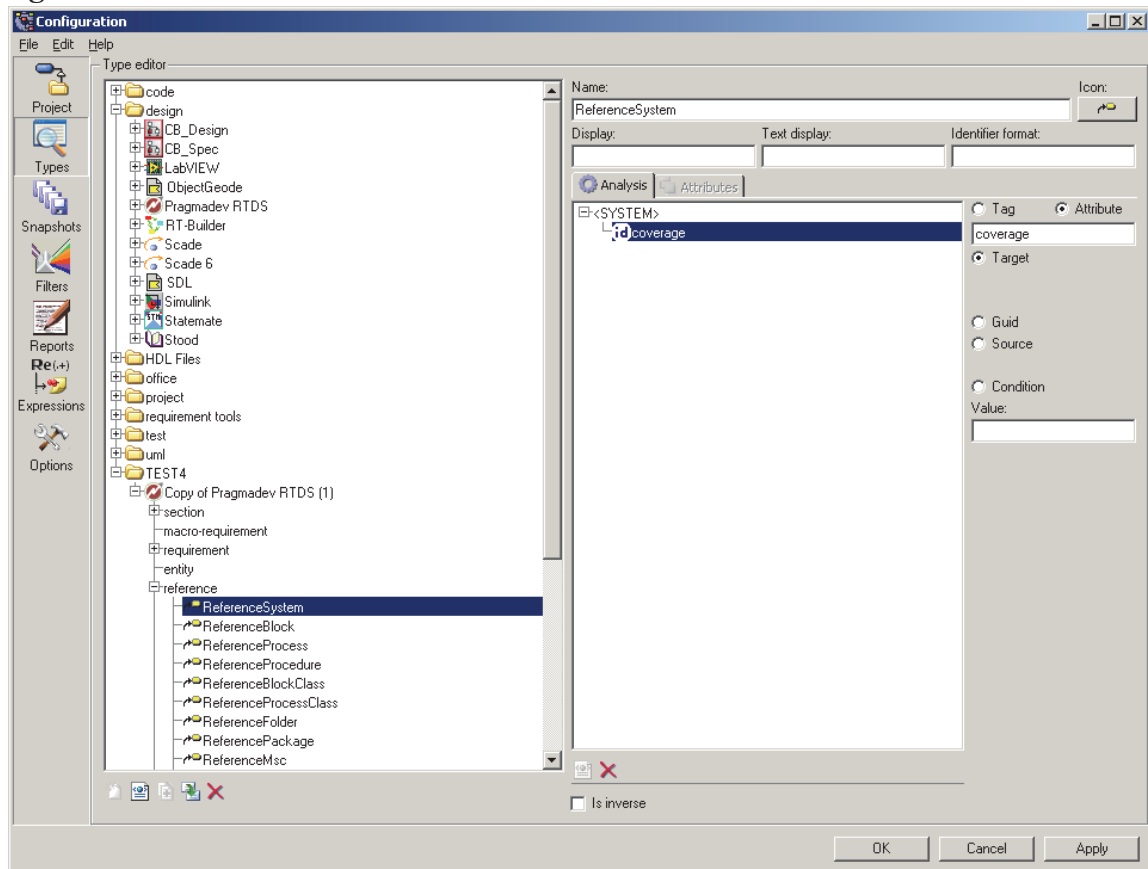
1.17.3.5 Format of traceability information

By default the traceability information for all component has the following format :
cover='<REQUIREMENT_IT_COVER>' requirement='<REQUIREMENT_TO_COVER>'

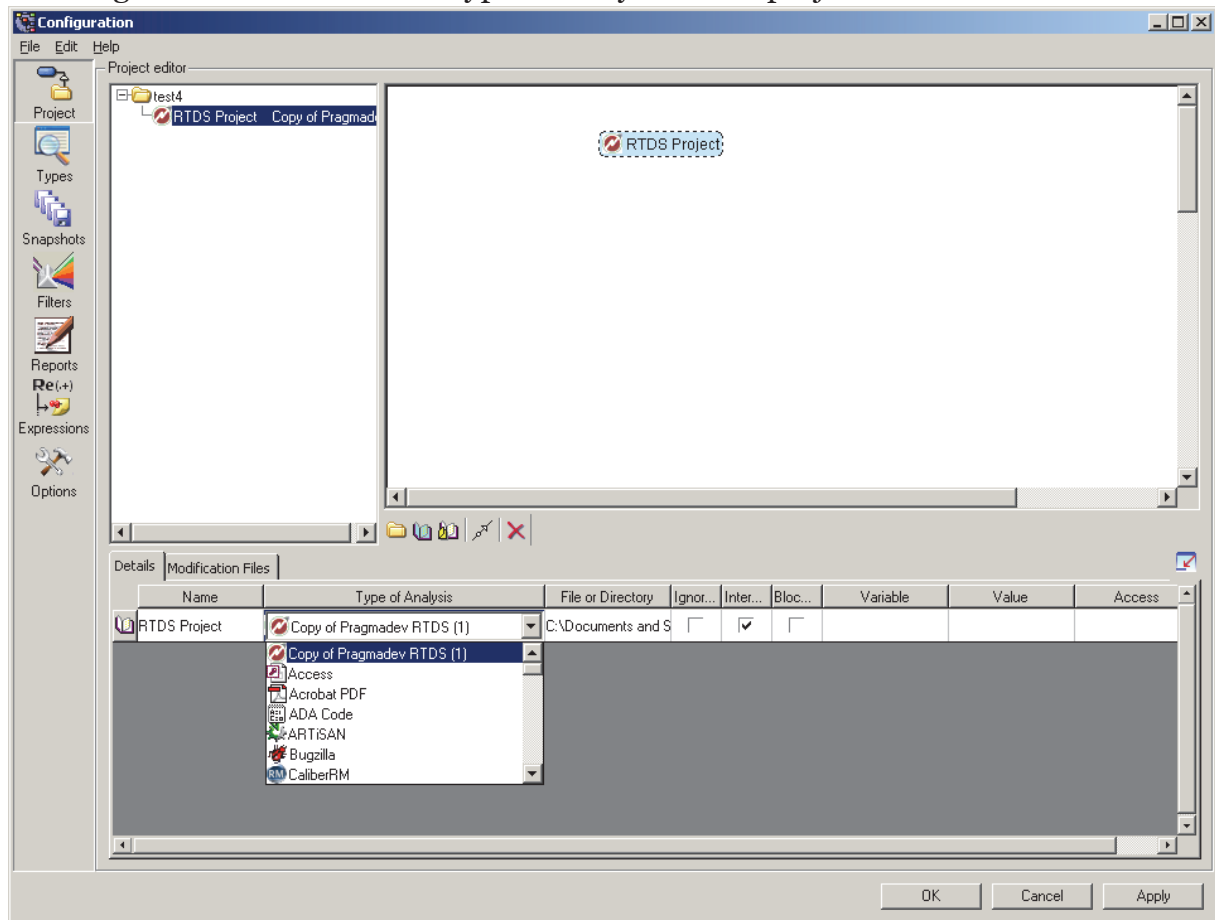
This can be customized with Reqtify type editor. For example would you like the traceability information to have the format coverage='<REQUIREMENT_IT_COVER>' for the SYSTEM component, right click on *PragmaDev RTDS* type and duplicate it :



Open the reference section of the SYSTEM component named ReferenceSystem to change the name of the attribute as shown below :



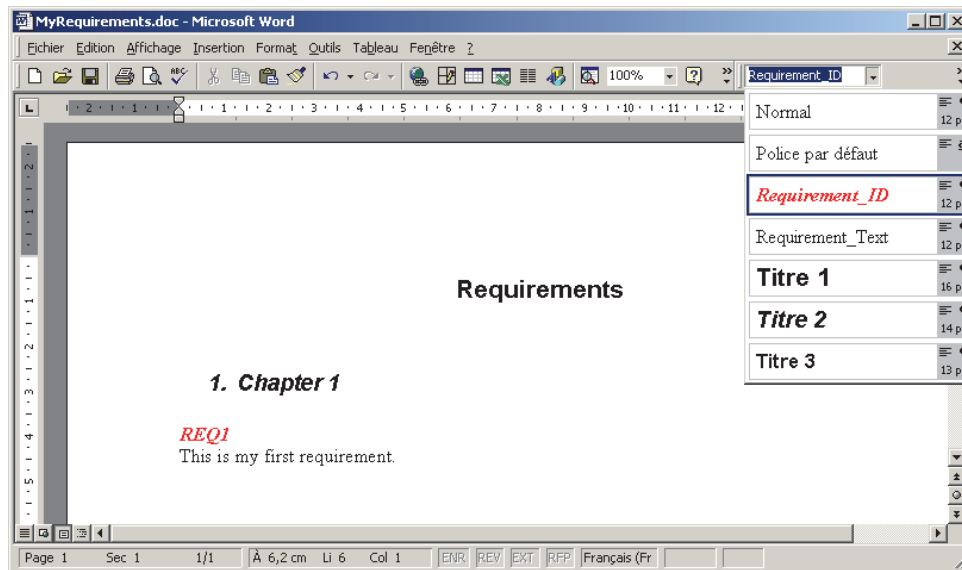
Before validating this new type, the project editor needs to know that the type of analysis has changed. To do so select this type of analysis in the project editor and click OK :



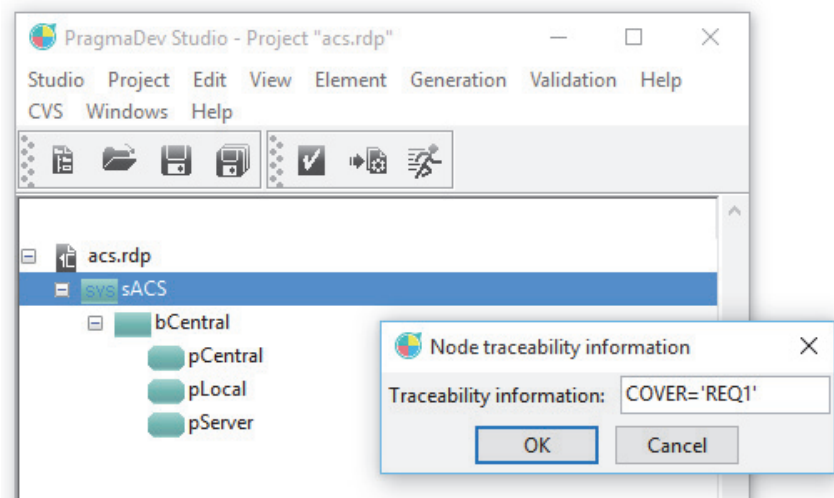
1.17.3.6 Example

The following example creates a link from a Word document to an PragmaDev Studio Project with the default Reqtify syntax.

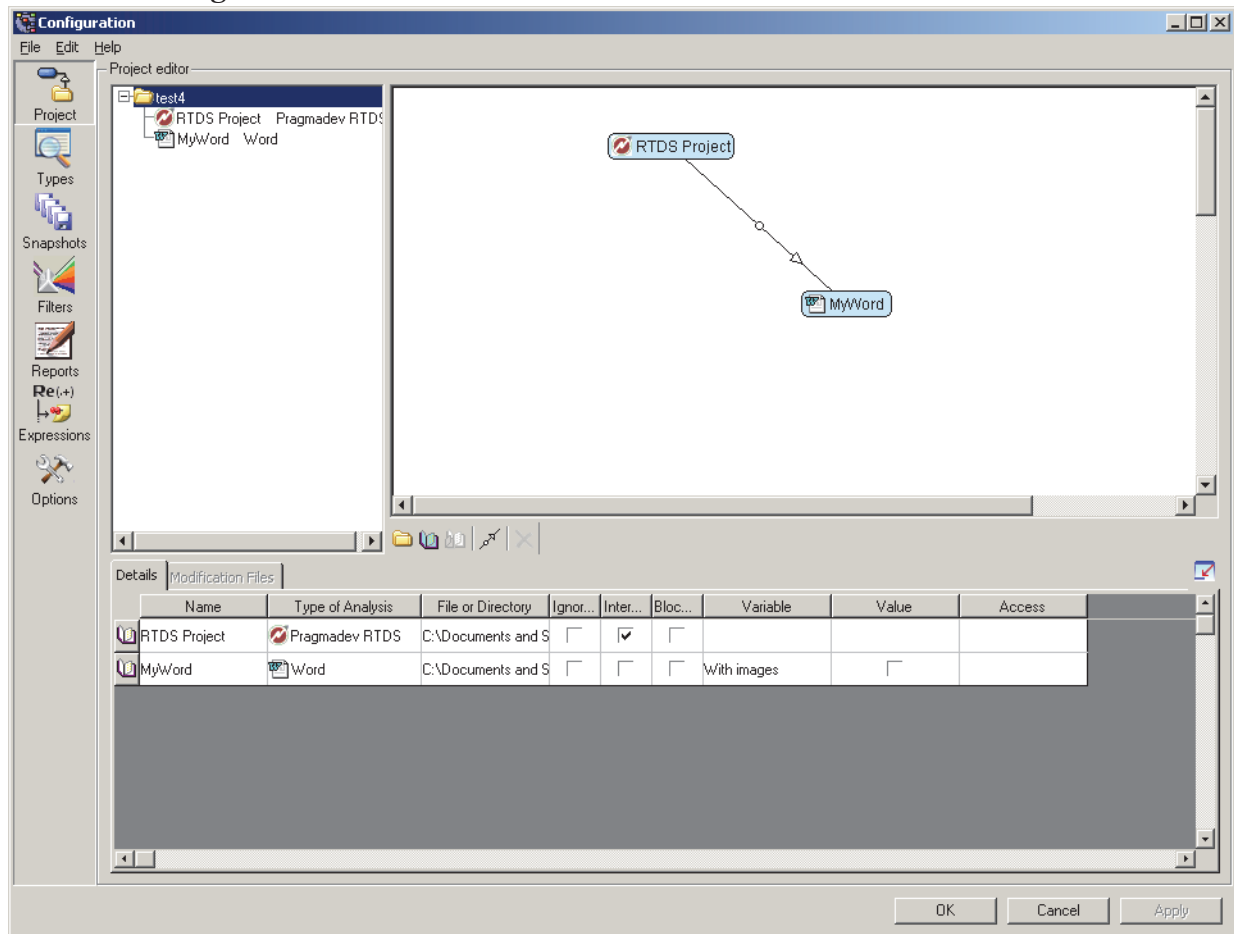
In a Word document, the requirement identifier REQ1 is defined with *Requirement_ID* style.



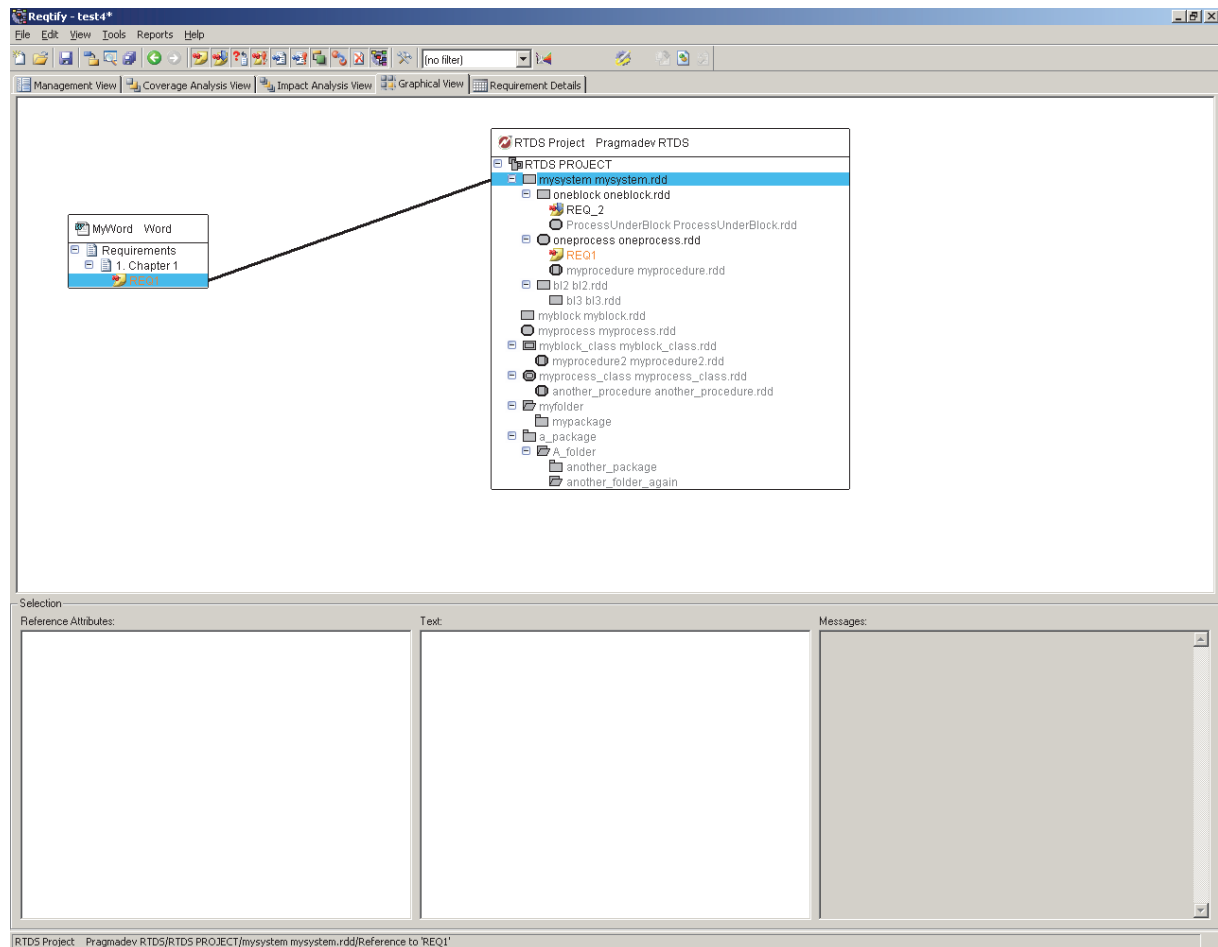
Within PragmaDev Studio select the element covering the requirement and go to the *Element / Traceability information...* menu. In the *Node traceability information* popup window, indicate which requirement is covered.



Add both documents in the Reqtify Project Editor with their respective analysis type and create a coverage link from one document to the other :



After analysis, Reqtify will show the links in the Graphical View editor :

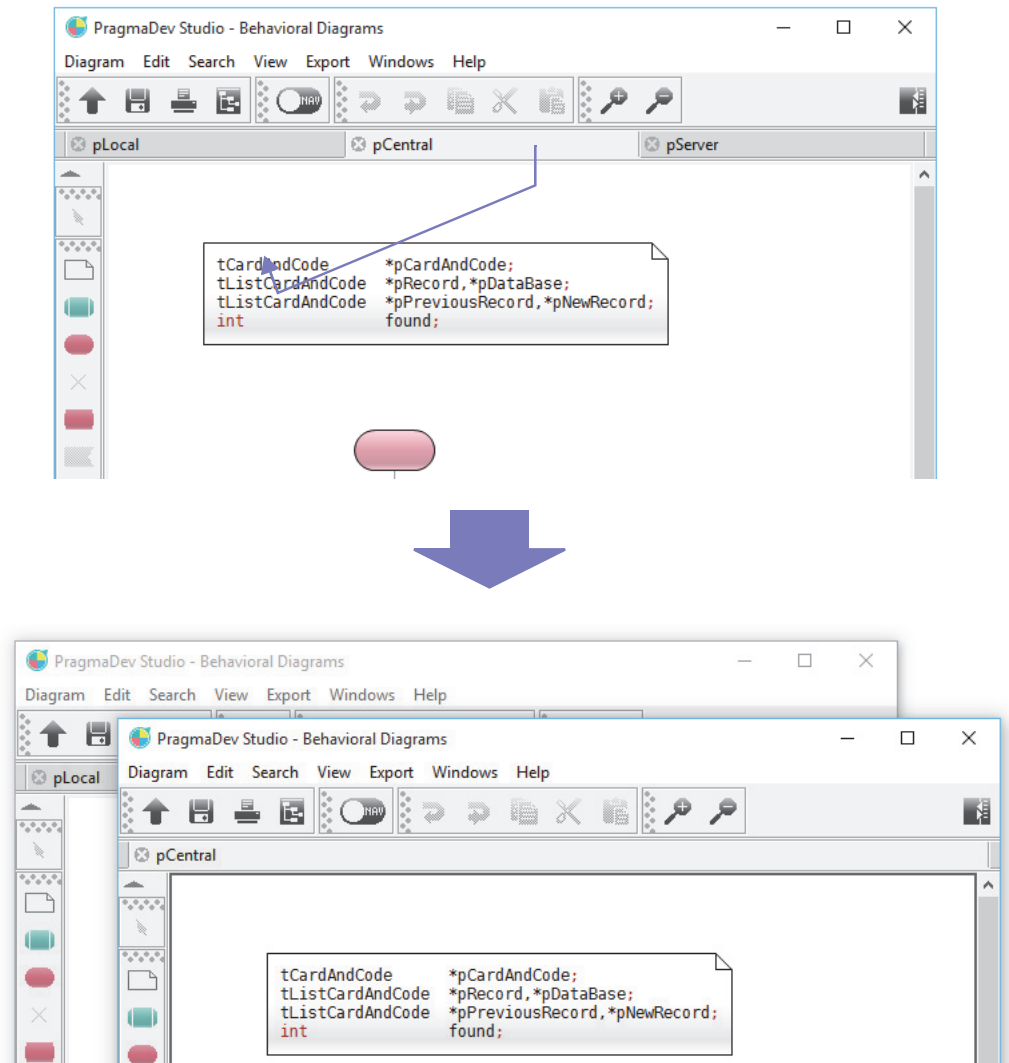


2 - Editor windows

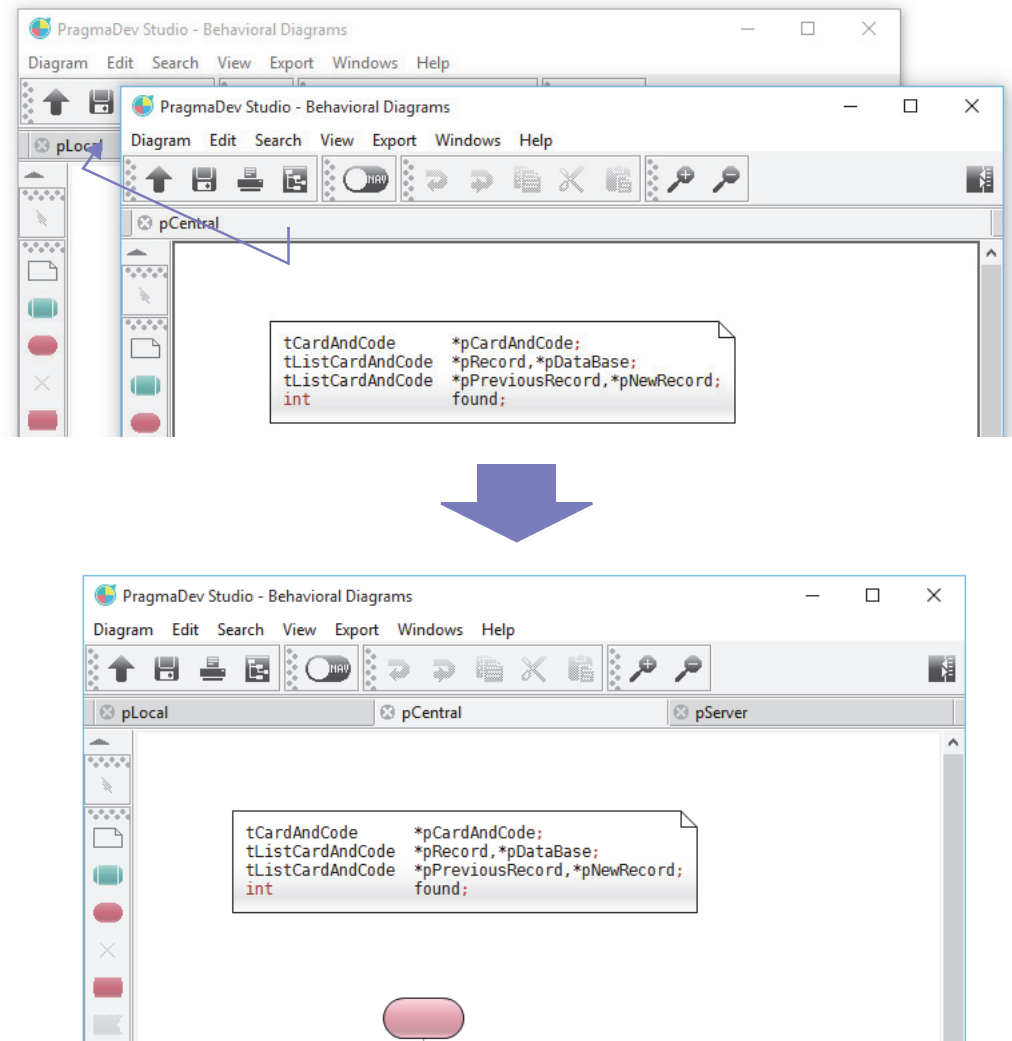
2.1 - Tab management

Most windows are organized into tabs: if a new element is opened from the project manager, it will open a new tab in an existing editor window for this kind of elements.

If needed, a tab can be dragged out of its parent window to create a new window:



A tab can also be moved from one window to another, just by dragging it from its former parent window's tab bar and dropping it in its new parent's:



If the former parent window had only one tab, it will be closed automatically.

The default tab order in all tab bars is given in PragmaDev Studio general preferences (see "General preferences" on page 33). It can be alphabetical, last opened first, or last opened last (the default). It is also possible to allow tab reordering, in which case a tab can be moved within its parent tab bar to put it in a new position. Note that tab reordering does not work well with the alphabetical tab order, as tabs will be reordered automatically each time a new element is opened in the same window.

2.2 - Windows menu

All windows in PragmaDev Studio have a menu labelled "Windows". This menu allows to perform common operations on windows, as well as to navigate between the opened ones.

The first part of this menu contains the following entries:

- *"Remember position and size..."* records the position and size of the current window and applies both automatically for all new windows of the same kind;
- *"Auto-place..."* forgets any recorded position and size for the windows of the same kind as the current one and reverts to the default behavior, which is to let the window manager place any new opened window.

These choices are actually recorded in PragmaDev Studio preferences file, so they will apply even after PragmaDev Studio is closed and reopened.

The last part in the *"Windows"* menu contains entries for all opened windows in the current session:

- If a window has tabs, a cascade menu entry will be created in the menu with the name of the editor window kind (e.g. "Diagram editor"). The sub-menu for this entry will have one entry for each tab in the window, with the file name for the opened diagram. Selecting this entry will raise the corresponding editor window, and select the specified tab.
- If a window does not have tabs, it will have a single entry in the menu, with the name giving the window type and the element displayed in it. Selecting this entry will raise the corresponding window.

Note that the project manager also always has an entry in this menu, which is always the first and is named "Project", followed by the name of the currently opened project.

During debug sessions, there is another entry placed between the first and last parts of the *"Windows"* menu in all diagram editor windows. It is named *"Set current tab as target for debugger"*. This entry will force all diagrams opened from the debugger to appear in the current tab. This tab will be automatically renamed to add a prefix "[D]" before the diagram name, and will be moved to the first position in the window's tab bar. Then, each time the execution stops in the debugger on a symbol in a diagram, the diagram will be opened in this tab, replacing the one that was already in it. This avoids to end debug sessions with a lot of tabs. Note that if a diagram is already opened in another tab or another window, it won't be moved to the debugger target tab; its window will just be raised and its tab selected.

3 - Diagram editor

It is possible to have two types of diagrams in PragmaDev Studio. Normal mode diagrams are diagrams made with PragmaDev Studio, and legacy mode diagrams are diagrams made with older version of PragmaDev Studio tool (called Real Time Developer Studio) and opened with PragmaDev Studio.

The diagram editor is the window where all types of diagrams may be edited. The editor window is the same for all types of diagrams, and its basic features are the same. It may however have a few extra features depending on the type of the displayed diagram.

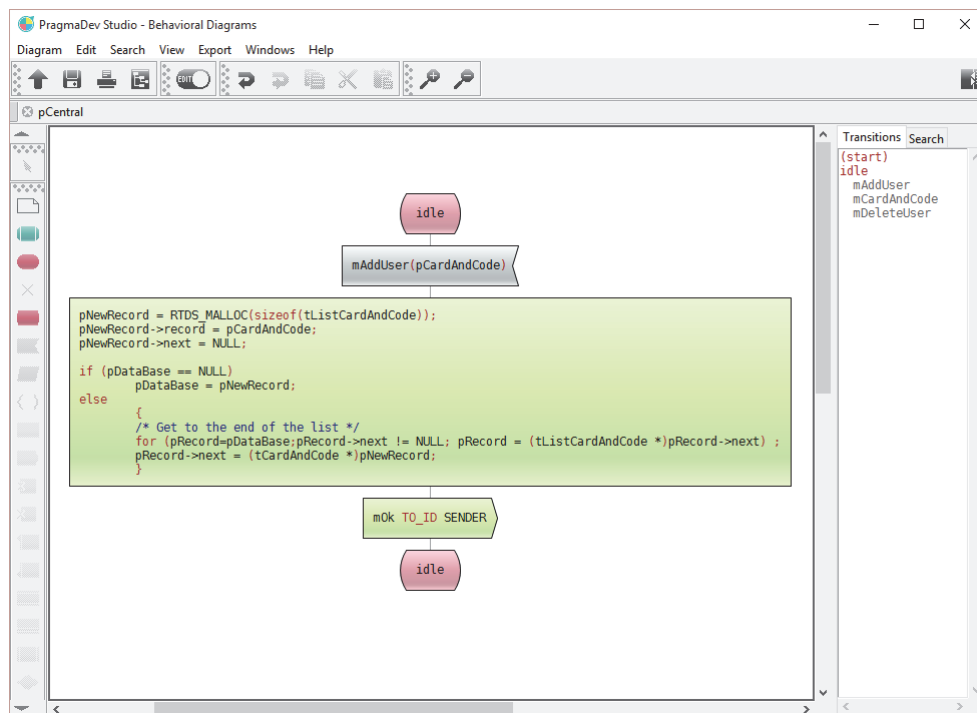




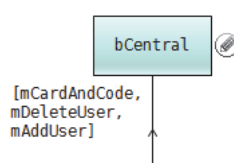
Diagram editor window

3.1 - Common features

3.1.1 Editor modes

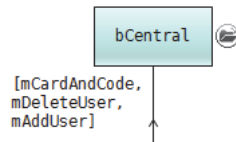
Most editors have two modes, that can be switched either via two options in the “View” menu, or via the  /  button in the toolbar:

- In navigation mode, symbol texts are only editable via a button appearing when the mouse pointer hovers over the symbol:



Double-clicking anywhere on the symbol opens the diagram associated to it if any.

- In edition mode, symbol texts are editable directly by double-clicking on the text. When the mouse pointer hovers over the symbol, another button may appear, allowing to open the diagram associated to the symbol if there is one:



The button will only appear if there is an associated diagram to the symbol, or if there can be one, even if it doesn't exist yet.

Note that in the UML class and deployment diagram editors, only the main text for symbols have an associated button. Attribute and operation texts are always edited by double-clicking on them.

3.1.2 Frame concept

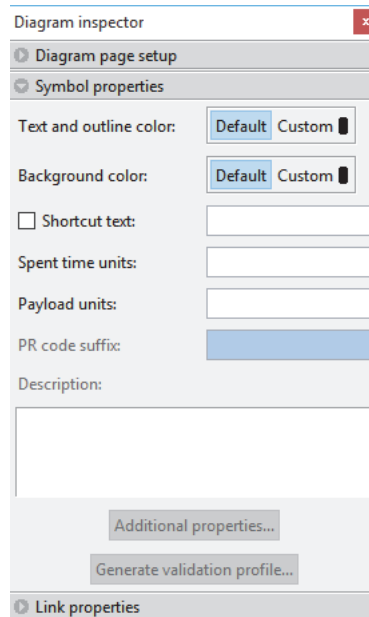
In legacy mode, all diagrams have a surrounding frame, containing all the symbols in the diagram. You can't put symbols outside that frame. For SDL system and block diagrams, the frame also represents the external boundary of the agent, and channels may be connected to it.

In normal mode, process diagrams have not surrounding frame.

3.1.3 Symbol and link properties

Each symbol or link has a property sheet allowing to enter all its features in a guided way. The actual properties depend on the type of the symbol or link. It may be opened by selecting a symbol or link and choose "Properties..." in the contextual menu.

The basic property sheet for a symbol is the following:



The image shows a 'Diagram inspector' dialog box with a 'Symbol properties' tab selected. The dialog contains several configuration options: 'Text and outline color' with 'Default' and 'Custom' buttons; 'Background color' with 'Default' and 'Custom' buttons; a 'Shortcut text' checkbox and text field; 'Spent time units' and 'Payload units' text fields; a 'PR code suffix' text field; and a 'Description' text area. At the bottom, there are buttons for 'Additional properties...', 'Generate validation profile...', and 'Link properties'.

- Text and outline color allows to select the color of the text in the symbol and its outline.
- Background color allows to select the background color of the symbol.
- The symbol shortcut text may be used to specify an alternate text to display in the symbol and to open its "real" text in an external editor. It may be used for symbols with a very long text to avoid taking too much space in the diagram. If this shortcut text is set, it will appear in the symbol instead of its actual text with a specific presentation, depending on the option chosen in the diagram preferences:
 - If the "*Prefix for shortcut text instead of underlining*" is not checked, it will appear underlined:

Declarations

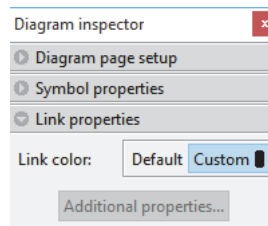
- If the "*Prefix for shortcut text instead of underlining*" is checked, it will appear with a prefix:

» Declarations

Double-clicking on the symbol shortcut text will open the external editor showing the symbol actual text.

- The PR code suffix is only available for symbols declaring a SDL agent. This text will be inserted after the agent declaration in exported PR files whenever the actual agent is not defined. This is used to keep Geode-style external references in PragmaDev Studio diagrams; It should usually be left empty.
- The symbol description is only used for documentation purposes.
- Spent time units and Payload units are values assigned to a symbol relevant for performance analysis (see "Performance Analyzer" on page 295).

The default property dialog for a link only includes its color:



Please refer to the paragraphs describing the editors for examples of property sheets for specific link types.

3.1.4 Moving symbols

3.1.4.1 Normal mode

The symbols in a diagram may be moved by using the mouse. It is not possible to move only one symbol, but only a state with all its transitions by moving the state itself. It is possible to invert transitions order of a state by moving the first symbol or the transitions.

3.1.4.2 Legacy mode

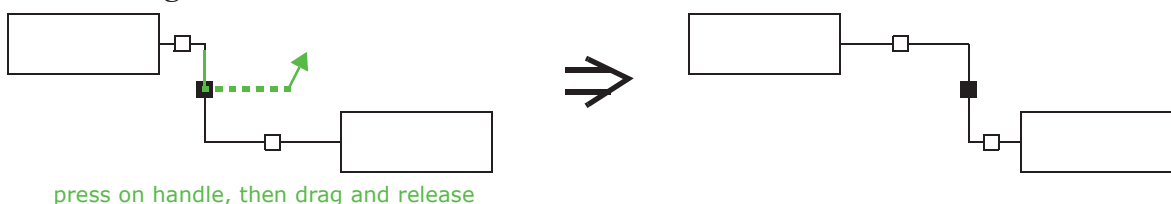
The symbols in a diagram may be moved by using the mouse or via the arrow keys: if a group of symbols is selected, pressing the arrow keys will move the symbols by one grid cell. Pressing the arrow keys while maintaining the Control key pressed will move the symbols by one point.

3.1.5 Modifying links

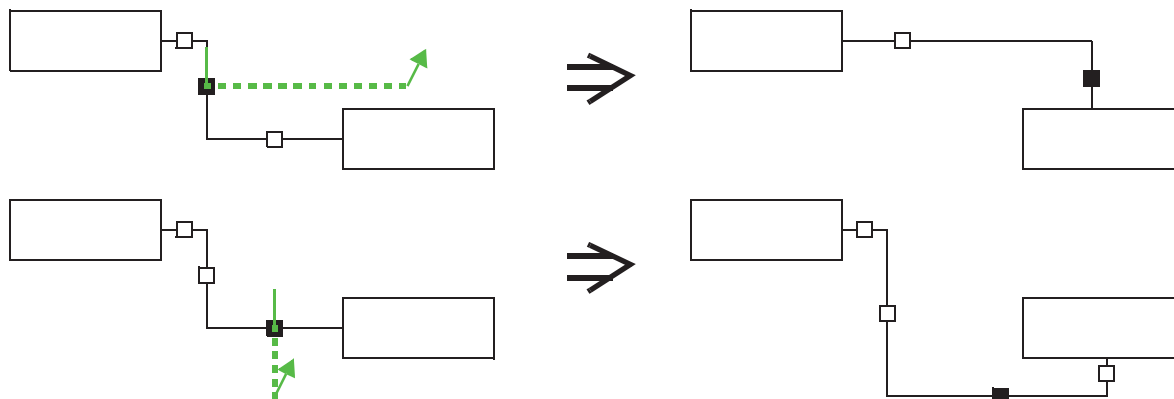
Modifying links is only available in legacy mode diagrams.

There are two ways of modifying existing links in the diagram editor:

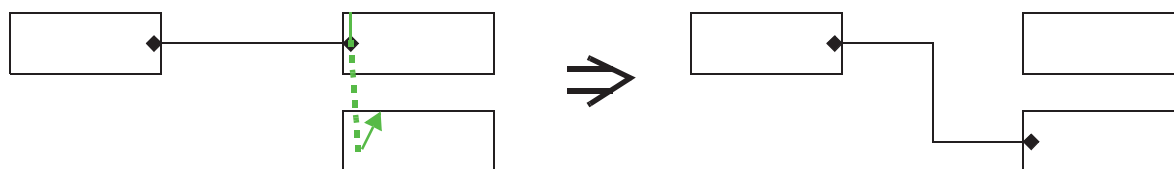
- For broken links, the segments may be moved by using the handle appearing in the segment middle:



If needed, segments will be automatically created or deleted:



- For all links, diamond-shaped handles at both ends allow to change the symbol to which the link connects:



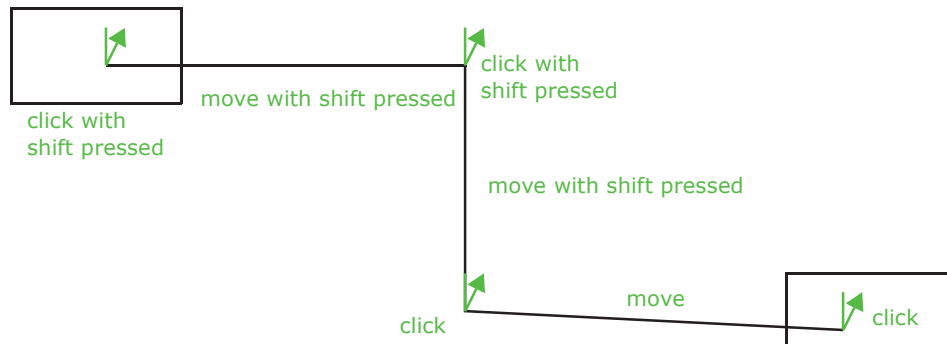
press on handle, then drag and release


3.1.6 Button and tool bars


As seen in the screen shot above, the diagram editor window has several button bars and tool bars. The button bars give access to all usual operations. The tool bars are at the left of the window and are separated in 3 parts:

- The top tool bar gives access to the general operations on the contents of the diagram. It's usually restricted to the selection tool, but MSC diagrams have a few extra tools (see "Specific tools" on page 77).
- The middle tool bar is the symbols tool bar. It allows to insert new symbols in the diagram:
 - In legacy mode, this insertion is made by clicking on the button corresponding to the symbol you want to insert, then clicking within the diagram's frame where you want to insert the symbol.
 - In normal mode, only start, state, label and declarations symbols can be inserted like above. All other symbols have to be inserted after an existing symbol. This insertion is made by selecting a symbol, and clicking on the wanted symbol in the tool bar. When a symbol is selected in the diagram, not all symbols are available in the tool bar, but only symbols that can be added after the selected symbol.
- The bottom tool bar (only available in legacy mode) is the links tool bar. It allows to insert new links between symbols within the diagram. There is a button for each link type allowed in the diagram. Make sure you select the right type for the symbols you want to link, or the insertion may be refused. The link insertion is made by clicking on the button corresponding to the link type you want to insert, then pressing the mouse button over the first symbol and dragging to the other symbol. If you want to manually indicate a path between

the two symbols, shift-click on the first symbol, then shift-click on each corner for the link. Holding the shift key allows to restrict the move to horizontal or vertical lines. For example:



In legacy mode, above the toolbars is a tiny lock icon that looks like this: . When this icon is not active, the button inserts a single symbol or link: Once one symbol or link has been inserted, the editor returns to selection mode. Clicking once on the icon makes it active. After that, any insertion button will be "sticky": Any number of symbols or links may be inserted in a row. To go back to "non-sticky" mode, press the lock icon again or select the selection tool.

The button and tool bars have a common aspect and behavior: By default, both have a header looking this: . It is at the left for button bars and at the top for tool bars. By clicking on this header and dragging it away, the bar can be detached from its parent window and displayed in its own window.



Symbol tool bar for blocks

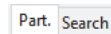
To put the tool bar back in its parent window, just close the tool bar window.

Note you can turn this feature off for tool bars if you don't need it in PragmaDev Studio preferences, 'General' tab (see "General preferences" on page 33). Button bars will however always be detachable.

3.1.7 Partitions

Large diagrams can be split into partitions. A partition is a set of pages that may contain any number of symbols and links. Partitions may be added, deleted and printed via the "Diagram" menu.

Navigating through partitions can be done by using the left-most toolbar in a diagram window or with the partition browser, displayed at the right of the diagram. If several browsers are available, the partition browser can be selected via the "Part." tab at the top of the browsers:

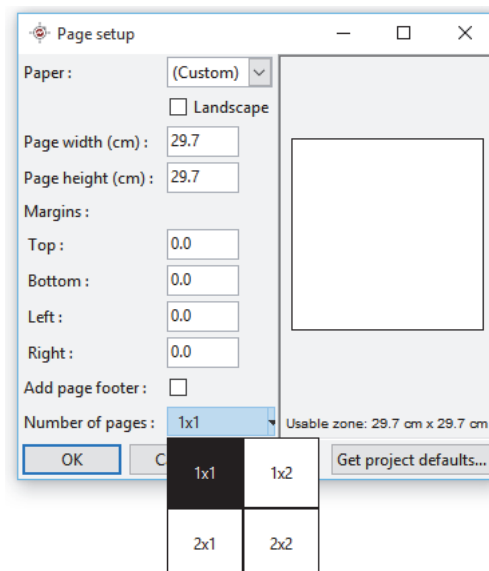


Notes:

- A partition can only be deleted if it doesn't contain any symbol or link.
- Partitions are neither available in MSC diagrams, or in normal mode behavioral diagram

3.1.8 Page setup

To ensure a WYSIWYG behavior, all partitions in a diagram include a page setup that will be the one used when printing it. This page setup may be edited via the "Partition page setup..." item in the "Diagram" menu in diagram editors. The following dialog appears:



The fields "*Paper*", "*Landscape*", "*Page width (cm)*" and "*Page height (cm)*" are used to specify the paper size. The "*Margins*" are automatically removed from the usable area for the partition and when printing. If the "*Add page footer*" option is checked, a footer will be displayed on each printed page, containing the name of the printed file and the page number. The height of this footer is also removed from the usable area for the diagram. This usable area is displayed with its dimensions in the right part of the dialog.

The field "*Number of pages*" one is used to specify the number of rows and columns of pages in the diagram.

A default page setup can be configured in the project manager (menu "File", "Page setup..."). Clicking the button "Get project defaults" in any page setup dialog will select this default page setup.

There is a shortcut to add a row or column of pages in diagrams: along each side of the edge pages, the following buttons appear:



These buttons will add a page column or row to the diagram respectively.

Notes:

- These options are not available for normal mode behavioral diagram.

3.1.9 Publications

3.1.9.1 General presentation

It is possible to attach to any diagram a set of publications. A publication is a set of symbols that will be exported as one or several external image file(s). These publications are dynamic: The set of exported symbols is remembered and you can re-export them at any time, or ask to have them exported automatically when you save the diagram (see paragraph about diagram preferences in “Diagram preferences” on page 28).

These publications can be used by importing them in any word processing software that has an "insert with link" or "import by reference" function. This function allows to insert a file into the current document, but keeps a reference to the inserted file so that any modification to the inserted file will be reflected automatically in the document. Since PragmaDev Studio will update the publications after each diagram modification, it will ensure that the contents of the document importing them will always be up to date.

Notes:

- When saving a diagram having publication, a dialog may appear asking whether to update the publications. This behavior is controlled by the "Update publication" option in the diagram preferences (see “Diagram preferences” on page 28).
- On Windows, the selected symbols may also be directly copied to the Windows clipboard by using the item "Copy as bitmap" in the "Edit" menu (shortcut: Shift-Ctrl-C). This feature is not available on Unix platforms.
- The publications may also be used in documents written using a markup language like SGML, XML or HTML. For example, with HTML, images may simply be imported via the tag `` without giving any dimension for the image so that they will always be read from the image file.
- Some word processors (e.g. FrameMaker) remember the size you gave to any imported graphics, even if these were imported by reference. PragmaDev Studio publications still work with these, but you may end up with distorted graphics in the final document if the size of a publications changes.

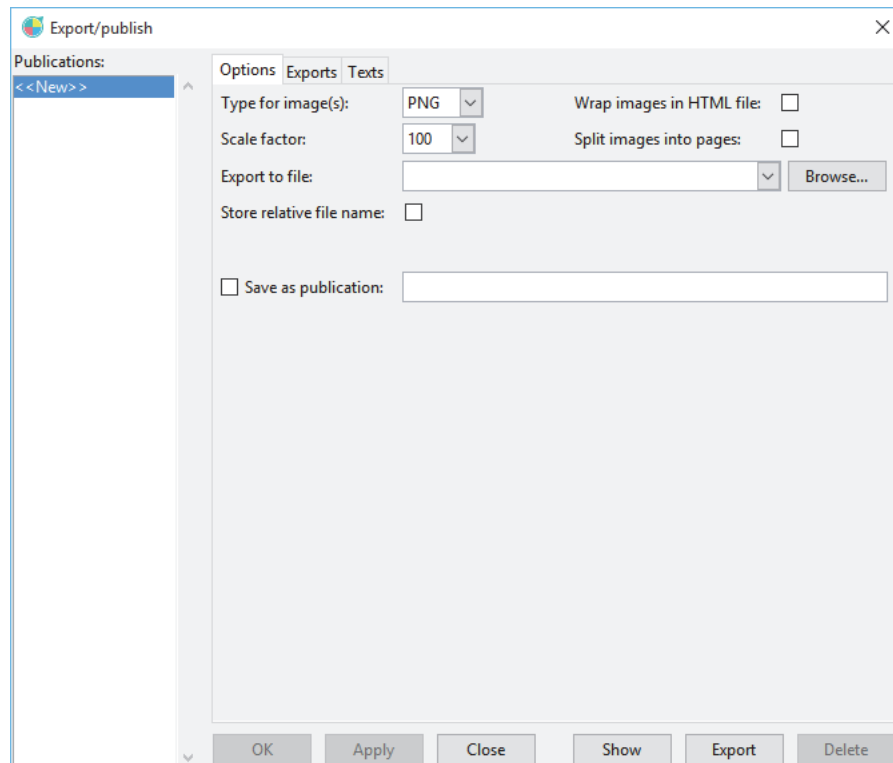
3.1.9.2 Creating a publication

There are currently 5 types of publications:

- Symbol publications will export a set of given symbols;

- Transition publications will export all symbols in a given transition, identified by its state and message input or continuous signal;
- State publications will export all symbols in all transitions attached to a given state symbol;
- Partition publication will export a whole partition;
- Diagram publications will export the whole diagram.

These publications are available via the entries "Export/publish ..." in the "Export" menu in diagram editors. All these entries open the following dialog:



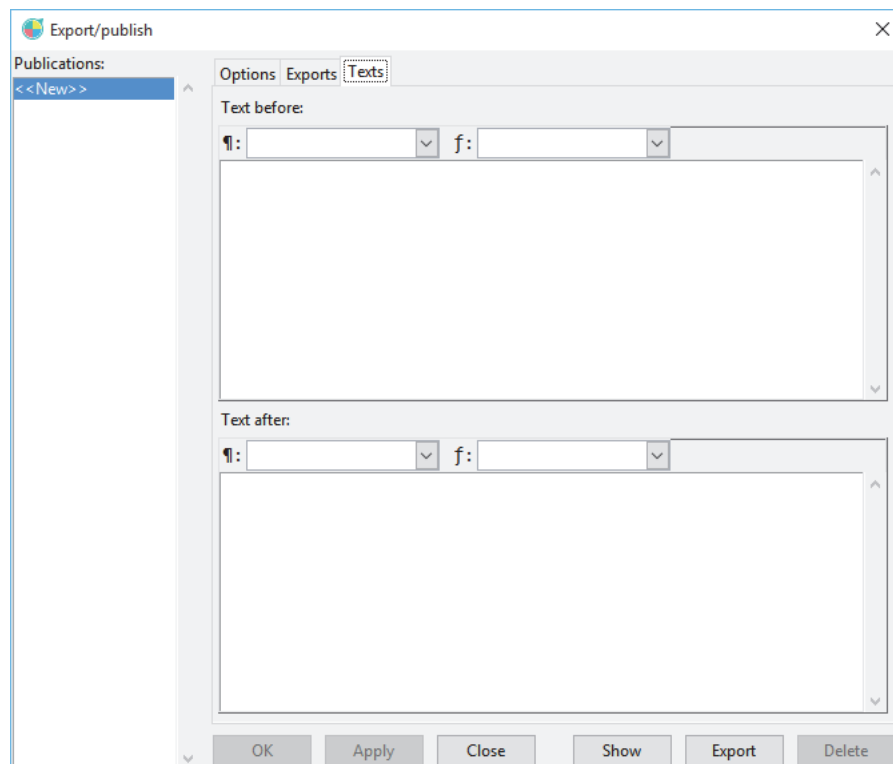
The dialog allows to set:

- The type for the exported image(s): PNG, JPEG, EPS for Encapsulated Postscript or CGM. Another type named 'Doc' is also available. It should be used for publications that are only used in PragmaDev Studio documents. These kind of publications don't actually export anything until the document they're included in is itself exported to a given format. For more information, see "Document editor" on page 99.
- Whether the exported image(s) should be wrapped in a HTML file. This should be used if the publication consists in several images or pages. Importing the HTML file in a word processing software allows to import the whole set of images in one operation, and to keep it up to date even if the grows or shrinks afterwards. Note that this option should only be used for PNG or JPEG images, as browsers usually can't display Encapsulated Postscript or CGM images.
- The zoom factor applied before exporting.

- For partition and diagram publications, each exported image contains by default a whole partition. Checking the "*Split images into pages*" option allows to export an image per printed page in the diagram or partition.
- The file name for the exported image; please note a suffix can be added to this name if the publication exports several images.
- Whether the file name for the publication should be remembered as an absolute pathname, or relative to the diagram file name.
- Whether the exported images should be saved in a publication or not. If the option is not checked, the export will be done one time, but not kept up to date with the diagram. If the option is checked, the publication will be saved with the name indicated in the text field.

3.1.9.3 Documenting a publication

The 'Texts' tab allows to associate texts with the publication. Clicking on it changes the dialog to show two text editors:

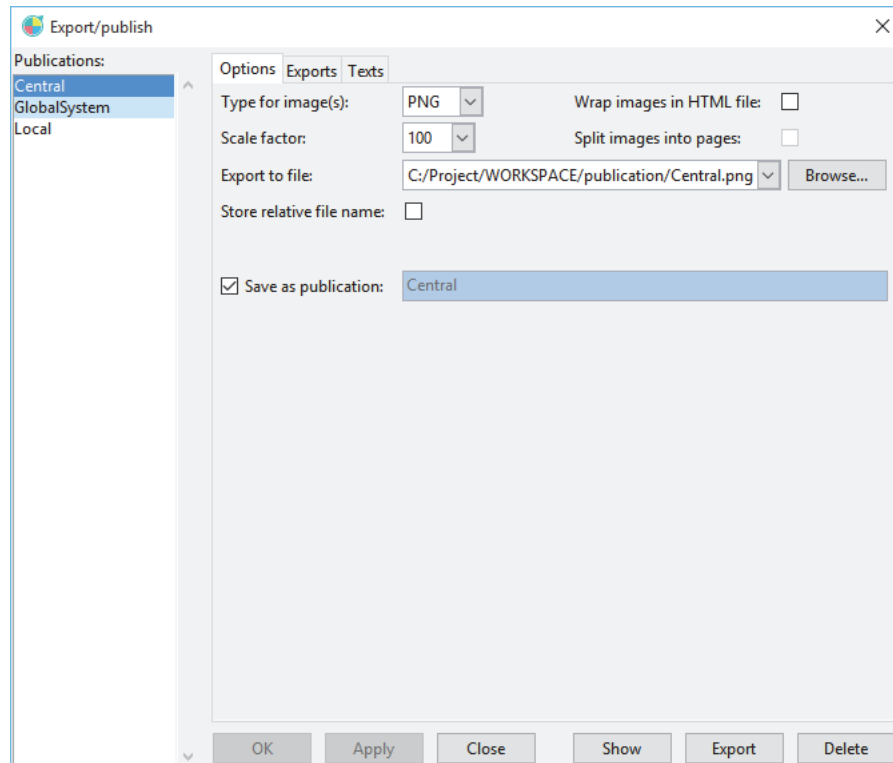


These editors are used to create or edit the styled texts that will automatically appear before and after the publication image when included in a document. For more information, see "Styled text editor" on page 111.

The list in the left part of the dialog contains the names for the already existing publications. Clicking on one of the names will display the attributes for the publication in the dialog as in the "Manage publications" dialog. It is possible to go back to the current one by clicking the '<<New>>' entry, which is always the first in the list. The actual export is done by clicking 'Apply' or 'OK'.

3.1.9.4 Managing publications

The last item in the "Export" menu is named "Manage publications..." and opens the same dialog as above, except it won't show a '<<New>>' entry:

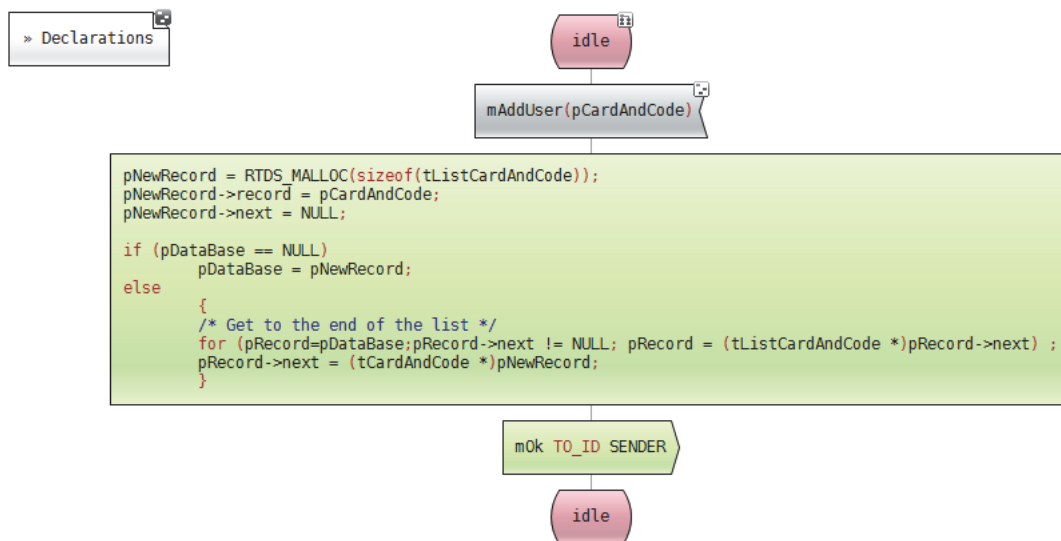


The zone in the dialog's left part lists the names for all publications in the current diagram. Clicking on one of the names will display the publication attributes in the right part, and allows to change some of them. The 'Show' button will display the exported part of the diagram; The 'Export' button allows to explicitly update an existing publication.


3.1.9.5 Documentation hints

PragmaDev Studio allows to display directly in a diagram which parts of it are documented via publications and basic information about the publications themselves. This is done via documentation hints, that are little icons appearing in the top right corner of symbols. These hints are turned off by default; To turn them on, select "Show symbol





documentation hints" in the "View" menu in the diagram editor. The hints are then shown in the diagram editor as follows:



There are two types of documentation hints that can appear in the top right corner of symbols, each with two variations:

-  indicates that the symbol itself is exported;
-  indicates that the symbol is exported with all the symbols that logically follows it. This symbol can only appear in behavioral diagrams. On a state symbol, it means that all the transitions attached to this state symbols are exported; On an input, continuous signal, start or connector in symbol, it means that the whole transition after it is exported.

The two variations of each hint indicate whether there are texts recorded in the associated publications:

- If the hint is white ( or ) , the associated publication doesn't contain any text;
- If the hint is grey ( or ) , the associated publication has some text associated.

This allows to visually identify the publications needing documentation, for example in the case of automatically generated publications (see "Full documentation generation" on page 102). When documentation hints are displayed, clicking on one of the hints automatically opens the publications dialog with the corresponding publication dialog (see "Managing publications" on page 65).

3.2 - SDL editor features

3.2.1 Creating and opening components

Some diagrams may contain symbols that are defined via another diagram. E.g., block and system diagrams may contain process and block symbols that will be described via process and block diagrams. These diagrams are displayed as children of the first diagram in the project tree in the main window (see “Project” on page 7).

The definition diagrams for symbols are not added to the project as described in the paragraph “Adding components to the system” on page 12, but as follows:

- Select the symbol for which you want to create the definition diagram;
- Select the item "Open definition..." in the contextual menu or the "Edit" menu or double-click on any part of the symbol, except its text (hint: the cursor should be an arrow, not a text caret);
- If the definition diagram for the selected symbol doesn't exist, the project manager will pop up and ask if you want to create it. If you answer "OK", the "Add component" dialog described in “Adding components to the system” on page 12 will allow you to enter the features for the new node.
- The definition diagram will then appear in the current window if it's reusable, or in another one if it's not.

You may open definition diagrams for the following symbol types:

- Block and process symbols in block or system diagrams, including instances of block or process types;
- Block types and process types in block or system types diagrams;
- Process creation symbols in process or procedure diagrams;
- Procedure declarations and procedure call symbols;
- Composite state declarations or usage symbols;
- Service symbols in composite state diagrams;
- Declaration text boxes containing an "INHERITS" line in process types;
- MSC references in MSC and HMSC diagrams.

Automatic creation of definition diagrams is also supported for all these symbols, except the process creation symbol, where the created process may be anywhere in the system.

Please note:

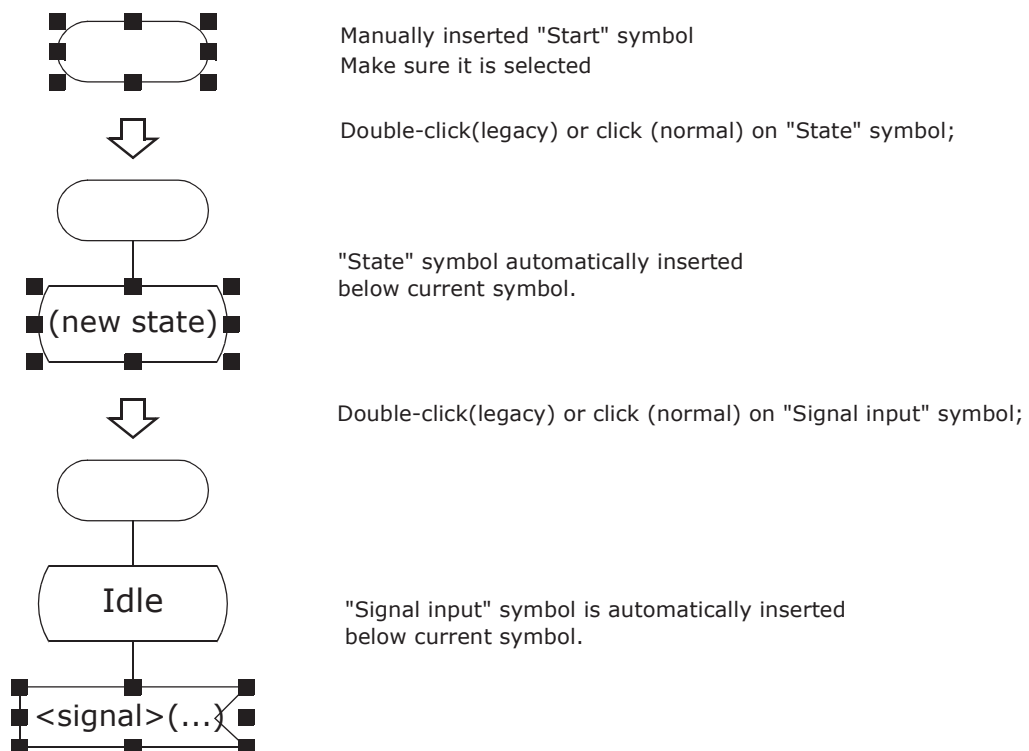
- Renaming or deleting a symbol having a definition diagram will rename or delete the node in the project tree. The deletion will display a confirmation dialog, asking whether the diagram file should be deleted.
- The reverse is not true: deleting or renaming a node in the project manager will not update the corresponding diagram. For example, if the name for a process diagram node is changed in the project manager, trying to open this process

from its parent diagram will fail and display the dialog asking if the process diagram must be created.

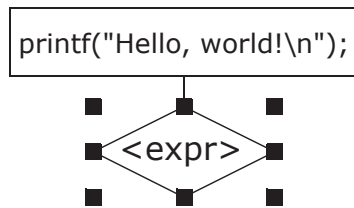
3.2.2 Automatic insertion

This feature allows to automatically create a symbol just below the current symbol with a link between the two. It is available in process, process type, procedure, macro and HMSC diagrams. It allows to easily create a vertical flow of symbols without having to create all symbols and links manually.

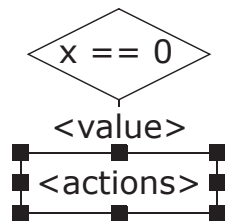
The automatic insertion is done by double-clicking in legacy mode or by simple click in normal mode on the symbol tool for the symbol you want to insert:



The automatic insertion will also choose automatically the "best" link type for linking the two symbols:

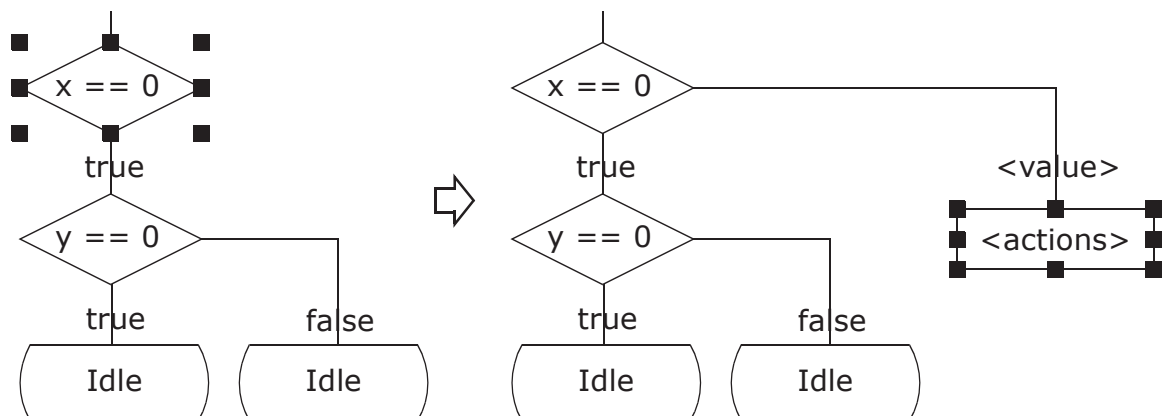
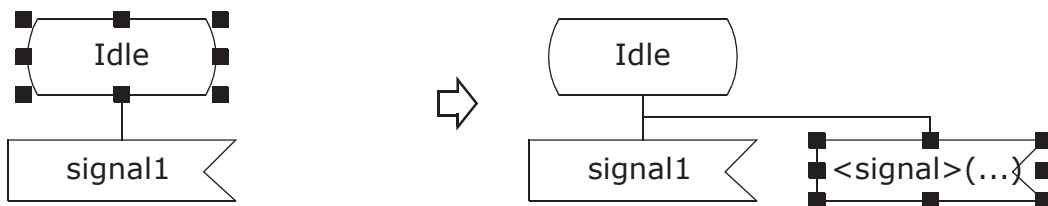


Automatic insertion of a condition after a task block

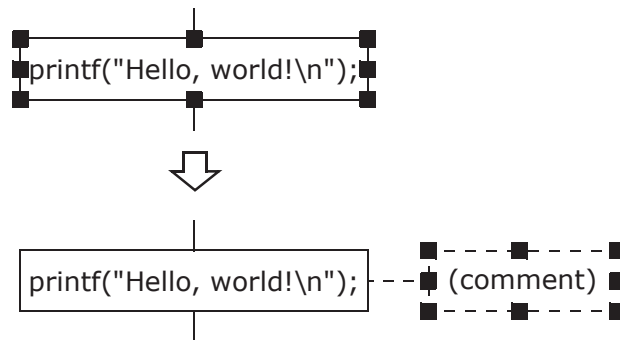


Automatic insertion of a task block after a condition

The position of the auto-inserted symbol is also computed to avoid symbol overlap:



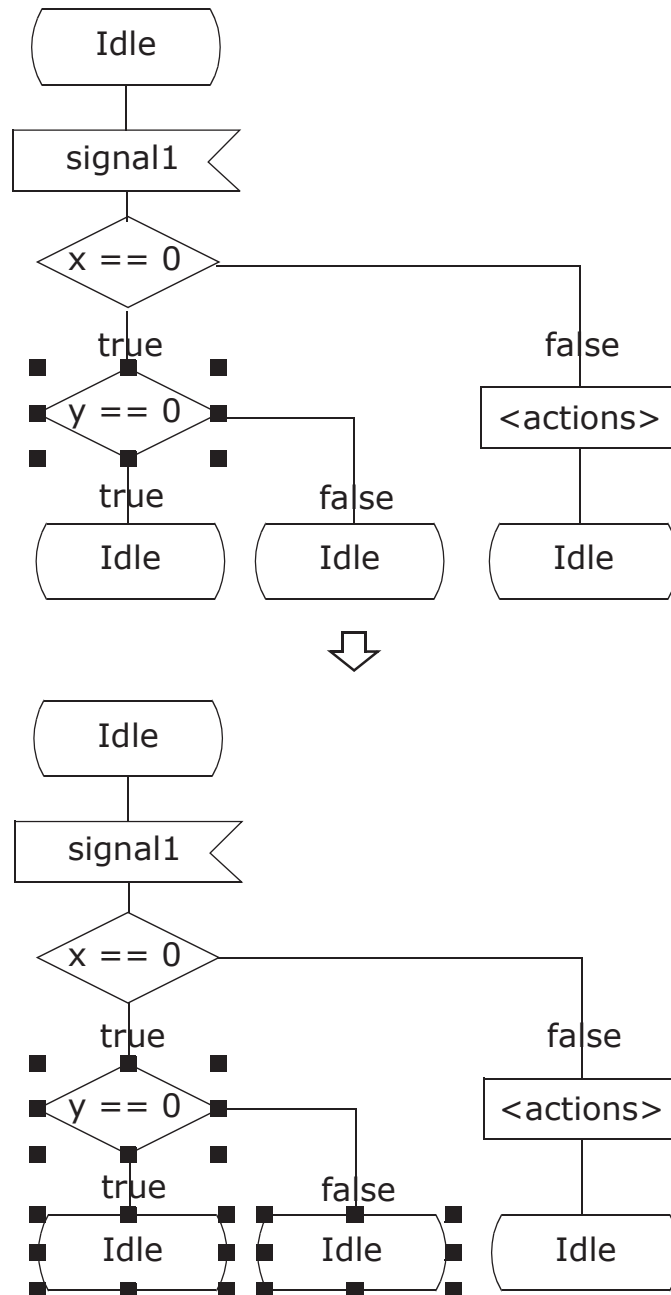
In legacy mode, auto-insertion is also provided for comment and text extension symbols . These symbols will be inserted at the right of the selected one:



3.2.3 Automatic transition selection in legacy mode

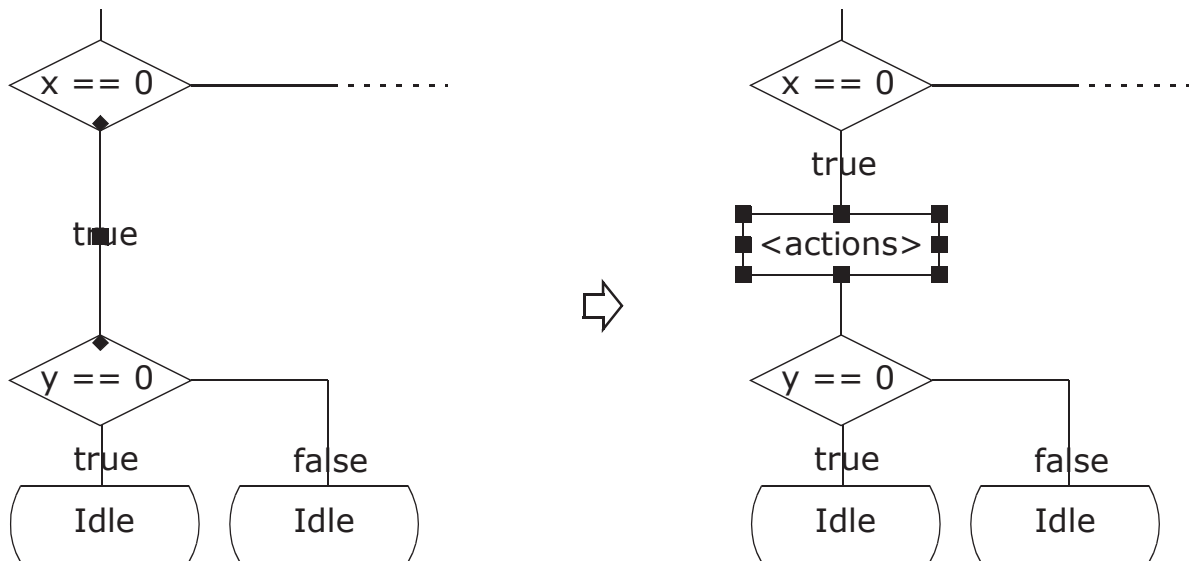
Sometimes, a symbol must be inserted in the middle of an existing transition. This can be easily done in normal mode by selecting upper symbol et clicking in the wanted symbol in the left toolbar. But in legacy mode, this case is not handled by the automatic insertion described above, as this feature won't move existing symbols. To ease such an insertion,

PragmaDev Studio allows to automatically select all symbols following a given one in a transition. This is done via the "Edit" or contextual menus, item "Select to end":



This allows to move all symbols following a given one to make space for a new one inserted before it.

Once this is done, the new symbol can be inserted in the middle of an existing link by selecting the link and double-clicking on the symbol tool, as for auto-insertion below symbols:



3.2.4 "View" / "Go to" menu and state / message browser

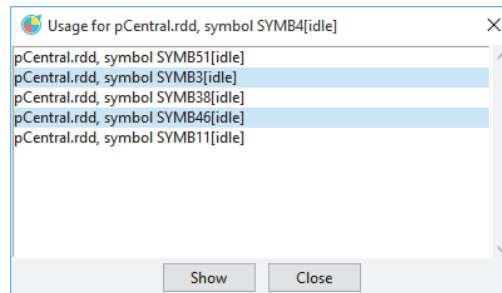
The process and process type editors include a feature allowing you to go directly to a given signal input in a given state. This feature is available:

- Via the "Go to" sub-menu in the "View" menu. This sub-menu includes one sub-menu per state defined in the process or process type. The sub-menu for a state includes an item for each signal input defined for this state. Selecting one of these items displays the corresponding symbol.
- Via the state / message browser that may appear in the right part of the editor window. This browser shows each state in the process, followed by each input message accepted in this state. Clicking on the message name displays the corresponding symbol in the diagram.

3.2.5 State and connector usage

The "View" / "Go to" menu and the state / message browser show the entry states and messages in transitions. PragmaDev Studio also allows to find where a given state is used as a next state. This is also available for connectors: for a given "in" connector, it is possible to show where it is used as an "out" connector. This is done by selecting the state or

"in" connector symbol and selecting "Show usage..." in the "Edit" or contextual menu. The following window appears, showing all symbols using the given state or connector:



Notes:

- This feature is not available with normal mode diagram.

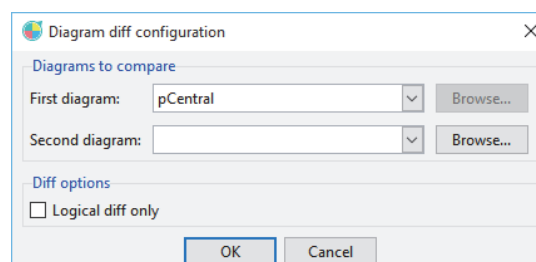
Each line contains a diagram file name, a symbol identifier and the text for this symbol. Double-clicking on a line (or selecting it and pressing the "Show" button) will open the given diagram and display the symbol.

Note that for process classes, the symbols using the state or connector in the super-class or all sub-classes are also displayed.

3.2.6 Diagram diff

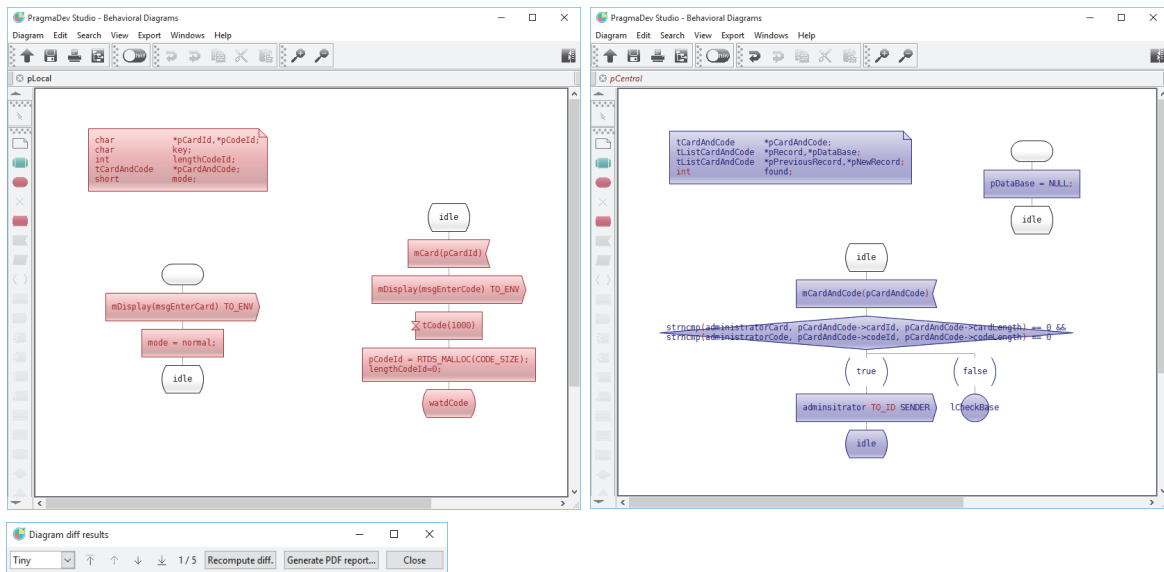
PragmaDev Studio allows to make a graphical diff of two diagrams within a project, or of a diagram within the project with an external diagram. This feature is available in the diagram editor via the "Compute differences with diagram..." item in the "Diagram" menu, or in the project manager in the "Make diff on diagram..." item in the "Element" menu.

The diff is configured via the following dialog:



The first diagram is always taken in the current project. The second diagram can either be chosen among the current project compatible diagrams in the drop-down menu, or read from an external file by using the "Browse..." button. If the "Logical diff only" option is checked, only logical differences between the two diagrams will be reported. If this option is not checked, differences may be reported for elements that appear in both diagrams if the symbol defining them have moved or has been resized.

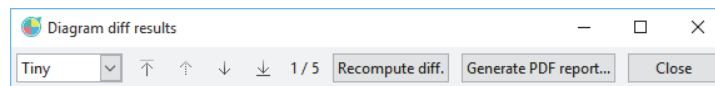
The results for the diff is displayed as follows:



Both compared diagrams are opened and automatically placed side by side on the screen. The different parts are identified by colors in the editors (NB: the colors set for symbols are ignored when in diff mode).

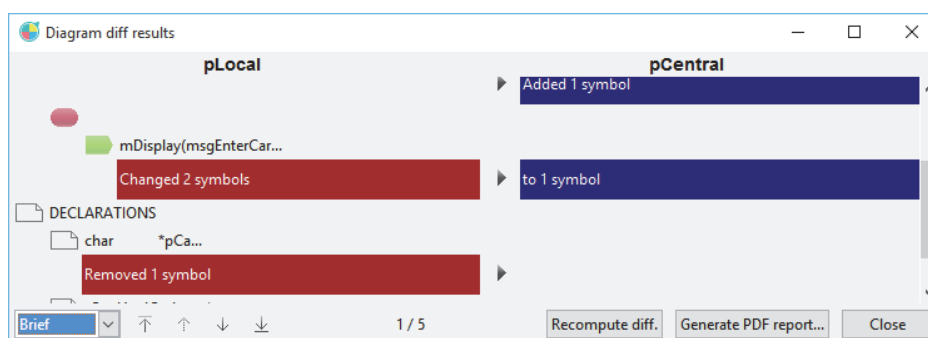
The window below the editors allow to navigate through the differences. This window has 3 possibles layouts:

- The "Tiny" layout is the one shown above:



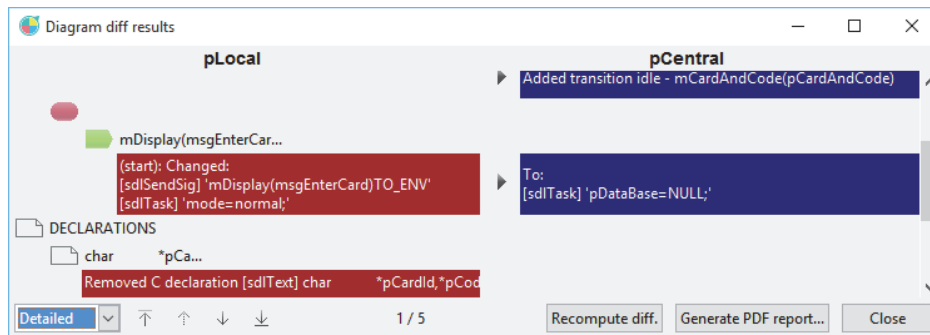
The only thing it allows is to navigate through the differences, to recompute the diff, and to generate a PDF report for the diff (see below).

- The "Brief" layout is as follows:



The differences are grouped by transition. For each difference, only the number of impacted elements is displayed. When navigating through differences, the currently displayed one is identified by the green borders on each side. Clicking on a difference will make it current and display the corresponding elements in the diagrams.

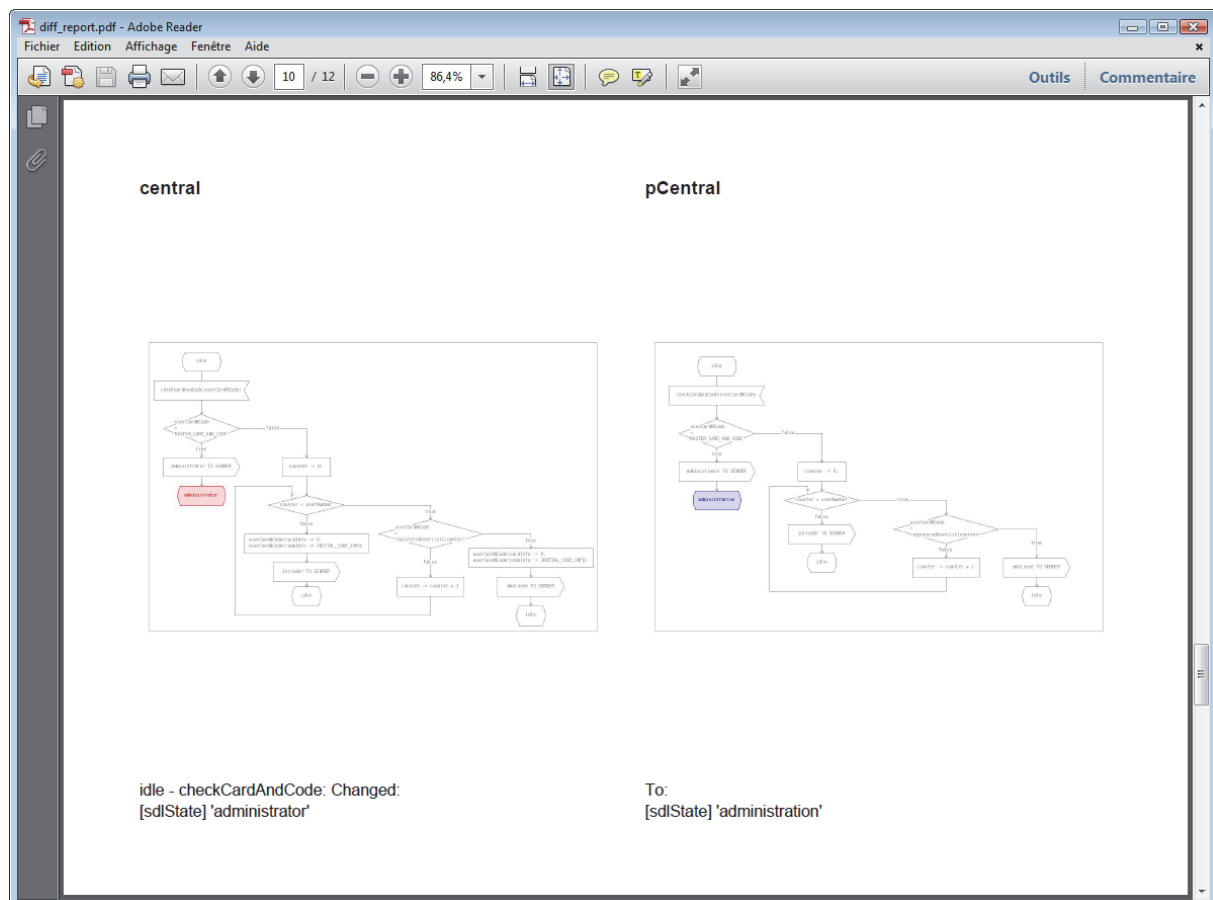
- The "Detailed" layout is as follows:



The differences are grouped and shown as in the "Brief" layout, but a full description is given for each difference.

If one of the diagrams is modified while showing differences, the differences can be recomputed using the "Recompute diff." button.

In all layouts, the “*Generate PDF report...*” button allows to generate a PDF document displaying all differences between the two compared diagrams. Each page in the document shows one difference, with the partition in the first diagram for the impacted elements on the page left side, and the partition in the second diagram on the page right side. The impacted elements are highlighted using the same colors as in the graphical diff. The description for the difference as displayed in the dialog is printed under the partitions. Here is an example of a page in a diff PDF report:



Note that it is possible that one of the sides does not appear in the report, in the case of an addition or a removal of elements between the two diagrams.

3.3 - MSC editor



The MSC editor is mainly the same as the editor for other SDL diagrams. However, there are a few extra features that maybe available depending on the type of MSC diagrams:

- *Normal* diagrams
These are the default diagrams created with PragmaDev Studio (v5.0 and above).
- *Legacy* diagrams
PragmaDev Studio allows editing of MSC diagrams created with previous versions of the tool (up to *Real Time Developer Studio* v4.6.1).

The features of the MSC editor and their availability are described in the following paragraphs.

3.3.1 Specific tools

The top tool bar for *legacy* MSC diagrams has two tools:

-  allows to select a time range within the diagram: all events occurring to all lifelines between a given start time and a given end time may be selected, copied, cut and/or pasted somewhere else in the diagram. While this tool is selected, horizontal guidelines follow the mouse cursor to indicate precisely what will be selected.
To select a time range, press the mouse button at the desired start time and drag to the desired end time.
Once a time range has been copied or cut, pasting is done by clicking at the desired insertion time. The paste operation also displays a horizontal guideline, and may be cancelled by hitting the *Esc* key or by selecting the regular selection tool.
Please note that it's impossible to select a time range in a diagram and to paste it in another.
-  allows to insert empty space in the diagram: press the mouse button at the desired insertion position and drag. When releasing the button, and if it's possible, a space having the length between the start and end position will be inserted in the diagram.

Note: These tools are not available in *normal* MSC diagrams. However,

- selecting a range in these diagrams is straightforward, i.e., simply click-and-drag the mouse to select the desired area;
- inserting an empty space is not possible.

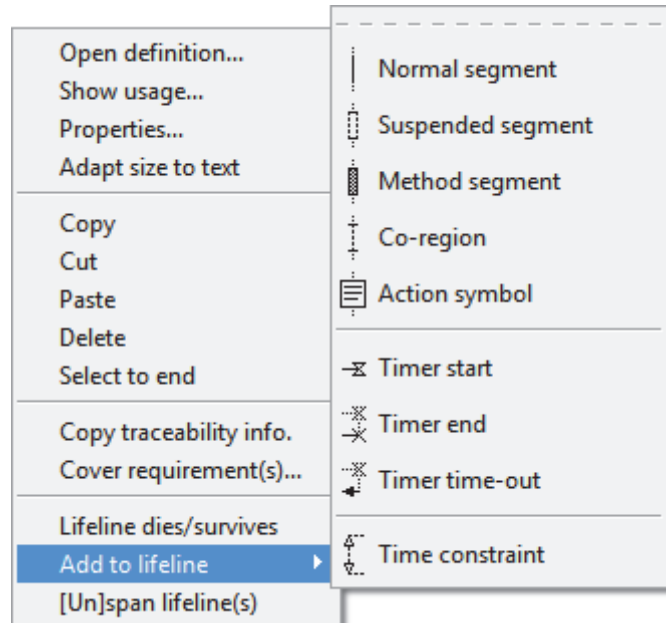
3.3.2 Manipulating components in lifelines

To the difference of all other symbols, lifeline are composite symbols: they may include several components like segments, timers or time constraints. They may also die before the end of the diagram or survive it.

In *normal* diagrams these features are managed via the toolbar; in *legacy* diagrams they are managed via the context menu (right-click on the lifeline):



Toolbar for normal diagrams

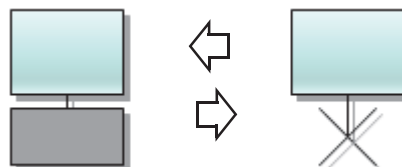



Context menu for legacy diagrams



The items (*button* for normal and menu item for *legacy*) carry out the following actions:

-  or “Lifeline dies/survives”

Toggles the instance tail / instance stop ending for the lifeline:



- Remaining buttons or “Add to lifeline”
Adds to the lifeline a segment, an action symbol, a timer or a time constraint. After selecting an item in the toolbar or sub-menu, press the mouse button at the desired position in the lifeline, and drag to its end position (if applicable). To cancel the insertion, hit the *Esc* key or select the selection tool .

Note: A *Message save* can be inserted via the  button in *normal* diagrams or  in *legacy* diagrams.

- “[Un]span lifeline(s)”
It attaches or detaches a spanning symbol from a set of lifelines. Spanning symbols are conditions or MSC references, which are attached to one or several lifelines: if the lifelines are moved, the size of the symbol is updated so that the lifelines are always spanned by the symbol. This menu item only works when a spanning symbol and one or several lifelines are selected:

- If the selected lifelines are not spanned by the symbol, they're added to the set of spanned lifelines;
- If the selected lifelines are already spanned by the symbol, they're removed from the set of spanned lifelines.

A dialog will inform you about the performed operation.

Note 1: there is no visual information on *legacy* diagrams about lifelines spanned by symbols. The lifelines are always behind spanning symbols, even if they're not spanned.

Note 2: In *normal* diagrams spanning is achieved via a simple resize of the symbol: it will automatically span if the new size will cross another lifeline.

The “Delete” menu item in *legacy* diagrams has a special meaning if a lifeline item such as a component, timer or time constraint is selected. In this case, only the item is deleted, not the entire lifeline. A confirmation dialog will always specify what is deleted. It is impossible to copy or cut a single item. To remind you of this, when a lifeline item is selected, the whole lifeline will still appear as selected, but with white handles instead of black ones.

Note: In *normal* diagrams these actions are distinct, i.e., “Delete symbol” and “Delete lifeline” depending on where the contextual menu was triggered.

3.3.3 Message parameters display

A specific sub-menu in the “View” menu controls the message parameter visibility:

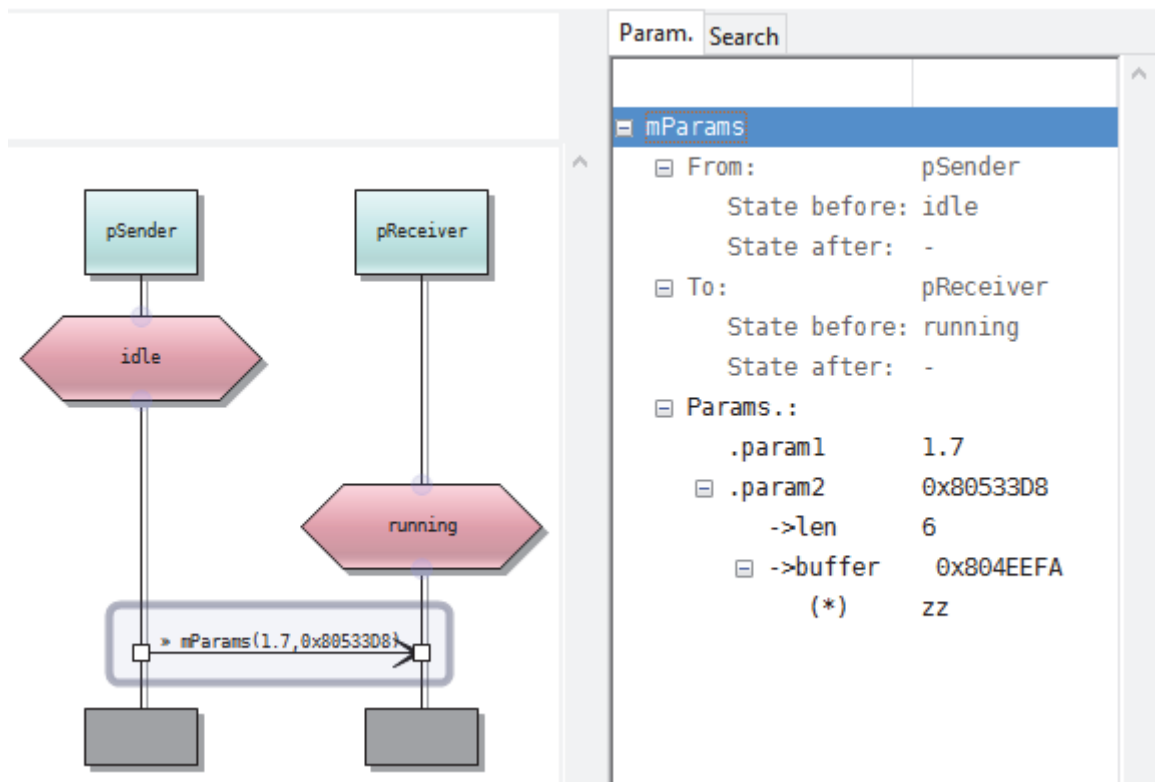
- A visibility set to “Full” displays the full text for the message parameters as it is recorded in the diagram file. The parameters for structured messages are then displayed in the format described in paragraph “Sending SDL messages to the running system” on page 252. This can make the diagram quite difficult to read, as this format is quite complex;
- A visibility set to “Abbreviated” still displays completely parameters for non-structured messages, but only displays the first level of parameter values in structured parameters. An example of this visibility can be seen below;
- A visibility set to “None” hides all message parameters.

This visibility setting is stored with the diagram. Please note it is only possible to modify the text for the message parameters if the visibility is set to “Full”.

When the visibility is set to “None” or “Abbreviated”, structured messages are indicated by a “»” before their name. Their parameters may be displayed by clicking on the message link: a panel then appears in the right part of the editor window displaying the parameters as a tree. For example, for a message with the full text:

```
mParams ( | {param1|=1.7| , param2|=0x80533D8| : | {len|=6| , buffer|=
0x804EEFA| : zz| } | } )
```

the display with parameter visibility set to “Abbreviated” and the link selected is:



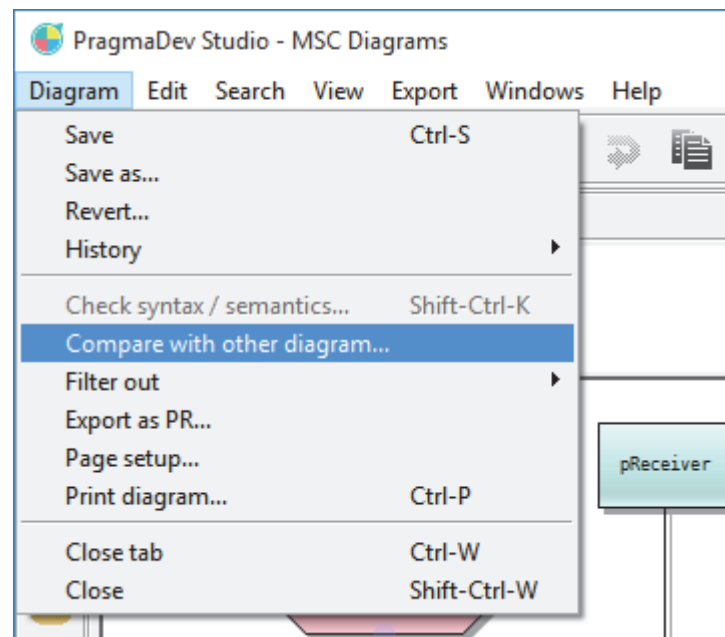
Other information is also displayed in the panel, such as the sender and receiver process and their states before and after they sent / received the message.

3.3.4 Compare diagrams

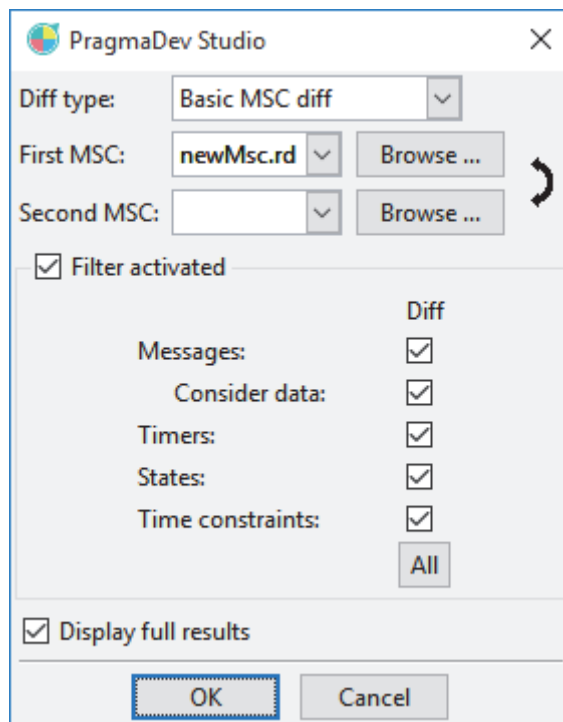
This allows to show the differences between two MSC diagrams.

3.3.4.1 Comparing

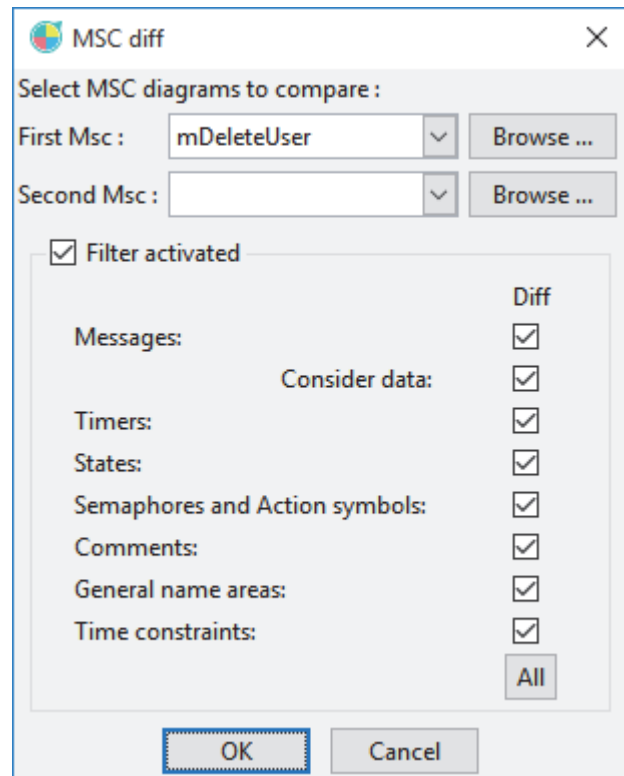
The “Compare with other diagram...” can be accessed from the MSC editor’s “Diagram” menu, or from the project manager’s contextual menu or “Element” menu when an MSC diagram is selected.



For *normal* diagrams the following window pops up:



Normal diagrams



Legacy diagrams

Here the two MSC diagrams can be selected from the current project or from the file system. Filters can also be applied on the result of the MSC comparison.

The *Diff type* option in *normal* diagrams can be one of:

- *Basic MSC diff*
This is a basic comparison and the same as in legacy diagrams. In this case nothing is interpreted: e.g., if you there is an “alt” inline expression in one of the diagram which doesn't exist in the second one, it will be considered as a difference, i.e., the “alt” is not interpreted.
- *Spec. vs trace*
Here the “alt”, “loop”, and all inline expressions are interpreted. Referring to the above example, if the second MSC contains one of the alternatives in an “alt” for example, this will not be considered as a difference.
- *Property match*
In this case the first MSC has to be a PSC (Property Sequence Chart), which describes a property that must be checked in the other MSC. For more on PSCs please see the PragmaDev Tracer Manual.

3.3.4.2 Result analysis

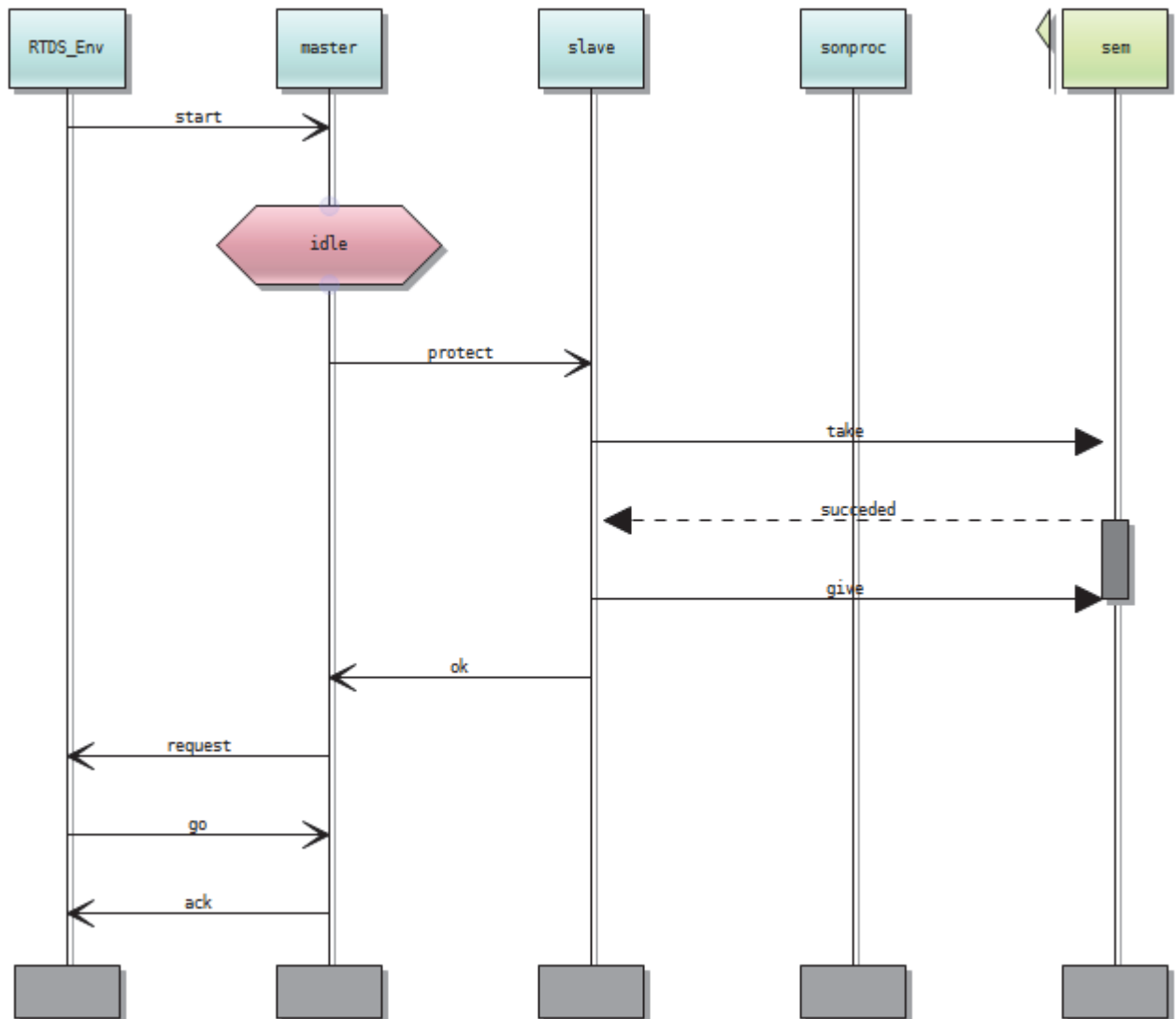
There is a substantial difference between *normal* and *legacy* in the way the results of the comparison are made available.

- For *normal* diagrams the results (i.e., the differences between the two diagrams) are listed in a new window which allows navigation through the differences, where each difference is marked in the original diagram.
- For *legacy* diagrams the comparison results in a new diagram, where the differences are marked with different colors. The resulting MSC identifies the lifelines (processes) between the two diagrams by their name and if necessary (if there are several processes with the same name), by the sequence of their events. If two lifelines in the two diagrams represent the same process but do not have the same name, they will not match.

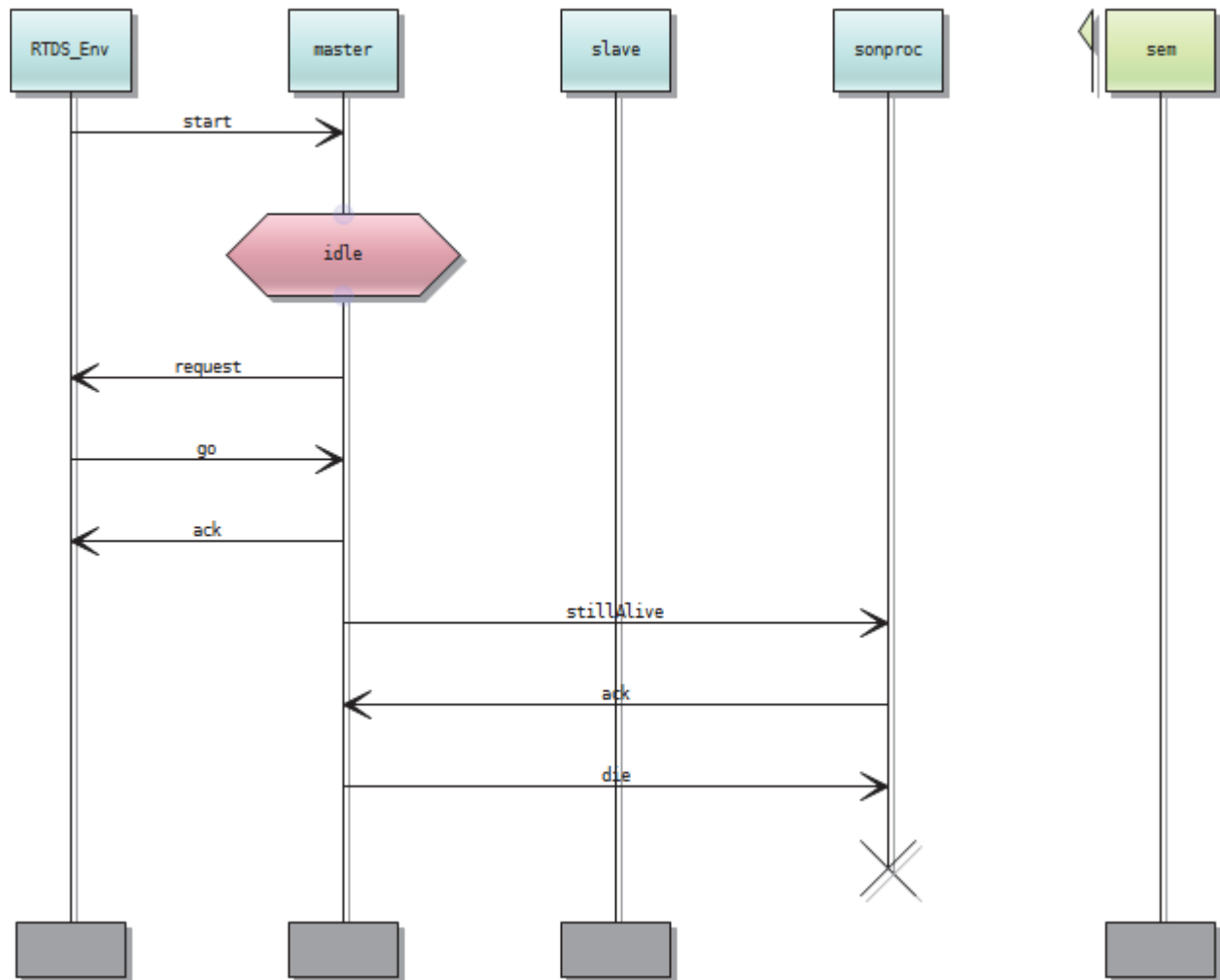
Note: The comparison searches for the difference in the sequence of events on the lifelines. The elapsed time between two events on the resulting diagram is not significant.

3.3.4.3 Example

The comparison is made on the following two MSC diagrams:



First MSC diagram in comparison



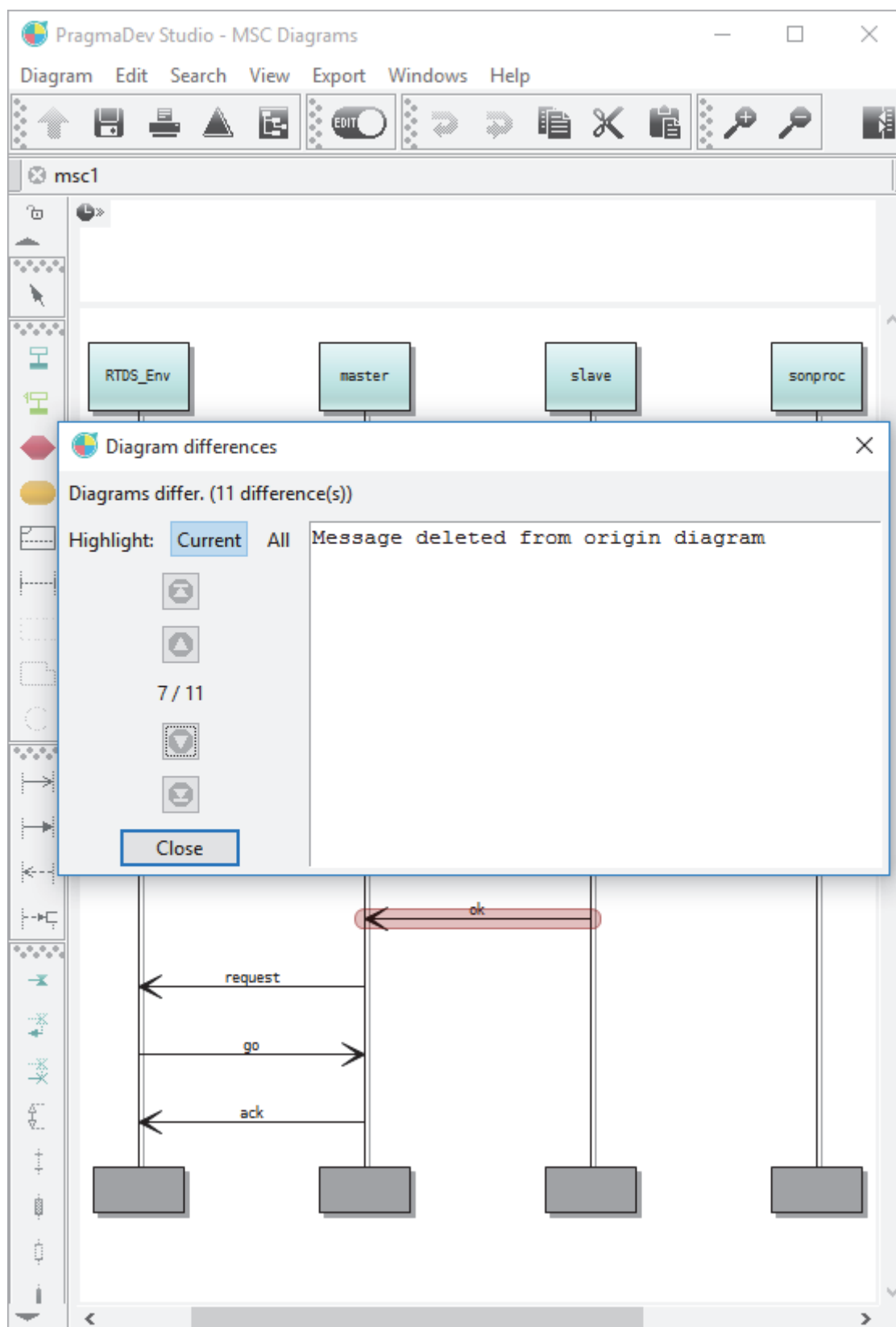
Second MSC diagram in comparison

The differences are quite clear:

- The `slave` process takes the semaphore `sem` in the first diagram, but not in the second.
- The processes `master` and `sonproc` exchange messages `stillAlive`, `ack` and `die` in the second diagram, but not in the first.
- Process `sonproc` dies in the second diagram, but not in the first.

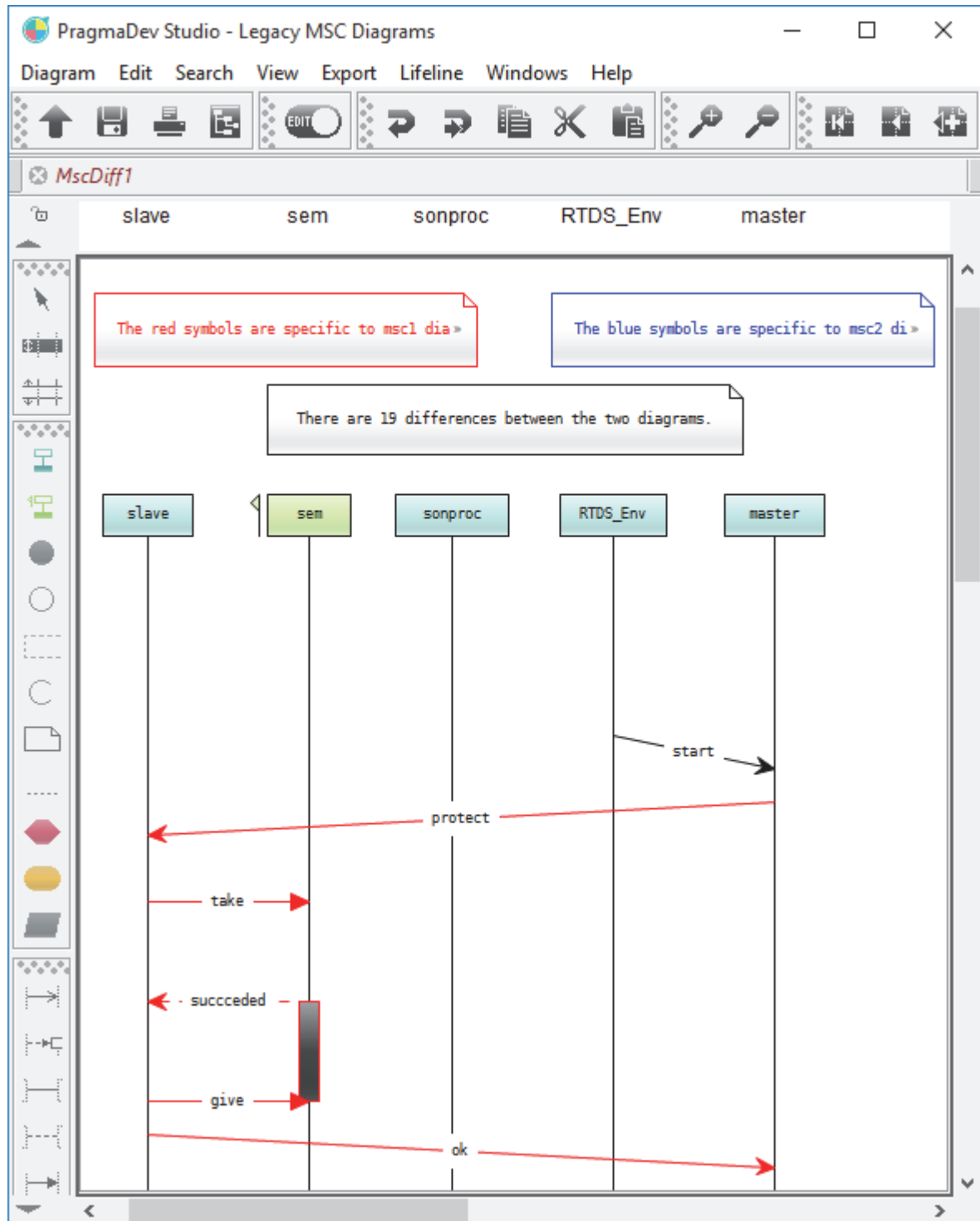
Here are the results of the comparison:

- *Normal diagrams:*



As you navigate through the differences (buttons on the left in the *Diagram differences* window), these will be marked in *red* in the first diagram (the one the comparison was launched) and *blue* in the second.

- *Legacy* diagrams:



The differences are identified by colors: in the diff diagram, the events specific to

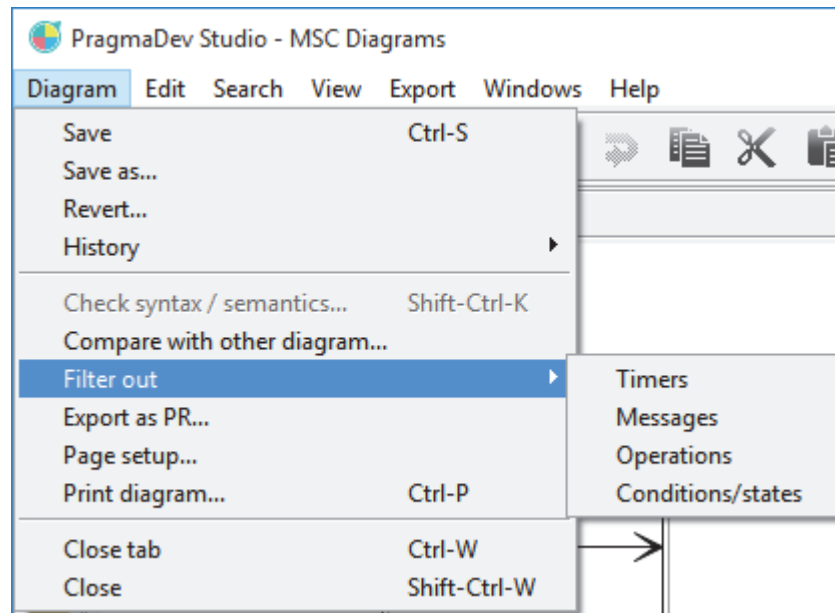
the first diagram are represented in red, and the events specific to the second diagram are represented in blue.

3.3.5 Filtering

Large MSCs might get very difficult to read; In that case the MSC filtering feature can be used to remove useless information from the diagram.

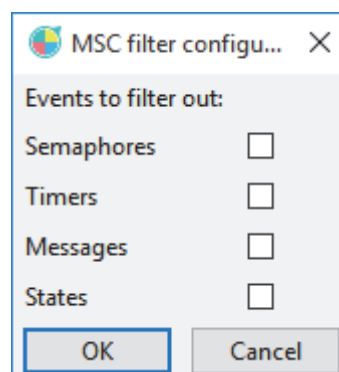
This functionality is accessed via the “Diagram” menu:

- For *normal* diagrams use the sub-menu “Filter out”:



Any selected filter will be applied directly to the current diagram

- For *legacy* diagrams use the menu item “Filter diagram...”; the following window will pop-up:

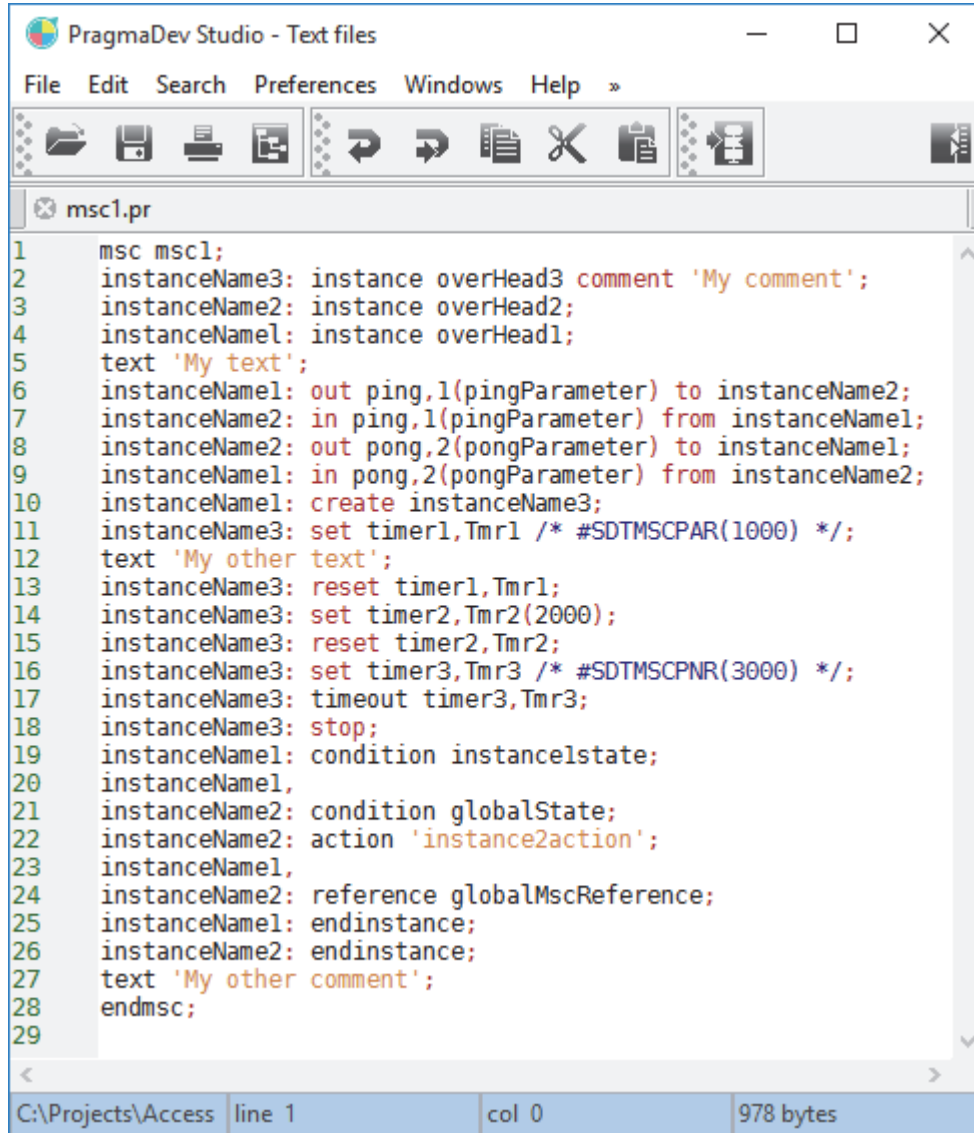


A new MSC diagram will be created (and added to the project) based on the original, but with the selected events filtered out.

Note: The resulting MSC might not have the same layout as the original MSC. This is because the original MSC is translated to a list of a MSC events that is filtered. The resulting MSC is built out of the sequence of left events. So for example the vertical space between events is always the same.

3.3.6 MSC PR import

It is possible to import MSCs stored in PR (Phrasal Representation) textual format as defined in ITU-T Z.120 recommendation. The MSC must be event oriented (as opposed to instance oriented) and the CIF information if any will be ignored.

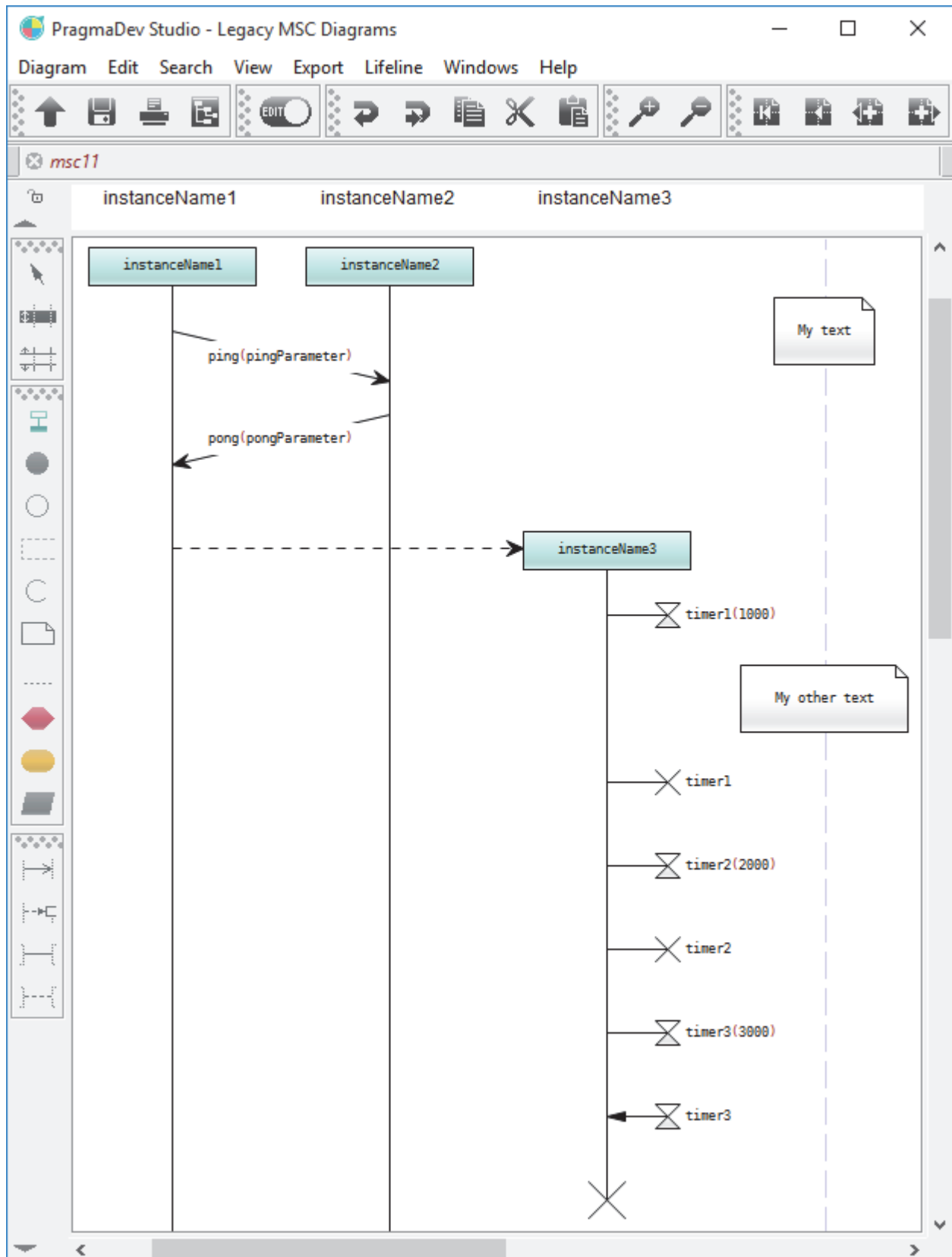


```
1 msc msc1;
2 instanceName3: instance overHead3 comment 'My comment';
3 instanceName2: instance overHead2;
4 instanceName1: instance overHead1;
5 text 'My text';
6 instanceName1: out ping,1(pingParameter) to instanceName2;
7 instanceName2: in ping,1(pingParameter) from instanceName1;
8 instanceName2: out pong,2(pongParameter) to instanceName1;
9 instanceName1: in pong,2(pongParameter) from instanceName2;
10 instanceName1: create instanceName3;
11 instanceName3: set timer1,Tmr1 /* #SDTMSCPAR(1000) */;
12 text 'My other text';
13 instanceName3: reset timer1,Tmr1;
14 instanceName3: set timer2,Tmr2(2000);
15 instanceName3: reset timer2,Tmr2;
16 instanceName3: set timer3,Tmr3 /* #SDTMSCPNR(3000) */;
17 instanceName3: timeout timer3,Tmr3;
18 instanceName3: stop;
19 instanceName1: condition instance1state;
20 instanceName1,
21 instanceName2: condition globalState;
22 instanceName2: action 'instance2action';
23 instanceName1,
24 instanceName2: reference globalMscReference;
25 instanceName1: endinstance;
26 instanceName2: endinstance;
27 text 'My other comment';
28 endmsc;
29
```

A Z.120 MSC PR example

In the project manager, go to the “Project / Import MSC-PR file...” menu and select the file to import.

The resulting MSC will be added to the project:

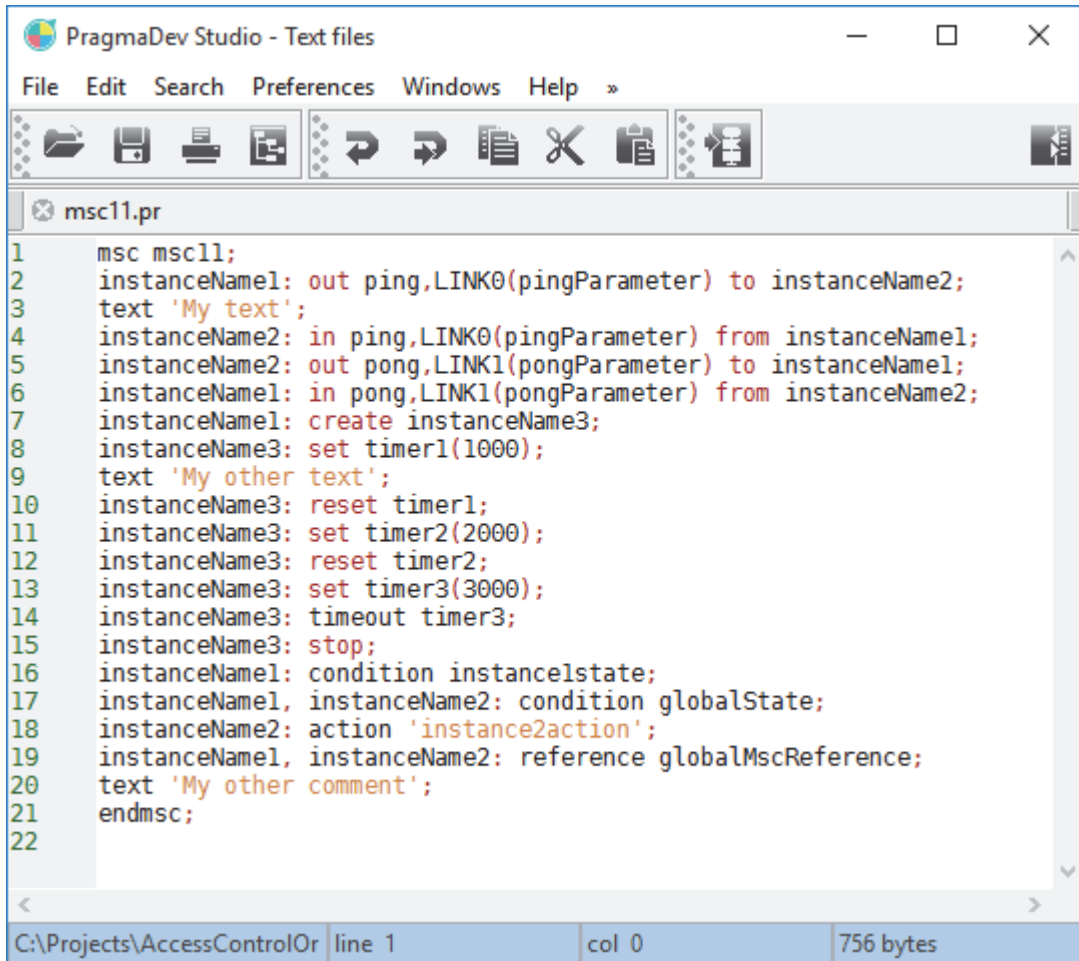


Imported MSC

3.3.7 MSC PR export

It is possible to export MSCs to an event oriented Z.120 MSC PR file. In the MSC diagram to export, go to the “Diagram / Export as PR...” menu.

The results of the export is as follows:



The screenshot shows the PragmaDev Studio interface with a text editor window titled 'msc11.pr'. The editor contains the following Z.120 MSC PR code:

```
1  msc msc11;
2  instanceName1: out ping, LINK0(pingParameter) to instanceName2;
3  text 'My text';
4  instanceName2: in ping, LINK0(pingParameter) from instanceName1;
5  instanceName2: out pong, LINK1(pongParameter) to instanceName1;
6  instanceName1: in pong, LINK1(pongParameter) from instanceName2;
7  instanceName1: create instanceName3;
8  instanceName3: set timer1(1000);
9  text 'My other text';
10 instanceName3: reset timer1;
11 instanceName3: set timer2(2000);
12 instanceName3: reset timer2;
13 instanceName3: set timer3(3000);
14 instanceName3: timeout timer3;
15 instanceName3: stop;
16 instanceName1: condition instance1state;
17 instanceName1, instanceName2: condition globalState;
18 instanceName2: action 'instance2action';
19 instanceName1, instanceName2: reference globalMscReference;
20 text 'My other comment';
21 endmsc;
22
```

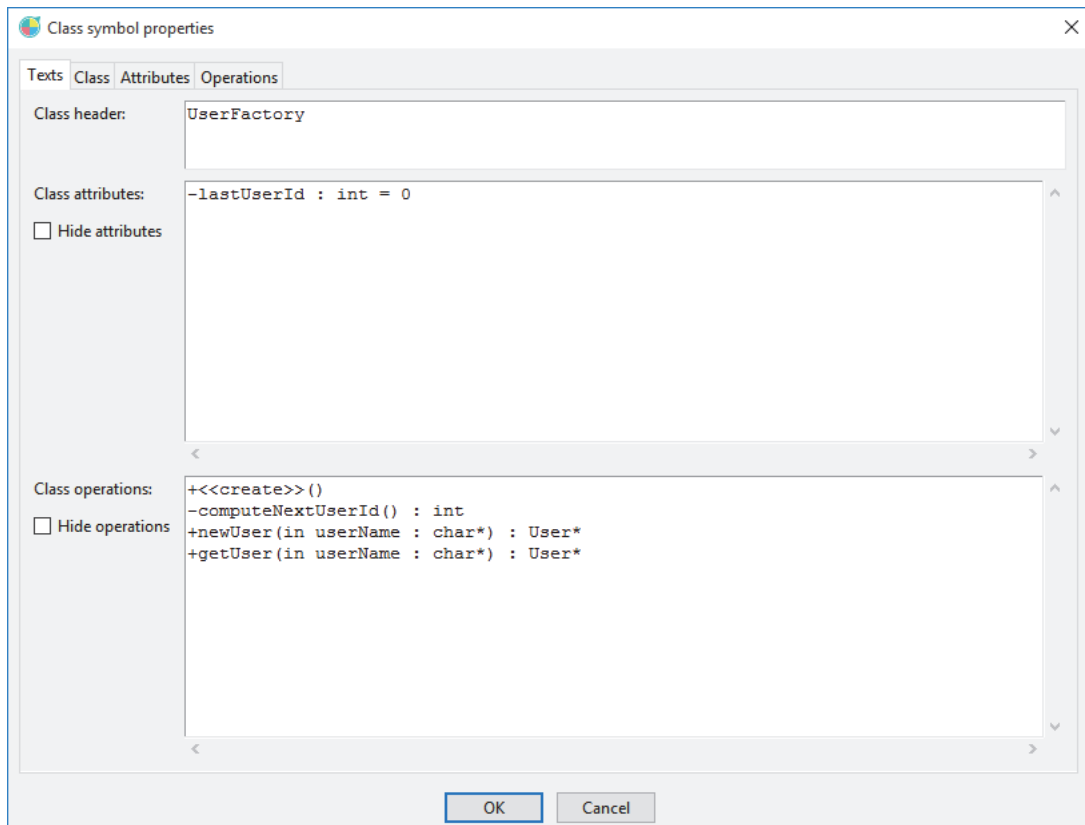
The status bar at the bottom indicates the file path 'C:\Projects\AccessControlOr', line 1, column 0, and a size of 756 bytes.

Export MSC PR example

3.4 - UML diagrams

3.4.1 Symbol properties

For symbols that may show attributes and/or operations (class symbols in class diagrams, nodes and components in deployment diagram), a specific property sheet is available:



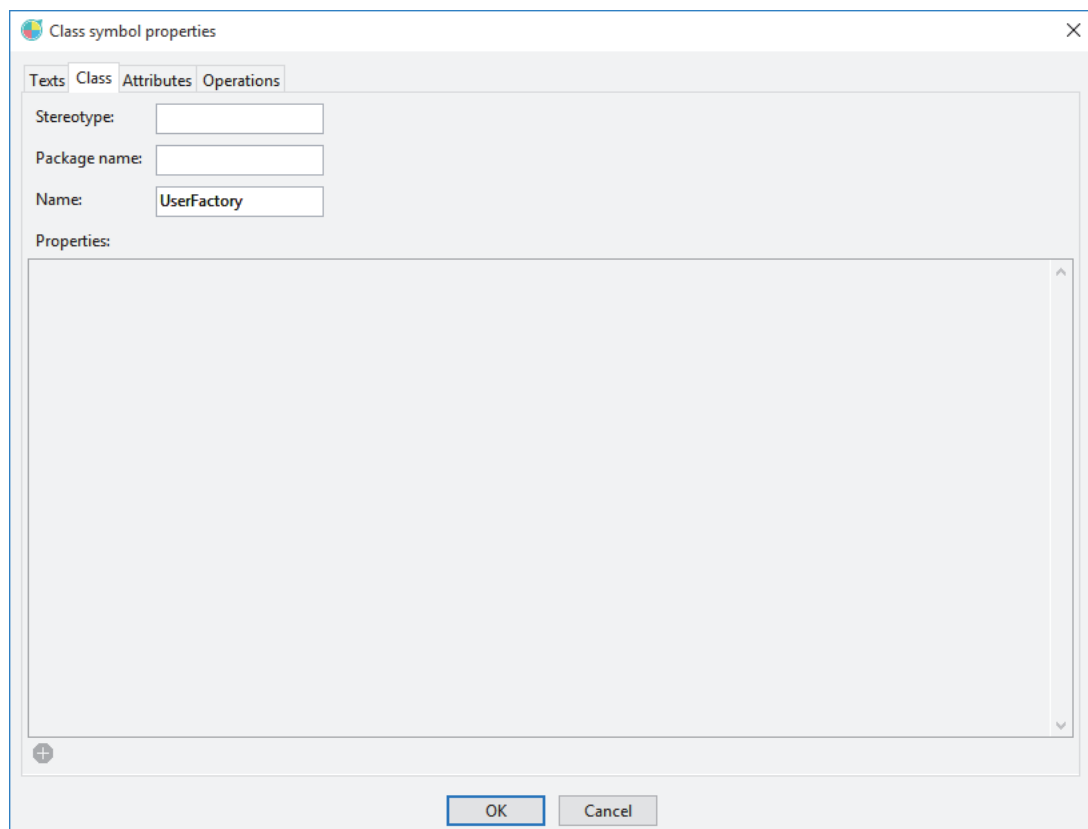
This property sheet is divided into 3 or 4 tabs depending on the selected symbol. These tabs are described in the following paragraphs.

3.4.1.1 "Texts" tab

This tab is shown above. It contains the standard symbol colors and the textual representations for all texts associated to the selected symbol: class header, attributes and operations. The operations do not appear for nodes or components since they are meaningless for these symbols. Two check-boxes are also available, allowing to hide the attribute and operation parts in the displayed symbol respectively.


3.4.1.2 “Class” tab

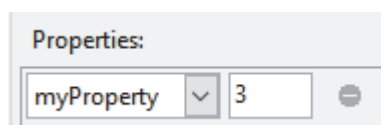
This tab is the second one in the dialog:



It contains the class's stereotype, package name, name, and list of properties. All these will be included in the class header as:

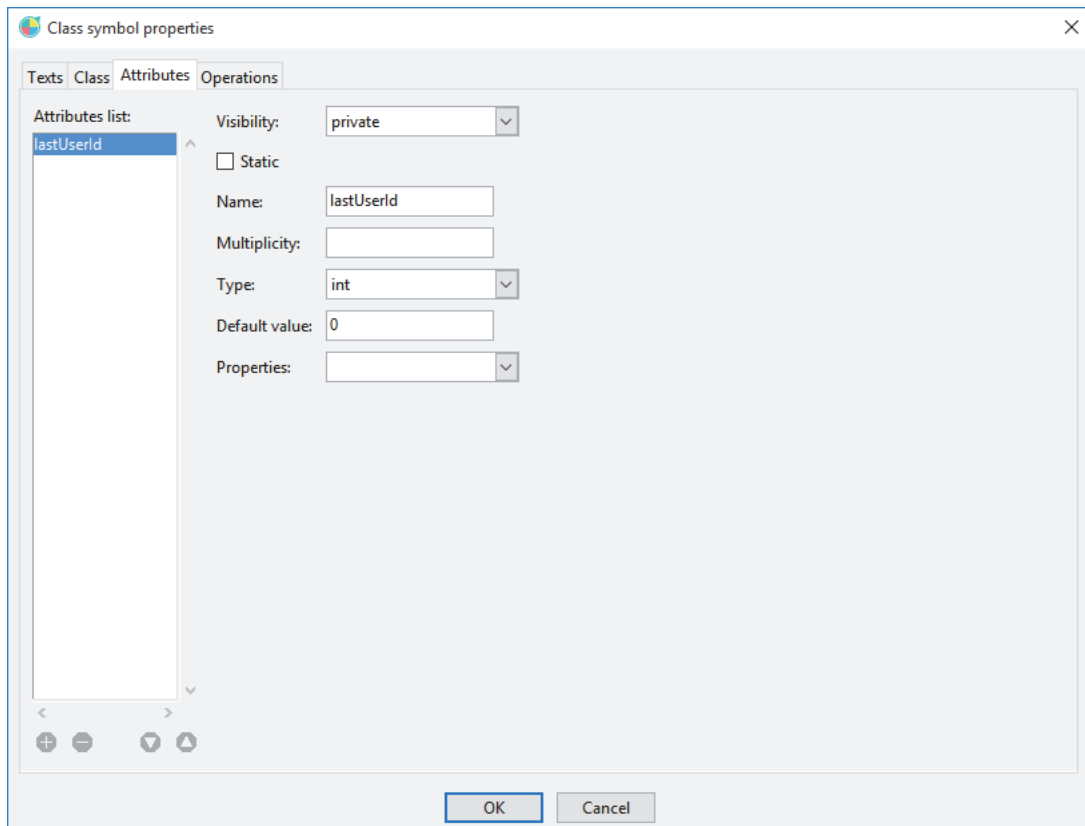
```
<<stereotype>> package-name::class-name {property-name=property-value}
```

Properties can be added with the button  at the bottom left; they can be removed with the  button appearing next to them:



3.4.1.3 “Attributes” tab

This tab is the third in the dialog:



The list in the left part allows to manage the attributes. Attributes may be created/ deleted via the buttons or re-ordered via .

The fields in the right part allows to modify the attribute selected in the list. The text for the attribute will be:

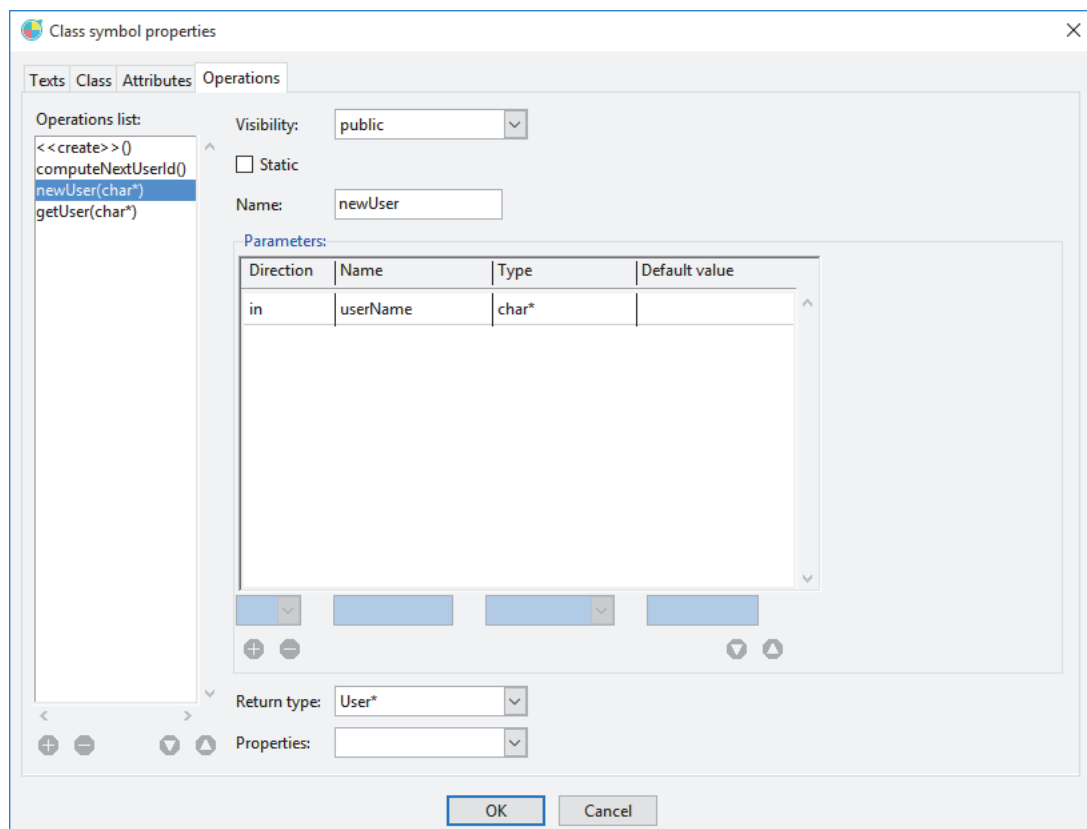
`visibility name[multiplicity] : type = default-value {property-string}`

The visibility is rendered as the standard UML character ('+' for public, '#' for protected, and '-' for private). The multiplicity, type, default value and property string are included only if they are not empty.

Any modified attribute will be automatically updated in the attributes text in the “Text” tab.

3.4.1.4 “Operations” tab

This tab is the fourth in the dialog:



This tab is not available for nodes or components (in deployment diagram), as these do not have any operations.

The list in the left part allow to manage the operations. The buttons are the same as for the attributes (see above).

The fields in the right part allow to modify the operation selected in the list. The text for the operation will be:

```
visibility name(parameters) : return-type {property-string}
```

where parameters is a comma-separated list of parameters formatted as follows:

```
direction name : type = default-value
```

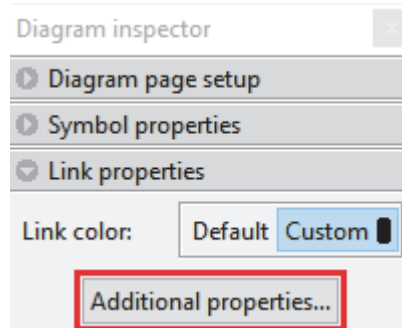
The visibility is rendered as the standard UML character ('+' for public, '#' for protected, and '-' for private). The return type and property string for the operation and the type and default value for the parameters are included only if they are not empty.

Constructors and destructor are defined with the UML stereotypes: <<create>> and <<delete>>.

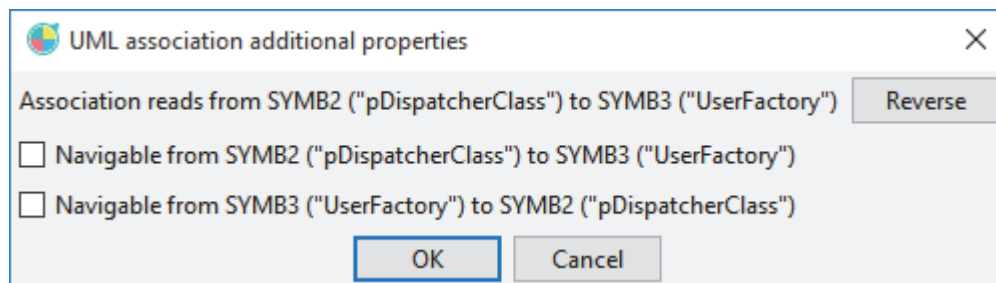
Any modified operation will be automatically updated in the operations text in the “Text” tab.

3.4.2 Link properties

Specific properties for association, aggregation and composition links may be modified via their *Additional properties...* These can be accessed (with the link selected) via “Properties...” in the contextual menu or “Edit” menu:



The *Additional properties...* window will pop-up:



The *Reverse* button will make the link read “from SYMB3 to SYMB2” (instead of “SYMB2 to SYMB3”).

For both class symbols at the ends of the link, the dialog allows to modify the navigability. The rules for the navigability are those defined by UML:

- If both navigation options are selected, each class will have an attribute representing the association.
- If a single navigation option is selected, only the class symbol “from” which the navigation is enabled will have an attribute for the association.

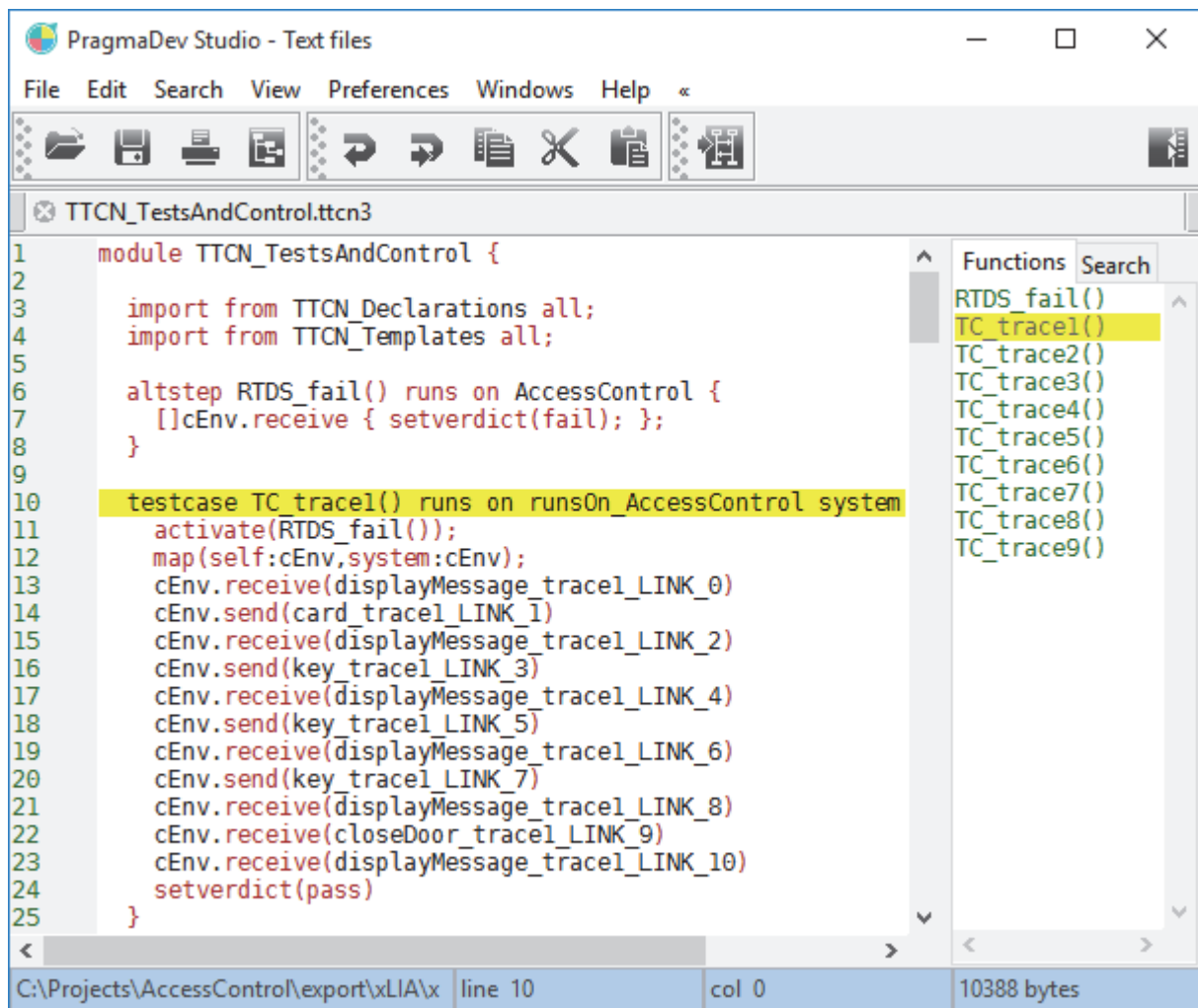
3.4.3 Access to generated C++ files

Double-clicking on a class symbol in a class diagram opens the generated C++ file for the class if there is one. Please note the file is not generated if it does not yet exist.

4 - Text Editor

The PragmaDev Studio Text Editor is the window where all types of text files may be edited. The window is the same for all type of files. It may have extra features depending on the type of the displayed file.

The Text Editor has predefined syntax highlighting for C/C++, ASN.1, TTCN-3, Python, and CORBA IDL languages. A browser at its right side also lists all the functions, classes and methods in the file, and to jump to any of these by clicking on these. This browser is available for C/C++, TTCN-3, Python, and partially for CORBA IDL:



4.1 - MSC generation from TTCN-3 source file.

PragmaDev Studio gives the possibility to generate an MSC view of a TTCN-3 module. An MSC diagram will be generated for each testcase and function of this module. To get this

MSC view, click on the *View graphical representation* button .

4.2 - SDL generation for comments in a C source file.

PragmaDev Studio also gives the possibility to generate SDL diagrams from specific comments in a C source file. To generate an SDL representation of a C file, click on the *View*

graphical representation button  when a C source file is open in the editor. The C macros are described in the PragmaDev Studio Reference Manual.

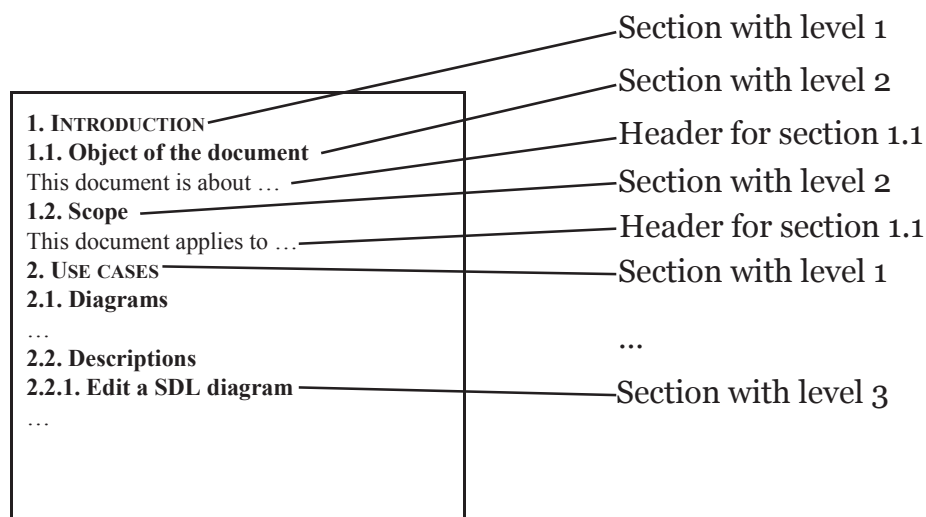
5 - Document editor

5.1 - General presentation

PragmaDev Studio's way of handling documents is a bit different from what one can find in word processors. In a word processor, a document is just a set of paragraphs, usually having a style - named or not -, and each paragraph typically contains ranges of characters, optionally styled, and/or images. There is sometimes a notion of document structure in terms of chapters, containing sub-chapters, containing sections, and so on, but this structure is deduced from the paragraphs.

On the contrary, PragmaDev Studio mainly uses the structure: a document is a set of chapters, sub-chapters, and so on, and the styles applied to the paragraphs are deduced from the structure. More precisely:

- A document is defined as a tree of sections. The top-level sections are the chapters; the ones under the chapters are the sub-chapters, and so on...
- Each section also has a section header, which is the text and images that will appear in the section *before* its sub-sections.



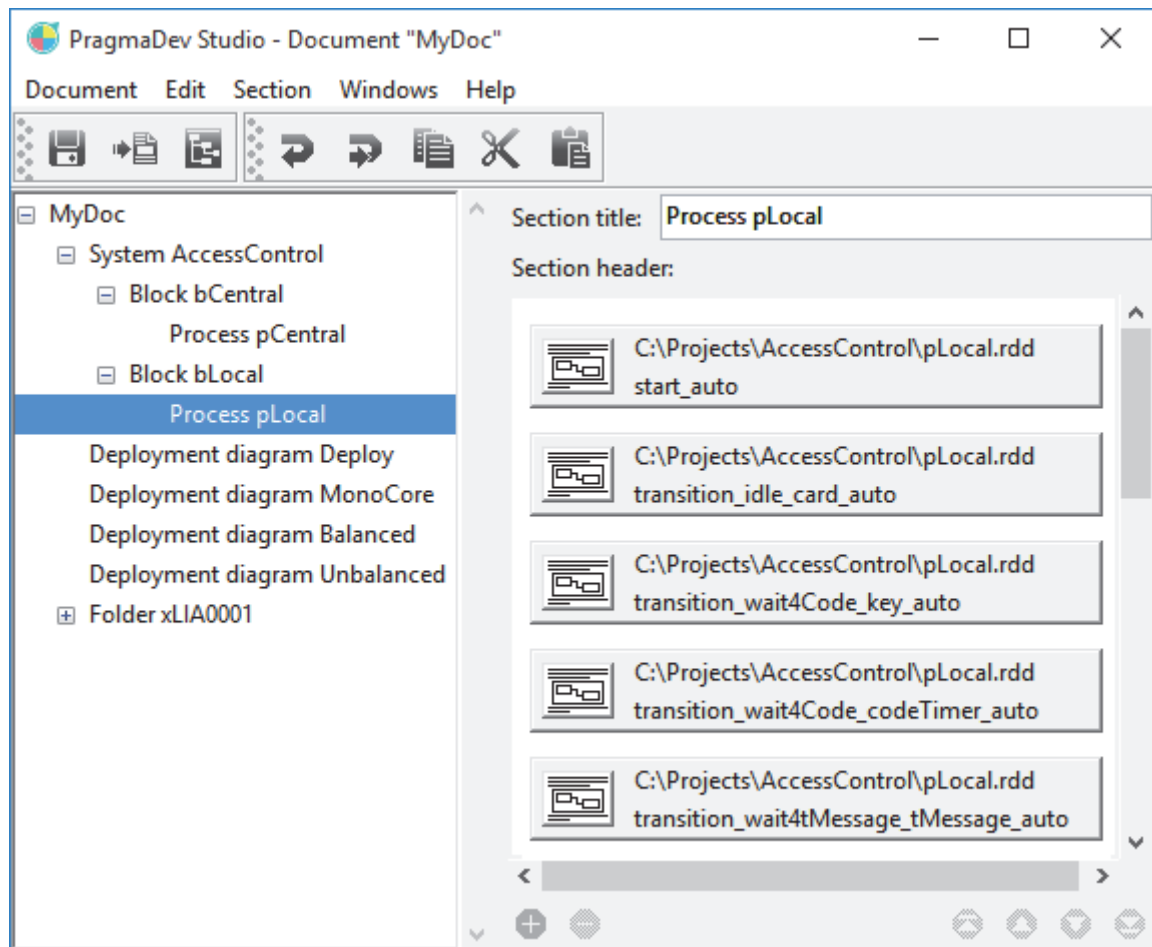
A section header is composed of header items, which can be:

- Text items: these items just contain styled text.
- Publication items: these items are references to publications in diagrams, as described in “Publications” on page 62. Such a header item will automatically include in the document:
 - The text before the publication if any;
 - The part of the diagram that is exported via the publication;
 - The text after the publication if any.
- Table items: these items contains a table, each of the table cells being a text item.
- External picture items: these items reference an image in a PNG file.
- External file items: these items just include the contents of an external file in the document, all text having a given style.

This allows to document the diagrams “on the fly” when the need arises by creating a publication and entering the texts before and after it. When the final document is written, it will just gather the parts already documented via publications, maybe with some additional explanation texts in text, external picture or table header items.



A fully documented system is available in the example files delivered with PragmaDev Studio. It can be found under PragmaDev Studio installation directory, under examples/Specifier/AccessControl.


Here is a typical view of the document editor:

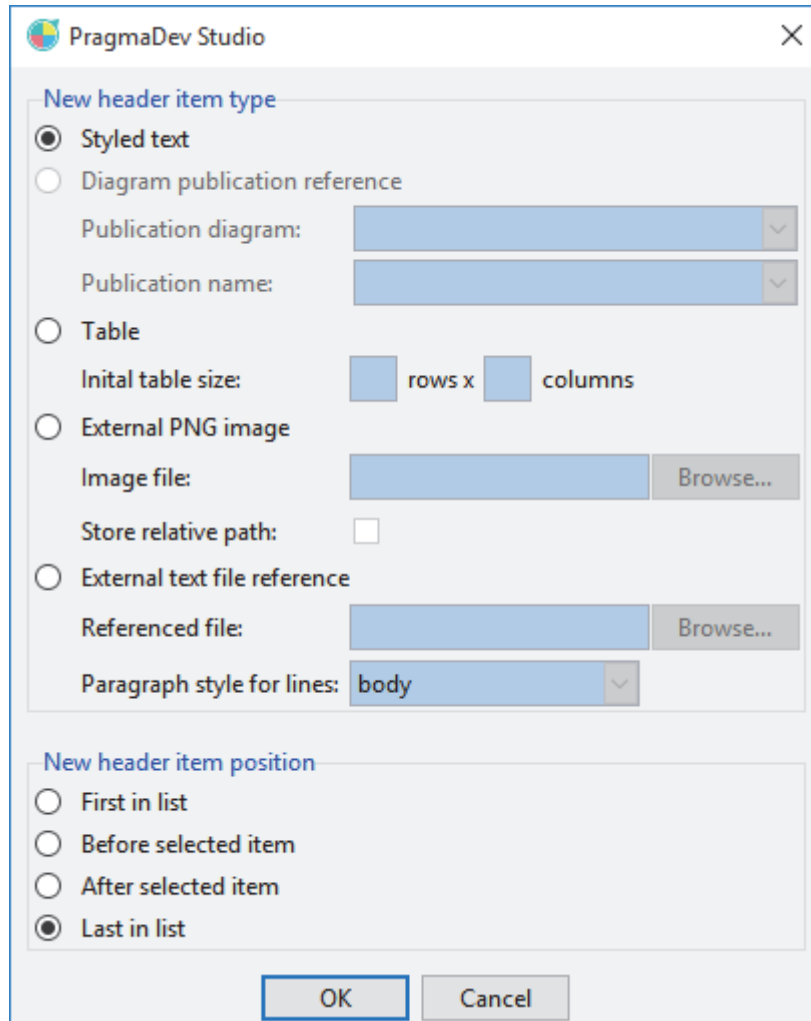


The left part of the window shows the section tree for the document, and the right part shows a text field allowing to change the selected section title and the list of its header items.

Sections can be added or removed from the section tree by using the “Section” menu, or the contextual menu in the section tree (right-click). The section tree can be rearranged via drag and drop.

Existing header items may be rearranged via the arrow buttons  in the right-bottom corner of the window, or deleted via the  button. Header items can also be copied, cut and pasted from section to section, or even from document to document.

New header items can be added by selecting “New header item...” in the “Section” menu, or by clicking the  button in the lower part of the window. This opens the following dialog:

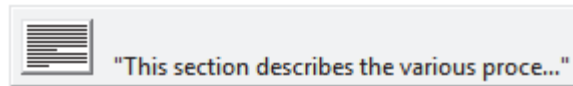


The upper zone allows to choose the type of header item to insert - text, publication, table, external image or external text file - and all required information for the header item:

- The parent diagram and the publication name for publication items. Note that only publications for currently opened diagrams are shown in the dialog, so the parent diagram for the publication to insert has to be opened before trying to add the header item.
- The initial number of rows and columns in table items.
- For external picture items, the file name for the referenced PNG image file and whether its path should be remembered as an absolute or a relative path from the document file's parent directory.
- For external text file items, the name of the file to include and the paragraph style to use for its text.

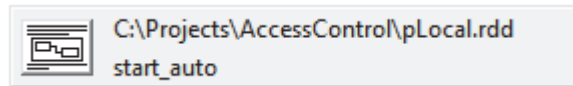
The lower zone in the dialog allows to specify where the new item will appear.

Once created, a text header item will appear like this:



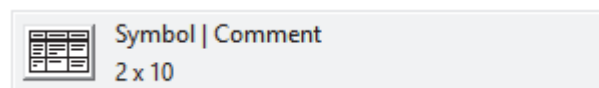
The text in the item is the beginning of the text actually entered in the item.

Publication header items will appear like this:



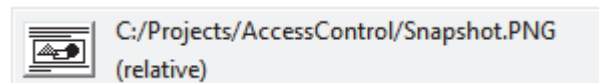
The first line of text is the diagram file name, and the second one is the publication name.

Table items will appear like this:



The first line includes the first table header line if any and the second one gives the table size.

External picture items will appear like this:



The first line is the full path of the referenced PNG file and the second line indicates if the reference is absolute or relative.

From the document editor window, it is also possible to export a document to a format that a word processor can handle. This is in fact the only way to get a form of the document that can be displayed or printed. The formats available today are:

- RTF, which can be used with all the major word processors;
- OpenDocument format, which is an ISO-standard for text documents, mainly used by OpenOffice.org and other open-source word processors;
- SGML; this format is for advanced users and won't be described further in this document. Please refer to the corresponding section in the reference manual for further details.

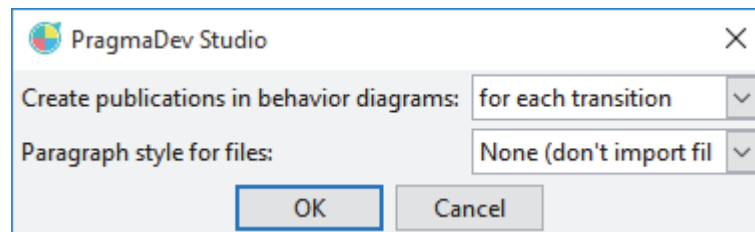
To be able to export a document in these formats, PragmaDev Studio needs to attach presentation attributes to the texts in the document. This is done via documentation display and export options, which are described in the sections "Documentation display options" on page 104 and "Documentation export options" on page 106 below.

5.2 - Full documentation generation

Sometimes, a project cannot be documented "on the fly", or just has not been. To be able to document all diagrams in a project more easily, PragmaDev Studio allows to automatically generate a document from a project. This feature will create all the necessary publi-

cations in all the diagrams in the project and gather them in a document with a standard structure.

To create a document automatically, just create a new empty document, open it, then select “Auto-generate from project...” in the “Document” menu. The following dialog is displayed:



The options in the dialog are the following:

- *Create publications in behavior diagrams*
Allows to specify the level at which the publications are created in behavioral diagrams such as processes or procedures. The available choices are:
 - *For each partition*
A publication will be created for each partition in the diagram, exporting all its symbols;
 - *For each state symbol*
A publication will be created for each state symbol in the diagram, exporting all the transitions connecting to it;
 - *For each transition*
A publication will be created for each input or continuous signal symbol in the diagram, exporting the transition attached to it.
- *Paragraph styles for files*
Allows to specify if the declaration files in packages should be included in the generated document or not, and if they should, which paragraph style to set for their text. The available choices for this options are *None (don't import files)* to prevent declaration files from being document, and all the available paragraph styles.

After validating the diagram, publications are automatically created in all diagrams in the project, the section tree is created in the document and publication items are created in all section headers. The structure will be created from the project tree, in the same order, but only including elements that actually have some contents. For example, if a package only contains C files, that are not documented, there will be no section for this package in the generated document. If one is needed, it can be added afterwards.

Note that publications are not always recreated by the generation process. If there is an existing publication that exports exactly the same set of symbols than the one that should be created, the document generation does not create anything and picks up the existing one. This allows to use the document generation of partially documented projects: If existing publications have attached texts before or after the exported symbols, these are kept and will end up in the final document. The diagrams can be reviewed after the generation to add missing texts, for example by using the documentation hints as described in section “Documentation hints” on page 65.

5.3 - Documentation display options

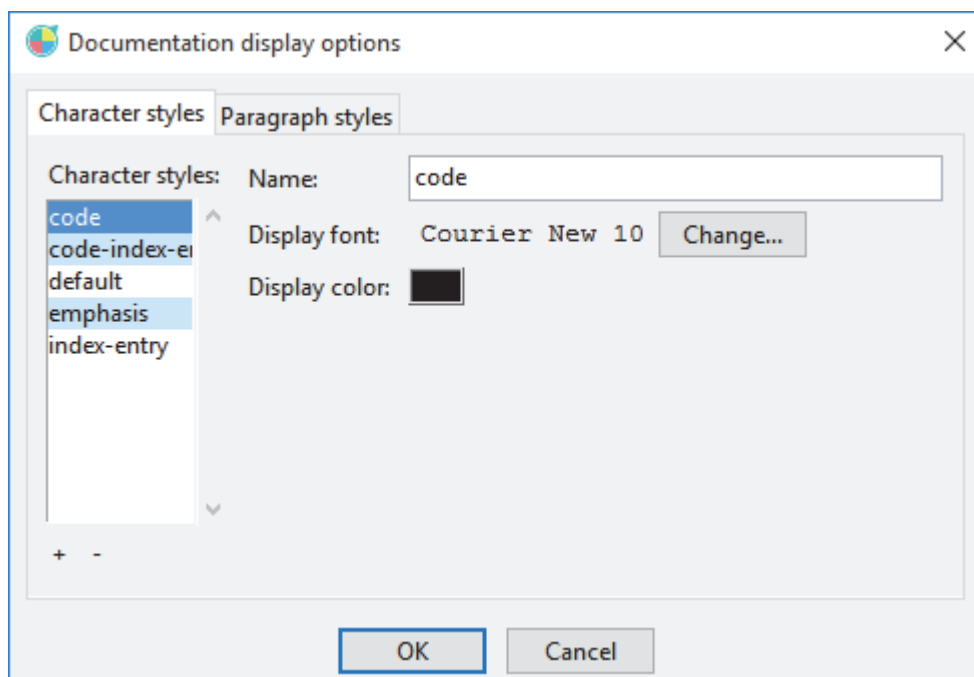
The documentation display options have two main objectives:

- Define what styles are available in publication texts and document section header text items;
- How these styles will be displayed on screen in the editors.

There are two types of styles:

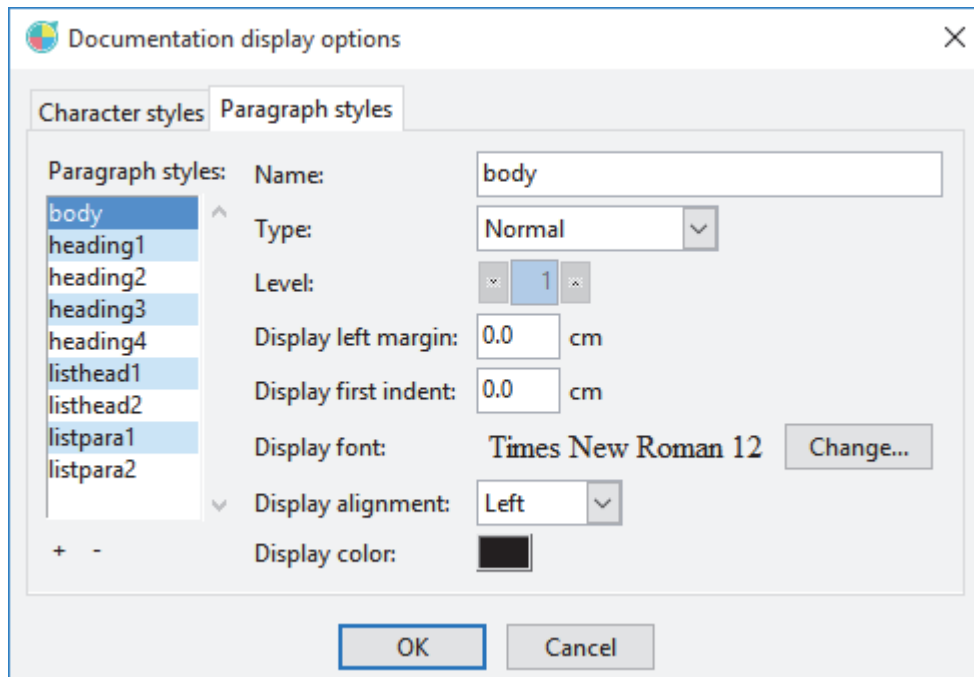
- Paragraph styles define the appearance for whole paragraphs. This concept is the same as the usual styles found in word processors. Typical display options for paragraphs include the font for the paragraph text, its margins, its alignment, and so on...
- Character styles define the appearance for range of characters within paragraphs. Display options for these styles are just a font (with its family, its size and its style) and a color.

Documentation display options are defined in a project, so that all styles will be available in all diagrams and all documents contained in the project. Setting these display options is done in the project manager window via the menu “Project / Documentation display options...”. The following dialog appears:



The first tab in the dialog allows to define character styles. Clicking on the “+” button under the list creates a new style by copying the selected one; clicking the “-” button deletes the selected style. The right part of the dialog allows to change the style name and to define the font and color for the characters with this style.

The second tab in the dialog allows to define the paragraph styles and looks like this:



Styles are created and deleted just as character styles. The style options are:

- *Name*
The name of the style.
- *Type*
A paragraph style may be either a normal paragraph in a text body, a list header, a paragraph inside a list or a style for section headings.
- *Level*
This option is not used for normal paragraphs in a text body. For the other types:
 - For section headings, the level is the section level;
 - For list headers or paragraphs in lists, the level is the list level.
 Please note that PragmaDev Studio is more strict than other word processors: it won't let you create a list with a level set to 3 after a normal paragraph, or directly inside a list with level 1.
- *Display left margin*
This is the distance in centimeters from the left border of the text editor to the position from which the text starts.
- *Display first indent*
It is the distance in centimeters from the left margin to the position from which the text of the paragraph's first line starts. For normal paragraphs, it is usually 0. For list headers, it is usually negative to add some space for the bullet or list number. It may also be used for section headings to set a fixed width for the section number.
- *Display font*
The font for its characters, including the font family, its size and its style (bold, italic, underlined).

- *Display alignment*

The alignment for the paragraph: left-aligned, right-aligned or centered. Please note that full justification is not available in the editors. It is only available in export styles (see “Documentation export options” on page 106).

- *Display color*

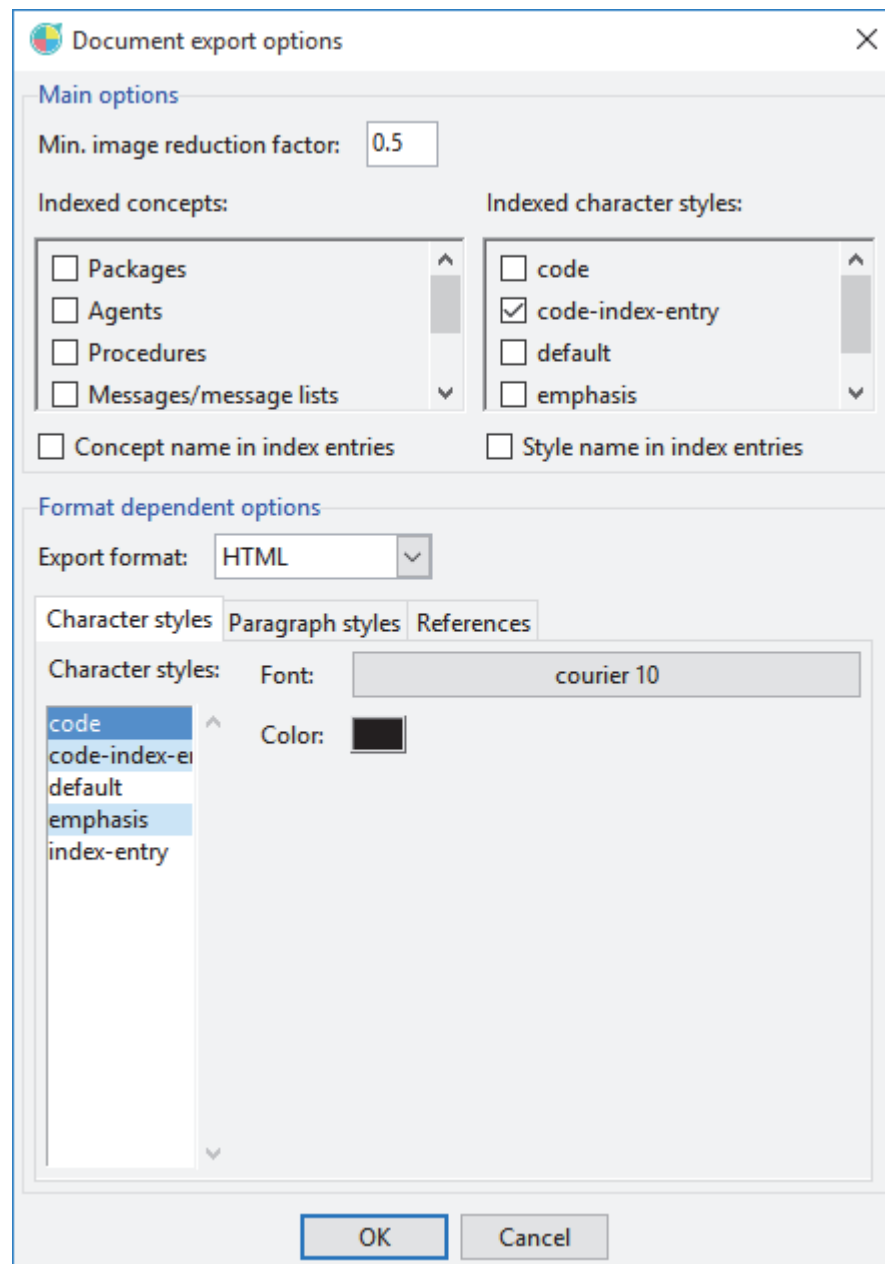
The color for the paragraph’s text.

NB: Styles for section heading paragraphs must be defined here, but they will not be directly available in editors, since the section headings should be defined via the document structure, and not directly within texts. The display options defined for these styles are also not significant today, as the document sections are never displayed as styled texts.

5.4 - Documentation export options

Display styles are only used when displaying styled texts on screen. When exporting a document to a format usable in a word processor, more options are available. These options are defined via the documentation export styles. The export styles are also attached to a project.

Defining export styles is done in the project manager via the “Project / Documentation export styles...” menu. This opens the following dialog:



The upper zone in the dialog defines some options that are not dependant on the export format:

- The minimum image reduction factor defines how images built from diagram publications will be sized when included in the exported document. The default is to adapt the image size to the width and/or height of the page. This option allows to specify that images should always be reduced by this factor before doing any other adaptation.
- The next two tables allows to automatically set up an index in the exported document:

- The left-most table defines what are the concepts that will automatically be included in the document index. These index entries will be figured out automatically by PragmaDev Studio each time an image from a diagram publication is exported. If this image includes any concept checked in this list, an index entry will be automatically set up, with the concept name as its text, referring this image in the document. Available concepts are packages, agents, procedures, messages and message lists, passive classes, class attributes, class operations and semaphores (in SDL-RT projects).
- The right-most table defines a set of character styles that will also automatically be included in the document index. The text for the entry will be the whole text with this style. This allows to explicitly define index entries in any text.

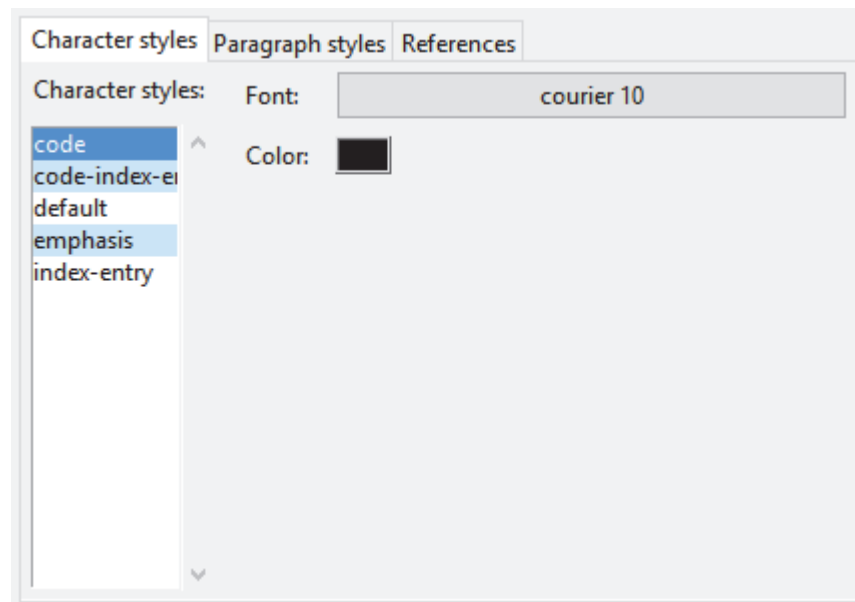
NB:

- This is actually the only way to define “custom” index entries today.
- This may lead to duplicate existing character styles to be able to define the index. For example, a character style can be defined for bits of code in a document, with typically a fixed-width font. If class, attribute or operation names are included in the text, and if the index should reference these names, it is possible to duplicate the code style to a style named `indexed_code` for example, and to use `indexed_code` for all these names.

Below these two tables are options to create “extended” index entries. These entries will not only include the text for the entry, but also a short description of what it is (concept name for concepts, style name for character styles).

The lower zone in the dialog defines the options that depend on the format used for export, arranged in tabs:

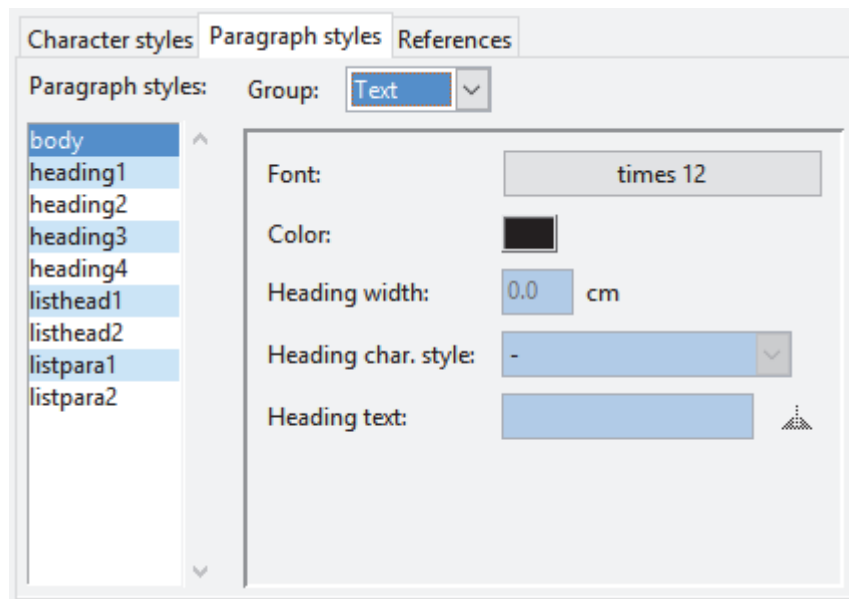
- The first tab shows the export options for the character styles:



The list in the left part shows the available character styles, as defined in the documentation display options. Please note that you cannot add, remove or rename styles here; they have to be removed from the display options (see “Documentation display options” on page 104).


The right part allows to change the actual export style for the character style, which is only the font and the color.

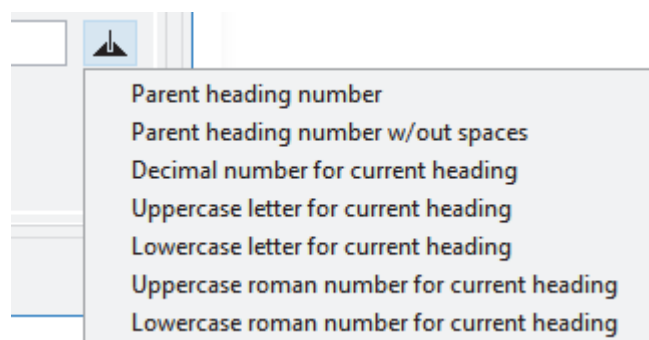
- The second tab shows the export options for the paragraph styles:



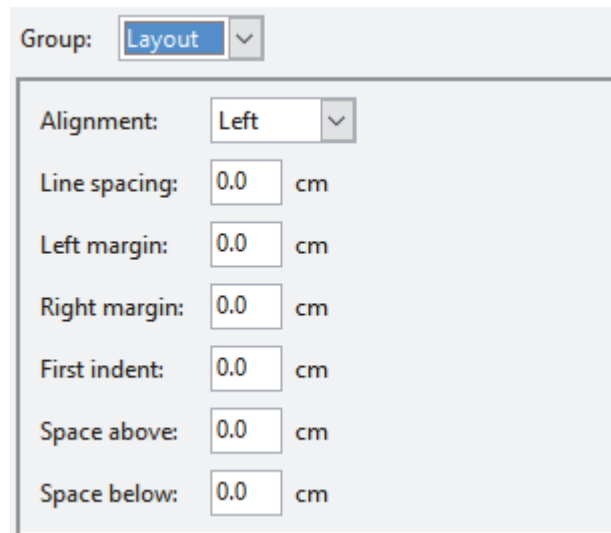
As for character styles, the list in the right part shows the list of available paragraph styles. Again, the available styles are defined via the documentation display options and cannot be added, removed or renamed here (see “Documentation display options” on page 104).

The right part shows the export options for the paragraph style. As there are many options, they are organized into groups:

- The *Text* group (shown above) includes the options for the text itself, including its font, its color, and the heading to automatically include in the paragraph if any. This heading is only available in section heading and list header paragraphs:
 - The heading width should be set only if the heading has a fixed width, typically for list headers.
 - The heading character style may be used to give a different style to the heading text. This may be typically used for list bullets.
 - The heading text may contain markers for the section or list number in many formats, and the number for the parent section or list. These markers don't need to be explicitly typed and can be entered by using the menu available via the  button:



- The *Layout* group includes the options for the text layout on the page:



The screenshot shows a dialog box with a 'Group:' dropdown set to 'Layout'. Below it, several settings are listed with input fields and units:

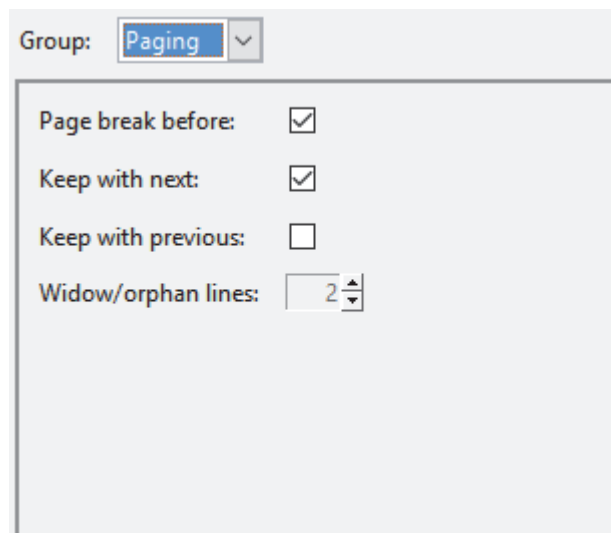
Property	Value	Unit
Alignment:	Left	
Line spacing:	0.0	cm
Left margin:	0.0	cm
Right margin:	0.0	cm
First indent:	0.0	cm
Space above:	0.0	cm
Space below:	0.0	cm

These options are quite usual:

- alignment (including the fully justified option),
- additional line spacing (that can be negative to make lines less high),
- left margin (from the page left border),
- right margin (from the page right border),
- first line indent (from the left margin),
- additional space above paragraph,
- additional space below paragraph.

All distances are specified in centimeters.

- The *Paging* group allows to specify the options that have an impacts on how page breaks will be handled:



The screenshot shows a dialog box with a 'Group:' dropdown set to 'Paging'. Below it, several settings are listed with checkboxes and a spinner:

Property	Value
Page break before:	<input checked="" type="checkbox"/>
Keep with next:	<input checked="" type="checkbox"/>
Keep with previous:	<input type="checkbox"/>
Widow/orphan lines:	2

- *Page break before* always forces a page break before a paragraph with this style.
- *Keep with next* forces the paragraph to be on the same page as the one following it.

- *Keep with previous* forces the paragraph to be on the same page as the one preceding it. Please note some formats or word processors may not be able to take this option into account.
- *Widow/orphan lines* specifies the minimum number of lines that are allowed to stay alone on one page for the paragraph.
- The third tab shows the format for cross-references. This format is not used today.

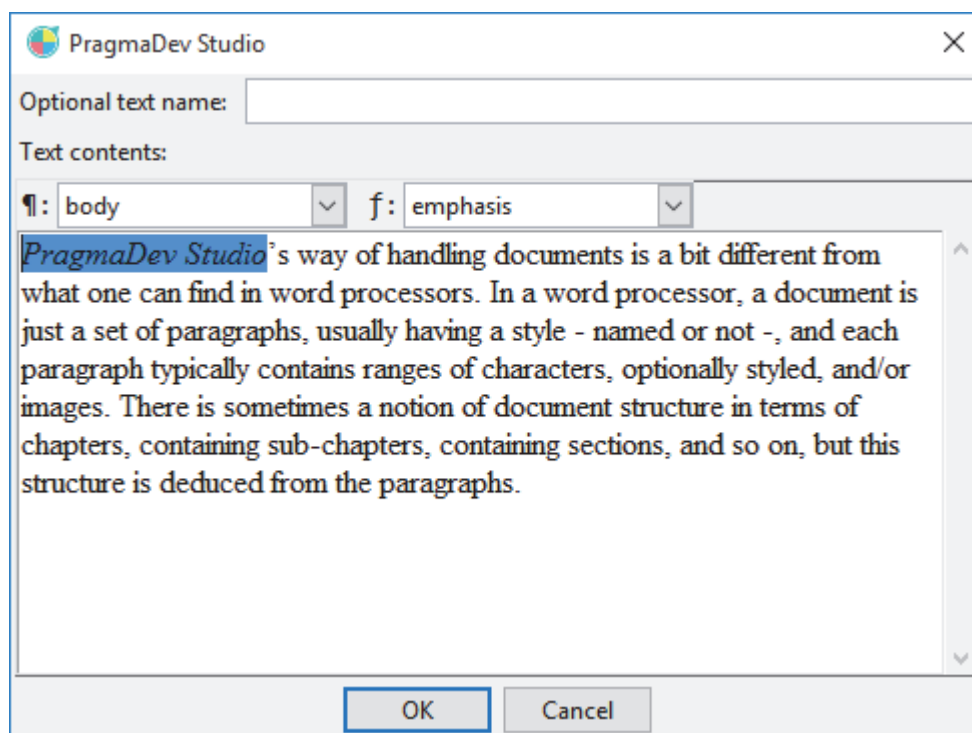
Export styles should be defined for all available styles in all export formats. If some export styles are missing, a warning will be displayed when any document is opened.

Opening the export options for the project will then create automatically the missing styles if you ask for it. Please note that the options set up for the export styles are copied from the display style if available, or set to default values. This may not be what you want, so the automatically created styles should be reviewed and updated where needed.

NB: As PragmaDev Studio works today, export styles are not used when exporting to OpenDocument format. This kind of export is actually based on a document template, from which the styles are imported. So you just have to make sure that all character and paragraph styles in PragmaDev Studio also exist in the template document with the same name.

5.5 - Styled text editor

Once the styles are set up, they will be available in all editors for texts, either in publications or in section header items. Such an editor is shown below:



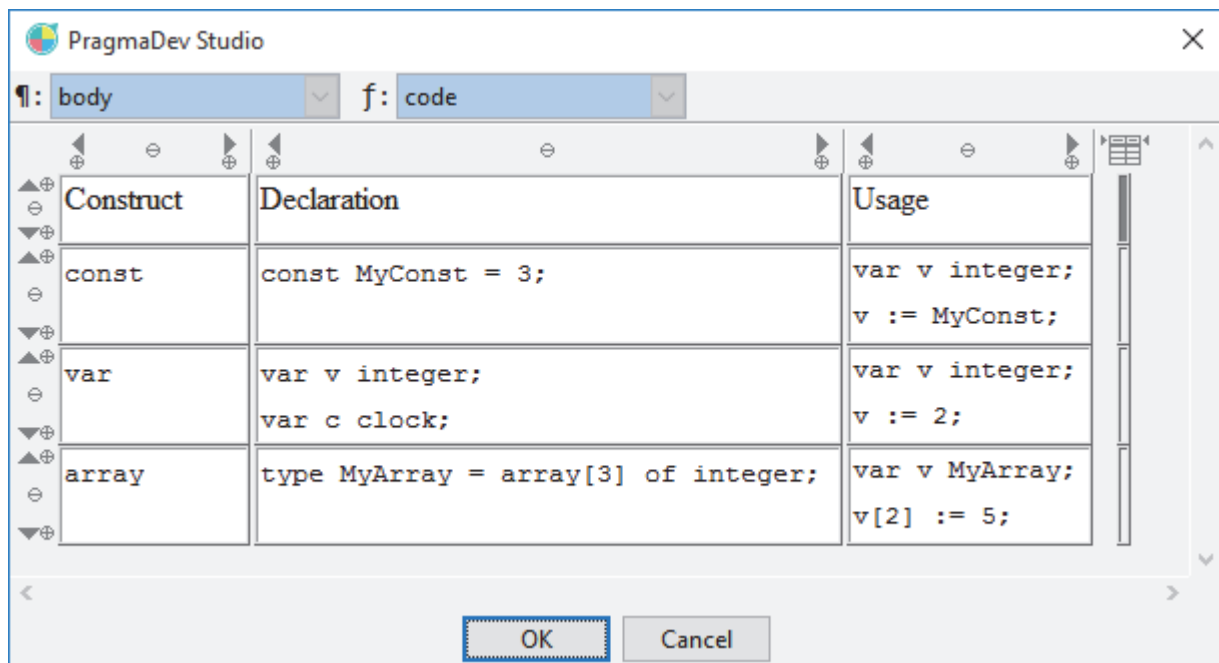
The bottom zone contains the actual text. The menus at the top allow to select:

- The paragraph style ("¶"). This style is applied to the current paragraph or to all selected paragraphs.
- The character style ("f"). This style is applied to the selected characters.

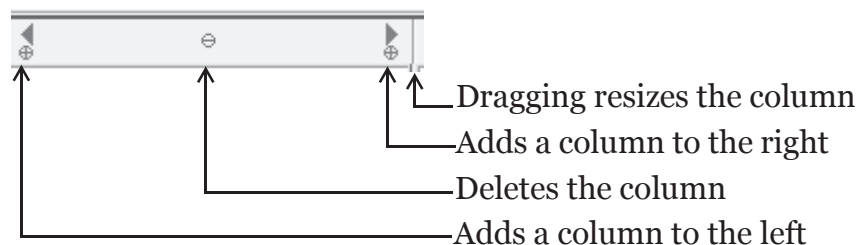
Note that not all paragraph styles are available in all contexts: as said in “Documentation display options” on page 104, PragmaDev Studio is more strict than usual word processors and will not allow to create inconsistent list nesting (like a paragraph inside a list with level 2 just after a list header with level 1). Trying to do an operation on the text that would create such an inconsistency will be refused.

5.6 - Table editor



Double-clicking on a table header item in a document section opens it in an editor such as the following one:



All table cells are editable as normal styled texts, except the paragraph and character style selectors are above the table. Navigating through table cells via the tab and shift-tab keys is possible. Columns can be manipulated via their header:



Adding and deleting rows is done via the similar buttons in the row header.

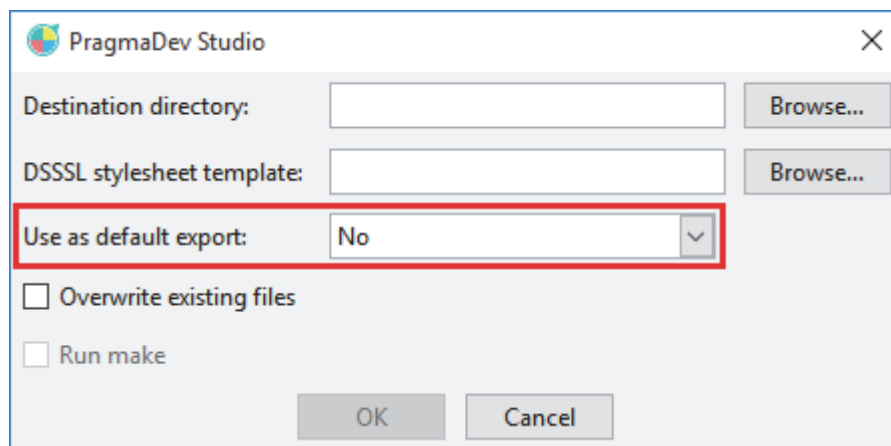
The special column  allows to define the header rows for the table: clicking on the  for a given row makes it the last row in the table header.

5.7 - Exporting document

Documents can be exported to 4 types of documents:

- Documents in Rich Text Format (RTF). This document format is accepted as input by a lot of word processor applications. The export for this format is totally handled by PragmaDev Studio.
- Documents in Open Document Text format (ODT). This format is the native one for Open Office and its derivatives. The export for this format requires a template actually defining the styles used in the document.
- Documents as a HTML page, or a set of HTML pages. This export can be based on a template defining the layout of the final document.
- Documents in Standard Generalized Markup Language (SGML). This export is for advanced users and will not be described in this manual. It is detailed in PragmaDev Studio Reference Manual.

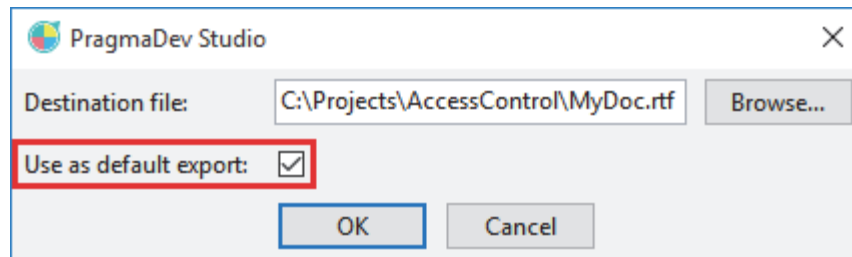
It is possible to record for a document a default export, that will record all the necessary information: document format, destination, and template if any. This default export can be specified either when exporting, or via the export options dialog. To specify the current export as the default while exporting, all export dialogs include the following choice:



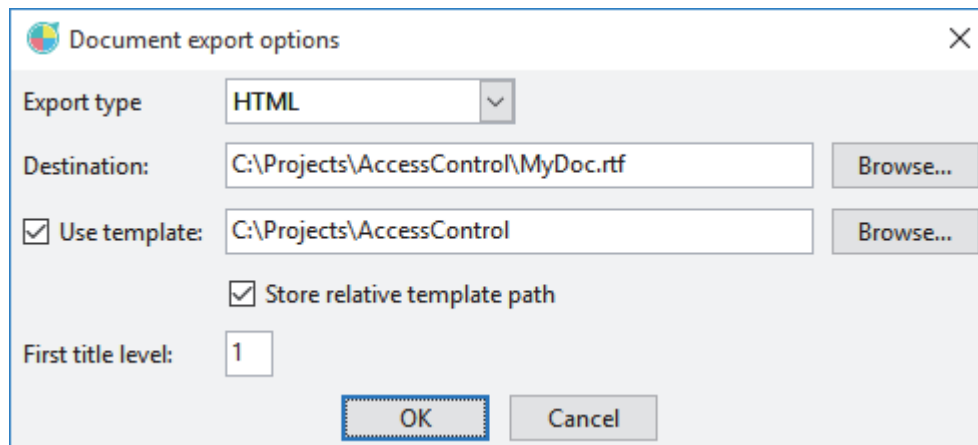
The options are:

- 'No' to perform the export only and not record the current export as the default one.
- 'Yes - absolute template path' to export and record the current export as the default export, storing the path to the template as an absolute path.
- 'Yes - relative template path' to export and record the current export as the default export, storing the path to the template as a relative path from the document path.

Note: this choice is only a checkbox for exports not supporting templates, such as RTF:



The document export options dialog is opened by selecting “Export options...” in the “Document” menu:



The available options are:

- *The export type*
HTML, Open Document, RTF or SGML.
- *Destination*
It can be either a file or a folder depending on the export type.
- *Use template*
Whether a template should be used or not, if applicable, and the template file itself. The template is not available for RTF exports, required for Open Document and SGML exports, and optional for HTML export.
- *Store relative template path*
An indicator specifying if the path to the template should be stored as a relative path from the document path, or as an absolute path.
- *First title level*
The first level of titles to export. This option can be used when the exported document is integrated in a larger document. If set to 2, for example, all sections at the highest level in the document will be exported using a style for the second level of heading, as defined in the documentation options, all sections under these at level 3, and so on.

5.7.1 Exporting as RTF

Exporting a document as an RTF file is quite straightforward: just select “Export as” -> “RTF” in the “Document” menu of the document editor window. PragmaDev Studio will ask for the file to export to, then export the document to that file.

5.7.2 Exporting as OpenDocument format

As said in “Documentation export options” on page 106, PragmaDev Studio does not actually use the export styles defined in the document when exporting to OpenDocument format. The styles are taken from a template, which is another file in OpenDocument format. So selecting “Export as” -> “OpenDocument” in the “Document” menu in the document editor will open a dialog asking for the destination file and the document template. Once both files are specified, PragmaDev Studio will export the document to the specified destination file.

5.7.3 Exporting as HTML

There are some limitations on the options actually taken into account for styles when exporting to HTML:

- The minimum image reduction factor is actually used to create thumbnails for images in the document. If set to less than 1, thumbnails will be created and inserted in the exported HTML file, with a link to the actual image with the full size.
- The font and text color specified in paragraph styles is not used today.
- The heading width is never used.
- Heading texts are only used for section headings. Those specified for lists are not used, as HTML provides its own list bullets or numbers. PragmaDev Studio only uses the heading text to try to figure out if a list should be bulleted or numbered.
- The *Full* alignment is replaced by *Left*, as full justification is not available in HTML.
- All margins and spacings are not used.
- The *Page break before* option is only used in some cases (see below).
- The other options related to pagination are not used as they are meaningless in HTML.

There are actually two kinds of HTML exports available:

- The first is the basic one, exporting the whole document to a single HTML file. In this mode, no table of contents or index is generated. This is the mode used when no template is specified for the export.
- The second mode allows much more control over the files that will be actually exported, and must be used if a table of contents or index is needed. This kind of export is based on templates.

Both modes actually export several files, including at least the main HTML file and the exported images (in PNG format). So exporting to HTML from the document editor will

actually ask for a destination directory where all the files will be created. To prevent files from being overwritten by the export, PragmaDev Studio will issue a warning if the chosen directory is not empty.

As said above, the mode allowing the greater amount of control over the exported file is template-based. A template consists in one or several HTML files containing specific tags that will be replaced by PragmaDev Studio during the export process. Available tags are:

- `<!--%REFTMPL[xxx.html]-->`
Indicates that the template references the file `xxx.html`, which is another template. This tag will typically be used for files referenced by the current one in any way, for example via a link or in a frame. This tag will not be replaced in the final exported document.
- `<!--%REFFILE[xxx.yyy]-->`
Indicates that the template references the file `xxx.yyy`, which is not a template and should not be parsed, but just copied in the destination directory. This tag will typically be used for included images for example. This tag will not be replaced in the final exported document.
- `<!--%TOC[target=xxx,indent=yyy]-->`
This tag will be replaced by the document's table of contents. The part "target=xxx" is optional and may be used to indicate the target for the links generated for the section titles (attribute target for HTML hyperlinks). The part "indent=yyy" is also optional and indicates the indentation width between two section levels. The value yyy should be a valid table column width specification (attribute width for HTML tag td).
- `<!--%INDEX[target=xxx]-->`
This tag will be replaced by the document index. The part "target=xxx" has the same semantics as the same part in the %TOC tag for the links generated for index entries.
- `<!--%DOC-->`
This tag will be replaced by the whole document contents. This contents will include anchors for the hyperlinks generated in the table of contents and the index if any.
- `<!--%PAGE-->`
This tag will trigger a multiple export for the file containing it. The file will be copied as many times as there are pages in the document. A suffix containing the page number will be added to each copy. The page break will be triggered by the attribute *Page break before* set in the export options for paragraphs.
The generated contents for all pages will include the destination anchors for all hyperlinks generated in the table of contents and index. A template should therefore either contain a %DOC tag, or a %PAGE tag. If it contains both, the behavior is undefined.
- `<!--%NEXTPAGE-->` and `<!--%/NEXTPAGE-->`
This two tags will be replaced respectively by the open tag and the close tag for the hyperlink to the next page if any. These tags should only appear in the same template as the %PAGE tag.

- `<!--%PREVPAGE-->` and `<!--%/PREVPAGE-->`

Same as the NEXTPAGE tags, but for the previous page.

All these tags must be alone on a line, with only whitespace before or after it, but not within.

The template is actually composed of the top-level template file specified in the HTML export dialog, plus all the file it references via a %REFTMPL or a %REFFILE tag, plus all files referenced via a %REFTMPL or a %REFFILE tag in these ones, etc. All these files will be copied to the destination directory for the export, and only these ones. So any file actually or potentially used by any of the templates *must* be referenced via a %REFTMPL or a %REF-FILE tag, or it won't be copied and will be unavailable in the exported document.

An example template for HTML export is available in PragmaDev Studio example projects, in \$RTDS_HOME/examples/Specifier/AccessControl.

5.8 - Using exported documents

When exporting a document to any format, PragmaDev Studio will only export the document body. Usual things like a title page, a table of contents or the document index cannot be created by PragmaDev Studio, since it can't know how to format them, or place them in the document.

It is however possible to create these items and link them with the document body created by PragmaDev Studio in all tools. This section presents how to do such a thing with the two major word processors: Microsoft Word and OpenOffice.org.

5.8.1 In Microsoft Word via RTF

Dynamically adding things to the part of the document generated by PragmaDev Studio in Microsoft Word is quite easy:

- Export the document body to a RTF file using the “Document” -> “Export as” -> “RTF” menu in the document editor.
- In Microsoft Word, create a new empty document.
- Define the page layout that your document should have: page size, margins, header, footer and so on...
- Add to the document all pages that should be inserted before the document body created by PragmaDev Studio. This would typically be a title page and a table of contents for example. If created here, the table of contents will be empty.
- At this point, do a dynamic insertion of the RTF file generated by PragmaDev Studio by selecting the “Insert” -> “File...” menu. In the file selection dialog that appears, select RTF in the file types menu in the bottom, then select the RTF file created by PragmaDev Studio. Then open the menu attached to the “Insert” button and select “Insert as link”. This will dynamically import the RTF file within the Word document.
- After that, create all items that should go after the document body in your document, typically the index.

- Select everything in your document and ask to update the fields via the F9 key. If asked whether to update only page numbers or the whole table or index, select the second option.

That's all; you should now have a full document including everything needed. Whenever your document changes in PragmaDev Studio, just export it again to the same RTF file, then open the Word document, select all and press F9.

5.8.2 In OpenOffice.org via OpenDocument format

Defining the items that should be added to the document body exported by PragmaDev Studio in OpenOffice.org is done via a specific document type called a master document. This kind of documents may contain text and refer to other external documents as well. To define such a document, just select "New" -> "Master document" in OpenOffice.org Writer "File" menu. This will display a regular document window with a navigator window near it. This navigator window allows to add items in the master document.

The page setup - page size, margins, header(s), footer(s), and so on... - should be done directly in the master document.

Regular text, such as the title page, can be added directly in the master document window as in any regular document. Creating such a text will create a line labelled "Text" in the navigator window.

Inserting a table of contents should be done via the navigator window by selecting the line before the point where it should be inserted and selecting "Index" in the "Insert" menu. The standard index / table of contents insertion dialog will appear; just set up the options as you would in a normal document.

To include the document body exported by PragmaDev Studio in the master document, select the item before the insertion position and select "File..." in the "Insert" menu. Select the file exported by PragmaDev Studio and validate.

To add an index, the operations are the same as those to insert a table of contents.

Note: Once created, the master document will include a copy of all character and paragraph styles that were in the document exported by PragmaDev Studio when it was inserted. Any following changes will only have an impact in the exported document, and *not* in the master document including it.

5.9 - Questions and answers

This section includes a number of questions that may arise when using the documentation system in PragmaDev Studio with their answers.

Q:How can I create a "bold" or "italic" character style as in a word processor?

A: If the character style should just put the characters in boldface or in italic without any font change, you can't: character styles in PragmaDev Studio *always* include a font family. If you plan to use the style in a regular text, just set the same font family as in your default paragraph style. If you mix several font families within the same

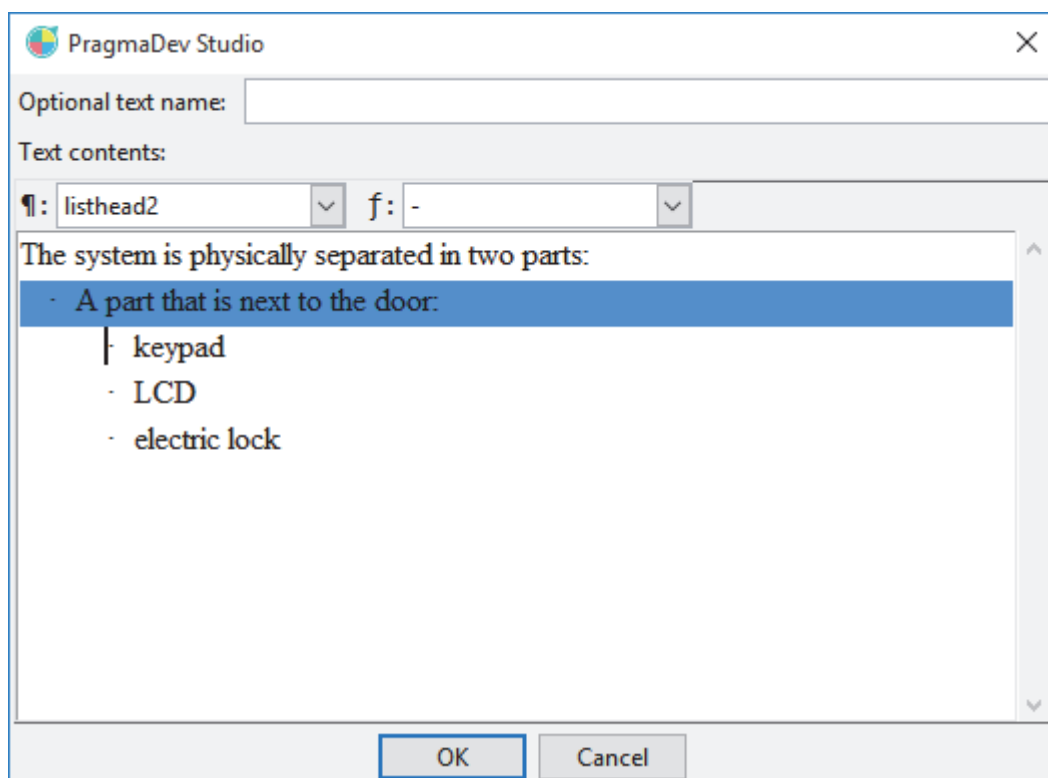
document, you'll have to define several "bold" or "italic" character styles: one per font family that you're using.

Q: Several paragraph styles I've defined in the documentation display options do not appear in the styled text editors. Why?

A: All paragraph styles marked as section headings in display options (*Type* is *Heading*) are intended to be used internally by PragmaDev Studio for section titles when it exports a document to a given format. They should not be used directly in texts, so they won't appear in the paragraph styles menu in the styled text editors.

Q: I've selected some paragraphs in a styled text editor that I want to delete, but nothing happens when I hit the "Del" or "Backspace" key!

A: As said in "Documentation display options" on page 104, PragmaDev Studio is quite strict when handling list nesting: you can't have a list with level 2 directly appearing after a normal paragraph, or have a paragraph declared as inside a list with level 2 appear after a list header with level 1. This is required to be able to export to structured document types such as HTML. If an operation would create an inconsistency in list nesting, PragmaDev Studio will forbid it. So for example, if you have the following selection:



PragmaDev Studio will forbid the deletion, since it would make the "keypad" list header with level 2 appear just after the paragraph "The system is physically separated in two parts:", which is a normal paragraph not inside a list. So this would create an inconsistency in list nesting. Trying to delete this selection will therefore do nothing.

To be able to delete these paragraphs, you first have to change the styles for the paragraphs after the selection or before it to ensure that list nesting will be consistent after the deletion.

6 - Requirements Table Editor

6.1 - Principles

Unlike other types of elements, requirements are not supposed to be created within PragmaDev Studio. Following general usage, the requirements have to be created within a spreadsheet application and exported in a format readable by PragmaDev Studio. The generated file can then be imported in PragmaDev Studio as a requirements table.

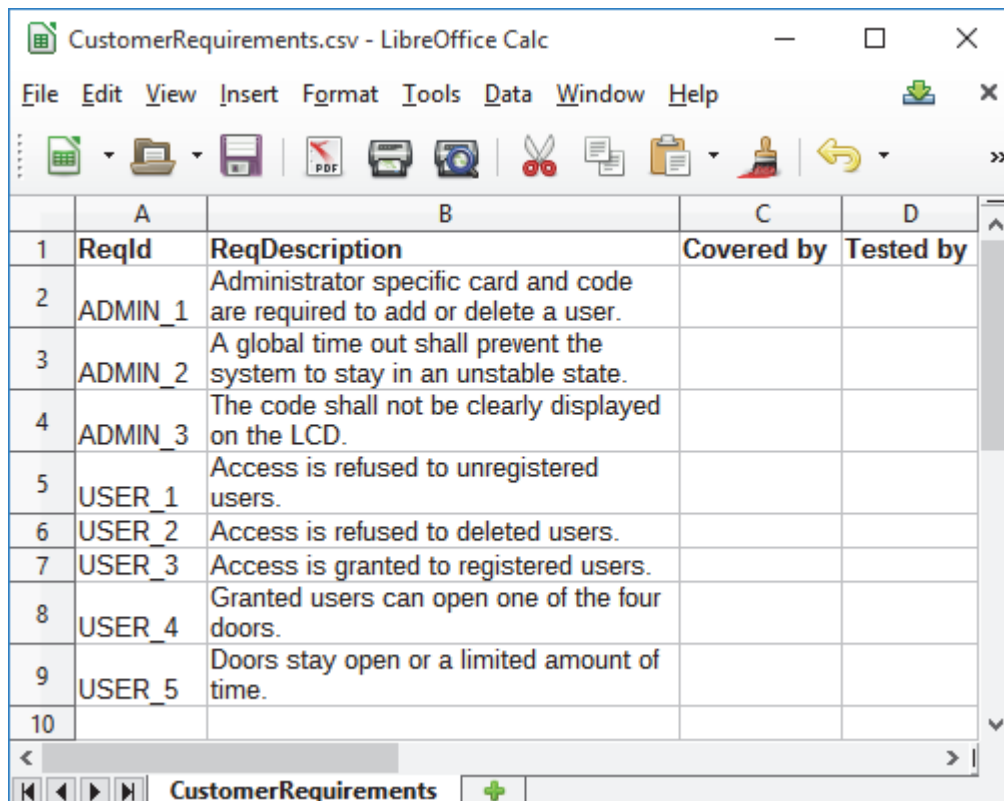
The formats supported by PragmaDev Studio are the text formats supported by Microsoft Excel:

- The CSV format, where each row is on its own line in the exported file, the cells separated by a semicolon (;) and double-quoted when necessary.
- The text format, where each row is also on its own line in the exported file, and the cells separated by a tab character and not quoted in any way.

PragmaDev Studio identifies the format with the extension of the file: .csv or .txt. Of course, requirements tables can be created with any other spreadsheet application, as long as they are exported using one of the 2 formats above.

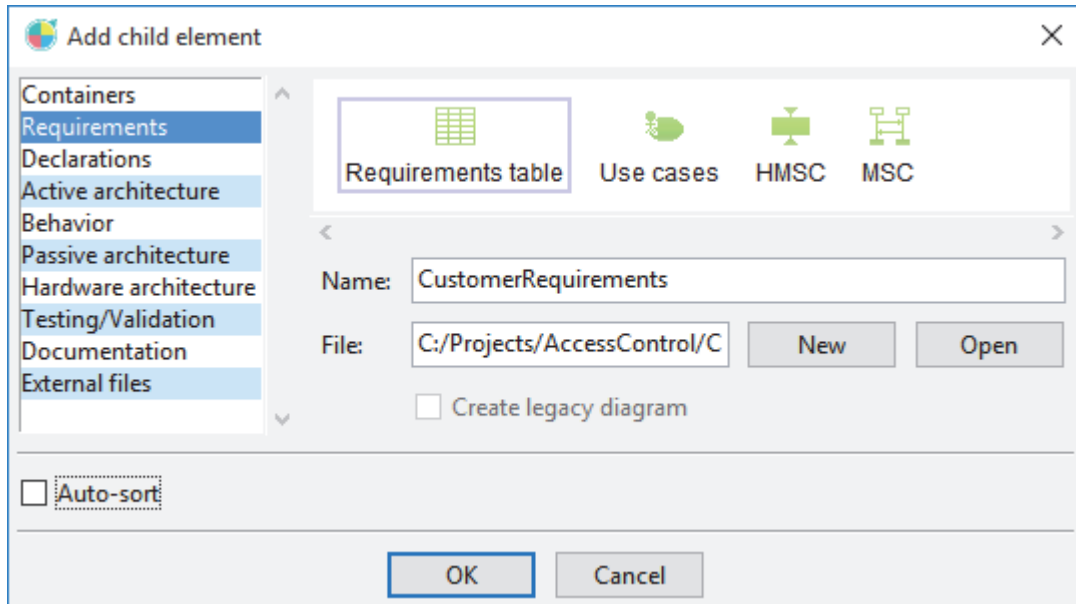
The requirements table must have at least 2 columns and at most 4. The first row is considered by PragmaDev Studio as the title row, all others describing one requirement. The first column is the requirement identifier, and the second one its description. Only the title is considered for the 2 last columns: PragmaDev Studio will use them to store its own information: the symbols and testcases covering the requirement; see “Covering symbols” on page 123 and “Covering testcases” on page 126.

Here is an example of a requirements table created in LibreOffice Calc:

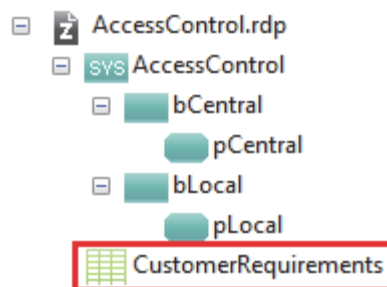


	A	B	C	D
	ReqId	ReqDescription	Covered by	Tested by
2	ADMIN_1	Administrator specific card and code are required to add or delete a user.		
3	ADMIN_2	A global time out shall prevent the system to stay in an unstable state.		
4	ADMIN_3	The code shall not be clearly displayed on the LCD.		
5	USER_1	Access is refused to unregistered users.		
6	USER_2	Access is refused to deleted users.		
7	USER_3	Access is granted to registered users.		
8	USER_4	Granted users can open one of the four doors.		
9	USER_5	Doors stay open or a limited amount of time.		
10				

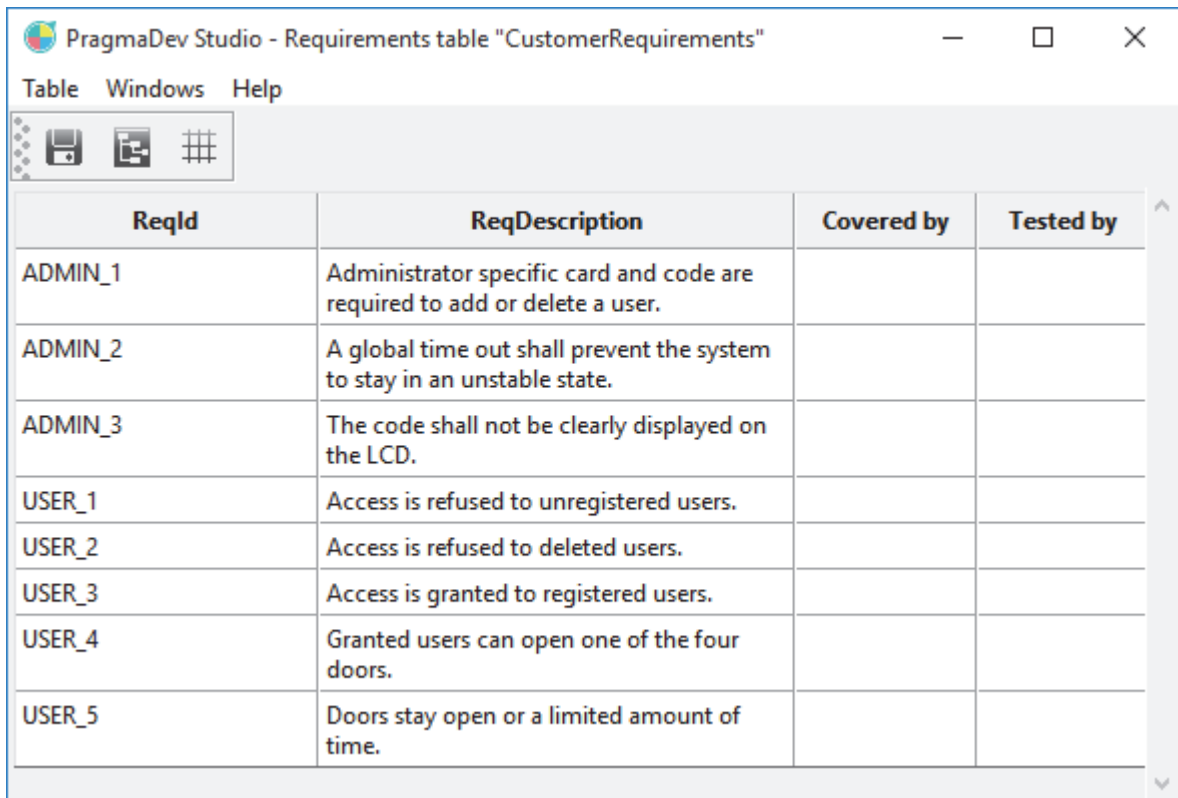
Once exported in CSV format with the parameters specified above, it can be inserted in a PragmaDev Studio project the usual way: select the parent node for the requirements table (usually the project node), then use the “Add child element...” item in the contextual menu or via the menu “Element / Add child...”. Then select the “Requirements table” in the “Requirements” category, and choose the exported CSV file via the “Open” button:



Once added, the table will appear in the project:



and double-clicking on the corresponding node will open the requirements table editor:



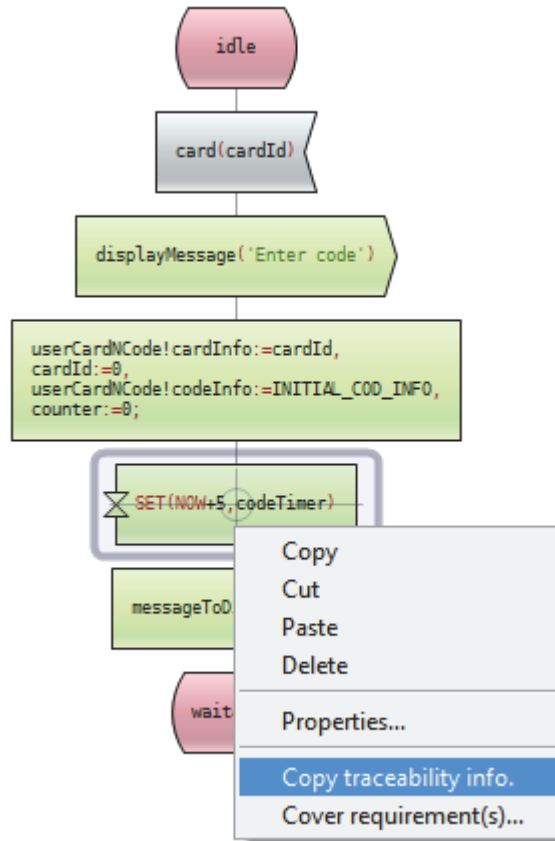
ReqId	ReqDescription	Covered by	Tested by
ADMIN_1	Administrator specific card and code are required to add or delete a user.		
ADMIN_2	A global time out shall prevent the system to stay in an unstable state.		
ADMIN_3	The code shall not be clearly displayed on the LCD.		
USER_1	Access is refused to unregistered users.		
USER_2	Access is refused to deleted users.		
USER_3	Access is granted to registered users.		
USER_4	Granted users can open one of the four doors.		
USER_5	Doors stay open or a limited amount of time.		

Note that requirements cannot be added to or deleted from the table within the editor. These operations have to be done in the original spreadsheet file. Updating the requirement identifiers or descriptions is possible by double-clicking on the corresponding cell.

Once the table has been inserted in the PragmaDev Studio project, it is possible to associate to each requirement the symbols covering it. After that is done, and if TTCN testcases have been written for the designed system, PragmaDev Studio can itself extract from the code coverage information gathered during a cosimulation of the system and its testcases the testcase coverage for each requirement. These topics are described in the next sections.

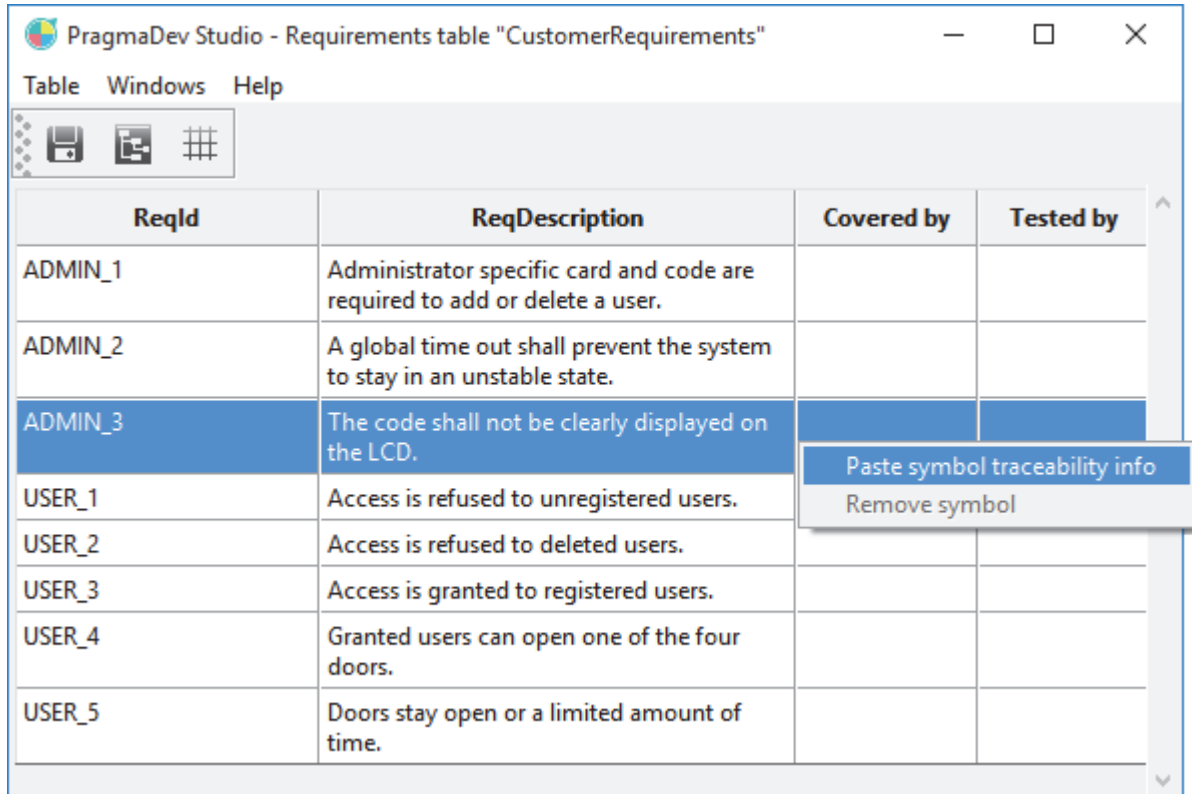
6.2 - Covering symbols

Specifying covering symbols for requirements is done via a copy/paste operation: the symbol must be selected in its parent diagram, its traceability information copied, and then pasted in the requirements table. Here are the steps in detail:



In an editor diagram, when a symbol is selected, an item “Copy traceability info.” is available in the contextual menu as well as in the “Edit” menu. Note that the traceability information for a symbol has no actual representation and cannot be pasted anywhere else than in a requirements table. The paste must also be done within the same PragmaDev Studio session as the copy. The requirements table must be in the same project as the symbol.

Once copied, the traceability information can be pasted in the requirements table by using the contextual menu on the table's third column:



PragmaDev Studio - Requirements table "CustomerRequirements"

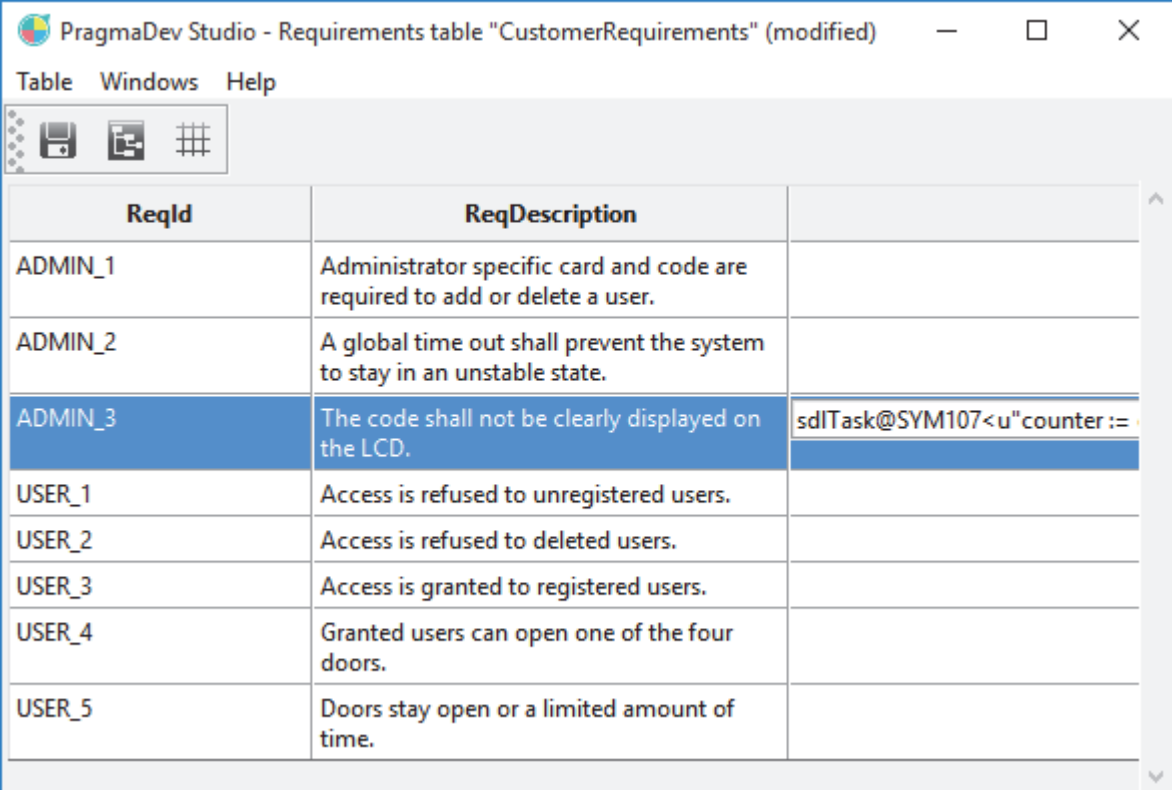
Table Windows Help

ReqId	ReqDescription	Covered by	Tested by
ADMIN_1	Administrator specific card and code are required to add or delete a user.		
ADMIN_2	A global time out shall prevent the system to stay in an unstable state.		
ADMIN_3	The code shall not be clearly displayed on the LCD.		
USER_1	Access is refused to unregistered users.		
USER_2	Access is refused to deleted users.		
USER_3	Access is granted to registered users.		
USER_4	Granted users can open one of the four doors.		
USER_5	Doors stay open or a limited amount of time.		

Contextual menu options:

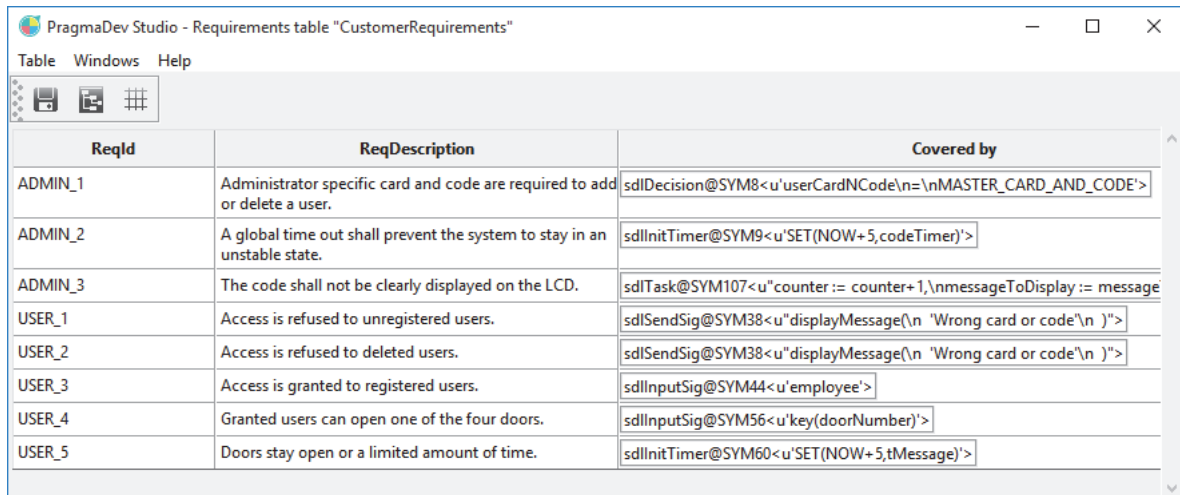
- Paste symbol traceability info
- Remove symbol

Once pasted, the covering symbol will appear as a text containing the file name for its parent diagram, its unique identifier and the beginning of its text:




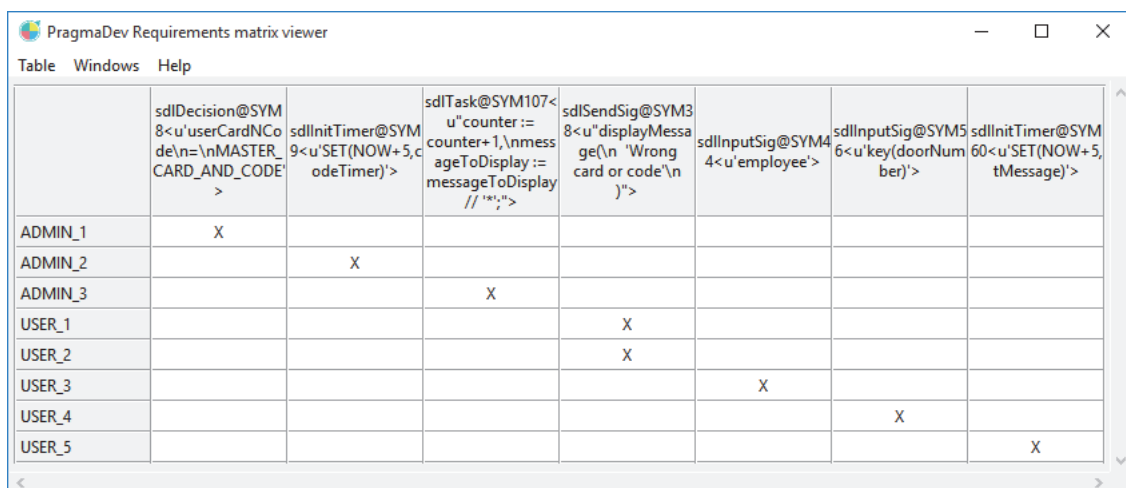
ReqId	ReqDescription	
ADMIN_1	Administrator specific card and code are required to add or delete a user.	
ADMIN_2	A global time out shall prevent the system to stay in an unstable state.	
ADMIN_3	The code shall not be clearly displayed on the LCD.	sdITask@SYM107<u"counter:=
USER_1	Access is refused to unregistered users.	
USER_2	Access is refused to deleted users.	
USER_3	Access is granted to registered users.	
USER_4	Granted users can open one of the four doors.	
USER_5	Doors stay open or a limited amount of time.	

Several symbols can be declared as covering the same requirement. In that case, the labels for the symbols will all appear in the third table column:



ReqId	ReqDescription	Covered by
ADMIN_1	Administrator specific card and code are required to add or delete a user.	sdlDecision@SYM8<u'userCardNCode\n=\nMASTER_CARD_AND_CODE'>
ADMIN_2	A global time out shall prevent the system to stay in an unstable state.	sdlInitTimer@SYM9<u'SET(NOW+5,codeTimer)'>
ADMIN_3	The code shall not be clearly displayed on the LCD.	sdlTask@SYM107<u"counter := counter+1,\nmessageToDisplay := messageToDisplay" >
USER_1	Access is refused to unregistered users.	sdlSendSig@SYM38<u"displayMessage(\n 'Wrong card or code'\n)">
USER_2	Access is refused to deleted users.	sdlSendSig@SYM38<u"displayMessage(\n 'Wrong card or code'\n)">
USER_3	Access is granted to registered users.	sdlInputSig@SYM44<u'employee'>
USER_4	Granted users can open one of the four doors.	sdlInputSig@SYM56<u'key(doorNumber)'>
USER_5	Doors stay open or a limited amount of time.	sdlInitTimer@SYM60<u'SET(NOW+5,tMessage)'>

PragmaDev Studio also offers another representation of the covering of the requirements by the symbols via the requirement matrix, that can be generated and displayed via the “Table / Generate requirement matrix...” menu, or simply by clicking the  button:



	sdlDecision@SYM8<u'userCardNCode\n=\nMASTER_CARD_AND_CODE'>	sdlInitTimer@SYM9<u'SET(NOW+5,codeTimer)'>	sdlTask@SYM107<u"counter := counter+1,\nmessageToDisplay := messageToDisplay" >	sdlSendSig@SYM38<u"displayMessage(\n 'Wrong card or code'\n)">	sdlInputSig@SYM44<u'employee'>	sdlInputSig@SYM56<u'key(doorNumber)'>	sdlInitTimer@SYM60<u'SET(NOW+5,tMessage)'>
ADMIN_1	X						
ADMIN_2		X					
ADMIN_3			X				
USER_1				X			
USER_2				X			
USER_3					X		
USER_4						X	
USER_5							X

Each line in the matrix represents a requirement, and each column a symbol. An ‘X’ in the cell indicates that the requirement is covered by the symbol.

6.3 - Covering testcases

Once the symbols covering the requirements have been specified in the requirements table, PragmaDev Studio can automatically extract the information of the covering testcases from a cosimulation of SDL and TTCN. This extraction is done in 2 steps:

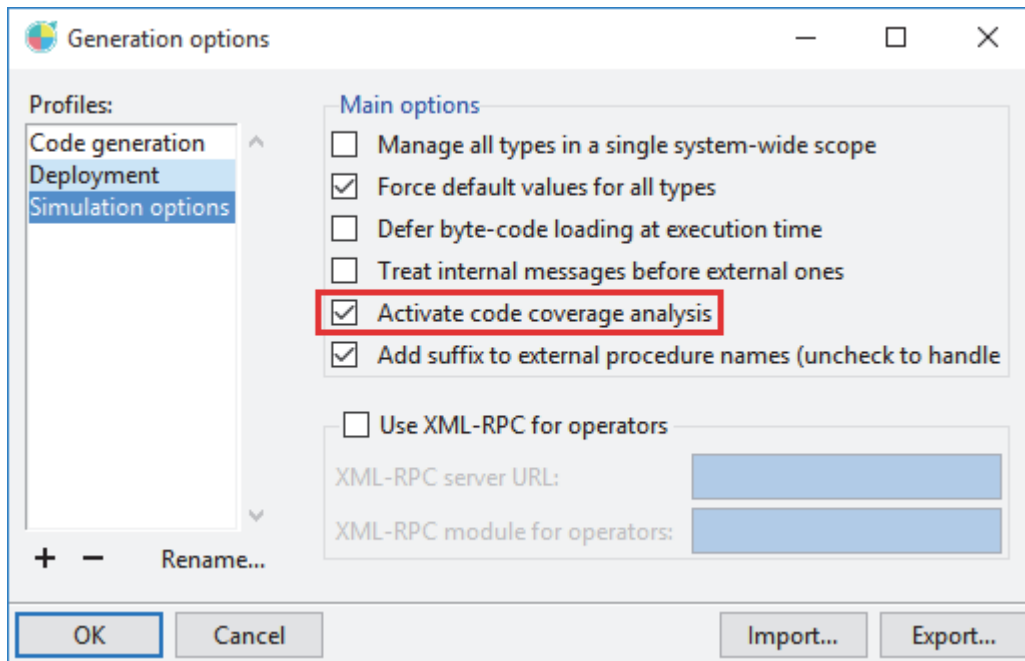
- A co-simulation must be run on the TTCN testcases and the system they test with code coverage analysis turned on, as explained in “Generating code coverage results” on page 325. The code coverage results must then be extracted.

- The results of the code coverage analysis must be imported in the requirements table.


These two steps are described in the following sections.

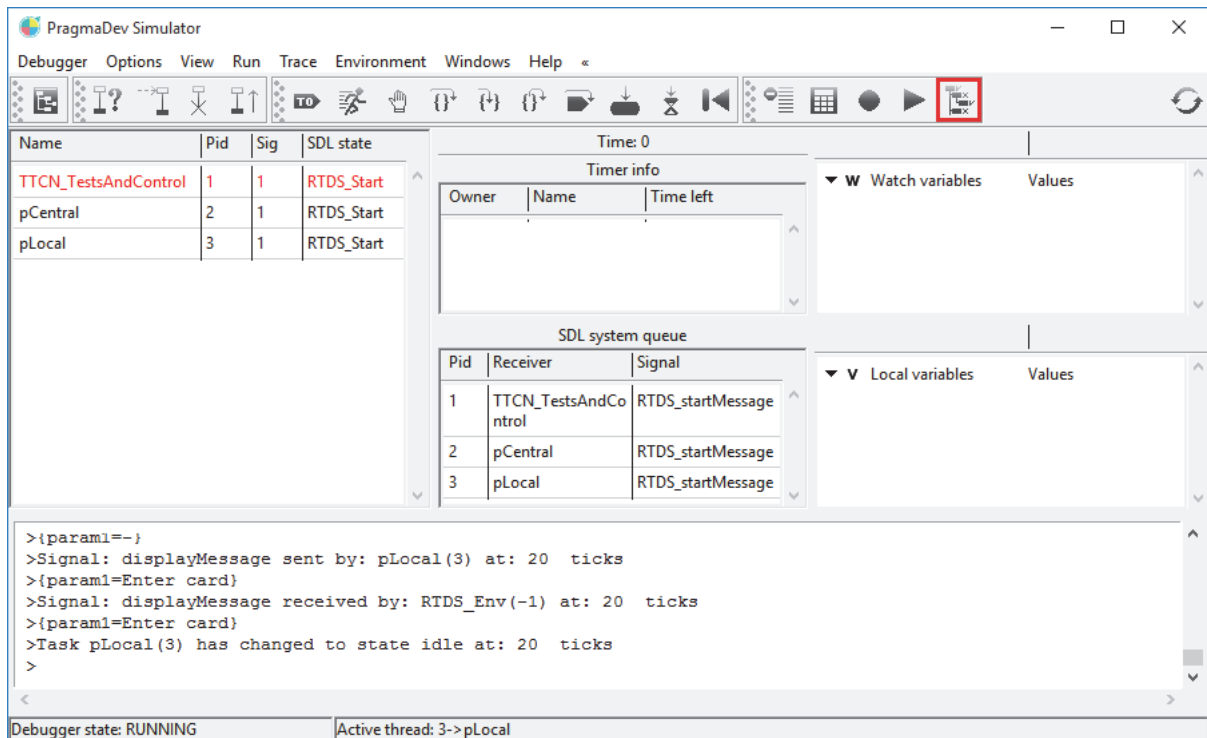
6.3.1 Co-simulation code coverage analysis

To allow PragmaDev Studio to extract this information, a simulation must be run using a simulation profile with code coverage analysis turned on:

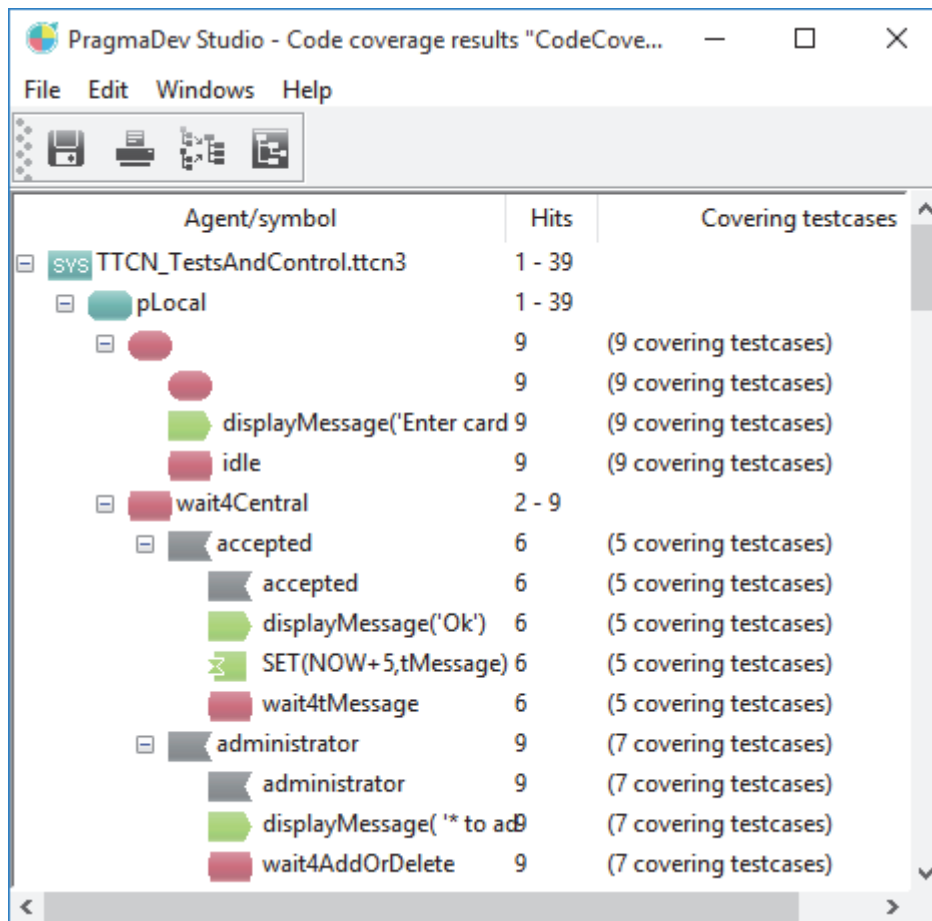


Then, the co-simulation of TTCN and SDL must be launched as explained in “TTCN-3 co-simulation” on page 329: a TTCN module containing a control part or testcases must be selected in the project manager, and the simulation run on it. The simulation can then be

done the usual way, as explained in “Simulation” on page 333. At the end of the simulation session, the code coverage can be extracted by using the  button:



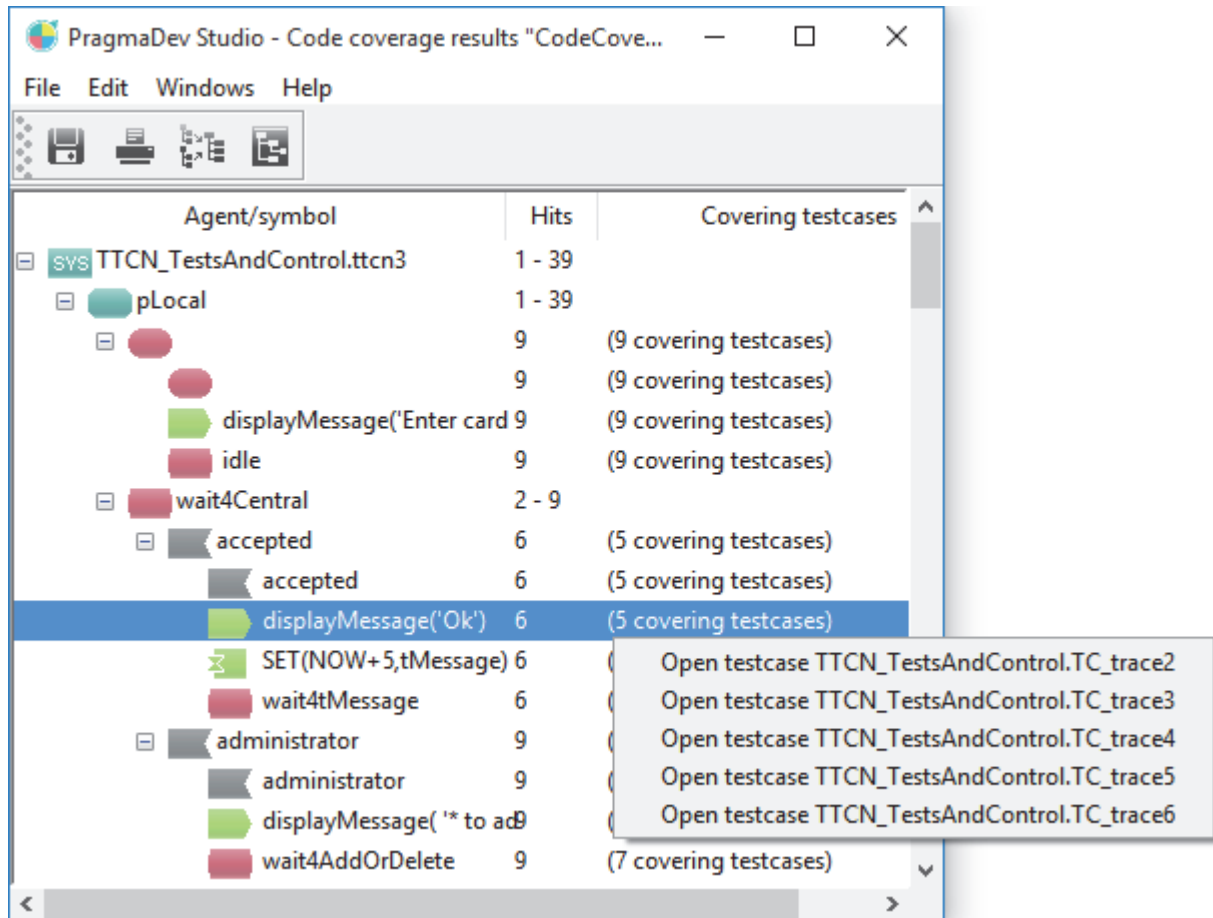
The extracted code coverage results will be opened automatically. In addition to the information of coverage on each symbol, they will also include the testcase coverage information:



Agent/symbol	Hits	Covering testcases
sys TTCN_TestsAndControl.ttcn3	1 - 39	
pLocal	1 - 39	
[red circle]	9	(9 covering testcases)
[red circle]	9	(9 covering testcases)
[green arrow] displayMessage('Enter card 9		(9 covering testcases)
[red circle] idle	9	(9 covering testcases)
wait4Central	2 - 9	
accepted	6	(5 covering testcases)
accepted	6	(5 covering testcases)
[green arrow] displayMessage('Ok')	6	(5 covering testcases)
[green square] SET(NOW+5,tMessage)	6	(5 covering testcases)
[red circle] wait4tMessage	6	(5 covering testcases)
administrator	9	(7 covering testcases)
administrator	9	(7 covering testcases)
[green arrow] displayMessage('* to ad9		(7 covering testcases)
[red circle] wait4AddOrDelete	9	(7 covering testcases)

To each symbol in the simulated system is not only associated the minimum and maximum number of executions of the symbol, but also the number of testcases that covered

it. To actually list these testcases, it is possible to right click on the number of testcases in the “Covering testcases” column:



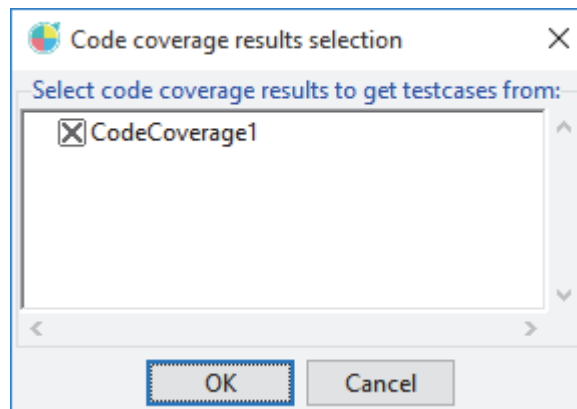
The contextual menu displays the names of the testcases that actually covered the symbols. Selecting one of the items will open its parent module in the text file editor, and position the insertion point directly on the declaration of the testcase.

6.3.2 Importing testcase coverage information in requirements

To be able to import testcase coverage information in a requirements table, the following prerequisites must be met:

- The requirements table must include symbols covering the requirements, entered as explained in “Covering symbols” on page 123.
- The results of a code coverage extraction must be available in the project, obtained during a TTCN + SDL co-simulation as explained in “Co-simulation code coverage analysis” on page 127.

To automatically include the testcase coverage information in the requirements table, open the table, then select “Get testcases from code coverage...” from the “Table” menu. A dialog appears:



In this window, select the set of code coverage results to extract the testcase coverage information from and press OK. The 4th column in the table will then be automatically filled with the names of the testcases covering the requirements:

ReqId	ReqDescription	Covered by	Tested by
ADMIN_1	Administrator specific card and code are required to add or delete a user.	sdIDecision@SYM8<u'userIDCode\n=\nMASTER_CARD_AND_CODE>	TTCN_TestsAndControl.TC_trace4 TTCN_TestsAndControl.TC_trace7 TTCN_TestsAndControl.TC_trace8 TTCN_TestsAndControl.TC_trace6 TTCN_TestsAndControl.TC_trace1 TTCN_TestsAndControl.TC_trace3 TTCN_TestsAndControl.TC_trace2 TTCN_TestsAndControl.TC_trace5
ADMIN_2	A global time out shall prevent the system to stay in an unstable state.	sdInitTimer@SYM9<u'SET(NOW+5,codeTimer)'>	TTCN_TestsAndControl.TC_trace4 TTCN_TestsAndControl.TC_trace9 TTCN_TestsAndControl.TC_trace7 TTCN_TestsAndControl.TC_trace8 TTCN_TestsAndControl.TC_trace6 TTCN_TestsAndControl.TC_trace1 TTCN_TestsAndControl.TC_trace3 TTCN_TestsAndControl.TC_trace2 TTCN_TestsAndControl.TC_trace5
ADMIN_3	The code shall not be clearly displayed on the LCD.	sdTask@SYM107<u'counter := counter+1,\nmessageToDisplay := messageToDisplay // **;'>	TTCN_TestsAndControl.TC_trace4 TTCN_TestsAndControl.TC_trace7 TTCN_TestsAndControl.TC_trace8 TTCN_TestsAndControl.TC_trace6 TTCN_TestsAndControl.TC_trace1 TTCN_TestsAndControl.TC_trace3 TTCN_TestsAndControl.TC_trace2 TTCN_TestsAndControl.TC_trace5

The testcases are all those covering any of the symbols present in the 3rd column of the table. Each testcase appears on a line in the cell, and double-clicking on it opens it in the text file editor.

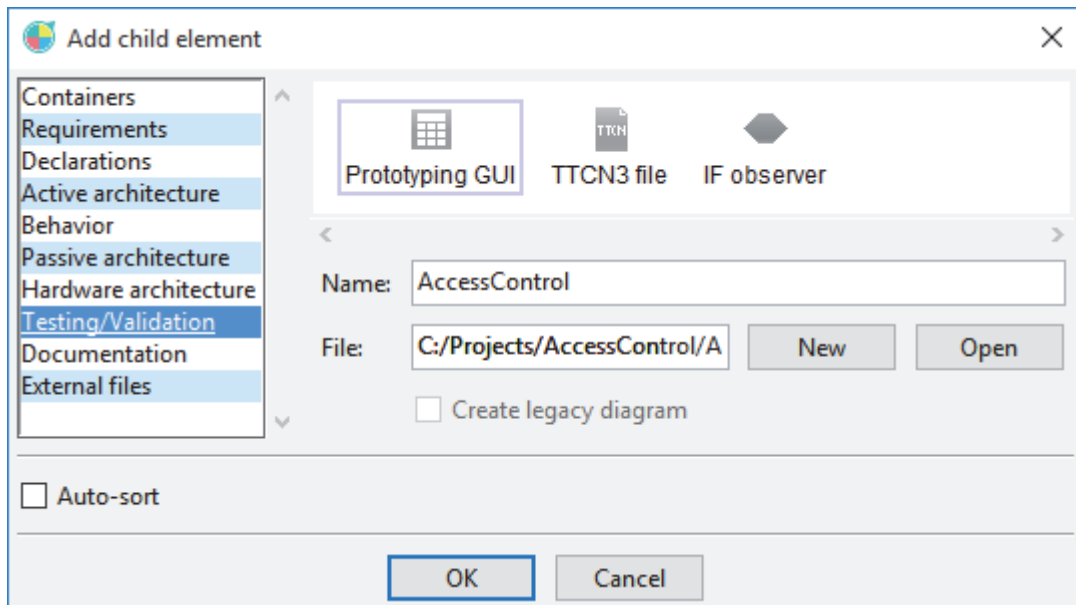
Note that these testcases are not modifiable, except by clearing all of them (item “Clear covering testcases...” in the “Table” menu), or by importing coverage information from another results set, which will replace the current covering information.

7 - Prototyping GUI

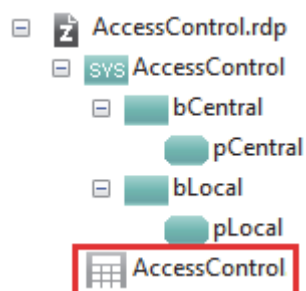
PragmaDev Studio has a built-in GUI editor to ease model validation. This allows to describe a graphical interface that will interact with the model.

7.1 - Prototyping GUI editor

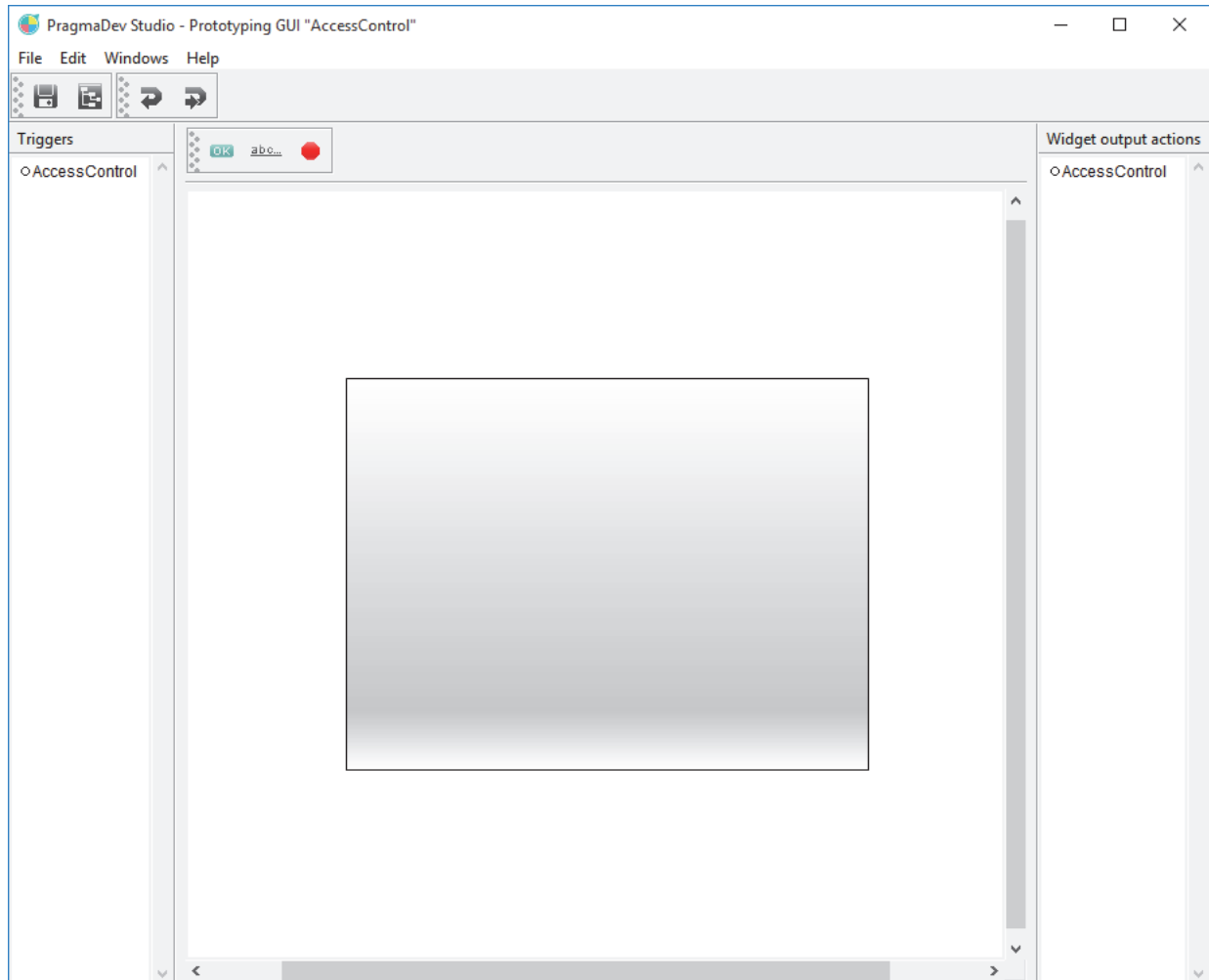
Insert a prototyping GUI element in the project:



This will create a prototyping GUI file and add the corresponding component in the project tree:



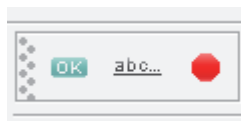
Double-clicking on this component will open a new empty GUI editor:



The editor is divided in three parts:

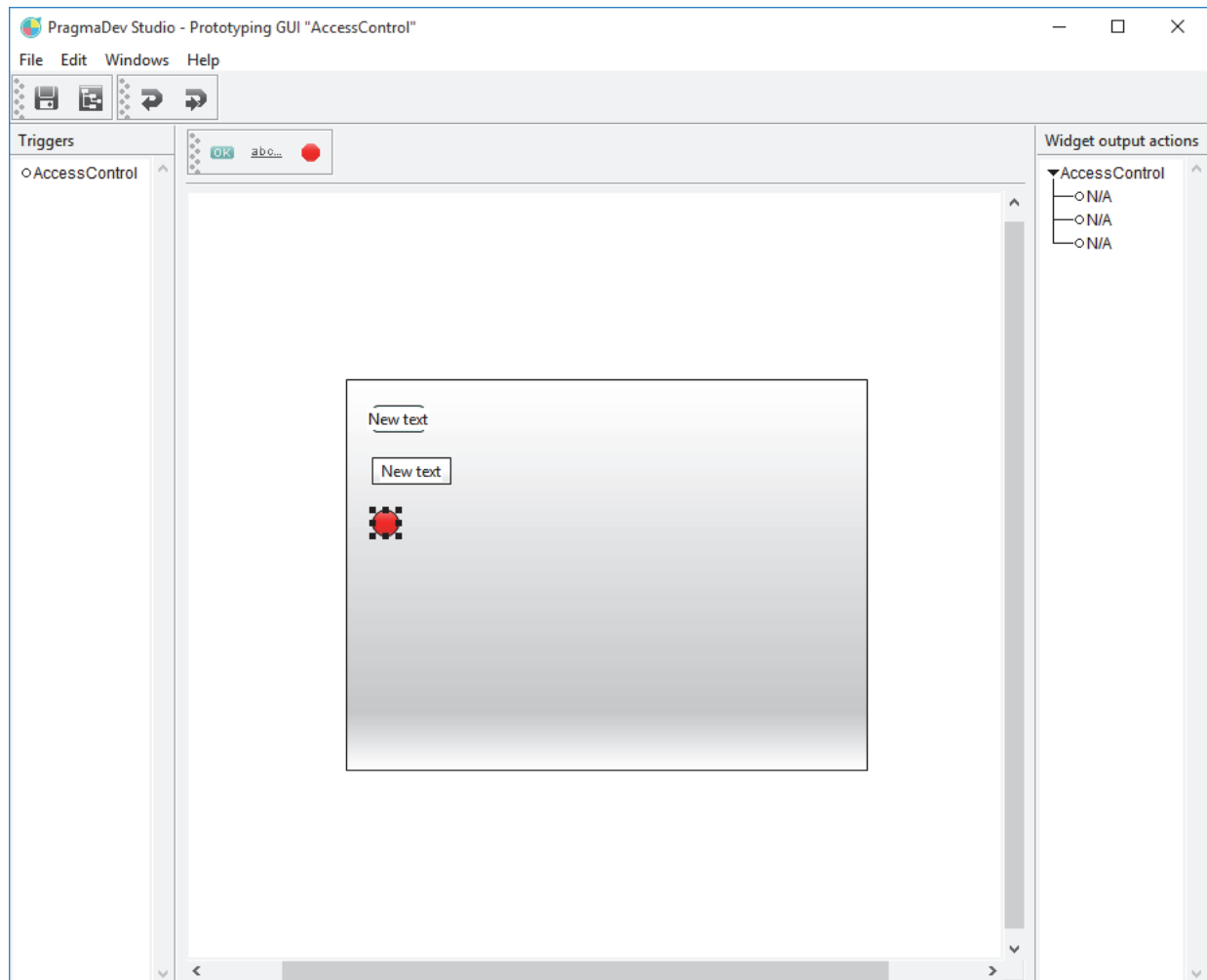
- The trigger tree on the left to describe which events will trigger actions on the display.
- The graphical layout in the middle to design what the GUI will look like.
- The output tree on the right to describe the actions to be executed when the GUI is stimulated.

In the middle area three types of graphical elements can be inserted: buttons, text displays, and LEDs:

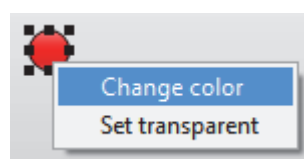


Each time one of these elements is inserted, a corresponding tree item is added in the right area.

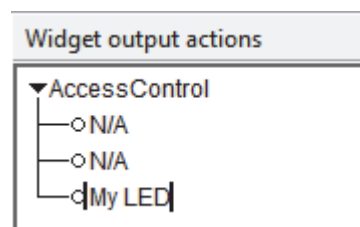
Let's consider for example we are adding one of each widget in the display:



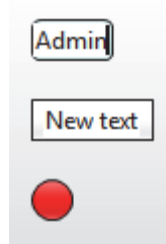
Right click on the widget to change its color:



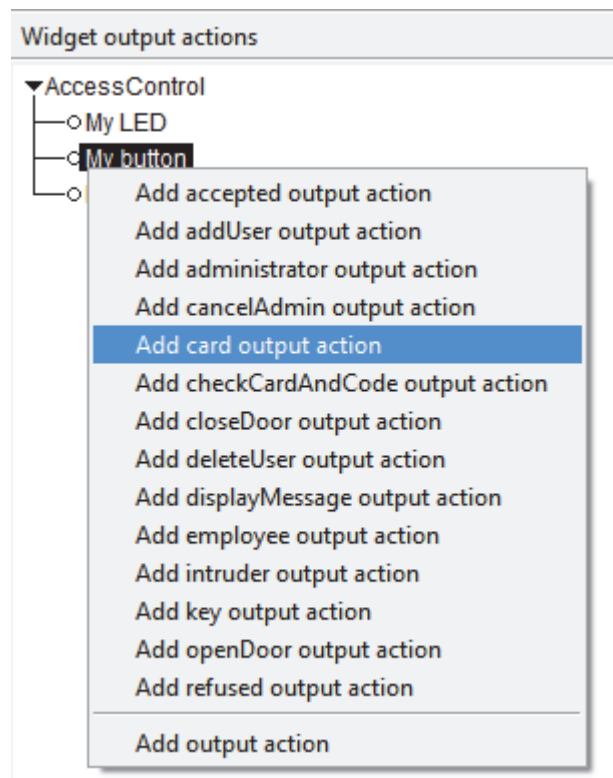
Double-click on the name in the tree to edit the name of the widget:



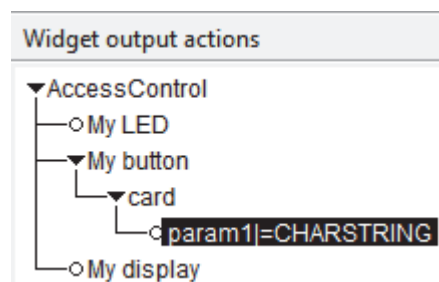
Click on the widget text to edit it:



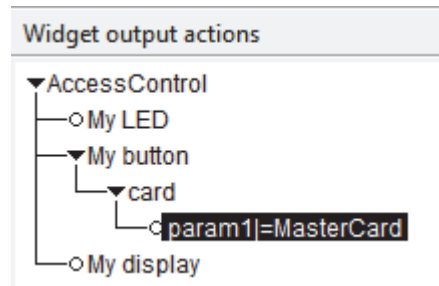
A text display and a LED generally do not generate any output action. Let's add an action to the Admin button: right click on the corresponding element in the output tree and all the available message in the system will be displayed:



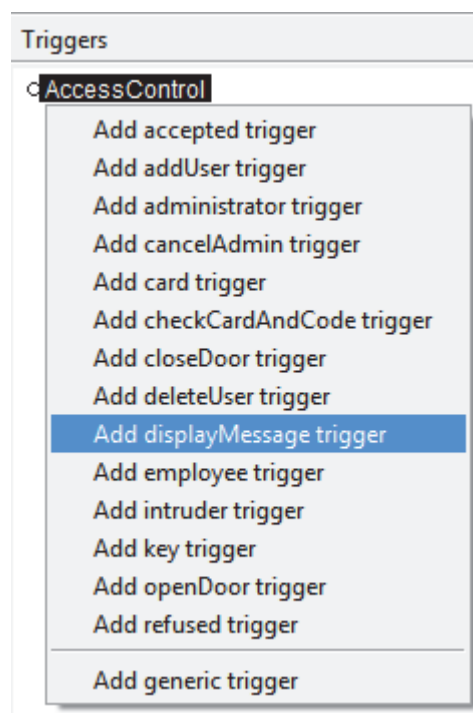
In this example, we are using the AccessControl SDL Z.100 example. Let's say the 'card' message will be sent to the system when the user clicks on 'My button'. The parameters associated with the message will be automatically displayed and the expected type is displayed to ease editing:



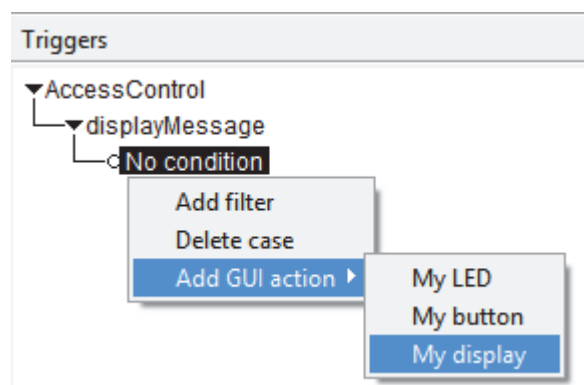
In that example the administrator card should have 'MasterCard' as its parameter. Double click on the type and enter the desired parameter value:



Now let's consider the incoming display message parameter value is to be displayed in the text display widget. Let's get to the trigger tree and create a case:



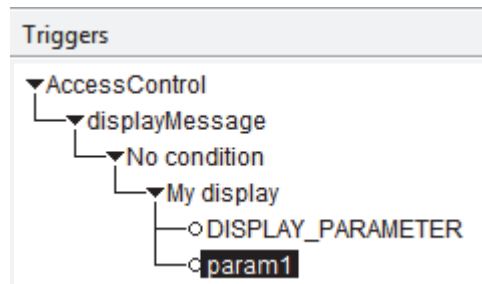
In the new element, it is now possible to add cases. Each case is a set of filters to be verified and actions to do when the filters are verified. In our case, we will not have any filter since we want to display the parameter value all the time.



The syntax to access parameter value has the following form:

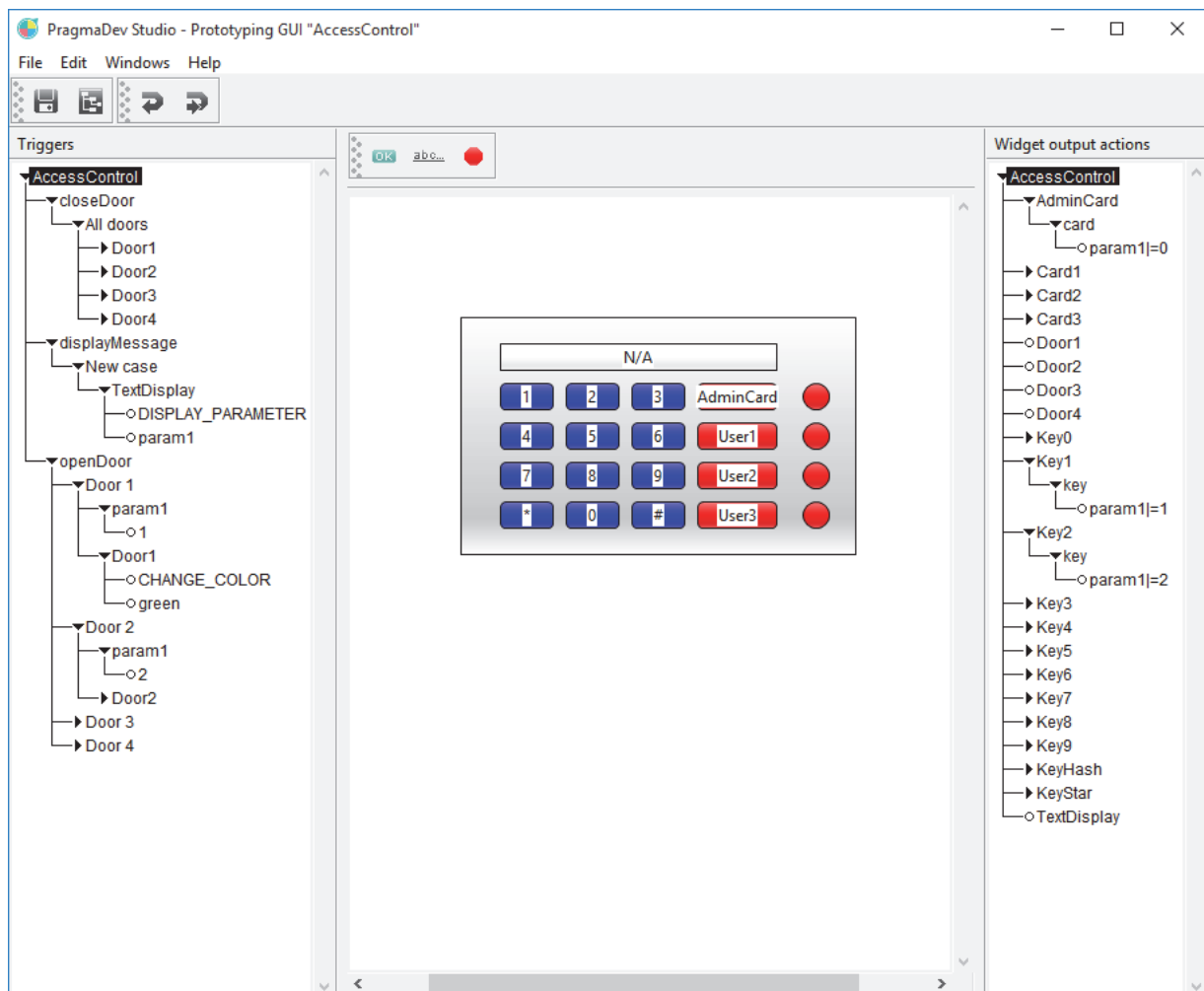
param<number>{.<field name>}]>*

In our example we just need to access the value of parameter 1:



We could also use the DISPLAY type of action to display a specific text that has nothing to do with the parameters value.

Let's have a look at some other cases with the AccessControl example:



In this example the three triggers: open, close, and displayMessage are considered.

- openDoor
Each door is a separate case with one filter. For the first door the filter Door 1 will check if the first parameter is equal to 1. If the filter is verified, the color of

the LED will switch to green. It is also possible to use the RVB form: #00FF00. In the Door 2 case, the filter verifies the value is 2 and sets the second LED to green. And so on...

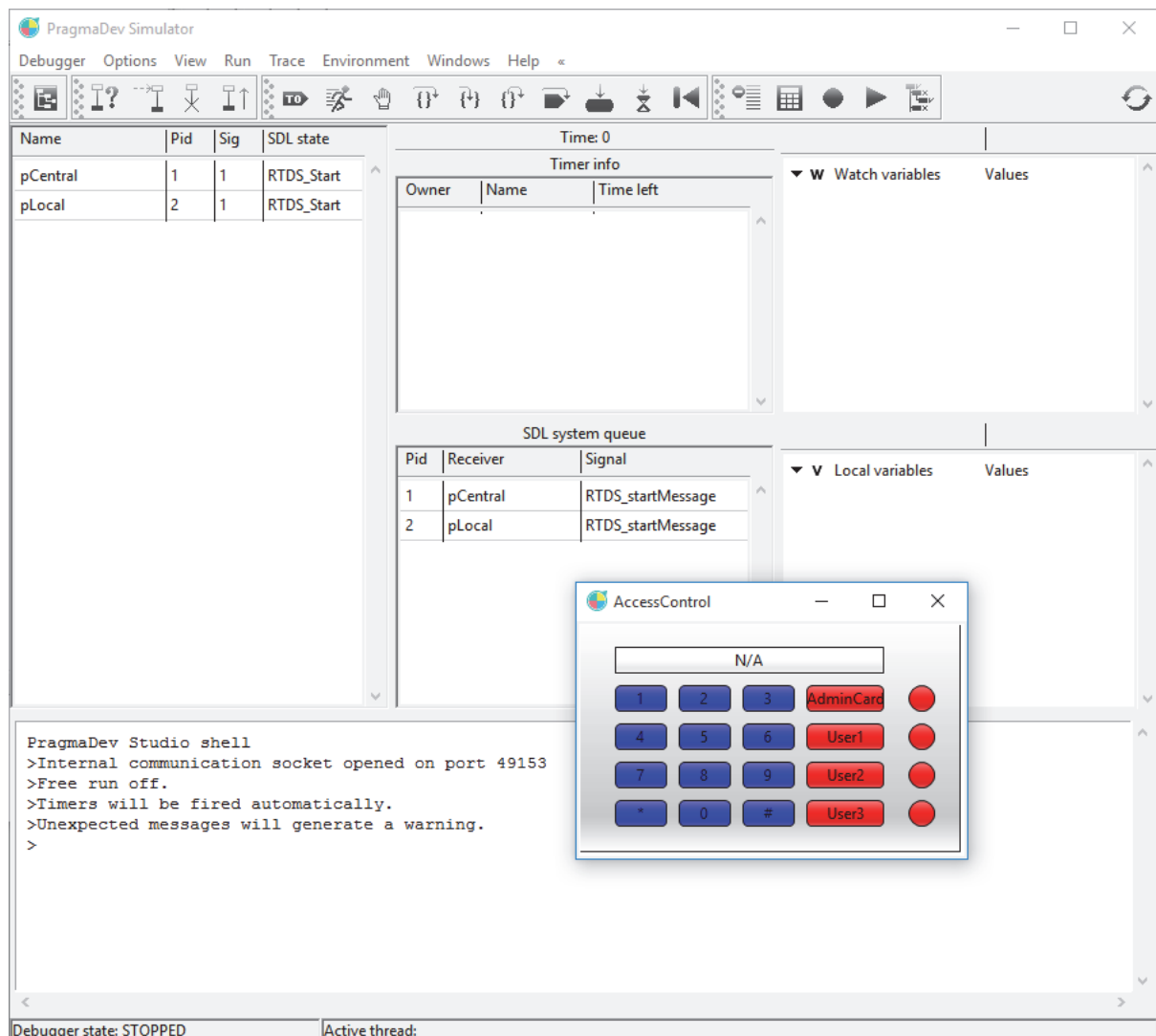
- **closeDoor**
There is only one case with no filter. That means when the close message is received all the actions are executed. In our case, all the LEDs are set back to red.
- **displayMessage**
There is no filter so the first parameter of the display message is printed in the display.

Each button generates an output. Let's consider three examples:

- **Key1**
When this button is pressed, the key message is sent to the system with its first parameter set to 1
- **Key2**
When this button is pressed, the key message is sent to the system with its first parameter set to 2
- **AdminCard**
When this button is pressed, the card message is sent to the system with its first parameters set to MasterCard.

7.2 - Prototyping GUI runner

To start the prototyping GUI, click on the  button in the Simulator or the Debugger:



The GUI will connect automatically to the running system. That's it.

8 - SDL-RT project

8.1 - Data and SDL types declarations

8.1.1 C types declarations

C types declaration can be made in several ways:

- in an external C header file that appears in the *Project manager*. The corresponding include will have to be done at block or process level in a text block,
- in an SDL text block; C code can be typed in directly. The C code covers different aspects depending on the SDL level it is found:
 - Block level
The C code contained at block level will be generated as a C header that will be included in the underlying SDL architecture (blocks, process, and procedures). It can contain C types and C global variables declaration but not the global variables themselves. The global variables must be declared in separate C files included in the *Project manager*.
 - Process level
The C code contained in text blocks in an SDL process will be inserted at the process function declaration level. It therefore contains local variables to the process.
 - Procedure level
The C code contained in text blocks in an SDL procedure will be inserted at the procedure function declaration level. It therefore contains local variables to the procedure.

8.1.2 SDL messages and message lists declaration

SDL messages and message lists are declared either in a dashed text box in a diagram, or in a SDL-RT declarations file (.rdm) in packages. The declaration statements are:

```
MESSAGE <message name> [ ( <parameter type> {, <parameter type> }* ) ] ;  
MESSAGE_LIST <list name> = <message name> { , <message name> }* ;
```

The parameter types in a MESSAGE declaration must be valid C types. Message lists may be nested by using (<list name>) in the list elements declaration.

Example:

```
MESSAGE msg1, msg2(int), msg3(char*, struct MyType*, double);  
MESSAGE_LIST myList = msg1, msg2, msg3;  
MESSAGE_LIST mySuperList = (subList1), (subList2), addMsg;
```

8.1.3 SDL timer declaration

SDL timers do not need any declaration. The code generator will browse the whole system and extract the used timers automatically.

8.1.4 Semaphore declaration

Semaphores need to be declared with the semaphore declaration symbol. The syntax in this symbol is:

```
<semaphore type> <semaphore name> (<option 1> [,<option 2>] [,<option 3>])
```

<semaphore type> can be:

- **BINARY**
 - <option 1> is:
 - **PRIO**
 - **FIFO**
 - <option 2> is:
 - **INITIAL_EMPTY**
 - **INITIAL_FULL**
- **MUTEX**
 - <option 1> is:
 - **PRIO**
Queue pending tasks on the basis of their priority
 - **FIFO**
Queue pending tasks on a first-in-first-out basis
 - <option 2> is:
 - **DELETE_SAFE**
Protect a task that owns the semaphore from unexpected deletion
 - <option 3> is:
 - **INVERSION_SAFE**
Protect the system from priority inversion
- **COUNTING**
 - <option 1> is:
 - **PRIO**
 - **FIFO**
 - <option 2> is:
 - <value for initial count (int)>

Example:

```
MUTEX mySemaphore(FIFO, DELETE_SAFE)
```

This example creates a mutual exclusion semaphore called `mySemaphore` with tasks pending on a first-in-first-out basis with protection from unexpected deletion of the owning task. <option 3> is omitted so the system is not protected against priority inversion.

8.1.5 Process declaration

A process is declared graphically in an SDL block diagram. The code generator will then generate the C function corresponding to the behavior description made in SDL.

The syntax is the following:

```
<process name> [( <initial number of instances, maximum number of instances> )] [:<process type>] [PRIO <priority>]
```

to create <initial number of instances> instance of <process type> named <process name> with priority <priority> at startup.

The default priority is defined by RTDS_DEFAULT_PROCESS_PRIORITY in RTDS_OS_basic.h. The default initial number of instance is 1. The maximum number of instances is just for documentation, no verification will be made at run time.

Examples:

```
myProcess
anotherProcess:aTypeOfProcess PRIO 80
aThirdProcess(0,10)
```

The last example is usually when the process is created by another process. They usually do not exist at startup.

8.1.6 Procedure declaration

A procedure is declared graphically with the procedure declaration symbol. The syntax is the syntax used to define a C function:

```
<return type> <function name> ({<parameter type> <parameter name>}*);
```

Example:

```
int myFunction (short myParameter);
```

8.1.7 Class description

A class is described graphically with a class symbol in a class diagram. The syntax used is the UML syntax:

- The class header identifying the class itself is formatted like follows:
[<< <stereotype> >>] [<package name>::]<class name> [{<properties>}]
The two recognized stereotypes are <<interface>> and <<system>> (for active classes; see below). The properties may be specified, but are ignored.
- The attributes are described via a set of lines having the following format:
[<visibility>] <name> [: <type>] [= <default value>] [{<properties>}]
The visibility may be '+' for public, '#' for protected and '-' for private. All basic C/C++ types are recognized. The default value and properties are ignored.
- The operations are described via a set of lines having the following format:
[<visibility>] <name>({<param>}*) [: <return type>] [{<properties>}]
where <param> has the format:
[<direction>] <name> [: <type>] [= <default value>]
The visibility is coded the same way than for attributes. Recognized types for parameters or return type are all C/C++ basic types, plus all classes known in the project. The direction for parameters may be "in" for input only parameters, "out" for output-only parameters, or "inout" for two-way parameters. The operation properties are ignored.

Note: parameters declared as "out" and "inout" are both passed as references to the method (<type>&).

Class constructor(s) and destructor are identified via the special names <<create>> and <<delete>> respectively. As in C++, there may be several constructors with different parameters, but only one destructor. There must be no return type for any constructor or destructor.

Systems, blocks, processes, block classes and process classes may also be referenced in class diagrams. In this case, they are represented as active classes, as explained in SDL-RT specification. These active classes may not have attributes, which are meaningless in this case. They may however have operations, representing incoming and outgoing signals for the object. These operations are indicated by special visibilities '>' for incoming signals and '<' for outgoing ones. These pseudo-operations may only accept one in parameter which is the data associated to the signal. For block and process classes, the properties are used to indicate via which gate goes the signal ({via:<gate name>}).

Example:



8.2 - SDL-RT symbols syntax

8.2.1 Task block

The task block contains standard ANSI C code as it would be written in a text file.

Example:

```

/* Say hi to you friend */
printf("Hello world!\n");

for (i = 0; i < MAX; ++i)
{

```

8.2.2 Next state

The syntax in the next state SDL graphical symbol is:

<new SDL state>

Of course, the new SDL state needs to be defined in the diagram.

Or it can also be:

" _ "

meaning the state is not changed.

8.2.3 Continuous signals

The continuous signal can contain any standard C expression that returns a C true/false expression. In the generated code the expression is put in an `if` statement as is. Since an SDL state can contain several continuous signal a priority level needs to be defined with the `PRI0` keyword. Lower values correspond to higher priorities. The syntax is:

```
<C condition expression>
PRI0 <priority level>
```

Example:

```
( a > 5 )
PRI0 3
```

8.2.4 Message input

```
<message name>
[(<parameter>
{, <parameter>}*)]
```

The message input symbol represent the type of message that is expected in an SDL-RT state. It always follows an SDL-RT state symbol and if received the symbols following the input are executed.

An input has a name and comes with optional parameters. To receive the parameters it is necessary to declare a variable for each expected parameter. The syntax in the message input symbol is the following:

```
<message name> [(<parameter name> {, <parameter name>}*)]
<parameter name> are variables that need to be declared.
```

Example:

```
MESSAGE
ConReq(t_struct, int, long),
ConConf;
```

```
t_struct myStruct;
int      myInt;
long     myLength;
```

```
ConReq(myStruct, myInt, myLength)
```

```
ConConf
```

`myStruct`, `myInt`, and `myLength` will be assigned to the value of the received message parameters.

```
<message name>
[(<data length>,
<pointer on data>)]
```

Even though it is not recommended, if a message is declared without any parameter it is possible to transmit undefined parameters with a length and a pointer on the parameter data. In that case it is necessary to declare 2 variables that will be the parameter length and the pointer on the parameters.

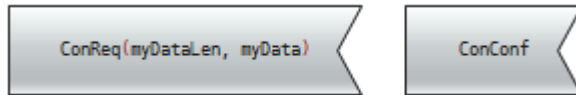
The syntax in the message input symbol is the following:

`<message name> [(<length of data>, <pointer on data>)]`

`<data length>` is a variable that needs to be declared.

`<pointer on data>` needs to be declared.

Examples:



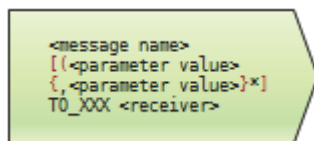
8.2.5 Message output

8.2.5.1 General aspects



The syntax in the message output symbol can be written in 3 ways depending on whether the queue Id of the receiver is known or not, and if its name is constant or variable. A message can be sent to a queue Id, a process name or via a channel or gate. When communicating with the environment, a special syntax is provided.

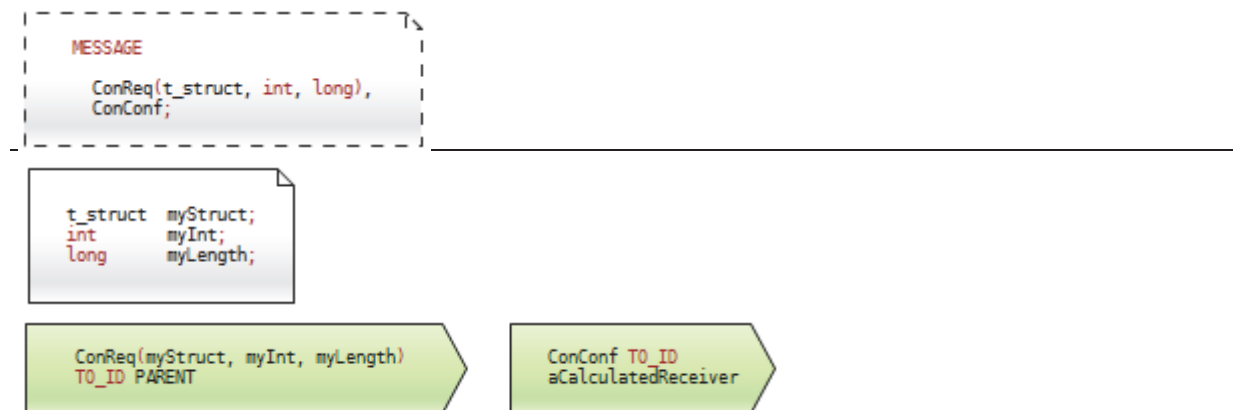
Messages can have parameters. The type of the parameters are defined in the message declaration.



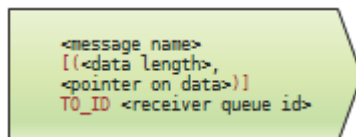
`<message name>[(<parameter value> {,<parameter value>}*) TO_XXX
<receiver>`

`<parameter value>` is the value of the parameter with the type declared in the message declaration. The parameters can be transmitted as values or references. When the parameters are direct values, the data is first copied and then sent out. When the parameters are references, the receiver and the sender end up with the same reference on the data. Is it then very important to define which process owns the data in order to avoid data corruption. It is usual to consider the sender does not own the data any more once it has been sent, and it is the receiver's responsibility to free the associated memory if needed.

Examples:



Even though it is not recommended, it is also possible to use a generic parameter assignment when there is no parameter type declaration:

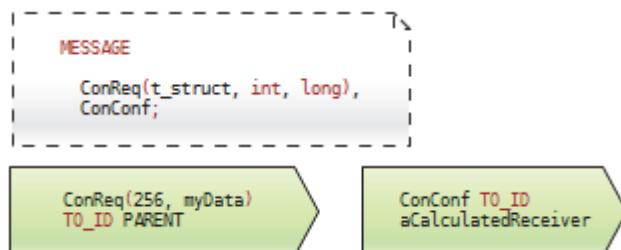


`<message name> [(<length of data>, <pointer on data>)] TO_ID <receiver queue id>`

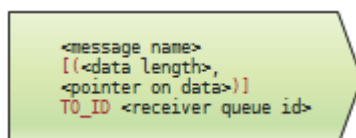
`<receiver queue id>` is of type `RTDS_QueueId`

The generated code will copy the data of size `<length of data>` pointed by `<pointer on data>`.

Examples:



8.2.5.2 Queue Id



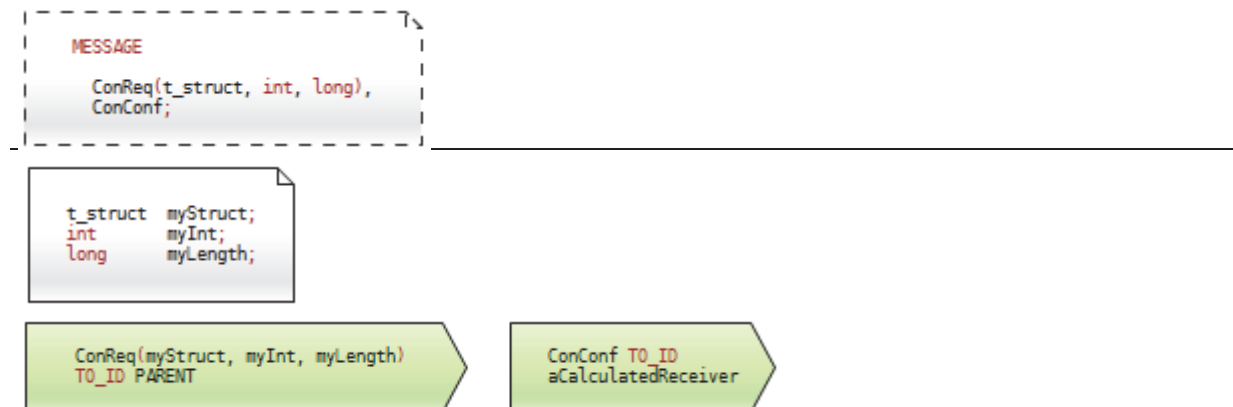
`<message name> [(<parameter value> {,<parameter value>}*) TO_ID <receiver queue id>`

- `<parameter value>` is the value of the parameter with the type declared in the message declaration,

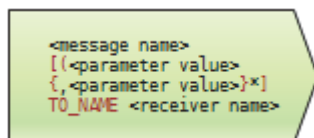
- <receiver queue id> is of type RTDS_QueueId.
It can take the value given by the SDL keywords:

PARENT	The queue id of the parent process.
SELF	The queue id of the current process.
OFFSPRING	The queue id of the last created process if any or NULL if none.
SENDER	The queue id of the sender of the last received message.

Examples:



8.2.5.3 Process name



<message name> [(<parameter value> {,<parameter value>}*) TO_NAME
<receiver name>

<receiver name> is the name of a process if unique or it can be ENV when simulating and the message is sent out of the SDL system.

Examples:



Note: If several instances have the same process name (several instances of the same process for example), the TO_NAME will send the message to the first created process with the corresponding name. Therefore this method should no be used when the process name is not unique within the system.

8.2.5.4 Via a channel or gate

```

<message name>
[(<parameter value>
{,<parameter value>}*)]
VIA <channel or gate name>
  
```

<message name> [(<parameter value> {,<parameter value>}*)] VIA <channel or gate name>

<channel or gate name> is the name channel or gate connected to the current process or process class.

Examples:

```

ConReq(256, myData)
VIA gateOut
  
```

```

ConReq(256, myData)
VIA extChannel
  
```

Note: The actual receiver is resolved statically. So there must be no ambiguity in the channels allowing the signal to pass. Also note that a VIA is resolved to a process name, so the note in paragraph 8.2.5.3 applies.

8.2.5.5 Environment

```

<message name>
[(<parameter value>
{,<parameter value>}*)]
TO_ENV <C macro name>
  
```

<message name> [(<parameter value> {,<parameter value>}*)] TO_ENV <C macro name>

<C macro name> is the name of the macro that will be called when this SDL output symbol is hit. If no macro is declared the message will be sent to the environment process; that obviously only works when simulating but not for the final code.

Examples:

```

ConReq(256, myData)
TO_ENV
  
```

```

ConReq(256, myData)
TO_ENV MESSAGE_TO_HDLC
  
```

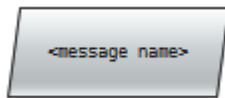
In this second example the generated code will be:

```

MESSAGE_TO_HDLC(Con-
Req,myDataLength,myData)
  
```

Note: When sending data pointed by <pointer on data>, the corresponding memory should be allocated by the sender and should be freed by the receiving process. This is because this memory area is not copied to the receiver; only the pointer value is transmitted. So after being sent the sender should not use it any more.

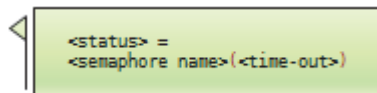
8.2.6 Saved message



The syntax to save an SDL message in the save graphical symbol is:

`<message name>`

8.2.7 Semaphore take



To take a semaphore, the syntax in the 'semaphore take SDL-RT graphical symbol' is:

`<status> = <semaphore name> (<timeout option>)`

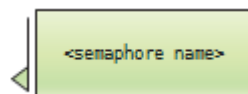
where `<timeout option>` is:

- `FOREVER`
Hangs on the semaphore forever if not available.
- `NO_WAIT`
Does not hang on the semaphore at all if not available.
- `<number of ticks to wait for>`
Hangs on the semaphore the specified number of ticks if not available.

and `<status>` is of type `RTDS_SemaphoreStatus` and can take the following values:

- `RTDS_OK`
If the semaphore has been successfully taken
- `RTDS_ERROR`
If the semaphore was not found or if the take attempt timed out.

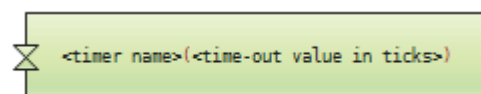
8.2.8 Semaphore give



To give a semaphore, the syntax in the 'semaphore give SDL-RT graphical symbol' is:

`<semaphore name>`

8.2.9 Timer start

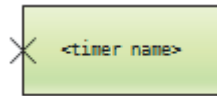


To start a timer the syntax in the 'start timer SDL-RT graphical symbol' is :

`<timer name> (<time value in tick counts>)`

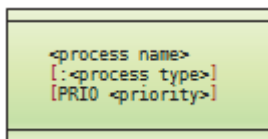
<time value in tick counts> is usually an 'int' but is RTOS and target dependant.

8.2.10 Timer stop



To cancel a timer the syntax in the 'cancel timer SDL-RT graphical symbol' is : <timer name>

8.2.11 Process



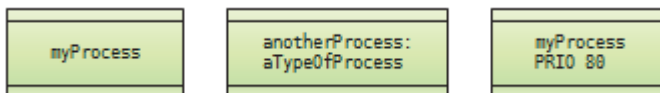
To create a process the syntax in the 'create process SDL-RT graphical symbol' is:

<process name>[:<process type>] [PRIO <priority>]

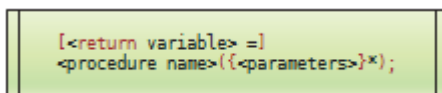
to create one instance of <process type> named <process name> with priority <priority>.

The default priority will be 150.

Examples:



8.2.11.1 Procedure call



The procedure call symbol is used to call an SDL-RT procedure (see "Procedure declaration" on page 142). Since it is possible to call any C function in an SDL-RT task block it is important to note SDL-RT procedures are different because they know the calling process context, e.g. SDL-RT keywords such as SENDER, OFFSPRING, PARENT are the ones of the calling process.

The syntax in the procedure call SDL graphical symbol is the standard C syntax:

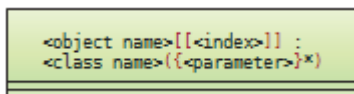
[<return variable> =] <procedure name>({<parameters>}*);

Examples:



Note: A procedure defined in SDL can not be called directly from a C statement. It has to be called from the procedure call graphical symbol. This is due to the fact that the procedure needs to know the process context so the generated code adds a parameter to the procedure definition and call.

8.2.12 Object initialization



The object initialization symbol is used when a class is attached to a process or process class via a composition link. This composition is made in a class diagram like follows:



In this case, one or several instances of `<class>` are part of the process `<process>`, depending on the composition's cardinality. These instances are known in the process via a variable named `<role name>`.

The object initialization symbol may be used like follows:

- If the maximum number of instances is 1, the object initialization symbol *must* be used to initialize the object in the start transition for the process. No `[<index>]` must follow the object name in the symbol.
- If the maximum number of instances is more than one, it is possible but not mandatory to use the object initialization symbol to initialize one of the objects in the list of associated instances. In this case, an `[<index>]` must be specified.

Examples:

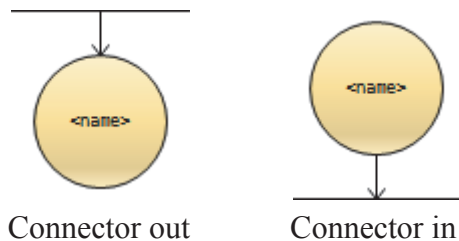


Note: In the generated code, using a task block containing:

```
<object> = <class>(<parameters...>);
```

is exactly the same than using the object initialization symbol, except for the added semantics checking in the case of compositions with a maximum cardinality of 1.

8.2.13 Connectors



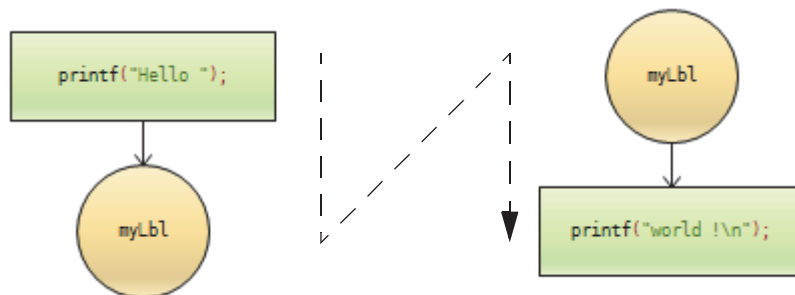
Connectors are used to:

- split a transition into several pieces so that the diagram stays legible,
- to gather different branches to a same point.

A connector-out symbol has a name that relates to a connector-in. The flow of execution goes from the connector out to the connector in symbol.

A connector contains a name that has to be unique in the process. The syntax is:
<connector name>

Examples:



8.2.14 Decision

The expression to evaluate in the symbol can contain:

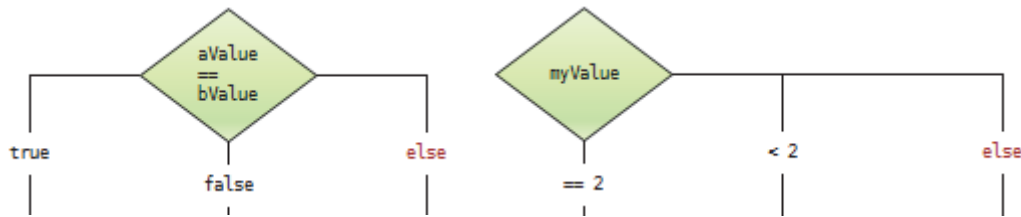
- any standard C expression that returns a C true/false expression,
- an expression that will be evaluated against the values in the decision branches.

The values of the branches have keyword expressions such as:

- >, <, >=, <=, !=, ==
- true, false, else

The else branch contains the default branch if no other branch made it.

Examples:



8.2.15 SDL keywords

8.2.15.1 Global keywords

The following SDL keywords are defined and can be used in all SDL symbols:

- PARENT The queue id of the parent process.
- SELF The queue id of the current process.
- OFFSPRING The queue id of the last created process if any or NULL if none.
- SENDER The queue id of the sender of the last received message.

8.2.15.2 Local keywords

The following keywords are dedicated to specific SDL symbols :

keywords	concerned symbols
PRI0	Task definition Task creation Continuous signal
TO_NAME TO_ID ENV	Message output
FOREVER NO_WAIT	semaphore manipulation
BINARY MUTEX COUNTING PRI0 FIFO INITIAL_EMPTY INITIAL_FULL DELETE_SAFE INVERSION_SAFE	semaphore declaration
>, <, >=, <=, !=, == true, false, else	decision branches


Table 1: Keywords in symbols

keywords	concerned symbols
USE SDL_MESSAGE_LIST	text symbol

Table 1: Keywords in symbols

8.3 - Code generation

The code generation for an agent is run from the project manager by selecting the agent, then choosing “If needed...” or “Force...” in the “Generation / Generate code” menu.

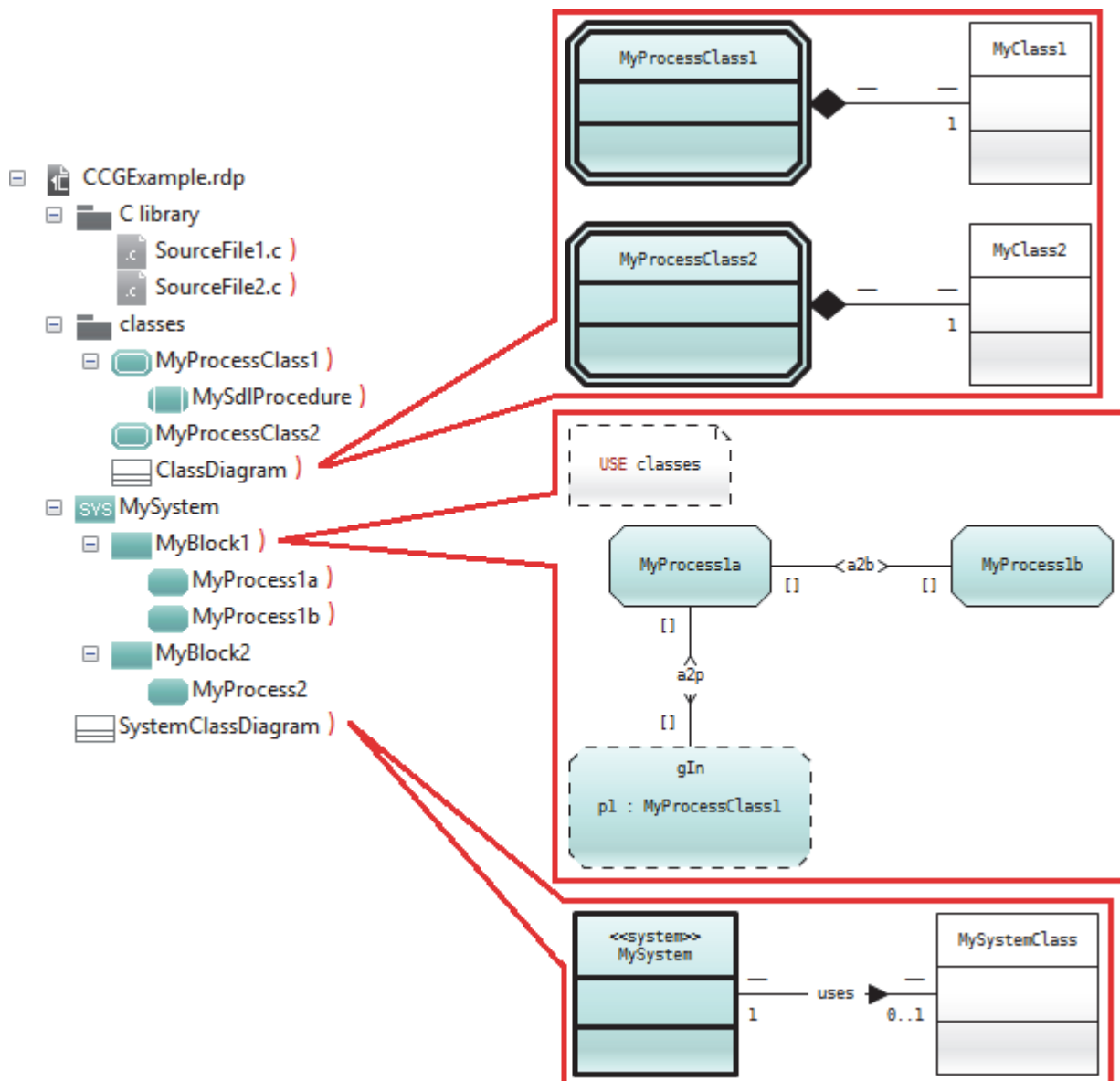
Clicking on the  button or selecting the “Generate code if needed...” or “Force code generation...” in the “Generation / Build” menu generates the code, but also runs the whole build process if enabled in the generation options (see “Profiles” on page 157).

8.3.1 Concerned elements

The elements concerned by the code generation are the following:

- All elements in the agent sub-tree, including SDL diagrams and C/C++ source files;
- All agent classes used in any diagram of the agent sub-tree, with the sub-tree for the agent class;
- All C/C++ source files that do not appear as children of SDL diagrams;
- All classes linked by any association to any involved agent or one of its parents, and all associated classes recursively.

For example, if the project tree is:



If we generate the code for the block **MyBlock1**, the included elements are in the diagrams marked in red:

- The block **MyBlock1**;
- The processes **MyProcess1a** and **MyProcess1b** since they are in the block's sub-tree;
- The process class **MyProcessClass1**, since the block contains an instance of the class;
- The procedure **MySdlProcedure** since it's in the sub-tree for the process class;
- The C source files **SourceFile1.c** and **SourceFile2.c** since they're not children of any SDL diagrams;
- The class **MyClass1**, since it is linked to **MyProcessClass1** in diagram **ClassDiagram**;

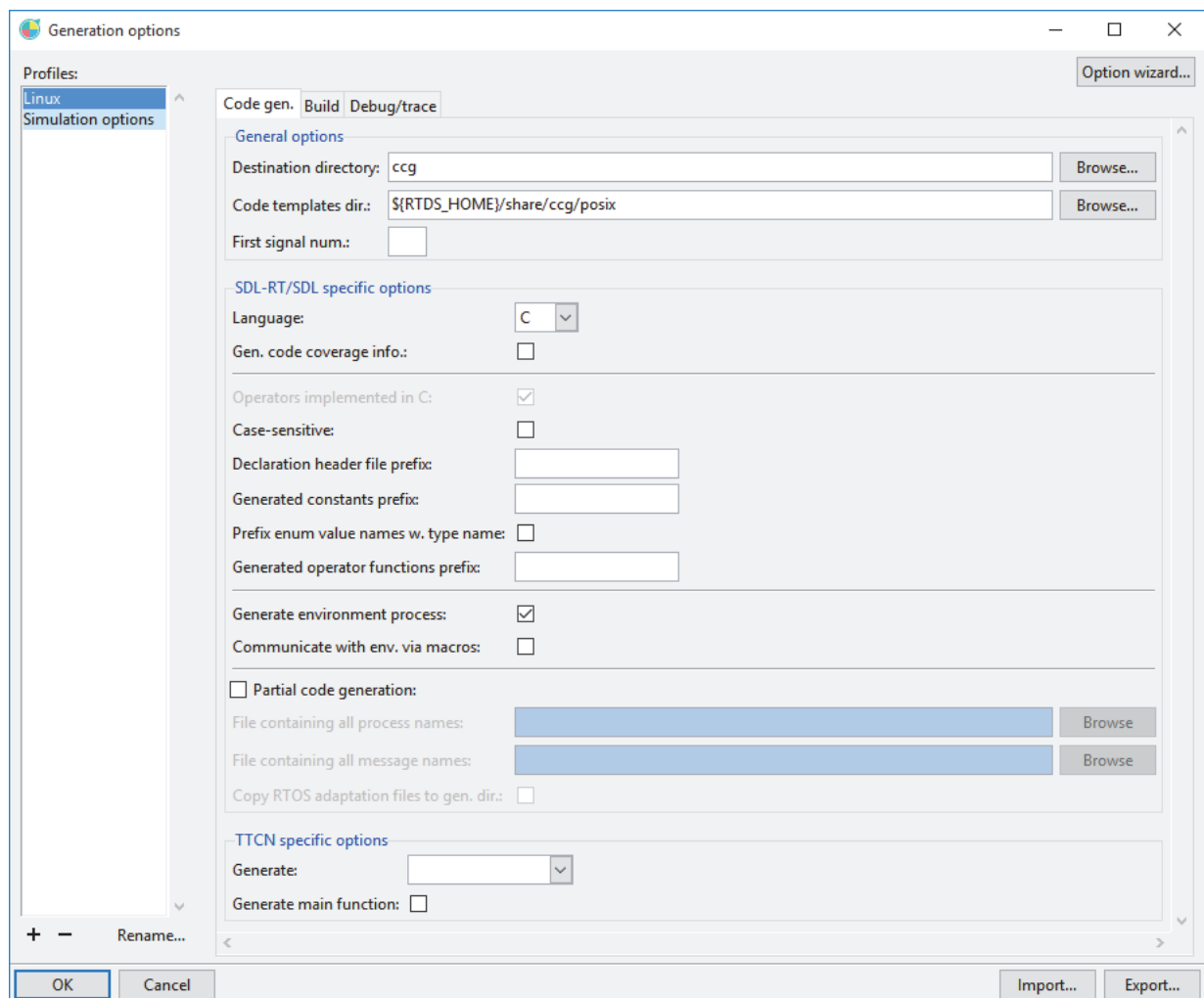
- The class MySystemClass, since it is linked to MySystem in diagram System-ClassDiagram and MySystem is a parent of MyBlock1.

The process class MyProcessClass2 is not included since no instance of this class appears in MyBlock1 or any of its descendants. The class MyClass2 is not included since it is only linked to the process class MyProcessClass2, which is not involved in the code generation.

8.3.2 Profiles

8.3.2.1 Description

The options concerning the code generation and compilation are managed via a set of profiles stored with the project. These profiles are displayed via the item “Options...” in the “Generation” menu of the project manager.



Code generation profiles

The left part of the dialog displays the names of the existing profiles. It also allows to add, delete or rename profiles via the “+”, “-” and “Rename...” buttons respectively. Adding a profile also allows to copy the selected one in the list. Selecting a profile name in the list displays the options set for this profile in the right part.

The “Option wizard...” button on the top of the dialog allows to quickly create a typical working profile depending on the platform, RTOS, and debugger. It is described in paragraph “Option wizard” on page 162.

The buttons “Import...” and “Export...” at the bottom right of the dialog allow to export a generation profile in a file, and then to import it back in another project.

The following paragraphs describe the options in the dialog tabs.

8.3.2.1.1 Code generation options

This tab is displayed above (paragraph “Description” on page 157). The options are:

- *General options* group:
 - *Destination directory* is the directory where all files will be generated.
 - *Code templates dir.* is the directory containing the files used to generate the code.
 - *First signal num.* is the lowest number to use for signal numerical values. This option is useful if your system includes processes defined outside PragmaDev Studio that use signal numerical values of their own. Setting this option to a value higher than any existing signal numerical value will ensure that PragmaDev Studio never generates an already used one.
- *SDL-RT / SDL specific options* group:
 - *Language* is the programming language used for the export. It can be C or C++.
 - If the *Gen. code coverage info.* check box is checked, the code to extract code coverage will be generated in addition to the normal code. See paragraph “Code coverage” on page 254.
 - The *Operators implemented in C* option is only meaningful if the generation language is C++ and the project language is SDL. It indicates that the functions implementing the SDL operators are C functions and that their declaration should be generated in an extern “C” block.
 - If the *Case-sensitive* option is checked, the code generation will be made in case-sensitive mode. This option is only meaningful if the project language is SDL. The default is to use for each identifier the case for the first time it is seen in the diagrams.
 - The *Declaration header file* prefix is added to the name of all generated declaration files.
 - The *Generated constants prefix* is added to all identifiers generated for SDL synonyms or literals. It has no effect in SDL-RT.
 - If the *Prefix enum values names w. type name* option is checked, the identifiers generated for SDL literals will be prefixed with the type name (in addition to the prefix above). This allows to have several types defining a literal with the same name. This option has no effect in SDL-RT.
 - The *Generated operator functions prefix* will be added to all declarations for the functions implementing the SDL operators. It has no effect in SDL-RT.
 - If the *Generate environment process* option is checked, a process simulating the environment will always be generated, even if none is present in the system.

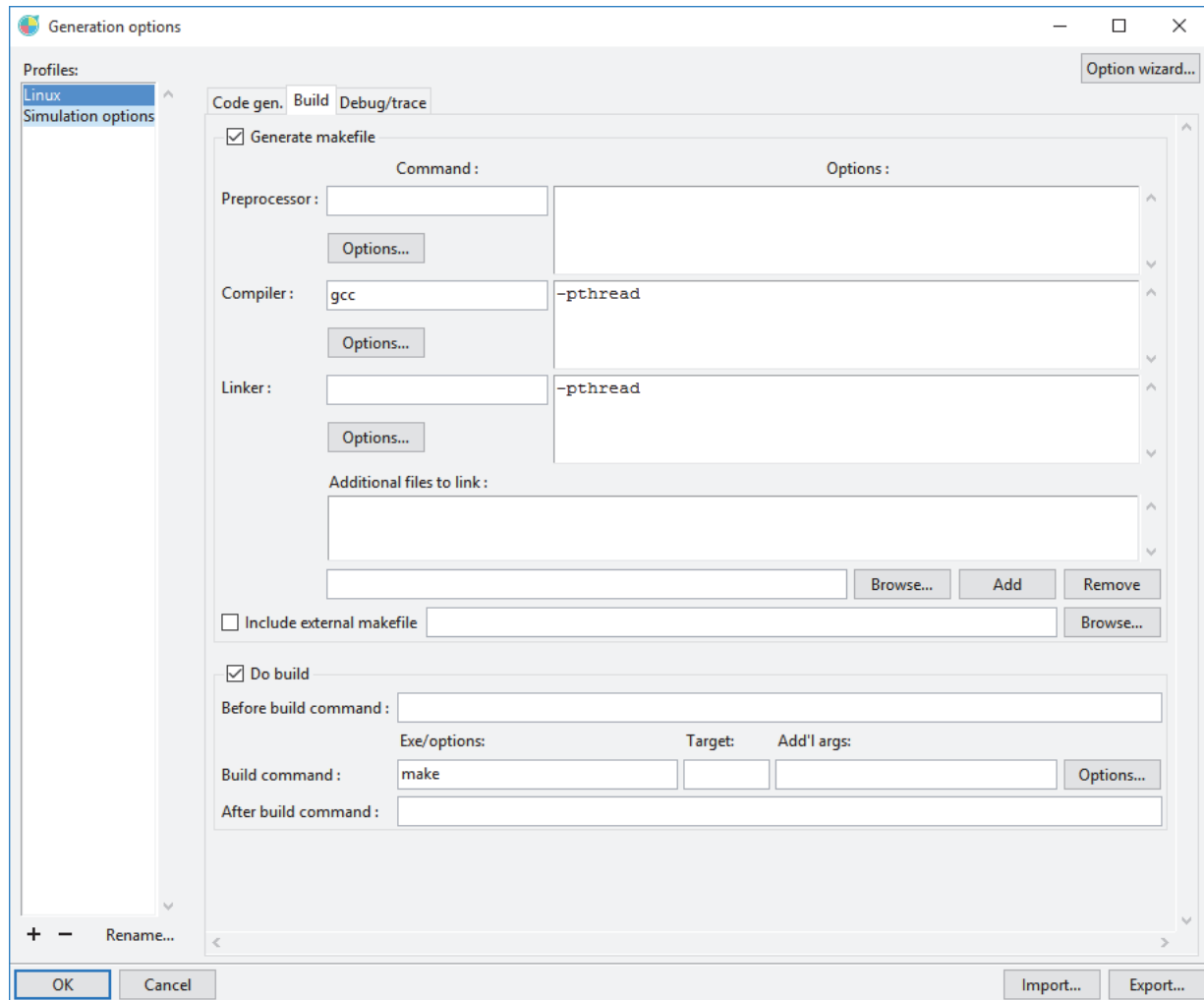
- If the *Communicate with env. via macros* option is checked, message output with the `T0_ENV` keyword followed by a macro name will always call the macro and not actually send a message. If the option is unchecked, an actual message sending is done, which requires an existing environment process. In this case, it is safer to check the previous option so that an environment process is always generated. If both options are unchecked and no explicit environment process exists in the system, code generation will fail.
- Checking the *Partial code generation* checkbox allows to generate code allowing easier integration with an external scheduler. In this case, two files must be provided:
 - One listing all the names of the PragmaDev Studio processes that will be included in the final system, one name per line;
 - One listing all the names of the messages that can be sent or received by all PragmaDev Studio processes in the final system, one name per line.These files are necessary to ensure the correct generation of files defining global information (e.g `RTDS_gen.h`).

If the *Copy RTOS adaptation files to gen. dir.* option is checked, all files in the profile directory used in the build will be copied to the generation directory, and the build process will use these files and not those in the PragmaDev Studio installation directory. This allows to have a completely standalone generated code, which doesn't need a PragmaDev Studio installation to be run. Partial code generation is described in detail in paragraph "Integration in external scheduler" on page 213.
- *TTCN specific options* group:

These options are specific to TTCN C++ code generation (see "C++ code generation" on page 335).

8.3.2.1.2 Build options

The tab for build options looks like follows:



The options are:

- If the *Generate makefile* check box is checked, a makefile will be generated. The following options set the commands and options that will be included in the makefile. The only mandatory command is the compiler. If the preprocessor is not set, there will be no explicit preprocessing phase; if the linker is not set, the compiler command will be used for linking.
- *Include external makefile*
 Some profiles such as Tornado, OSE, and CMX require a pre-defined external makefile. To include a user defined external makefile, the pre-defined include should be done in the new external makefile.
 For profiles using GNU make (Cygwin, Gnu and Tornado), the external makefile can access environment variables via the regular makefile macro syntax (e.g., `${RTDS_HOME}`). Other flavors of make may also offer this possibility.
- If the *Do build* check box is set, the actual compilation will also be run by the code generation operation. The command run for the compilation itself is split-
 ted in 3 parts:

- The executable name with the options, e.g "make", or "make -f MyMakefile";
- The target name, usually empty or "RTDS_ALL";
- The additional arguments that should be passed after the target, e.g., makefile macro definitions ("MACRO_NAME=value").

In addition to the compilation command, you may also enter two commands that will be run before and after the compilation respectively. In these command, the final executable name can be accessed with the following environment variables

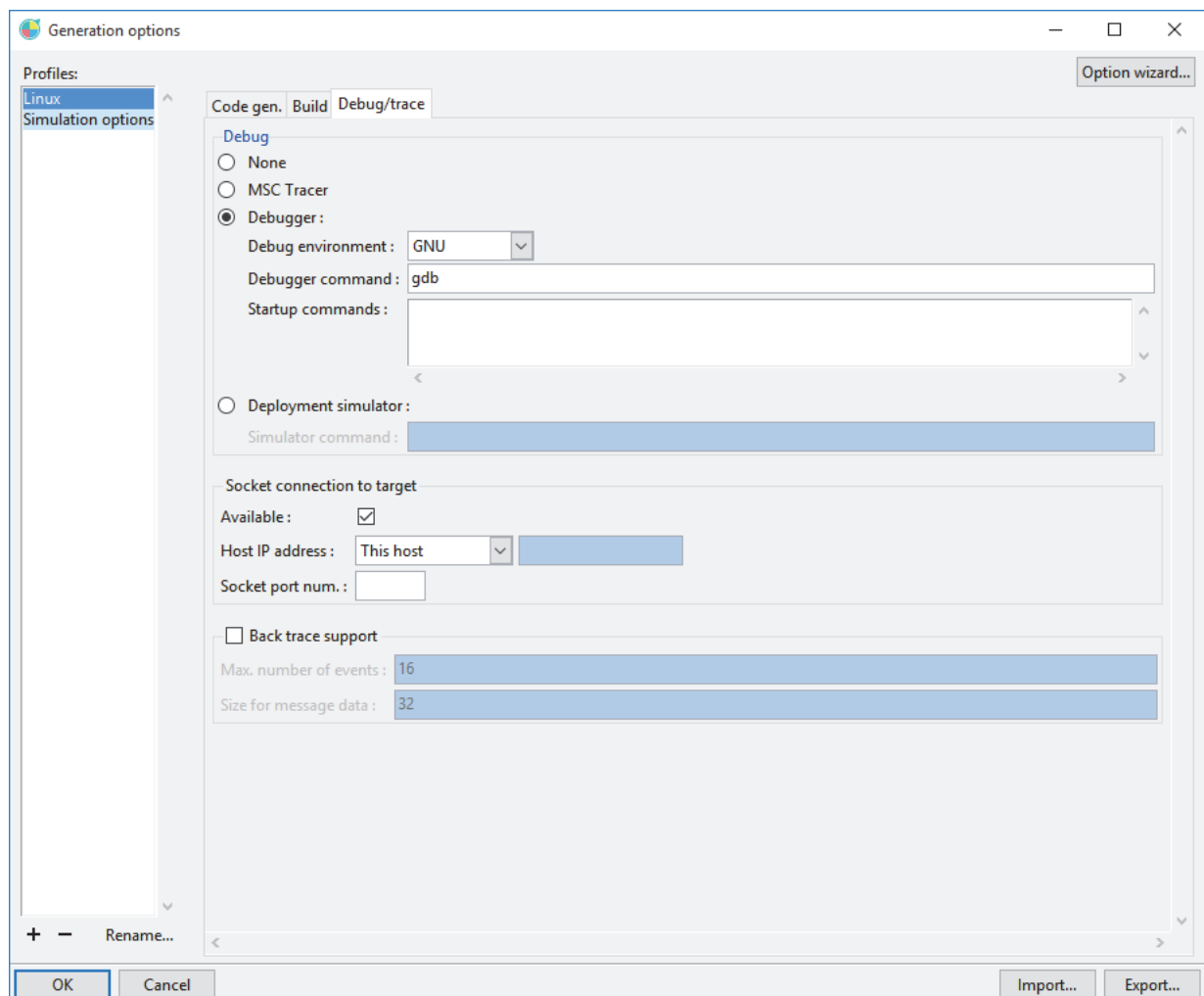
- RTDS_TARGET_NAME executable name with extension,
- RTDS_TARGET_BASE_NAME executable name without extension.

Please note the syntax to access environment variables depending on the host platform:

- DOS: %RTDS_TARGET_NAME%
- Unix: \$RTDS_TARGET_NAME

8.3.2.1.3 Debug and trace options

The tab for debug and trace options looks like follows:



The *Debug* group defines the debug environment to use.

- If set to *None*, the profile is considered to be for generating the final code, meaning it will:

- use the options in the common section of the `DefaultOptions.ini` file in the Code templates directory (Cf. Reference Manual),
- not start any debugger.
- If set to *MSC Tracer*, the profile is also for generating the final code, but the generated executable will be able to trace all its actions using PragmaDev MSC Tracer.
- If set to *Debugger*, the profile is a debug profile, meaning it will:
 - use the options in the common and debug section of the `DefaultOptions.ini` file in the Code templates directory (Cf. Reference Manual),
 - start the C debugger and the corresponding SDL-RT debugger interface. The command for the debugger is set in the corresponding field. The field *Startup commands* contains commands to be sent to the debugger once it is started.
- If set to *Deployment simulator*, the profile is a deployment simulation profile, meaning it will:
 - use the options in the common section of the `DefaultOptions.ini` file in the Code templates directory (Cf. Reference Manual),
 - start the deployment simulator using the command in the corresponding field.

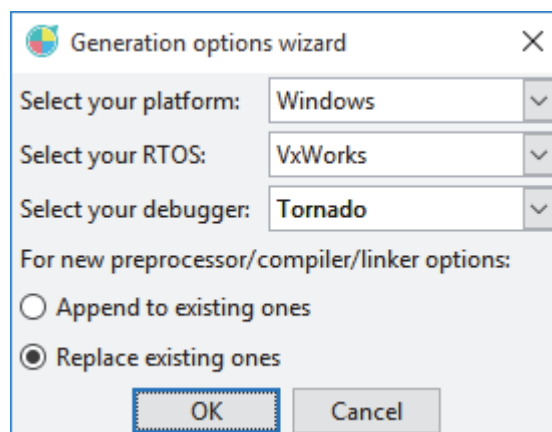
The options in the *Socket connection to target* group configure the host to target communication. They are used by the target program to send trace information:

- to PragmaDev Studio graphical debugger during a debug session;
- to PragmaDev MSC Tracer if its support has been activated.

The value for the *Host IP address* option can be *This host*, meaning the machine where PragmaDev Studio is currently running, *Target host*, meaning the machine where the target will run, or *Forced host* to specify a user-defined IP address.

8.3.2.2 Option wizard

Almost all of the options described in paragraph “Description” on page 157 may be automatically set by using the “Option wizard...” button at the top of the generation profile dialog. Pressing this button will display the following dialog:



The dialog allows to modify the currently displayed profile to work with a standard environment provided with PragmaDev Studio, identified by the platform it runs on, the

RTOS and the debugger used if needed. If the updated profile already has any field set, it's better to choose the *Append to existing ones* option.

Validating the dialog will then automatically set the following fields:

- *Code templates directory*,
- Preprocessor, compiler, linker commands and options,
- For some RTOSs, *Additional files to link* and/or *Include external makefile*,
- *Compilation command*,
- For some RTOSs, *Before build command* and/or *After build command*,
- *Debug* environment, plus *Debugger command* if environment is not *None*.

The check-boxes *Generate makefile* and *Do build* are also automatically checked.

8.3.2.3 VxWorks profile

PragmaDev Studio includes a code template directory to generate VxWorks applications and the SDL-RT debugger is interfaced with Tornado providing a consistent environment.

The profile characteristics are:

- Timer values are set in number of system ticks,
- The SDL-RT process priorities are the ones of VxWorks. The default value is 150.

To generate VxWorks application, it is recommended to use Wind River special make utility and define the CPU. The corresponding makefile will be automatically generated and compiled with the right options. Classical CPU definitions are:

- **SIMNT**
When generating applications to be executed on VxSim on Windows
- **SIMSPARCSOLARIS**
When generating applications to be executed on VxSim on Solaris
- **PENTIUM**
When generating applications to be executed on a Pentium target
- **PPC860**
When generating applications to be executed on a PowerPC 860 target
- ...

The compiler command, the linker command, and the compiler options should be set to use the ones from Wind River:

- compiler command: `$(CC)`
- compiler options: `$(CFLAGS)`
- linker command: `$(LD_PARTIAL)`

The Tornado debugger command is different for each target. Classical debugger commands are:

- **gdbsimnt**
When debugging on Windows host with VxSim
- **gdbsimso**
When debugging on Solaris host with VxSim
- **gdbi86**
When debugging on Pentium based targets whatever the communication link is (serial or ethernet)
- **gdbppc**
When debugging on PowerPC based targets whatever the communication link is (serial or ethernet)
- ...

Note the generated makefile requires to include a Wind River specific part:

```
include C:\RTDS\share\ccg\vxworks\make\WindMake.inc
```


to be interpreted and expanded by Wind River make utility.

On Windows:

Generation options

Profiles: Tornado for Windows

Code gen. Build Debug/trace

General options

Destination directory: ccg Browse...

Code templates dir.: \${RTDS_HOME}\share\ccg\vxworks Browse...

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: ☐

Generate environment process: ☒

Communicate with env. via macros: ☒

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☒

Option wizard...

Generation options

Profiles: Tornado for Windows

Code gen. Build Debug/trace

☒ Generate makefile

Command : Options :

Preprocessor : Options...

Compiler : \$(CC) \$ (CFLAGS) Options...

Linker : \$(LD_PARTIAL) Options...

Additional files to link :

Browse... Add Remove

☒ Include external makefile \${RTDS_HOME}\share\ccg\vxworks\make\WindMake.inc Browse...

☒ Do build

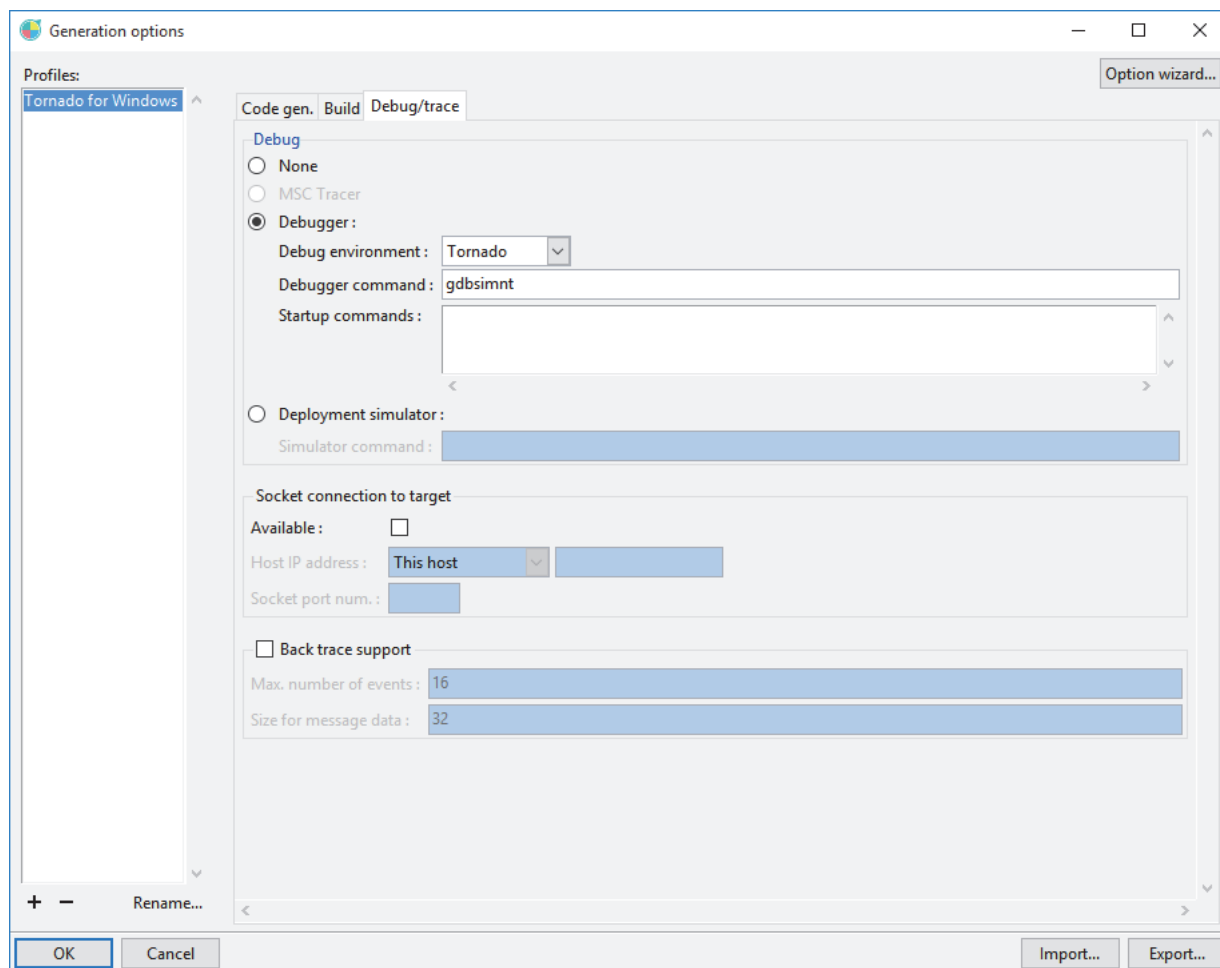
Before build command :

Build command : Exe/options: make Target: all Add'l args: CPU=SIMNT Options...

After build command :

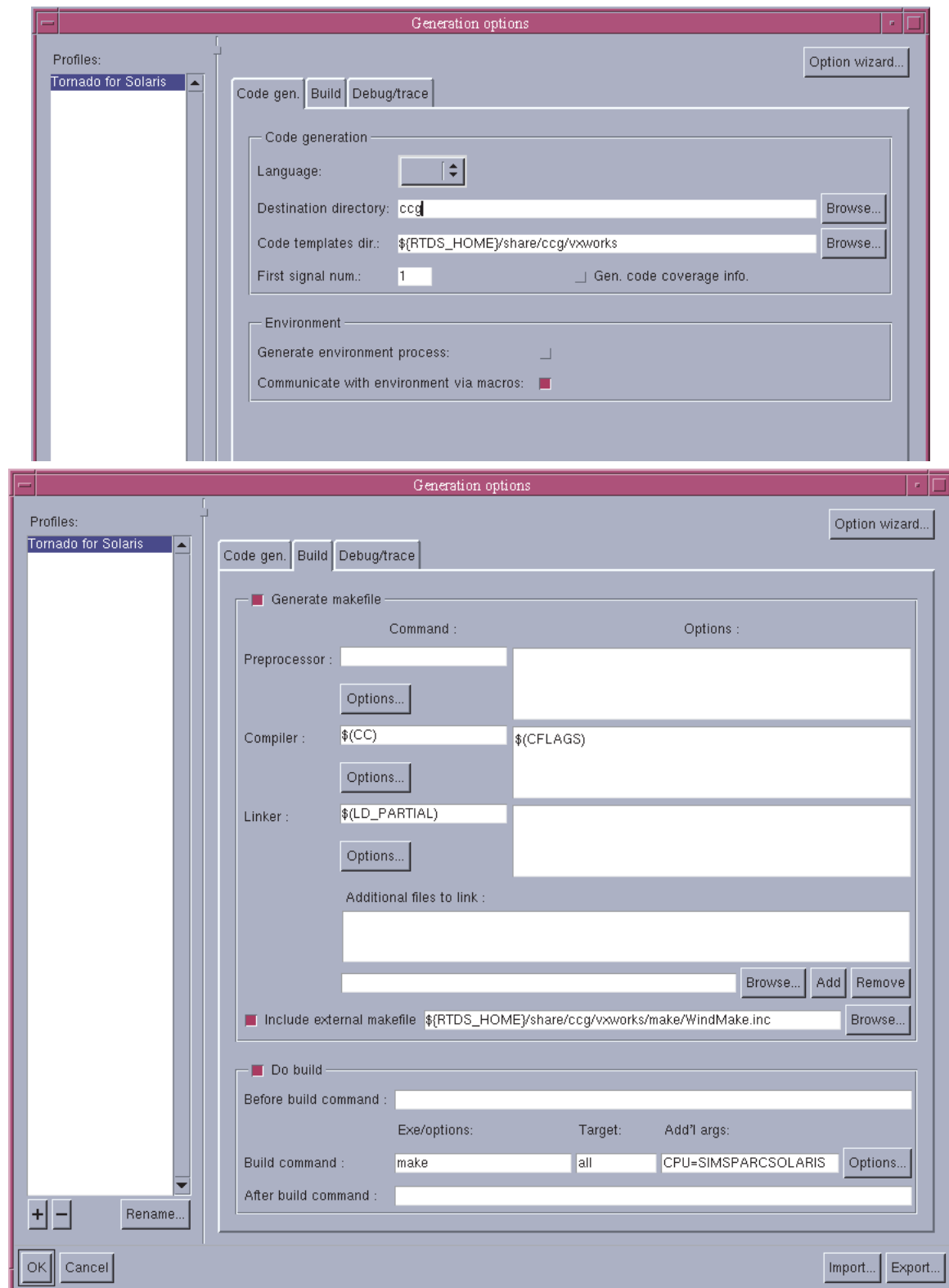
+ - Rename...

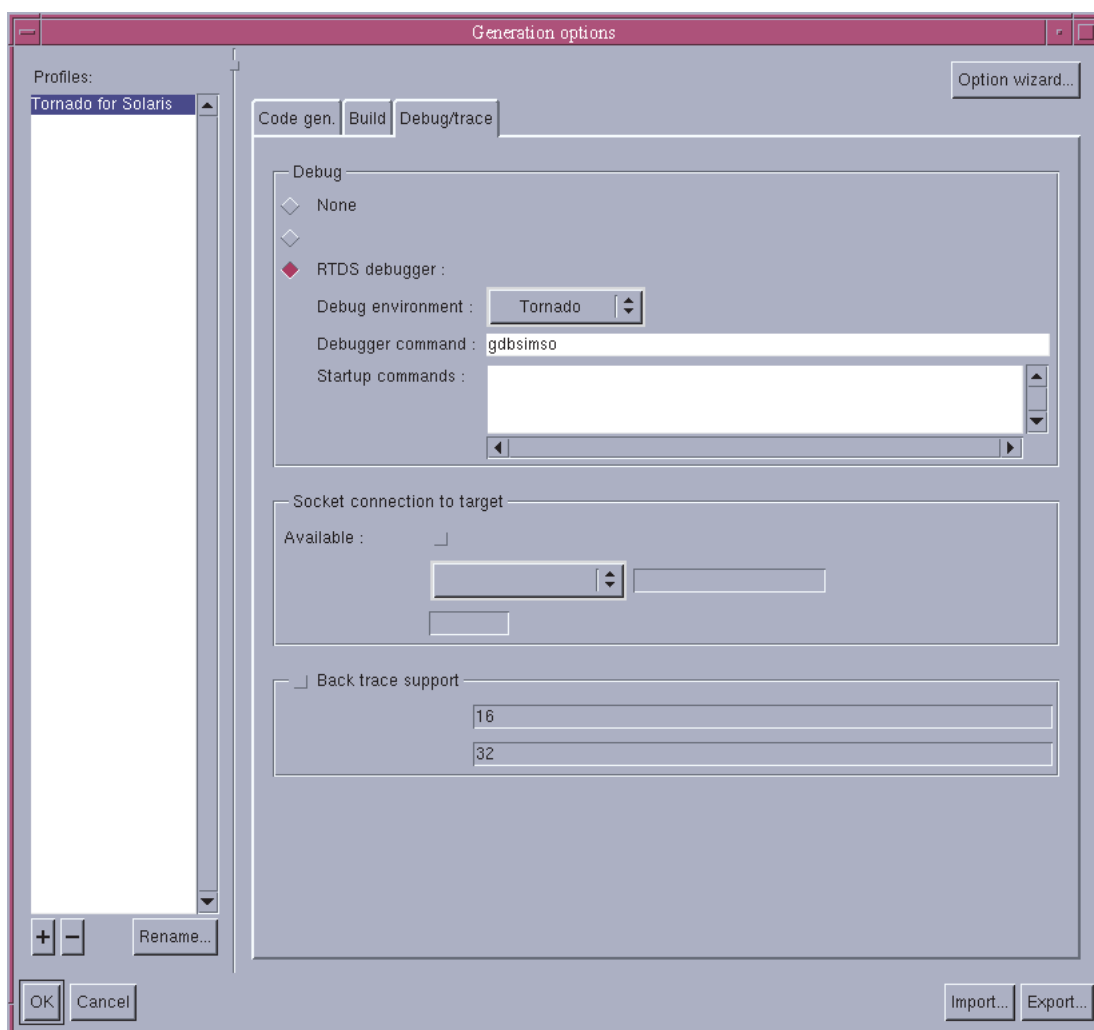
OK Cancel Import... Export...



Typical generation profile to debug with VxSim on Windows

On Solaris:





Typical generation profile to debug with VxSim on Solaris

8.3.2.4 CMX RTX profile

PragmaDev Studio includes a code template directory to generate CMX applications and the SDL-RT debugger is interfaced with Tasking Cross View Pro providing a consistent environment. In the current release the Tasking debugging profile requires the application to run on CMX RTOS and that integration is only available on Windows.

The profile characteristics are:

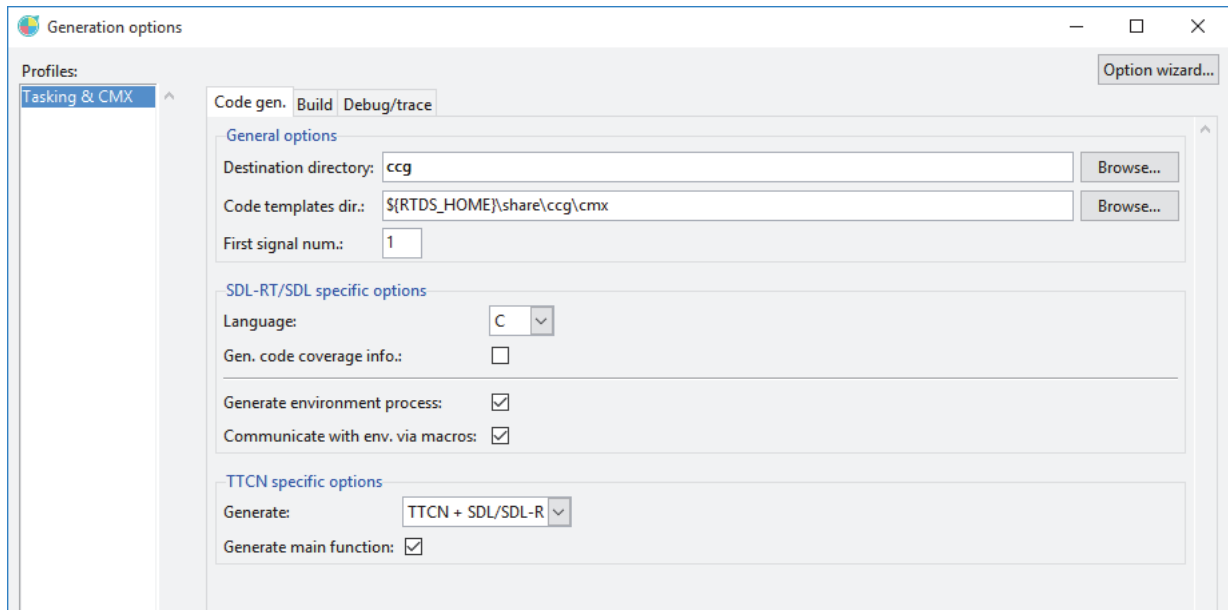
- Timer values are set in number of system ticks,
- CMX supports only counting semaphores with a FIFO queueing mechanism. SDL-RT mutex and binary semaphores have been mapped to counting semaphore,
- The SDL-RT process priorities are the ones of CMX. The default value is 150.
- When an SDL-RT task is deleted, its stack is not claimed. This is because the CMX `K_Task_Delete` does not free the stack space. Therefore systems creating and deleting a lot of tasks will run out of stack after a while. If dynamic creation and deletion is necessary the user should the CMX integration files the following way:
 - add a stack address field in the `RTDS_GlobalProcessInfo` structure in `RTDS_OS.h` file,
 - manually allocate memory for stack and store it in the `RTDS_globalProcessInfo` chained list,
 - use the `K_Task_Create_Stack` function instead of the `K_Task_Create`, in function `RTDS_ProcessCreate` in `RTOS_OS.c` file,
 - free the memory allocated for the stack after the `K_Task_Delete` in function `RTDS_ProcessKill` in `RTDS_OS.c` file.

Please read CMX manual chapters on stacks and on `K_Task_Create_Stack` function for implementation details.

Tasking profile for CMX has the following characteristics:

- CMX RTOS needs to be compiled with the application because SDL-RT debugger will look for some CMX symbols. In order to do so:
 - *Include external makefile* can be used to compile CMX kernel
 - *Additional files to link* can be used to link with CMX libraries
- a utility is needed to generate the final code such as the *ieee166*. To do so the *After compil.* command can be used but watch the file names:
 - the target file name after the makefile is done is referenced by `%RTDS_TARGET_NAME%`,
 - the target file name without extension is referenced by `%RTDS_TARGET_BASE_NAME%`,
 - the SDL-RT debugger will try to load the `<SDL-RT system name>.abs` first and `<SDL-RT system name>.exe` if it did not work.

Last but not least the debugger command uses a configuration file so that the debugger is properly set up straight away. Check Tasking manuals for more information.



Generation options

Profiles: Tasking & CMX

Code gen. Build Debug/trace

General options

Destination directory: ccg Browse...

Code templates dir.: \${RTDS_HOME}\share\cgg\cmx Browse...

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: ☐

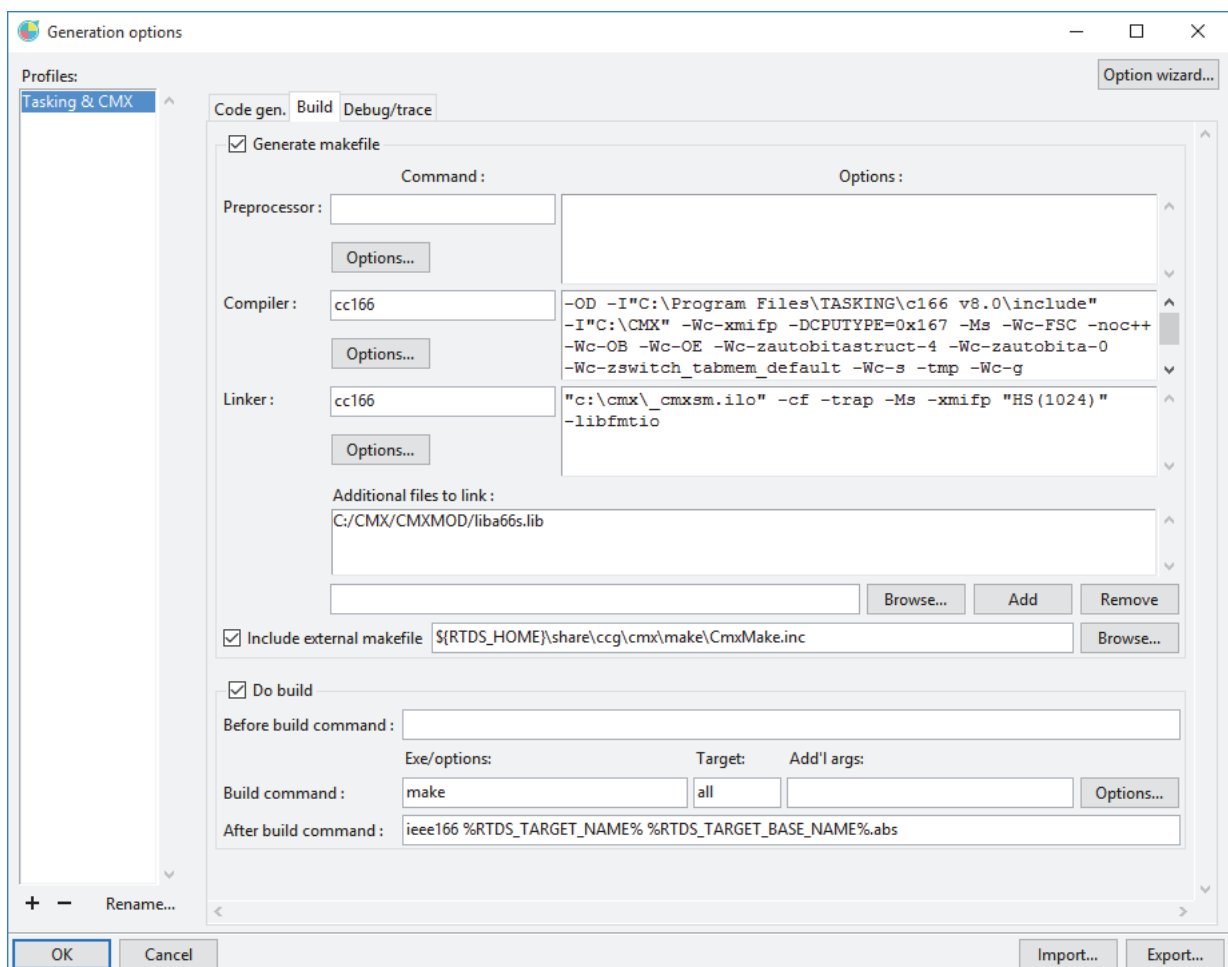
Generate environment process: ☒

Communicate with env. via macros: ☒

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☒



Generation options

Profiles: Tasking & CMX

Code gen. Build Debug/trace

☒ Generate makefile

Command : Options :

Preprocessor : Options...

Compiler : cc166 Options...

Linker : cc166 Options...

Additional files to link : C:/CMX/CMXMOD/liba66s.lib

☒ Include external makefile \${RTDS_HOME}\share\cgg\cmx\make\CmxMake.inc

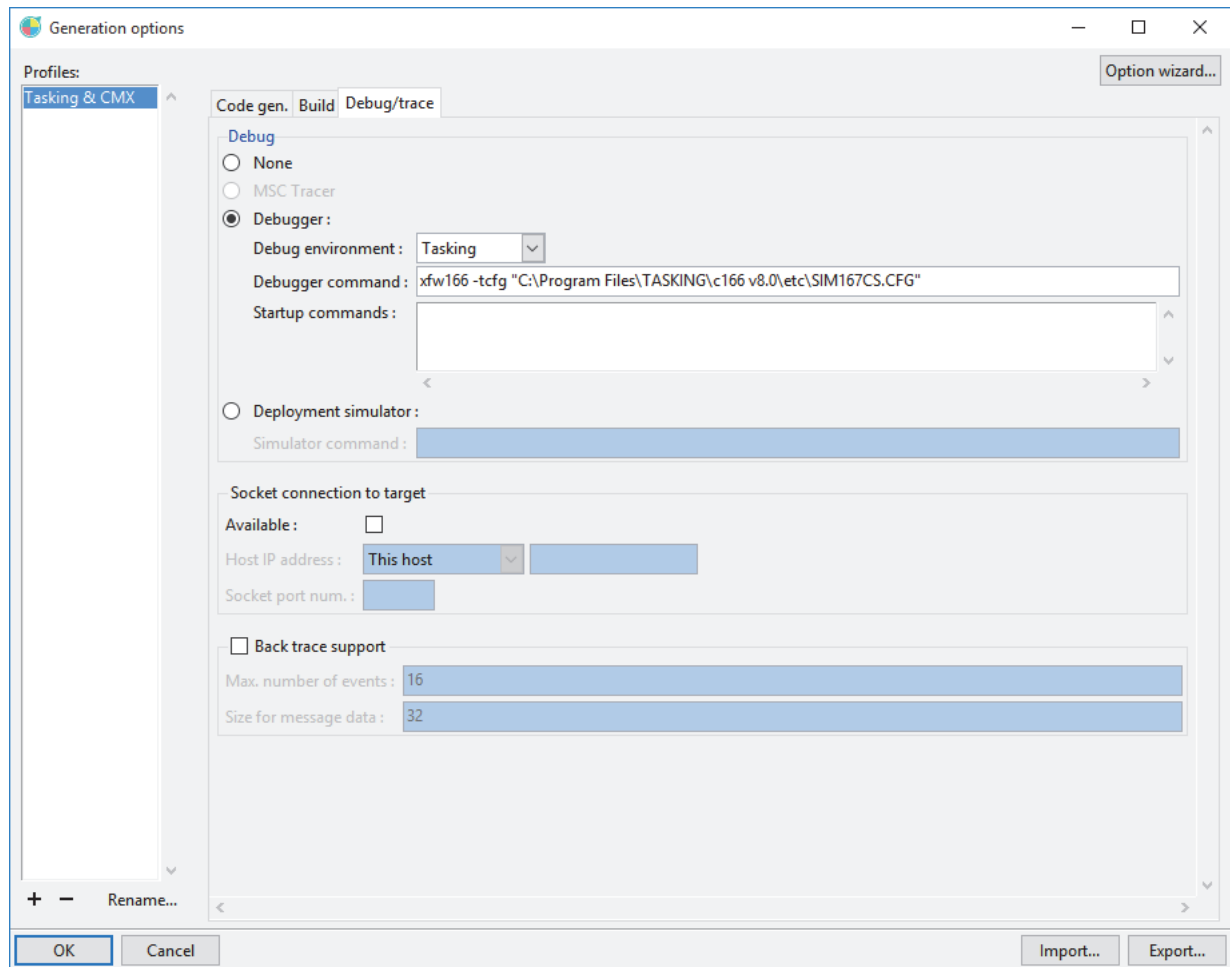
☒ Do build

Before build command :

Build command : make Exe/options: all Target: Add'l args: Options...

After build command : ieee166 %RTDS_TARGET_NAME% %RTDS_TARGET_BASE_NAME%.abs

OK Cancel Import... Export...



Typical generation profile to debug with Tasking a CMX application

Please note the Cygwin make is used in the generation profile above.

8.3.2.5 ThreadX profile

PragmaDev Studio includes a code template directory to generate ThreadX applications.

The profile characteristics are:

- Timer values are set in number of system ticks,
- ThreadX supports counting semaphores and mutex with FIFO queueing mechanism. SDL-RT counting and binary semaphores have been mapped to counting semaphore,
- The SDL-RT process priorities are the ones of ThreadX. Valid numerical priorities range between 0 and 31, where a value of 0 indicates the highest thread priority and a value of 31 represents the lowest thread priority. The default value is 15.

Here is an example profile to compile with Green Hills compiler:

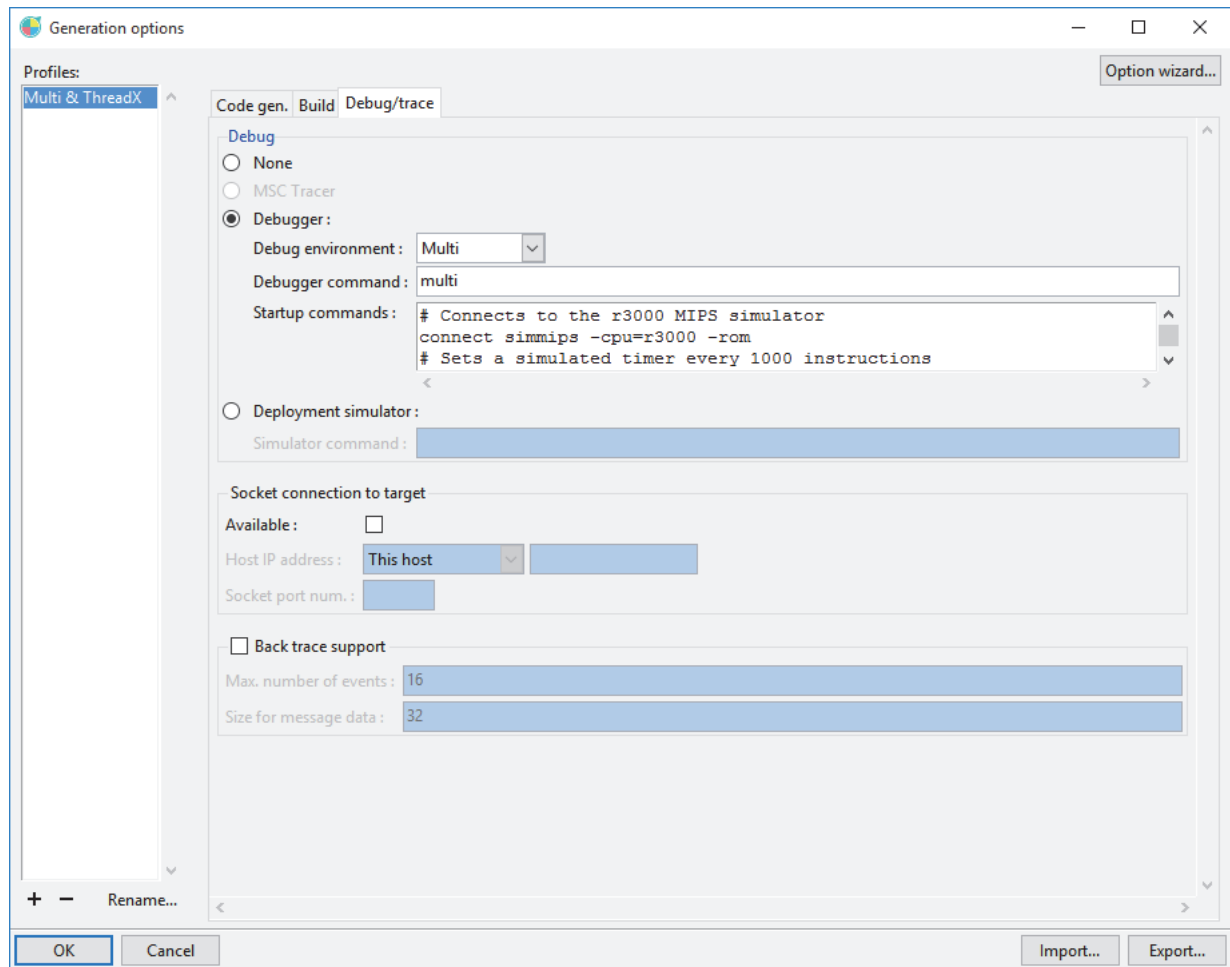
The image displays two screenshots of the 'Generation options' dialog box, showing different tabs for configuration.

Top Screenshot (Code gen. tab):

- Profiles:** Multi & ThreadX
- Code gen. Build Debug/trace**
- General options**
 - Destination directory: ccg
 - Code templates dir.: \${RTDS_HOME}\share\ccg\threadx
 - First signal num.: 1
- SDL-RT/SDL specific options**
 - Language: C
 - Gen. code coverage info.: ☐
 - Generate environment process: ☒
 - Communicate with env. via macros: ☒
- TTCN specific options**
 - Generate: TTCN + SDL/SDL-R
 - Generate main function: ☒

Bottom Screenshot (Build tab):

- Profiles:** Multi & ThreadX
- Code gen. Build Debug/trace**
- Generate makefile** ☒
 - Command: Preprocessor: Options: Compiler: ccmips Options: Linker: Options: Additional files to link: Include external makefile: \${RTDS_HOME}\share\ccg\threadx\make\ThreadX.inc
- Do build** ☒
 - Before build command: Exe/options: Target: Add'l args: Build command: make After build command:



Typical generation profile to compile a ThreadX application with Multi 2000 for MIPS

8.3.2.6 Posix profile

The profile characteristics are:

- Process creation
SDL Task priority depend of the OS or RTOS you are using.
 - On Solaris the priority parameter should be between 0 and 59 .
 - On Linux the priority parameter should be between 0 and 99. Note the priority will only work if user has *superuser* privilege.Default SDL-RT priority is 20 for Solaris and 50 for Linux and for both platforms higher values correspond to higher priorities.
- Timers
The time value is set in milliseconds.
- Semaphores
 - FIFO and PRI0 parameters are ignored. Semaphore waiting queues are always FIFO based.
 - When using mutex semaphores DELETE_SAFE and INVERSION_SAFE have no effect.
 - When taking a semaphore the only options available are NO_WAIT and FOREVER and any other number of milliseconds will be understood as FOREVER.

Here is an example of generation options for Posix under Linux:

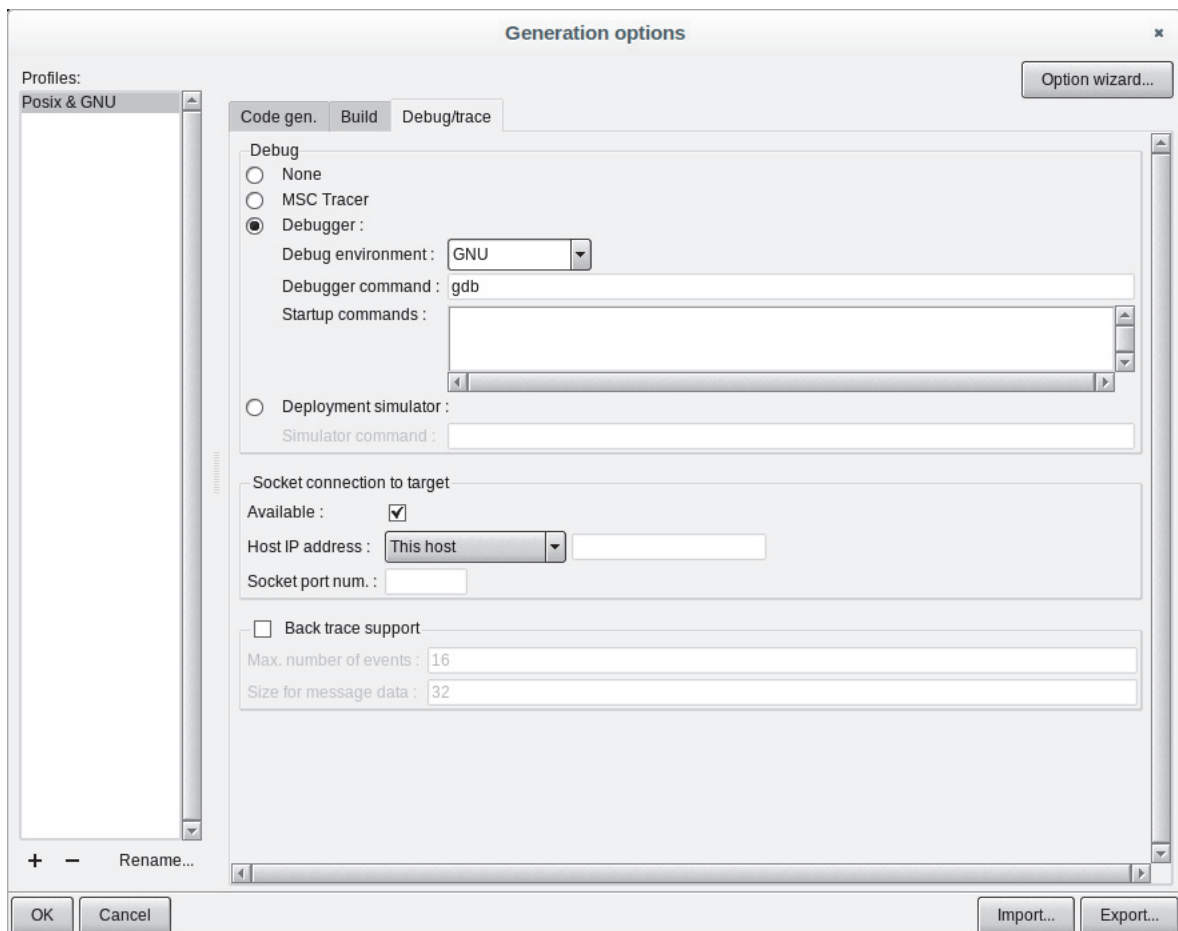
The image displays two screenshots of the "Generation options" dialog box in PragmaDev Studio, specifically for the "Posix & GNU" profile.

Top Screenshot (Code gen. tab):

- Profiles:** Posix & GNU
- Code gen. tab:**
 - General options:**
 - Destination directory: ccg
 - Code templates dir.: \${RTDS_HOME}/share/ccg/posix
 - First signal num.: 1
 - SDL-RT/SDL specific options:**
 - Language: C
 - Gen. code coverage info.: ☐
 - Generate environment process: ☒
 - Communicate with env. via macros: ☒
 - TTCN specific options:**
 - Generate: TTCN + SDL/SDL-RT
 - Generate main function: ☒

Bottom Screenshot (Build tab):

- Profiles:** Posix & GNU
- Build tab:**
 - ☒ **Generate makefile**
 - Command :
 - Options :
 - Preprocessor :
 - Compiler : gcc
 - Linker :
 - Additional files to link :
 - ☐ **Include external makefile**
 - ☒ **Do build**
 - Before build command :
 - Build command : make
 - After build command :



The code template directory is: `${RTDS_HOME}/share/ccg/posix`

The linker options depend on the target platform:

- Solaris
-lpthread -lrt
- Linux
-lpthread

Note: When debugging on Solaris with gdb the linker options are:

`-lpthread -lrt -lsocket -lnsl`

8.3.2.7 Windows profile

The profile characteristics are:

- Process creation

The priority parameter should be one of the following numerical values defined in `winbase.h`:

- `THREAD_PRIORITY_TIME_CRITICAL` = 15
- `THREAD_PRIORITY_HIGHEST` = 2
- `THREAD_PRIORITY_ABOVE_NORMAL` = 1
- `THREAD_PRIORITY_NORMAL` = 0
- `THREAD_PRIORITY_BELOW_NORMAL` = -1
- `THREAD_PRIORITY_LOWEST` = -2
- `THREAD_PRIORITY_IDLE` = -1

Default SDL-RT priority is 0 (zero)



SDL-RT process definition and creation example with Windows priority

In the current release it is not possible to use the macro itself. The numerical value should be used instead.

- Timers
The time value is set in milliseconds.
- Semaphores
 - FIFO and PRIO parameters are ignored. Semaphore waiting queues are always FIFO based.
 - When using mutex semaphores `DELETE_SAFE` and `INVERSION_SAFE` have no effect.

Here is an example of Generation options for Windows with MinGW compiler, make utility and debugger:

Generation options

Profiles: Win32 & MinGW

Code gen. Build Debug/trace

General options

Destination directory: ccg

Code templates dir.: \${RTDS_HOME}\share\ccg\windows

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.:

Generate environment process: [checked]

Communicate with env. via macros: [checked]

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: [checked]

Generation options

Profiles: Win32 & MinGW

Code gen. Build Debug/trace

Generate makefile [checked]

Command :

Options :

Preprocessor :

Options...

Compiler : mingw32-gcc

Options...

Linker :

Options...

Additional files to link :

\${RTDS_HOME}\share\3rdparty\MinGW\lib\libws2_32.a

Browse... Add Remove

Include external makefile [unchecked]

Do build [checked]

Before build command :

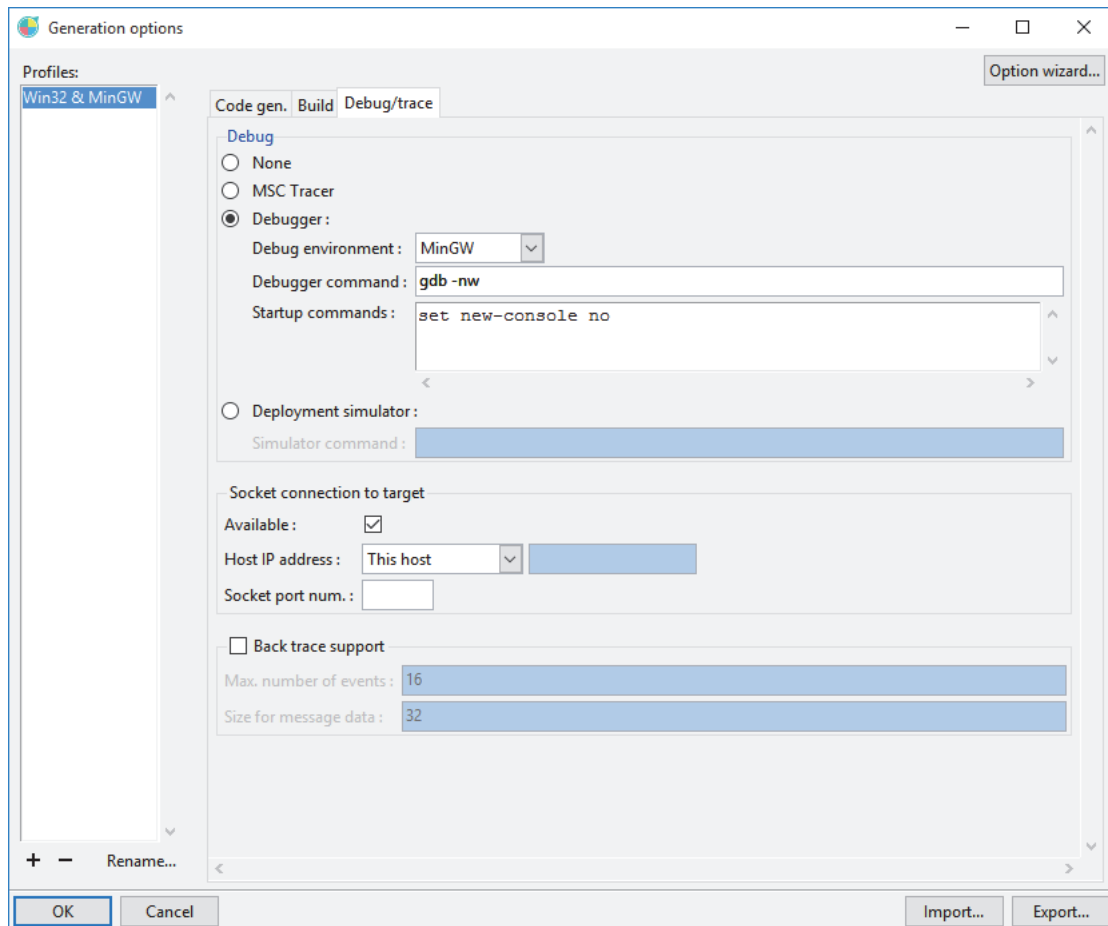
Exe/options: Target: Add'l args:

Build command : mingw32-make

Options...

After build command :

OK Cancel Import... Export...



The code template directory is: `${RTDS_HOME}/share/ccg/windows`

Note: The debugger command should be `gdb -nw` and the console should be removed for MinGW version of gdb: `set new-console no`.

8.3.2.8 uITRON 3.0 profile

This profile has been developed and tested under Linux with the uITRON interface for eCos, the XRAY debugger and the ARM emulator: Armulator.

The profile characteristics are:

- Process creation
uITRON 3.0 specification allows task priorities between 1 and 8. The smaller the value, the higher the priority. Default SDL-RT task priority is 4.
- Timer
SDL-RT timers are mapped on uITRON alarm mechanism. The SDL-RT time out value is used as is in the alarm parameter.
- Message
SDL-RT messages are mapped uITRON mailbox mechanism.
- Semaphores
 - All three types of semaphore are based on uITRON counting semaphore.
 - When using mutex semaphores, DELETE_SAFE and INVERSION_SAFE have no effect.

Here is an example of Generation options for uITRON as implemented in eCos:

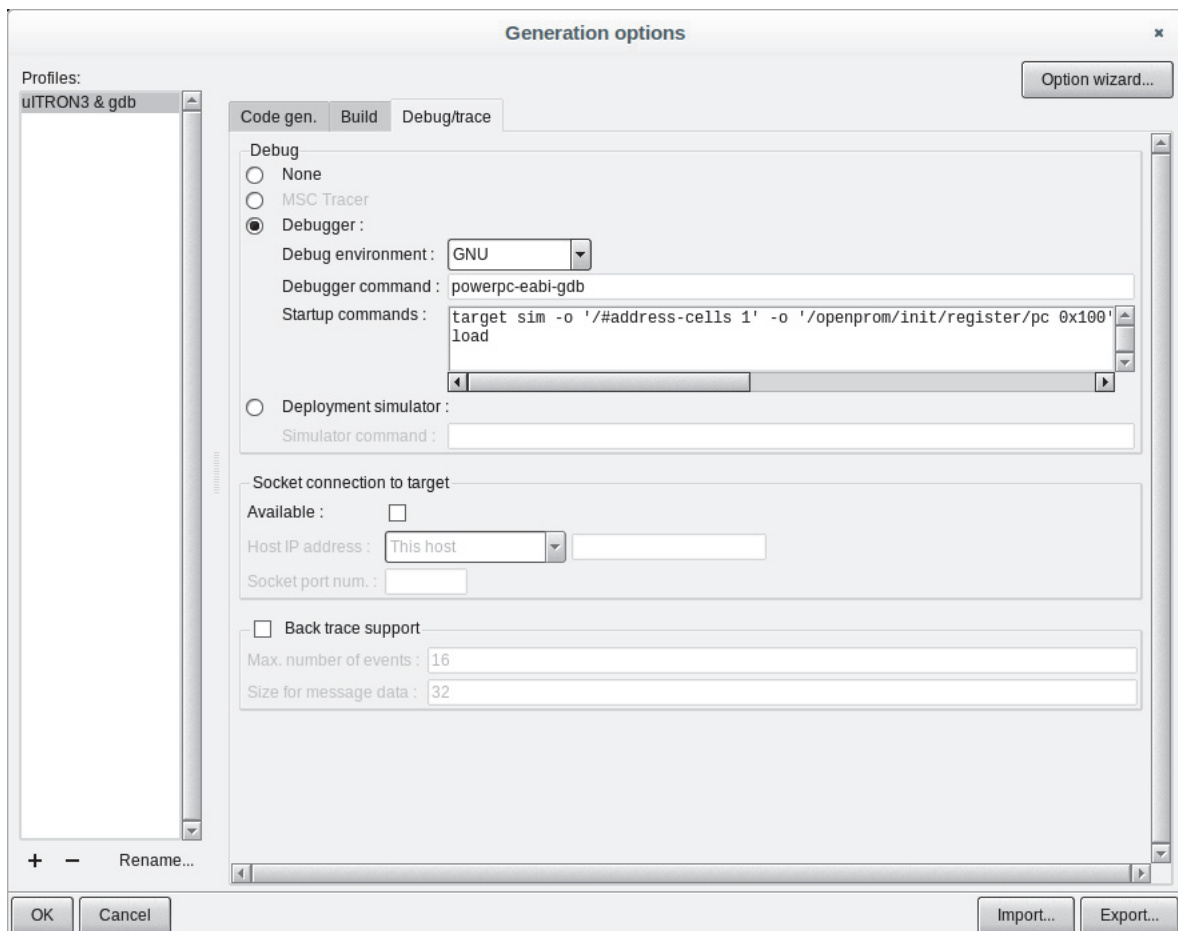
The image displays two screenshots of the "Generation options" dialog box, showing different tabs for configuration.

Top Screenshot (Code gen. tab):

- Profiles:** uITRON3 & gdb
- Code gen. | Build | Debug/trace**
- General options:**
 - Destination directory: ccg
 - Code templates dir.: \${RTDS_HOME}/share/ccg/uitron3_0
 - First signal num.: 1
- SDL-RT/SDL specific options:**
 - Language: C
 - Gen. code coverage info.: ☐
 - Generate environment process: ☒
 - Communicate with env. via macros: ☒
- TTCN specific options:**
 - Generate: TTCN + SDL/SDL-RT
 - Generate main function: ☒

Bottom Screenshot (Build tab):

- Profiles:** uITRON3 & gdb
- Code gen. | Build | Debug/trace**
- ☒ **Generate makefile**
- Command :**
 - Preprocessor :
 - Compiler : powerpc-eabi-gcc
 - Linker :
- Options :**
 - I "\${ECOS_UITRON3_HOME}/include" -I "\${ECOS_UITRON3_HOME}/include/cyg/compat/uitron" -D _RTDS_BOOLEAN_ -D RTDS_BOOLEAN=short
 - L "\${ECOS_UITRON3_HOME}/lib" -Ttarget.ld -nostdlib
- Additional files to link :**
- ☐ **Include external makefile**
- ☒ **Do build**
 - Before build command :
 - Build command : make
 - After build command :



The code template directory is:

```
${RTDS_HOME}/share/ccg/uitron3_0
```

The linker options depend on the RTOS and should add the uITRON library path. For example when using eCos's uITRON interface you should add:

```
-L [uITRON_Library_Path] -Ttarget.ld -nostdlib
```

8.3.2.9 uITRON 4.0 profile

This profile has been tested under Windows with the uITRON interface for NUCLEUS and MinGW GDB debugger.

The profile characteristics are:

- **Process creation**
uITRON 4.0 specification allows task priorities between 1 and 16. Nucleus uITRON API allows to use 1 to 255 task priorities. Default SDL-RT task priority is 4.
- **Timer**
SDL-RT timers are mapped on uITRON alarm mechanism. The SDL-RT time out value is used as is in the alarm parameter.
- **Message**
SDL-RT messages are mapped uITRON mailbox mechanism.
- **Semaphores**
 - Only binary and counting semaphores are supported
 - These two types of semaphore are based on uITRON counting semaphore.
 - When using mutex semaphores, an error is raised

Here is an example of Generation options for uTRON as implemented in NUCLEUS:

The image displays two screenshots of the "Generation options" dialog box, showing different tabs for configuration.

Top Screenshot (Code gen. tab):

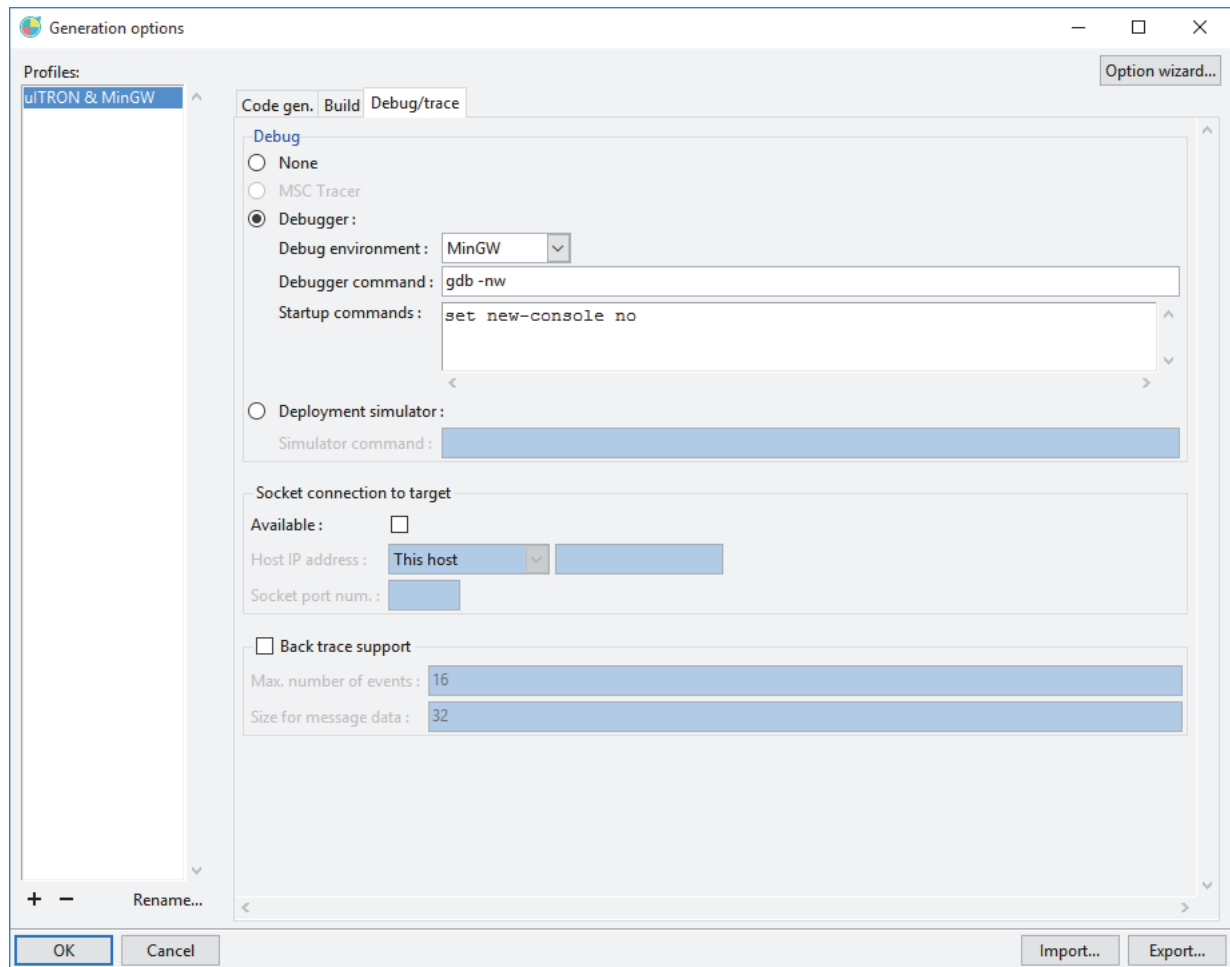
- Profiles:** uTRON & MinGW
- Code gen. | Build | Debug/trace**
- General options:**
 - Destination directory: ccg
 - Code templates dir.: \${RTDS_HOME}\share\ccg\uitron4_0
 - First signal num.: 1
- SDL-RT/SDL specific options:**
 - Language: C
 - Gen. code coverage info.: ☐
 - Generate environment process: ☒
 - Communicate with env. via macros: ☒
- TTCN specific options:**
 - Generate: TTCN + SDL/SDL-R
 - Generate main function: ☒

Bottom Screenshot (Build tab):

- Profiles:** uTRON & MinGW
- Code gen. | Build | Debug/trace**
- Generate makefile:** ☒
 - Command:

Preprocessor :	Options :
Compiler : mingw32-gcc	-mno-cygwin -I"\${SIMTEST_ROOT}\..\nucleus" -I"\${RTDS_HOME}\share\ccg\uitron4_0\Miplus" -DNU_SIMULATION
Linker :	-L"\${SIMTEST_ROOT}\..\nucleus\lib\mingw" -lsim -lplus -lvt -lwinmm -luipplus -mno-cygwin
 - Additional files to link:
 - Browse... Add Remove
 - ☐ Include external makefile Browse...
 - Do build:** ☒
 - Before build command:
 - Build command: mingw32-make -f "%RTDS_HOME%/

Exe/options:	Target:	Add'l args:
	all	
 - After build command:



The code template directory is:

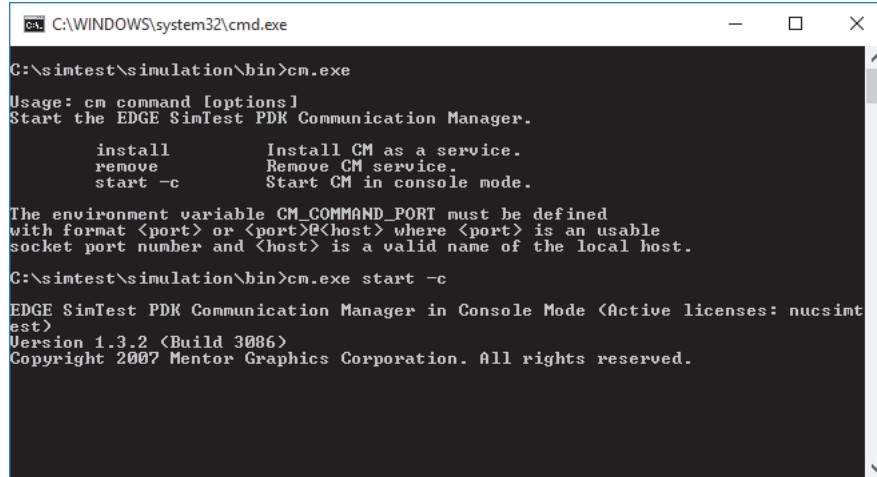
`${RTDS_HOME}/share/ccg/uitron4_0`

The linker options depend on the RTOS and should add the uITRON 4.0 library path. for example when using NUCLEUS's uITRON 4.0 interface you should add:

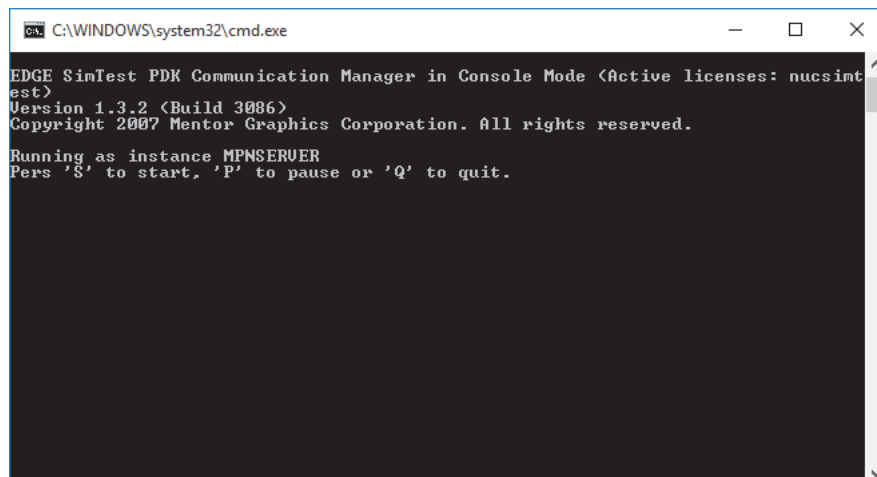
`-L"NUCLEUS_uiPLUS_path" -lsim -lplus -lvt -lwinmm -luiplus`

In order to debug a uITRON4 application with SimTest kernel simulator, some manual operations are required:

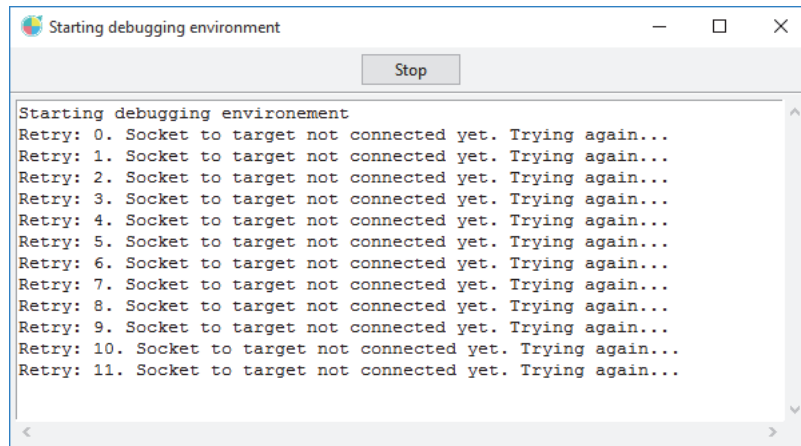
- Start the EDGE communication manager:
%SIMTEST_R00T%\bin\cm.exe start -c



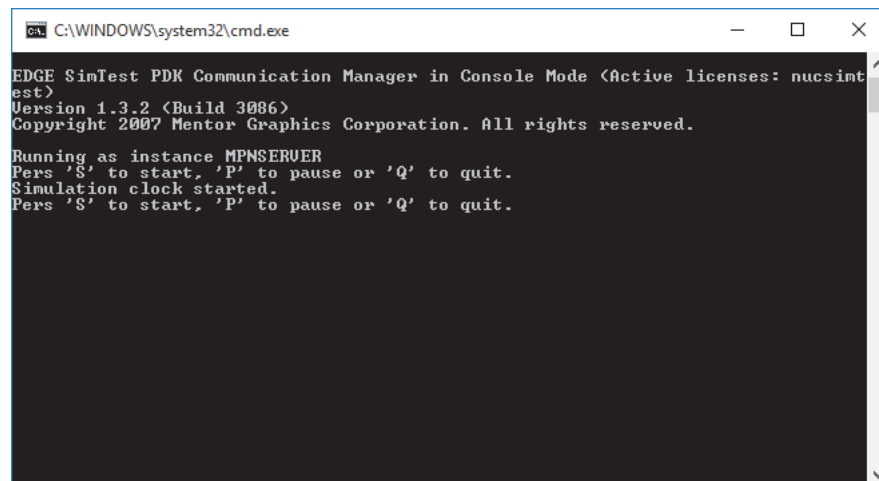
- Start the MPN server:
%SIMTEST_R00T%\..\nucleus\simulation\mpn\bin\mpnserver.exe



- Start PragmaDev Studio debugger, when the debugger tries to connect to the executable:



- Then type 'S' in the MPN server window:



Then the debugger can connect to the target and you can debug normally.

8.3.2.10 OSE Delta 4.5.2 profile

PragmaDev Studio includes a code template directory to generate OSE applications and the SDL-RT debugger is interfaced with gdb providing a consistent environment.

The profile characteristics are:

- **Build process**
 The OSE build process is based on dmake utility and makefile.mk and user-conf.mk makefiles. When using OSE code generation, PragmaDev Studio generates pragmadev.mk makefile in the code generation directory. The makefile.mk and userconf.mk should be put somewhere else because they are not generated file and makefile.mk must include the generated pragmadev.mk file, e.g.:

```
include ./pragmadev.mk
```

 The make command should set the target to RTDS_ALL. For example under windows:

```
dmake -f ../makefile.mk RTDS_ALL RTDS_HOME=%RTDS_HOME%
```

 or under Unix:

```
dmake -f ../makefile.mk RTDS_ALL RTDS_HOME=$RTDS_HOME
```
- **SDL-RT system start**
 The pragmadev.mk generated file includes

```
${RTDS_HOME}\share\ccg\ose\make\OseMake.inc
```

 that tells OSE kernel to statically start RTDS_Start process:

```
PRI_PROC(RTDS_Start, RTDS_Start, 1024, 15, DEFAULT, 0, NULL)
```

 That means SDL-RT static process are not defined as static to OSE kernel. The startup procedure takes care of creating the processes and synchronizing them.
- **Timers handling**
 SDL-RT timers are based on OSE Time-Out Server (TOSV). TOSV should therefore be included in userconf.mk file:

```
INCLUDE_OSE_TOSV *= yes
```
- **Priorities**
 SDL-RT process priorities are the ones of OSE. The default value is 15.
- **Memory management**
 The generated code memory allocation RTDS_MALLOC and RTDS_FREE are based on heap_alloc_shared and heap_free_shared OSE functions. This is because it happens a process frees memory allocated by another process.
- **Signal header and definitions**
 All generated OSE signals will have the following header:

```
typedef struct RTDS_MessageHeader
{
    SIGSELECT      sigNo;
    long           messageNumber;
    long           timerUniqueId;
    RTDS_QueueId   sender;
    long           dataLength;
    unsigned char  *pData;

    struct RTDS_MessageHeader *next;
}
```

```
} RTDS_MessageHeader;
```

```
typedef union SIGNAL
{
    SIGSELECT sigNo;
    struct RTDS_MessageHeader messageHeader;
}SIGNAL;
```

An OSE signal file is generated that contains all SDL-RT signal definitions: `RTDS_gen.sig`. All signals have the `RTDS_MessageHeader` data content. Below is an example of a generated signal file:

```
#include "ose.h"
#include "RTDS_OS.h"

#define ping (1) /* !-SIGN0(struct RTDS_MessageHeader)-! */
#define tWait (2) /* !-SIGN0(struct RTDS_MessageHeader)-! */
#define pong (3) /* !-SIGN0(struct RTDS_MessageHeader)-! */
#define begin (4) /* !-SIGN0(struct RTDS_MessageHeader)-! */
#define myStart (5) /* !-SIGN0(struct RTDS_MessageHeader)-! */
#define myStop (6) /* !-SIGN0(struct RTDS_MessageHeader)-! */
```

Below is an OSE generation profile example with debug based on gdb:

The image displays two screenshots of the 'Generation options' dialog box in PragmaDev Studio, showing the configuration for an OSE generation profile with debug based on gdb.

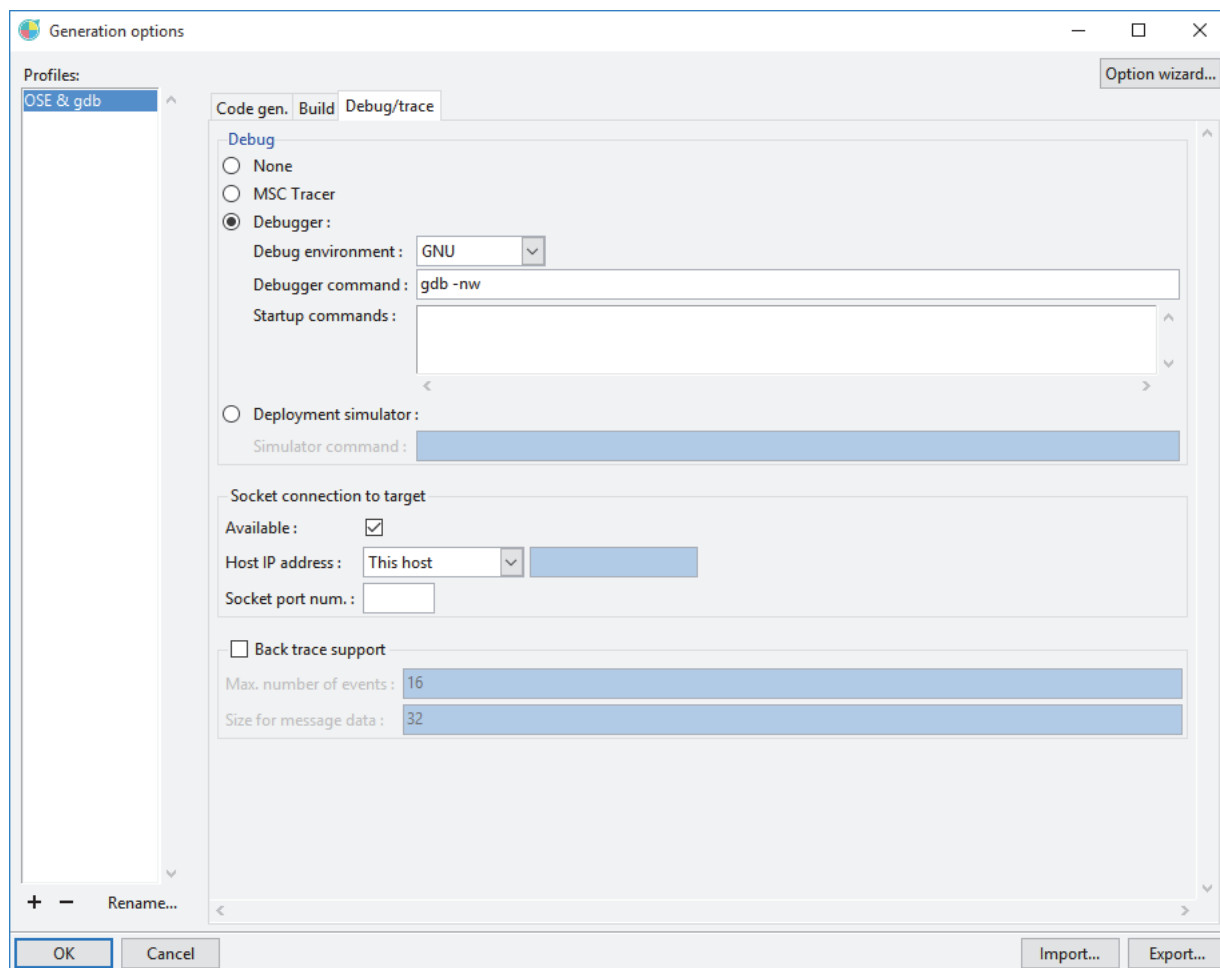
Top Screenshot: Code gen. tab

- Profiles:** OSE & gdb
- Code gen. Build Debug/trace**
- General options**
 - Destination directory: ccg
 - Code templates dir.: \${RTDS_HOME}\share\ccg\ose
 - First signal num.: 1
- SDL-RT/SDL specific options**
 - Language: C
 - Gen. code coverage info.: ☐
 - Generate environment process: ☒
 - Communicate with env. via macros: ☒
- TTCN specific options**
 - Generate: TTCN + SDL/SDL-R
 - Generate main function: ☒

Bottom Screenshot: Build tab

- Generate makefile**: ☒
- Command :**
 - Preprocessor: Options...
 - Compiler: \$(CC) \$(CFLAGS) \$(DEFINES) \$(STDINCLUDE) \$(INCLUDE) \$(CCOUT) Options...
 - Linker: (unused) Options...
- Additional files to link :**
 - Browse... Add Remove
- Include external makefile**: ☒ \${RTDS_HOME}\share\ccg\ose\make\OseMake.inc Browse...
- Do build**: ☒
 - Before build command:
 - Build command: dmake -f %RTDS_HOME%\share\ccg\o Target: RTDS_ALL Add'l args: RTDS_HOME=%RTDS_HOME% Options...
 - After build command:

Buttons at the bottom: OK, Cancel, Import..., Export...



Typical generation profile to debug an OSE application with gdb

8.3.2.11 OSE Epsilon profile

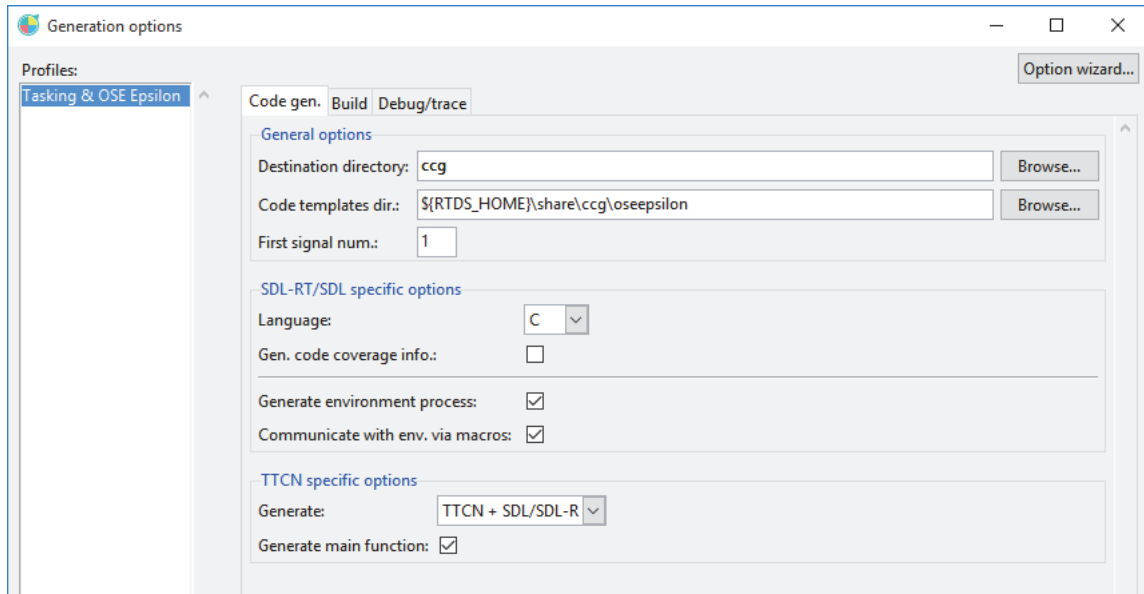
PragmaDev Studio includes a code template directory to generate OSE Epsilon applications.

The profile characteristics are:

- **Build process**
OSE Epsilon is a static RTOS. That means the RTOS is compiled with the application and no dynamic task creation is possible. To do so OSE Epsilon needs to know at compile time the list of task in the system.
During code generation, PragmaDev Studio produces `RTDS_gen.inf` that contains the list of tasks, semaphores, and signals used in the system. A shell script based on `awk` extracts the necessary information out of the file and generates the OSE Epsilon `.con` file needed to configure the kernel. The templates provided in the distribution should run with Tasking C166 cross compiler. The OSE Epsilon file is built based on `os166.con.pre`, the result of the `awk` script and `os166.con.post`. These templates can be adapted to any processor or cross compiler.
- **SDL-RT system start**
As a result of the `.con` file, OSE Epsilon will start all task by itself at startup. `RTDS_Start` task is started with the highest priority in order to initialize the execution environment: create semaphores, message unique id pool, and back trace circular buffer.
All other task will create their own context variable and wait a very short delay to let the other tasks do the same.
- **Timers handling**
SDL-RT timers are based on OSE Time-Out Server (TOSV). TOSV should therefore be included in the `.con` file:

```
%TI_PROC    tosv,C,256,256,1
```
- **Memory management**
The generated code memory allocation `RTDS_MALLOC` and `RTDS_FREE` are based on `malloc` and `free` functions.

The example profile given below is based on Tasking C166 cross compiler:



Generation options

Profiles: Tasking & OSE Epsilon

Code gen. Build Debug/trace

General options

Destination directory: ccg Browse...

Code templates dir.: \${RTDS_HOME}\share\ccg\osepsilon Browse...

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: ☐

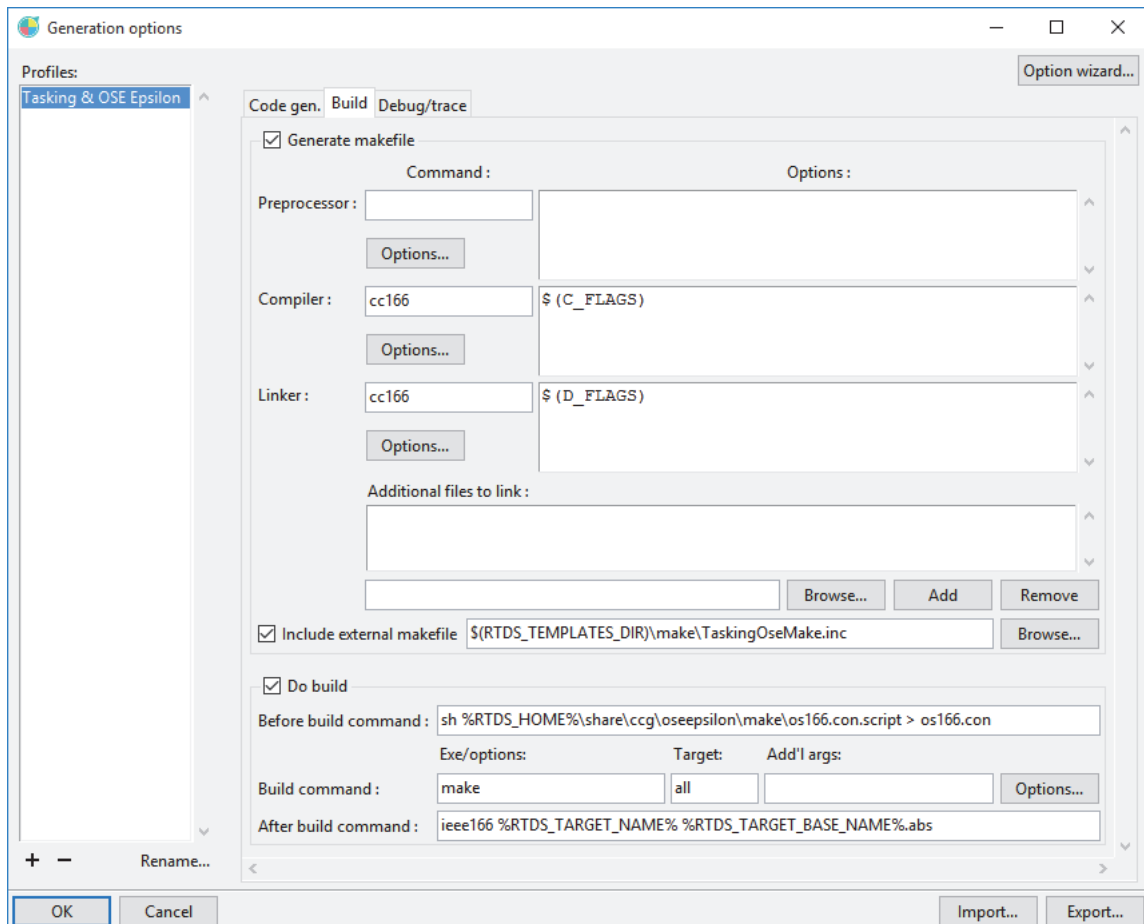
Generate environment process: ☒

Communicate with env. via macros: ☒

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☒



Generation options

Profiles: Tasking & OSE Epsilon

Code gen. Build Debug/trace

☒ Generate makefile

Command: Options:

Preprocessor: Options...

Compiler: cc166 \$ (C_FLAGS) Options...

Linker: cc166 \$ (D_FLAGS) Options...

Additional files to link:

Browse... Add Remove

☒ Include external makefile \${RTDS_TEMPLATES_DIR}\make\TaskingOseMake.inc Browse...

☒ Do build

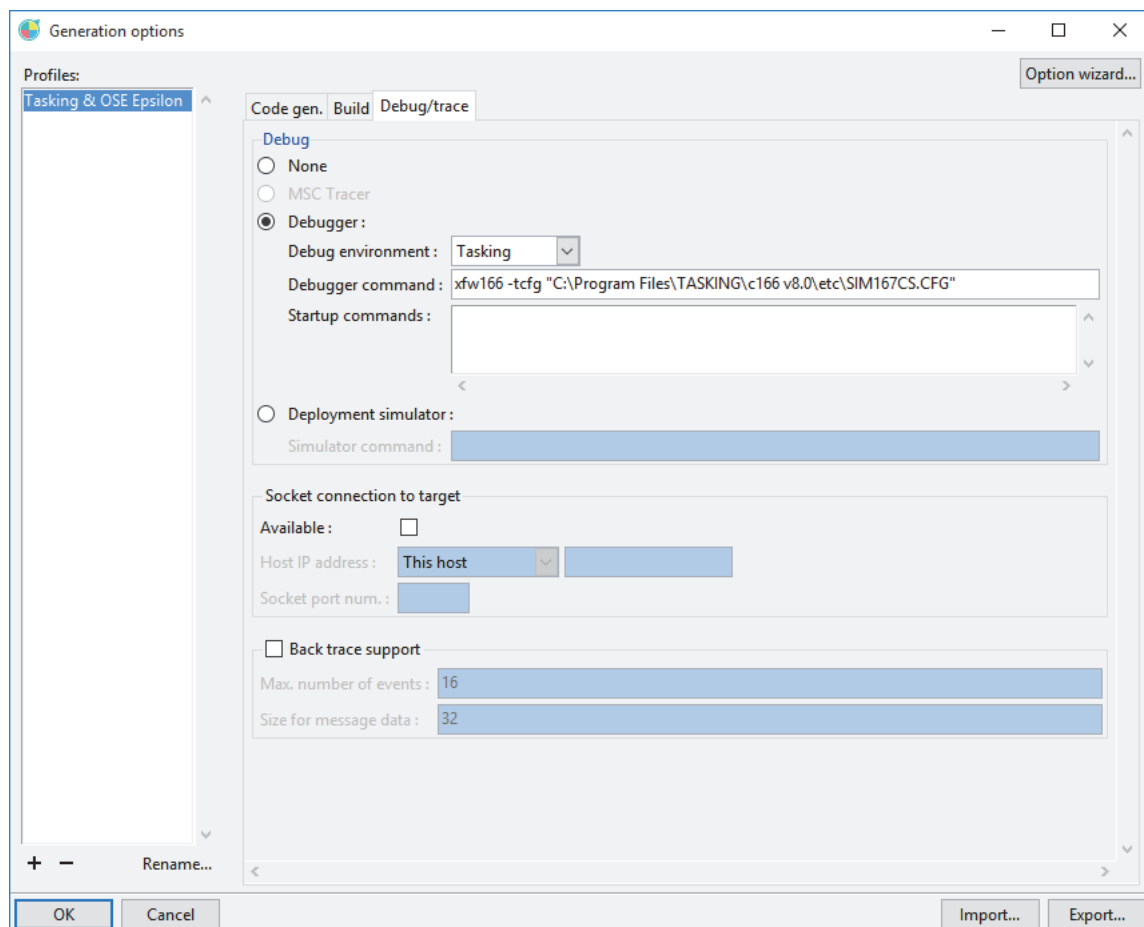
Before build command: sh %RTDS_HOME%\share\ccg\osepsilon\make\os166.con.script > os166.con

Exe/options: Target: Add'l args:

Build command: make all Options...

After build command: ieee166 %RTDS_TARGET_NAME% %RTDS_TARGET_BASE_NAME%.abs

OK Cancel Import... Export...



Typical generation profile to debug an OSE Epsilon application with Tasking

8.3.2.12 Nucleus profile

PragmaDev Studio includes a code template directory to generate Nucleus applications and the SDL-RT debugger is interfaced with a special version of gdb provided by the EDGE environment tool and the SIMTEST tool. In the current release the gdb debugging profile is only supported on Windows.

The profile characteristics are:

- **Build process**
The Nucleus build process is based on mingw32-make utility. Note the Nucleus build process relies on the SIMTEST_ROOT environment variable which is defined by the simtest installer.
- **SDL-RT system start**
The file `${RTDS_HOME}\share\ccg\nucleus\RTDS_OS.c` defines the function `Application_Initialize` that creates the `RTDS_Start` task that initialize PragmaDev Studio environment and objects creation.
- **Priorities**
SDL-RT process priorities are the ones of Nucleus. The default value is 125.
- **Memory management**
The generated code memory allocation `RTDS_MALLOC` and `RTDS_FREE` are based on `NU_Allocate_Memory` and `NU_Deallocate_Memory` Nucleus functions. The memory management is done by using a unique memory pool.

Below is an Nucleus generation profile example with debug based on gdb:

Generation options

Profiles: Nucleus & SimTest

Code gen. Build Debug/trace

General options

Destination directory: ccg Browse...

Code templates dir.: \${RTDS_HOME}\share\ccg\nucleus Browse...

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: ☐

Generate environment process: ☒

Communicate with env. via macros: ☒

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☒

Generation options

Profiles: Nucleus & SimTest

Code gen. Build Debug/trace

☒ Generate makefile

Command:

Preprocessor: Options...

Compiler: mingw32-gcc Options...

Options: -mno-cygwin -I\"\$(SIMTEST_ROOT)\\..\\nucleus\" -DNU_SIMULATION

Linker: Options...

Options: -L\"\$(SIMTEST_ROOT)\\..\\nucleus\\lib\\mingw\" -lsim -lplus -lvt -lwinmm -mno-cygwin

Additional files to link:

Browse... Add Remove

☐ Include external makefile Browse...

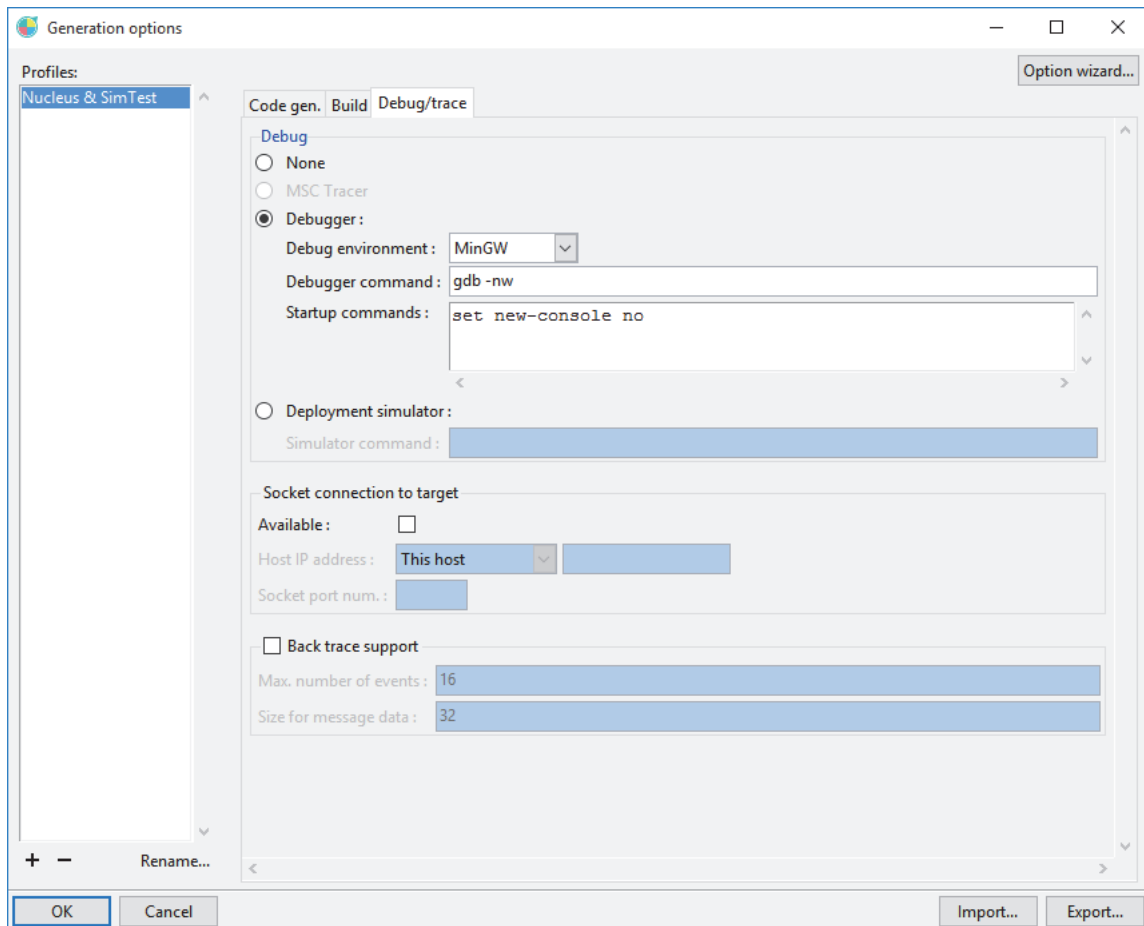
☒ Do build

Before build command:

Build command: mingw32-make -f \"%RTDS_HO all Exe/options: Target: all Add'l args: Options...

After build command:

OK Cancel Import... Export...



Typical generation profile to debug a Nucleus application with gdb

In order to debug a Nucleus application with SimTest kernel simulator, some manual operations are required:

- Start the EDGE communication manager:
%SIMTEST_R00T%\bin\cm.exe start -c

```

C:\WINDOWS\system32\cmd.exe

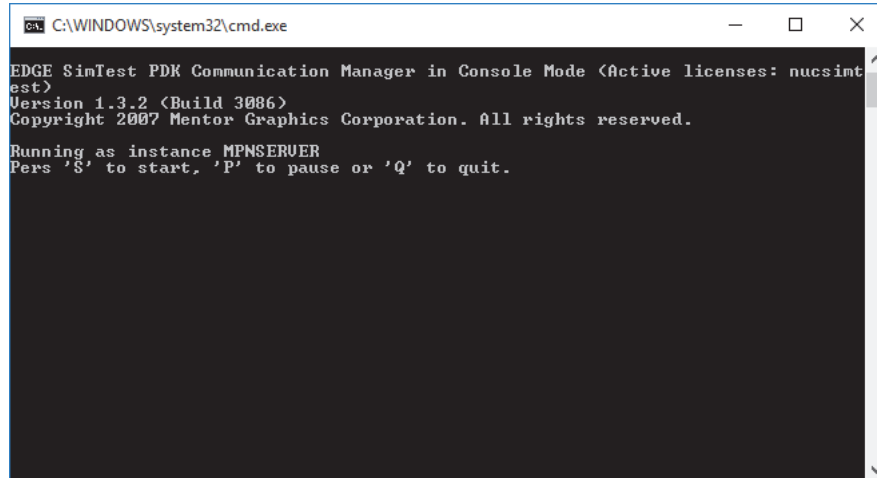
C:\simtest\simulation\bin>cm.exe
Usage: cm command [options]
Start the EDGE SimTest PDK Communication Manager.

    install      Install CM as a service.
    remove      Remove CM service.
    start -c     Start CM in console mode.

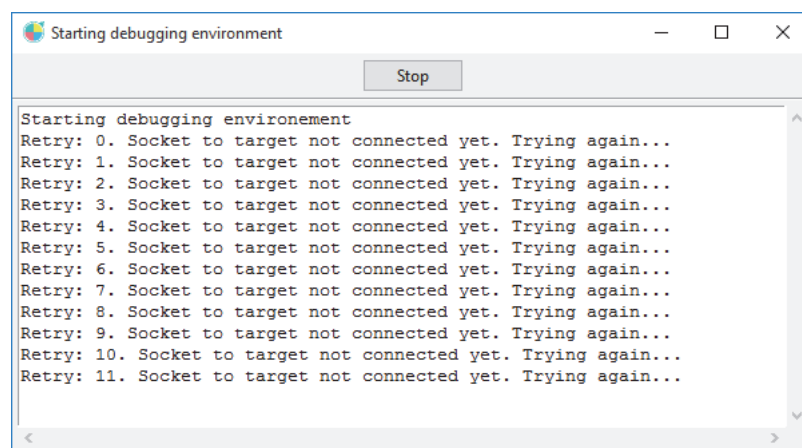
The environment variable CM_COMMAND_PORT must be defined
with format <port> or <port>@<host> where <port> is an usable
socket port number and <host> is a valid name of the local host.

C:\simtest\simulation\bin>cm.exe start -c
EDGE SimTest PDK Communication Manager in Console Mode <Active licenses: nucsint
est>
Version 1.3.2 <Build 3086>
Copyright 2007 Mentor Graphics Corporation. All rights reserved.
  
```

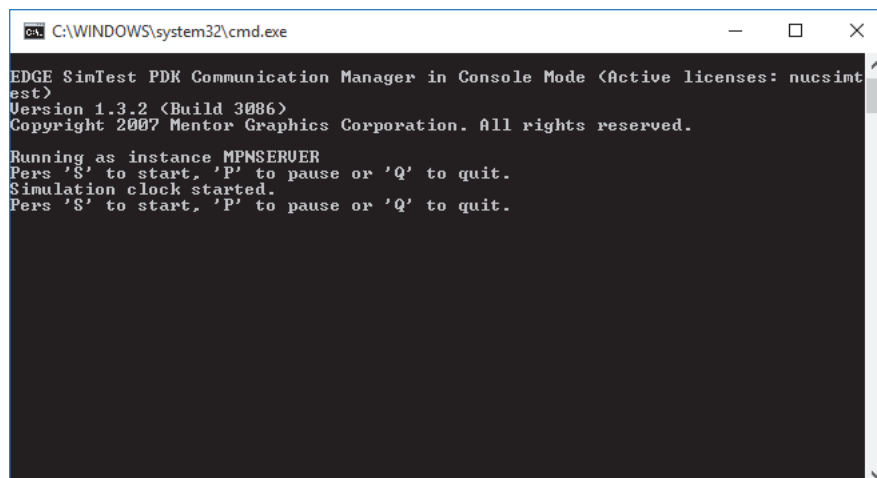
- Start the MPN server:
`%SIMTEST_ROOT%\..\nucleus\simulation\mpn\bin\mpnserver.exe`



- Start PragmaDev Studio debugger, when the debugger tries to connect to the executable:



- Then type 'S' in the MPN server window:



Then the debugger can connect to the target and you can debug normally.

8.3.2.13 FreeRTOS profile

PragmaDev Studio includes a code template directory to generate FreeRTOS applications. This integration has been done with the FreeRTOS simulator on Windows and is using the MinGW gdb coming with PragmaDev Studio as a debugger integration. This integration was done on FreeRTOS V7.1.0 and Windows 7 Professional Service Pack 1.

The profile characteristics are:

- **Build process**
The build process created by the wizard assumes the FreeRTOS directories are in `C:\FreeRTOS\FreeRTOSV7.1.0`. If not the case the include paths in the compiler options need to be adjusted in the generation profile.
- **FreeRTOS Windows Simulator**
To ease the integration development, we used the FreeRTOS Windows Simulator. In order to have an efficient integration, communication through socket has been implemented between the FreeRTOS Simulator and the PragmaDev Studio Debugger. The socket communication is implemented in a Windows thread that is created on the side of the other FreeRTOS Windows threads. We tried to gather all these specific FreeRTOS Simulator aspects in the `RTDS_TCP_Client.c` file. The `RTDS_FREERTOS_WINDOWS_SIMULATOR` define surrounds these specific part in other files as well.
- **FreeRTOS on target**
To build for a specific target:
 - Remove `-DRTDS_FREERTOS_WINDOWS_SIMULATOR` from the generation options,
 - Remove `RTDS_TCP_Client.o $(RTDS_HOME)/share/3rdparty/MinGW/lib/libws2_32.a` from the `$(RTDS_HOME)/share/ccg.freertos/make/FreeRtosMake.inc` file.
- **Debugging issues**
Please note that if the command “info threads” is sent to gdb, the integration will crash with a segmentation fault when the system continues execution.
- **Priorities**
The priority values are unclear in FreeRTOS. With the FreeRTOS Windows Simulator it looks like only 7 levels are available so the default priority `RTDS_DEFAULT_PROCESS_PRIORITY` has been set to 3.
- **Semaphores**
The initial value of a semaphore with FreeRTOS is always available. In the case of a binary semaphore, a take is executed after creation if the semaphore initial state is empty.
- **Memory management**
The integration is using `pvPortMalloc` and `vPortFree` for memory allocation.

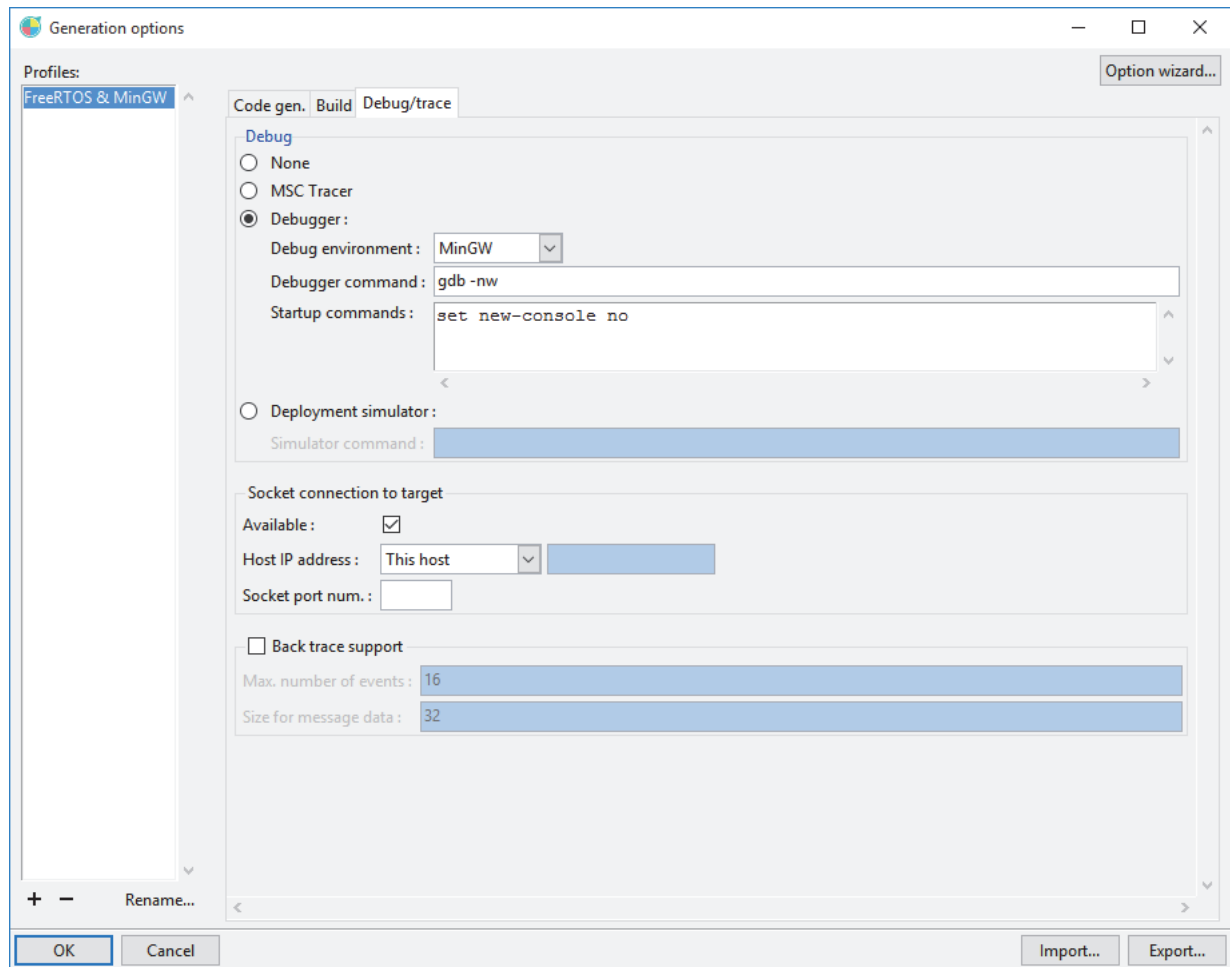
The image displays two screenshots of the 'Generation options' dialog box, showing different tabs and settings.

Top Screenshot: Code gen. tab

- Profiles:** FreeRTOS & MinGW
- Code gen. tab:**
 - General options:**
 - Destination directory: ccg
 - Code templates dir.: \${RTDS_HOME}\share\ccg\freertos
 - First signal num.: 1
 - SDL-RT/SDL specific options:**
 - Language: C
 - Gen. code coverage info: ☐
 - Generate environment process: ☒
 - Communicate with env. via macros: ☒
 - TTCN specific options:**
 - Generate: TTCN + SDL/SDL-R
 - Generate main function: ☒

Bottom Screenshot: Build tab

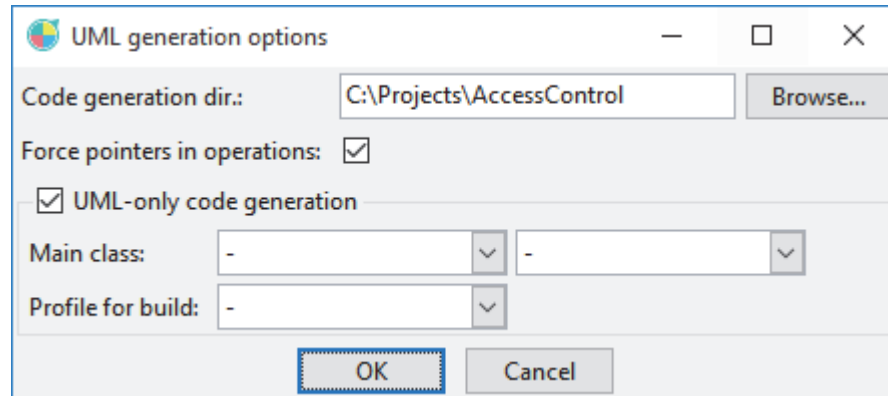
- Profiles:** FreeRTOS & MinGW
- Build tab:**
 - ☒ **Generate makefile**
 - Command :**
 - Preprocessor :** [Empty field]
 - Compiler :** mingw32-gcc.exe
 - Linker :** mingw32-gcc.exe
 - Options :**
 - Options... (for Preprocessor)
 - Options... (for Compiler)
 - Options... (for Linker)
 - Additional files to link :** [Empty field]
 - Include external makefile :** ☒ \${RTDS_HOME}\share\ccg\freertos\make\FreeRtosMake.inc
 - Do build :** ☒
 - Before build command :** [Empty field]
 - Build command :** mingw32-make
 - Target :** all
 - Add'l args :** [Empty field]
 - After build command :** [Empty field]



Typical generation profile to debug a FreeRTOS application with gdb

8.3.3 UML options

There are specific options used when generating the C++ code for classes described in UML class diagrams. These options are set via the “UML options...” in the “Generation” menu:



The available options are:

- The directory where the files for the UML classes must be generated;
- Whether parameters and return values in operations should be forced as pointers when their type is a user-defined class;
- Options for UML-only projects:
 - The main class for the application. This class will be automatically instantiated when the application starts. The equivalent of the C/C++ main function should be written in this class's constructor.
 - The profile used for generating the makefile and doing the build for the project.

For projects mixing SDL and UML, the generation directory for the passive classes should be different from the generation directory set in the profile for the following reasons:

- The generation directory set in the profile will contain the generated source files for all processes and blocks in the system. These files are just a result of the code generation: the real source code for the system is in the diagrams. So all files in this directory may be safely deleted;
- On the contrary, the class diagrams only describe the interface of the classes. The actual code implementing the operations are in the C++ source file. So this file must not be deleted, or actual code will be lost.

For this reason, PragmaDev Studio uses a different policy for the two generation directories:

- In the generation directory set in the profile, if any generated source file needs updating because of changes in the corresponding diagram, it is always overwritten;
- In the UML generation directory, if a change was made in any diagram describing a class, the header file for the class is overwritten, but *not* the C++ file for the class, as it may contain actual code for the operations.

However, added operations in the diagram will be added to the C++ source file, and warnings will be issued for each operation found in the file that is not in the diagrams. But nothing will ever be removed from the C++ source file.

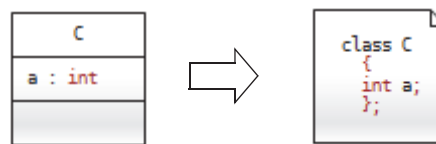
8.3.4 Generated C++ code

8.3.4.1 Attributes and operations

For any class involved in a code generation process, the following attributes and operations are generated in the class header file:

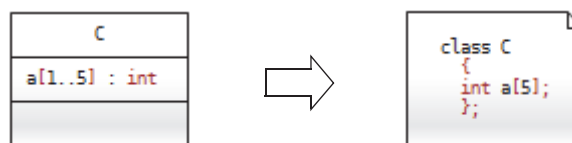
- For an attribute described as “*name* : *type*” (no multiplicity), an attribute with this name and type is generated.

Example:



- For an attribute described as “*name* [*mult*] : *type*” with a finite multiplicity (no ‘*’), an attribute with this name and the type “*type* [*max*]” is generated, with *max* being the maximum multiplicity found in *mult*.

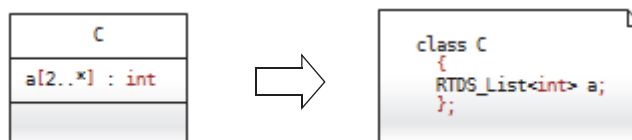
Example:



Maximum multiplicity for attribute a is 5 =>
array of 5 ints

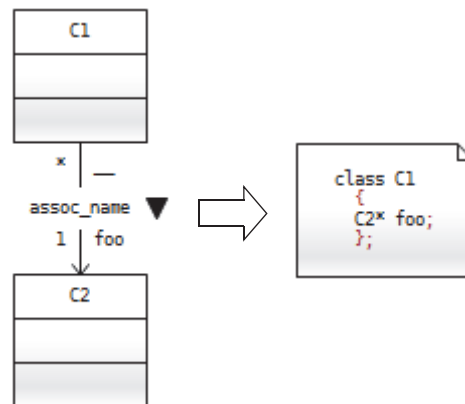
- For an attribute described as “*name* [*mult*] : *type*” with an infinite multiplicity (‘*’ somewhere in *mult*), an attribute with this name and the type `RTDS_List<type>` is generated (see notes below on page 206).

Example:



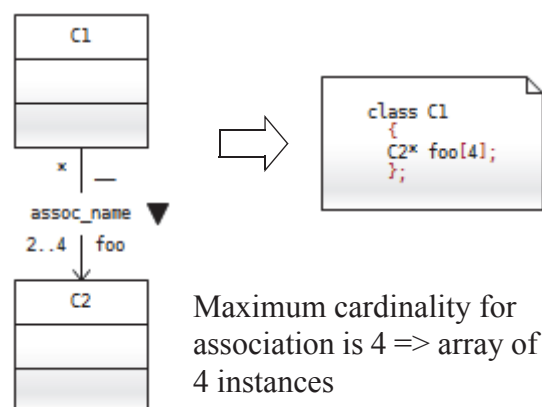
- For a navigable association to class *Class* with a cardinality of 1 or 0..1 and a role name *role*, an attribute named *role* with type *Class** is generated.

Example:



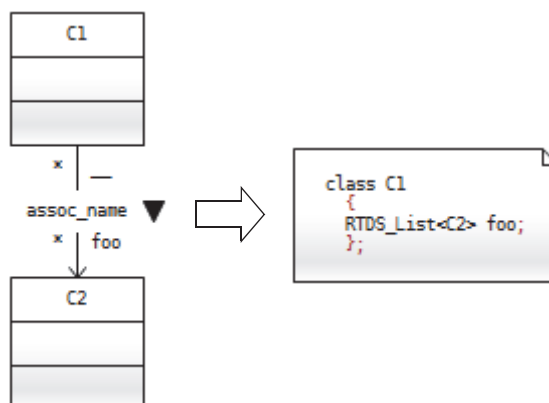
- For a navigable association to class *Class* with a multiple finite cardinality (not 1, not 0..1 and no '*' in cardinality) and a role name *role*, an attribute named *role* with type *Class*[max]* is generated, where *max* is the maximum cardinality for the association.

Example:



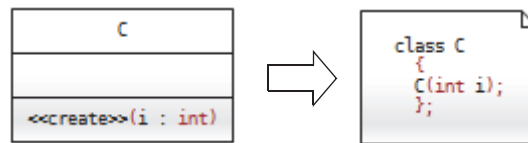
- For a navigable association to class *Class* with an infinite cardinality ('*' somewhere in cardinality) and a role name *role*, an attribute named *role* with type *RTDS_List<Class>* is generated (see notes below on page 206).

Example:



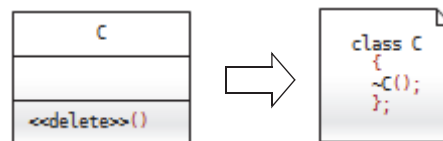
- For an operation named <<create>>, the corresponding C++ constructor is generated with the same parameters and no return type.

Example:



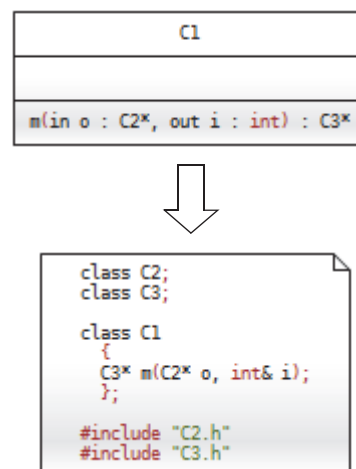
- For an operation named <<delete>>, the corresponding C++ destructor is generated with no return type. This operation must not have any parameters.

Example:



- For any other operation, the corresponding method is generated. A parameter declared as "out name : type" or "inout name : type" is generated with type `type&`. If a parameter type is an instance of a known class, it is recognized and the corresponding declaration is included in the class header file. The same applies for the operation's return type.

Example:



If no C++ file for the class exists when the code generation is run, a basic C++ file containing a skeleton for the implementation of all known methods will also be generated.

Notes:

- RTDS_List is a template class to manage lists of pointers. It is delivered with PragmaDev Studio and located in directory \$RTDS_HOME/share/ccg/cpp-templates. The parameter for the template is the type for the elements (e.g., RTDS_List<int> is a list of pointers to integers).

Its public methods are:

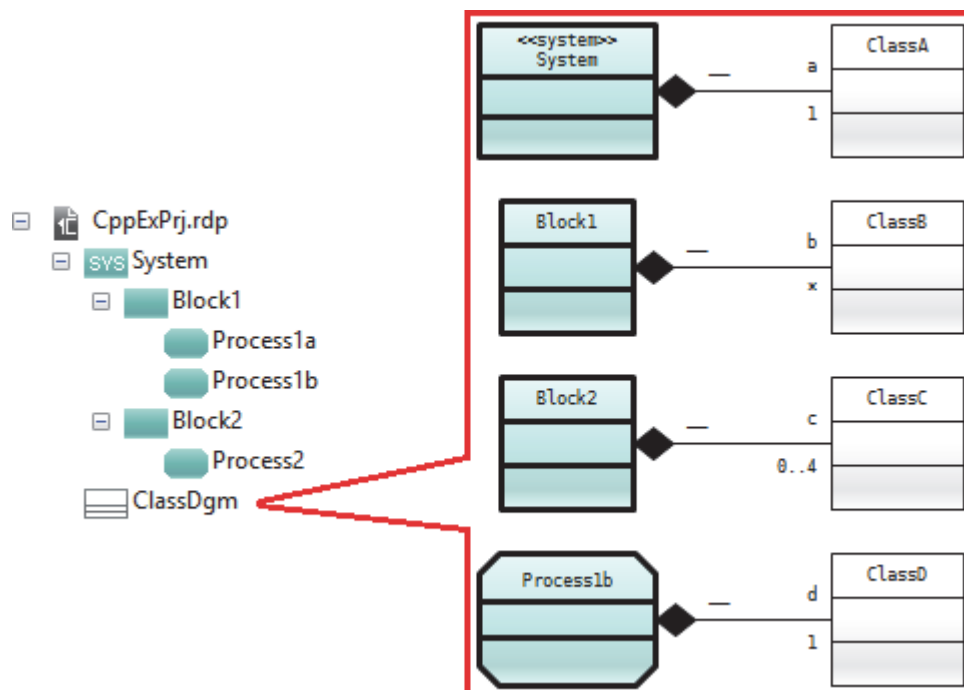
- append to add a new element at the end of the list. The only argument is a pointer to the element to add;

- del to remove an element from the list. The argument is the index of the element to delete;
- length returns the length of the list as an integer;
- the operator [] is also redefined to give access to any element in the list by its index. The element can be accessed for reading (eltPtr = list[index];) and writing (list[index] = eltPtr;).
- No attribute should be declared with a type being a class defined in a diagram. If it is, the class won't be recognized and the class will be undeclared when it is used. To define such an attribute, it is mandatory to use an association.
- For attributes generated for associations, if the role name is not set or invalid, a modified version of the association's name will be used.

8.3.4.2 Declared variables

For any class involved in an association with a block or a process, the corresponding instances will be known to all elements in the block or process sub-tree.

For example:



The following variables are known:

- A variable named a with type ClassA* is known in all blocks and processes, since it's attached to the system itself;
- A variable named b with type RTDS_List<ClassB> is known in Block1 and all its descendants, i.e. Process1a and Process1b;
- A variable named c with type ClassC*[4] is known in Block2 and all its descendants, i.e. Process2;
- A variable named d with type ClassD* is known only in Process1b.

This is achieved by generating the following files:

- For all associations attached to a system or block, a C file is generated for the system or block. It contains the declaration of all the variables as global. An extern declaration is generated in the system or block's header file, which is included in all descendants.

In the example, variables a, b and c are declared as global, with their extern declarations inserted in System.h, Block1.h and Block2.h respectively.

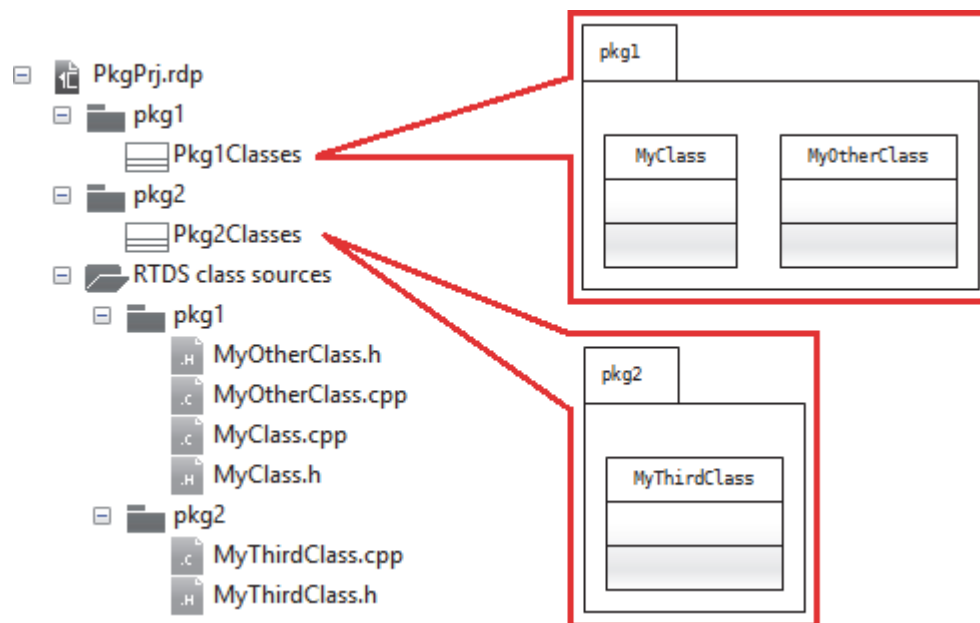
- For all associations attached to a process, no additional file is generated. The variable is automatically declared as local in the function generated for the process.

In the example, the variable d is declared as local to the function generated for process Process1b in Process1b.c.

8.3.4.3 Access to generated code

After a code generation, all generated code is made available in the project:

- The code for all processes and blocks is inserted in a package named "RTDS generated code";
- The code for all classes is inserted in a package named "RTDS class sources". The structure of this package is mapped to the package structure for classes. For example:



In "RTDS class sources":

- MyClass.h, MyClass.cpp, MyOtherClass.h and MyOtherClass.cpp are in sub-package pkg1 because classes MyClass and MyOtherClass are in package pkg1 in the diagrams.
- MyThirdClass.h and MyThirdClass.cpp are in sub-package pkg2 because class MyThirdClass is in package pkg2 in the diagrams.

8.3.5 Built in scheduler

The code generation described in “Code generation” on page 155 maps by default each SDL or SDL-RT process instance to a task in the target RTOS. PragmaDev Studio also offers the possibility to execute several process instances in a single RTOS task. This allows for example to execute a whole block in a single task. If the whole system is executed in a single task, this even allows to execute it on a bare target without any RTOS. This feature is available for both SDL and SDL-RT projects.

When several processes are executed in a single task, no actual parallelism is involved: the instances are scheduled within the task, and transitions will be executed one at a time.

To turn process instance scheduling on, the following steps are required:

- Define which process instances will be scheduled and which ones will have their own RTOS task. This is done via a UML deployment diagram.
- Define a code generation profile that allows scheduling.
- Make sure the system is compatible with the scheduled mode. This mainly involves taking care about how semaphores are used in SDL-RT systems.
- For wholly scheduled systems without any RTOS, make sure messages coming from the environment and time will be handled correctly. This involves writing some external code.

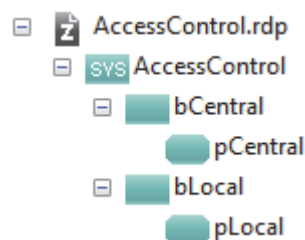
Generating scheduled code also allows integration in an external scheduler.

The following paragraphs describe these points in detail.

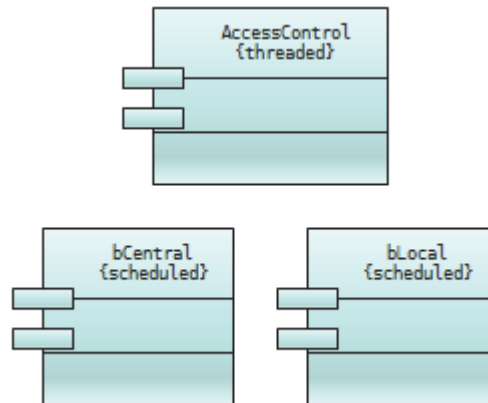
8.3.5.1 Deployment diagram for scheduling policy

To indicate which processes will have their instances scheduled and which will map to their own task, a UML deployment diagram has to be defined. In this diagram, agents will be represented as components, and the component properties will indicate if the agent is scheduled or threaded. The code generation should then be run on this diagram to actually turn scheduling on in the generated code.

For example, for the following system:

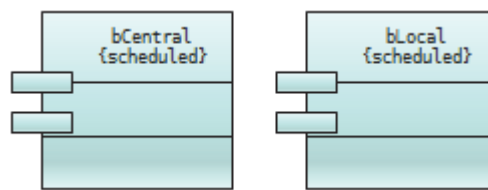


the components in the deployment diagram may be:

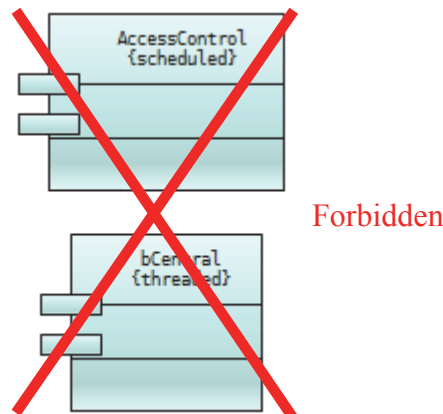


This means that all process instances within blocks `bCentral` will be scheduled together, as well as all process instances within `bLocal`. As the whole system is “threaded”, a RTOS task will be created for each block `bCentral` and `bLocal`.

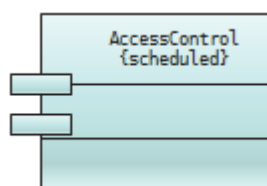
The default policy for agents is actually “threaded”, so the diagram above is in fact equivalent to this one:



Of course, if an agent is scheduled, all the agents it contains will be “scheduled” too, so the following deployment diagram is illegal:

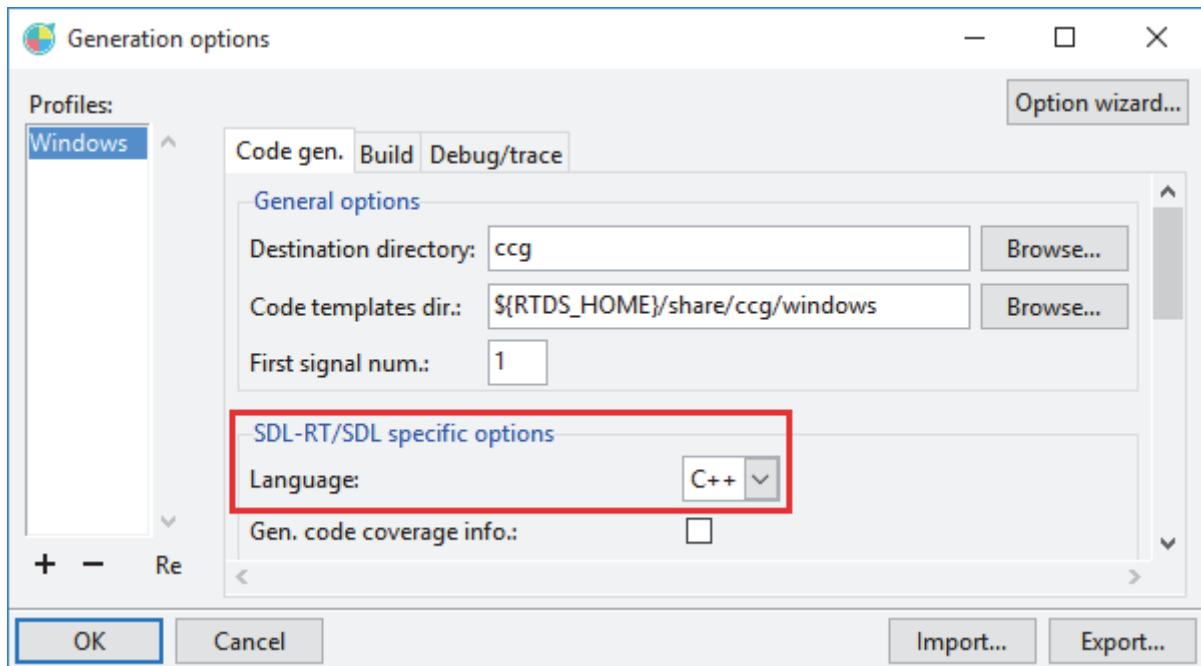


To schedule the whole system, allowing it to be executed without any RTOS, only the following component is necessary:



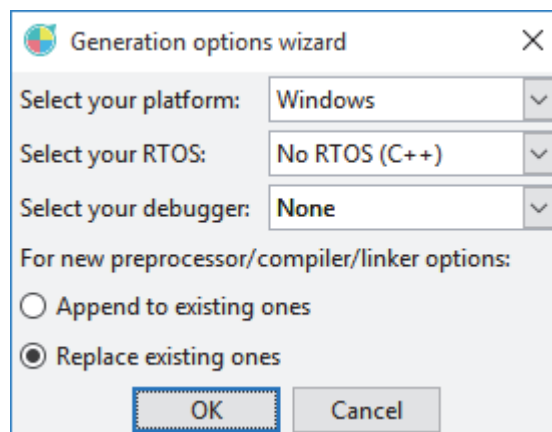
8.3.5.2 Profiles for scheduling

There is only one limitation on code generation profiles to make them compatible with scheduling: in SDL-RT projects, the profile must be set to generate C++ code in the “Code gen.” tab, option “Language”:



This limitation does not exist in SDL projects. However, if generating code for the C scheduler, object-oriented features such as process classes are not supported.

If the whole system is scheduled, a specific target can be used to generate code for a bare target without any RTOS. This can be chosen via the “Option wizard...” in the code generation options dialog:



Setting the RTOS to “No RTOS” will use a specific integration in <PragmaDev Studio installation dir.>/share/ccg/rtosless that does not require any service from the OS, except dynamic memory allocation.

When a code generation is run on a deployment diagram specifying scheduled agents, the files for the built-in scheduler will be automatically integrated in the generation and

will appear in the “RTDS RTOS adaptation” folder. These files are not generated but included in the following directories:

- <PragmaDev Studio installation dir.>/share/ccg/cppscheduler for files used in C++ code generation;
- <PragmaDev Studio installation dir.>/share/ccg/cscheduler for files used in a C code generation (SDL projects only).

Generated code for scheduled processes will be very different from the generated code for threaded ones: each transition will be generated in its own function or method, allowing to call it from the scheduler itself. For more details, see PragmaDev Studio Reference Manual.

8.3.5.3 Semaphore handling

In the context of a scheduler, everything that might interrupt a transition in the middle of its execution needs a special attention. In SDL, only procedure calls may interrupt the execution of a transition if the called procedure contains state changes. This case is handled in the generated code in a specific way (see PragmaDev Studio Reference Manual).

In SDL-RT, in addition to procedure calls, transition execution may also be interrupted by semaphore takes. This case is much more difficult as several case may occur depending on the instance that takes the semaphore and the instance that already has already taken it:

- If each instance has its own task, the case can be handled as in normal code generation, i.e., by using the semaphores provided by the RTOS;
- If the whole system is scheduled, it is possible to provide a specific implementation for semaphores that will handle the case (see PragmaDev Studio Reference Manual);
- In all other cases, the problem is very difficult to solve. For example, if the instance holding the semaphore is in a task, but the instance taking it is scheduled with other instances in another task, a semaphore provided by the RTOS has to be used since there are two different tasks, but it will block not only the instance trying to take it, but all other instances in its task. A mutex semaphore may also be very difficult to handle if it can be taken by instances scheduled in the same task or by instances in different tasks in the same system.

The solution chosen in is quite simple: except if the whole system is scheduled, PragmaDev Studio will always use semaphores provided by the underlying RTOS. This means a system designed to work in threaded mode *may not work* if it's switched to partially scheduled mode if it handles semaphores. The architecture of the system and/or the way semaphores are used may have to be changed to get the system to work.

8.3.5.4 External messages and time management

Whenever an RTOS is used, external messages and time management are usually not a problem:

- External incoming messages are usually handled via interrupts, calling a routine that will build the message and put it in a message queue. Then the system resumes its normal execution.

- Timers are handled by system calls.

The case where the whole system is scheduled and no RTOS is used is more complicated to handle: external messages can be handled via interrupts, but the instance expecting them can't be simply waiting on a message queue and woken up automatically when it arrives. The instance will be scheduled and there must be a way to inform the scheduler that a message has arrived. As for timers, since there is no RTOS, there is available system call to handle time.

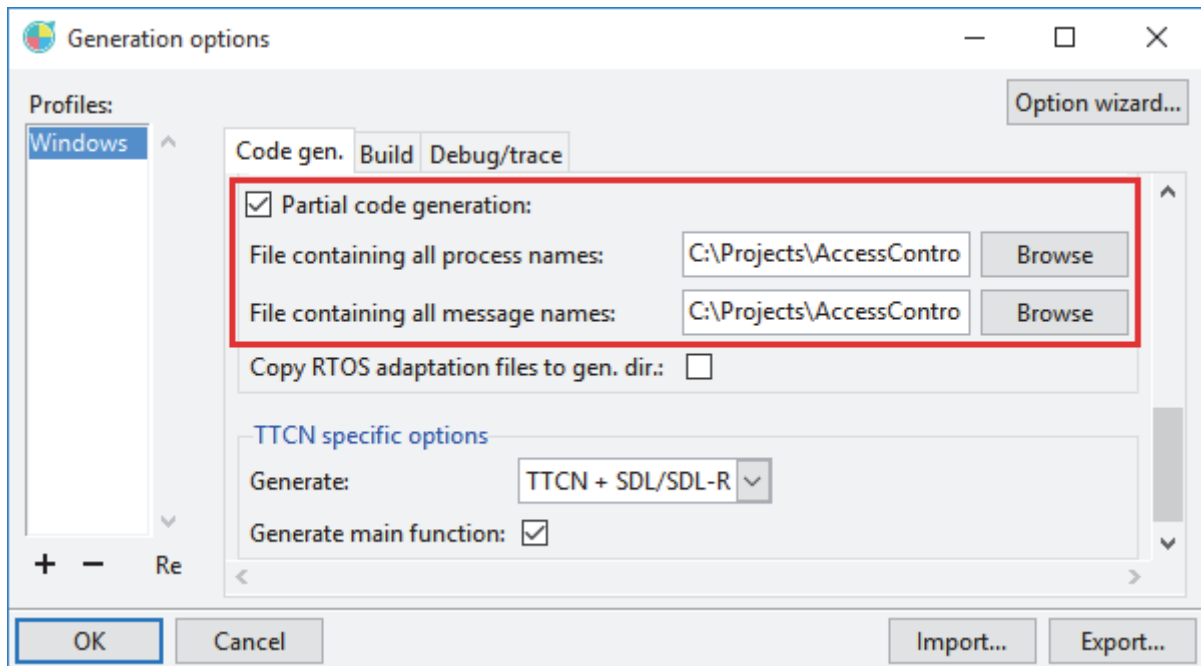
In this case, the handling of time is done via 2 specific functions, called `RTDS_incomingSdlEvent` and `RTDS_SystemTick`:

- `RTDS_incomingSdlEvent` is called automatically by the scheduler whenever there is no internal message to handle. It should be written by the user. Its parameters are a pre-allocated message header structure for the received message if any, and the time left until the next timer should fire. This function should wait for an incoming message, at most for the time passed as parameter, and then return to the caller. It returns a boolean which should be true if a message has been received, or false if the reception timed-out.
- `RTDS_SystemTick` increases the system time by one tick and updates the list of running timers to decrease the delay before they should time-out. This routine is provided by PragmaDev Studio, but should be called by user-code, for example on a cyclic interrupt used to handle time.

A default implementation for `RTDS_incomingSdlEvent` is provided by PragmaDev Studio if the macro `RTDS_HANDLE_EXTERNAL_EVENTS` is defined in the compilation options. This default implementation just calls `RTDS_SystemTick` and returns false to indicate no external message has arrived. This default implementation just ensures that systems will work in the PragmaDev Studio Debugger without writing any additional code. It should obviously not be used in real systems.

8.3.5.5 Integration in external scheduler

It is possible to generate code for a set of processes that will be used in an external scheduler, different from the PragmaDev Studio built-in one. To do this, a special code generation profile should be created with the option "Partial code generation" turned on (in the "Code gen." tab):



If this option is turned on, the generated code will basically contain only the source and header files generated for the processes themselves, with only a few additional global header files. No entry point and no makefile will be generated. Note that this feature is only available when the target language is C. Partial code generation in C++ is not supported yet.

Since the code generator needs to know the existing processes and the existing messages to generate some global constants and types correctly, two files must be specified when partial code generation is on:

- The *File containing all process names* entry should reference a file containing the names for all processes that will be integrated in the final build. The names should be specified one per line and are case-sensitive. This information is used to generate the structures holding the instances local variables.
- The *File containing all message names* entry should reference a file containing the names of all incoming and outgoing messages for all processes. The names should also be specified one per line and are also case-sensitive. This information is used to generate the transport structures for messages and the macros handling them.

Please note these two files must only include processes and messages handled in PragmaDev Studio. If another process or message is inserted here, the generated code may use a type or constant that won't be defined and may fail to compile.

More details about the generated code and how it can be integrated in an external scheduler is given in PragmaDev Studio Reference Manual. An example is also available in PragmaDev Studio distribution, showing how results of different partial code generations can be integrated together. This example is located in <PragmaDev Studio installation dir.>/examples/Studio/Advanced/PartialCodeGen.

8.3.6 Deployment simulator

PragmaDev Studio offers the possibility to simulate the deployment of process instances in a distributed infrastructure. The *ns-3 network simulator* is used to setup the underlying communication infrastructure on top of which the SDL (or SDL-RT) process instances will execute. Due to the nature of ns-3 simulations (i.e., single task discrete event simulation), it is required for all process instances to execute in the same task with the ns-3 simulator. This implies that the system must be fully scheduled so that no parallelism is involved as described in “Built in scheduler” on page 209.

To enable deployment simulation, in addition to the system being fully scheduled, the following are required:

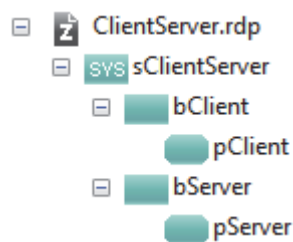
- Define how process instances will be deployed via a UML deployment diagram.
- Define a code generation profile that requires scheduling and supports deployment simulation.
- Make sure external messages coming from the environment are handled correctly.

The following paragraphs describe these points in detail.

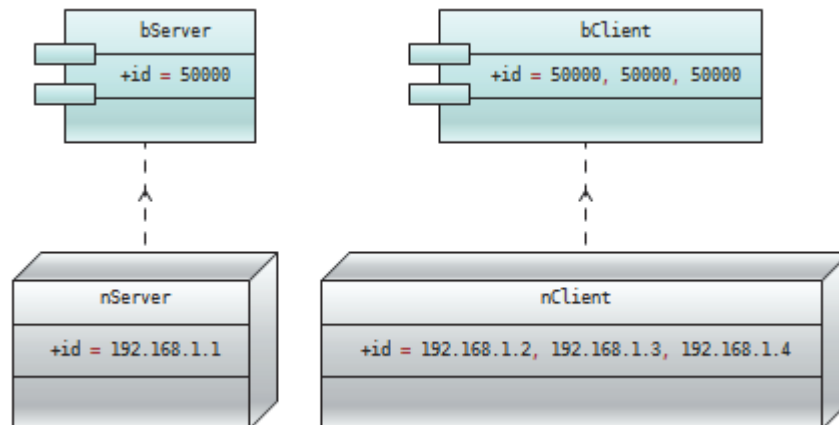
8.3.6.1 Deployment diagram for simulation

A UML deployment diagram has to be defined to indicate how processes will be deployed for simulation. Agents are represented as components which must be linked to at least one node via a dependency relation. The nodes in the diagram represent ns-3 objects that interface agents with the underlying communication infrastructure provided by ns-3. The code generation should then be run on this diagram to actually generate a simulation executable.

For example, for the following system:

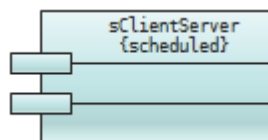


the components and nodes in the deployment diagram may be:



This means that all process instances within the block bServer will use the node nServer to get access to ns-3 communication layers. The same is true also for bClient and nClient. To successfully communicate between them using ns-3, each agent has to be uniquely identified in the diagram. This is done via the `id` attribute in both component and node. Values for this attribute are the IP address for the node and the TCP port for the component. This implies that a pair (address, port) has to be unique in the diagram. A comma separated list of values can be assigned to the attribute. In this case the number of values for the node and attached components must match. The use of lists simplifies the definition of large scale deployments (e.g., hundreds or thousands of nodes and components) and maintains the overall readability of the diagram.

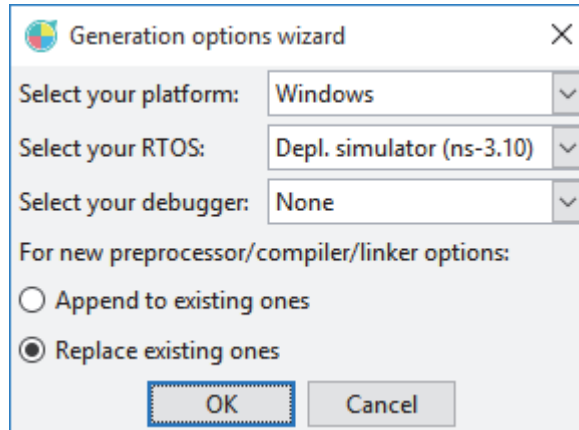
As deployment simulation requires always a fully scheduled system, by default each deployment diagram used for simulation implicitly includes a system agent component with the “scheduled” property. So, even if it is not present, the diagram above is considered to contain:



This automatically fulfills the requirement for such component as described in “Deployment diagram for scheduling policy” on page 209, thus it can be safely omitted (unless it is necessary, i.e., must be attached to a node). The same is true also for the property of every other component in the diagram; the scheduling policy for all agents is set by default to *scheduled* and any attempt to change it will be silently ignored by the code generator.

8.3.6.2 Profiles for deployment simulation

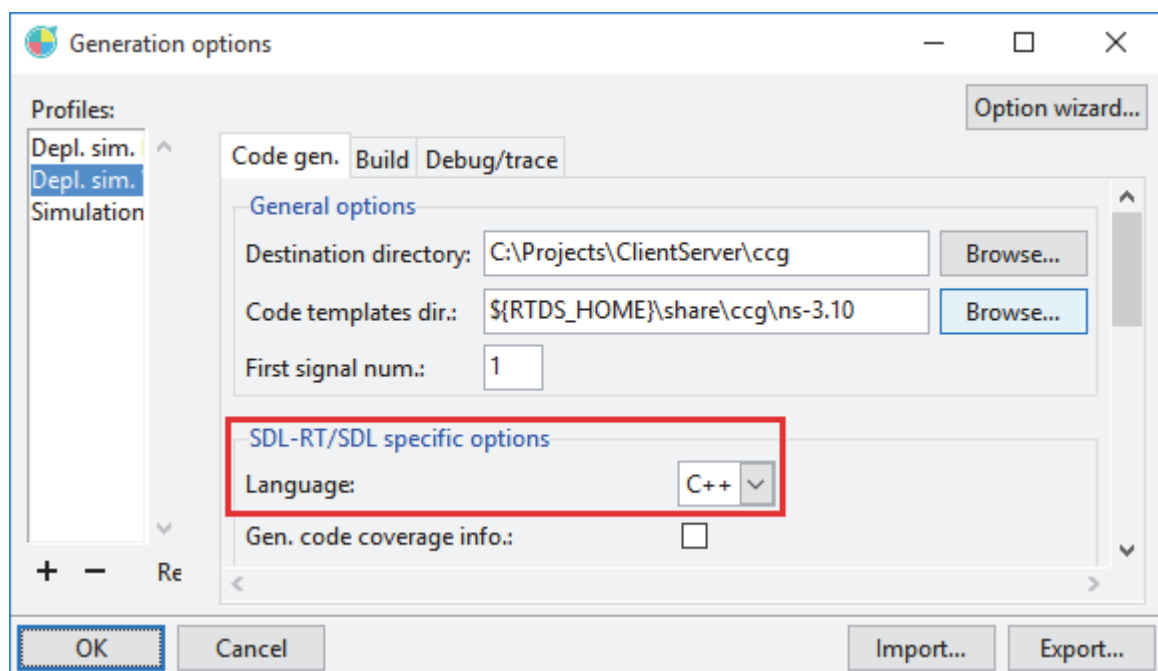
Code generation for deployment simulation is available on Linux and Windows using a specific target. This can be chosen in the “Option wizard” in the code generation options dialog:



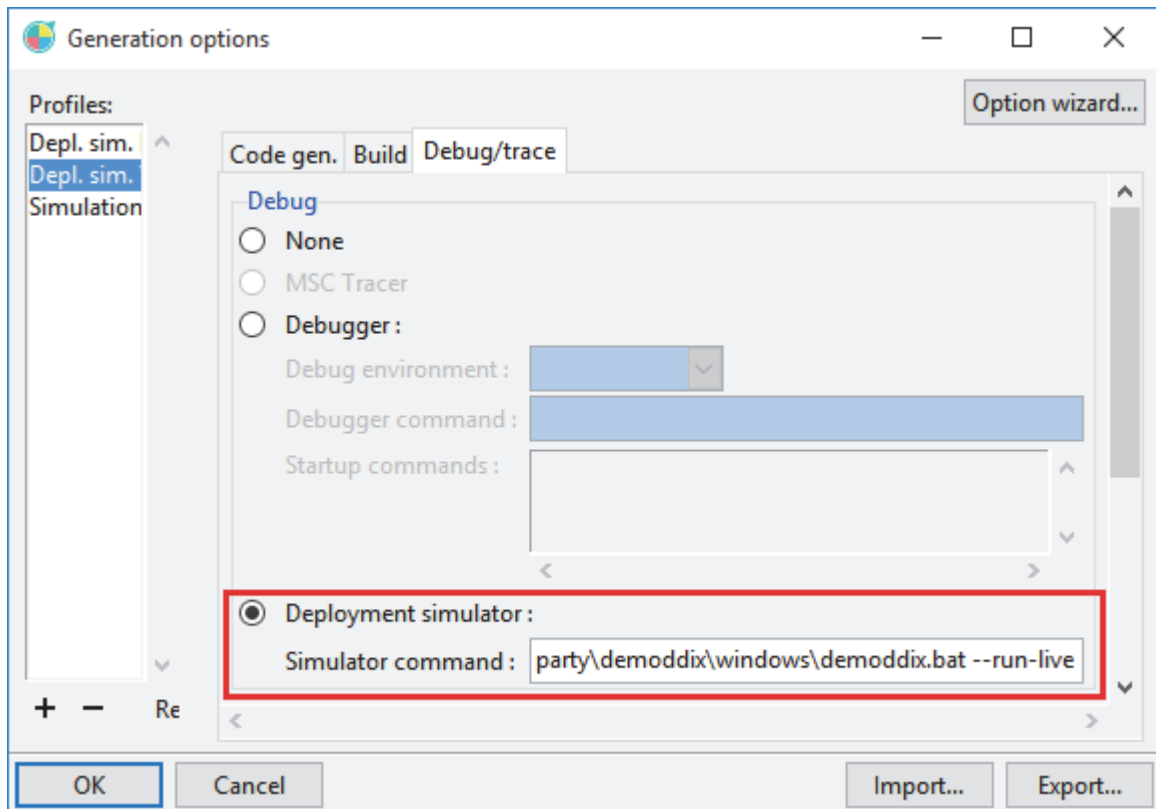
Setting the RTOS to “Depl. simulator (ns-3.10)” will use a specific integration in <PragmaDev Studio installation dir.>/share/ccg/ns-3.10 that contains all the files needed for the build. The header files and prebuilt libraries of ns-3 can be found in <PragmaDev Studio installation dir.>/share/3rdparty/ns-3.10/pragmadev.

In general, in addition to the system being fully scheduled, there are two additional requirements for code generation profiles to make them compatible with the deployment simulator:

- The profile must be set to generate C++ code in the “Code gen.” tab:



- The debugger must be set to “Deployment simulator” in the “Debug/trace” tab:

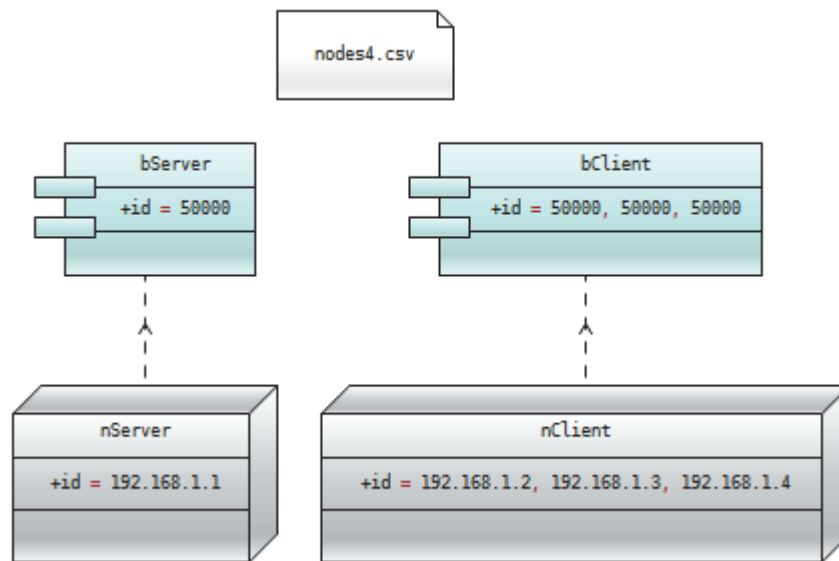


The generated code for deployment simulation will be similar to that of fully scheduled systems, however there are some important differences that ensure correct integration with the ns-3 simulator. For more details see the PragmaDev Studio Reference Manual.

8.3.6.3 External messages

Handling of external messages in deployment simulation is quite different compared to a RTOS or even scheduling. This is due to the nature of ns-3 simulations in general. Every event (e.g., message) that is external to the simulator, i.e., that is not registered to or generated by the ns-3 scheduler during execution, has to be made available to the scheduler before running the simulation. This implies that the information about external messages should be available before generating the code for deployment simulation. Infor-

mation about external messages can be included in the UML deployment diagram via a file symbol:



PragmaDev Studio expects a CSV file format, where each row is on its own line and the cells separated by a semicolon ‘;’. The contents can be created within a spreadsheet application. Here is an example of external messages created in LibreOffice Calc:


	A	B	C	D
1	TIME	192.168.1.2:50000	192.168.1.3:50000	192.168.1.4:50000
2	1000	mStart(10)		
3	2000		mStart(20)	
4	3000			mStart(30)
5				

The first row is considered as the title row; its first cell must be “TIME” (case sensitive), and all other cells may contain the unique identifiers of the components in the UML deployment diagram. The identifiers are formed by the `id` attribute of the node and component, separated by a colon ‘:’ (in the form “address:port”).

The first cell of every other row should contain the time in milliseconds at which the messages appearing in that row will be scheduled (by the ns-3 scheduler) to be received by the corresponding agent in the component. Time values in rows must be unique and in ascending order.

All remaining cells may contain external messages with parameters (if any). The format must adhere to that shown in the table above, i.e., message name followed by a comma-separated list of parameters in parenthesis. The parenthesis are required even if the message expects no parameters. Only integer values are accepted as message parameters.

8.3.6.4 The deployment simulator

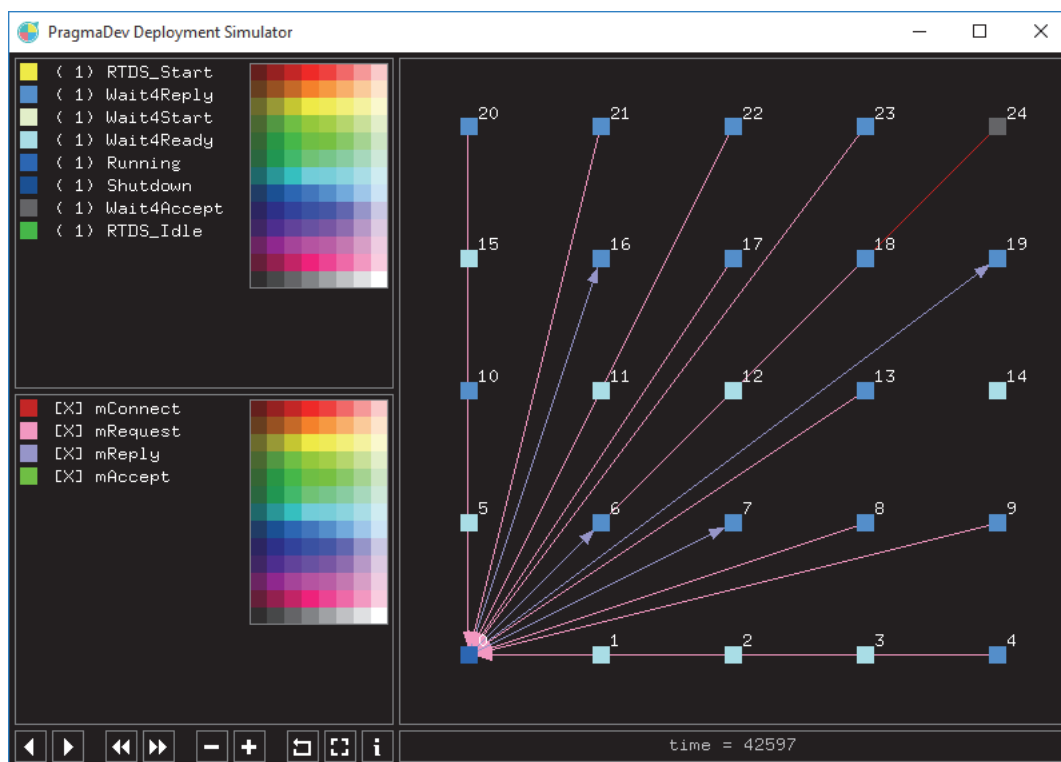
The deployment simulator is started from the “Generation / Execute” menu or the  quick button. Once a profile for deployment simulation is selected, PragmaDev Studio will:

- check syntax and semantic of the deployment diagram (and CSV file if any),
- check syntax and semantic of the SDL-RT (or SDL) system,
- generate the C++ code, compile and link it producing an executable, and
- start the *Deployment Simulator* which will run the executable and trace events.

If a CSV file is attached to the diagram and edited with an external spreadsheet application, it is recommended to launch the simulation via “Generation / Execute / Force code generation...”. This is necessary in cases where only the CSV file has been edited.

8.3.6.4.1 Live tracing

At start, the deployment simulator will launch the generated executable in the background and display the nodes, state-changes, message send and receives, and time during simulation:



The time is displayed at the bottom (progress bar) and updated with the timestamp of the most recent traced event. A pulsing effect is applied on the displayed time to show that the simulation is still running (or that the tool has not finished yet reading the traced events).

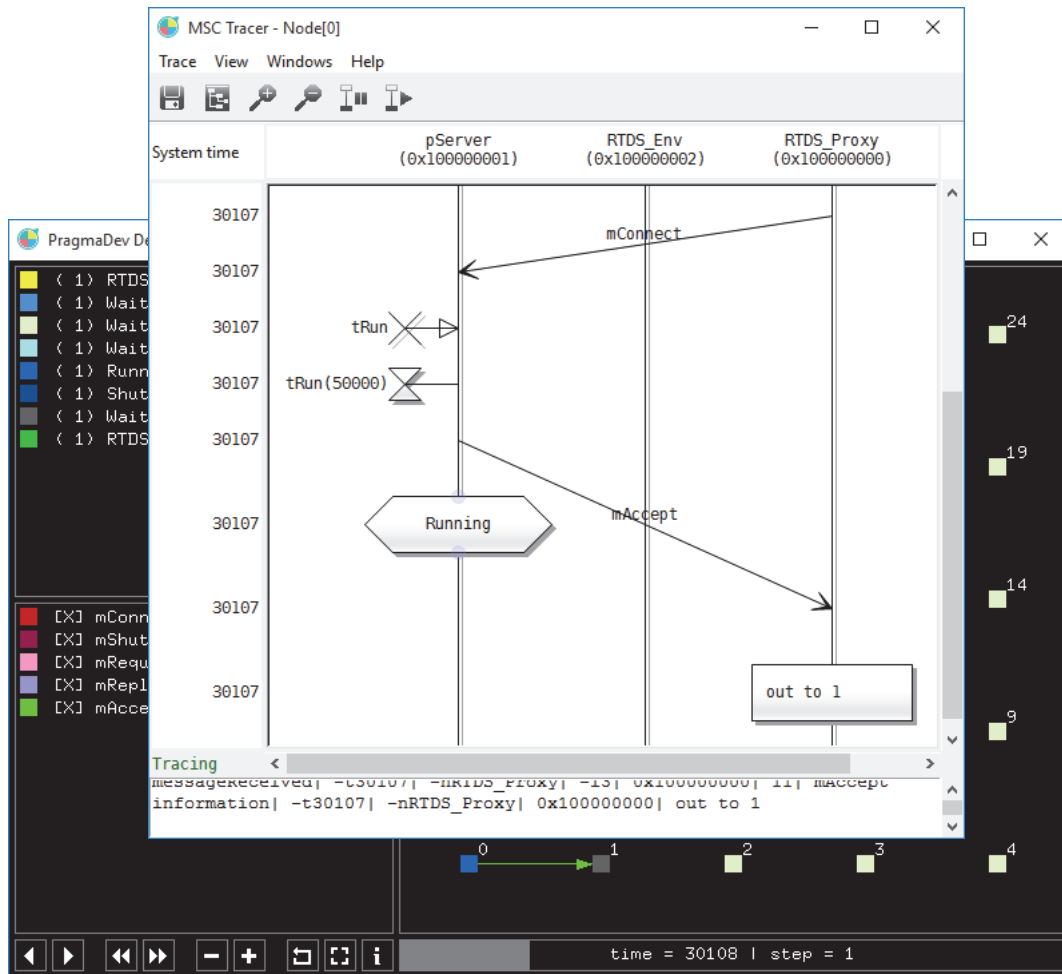
All nodes are displayed by default in a grid-like topology with an identifier assigned to them automatically. The color of each node represents the current state. This is the most recent state-change triggered by a process instance running on the node.

The list of states is displayed in the upper-left. Each state has a color and a priority assigned to it:

- The color of the state can be changed by a *Left-Click* on a state and then on the desired color in the color chooser. To cancel a color change, a *Right-Click* on the color chooser is sufficient.
- The priority of the state can be used to assign a precedence to it during visualization. A state-change will be displayed if the priority assigned to the new state is greater than or equal to the priority of the current one. A value of 0 (zero) means that the state-change will not be displayed, regardless of the priority of the current state. The priority can be changed by holding down the *Ctrl* key and *Left-* or *Right-Click*-ing on the state.

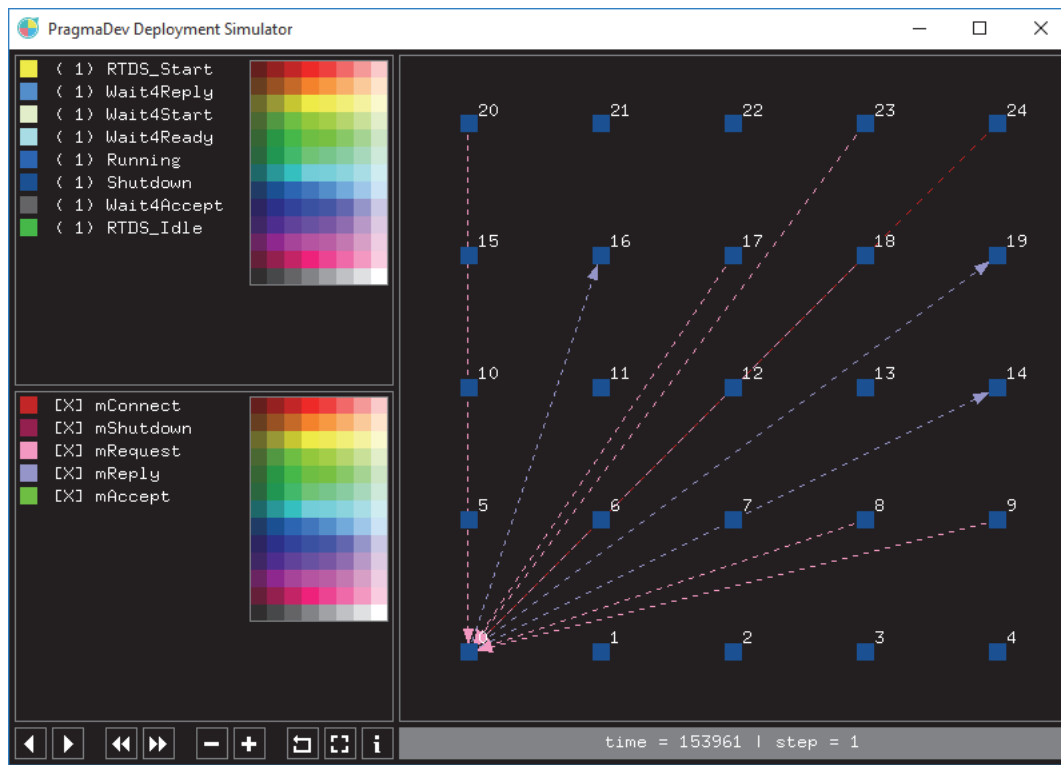
Messages are displayed with an arrow from the sender to the receiver. The color represents the type of the message. The message is removed from the view the moment it is received. Lost messages are displayed with a dotted line, however the dotted pattern will be visible only at the end of the simulation. The list of messages is displayed in the bottom-left. Each message has a color and visibility assigned to it. If visibility is enabled, the message will be displayed during simulation, otherwise it will be hidden. The color and visibility of messages can be changed in the same way as states.

Holding down the *Crtl* key and *Left-Click*-ing on a node will launch an instance of the PragmaDev Tracer for displaying the events of all process instances running on the node.










8.3.6.4.2 Post-mortem tracing

After successful termination of the simulation, the progress bar will advance to the end, and the pulsing effect on the time will stop:



However, if simulation fails for some reason, an error message will be displayed at the bottom beside the time.










On success, a set of additional functionalities will become available. These allow navigation into all traced events in a post-mortem fashion:

-  (or 'R' key) resets simulation (traced events) at its starting point. This sets all nodes to their initial state, no messages are displayed, and the time is set to zero.
-  and  (or *Down-Arrow* and *Up-Arrow* keys) allow navigation one event back or forward accordingly. The event can be a node state-change, message sent, or message received.
-  and  (or *Left-Arrow* and *Right-Arrow* keys) allow back and forward navigation in time steps (in milliseconds). The size of the step can be changed using  and  (or *Page-Down* and *Page-Up* keys).

Displaying of events is the same as in live tracing with only exception of lost messages. In post-mortem tracing such messages are also displayed with a dotted line, but the dotted pattern is visible immediately, i.e., at the moment the message is sent.

8.3.6.4.3 Commands summary

The complete list of commands of the deployment simulator is given as follows:

- Color Chooser
 - *Left-Click*: change color and close.
 - *Right-Click*: cancel change and close.
- States
 - *Left-Click*: open color chooser.
 - *Ctrl + Left-Click*: increase priority by 1.
 - *Ctrl + Right-Click*: decrease priority by 1.
 - *Mouse-Wheel*: scroll through the list of states.
- Messages
 - *Left-Click*: open color chooser.
 - *Ctrl + Left-Click*: enable / disable visibility.
 - *Mouse-Wheel*: scroll through the list of messages.
- Nodes
 -  or 'I' key: show / hide additional information.
 - *Ctrl + Left-Click*: launch PragmaDev Tracer.
 - *Mouse-Move* while holding *Left-Click*: pan view.
 - *Mouse-Wheel*: zoom in / out view.
 -  or *Home* key: reset pan and zoom.
- Events
 -  or 'R' key: reset.
 -  /  or *Down-Arrow* / *Up-Arrow* key: back / forward one event.
 -  /  or *Left-Arrow* / *Right-Arrow* key: back / forward one time step.
 -  /  or *Page-Up* / *Page-Down* key: decrease / increase time step.

8.3.6.5 Simulation modes

The PragmaDev Deployment Simulator is provided as a standalone tool in <PragmaDev Studio installation dir.>/share/3rdparty/demoddix.

The tool supports two modes of execution:

- *Live* (or pseudo-live) mode allows visualization of events as they are generated by the ns-3 simulation. In this mode, the simulation executable generated by PragmaDev Studio is passed as an argument to the tool, which will launch it in a separate thread of execution and read and display the XML traces as they are produced. This mode has two variants, that can be enabled with the following options:
 - *--run* will read and display all available traced events at once,
 - *--run-live* will read and display traced events one at a time.
- *Post-mortem* mode allows visualization of events from a pre-generated trace file. Every deployment simulation generated by PragmaDev Studio will produce such file during execution. In this mode the trace file (and not the executable) is passed as an argument using one of the options:
 - *--trace* will read all traced events in the file at once,

- `--trace-live` will read and display traced events one at a time.

These modes define if and how the traced events should be displayed while loading them. Once all events are loaded, the navigation functionality will be enabled allowing replay of the simulation.

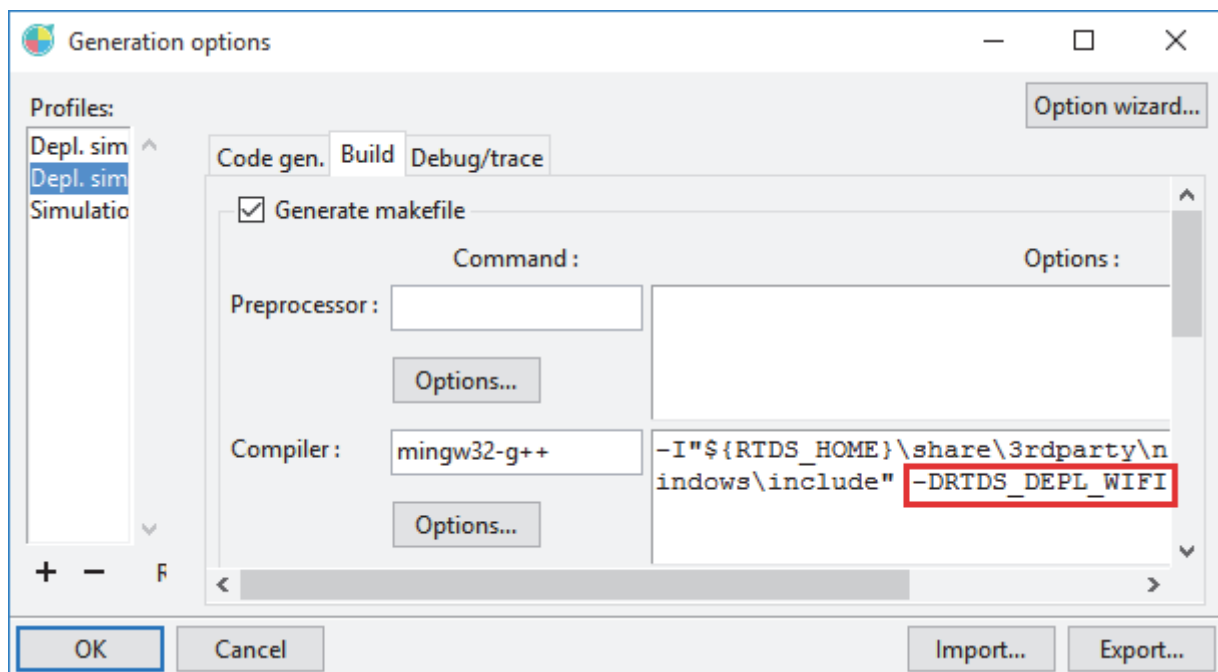
If the tool is launched from PragmaDev Studio, it should always be in live mode (`--run` or `--run-live`). This will ensure the execution and display of the generated simulation and not some other trace file that may have been produced during previous simulations.

8.3.6.6 ns-3 configuration

The underlying communication infrastructure, on top of which the SDL or SDL-RT process instances execute, is provided by ns-3. This infrastructure is composed of:

- Topology - position of nodes (coordinates).
- Network devices and channels - devices are attached to channels to create a communication medium that allows nodes to exchange messages.
- Protocol stack - set of communication protocols (e.g., TCP, IP, etc.)

The code generated by PragmaDev Studio provides a default configuration for the infrastructure: nodes are positioned in a grid topology, connected via wired communication medium (Ethernet/CSMA devices and channel), and TCP/IP protocols. A pre-configured wireless communication medium can be used (instead of wired) simply by adding the compilation macro `RTDS_DEPL_WIFI` in the generation options:



These default configurations can be found in `<PragmaDev Studio installation dir.>/share/ccg/ns-3.10/bricks/RTDS_Startup_begin_cpp.c`. Changes can be made here in order to provide other configurations or modify the existing ones. Header files that maybe required in case of changes should be included in `<PragmaDev Studio installation dir.>/share/ccg/ns-3.10/bricks/RTDS_Startup_begin.c`.

It is advised to carefully study the ns-3 documentation prior to making any changes to the configurations. The ns-3 distribution used with PragmaDev Deployment Simulator can be found in <PragmaDev Studio installation dir.>/share/3rdparty/ns-3.10/ns-3.10.zip.

8.4 - Good coding practise

8.4.1 Memory allocation

Memory allocation has to be handled very carefully in real time systems since it can generate memory leaks leading to system crashes that can be very difficult to debug. Considering PragmaDev Studio is hiding the basics of the finite state machines it is important to point out what should be done in the code to have things work properly.

First of all when SDL messages are sent, received or saved, or when timers are set or reset, the generated code will handle memory allocation and de-allocation automatically so that the user does not have to deal with it. On the other hand, when user data is transmitted in a message from a process to another one, it is very important to define which process has the responsibility to free the corresponding memory. We strongly suggest the sender process always allocates the necessary memory and the receiver process always frees. It implies the sender process should not deal with the data any more after it has been sent. A good way to do so is to set the corresponding pointer to NULL after it has been sent out.

8.4.2 Shared memory

It is very common to use global variables or shared memory areas to exchange information between tasks. It is also very dangerous because two tasks could access the same information at the “same time” and read or write inconsistent information.

The same problem exists when using an instance of a class attached to a block or the whole system: all tasks know this instance, and can access its attributes or call its methods, leading to the same concurrency problems than for a global variable.

To avoid such problems we suggest to use a semaphore. Whenever a task needs to write or read a shared memory area, or to access an attribute or call a method on a shared object, it takes the semaphore. When the task is done, it gives it back to the system allowing another task to access the memory or the object. It is very important to do so even when reading memory or attributes since the reading task could be interrupted by a writing task. In that case the information read would be inconsistent.

8.4.3 C macros and functions

As explained in this user’s manual the code generator is based on C macros and C functions. Since these macros and functions are explained and delivered as source code it is very tempting to use them directly in C or to modify their source code.

It is important to realize these macros and functions have been designed to work with the generated code. They very often rely on the code generator to generate some complementary code to create a consistent behavior. They have also been deeply tested to guarantee safe code generation.

Before using these directly or trying to modify them it is important to deeply study and understand the delivered files to measure the impact of any modification. It is also very important to test any modification.

Furthermore the use of these macros and functions will make the design less legible where it is one of the key features of the tool.

8.5 - PragmaDev Debugger

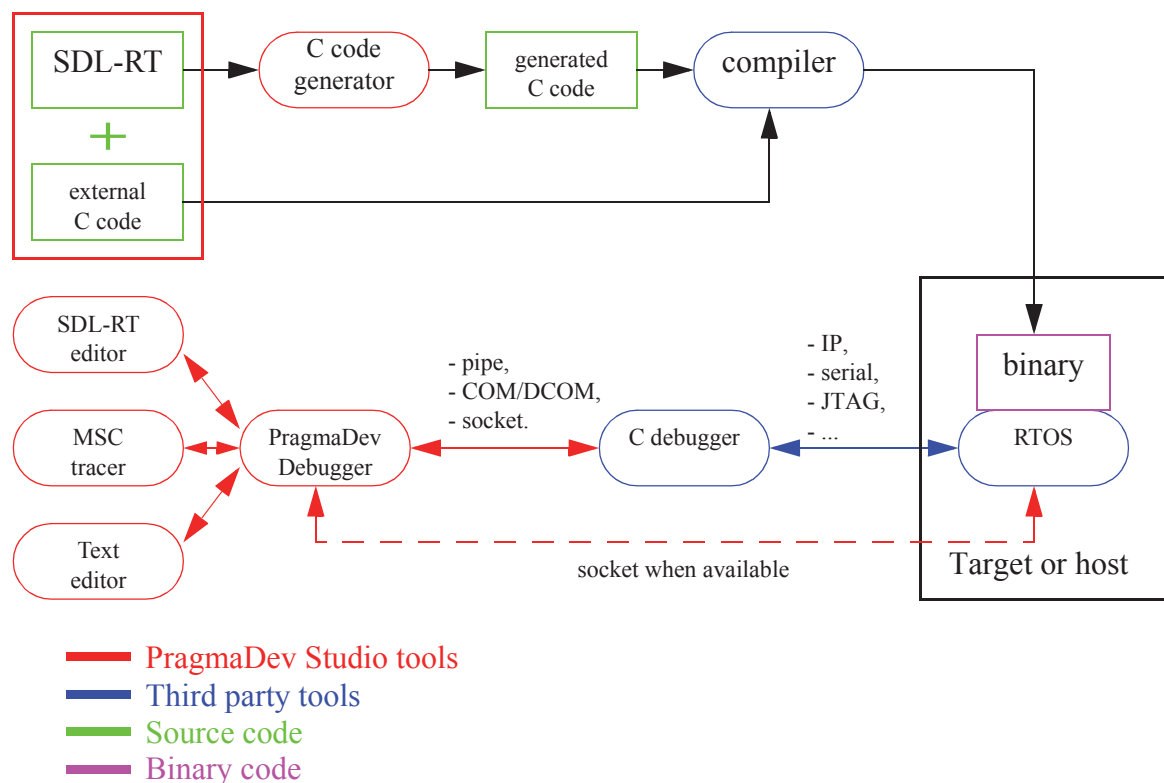
The PragmaDev Debugger relies on classical C debuggers or cross debuggers to allow graphical debugging and SDL-RT oriented information.

Currently supported debuggers are:

- Tornado
- gdb
- MinGW
- Tasking Cross View Pro C166/ST10
- XRAY
- Multi 2000

8.5.1 Debugger architecture

The PragmaDev Debugger allows you to execute your SDL-RT system and the associated C code. To do so PragmaDev Studio generates the code necessary to execute the SDL-RT processes on host or target and interfaces with a debugger or a cross debugger.



The PragmaDev Debugger has all the expected features allowing you to:

- Graphically trace the internal behavior of the system
- Graphically step in the SDL or C source code

- Visualize all key internals of your system such as:
 - Tasks,
 - Semaphores,
 - Timers,
 - Local variables in the current frame,
 - Global variables.
- Send SDL messages to your system,
- Modify SDL state,
- Modify variables value.

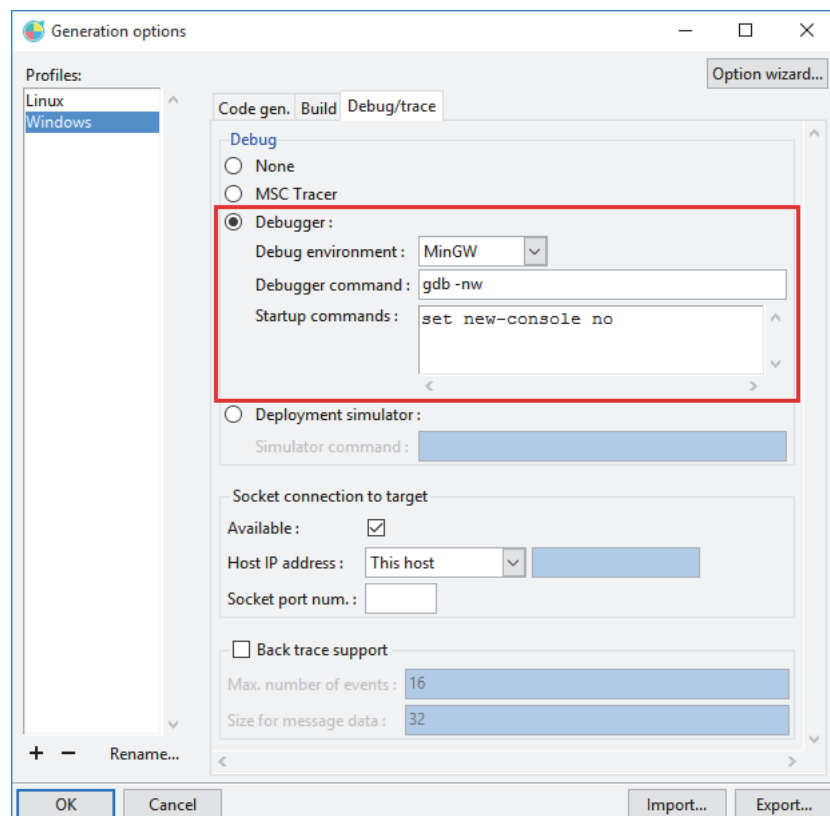
8.5.2 Launching the PragmaDev Debugger

Before starting the PragmaDev Debugger the generation profile should be verified. Graphical debugging has a specific profile since it will:

- automatically define some compiler options such as `-g` and `-DRTDS_SIMULATOR` that are defined in the `DefaultOptions.ini` file in the `$(RTDS_HOME)/share/ccg/<RTOS>` directory,
- launch the debugger automatically.


A generation profile is considered a debug profile as soon as *Debugger* is selected in the “Debug” section in the “Debug / trace” tab.

Generation profiles are edited from “Generation / Options...” menu. A typical debugging profile would look like this:

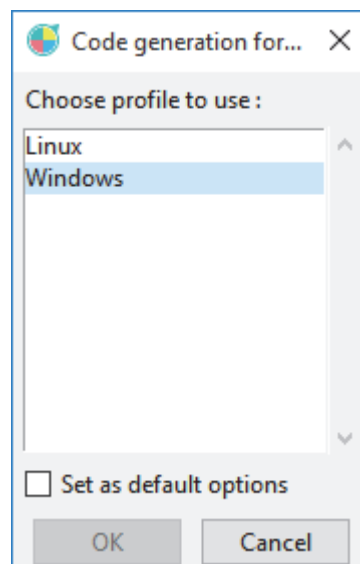


Once the code generation profile is selected the tool will:

- Check syntax and semantic of the SDL-RT system,
- Generate the C code,
- Compile and link the C code,
- Start the selected debugger environment,
- Load the executable,
- Start the executable with a breakpoint on it so it will stop on `RTDS_Start` function.

The PragmaDev Debugger is started from the “Generation / Execute” menu or from the  quick button.

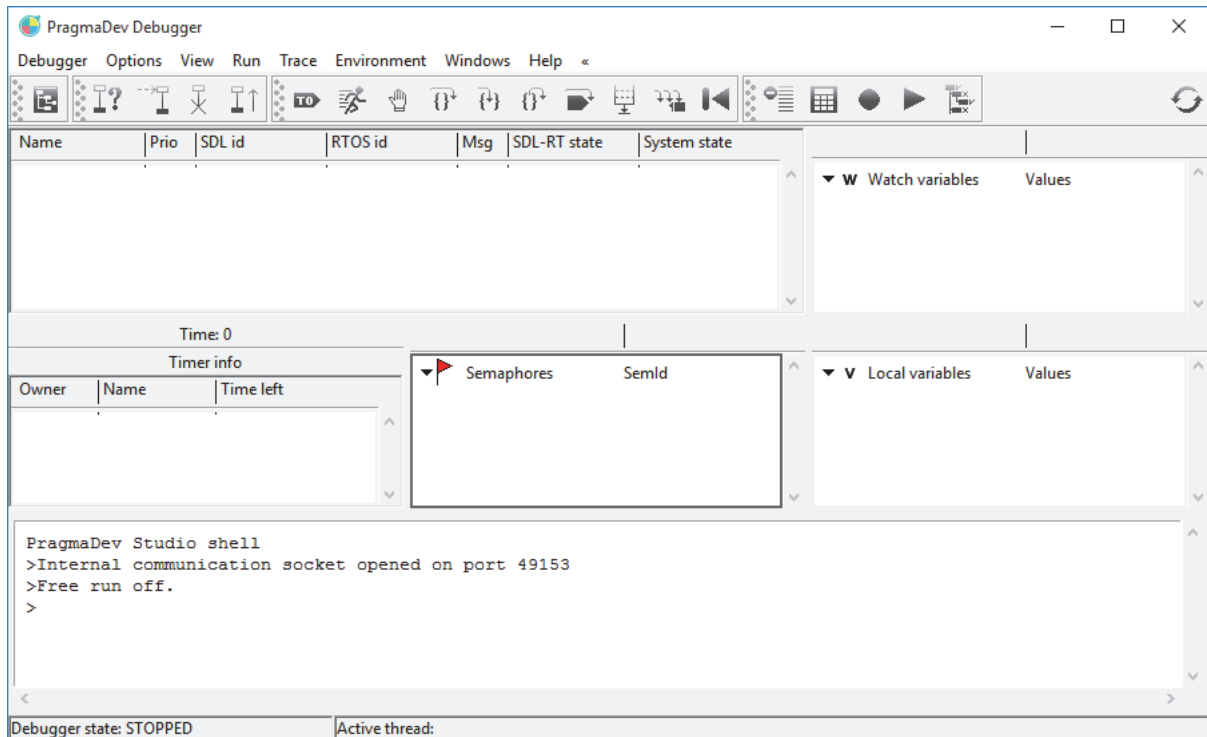
If several debug profiles are available a pop-up window will ask to select the desired profile:



When always using the same profile it is possible to set a default profile to launch so that the selection window does not pop up.

Syntactic check, semantic check, code generation, and compilation are done. The selected debugging environment is started with the *Debugger command* defined in the generation profile.

The PragmaDev Debugger window is started automatically and you are ready to debug your system!






The PragmaDev Debugger window



The PragmaDev Debugger can be restarted at any time with the reset button or shell command. The underlying C debugger is restarted and the executable is reloaded so that the environment is cleaned up.

8.5.3 Stepping levels

Since your source code is a composite of SDL and C and considering some code has also been generated by the code generator, the debugger offers several ways to execute the code:

- Run with SDL key events trace information,

 Menu "Options / Free run" de-activated. This is the default setup where the PragmaDev Debugger traces all SDL key events and displays textual and / or SDL and / or MSC traces.
- Run without SDL key events trace information,

 Menu "Options / Free run" activated. The tracing mechanism uses a breakpoint on `RTDS_DummyTraceFunction` in the generated C code. When this option is activated the breakpoint is removed and the system runs freely. Of course no trace information is available then.
- Stop execution,

 Stops execution of the running system.
- C step mode



Step line by line in any C code,



Step-out a C function,



Step-in a C function.

- SDL-RT automatic stepping,



Steps automatically at C level until it reaches a generated C line corresponding to an SDL-RT graphical source code symbol. Note it might generate a lot of C steps and the expected result depends on the underlying debugger and RTOS integration. For example with gdb and windows integration it will step in the same task and let the other tasks run. But with Tasking and CMX it will step from one task to the other following the RTOS scheduling mechanism.

- Step until the next SDL key event such as:



- Message sending,
- Message received,
- Timer started,
- Timer cancelled,
- Timer went off,
- Semaphore take attempt,
- Semaphore take succeeded,
- Semaphore give,
- SDL state modification,
- SDL process created,
- SDL process deleted.

For your information these key events are traced via a breakpoint on an empty C function called `RTDS_DummyTraceFunction()`. So do not be surprised if you end up in this empty function; it is normal...

- Run until RTOS message queue is empty



This will run the system until one of the external message queue is empty. When using the scheduler, its internal queue is read until it is empty before the external queue is read. If the whole system is scheduled this feature can be used as a run until timer.

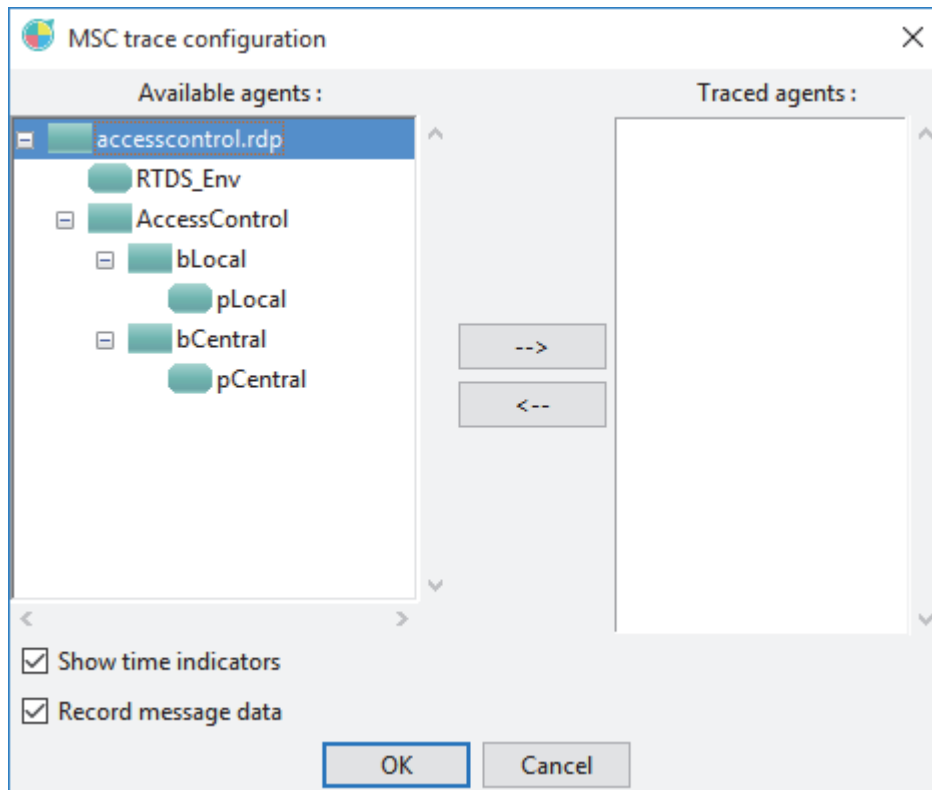
8.5.4 MSC trace

The MSC Tracer allows you to graphically trace execution of the system with its SDL key events. It is possible to configure the MSC trace to define at which level of details the architecture of the system should be represented. The MSC trace can be made at system, block, process or any combination of agents. Any agent selected will be represented by a




lifeline in the MSC diagram. Any messages exchanged inside the agent will not be seen on the MSC. The default view is the most detailed one, with a lifeline for each process.

- Configure the MSC trace

The  quick button opens the *MSC trace configuration* window:



The following options are available:

- Show system time information,
- Record and display message parameters,
- SDL-RT architecture elements to trace.
- Start the MSC trace
 - The  quick button starts the MSC Tracer. By default the trace is active.
- Stop the MSC trace
 - The  quick button stops the MSC Tracer.
- Trace the last SDL-RT events (backTrace)
 - The  quick button opens an MSC Tracer and displays the last SDL-RT events. The number of events traced are configured in the generation options.

8.5.5 Displayed information

The PragmaDev Debugger window is divided in 5 parts described below. Each time an SDL key event is received all the information is updated.



If needed the displays can be refreshed at any time with the refresh button or shell command.

The information to refresh can be setup in the “Options / Refresh options...” menu as explained in “Refresh options” on page 239.

8.5.5.1 Processes

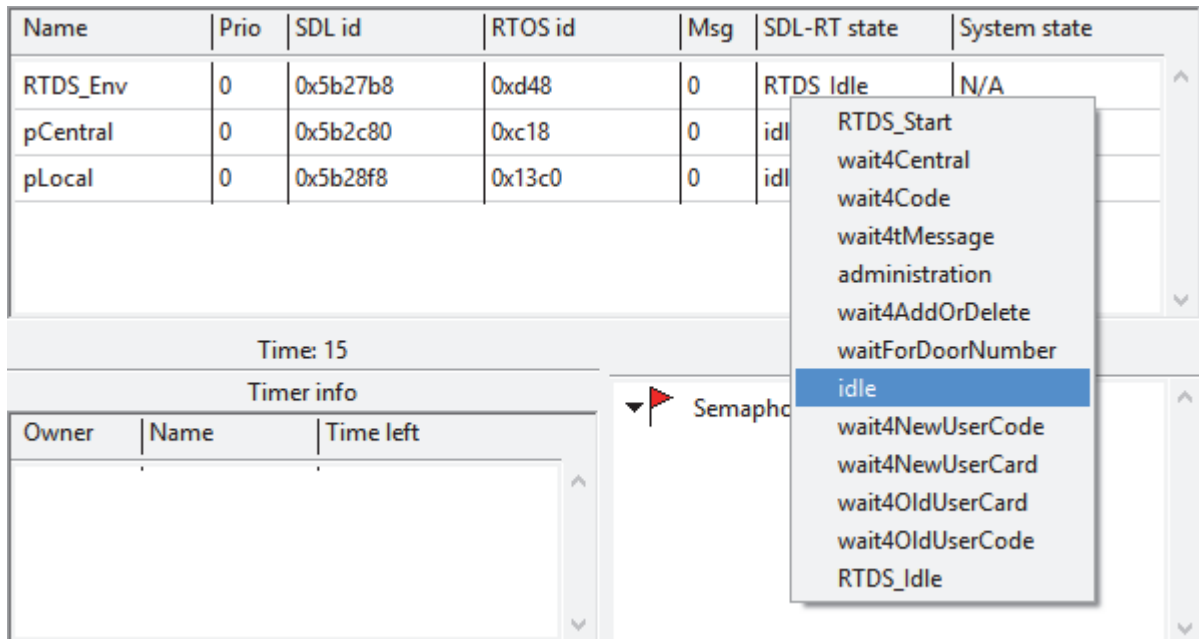
The *Process information* part list all processes defined in the SDL-RT system. It will not list any other processes running on the RTOS. The displayed information is:

- **Name**
This field displays the name of the process as defined in the Process create SDL symbol. Several tasks can have the same name. The SDL id should then be used to distinguish them. When using the SDL output TO_NAME symbol it will search for the value of that field on the target to find the receiver.
- **Prio**
This field displays the priority of the task as defined in SDL process create symbol. The value is expressed in decimal. This value is not available on all integrations.
- **RTOS id**
This field shows the Process Identifier of the task as defined by the RTOS. As several processes can be run within the same RTOS task with the scheduler, please note several processes can have the RTOS id.
- **SDL id**
This field is a unique identifier of the running process.
- **Msg**
This field shows the number of messages waiting in the task’s queue. It does not include saved messages.
- **SDL state**
This field is the internal SDL state of the SDL process as defined in the SDL diagram.
- **System state**
This field is the task state from the RTOS point of view. Typically if a process is hanging on its queue or a semaphore it is in the PEND state. If the task is running it is in the READY state. This information is not available on all integrations.

When the system is running the active process line is printed in red. Double-clicking on any process name in the list will open the corresponding diagram in an editor.

The *Process information* window also allows to modify the SDL-RT state of a process. To do so right click on the SDL state column of the process line. A pop up menu will list all

the available SDL state that have been defined in the system. Select one and the SDL state is modified.



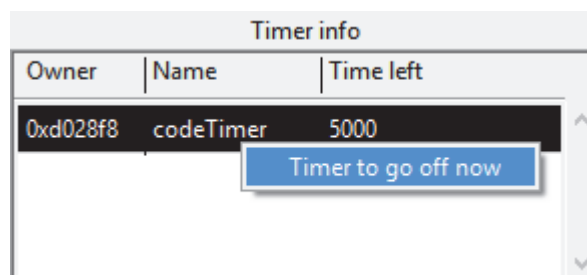
SDL-RT state modification example

8.5.5.2 Timers

The *Timer info* part displays all on-going timers started from the SDL-RT design. The displayed fields are:

- Name
Name of the timer as defined in the SDL-RT design.
- Pid
Identifier of the task that started the timer.
- Time left
Time left before the timer goes off. The display is updated when an SDL key event occur so the value displayed here is the time left when the last SDL key event occurred.

With windows and posix integrations it is possible to simulate discrete time. To do so RTDS_DISCRETE_TIME must be defined in the compiler options. In that configuration time will never increase until the user fires a timer. To make a timer go off, right click on the timer's name.



Example: have codeTimer to go off

8.5.5.3 Semaphores

The semaphore tree lists all semaphore declared in the SDL-RT system and their address. When expanded it shows the current state, type and options of the semaphore. If processes are blocked on the semaphore they will all be listed after the information line.

It is important to understand how the trace works with semaphore in order to understand that what you see might not be what is really happening on the target. When taking a semaphore the PragmaDev Debugger distinguishes two key events: an attempt to take the semaphore and a successfully taken semaphore. If the second key event is not seen, the semaphore tree is not updated but it might be because the semaphore is blocked on it. The information will be displayed at the next SDL key event; not before. Let's take an example to make it clear: process P1 has taken semaphore S1 and process P2 makes an attempt to take S1. The PragmaDev Debugger trace will display:

Semaphore: S1(0x4b3eeb8) take attempt by: P2 at: 0x73d ticks

P2 will get blocked on S1 and if there is no SDL key event happening the semaphore tree will not be refreshed and display no blocked process on S1... In such a case you should use the *refresh* button to update the tree.

8.5.5.4 Watch

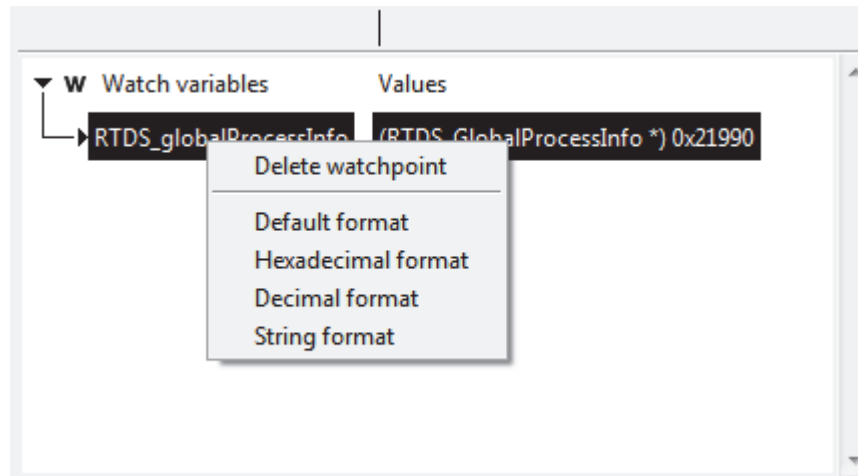
There are several ways to add a variable in the *Watch window*:

- From the shell
Type the following command in the shell:
`watch add <variable name>`
- From the text editor when the PragmaDev Debugger window is open
Select an expression in the editor and go to the "Debug / Add watch" menu to add the expression in the *Watch window*.
- From the SDL-RT editor
Select an expression in the SDL-RT editor and go to the "Debug / Add watch" menu to add the expression in the *Watch window*.

Variables can be removed from the *Watch window*:

- from the shell with the following command:
`watch del <variable name>`

- from the *PragmaDev Debugger Watch window* with right mouse button as shown below:

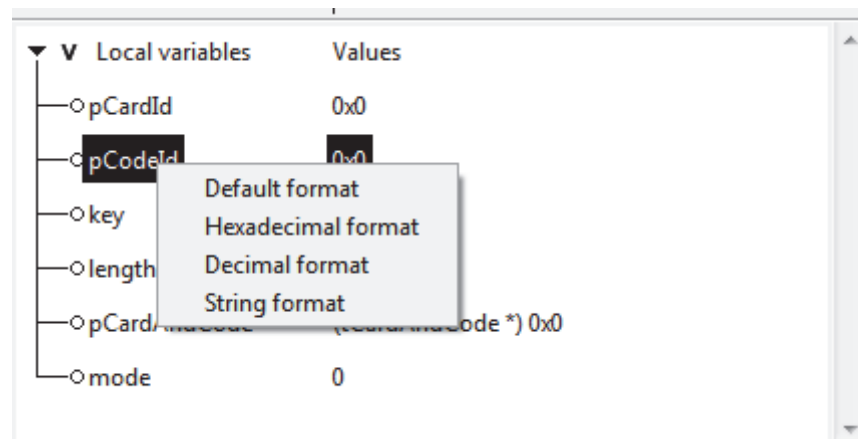


The *Watch window* also allows to modify the value of variables. To do so double click on the value of the variable to be modified. Press <Return> and the value is updated.

8.5.5.5 Local variables

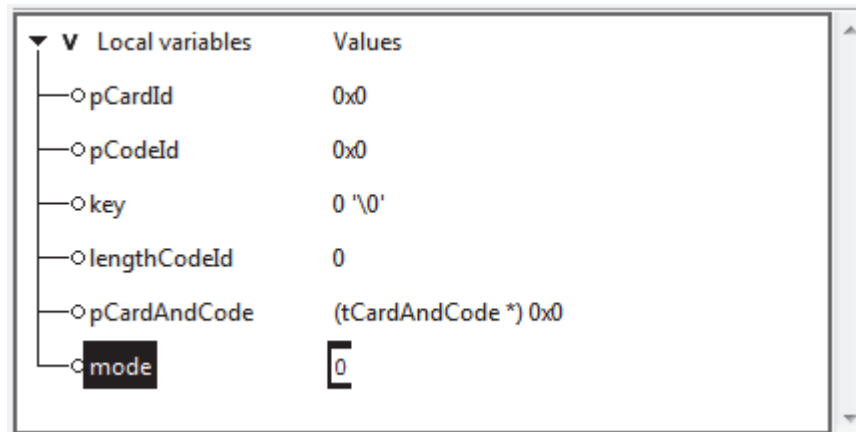
When stepping through the code the PragmaDev Debugger automatically displays the local variables of the current stack frame. That gathers all local variables of the current C function including the arguments of the function. Nothing has to be done to update the PragmaDev Debugger *Local variable window*.

Depending on the type of the variable the best display format is automatically selected but it is possible to select a specific format to display a value. To do so, right click on the variable and the following pop up menu will be displayed:



When stepping in the generated C code the current stack frame contains local variables used by PragmaDev Studio to handle internal information. All these variables name start with *RTDS_* so that there is no confusion with any other variable. Since the PragmaDev Debugger is designed to debug the SDL-RT system these variables are hidden from the *Local variables window*. But it is possible to display them with the “Option / Show internals” menu.

The *Local variables window* also allows to modify the value of variables. To do so double click on the variable to edit the value. Press <Return> and the value is updated.



Setting a local variable value example

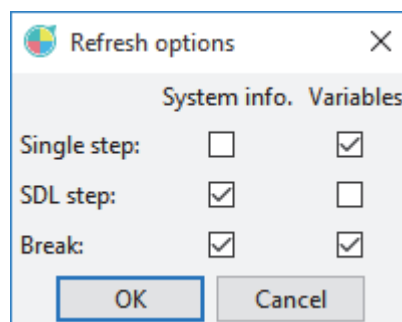
8.5.5.6 Refresh options

The information displayed in the PragmaDev Debugger windows are divided in 2 categories:





- System info
 - Process information
 - Timer information
 - Semaphore information
- Variables
 - Local variables
 - Watch variables

Retrieving any information from the target is time consuming. In order to optimize the response time it is possible to configure which category of information is refreshed.

The configuration is done in the “Options / Refresh options...” menu.



Default Refresh options

- C step means the use of one of the following step button:   ,
In the default options, only the Variables category is refreshed since there is no reason the System information category has changed in the meantime.
- SDL step means the use of  step button,

When stepping from an SDL event to another, only the System information category is interesting to update.

- Break means the system has hit a breakpoint.
In that case it is recommended to update all the information.

Anyway, at any time it is possible to refresh all information: 

Note signals (SIGINT, SIGSEV...) will be reported in the PragmaDev Debugger but no refresh action is done.

8.5.6 Shell

The PragmaDev Studio shell allows to enter all commands listed above and is used as a textual trace.

The available commands are grouped in categories. To list all the available categories type:
help

It will list the following categories:

Type help followed by a category to list available commands

```
-----  
shell  
execution  
interaction  
variables  
trace  
customization
```

Type help followed by a category name to list the corresponding commands.

To list all the available commands, type:

h

It will list the following commands:

Command - Explanation

```
-----  
h - lists all commands  
history - list the last entered valid commands  
clear - clears the shell  
echo <string> - echos a string in the shell  
include <file name>  
resume - resumes the scenario  
repeat <repeat count> <shell command> [||; <shell command>]*  
# <comment>  
! <any host command>  
refresh - refreshes all data in the window  
run - runs the SDL system
```

stop - stops the SDL system

step - step in the code

stepin - step in function calls

stepout - step out a function call

keySdlStep - run until the next key SDL event

sdlTransition - run until the end of the SDL transition

runUntilTimer - run all transitions until timers

runUntilQueueEmpty - run all transitions until RTOS queue is empty

resetSystem - resets the running system

list - list breakpoints

watch add [<pid>:]<variable name>[<field separator><field name>]*

watch del [<pid>:]<variable name>[<field separator><field name>]*

break <break condition> [<ignoreCount> <volatile>]

delete <breakPoint number>

db <any debugger command>

set time <new time value>

send2name <sender name> <receiver name> <signal number or name> [<parameters>]

send2pid <sender pid> <receiver pid> <signal number or name> [<parameters>]

sendVia <sender pid> <channel or gate name> <signal number or name> [<parameters>]

send <sender pid> <signal number or name> [<parameters>]

systemQueueSetNextReceiverName <receiver name>

systemQueueSetNextReceiverId <receiver id>

extractCoverage <file name>

connect <port number>

connectxml <port number>

disconnect

varFromType <variable name> <variable type>

varFromValue <variable name> = <initial value>

varFieldSet <variable name>[.<field name>]* = <field value>

dataTypes <on | off>

print <variable name>

sdlVarSet [-x] [<process id>:]<sdl variable name>=<value>

sdlVarGet [-x] [<process id>:]<sdl variable name>

backTrace - display last events traced when activated in profile

setupMscTrace <time information> <message parameters> [<agents>]

startMscTrace

stopMscTrace

saveMscTrace <file name>

setEnvInterfaceFilter 1|0

buttonWindowCreate <button window name>

buttonWindowAdd <button window name> <button name> = <shell command> [|; <shell command>]*

buttonWindowDel <button window name> <button name>

```
buttonWindowLabelAdd <button window name> <label name>
buttonWindowLabelDel <button window name> <label name>
startPrototypingGui
```

In any of the shell commands the following can be used:

```
$(<os environment variable>) to acces an operating system environment variable
${<interactive label>} pops up an interactive window to get variable value, /s, /b and others can be used
${<shell variable name>} will be replaced by the shell variable value
${<process name>:<instance number>} will be replaced by the pid of the instance of the process
& <any command> will prevent the above pre-processing
<partial command>\ and continue the command on the next line of the shell
```

The last valid commands can be recalled with the upper arrow.

Some of these commands are the equivalent to buttons in the button bar. Some are specific to the shell and will be further explained below.

8.5.6.1 shell commands

To list all the available commands in this category, type:

```
help shell
```

It will list the following commands:

```
Command      - Explanation
```

```
h              - lists all commands
history        - list the last entered valid commands
clear          - clears the shell
echo <string>  - echos a string in the shell
include <file name>
    run a scenario of commands out of a file
resume         - resumes the scenario
repeat <repeat count> <shell command> [||; <shell command>]*
    repeat a set of shell commands
# <comment>
    does nothing
! <any host command>
    runs any host command
```

In any of the shell commands the following can be used:

```
$(<os environment variable>) to acces an operating system environment variable
${<interactive label>} pops up an interactive window to get variable value, /s, /b and others can be used
${<shell variable name>} will be replaced by the shell variable value
${<process name>:<instance number>} will be replaced by the pid of the instance of the process
& <any command> will prevent the above pre-processing
<partial command>\ and continue the command on the next line of the shell
```

- Running scenarios

A set of commands can be saved to a script file with the red circle button in the tool bar. The include command or the play button allows to run a script file. The script file is stopped when a breakpoint is hit or when the stop button is pressed. Type the resume command to resume the scenario.

- Process instances pid

It is possible to get a process instance pid with the `|$< <process name> >` syntax.

Example:

In the following configuration:

Name	Prio	SDL id	RTOS id	Msg	SDL-RT state	System :
pCentral	0	0xc919d8	0xc70	0	idle	N/A
RTDS_Env	0	0xc91aa8	0x318	0	RTDS_Idle	N/A
pLocal	0	0xc91b78	0xf10	0	RTDS_Start	N/A

```
echo |$<pCentral:0>
```

prints the pid (SDL id) of the first instance of pCentral:
0xc919d8

Note:

This feature does not work if the “Options / Free run” is activated.

- Environment variables

Operating system environment variables can be accessed with the `|$(<variable name>)` syntax.

Example:

```
echo |$(RTDS_HOME)
```

prints:

C:\RTDS

8.5.6.2 execution commands

To list all the available commands in this category, type:
help execution

It will list the following commands:

Command	- Explanation

refresh	- refreshes all data in the window
run	- runs the SDL system
stop	- stops the SDL system
step	- step in the code

```

stepin          - step in function calls
stepout         - step out a function call
keySdlStep      - run until the next key SDL event
sdlTransition   - run until the end of the SDL transition
runUntilTimer   - run all transitions until timers
runUntilQueueEmpty - run all transitions until RTOS queue is empty
resetSystem     - resets the running system
list            - list breakpoints
startPrototypingGui

watch add [<pid>:<variable name>[<field separator><field name>]*
  adds a variable to watch:
  <pid> is the process id in which the variable is. Only available in Z.100 simulation.
  <variable name> is the name of the variable
  <field separator> is '!' in SDL Z.100 or '.' in SDL-RT
  <field name> is the name of the variable field or sub-field
watch del [<pid>:<variable name>[<field separator><field name>]*
  remove a variable to watch
  <pid> is the process id in which the variable is. Only available in Z.100 simulation.
  <variable name> is the name of the variable
  <field separator> is '!' in SDL Z.100 or '.' in SDL-RT
  <field name> is the name of the variable field or sub-field
break <break condition> [<ignoreCount> <volatile>]
  break condition is a function name or '*'break-address or file-name':line-number or
  diagram-file-name':symbol-id':line-number
  ignoreCount is a number
  volatile is a boolean: 'true' or 'false'
delete <breakPoint number>
db <any debugger command>
  the command is directly sent to the debugger with no verification

```

- **db**

The shell offers a way to directly type debugger commands. You just have to type "db " before the actual command and it will be directly passed to the debugger without any verification except for one command that is "set annotate" in the Gnu and Tornado integration. This is because with these debuggers, the PragmaDev Debugger is running with annotate level 2 and would not be able to synchronize anymore with *gdb* if you change it. The consequence for you is that the format of the answer is different from what you are used to but you will get the information. Check *gdb* reference manual for more information.

8.5.6.3 interaction commands

To list all the available commands in this category, type:
 help interaction

It will list the following commands:

Command - Explanation

set time <new time value>

new time value can be absolute time or '+'delta

send2name <sender name> <receiver name> <signal number or name> [<parameters>]

send2pid <sender pid> <receiver pid> <signal number or name> [<parameters>]

sendVia <sender pid> <channel or gate name> <signal number or name> [<parameters>]

send <sender pid> <signal number or name> [<parameters>]

environment name is 'RTDS_Env' and environment pid is '-1'

parameters are |{field1|=value|,field2|=value|,...}|

systemQueueSetNextReceiverName <receiver name>

systemQueueSetNextReceiverId <receiver id>

extractCoverage <file name>

connect <port number>

to connect to an external tool on a socket using the shell format

connectxml <port number>

to connect to an external tool on a socket using the xml-rpc format

disconnect

to disconnect from the external tool

startPrototypingGui

- **set time**
This command sets a new system time value if the debugger allows it. Please check the reference manual for more information.
- **connect**
This command opens a socket in server mode to connect an external tool to the PragmaDev Studio shell. The parameter is the port number on the host IP address. This command should be done before starting the client.
- **disconnect**
Disconnect the socket from the external tool.
- **System queue manipulation**
It is possible to re-organize the system queue order from the shell. The system-QueueSetNextReceiverName will put up front in the system the next message for the defined receiver name, and systemQueueSetNextReceiverId will put up front in the system queue the next message for the defined receiver pid.
- **extractCoverage**
Extracts the code coverage for the current debug session so far and stores it in the specified file. If the file name is relative, it will be taken from the project directory. Please note that if this command is used in a debug session run via the rtdsSimulate command line utility, the project will be saved in the end and the code coverage results stored in it.

8.5.6.4 variables commands

To list all the available commands in this category, type:
 help variables

It will list the following commands:

Command	- Explanation

varFromType <variable name> <variable type>	creates a variable of the given type to be used in the shell
varFromValue <variable name> = <initial value>	creates a variable with the given initial value to be used in the shell
varFieldSet <variable name>[.<field name>]* = <field value>	sets a single variable field to a given value
dataTypes <on off>	prints the type of the variable
print <variable name>	prints the variable value
sdlVarSet [<process id>:]<variable name>=<value>	
sdlVarGet [<process id>:]<variable name>=<value>	

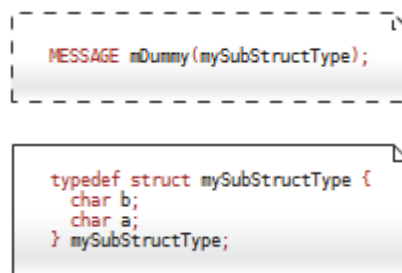
- shell variables

It is possible to define variables in the shell and to use them in send2xxx commands using the |\$(<variable name>) syntax.

- varFromType

This command allows to declare a variable based on a type defined in the SDL-RT system. Only the types used as parameters in messages are available. The message parameters need to be defined in a super-structure in order to be compliant with the generated code (except if there is a unique pointer type parameter).

Example:



Shell commands to define a variable based on the type:

```
>varFromType myVar mySubStructType
>print myVar
|{b|= |,a|=0|}
>varFieldSet myVar.b=z
>varFieldSet myVar.a=666
>print myVar
|{b|=z|,a|=666|}
```

```
>send2name pPing normal mDummy |{ |$(myVar) |}
send2name pPing NORMAL_SIGNAL mDummy |{ |{b|=z |,a|=666 |} |}
>
```

- **varFromValue**

This command allows to declare a variable with no type based on its value. Shell commands to define a variable based on its values:

```
>varFromValue myVar={b|= |,a|=0 |}
>print myVar
|{b|= |,a|=0 |}
>varFieldSet myVar.b=z
>varFieldSet myVar.a=666
>send2name pPing normal mDummy |{ |$(myVar) |}
send2name pPing NORMAL_SIGNAL mDummy |{ |{b|=z |,a|=666 |} |}
>
```

- **varFieldSet**

This command sets a field of the variable. This can only be used on simple type fields.

- **print**

This command prints a shell variable value.

- **dataTypes**

This command is a verbose mode that displays the type when printing data.

- **Accessing variables**

- **Shell variables**

Shell variables can be accessed with the `|${<variable name>}` syntax.

Example:

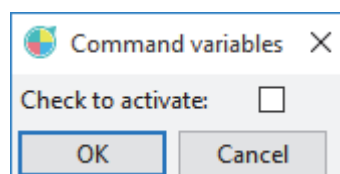
```
varFromType myVar mySubStructType
print myVar
|{b|= |,a|=0 |}
echo |${myVar}
echos:
|{b|= |,a|=0 |}
```

- **Interactive variables**

It is possible to ask the user for a value with the `|${<input label>}` syntax. Options for the input label are: For strings, the only option is its length (default: 20). For booleans, options are the value when checked and the value when unchecked, separated by a comma. For example, a field with type “b[-r,]” will be replaced in the command by “-r” if the user checks the corresponding checkbox, and by the empty string otherwise. The defaults are “1” for checked and “0” for unchecked.

Example:

```
echo |${Check to activate: /b}
pops up the following window:
```



prints 1 if checked or 0 if unchecked.

8.5.6.5 trace commands

To list all the available commands in this category, type:
help trace

It will list the following commands:

Command	- Explanation

backTrace	- display last events traced when activated in profile
setupMscTrace <time information> <message parameters> [<agents>]	
sets up the MSC trace where:	
<time information> is 0 or 1	
<message parameters> is 0 or 1	
<agents> is the list of agent names to trace separated by spaces	
startMscTrace	
stopMscTrace	
saveMscTrace <file name>	
setEnvInterfaceFilter <filter status>	
<filter status> is 1 or 0, when active only messages with the environment will be traced	

- **MSC trace**
The MSC trace can be configured, started, stopped, and saved from the shell.
Example:
setupMscTrace 0 1 pPing
Will only trace pPing instance with no time information but with parameters.
- **Filtering the interface between the environment and the system**
The setEnvInterfaceFilter command is not available in SDL-RT.

8.5.6.6 customization commands

To list all the available commands in this category, type:
help customization

It will list the following commands:

Command	- Explanation

buttonWindowCreate <button window name>	
creates a window to contain user defined buttons	
buttonWindowAdd <button window name> <button name> = <shell command> [; <shell command>]*	
adds a button to previously created button window	
<button window name> is the name of the button window	
<button name> is the text to be displayed on the button	
<shell command> is the command associated with the button	

`buttonWindowDel <button window name> <button name>`

removes a button from a button window

<button window name> is the name of the button window

<button name> is the text of the button to be removed

`buttonWindowLabelAdd <button window name> <label name>`

adds a label to previously created button window

<button window name> is the name of the button window

<label name> is the text to be displayed on the label

`buttonWindowLabelDel <button window name> <label name>`

removes a label from a button window

<button window name> is the name of the button window

<label name> is the text of the label to be removed

- **Button windows**

It is possible to create user-defined buttons and to associate shell commands.

Here is an example of a button window:

```
>buttonWindowCreate myWindow
```

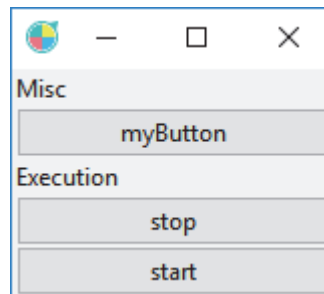
```
>buttonWindowLabelAdd myWindow Misc
```

```
>buttonWindowAdd myWindow myButton = help
```

```
>buttonWindowLabelAdd myWindow Execution
```

```
>buttonWindowAdd myWindow stop = send2name pPing normal mStop
```

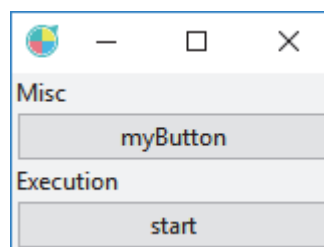
```
>buttonWindowAdd myWindow start = send2name pPing normal mStart  
|{param1|=12345|}
```



So clicking on *myButton* will actually execute the help command in the shell.

It is also possible to remove labels or buttons:

```
>buttonWindowDel myWindow stop
```



It is possible to create several button windows.

To stop one of the windows, just close the window.

8.5.7 Status bar

The status bar is divided in two parts:

- The PragmaDev Debugger internal state
 The PragmaDev Debugger can have the following internal states:

State	Meaning
STOPPED	The system is stopped
STOPPING	The system is trying to stop. No commands are allowed in that intermediate state.
RUNNING	The system is running. The traces might be active or not (Options / Free run). A stop is possible in that state.
STEPPING	C code classical stepping. Note a classical step might take a lot of time. A stop is possible in that state.
KEY_SDL_STEPPING	Step to the next SDL key event. Note an SDL step might take a lot of time. A stop is possible in that state.
ERROR	An error has occurred and the PragmaDev Debugger is stuck. Restart the PragmaDev Debugger.

Table 2: PragmaDev Debugger internal states

- The active thread
 The active thread is displayed in the right part of the status bar when known.

8.5.8 Breakpoints



8.5.8.1 Setting breakpoints

There are three ways to set breakpoints:

- In the PragmaDev Studio shell
`break <break condition> [<ignoreCount> <volatile>]`
 - break condition can be
 - a function name
 - a break address starting with '*'
 - a specific line in a file with the following form: `file_name:line_number`
 - a specific line in a symbol in a diagram with the following form:
`diagram_file_name:symbol_identifier:line_number`
 - ignoreCount is a number meaning how many times the breakpoint should be ignored. For example: to stop when the break condition is hit the 5th time ignoreCount should be set to 4. The default value is 0.


- `volatile` is a boolean that can take value “true” or “false”. When true the breakpoint is deleted when hit.

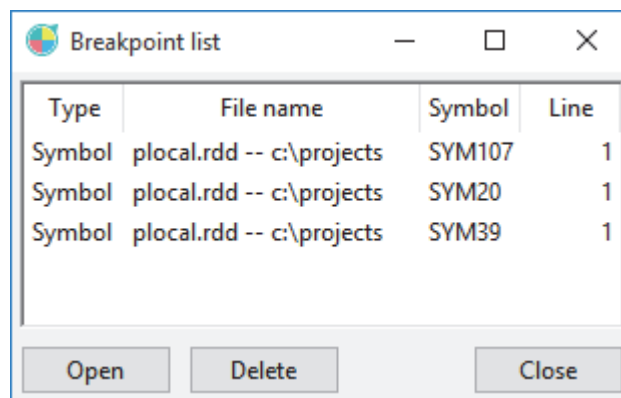
Note that setting a breakpoint interactively will record the “break” command in the shell history. This can be especially useful for breakpoints set on symbol, where the symbol identifier is not directly visible in the diagram.

- In the text editor, select a line in a source file and go to “Debug / Set breakpoint” menu to set a simple breakpoint, or click on the  button in the debug toolbar.
- In the SDL-RT diagram editor, select an SDL-RT symbol and go to “Debug / Set breakpoint” menu to set a simple breakpoint, or click on the  button in the debug toolbar.

8.5.8.2 Listing breakpoints

The breakpoints set can be listed:

- In the PragmaDev Studio shell with the `list` command.
- In the breakpoint list window by clicking on the  button in the toolbar. This window looks like follows:



For each breakpoint is given:



- its type: symbol or file,
- the file name for the diagram or source file where it is set,
- the internal identifier for the symbol where it is set if applicable,
- and the line number in the source file or symbol text where it is set.

From this window, selecting a breakpoint and clicking on “Open” or double-clicking on a breakpoint line will display the symbol or file at the position of the breakpoint, and selecting a breakpoint and clicking “Delete” will delete the breakpoint.

It is important to note this listing will only show breakpoints that have been set with PragmaDev Studio tools. For example, if a breakpoint has been directly set with `gdb`, it will not appear.

8.5.8.3 Deleting breakpoints

Breakpoints can be deleted from:

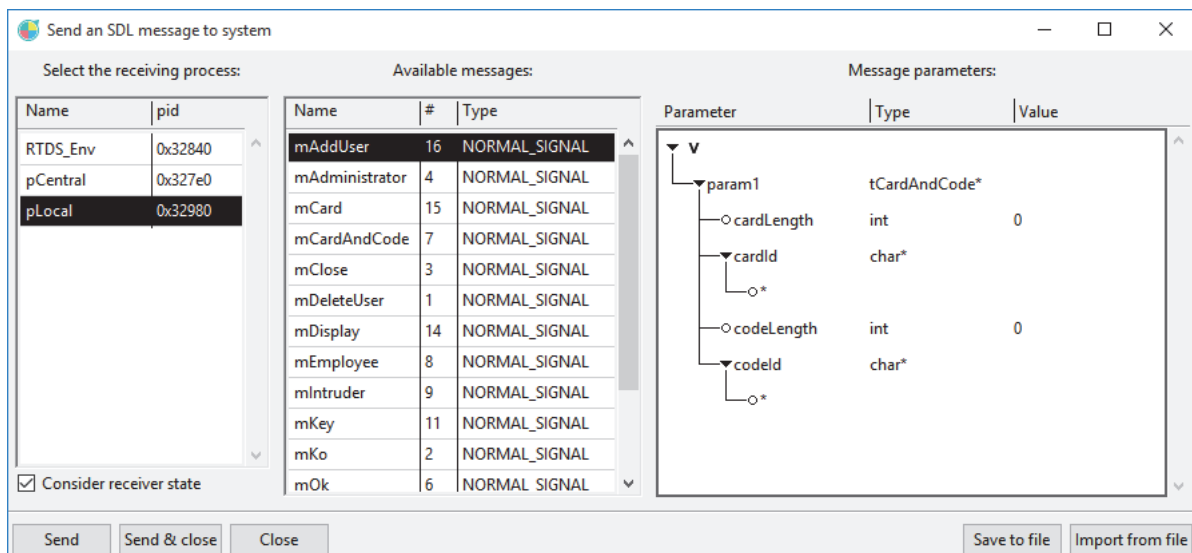
- The shell with the delete command:
`delete <breakpoint number>`
 where the breakpoint number is the number listed from the list command.
- The breakpoint list window, as explained in “Listing breakpoints” on page 251.
- The text editor: select a line where a breakpoint is set and press the  button in the debug toolbar.
- The diagram editor: put the text cursor in a symbol at a line where a breakpoint is set and press the  button in the debug toolbar.

8.5.9 Sending SDL messages to the running system

8.5.9.1 Send SDL message window



The PragmaDev Debugger’s *Send an SDL message to the running system* button opens the *Send an SDL message to system* window. It will list the possible receivers, and the available messages in the system:



The SDL message send Window

Please note that it is not possible to set a real pointer value with this interface. All pointers are considered NULL or to be allocated dynamically on the target. The authorized pointer values are:

- o for null pointers,
- '' for pointers to be allocated on the target.

To specify real pointer values please use the shell command described below.

8.5.9.2 Send SDL message shell commands

The equivalent commands in the shell are:

```
send2name <sender name> <receiver name> <signal number or name> [<parameters>]
send2pid <sender pid> <receiver pid> <signal number or name> [<parameters>]
```


Verifications are made on the sender pid and receiver pid only.

The format for the <parameters> argument depends on whether the message is structured or not. Structured parameters are fully described in PragmaDev Studio Reference Manual. In short, a message is structured if and only if it is declared with several parameters or with one parameter that is a pointer to a struct or a union.

- For a non-structured message, the text for the parameter must be a sequence of bytes written in hexadecimal format, exactly as they will appear in the target program memory.
- For a structured message, the text for the parameter must be written as follows:
 - The values for base types are written as in C: for example 12 or 871 are valid values for an int, X is a valid value for a char, and so on...
 - The values for pointers are written in hexadecimal, optionally prefixed by 0x, and followed by | : and the pointed value. If the value for the pointer is not specified, a new block will be automatically allocated on the target. For example, for an int*:
 - 804A51FE| :67 will set the pointer to the hexadecimal value 0x804A51FE and put 67 in the pointed value;
 - | :123 will allocate a new int* on the target and put the value 123 in it;
 - 0x0 will set the pointer to NULL.

There is a special case for char* pointers: the value can be a full string instead of just a single char. Please note all ' ' characters must be doubled in this string.

- The values for structs or unions are coded as follows:

```
|{field1|=value|,field2|=value|,...|}
```

For example, for a struct defined as:

```
struct MyStruct { int i; char *s; };
```

a valid format is:

```
|{i|=4|,s|=|:abcd|}
```

In the struct created on the target, the field i will be set to 4 and the field s will be automatically allocated with length 5 and the string will be set to "abcd".

Please note that what is significant in the formatted text is not the field names, but the field order; so in the example above, you can't write:

```
|{s|=|:abcd|,i|=4|}/* INVALID! */
```

As a consequence, the field names are in fact optional, so you can write:

```
|{|=4|,|=|:abcd|}
```

Please also note that if no value is specified for a field, the field is left as is.

This can be used to set the value for fields in a union. For example, for:

```
union MyUnion { int i; void *p; };
```

a valid format is:

```
|{i|=|,p|=0x0|}
```

The field i won't be set and the field p will be set to NULL.


- Escape sequences
 - Use a || to introduce a | in the message parameters,
 - Use a |. to introduce a carriage return in the message parameters.

Please note the transport structures automatically generated by PragmaDev Studio must be taken into account. So for a message declared via "MESSAGE msg(int, char*);", an example text for the parameters is:

```
|{param1|=12|,param2|=|:my string|}
```

Please refer to the Reference Manual for details on transport structures for messages.

8.5.9.3 Prototyping GUI

This is the easiest way to interact with the system. The interface editor is described in “Prototyping GUI” on page 132. The interface is started with the  quick button.

8.5.9.4 Button windows

Interaction commands can easily be assigned to graphical buttons as described in “interaction commands” on page 244.

8.5.10 Testing

If no process called `RTDS_Env` is defined in the SDL-RT system, one is generated automatically by the code generator to represent the environment. When handling complex messages with the environment it is not handy to define the messages manually. The easiest way is to define a test process or block called `RTDS_Env` that will be the testing scenario. Sending messages from the PragmaDev Debugger is a good way to trigger specific test scenarios.

8.5.11 Code coverage



The debugger’s *Get code coverage* button gets the code coverage analysis results for the running system so far. This feature is available only if the *Gen. code coverage info.* is checked in the generation options (see “Profiles” on page 157).

For more details on code coverage results, see “Code coverage results” on page 325.

8.5.12 Connecting an external tool

It is possible to connect an external tool to the PragmaDev Debugger through a socket. To allow connections to the PragmaDev Debugger the `connect` command should be entered in the shell:

```
connect <port number>
```

The IP address used is the IP address of the host where the PragmaDev Debugger is running. Only the port number can be configured.

The PragmaDev Debugger is seen as a server so the `connect` command should be executed before starting the client.

Once the connection is made the client has basically a direct access to the shell commands: whatever is sent goes to the PragmaDev Studio shell and whatever the shell replies goes to the socket. Therefore the syntax is the one used in the shell. Note the external tool connected will also receive any information that is printed out in the shell.

To close the socket use the `disconnect` command in the PragmaDev Studio shell.

Here is a sample code in Python (<http://www.python.org>) that connects to port 50010:

First start the server in the PragmaDev Studio shell:

```
connect 50010
```

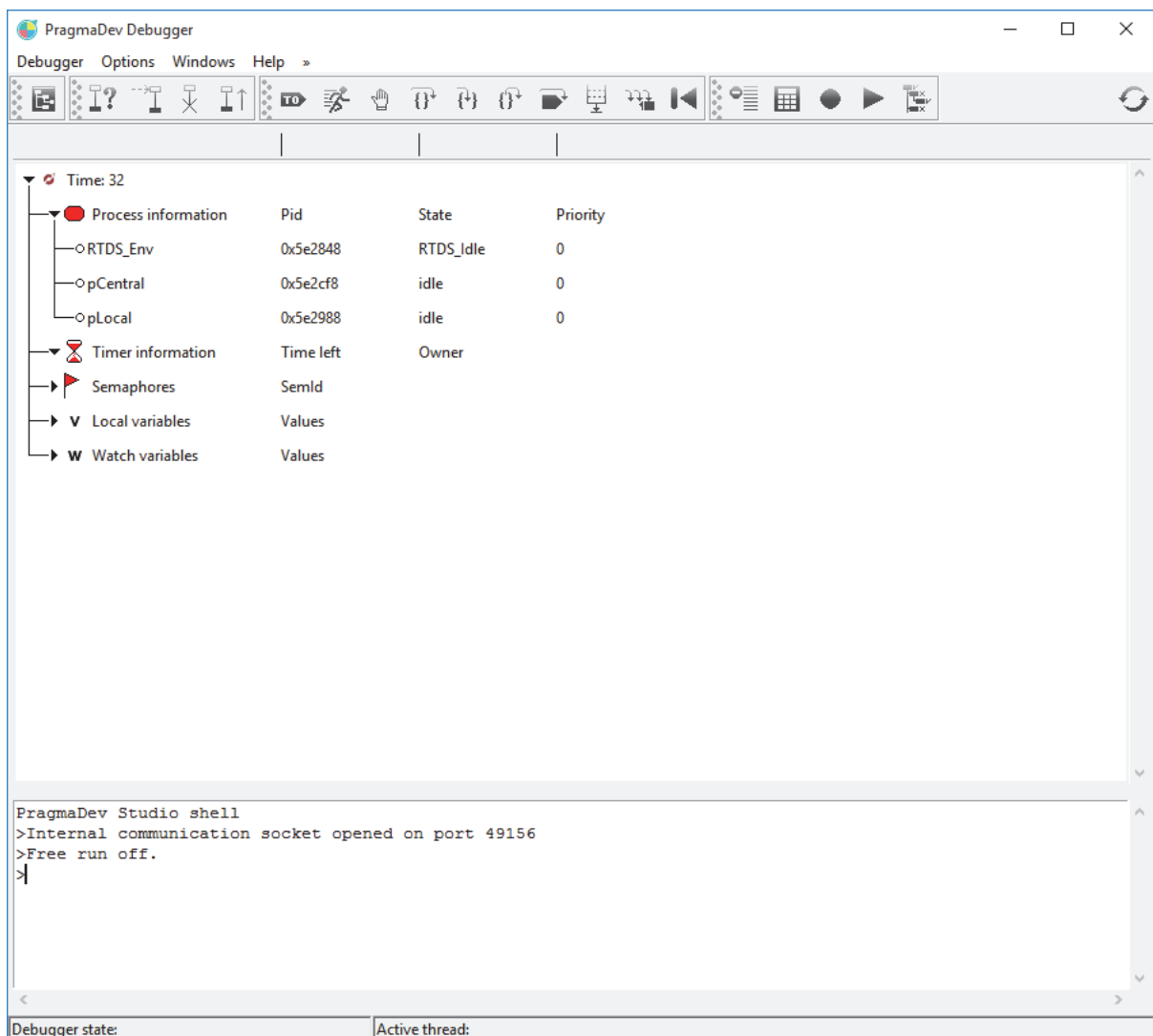
Then go to a shell or DOS window and type:

```
python
>> from socket import *
>> s=socket(AF_INET, SOCK_STREAM)
>> s.connect((gethostname(), 50010))
>> s.send('help\n')
>> print s.recv(500)
```

It will print out the 500 first characters of the PragmaDev Debugger help and display it in the shell.

8.5.13 Debugger tree view

An alternate view of the SDL-RT system information is available displayed as a Tree. It is possible to switch from one to the other during a debug session through the “Windows / Change to tree debugger window” menu. This view is interesting because it is more compact than the classical one.



It is possible to expand or collapse part of the information as well as drag and drop to re-order the tree. The “Windows / Change to classical debugger window” menu item returns to the classic display of the PragmaDev Debugger.

8.5.14 Command line debug

The PragmaDev Debugger can be started from a shell or a DOS console and run an execution script automatically with the `rtdsSimulate` command. Check the Reference manual for more information.

9 - SDL Z.100 project

9.1 - SDL types and data declarations

The types and data declarations in SDL projects are defined in the ITU-T recommendation Z.100. The supported version is SDL-92 some restrictions or additions described in the following paragraphs.

All declarations are made in standard text boxes:



The dashed text-box used for SDL-RT declarations in SDL-RT projects is not used.

9.1.1 General restrictions

The following SDL-92 features are not supported in PragmaDev Studio:

- Declarations referencing each other will not work. For example:

```
/* Uses synonym toto_dflt as default value */  
NEWTYPE toto  
STRUCT  
    i INTEGER;  
    s CHARSTRING;  
DEFAULT toto_dflt;  
ENDNEWTYPE;  
  
/* Uses type toto for synonym type */  
SYNONYM toto_dflt toto = (. 0, 'xxx' .);  
will not work.
```
- Qualifiers in identifiers (<<qualifier>> name) are not supported.
- Optional definitions via SELECT are not supported.
- Context parameters are not supported.

9.1.2 Pre-defined sorts

The following pre-defined sorts are available:

- Boolean
- Integer
- Natural
- Real
- Character
- CharString
- Time

- Duration
- Pid

Literal names for control characters such as NUL, STX or DEL are not supported. These characters must be created via the Num standard operator with the corresponding ASCII code.

All standard operators are available, with the following additions and restrictions:

- The operations available on the CharString sort are also available on the Character sort. So the expression:
`s := 'foo' // 'o'`
is valid (in strict SDL-92, this should be written:
`s := 'foo' // MkString('o')`).
The standard MkString operator is however still supported.
- The internal operators (operators with name ending with '!') are not supported.

Here is a complete list of supported operators for all pre-defined sorts:

- Num : character -> integer
- Chr : integer -> character
- MkString : character -> charstring
- Length : charstring -> integer
- First : charstring -> character
- Last : charstring -> character
- Substring : charstring, integer, integer -> charstring

9.1.3 NEWTYPE declarations

The standard SDL-92 NEWTYPE declaration is supported, with the following additions and restrictions:

- By default, the scope for NEWTYPE's is global: having two types with the same name in two different agents will not work. To get regular SDL behavior, the option "Manage all types in a single system-wide scope" in the project generation options must be unchecked.
- Inheritance is not supported.
- Choice sorts are supported. These can be written either in SDL-2000 syntax:
`NEWTYPE MyChoiceType
CHOICE
 field1 Type1;
 field2 Type2;
ENDNEWTYPE;`
or in ObjectGeode syntax:
`NEWTYPE MyChoiceType
CHOICE {
 field1Type1,
 field2Type2
}`

ENDNEWTYPE ;

ObjectGeode syntax will however issue a warning when used since it is non-standard.

- In STRUCT or CHOICE types defined via the SDL-92 syntax, the ';' after the list of fields is mandatory.
- Optional fields in STRUCT types are supported: for each optional field *x* in a STRUCT, a read-only boolean pseudo-field *xPresent* is added, indicating whether *x* is present or not. A field is set present when a value is assigned to it. There is no way of setting back a present field 'non present'. The notation for STRUCT initializers with missing fields '(. *x* , , *z* .)' is not supported.
- The available pre-defined generators are:
 - Array with the standard notation for array initialization '(. *x* .)' and element access for reading and writing `my_array(index)`;
 - PowerSet with the standard operators `incl`, `del`, `take` and `length`;
 - String with the standard operators `mkstring`, `length` and `//`. Operators `first`, `last` and `substring` are only available for the CharString sort.
- User-defined generators are not supported.
- LITERALS in a NEWTYPE can only be used alone. Mixing literals with STRUCT or CHOICE or a generator will issue a syntax error.
- The CONSTANTS clause in a NEWTYPE is not supported.
- Ordered literal types (via OPERATORS ORDERING) are not supported.
- Operator diagrams are not supported, as well as textual operator declarations.
- Operators defined without parameters are supported, but will generate a warning. Calls to these operators may be written `operator ()` or just `operator`.
- Polymorphism for operators is not supported, i.e. two operators cannot have the same name, even if their parameter types are different.
- Quoted infix operator names are not supported ("+", "*", etc.)
- Operators declared EXTERNAL are not supported.
- Extended literal or operator names are not supported.
- AXIOMS are not supported.
- Indices for ARRAY type can only be of basic types. Indices of complex types such as STRUCT or CHOICE are not supported.

9.1.4 SYNTYPE declarations

The standard SDL-92 SYNTYPE declaration is supported, with the following additions and restrictions:

- The scope for SYNTYPE's is managed as the one for NEWTYPE's: global by default, hierarchical on option (see "NEWTYPE declarations" on page 258).
- The closed ranges in CONSTANTS clauses may be written either `min:max` (standard), or `min..max`.

- ObjectGeode syntax for SIZE constraints is supported, so it's possible to write:
`SYNTYPE MyStringType = CharString(SIZE(0:16)) ENDSYNTYPE;`
instead of:
`SYNTYPE MyStringType = CharString SIZE(0:16) ENDSYNTYPE;`
The first syntax will however issue a warning since it is non-standard.

9.1.5 SYNONYM declarations

The standard SDL-92 SYNONYM declaration is supported, with the following restrictions:

- The scope for SYNONYM's is managed as the one for NEWTYPE's: global by default, hierarchical on option (see "NEWTYPE declarations" on page 258).
- A synonym for a structure or array primary (.) will not work if the type is not specified. So the following SYNONYM declaration:
`SYNONYM myValue = (. 'xxx', 0 .);`
is *invalid* and must be re-written:
`SYNONYM myValue MyStructType = (. 'xxx', 0 .);`
- External synonyms are not supported.

9.1.6 FPAR & RETURNS declarations

The standard SDL-92 FPAR declaration is supported with no restriction in process, process type and procedure diagrams, as well as the RETURNS declaration in procedures; RETURNS declarations with a variable name are not supported.

These two declarations must however be placed alone in a text box, preferably together if both are present.

9.1.7 TIMER declarations

The standard SDL-92 TIMER declaration is supported with the following restriction: timer parameters are not supported. The unit for the timer duration is seconds.

Please note that the declaration for a timer is not mandatory: if a SET with a time-out value is present in a process or procedure, the timer is automatically declared. The declaration must however be done if the timer is started using a SET without time-out value.

9.1.8 SIGNAL & SIGNALLIST declarations

The standard SDL-92 SIGNAL and SIGNALLIST declarations are supported with the following restrictions:

- Signal and signal list names are case-sensitive.
- Signal inheritance is not supported.
- Refining signals to sub-signals is not supported.

9.1.9 SIGNALSET declarations

SIGNALSET declarations are not needed and not supported in PragmaDev Studio.

9.1.10 USE declarations

The USE declarations have today the semantics it has in SDL-RT projects, which means:

- USE clauses may appear at all levels in the architecture.
- Only one USE clause is accepted per diagram.
- USE clauses are inherited only in child diagrams that do not have their own USE clause. If they have one, it will shadow their parent's USE clause.
- Specifying imported items in USE clauses is not supported (e.g. "USE MyPackage / NEWTYPE MyType" is invalid)

USE clauses may not be mixed with other declarations in a text box.

Please note these differences with the standard are considered when exporting to a PR file: all USE clauses are inserted at system level, in conformance with the Z100 PR format.

9.1.11 INHERITS declaration

The standard SDL-92 INHERITS declaration is supported in process type diagrams. Inheritance is not available for block types. This declaration must be placed alone in a text box.

9.1.12 Data declarations (DCL)

The standard SDL-92 DCL declaration is supported with no restrictions. Remote variables are not available, so the REMOTE, IMPORTED, EXPORT and IMPORT declarations are not supported.

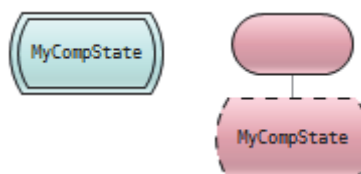
9.1.13 Structural element declarations

All standard SDL-92 structural element declarations are supported, except system types, procedure types, and service types, and with the addition of the SDL-2000 composite state:

- System
- Block
- Process
- Procedure
- Package
- Block type (named *block class* in PragmaDev Studio)
- Process type (named *process class* in PragmaDev Studio)
- Composite state
- Service
- Macro

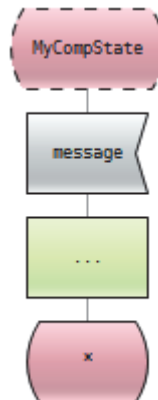
The following restrictions apply:

- Names for these elements are case-sensitive.
- Process types are only allowed in packages and cannot be defined in systems, blocks or block types.
- Inheritance in block types is not supported, so VIRTUAL, REDEFINED or FINALIZED processes or process types in block types are not needed and not supported.
- Inheritance for procedures is not supported.
- Remote procedures are not supported.
- Package diagrams are not supported. To define the contents of the package, the following diagrams and files are used:
 - For textual declarations (e.g. types, synonyms, signals, ...), a SDL declaration file (.rdm) is used, similar to the one used in SDL-RT projects.
 - For agent classes declarations, a class diagram is used as in SDL-RT projects (see “Class description” on page 142). In SDL projects, these class diagrams only allow to declare process and block classes.
- Gates in process and block classes are declared as in SDL-RT projects, i.e. in their parent package’s class diagram. SDL-style gate definitions inside class diagrams are not supported.
- Macro diagrams can only contain a single pseudo-transition, starting with a macro inlet. State symbols are not supported in macro diagrams, as well as inputs, priority inputs, continuous signals or saves. Declarations are not supported in macro diagrams, except for the macro formal parameters.
- The graphical representations for SDL-2000 composite states and for SDL-92/96 services have been merged:
 - A composite state never directly contains a state machine. The diagram associated to a composite state always contains one or more “concurrent state machines”, similar to SDL-92/96 services and represented the same way. The channels from the parent process boundary services and between services have also been re-introduced in composite state diagrams.
 - Composite state diagrams cannot define entry points for the state machines it contains (restriction from SDL-2000).
 - Composite states in processes must be declared with the symbol used in SDL-2000 for composite state types, and should be used with the symbol used in SDL-2000 for composite state type instantiation



This allows to graphically distinguish composite states from “normal” states.

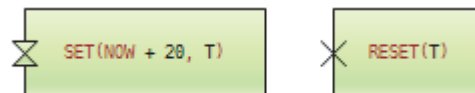
- History states are available with the SDL-2000 syntax:



9.2 - SDL symbols syntax

The syntax for all symbols is compliant with the SDL-92 Z100 recommendation, with the following exceptions:

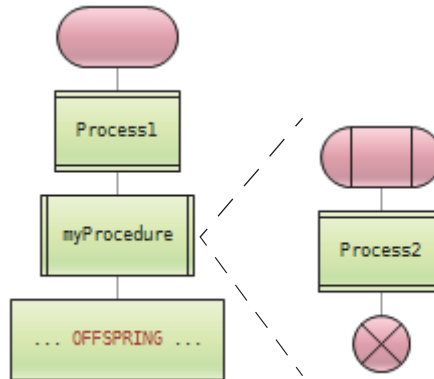
- Virtualities (VIRTUAL, REDEFINED, FINALIZED) in start, input, save and continuous signal symbols are valid, but ignored: all transitions are considered virtual by default.
- For timers, SET and RESET symbols have the shape they have in SDL-RT:



- In signal input and save symbols, NONE or PROCEDURE ... are not supported.
- In signal output symbols:
 - Only one signal can be specified.
 - Specifying the receiver with both T0 and VIA is not supported.
 - VIA may only be followed by a single gate or channel name.
- In expressions, the pseudo-operator ANY or ANY (sort) is not supported.

The pre-defined variables SELF, PARENT, OFFSPRING and SENDER are available in processes, process types and procedures. Please note however that procedures defined in

processes do not use their parent process's SENDER and OFFSPRING variables, but have their own local ones. So for example, in this case:



the OFFSPRING used in the process's task block will be the pid for Process1, not the one for Process2. Note however that the values for the SENDER and OFFSPRING variables are initialized with the values found in the parent process when entering a procedure.

The pre-defined variable THIS in process types is not supported.

9.3 - PragmaDev Simulator

Two ways of debugging a SDL system are available in PragmaDev Studio:

- The system can be transformed to C code and debugged with the same debugger as for SDL-RT;
- The system can be executed within PragmaDev Studio using an internal simulator.

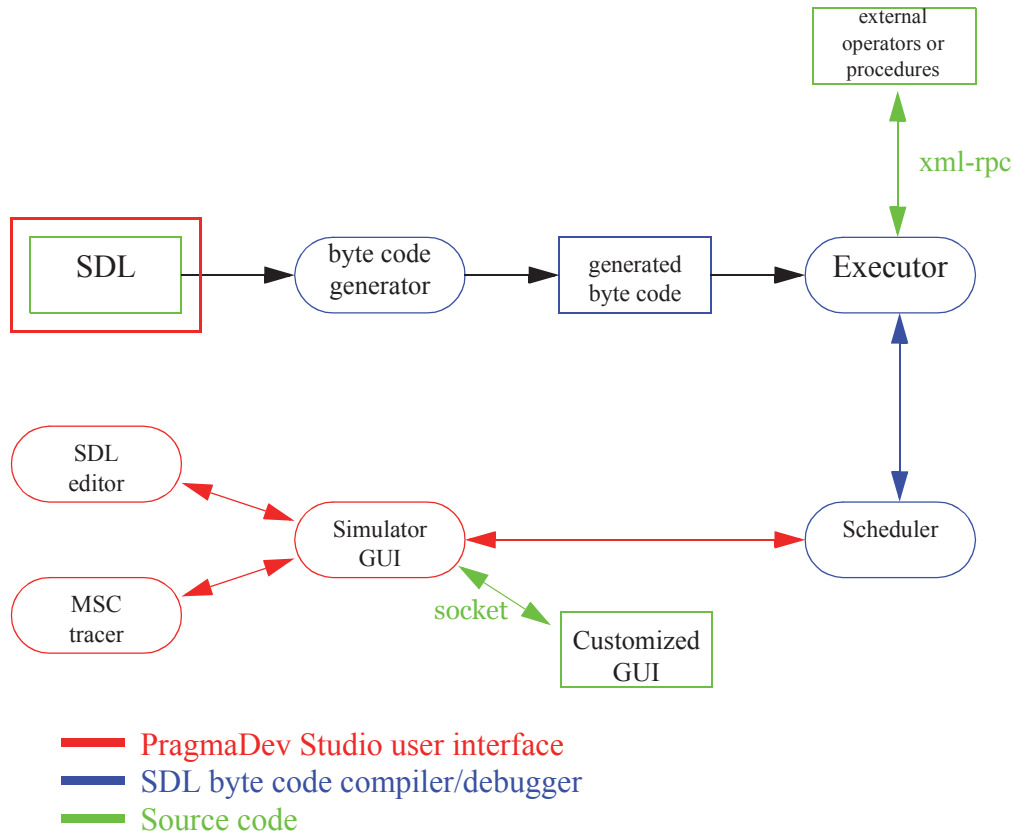
Both approaches have their advantages and drawbacks:

- When exporting the system to C code:
 - The C code is compiled to a native executable, so the execution is quite fast.
 - The supported concepts are limited; the limitations are the same as in SDL to SDL-RT conversion, as described in the corresponding section in the Reference Manual.
 - The execution semantics is not the one described in the Z100 standard, but the one of the underlying RTOS, which can be quite different.
- When using the internal simulator:
 - Since the execution happens within PragmaDev Studio, a far better control of the running system is available.
 - The code in the SDL system is actually interpreted, so the execution is slower than when exporting to C.

The architecture and options for the C code generation approach are exactly the same for SDL and SDL-RT, so they won't be described again here. For details, please refer to the sections "Code generation" on page 155 and "PragmaDev Debugger" on page 229. This section describes the internal PragmaDev Simulator.

9.3.1 Simulator architecture

The PragmaDev Simulator allows you to execute and debug your SDL system. To do so PragmaDev Studio generates a byte code out of the SDL description and executes it.

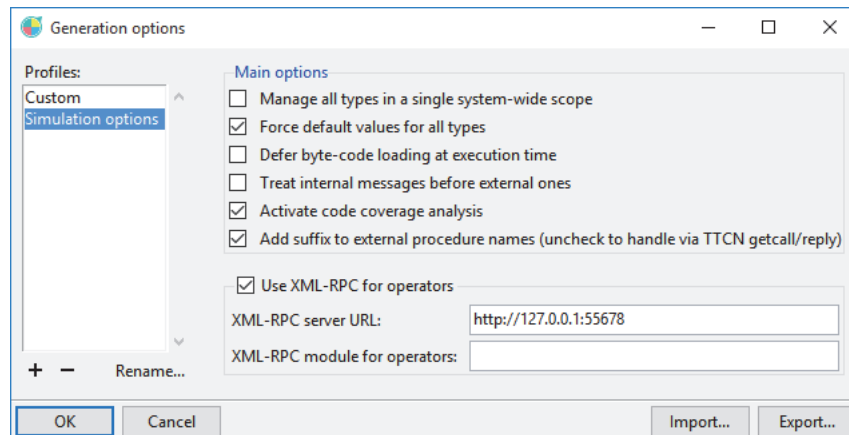


The PragmaDev Simulator has all the expected features of a debugger. It allows you to:

- Graphically trace the internal behavior of the system
- Graphically step in the SDL diagrams
- Visualize all key internals of your system such as:
 - Processes,
 - Timers,
 - Local variables in the current process,
 - Pending messages in the system.
- Send SDL messages to your system,
- Modify SDL state,
- Modify variables value.

9.3.2 Simulator options

A few options are available when simulating the model. They can be configured in the “Generation / Options...” menu:



Please note the same window might contain code generation options that are not described in this paragraph.

The available options are:

- *Manage all types in a single system-wide scope*
SDL declarations have scope. That means the types declared in an agent are only visible in the declaring agent and all its sub-agents. For example two different types with the same name could be declared in two agents at the same level in the architecture. Even though this is supported by PragmaDev Studio it might create confusion or generate major issues when generating code. For that reason it is possible to for a single system level scope to make sure all declarations are unique.
- *Force default values for all types*
All SDL variables will be initialized with a default value if no initialization is done in the model.
- *Defer byte-code loading at execution time*
The process for simulation starts with an internal byte-code generation representing the system to be simulated. This byte code is loaded and is executed by the executor. If the system is very large, loading the byte code might take a long time and consume a lot of memory. This option allows to load the byte code in memory only when it is to be executed.
- *Treat internal messages before external ones*
By default, internal and external messages all end up in the same unique system queue. The order of execution is the order of the messages in the queue: first in first out. This option allows to execute first all messages exchanged internally in the system, and when all internal messages have been executed the messages coming from the environment. This is to reflect that in a lot of systems messages are exchanged internally much faster than with the environment.
- *Activate code coverage analysis*

In order to optimize performance and memory consumption, code coverage information is not handled during simulation by default. It is necessary to activate this option to retrieve it.


- *Add suffix to external procedure names*

With this option checked it is possible to “catch” external procedure calls in TTCN tests via getcall/reply. The drawback being that SDL operators are handled the same way, so it is not possible to have an external procedure and an operator with the same name.

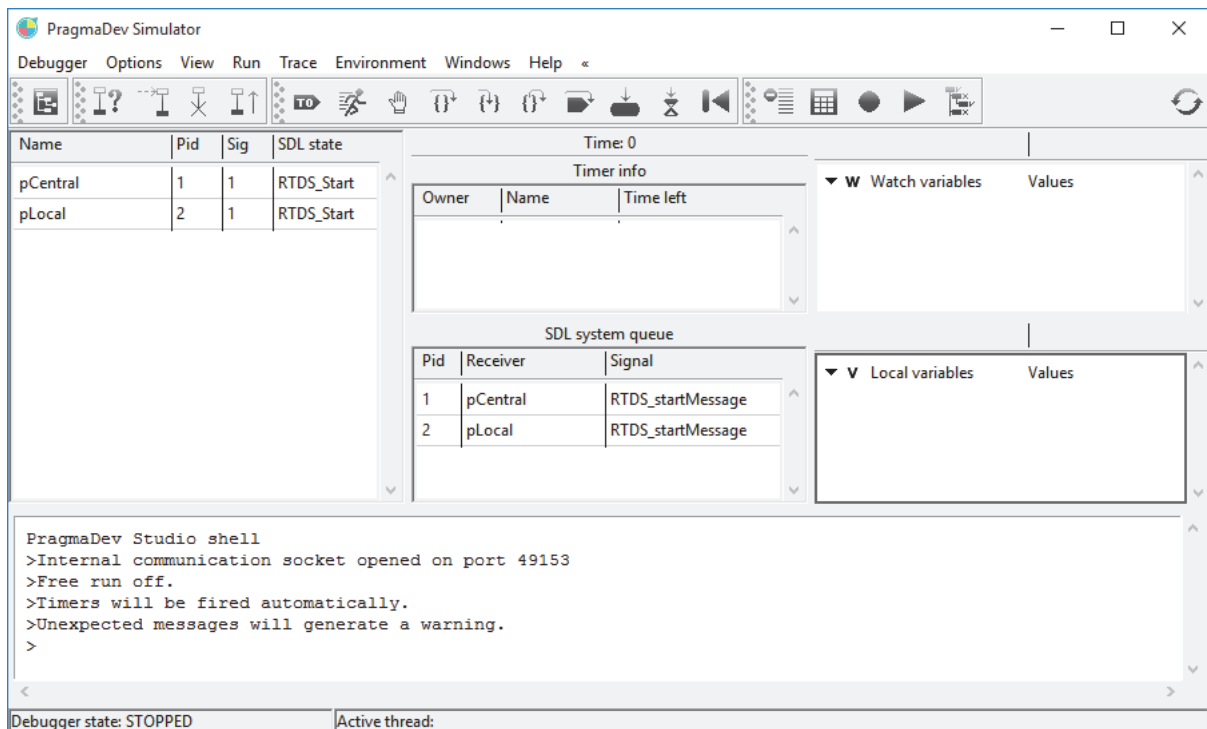
- *Use XML-RPC for operators*

During simulation when undefined operators or external procedures are called, a window will pop up to ask for the return value of the operator or the external procedure. It is also possible to call an implementation of the operator or the external procedure via XML-RPC. XML-RPC has a standardized way of formatting and communicating so that the XML-RPC server can call any type of implementation such as C, Java, Perl... The first option identifies where the XML-RPC server is (IP address or host name) and on which port it communicates. The second option identifies an optional module in which the operator or procedure is actually implemented. More information can be found in “Communication with an external XML-RPC server” on page 291.

9.3.3 Launching the PragmaDev Simulator

The PragmaDev Simulator is started from the “Generation / Execute” menu or from the  quick button.

Byte code is generated out of the SDL description and the simulation environment is started in the background. The PragmaDev Simulator window is started automatically and you are ready to debug your system.



The PragmaDev Simulator window




All static processes are already present in the Process information list and each static process has its Start message pending in the SDL system queue.



The PragmaDev Simulator can be restarted at any time with the reset button or shell command. The underlying simulation environment is restarted and cleaned up.

9.3.4 Stepping levels

Since your source code is a composite of graphical SDL symbols and textual SDL lines of code, the PragmaDev Simulator offers several ways to execute the code:

- Run with SDL key events trace information,
 Menu “Options / Free run” de-activated. This is the default setup where the PragmaDev Simulator traces all SDL key events and displays textual and / or SDL and / or MSC traces.
- Run without SDL key events trace information,
 Menu “Options / Free run” activated. When this option is activated the system runs freely and no trace information is printed.
- Stop execution,
 Stops execution of the running system.

- Stepping



Step line by line in the SDL code,



Step-out of an SDL procedure function,



Step-in a SDL procedure.

- Step until the next SDL key event such as:



- Message sending,
- Message received,
- Timer started,
- Timer cancelled,
- Timer went off,
- SDL state modification,
- SDL process created,
- SDL process deleted.

- Run until the end of the transition,



- Run until all signals are consumed except timers.

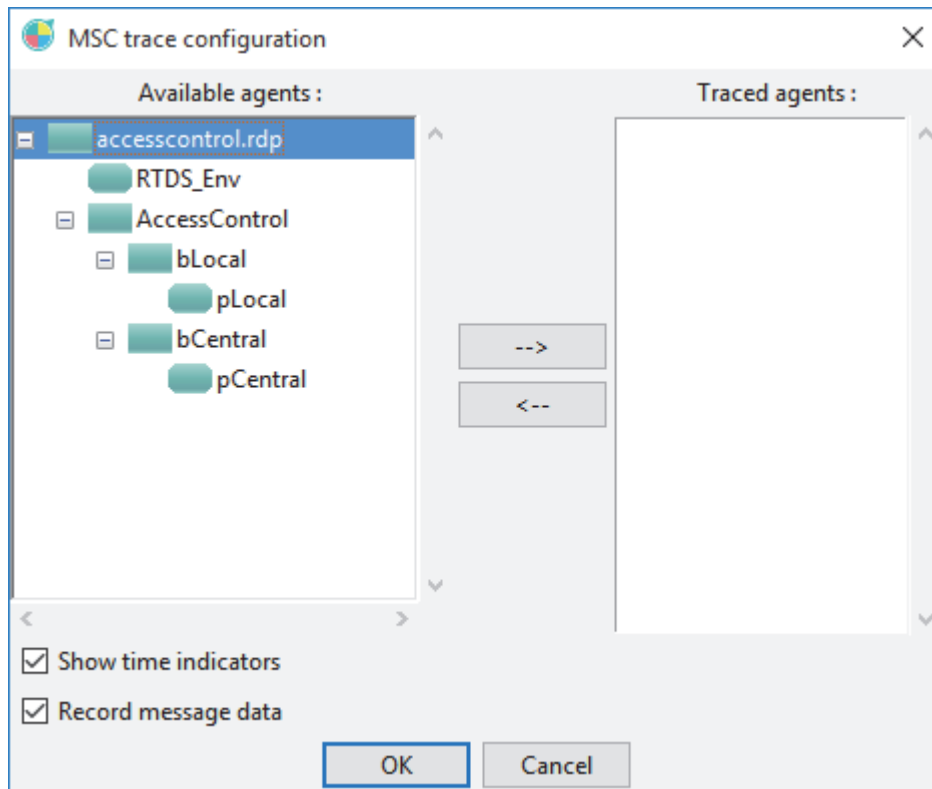


9.3.5 MSC trace




The MSC Tracer allows you to graphically trace execution of the system with its SDL key events. It is possible to configure the MSC trace to define at which level of details the architecture of the system should be represented. The MSC trace can be made at system, block, process or any combination of agents. Any agent selected will be represented by a lifeline in the MSC diagram. Any messages exchanged inside the agent will not be seen on the MSC. The default view is the most detailed one, with a lifeline for each process.

- Configure the MSC trace

The  quick button opens the *MSC trace configuration* window:



The following options are available:

- Show system time information,
- Record and display message parameters,
- SDL architecture elements to trace.
- Start the MSC trace
 - The  quick button starts the MSC Tracer. By default the trace is active.
- Stop the MSC trace
 - The  quick button stops the MSC Tracer.
- Trace the last SDL events (backTrace)
 - The  quick button opens a MSC Tracer and displays the last SDL events. The number of logged events is between 50 to 100.

9.3.6 Displayed information

The PragmaDev Simulator window is divided in 5 parts described below.



If needed, the displays can be refreshed at any time with the refresh button or shell command.

The information to refresh can be setup in the “Options / Refresh options...” menu as explained in “Refresh options” on page 278.

9.3.6.1 Processes

The *Process information* part list all processes defined in the SDL system. The displayed information is:

- **Name**
This field displays the name of the process as defined in the Process create SDL symbol. Several tasks can have the same name. The Pid should then be used to distinguish them.
- **Pid**
This field shows the unique internal Process Identifier of the process.
- **Sig**
This field shows the number of signals waiting in the process queue.
- **SDL state**
This field is the internal SDL state of the SDL process as defined in the SDL diagram. The RTDS_Start signal is a signal used to execute the start transition of the process.

When the system is running the active process line is printed in red.

Name	Pid	Sig	SDL state
pCentral	1	0	idle
pLocal	2	0	idle

Process information window

To distinguish processes with the same name but in a different block, a tool tip shows up when the cursor is over the process name and displays the full architecture path down to the process:

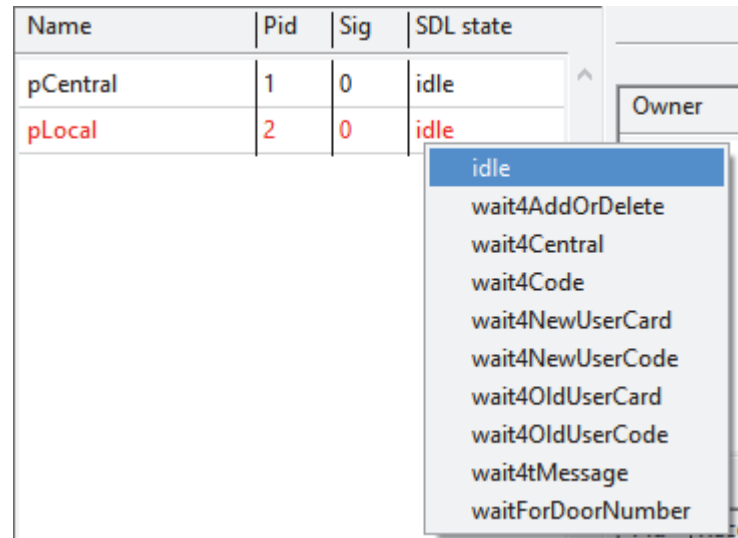
Name	Pid	Sig	SDL state
pCentral	1	0	idle
pLocal	2	0	idle

AccessControl:bLocal:pLocal

Double-clicking on a process name will also open the corresponding diagram in an editor window.

The *Process information* window also allows to modify the SDL state of a process. To do so right click on the SDL state column of the process line. A pop up menu will list all the

available SDL state that have been defined in the system. Select one and the SDL state is modified.



9.3.6.2 System queue

PragmaDev Simulator handles all pending signals in a single system queue.

SDL system queue		
Pid	Receiver	Signal
1	pCentral	RTDS_startMessage
2	pLocal	RTDS_startMessage

The displayed information is:

- **Pid**
This field shows the unique internal Process Identifier of the receiver process of the pending signal.
- **Receiver**
This field displays the name of the receiver process of the pending signal.
- **Signal**
This field shows the name of the pending signal.

That allows to:

- Execute the signal inputs in the order the signals have been sent.

- Re-order the pending signals.

That is a key feature of the PragmaDev Simulator since it makes the process scheduling indeterministic allowing full system validation whatever the ordering is.

The signal on the top is the next to be executed. Double click on a signal in order to put it up front in the system queue:

SDL system queue		
Pid	Receiver	Signal
1	pCentral	RTDS_startMessage
2	pLocal	RTDS_startMessage

SDL system queue		
Pid	Receiver	Signal
2	pLocal	RTDS_startMessage
1	pCentral	RTDS_startMessage

System queue re-organization example

When signals are saved in an instance, the saved signal will also appear in the system queue, just after the first signal that will be received by the instance that saved it, or at the end of the queue if there is no such signal. To indicate that the signal is a saved one, it will be displayed in italics and its name will be surrounded with slashes:

SDL system queue		
Pid	Receiver	Signal
1	p1	start2
1	p1	/start1/

Such messages cannot be put to the top, so double-clicking on them will display an error message in the PragmaDev Studio shell and do nothing.

9.3.6.3 Timers

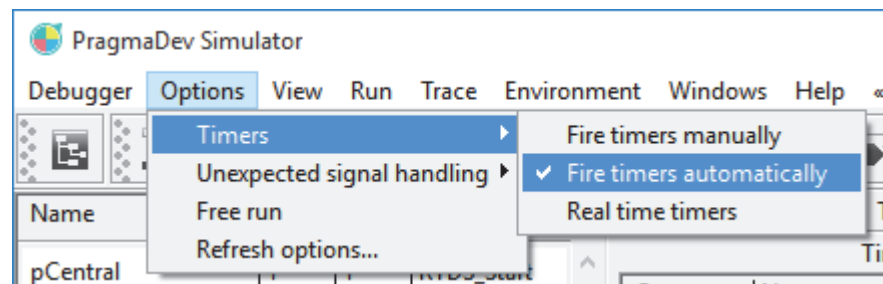
The *Timer info* part displays all on-going timers started from the SDL design. The displayed fields are:

- Pid
Identifier of the process that started the timer.
- Name
Name of the timer as defined in the SDL design.
- Time left

Time left before the timer goes off.

SDL semantic specifies a transition takes no time, so system time does not increase unless a timer goes off or a new system time value is set. However, this is not the case when performance simulation is enabled (see “SDL Z.100 performance simulation” on page 303).

There are 3 ways to manipulate timers in the PragmaDev Simulator. Selection is done through the “Options / Timers” menu:



Timer handling selection menu

- **Fire timers manually**
In that mode, once all signals in the system have been executed, the system hangs. To make a timer go off, double click on the timer line. System time will increase by the value of the timer’s time left and all other timers with a value less or equal will also be fired.
- **Fire timers automatically**
In that mode, once all signals in the system have been executed, the internal scheduler will automatically fire the first timers in the list and increase system by the timer left value.
- **Real time timers**
In that mode, once all signals in the system have been executed, a timer thread is started that generates a timer tick every second. When the timer tick is received, the system time value is increased by one. When the time left value of the first timer reaches 0 (zero), the timer is fired. That implicitly means the delay expressed when starting a timer is set in seconds in that specific mode.

When the cursor is over an owner in the timer list, a tool tip indicates the architecture path down to the timer receiver.

Timer info		
Owner	Name	Time left
2	codeTimer	5
AccessControl:bLocal:pLocal		

9.3.6.4 Watch

There are several ways to add a variable in the *Watch window*:

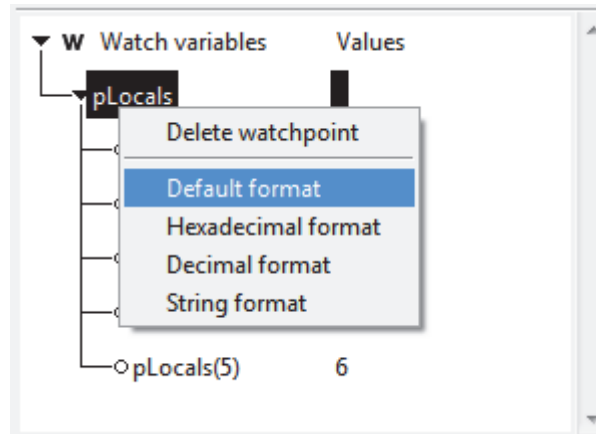
- From the shell
Type the following command in the shell:

`watch add <variable name>`

- From the SDL editor
Select an expression in the SDL editor and go to the “Debug / Add watch” menu to add the expression in the *Watch window*.

Variables can be removed from the *Watch window*:

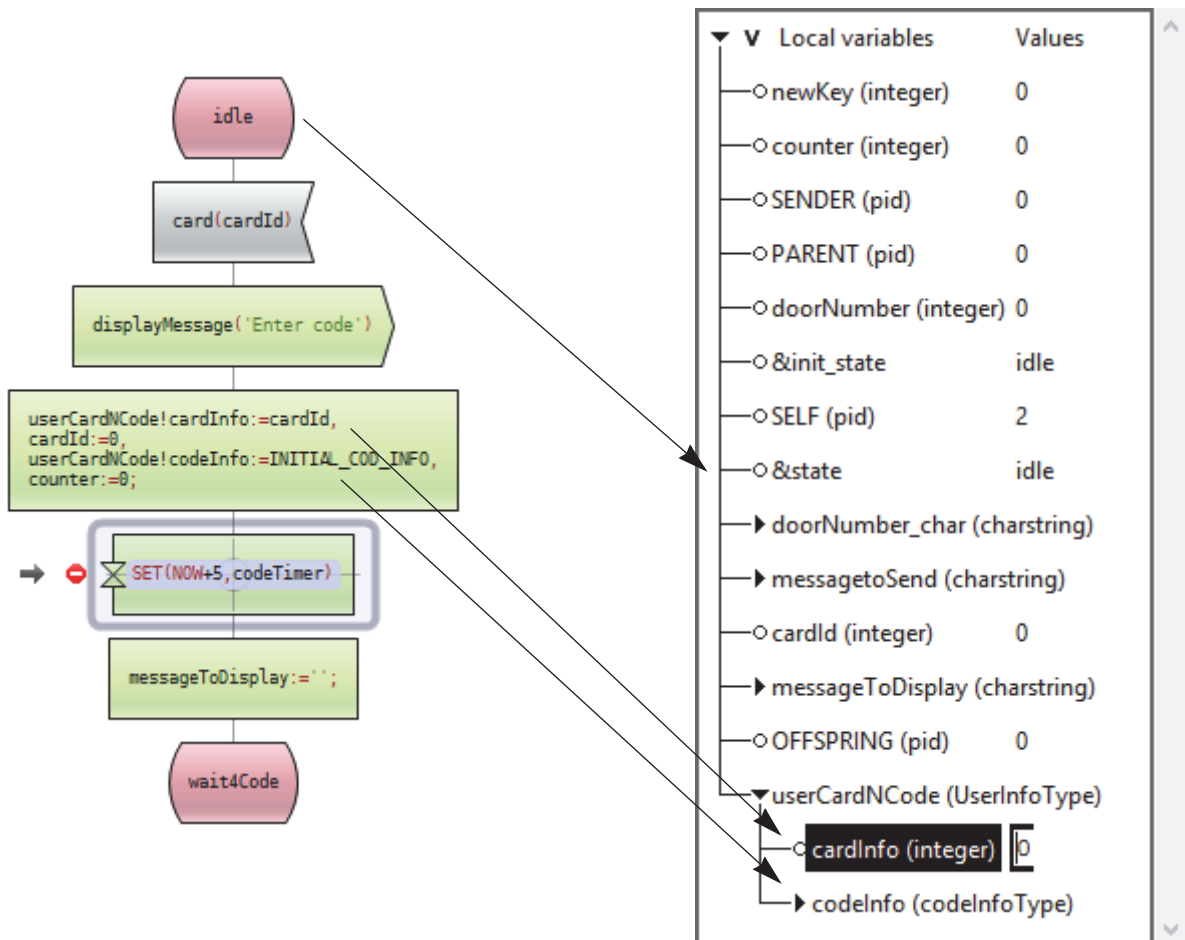
- from the shell with the following command:
`watch del <variable name>`
- from the *Watch window* with right mouse button as shown below:



The *Watch window* also allows to modify the value of variables. To do so select the value of the variable, and double click on it to edit the value. Press <Return> and the value is updated!

9.3.6.5 Local variables

When stepping through the code the PragmaDev Simulator automatically displays the local variables of the current process:



Local variables example

Some key variables are always present:

- OFFSPRING
SDL keyword indicating the pid of the last process created dynamically within the current process; 0 if none have been created.
- SELF
SDL keyword indicating the current process pid.
- PARENT
SDL keyword indicating the pid of the process that created the current process; 0 if the process was created statically at startup.
- SENDER
SDL keyword indicating the pid of the sender of the last received message.
- &state
Internal variable representing the current state of the process.

The *Local variables* window also allows to modify the value of variables. To do so double click on the variable value. Press <Return> and the value is updated.

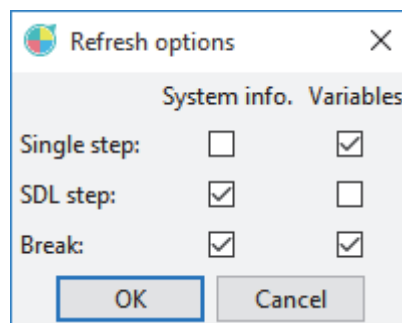
9.3.6.6 Refresh options

The information displayed in the PragmaDev Simulator windows are divided in 2 categories:





- System info
 - Process information
 - Timer information
 - Semaphore information
- Variables
 - Local variables
 - Watch variables

Retrieving any information from the target is time consuming. In order to optimize the response time it is possible to configure which category of information is refreshed.

The configuration is done in the “Options / Refresh options...” menu:



Default Refresh options

- Single step means the use of one of the following step button:   ,
In the default options, only the Variables category is refreshed since there is no reason the System information category has changed in the meantime.
- SDL step means the use of the  step button,
When stepping from an SDL event to another, only the System information category is interesting to update.
- Break means the system has hit a breakpoint.
In that case it is recommended to update all the information.

Anyway, at any time it is possible to refresh all information: 

9.3.7 Shell

The PragmaDev Studio shell allows to enter all commands listed above and is used as a textual trace.

The available commands are grouped in categories. To list all the available categories type:
help

It will list the following categories:

Type help followed by a category to list available commands

```
-----  
shell  
execution  
interaction  
variables  
trace  
customization
```

Type help followed by a category name to list the corresponding commands.

To list all the available commands, type:

h

It will list the following commands:

Command	- Explanation
h	- lists all commands
history	- list the last entered valid commands
clear	- clears the shell
echo <string>	- echos a string in the shell
include <file name>	
resume	- resumes the scenario
repeat <repeat count> <shell command> [; <shell command>]*	
# <comment>	
! <any host command>	
refresh	- refreshes all data in the window
run	- runs the SDL system
stop	- stops the SDL system
step	- step in the code
stepin	- step in function calls
stepout	- step out a function call
keySdlStep	- run until the next key SDL event
sdlTransition	- run until the end of the SDL transition
runUntilTimer	- run all transitions until timers
resetSystem	- resets the running system
list	- list breakpoints
watch add [<pid>:]<variable name>[<field separator><field name>]*	
watch del [<pid>:]<variable name>[<field separator><field name>]*	
break <break condition> [<ignoreCount> <volatile>]	
delete <breakPoint number>	
db <any debugger command>	
set time <new time value>	
send2name <sender name> <receiver name> <signal number or name> [<parameters>]	
send2pid <sender pid> <receiver pid> <signal number or name> [<parameters>]	
sendVia <sender pid> <channel or gate name> <signal number or name> [<parameters>]	
send <sender pid> <signal number or name> [<parameters>]	
systemQueueSetNextReceiverName <receiver name>	
systemQueueSetNextReceiverId <receiver id>	
interruptCall <procedure name>	
extractCoverage <file name>	
connect <port number>	
connectxml <port number>	
disconnect	
varFromType <variable name> <variable type>	
varFromValue <variable name> = <initial value>	
varFieldSet <variable name>[.<field name>]* = <field value>	
dataTypes <on off>	
print <variable name>	
sdlVarSet [<process id>:]<variable name>=<value>	
sdlVarGet [<process id>:]<variable name>=<value>	

```
backTrace      - display last events traced when activated in profile
setupMscTrace <time information> <message parameters> [<agents>]
startMscTrace
stopMscTrace
saveMscTrace <file name>
setEnvInterfaceFilter 1|0
buttonWindowCreate <button window name>
buttonWindowAdd <button window name> <button name> = <shell command> [||; <shell command>]*
buttonWindowDel <button window name> <button name>
buttonWindowLabelAdd <button window name> <label name>
buttonWindowLabelDel <button window name> <label name>
```

In any of the shell commands the following can be used:

```

|$(<os environment variable>) to access an operating system environment variable
|${<interactive label>} pops up an interactive window to get variable value, /s, /b and others can
be used
|${<shell variable name>} will be replaced by the shell variable value
|${<process name>:<instance number>} will be replaced by the pid of the instance of the process
& <any command> will prevent the above pre-processing
<partial command>\ and continue the command on the next line of the shell

```

The last valid commands can be recalled with the upper arrow.

Some of these commands are the equivalent to buttons in the button bar. Some are specific to the shell and will be further explained below.

9.3.7.1 shell commands

To list all the available commands in this category, type:

```
help shell
```

It will list the following commands:

Command	- Explanation

h	- lists all commands
history	- list the last entered valid commands
clear	- clears the shell
echo <string>	- echos a string in the shell
include <file name>	- run a scenario of commands out of a file
resume	- resumes the scenario
repeat <repeat count> <shell command> [; <shell command>]*	- repeat a set of shell commands
# <comment>	- does nothing
! <any host command>	- runs any host command

In any of the shell commands the following can be used:

```

|$(<os environment variable>) to access an operating system environment variable
|${<interactive label>} pops up an interactive window to get variable value, /s, /b and others can
be used
|${<shell variable name>} will be replaced by the shell variable value
|${<process name>:<instance number>} will be replaced by the pid of the instance of the process
& <any command> will prevent the above pre-processing
<partial command>\ and continue the command on the next line of the shell

```

- Running scenarios

A set of commands can be saved to a script file with the red circle button in the tool bar. The include command or the play button allows to run a script file. The script file is stopped when a breakpoint is hit or when the stop button is pressed. Type the resume command to resume the scenario.

- Process instances pid

It is possible to get a process instance pid with the `|$< <process name> >` syntax.

Example:

In the following configuration:

Name	Pid	Sig	SDL state
ping	1	0	idle
pong	2	0	idle

```
echo |$<pong:0>
```

echos the pid of the first instance of pong:

2

Note:

This feature does not work if the “Options / Free run” is activated.

- Environment variables

Operating system environment variables can be accessed with the `|$(<variable name>)` syntax.

Example:

```
echo |$(RTDS_HOME)
```

echos:

C:\RTDS

9.3.7.2 execution commands

To list all the available commands in this category, type:

help execution

It will list the following commands:

```
Command      - Explanation
-----
refresh      - refreshes all data in the window
run          - runs the SDL system
stop         - stops the SDL system
step         - step in the code
stepin       - step in function calls
stepout      - step out a function call
keySdlStep   - run until the next key SDL event
sdlTransition - run until the end of the SDL transition
runUntilTimer - run all transitions until timers
resetSystem  - resets the running system
list         - list breakpoints
watch add [<pid>:<variable name>[<field separator><field name>]*
  adds a variable to watch:
  <pid> is the process id in which the variable is. Only available in Z.100 simulation.
  <variable name> is the name of the variable
  <field separator> is '!' in SDL Z.100 or '.' in SDL-RT
  <field name> is the name of the variable field or sub-field
watch del [<pid>:<variable name>[<field separator><field name>]*
  remove a variable to watch
  <pid> is the process id in which the variable is. Only available in Z.100 simulation.
  <variable name> is the name of the variable
  <field separator> is '!' in SDL Z.100 or '.' in SDL-RT
  <field name> is the name of the variable field or sub-field
break <break condition> [<ignoreCount> <volatile>]
break condition is a function name or '*'break-address or file-name':line-number or
  diagram-file-name':symbol-id':line-number
ignoreCount is a number
```

volatile is a boolean: 'true' or 'false'
 delete <breakPoint number>
 db <any debugger command>
 the command is directly sent to the debugger with no verification

- db
 This command is not effective in SDL Z.100 simulation.

9.3.7.3 interaction commands

To list all the available commands in this category, type:
 help interaction

It will list the following commands:

Command	- Explanation

set time <new time value>	new time value can be absolute time or '+'delta
send2name <sender name> <receiver name> <signal number or name> [<parameters>]	
send2pid <sender pid> <receiver pid> <signal number or name> [<parameters>]	
sendVia <sender pid> <channel or gate name> <signal number or name> [<parameters>]	
send <sender pid> <signal number or name> [<parameters>]	environment name is 'RTDS_Env' and environment pid is '-1'
	parameters are {field1 =value ,field2 =value ,... }
systemQueueSetNextReceiverName <receiver name>	
systemQueueSetNextReceiverId <receiver id>	
interruptCall <procedure name>	calls the procedure with the given name immediatly. The procedure must be known in the current context. Usually used in conjunction with a breakpoint to simulate an interrupt handler called in the middle of a transition.
extractCoverage <file name>	
connect <port number>	to connect to an external tool on a socket using the shell format
connectxml <port number>	to connect to an external tool on a socket using the xml-rpc format
disconnect	to disconnect from the external tool

- set time
 This command sets a new system time value if the debugger allows it. Please check the PragmaDev Studio Reference Manual for more information.
- connect
 This command opens a socket in server mode to connect an external tool to the PragmaDev Studio shell. The parameter is the port number on the host IP address. This command should be done before starting the client.
- disconnect
 Disconnect the socket from the external tool.
- System queue manipulation
 It is possible to re-organize the system queue order from the shell. The system-QueueSetNextReceiverName will put up front in the system the next message for the defined receiver name, and systemQueueSetNextReceiverId will put up front in the system queue the next message for the defined receiver pid.
- interruptCall
 Calls a procedure in the current execution context immediately. This command can be used in conjunction with a breakpoint to simulate the call of an interrupt

handler in the middle of a transition: the breakpoint should set where the call should occur, and the command issued after the execution has stopped. The procedure must be visible in the current context. The semantics of the call will respect the SDL semantics: if the procedure modifies a variable in its parent, the variable will actually be modified.

This command is only available in the PragmaDev Simulator.

- **extractCoverage**
 Extracts the code coverage for the current debug session so far and stores it in the specified file. If the file name is relative, it will be taken from the project directory. Please note that if this command is used in a debug session run via the `rtDsSimulate` command line utility, the project will be saved in the end and the code coverage results stored in it.

9.3.7.4 variables commands

To list all the available commands in this category, type:
`help variables`

It will list the following commands:

Command	- Explanation

<code>varFromType <variable name> <variable type></code>	creates a variable of the given type to be used in the shell
<code>varFromValue <variable name> = <initial value></code>	creates a variable with the given initial value to be used in the shell
<code>varFieldSet <variable name>[.<field name>]* = <field value></code>	sets a single variable field to a given value
<code>dataTypes <on off></code>	prints the type of the variable
<code>print <variable name></code>	prints the variable value
<code>sdlVarSet [<process id>:]<variable name>=<value></code>	
<code>sdlVarGet [<process id>:]<variable name>=<value></code>	

- **shell variables**
 It is possible to define variables in the shell and to use them in `send2xxx` commands using the `$(<variable name>)` syntax.
 - **varFromType**
 This command allows to declare a variable based on a type defined in the SDL system. Only the types used as parameters in messages are available. The message parameters need to be defined in a super-structure in order to be compliant with the executor.

Example:

```
SIGNAL mDummy(t_SubStruct, t_Struct);

NEWTYPE t_SubStruct
STRUCT
  subField1 Real;
  subField2 Boolean;
ENDNEWTYPE;

NEWTYPE t_Struct
STRUCT
  field1 integer;
  field2 charstring;
  field3 t_SubStruct;
OPERATORS
  myEcho: charstring->charstring;
  echo2: integer,charstring, t_SubStruct->charstring;
  myPower: integer->integer;
  myMultiply: integer, integer->integer;
ENDNEWTYPE;
```

Shell commands to define a variable based on the type:

```
>varFromType a t_SubStruct
>print a
|{subField1|=0.0|,subField2|=0|}
>varFieldSet a.subField1=3.14
>varFieldSet a.subField2=1
>print a
|{subField1|=3.14|,subField2|=1|}
>varFromType b t_Struct
>varFieldSet b.field1=666
>varFieldSet b.field2=Hello world
>varFieldSet b.field3.subField1=6.55957
>varFieldSet b.field3.subField2=0
>print b
|{field1|=666|,field2|=Hello world|,field3|=|{subField1|=6.55957|,subField2|=0|}|}
>send2name pPing normal mDummy |{param1|=|$(a)|,param2|=|$(b)|}
send2name pPing NORMAL_SIGNAL mDummy
|{param1|=|{subField1|=1.23|,subField2|=0|}|,param2|=|{field1|=666|,field2|=Hello
world|,field3|=|{subField1|=6.55957|,subField2|=0|}|}|}
>Signal: mDummy sent by: RTDS_Env(-1) at: 0 ticks
>{param1={subField1=1.23,subField2=0},param2={field1=666,field2=Hello
world,field3={subField1=6.55957,subField2=0}}
>
```

- **varFromValue**

This command allows to declare an untyped variable based on its value. Shell commands to define a variable based on its values:

```
>varFromValue c=|{subField1|=1.23|,subField2|=0|}
>varFromType b t_Struct
>varFieldSet b.field1=666
>varFieldSet b.field2=Hello world
>varFieldSet b.field3.subField1=6.55957
>varFieldSet b.field3.subField2=0
>print b
|{field1|=666|,field2|=Hello world|,field3|=|{subField1|=6.55957|,subField2|=0|}|}
>send2name pPing normal mDummy |{param1|=|$(c)|,param2|=|$(b)|}
send2name pPing NORMAL_SIGNAL mDummy
|{param1|=|{subField1|=3.14|,subField2|=1|}|,param2|=|{field1|=666|,field2|=Hello
world|,field3|=|{subField1|=6.55957|,subField2|=0|}|}|}
>Signal: mDummy sent by: RTDS_Env(-1) at: 0 ticks
>{param1={subField1=3.14,subField2=1},param2={field1=666,field2=Hello
world,field3={subField1=6.55957,subField2=0}}
>send2name pPing normal mDummy |{param1|=|$(c)|,param2|=|$(b)|}
```



```
send2name pPing NORMAL_SIGNAL mDummy
|{param1|=|{subField1|=1.23|,subField2|=0|}|,param2|=|{field1|=666|,field2|=Hello
world|,field3|=|{subField1|=6.55957|,subField2|=0|}|}|}
>Signal: mDummy sent by: RTDS_Env(-1) at: 0 ticks
>{param1={subField1=1.23,subField2=0},param2={field1=666,field2=Hello
world,field3={subField1=6.55957,subField2=0}}}
```

- **varFieldSet**
This command sets a field of the variable. This can only be used on simple type fields.
- **print**
This command prints a variable value.
- **dataTypes**
This command is a verbose mode that displays the type when printing data.

- **Accessing variables**

- **Shell variables**

Shell variables can be accessed with the `|${<variable name>|}` syntax.

Example:

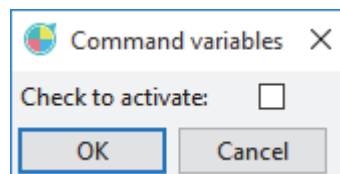
```
varFromType myVar mySubStructType
print myVar
|{b|= |,a|=0|}
echo |${myVar}
echos:
|{b|= |,a|=0|}
```

- **Interactive variables**

It is possible to ask the user for a value with the `|${<input label>|}` syntax. Options for the input label are: For strings, the only option is its length (default: 20). For booleans, options are the value when checked and the value when unchecked, separated by a comma. For example, a field with type “b[-r,]” will be replaced in the command by “-r” if the user checks the corresponding checkbox, and by the empty string otherwise. The defaults are “1” for checked and “0” for unchecked.

Example:

```
echo |${Check to activate: /b}
pops up the following window:
```



echos 1 if checked or 0 if unchecked.

9.3.7.5 trace commands

To list all the available commands in this category, type:

```
help trace
```

It will list the following commands:

Command	- Explanation
---------	---------------

backTrace	- display last events traced when activated in profile
setupMscTrace	<time information> <message parameters> [<agents>]

```

sets up the MSC trace where:
<time information> is 0 or 1
<message parameters> is 0 or 1
<agents> is the list of agent names to trace separated by spaces
startMscTrace
stopMscTrace
saveMscTrace <file name>
setEnvInterfaceFilter <filter status>
  <filter status> is 1 or 0, when active only messages with the environment will be traced

```

- **MSC trace**

The MSC trace can be configured, started, stopped, and saved from the shell.

Example:

```
setupMscTrace 0 1 pPing
```

Will only trace pPing instance with no time information but with parameters.

- **Filtering the interface between the environment and the system**

The setEnvInterfaceFilter command allows to filter out SDL events that are not related to the interface of the system at a very low level in the PragmaDev Simulator. This feature should be used to increase simulation speed and when internal information is not needed.

9.3.7.6 customization commands

To list all the available commands in this category, type:
 help customization

It will list the following commands:

Command	- Explanation
---------	---------------

buttonWindowCreate <button window name>	creates a window to contain user defined buttons
buttonWindowAdd <button window name> <button name> = <shell command> [; <shell command>]*	adds a button to previously created button window <button window name> is the name of the button window <button name> is the text to be displayed on the button <shell command> is the command associated with the button
buttonWindowDel <button window name> <button name>	removes a button from a button window <button window name> is the name of the button window <button name> is the text of the button to be removed
buttonWindowLabelAdd <button window name> <label name>	adds a label to previously created button window <button window name> is the name of the button window <label name> is the text to be displayed on the label
buttonWindowLabelDel <button window name> <label name>	removes a label from a button window <button window name> is the name of the button window <label name> is the text of the label to be removed

- **Button windows**

It is possible to create user-defined buttons and to associate shell commands.

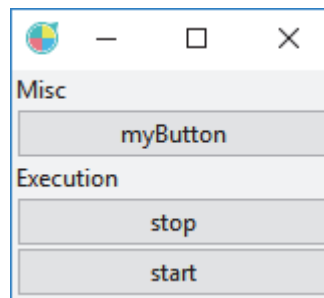
Here is an example of a button window:

```

>buttonWindowCreate myWindow
>buttonWindowLabelAdd myWindow Misc
>buttonWindowAdd myWindow myButton = help
>buttonWindowLabelAdd myWindow Execution
>buttonWindowAdd myWindow stop = send2name pPing normal mStop

```

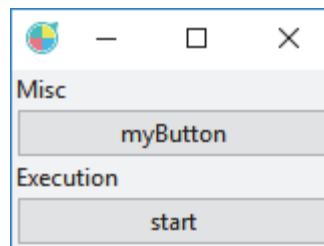
```
>buttonWindowAdd myWindow start = send2name pPing normal mStart |{param1|=12345|}
```



So clicking on *myButton* will actually execute the help command in the shell.

It is also possible to remove labels or buttons:

```
>buttonWindowDel myWindow stop
```



It is possible to create several button windows.

To stop one of the window, just close the window.

9.3.8 Status bar

The status bar is divided in two parts:

- The PragmaDev Simulator internal state
The PragmaDev Simulator can have the following internal states:

State	Meaning
STOPPED	The system is stopped
STOPPING	The system is trying to stop. No commands are allowed in that intermediate state.
RUNNING	The system is running. The traces might be active or not ("Options / Free run"). A stop is possible in that state.
STEPPING	C code classical stepping. Note a classical step might take a lot of time. A stop is possible in that state.
KEY_SDL_STEPPING	Step to the next SDL key event. Note an SDL step might take some time. A stop is possible in that state.

Table 3: PragmaDev Simulator internal states

State	Meaning
SDL_TRANSITION	Step until the end of the SDL transition. Note an SDL step might take some time. A stop is possible in that state.
ERROR	An error has occurred and the PragmaDev Simulator is stuck. Restart the PragmaDev Simulator.

Table 3: PragmaDev Simulator internal states

- The active thread
The active thread is displayed in the right part of the status bar when known.

9.3.9 Breakpoints

9.3.9.1 Setting breakpoints

Breakpoints are set in the SDL editor. Select an SDL symbol and click on  quick button or via the “Debug / Set breakpoint” menu to set a simple breakpoint.


Breakpoints can also be set via the “break” command with the following syntax:

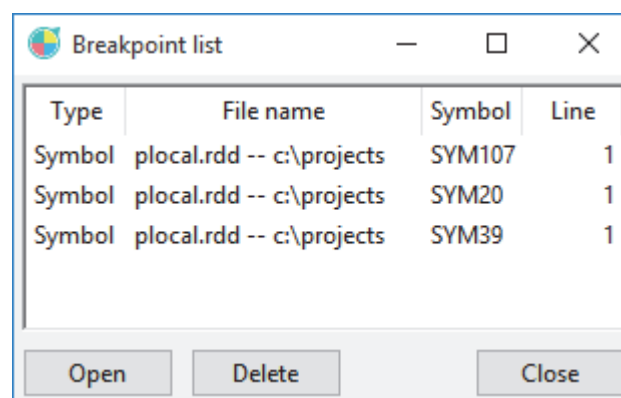
`break diagram-file-name:symbol-identifier:line-number-in-symbol`

Since symbol identifiers are not directly visible in the diagram editor, the best way to get the command is to set the breakpoint interactively, which will record the corresponding command in the shell history. Note that symbol identifiers never change, so it is safe to put such a command in a scenario file that will be executed several times.

9.3.9.2 Listing breakpoints

The breakpoints that have been set can be listed:

- In the PragmaDev Studio shell with the `list` command.
- In the breakpoint list window by clicking on the  button in the toolbar. This window looks like follows:





For each breakpoint is given:

- its type: symbol or file,
- the file name for the diagram or source file where it is set,
- the internal identifier for the symbol where it is set if applicable,
- and the line number in the source file or symbol text where it is set.

From this window, selecting a breakpoint and clicking on “Open” or double-clicking on a breakpoint line will display the symbol or file at the position of the breakpoint, and selecting a breakpoint and clicking “Delete” will delete the breakpoint.

9.3.9.3 Deleting breakpoints

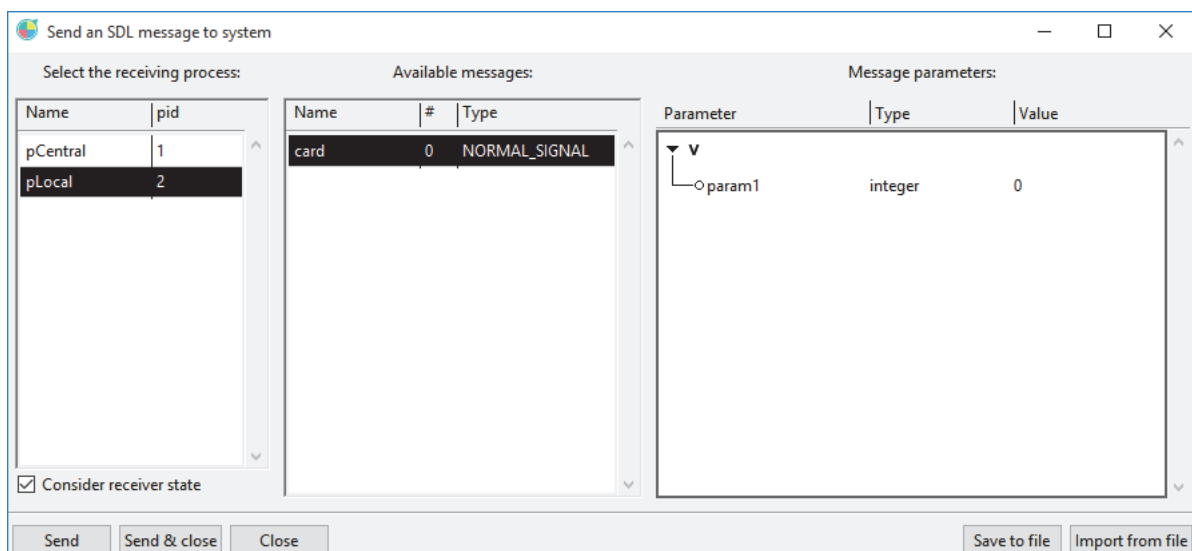
Breakpoints can be deleted from:

- The shell with the delete command:
`delete <breakpoint number>`
 where the breakpoint number is the number listed from the list command.
- The breakpoint list window, as explained in “Listing breakpoints” on page 288.
- The text editor: select a line where a breakpoint is set and press the  button in the debug toolbar.
- The diagram editor: put the text cursor in a symbol at a line where a breakpoint is set and press the  button in the debug toolbar.

9.3.10 Sending SDL messages to the running system



The PragmaDev Simulator’s *Send an SDL message to the running system* button opens the *Send an SDL message to system* window. It will list the possible receivers, and the available messages in the system:



The send an SDL message to system Window

An equivalent command can be found in the shell:

```
send2pid <sender pid> <receiver pid> <signal number or name> [<parameters>]
```

where signal type can be normal or timer, or:

```
send2name <sender name> <receiver name> <signal number or name> [<parameters>]
sendVia <sender pid> <channel or gate name> <signal number or name> [<parameters>]
send <sender pid> <signal number or name> [<parameters>]
```

Verifications are made on the sender pid and receiver pid only.

Structured parameters are updated and displayed at the right of the window when selecting a signal. The equivalent format for the shell command depends on whether the message is structured or not. Structured parameters are fully described in PragmaDev Studio Reference Manual. In short, a message is structured if and only if it is declared with several parameters or with one parameter that is a pointer to a struct or a union.

- For a non-structured message, the text for the parameter must be a sequence of bytes written in hexadecimal format, exactly as they will appear in the target program memory.
- For a structured message, the text for the parameter must be written as follows:
 - The values for base types are written as in C: for example 12 or 871 are valid values for an int, X is a valid value for a char, and so on...
 - The values for structs or choices are coded as follows:
`|{field1|=value|,field2|=value|,...|}`
 For example, for a struct defined as:

```
MyStruct STRUCT { i integer; s charstring; };
```

 a valid format is:
`|{i|=4|,s|=|:abcd|}`
 In the struct created on the target, the field i will be set to 4 and the field s will be set to "abcd".
Please note that what is significant in the formatted text is not the field names, but the field order; so in the example above, you can't write:
`|{s|=|:abcd|,i|=4|}/* INVALID! */`
 As a consequence, the field names are in fact optional, so you can write:
`|{|=4|,|=|:abcd|}`
 Please also note that if no value is specified for a field, the field is left as is. This can be used to set the value for fields in a choice. For example, for:

```
CHOICE MyChoice { i integer; c character; };
```

 a valid format is:
`|{i|=|,p|='a'|}`
 The field i won't be set and the field p will be set to 'a'.
- Escape sequences
 Use a | | to introduce a | in the message parameters,
 Use a | . to introduce a carriage return in the message parameters.

9.3.11 Code coverage



The PragmaDev Simulator's *Get code coverage* button gets the code coverage analysis results for the running system so far. This feature is available only if the *Activate code coverage analysis* is checked in the simulation options (see "Simulator options" on page 267).

For more details on code coverage results, see "Code coverage results" on page 325.

9.3.12 Connecting an external tool

It is possible to connect an external tool to the PragmaDev Simulator through a socket. To allow connections to the PragmaDev Simulator the connect command should be entered in the shell:

```
connect <port number>
```

The IP address used is the IP address of the host where the PragmaDev Simulator is running. Only the port number can be configured.

The PragmaDev Simulator is seen as a server so the connect command should be executed before starting the client.

Once the connection is made the client has basically a direct access to the shell commands: whatever is sent goes to the PragmaDev Studio shell and whatever the shell replies goes to the socket. Therefore the syntax is the one used in the shell. Note the external tool connected will also receive any information that is printed out in the shell.

To close the socket use the disconnect command in the PragmaDev Studio shell.

Here is a sample code in Python (<http://www.python.org>) that connects to port 50010:

First start the server in the PragmaDev Studio shell:

```
connect 50010
```

Then go to a shell or DOS window and type:

```
python
>> from socket import *
>> s=socket(AF_INET, SOCK_STREAM)
>> s.connect((gethostname(), 50010))
>> s.send('help\n')
>> print s.recv(500)
```

It will print out the 500 first characters of the PragmaDev Simulator help and display it in the shell.

9.3.13 Command line simulation

The Simulator can be started from a shell or a DOS console and run an execution script automatically with the rtdsSimulate command. Check the Reference Manual for more information.

9.3.14 Communication with an external XML-RPC server

An operator or an external procedure can be implemented outside the SDL system. To do so the “Generation / Options...” must define an XML-RPC server and an optional module name as explained in “Simulator options” on page 267.

An operator call in the PragmaDev Simulator will result in a call to:

```
[<module name>.]<operator name>
```

on the server.

An external procedure call in the PragmaDev Simulator will result in a call to:

```
[<module name>.]<procedure name>:external_proc
```

on the server.

Please note the <procedure name> capitalization must be the one used when calling the procedure in the SDL system.

The types used for the parameters and return value of the operator or procedures are transformed into their XML-RPC equivalent and used the following way:

- For an operator, the implementation is called with the same parameters as the operator itself, and returns the same return value.
- For external procedures, the implementation is called with the same parameters as the procedures. However, some parameters may be passed as IN/OUT, allowing their value to be modified by the implementation.
 To do that, the return value for the implementation of the procedure does not only contain its actual return value, but also the values of all its IN/OUT parameters. The return value for the implementation is therefore a XML-RPC struct, with one field for each IN/OUT parameter, having the same name as the declared procedure parameter, and if needed, an additional special field named "return value" containing the actual return value for the procedure. This field name has been chosen to make sure it won't conflict with any procedure parameter name while still being readable.

The rules to represent SDL data types in XML-RPC are summarized in the following table:

SDL data type	XML-RPC representation
Boolean	<boolean>
Integer	<int>
Natural	<int>
Real	<double>
Character	<string> with length 1
CharString	<string>
BitString	<i>Not available.</i>
OctetString	<i>Not available.</i>
PID	<int>
Duration	<int>
Time	<int>
STRUCT	<struct> with the same fields as the SDL STRUCT in the same order

Table 4: XML-RPC representation of SDL data types

SDL data type	XML-RPC representation
CHOICE	<array> of 1 or 2 elements: the first is a <string> containing the value for the SDL 'present' field in the CHOICE. The second is the value for the selected field if it's valid.
LITERALS	<string> containing the literal name
Array(IndexSort, ElementSort)	<p><array> of <struct> containing each a field named 'index' containing the value for the index as a <string>, and a field named 'element' containing the element at this index. The type for this field is the XML-RPC representation of <i>ElementSort</i>.</p> <p>The array may also contain an additional single <struct> with a single field called 'default', its type also being the XML-RPC representation of <i>ElementSort</i>. This <struct> gives the default value for the SDL Array elements that do not have an explicit value. It is typically used when the array is initialized via:</p> <pre>array := (.)</pre>
String(<i>ElementSort</i>)	<i>Not available.</i>
Bag(<i>ElementSort</i>)	<i>Not available.</i>

Table 4: XML-RPC representation of SDL data types

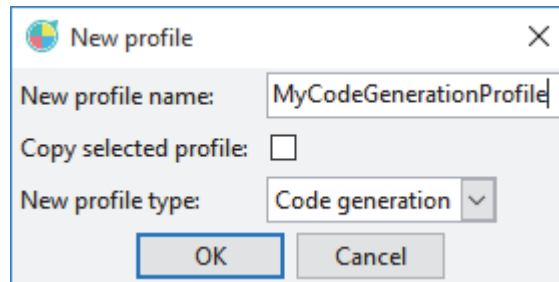
An example is available in our distribution.

9.4 - SDL Z.100 Code generation

An SDL Z.100 system can be generated to C code, integrated with an RTOS, and graphically debugged with one of the supported debuggers.

The SDL Z.100 to C code generation translation rules are explained in the SDL to SDL-RT conversion chapter of the Reference Manual.

C code is generated with a code generation profile. When creating a new profile a pop up window will ask if the profile is for simulation or code generation:



Select “Code generation” and fill in the options as explained in the “Profiles” on page 157.

The graphical debugger architecture and features are described in “PragmaDev Debugger” on page 229.

10 - Performance Analyzer

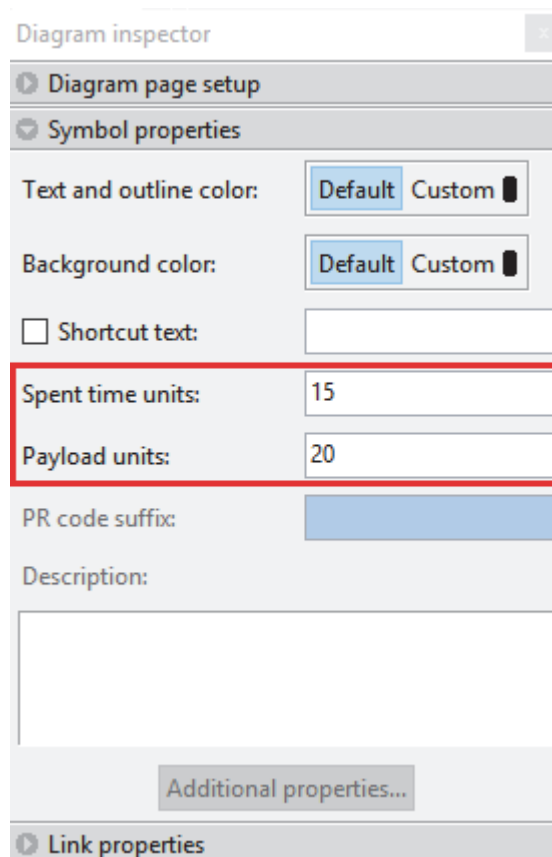
10.1 - Objectives

The *PragmaDev Performance Analyzer* helps to identify the most suited architecture for best time performance (lower time) and/or best consumption performance (lower payload). For that purpose it evaluates the time and the payload (consumption) of a number of scenarios against a set of architectures.

10.2 - Time and payload information

10.2.1 The model

Time and payload can be associated to each symbol in the behavior diagram. Whenever the symbol is executed time and payload will be increased based on the values indicated in the properties of the symbol (see “Symbol and link properties” on page 56), namely “Spent time units” and “Payload units”:



The image shows a 'Diagram inspector' dialog box with a close button in the top right corner. It has two main sections: 'Diagram page setup' and 'Symbol properties'. The 'Symbol properties' section is expanded and contains several fields. The 'Spent time units' field, which contains the value '15', and the 'Payload units' field, which contains the value '20', are highlighted with a red rectangular border. Other fields include 'Text and outline color' (with 'Default' and 'Custom' buttons), 'Background color' (with 'Default' and 'Custom' buttons), a 'Shortcut text' checkbox and text box, 'PR code suffix' (with a blue text box), and a 'Description' text area. At the bottom of the 'Symbol properties' section is an 'Additional properties...' button. Below this section is a 'Link properties' section, which is currently collapsed.

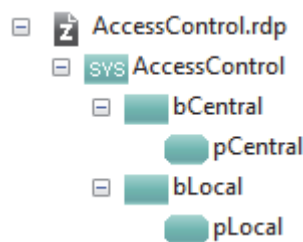
Expected values for time and payload units are positive integers. If an incorrect (or empty) value is given, then the default value of 0 (zero) will be assigned to both. It should be noted that the given values represent *units* of time and payload; the actual values are obtained by multiplying the *units* with the *factors* given in the architecture.

10.2.2 The architecture

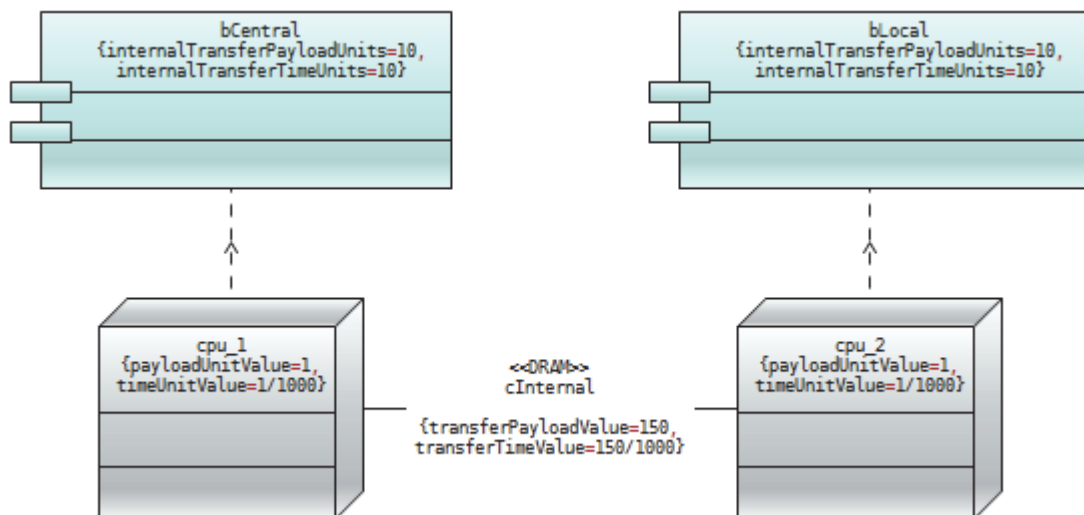
The possible allocations of the system are defined in deployment diagrams. The diagram will define:

- On which nodes the agents are executed.
- The time and payload units of the messages exchanged within the same agent.
- The time and payload multiplying factors that apply on the units.
- The time and payload values of the messages going from one node to the other.

For example, for the following model of a system:



a possible allocation with deployment diagram can be:

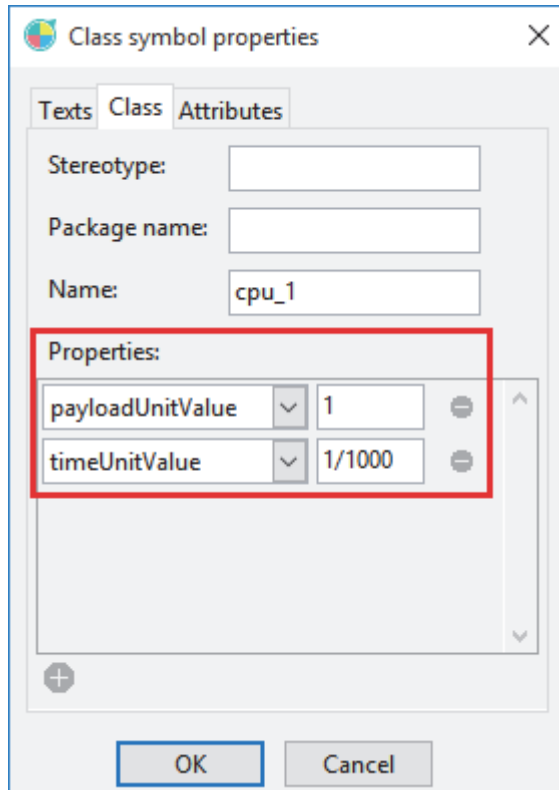


The diagram implies execution of the agent `bCentral` in node `cpu_1` and `bLocal` in `cpu_2`. Every message exchanged within the agent (e.g., `bCentral`) is associated to the units given by the component's properties: `internalTransferPayloadUnits` and `internalTransferTimeUnits`. The values expected here follow the same principles as those in the model (see "The model" on page 295).

The unit multiplying factors are given by the node's properties: `payloadUnitValue` and `timeUnitValue`. Expected values for `payloadUnitValue` are positive integers; for `timeUnitValue` are positive integers divided by a power of 10 (e.g., `1/1000`). The aim of the divisor in `timeUnitValue` is to create a ratio between the time spent in symbols (and communication within an agent) and the time in the model (e.g., values used in SDL timers). If incorrect or empty values are detected, then the default value of 1 (one) is assigned.

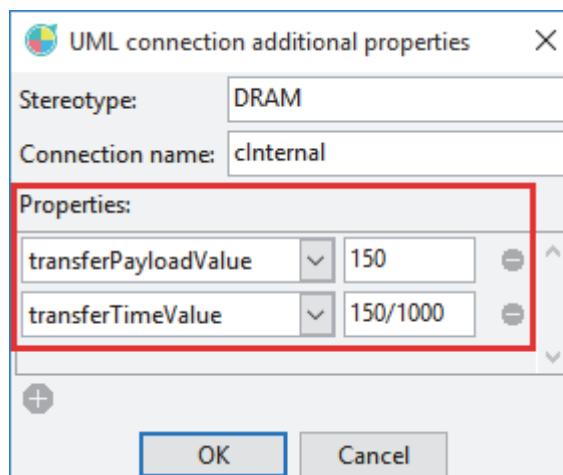
Messages exchanged between nodes (i.e., agents executing in different nodes) are associated to the values given by the connection's (e.g., cIntern) properties: `transferPayloadValue` and `transferTimeValue`. These are actual values for the payload and time (not units, hence no multiplying factors apply). Expected values for `transferPayloadValue` are positive integers; for `transferTimeValue` are positive integers divided by a power of 10 (e.g., 1/1000). If incorrect or empty values are detected, then the default value of 0 (zero) is assigned.

The payload and time properties of either node or component can be entered (edited) directly in the symbol text or via its "Additional properties..." in the "Class" tab:



The screenshot shows the 'Class symbol properties' dialog box with the 'Class' tab selected. The 'Name' field contains 'cpu_1'. The 'Properties' section, highlighted with a red box, contains two entries: 'payloadUnitValue' with a value of 1, and 'timeUnitValue' with a value of 1/1000. The 'OK' and 'Cancel' buttons are at the bottom.

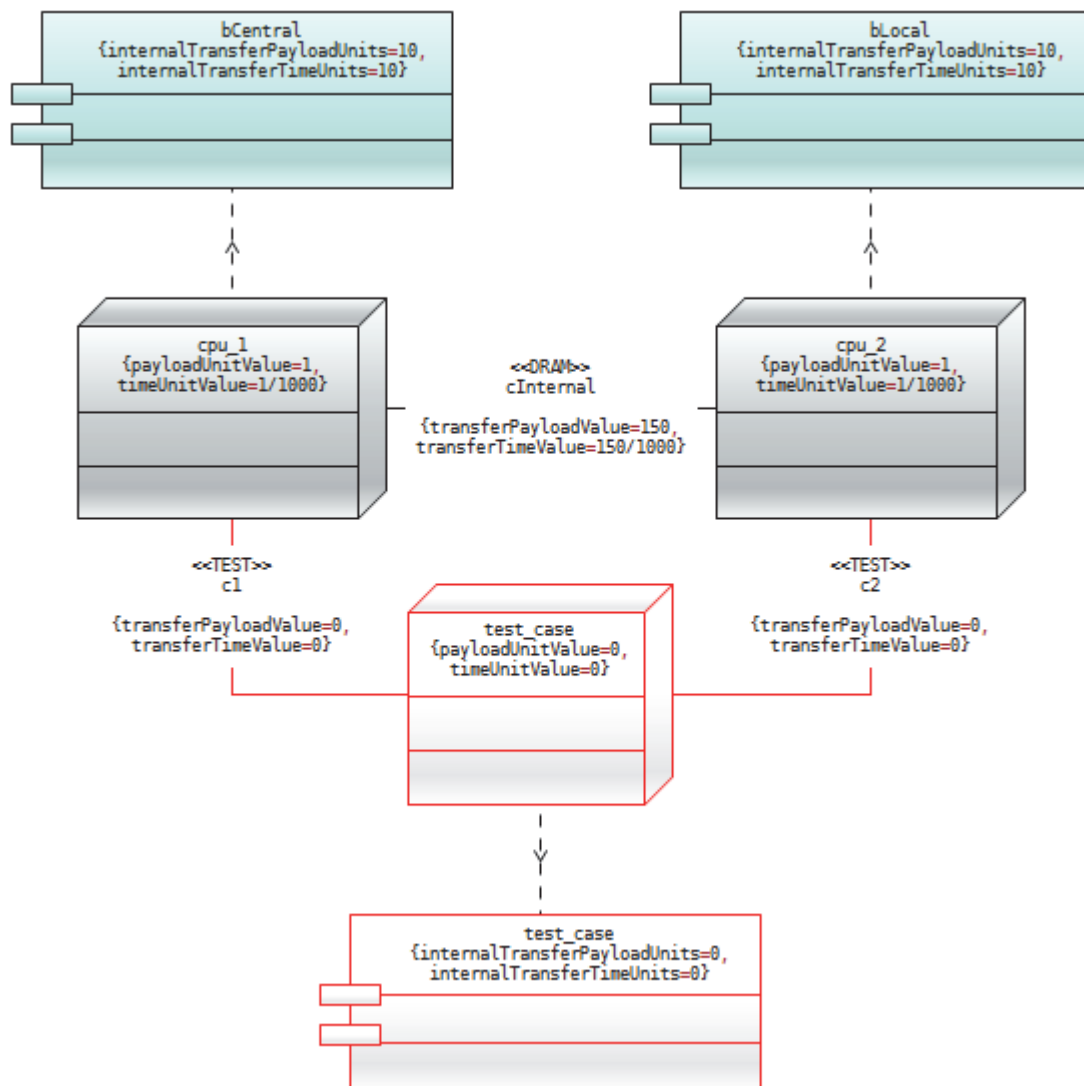
The same is true for the connection, where the properties can be entered in the link text or via its "Additional properties...":



The screenshot shows the 'UML connection additional properties' dialog box. The 'Stereotype' field contains 'DRAM' and the 'Connection name' field contains 'cInternal'. The 'Properties' section, highlighted with a red box, contains two entries: 'transferPayloadValue' with a value of 150, and 'transferTimeValue' with a value of 150/1000. The 'OK' and 'Cancel' buttons are at the bottom.

10.2.3 The test cases

Performance analysis is driven by scenarios (test cases) described in TTCN-3. However, test case execution is associated to neither payload nor time units (or values). The payload and time of messages exchanged between the system and the test is 0 (zero). This is equivalent to having (implicitly in the deployment diagram) a `test_case` component attached to a `test_case` node, which in turn is connected to all other nodes. The performance properties of the component, node, and connection are set to 0 (zero).



10.2.4 Semantics

10.2.4.1 Time

Execution of agents on the same node is sequential, and execution on different nodes is parallel. Each node is considered to have an internal clock which advances based on the time spent by the agents executing on that node. Sequential execution (only one node) is straightforward because there is only one clock. If two (or more) nodes are involved, their respective clocks are synchronized based on message exchange. For example, let's suppose that the internal clock of `cpu_1` is 100 and that of `cpu_2` is 80. If a message is

sent from `cpu_1` to `cpu_2`, then its estimated time of arrival will be $100 + 150/1000$ (because it will travel through `cInternal`). Because the clock of `cpu_2` is less than the estimated time, upon receiving the message, `cpu_2` will:

- advance its clock to the estimated time of arrival of the message,
- handle any events (e.g., SDL timer time-out) that may have been triggered because of the clock change, and
- handle the received message.

A message sent from `cpu_2` to `cpu_1` requires no changes to the clock, because the clock of `cpu_1` is greater than the estimated time of arrival of the message. In this case the message is handled immediately.

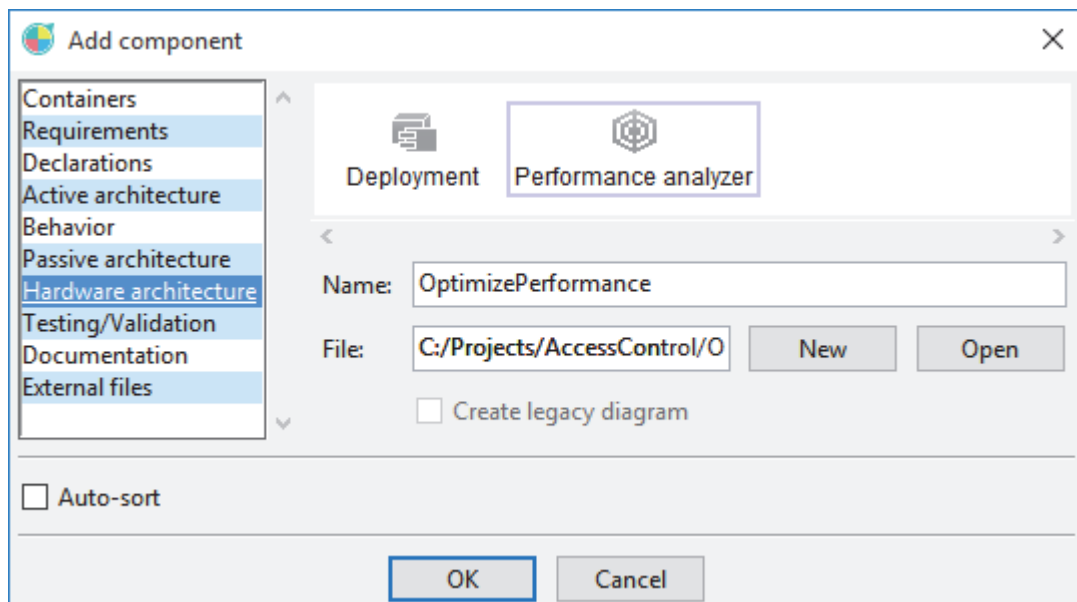
The resulting time the end of the analysis is the highest value among the clocks of all nodes, e.g., 100 in the example above.

10.2.4.2 Payload

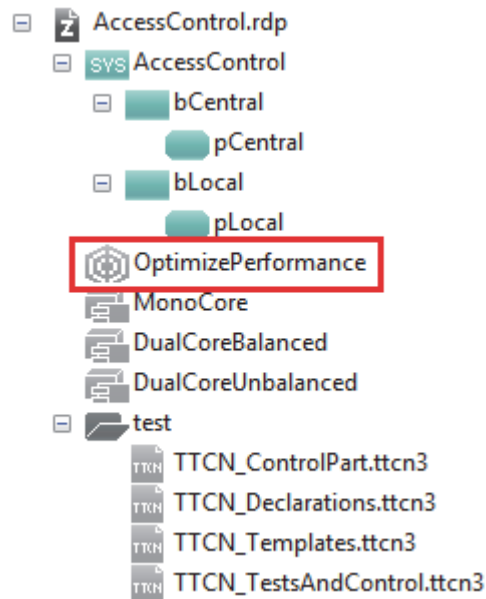
Each node is considered to have an internal payload accumulator which is incremented based on the payload associated to the agents (symbols and message exchange) executing on that node. The resulting payload the end of the analysis is the sum of the accumulators of all nodes. For example, if the accumulated payload of `cpu_1` is 40 and that of `cpu_2` is 30, then the resulting payload will be 70.

10.3 - Table and graphical analysis

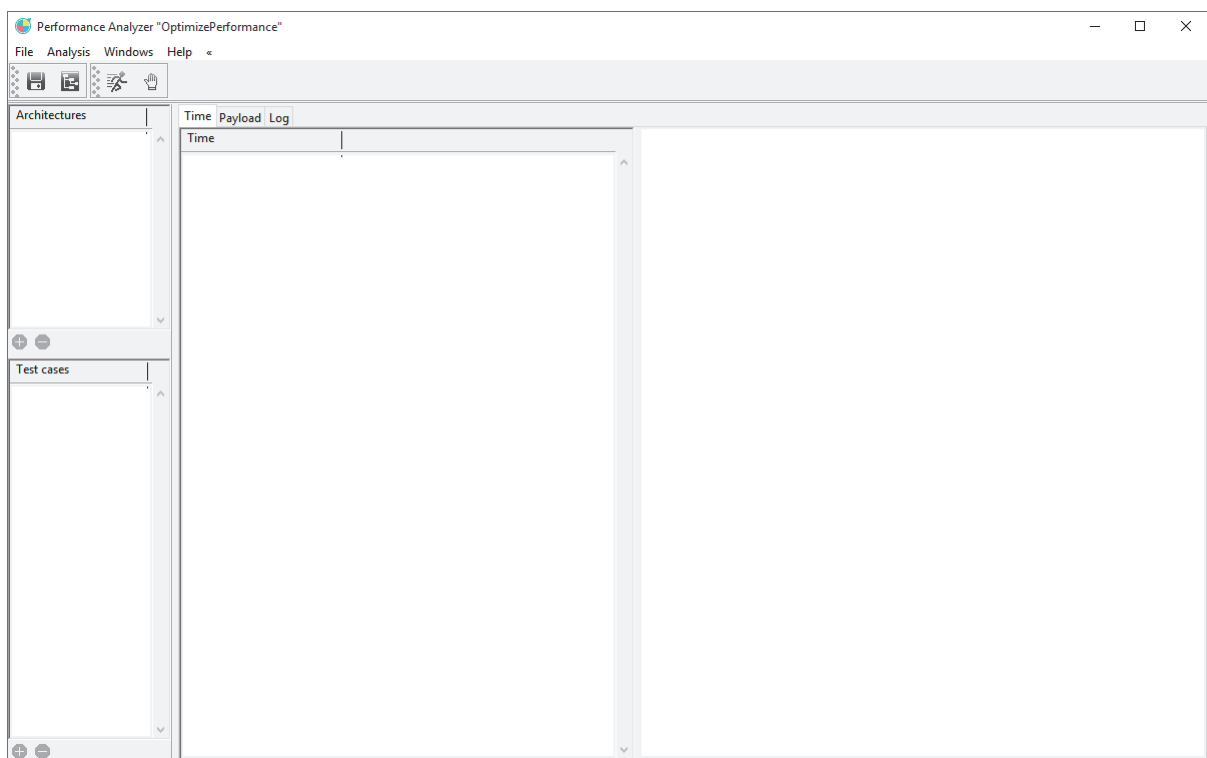
The PragmaDev Performance Analyzer requires a specific component to be added in the project (see “Adding a single component” on page 12):



This will create a performance analysis file and add the corresponding component in the project tree:



Double-clicking on this component will open the Performance Analyzer:





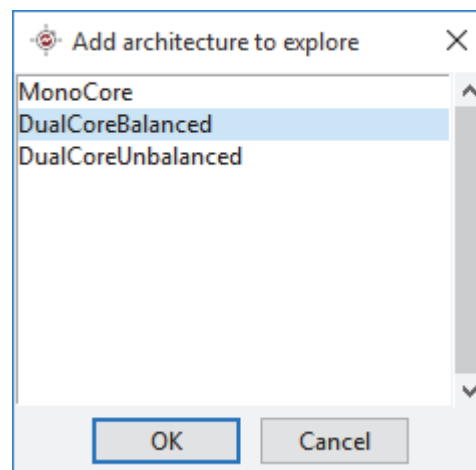
The user interface is divided in two parts:

- The left part is used to add / remove architectures and test cases (scenarios) to the analysis.
- The right part displays the results of the analysis in tabular form (left) and graphical form (right).

The three tabs at the top of the right part (“Time”, “Payload”, and “Log”) can be used to switch between the results of time and payload respectively, or show the log messages generated during the analysis which contain all the details.



10.3.1 Architectures and test cases

Adding / removing an architecture (or test case) to the analysis is achieved via the corresponding   quick-buttons. When trying to add an architecture (or test case) to the analysis, a list of all available choices will pop-up:

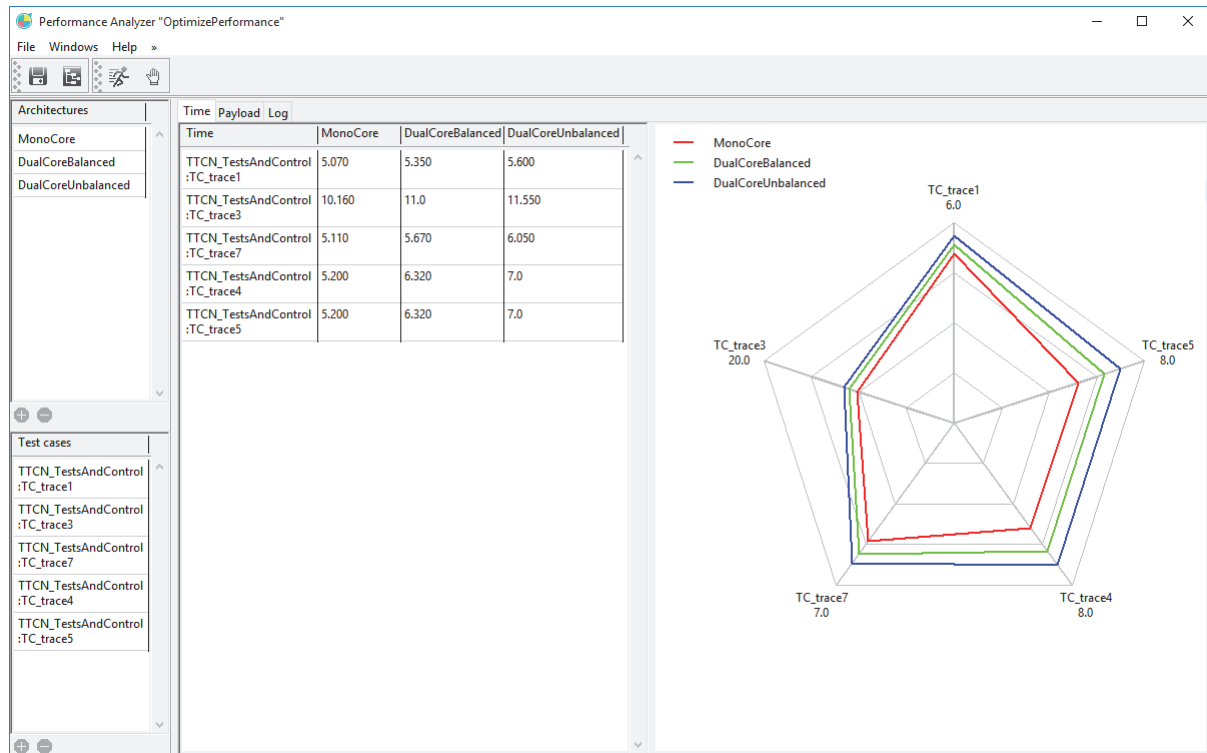


This list is computed automatically based on the available deployment diagrams and TTCN-3 files present in the project tree.

10.3.2 Running the analysis

After the architectures and test cases have been added, the analysis can be started via the menu “Analysis / Start” or the  quick button, and it can be stopped via “Analysis / Stop” or . As the analysis proceeds, the current pair under analysis (architecture, test case) will be marked with “Analyzing...” in the corresponding cell of the table. The table


and the graph is updated with the results as soon as they are available (i.e., the analysis of the current pair is finished).



The result shown in the table and graph depends on the verdict of the test case:

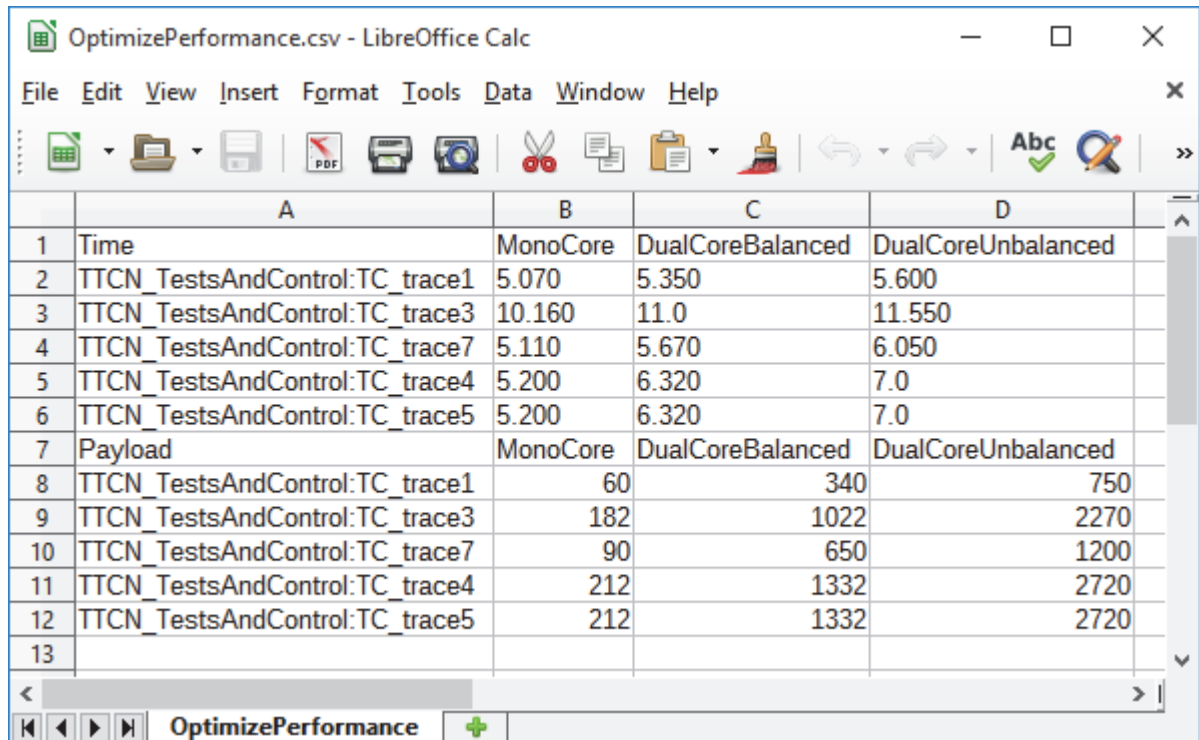
- If the verdict is pass, then the values of time and payload will be shown in both table and graph.
- If the verdict is anything else (e.g., fail), then the verdict of the test case will be shown in the table and a 0 (zero) value in the graph.

10.3.3 Saving the results

The results of an analysis can be saved via the menu “File / Save” or the  quick button. It is possible to undo any changes to the results by reverting to the last saved state via the menu “File / Revert...”.

The Performance Analyzer provides also an export functionality that will save the results in CSV format. This functionality can be accessed via the menu “File / Export to CSV...”.

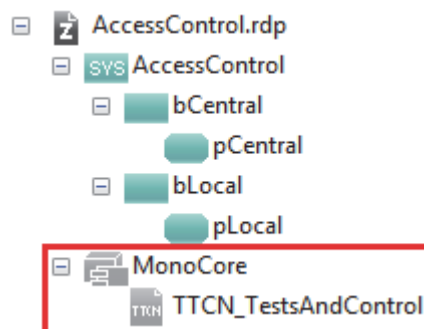
and the resulting file can be then imported and edited with any spreadsheet application. Here is an example of the results imported in LibreOffice Calc:



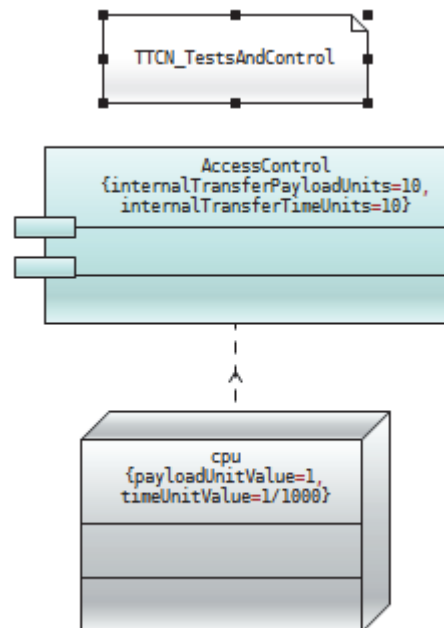
	A	B	C	D
1	Time	MonoCore	DualCoreBalanced	DualCoreUnbalanced
2	TTCN_TestsAndControl:TC_trace1	5.070	5.350	5.600
3	TTCN_TestsAndControl:TC_trace3	10.160	11.0	11.550
4	TTCN_TestsAndControl:TC_trace7	5.110	5.670	6.050
5	TTCN_TestsAndControl:TC_trace4	5.200	6.320	7.0
6	TTCN_TestsAndControl:TC_trace5	5.200	6.320	7.0
7	Payload	MonoCore	DualCoreBalanced	DualCoreUnbalanced
8	TTCN_TestsAndControl:TC_trace1	60	340	750
9	TTCN_TestsAndControl:TC_trace3	182	1022	2270
10	TTCN_TestsAndControl:TC_trace7	90	650	1200
11	TTCN_TestsAndControl:TC_trace4	212	1332	2720
12	TTCN_TestsAndControl:TC_trace5	212	1332	2720
13				

10.4 - SDL Z.100 performance simulation

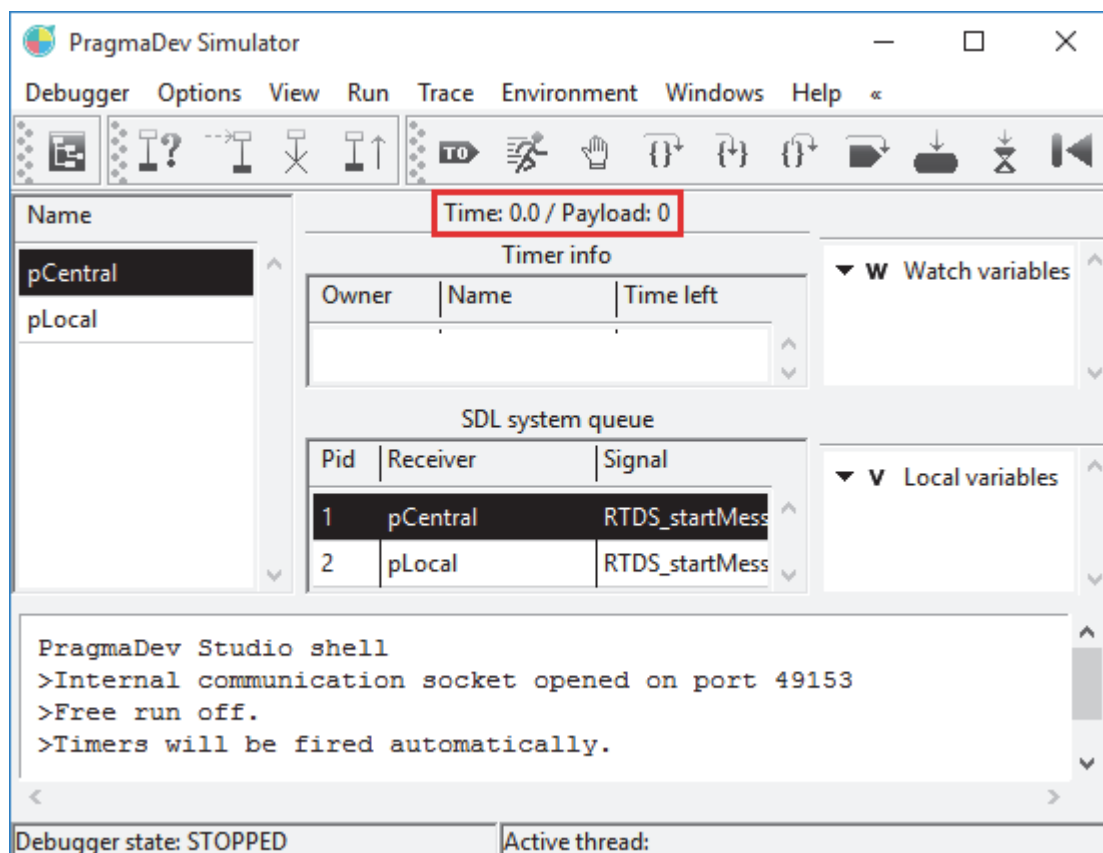
The core of *PragmaDev Performance Analyzer* is the *SDL Z.100 Simulator* with a specific *scheduler* (see “Simulator architecture” on page 266) that takes into account the time and payload properties in the model and deployment diagram. This allows the use of the *SDL Z.100 Simulator* with the *performance scheduler*. To trigger the use of such scheduler the simulator should be started on a deployment diagram or TTCN-3 file attached to it, instead of an agent or TTCN-3 file (not attached to a deployment diagram):



Also, the selected deployment diagram should contain a single component (agent) executing on a single node (and the TTCN-3 file-s if any):



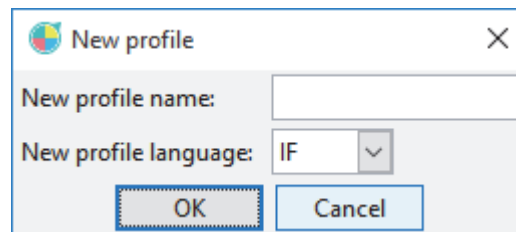
The interface and functionality of the simulator remain the same, with the only small difference that the payload is also shown next to the time:



10.5 - Validation and Model-Based Testing

10.5.1 Scope

PragmaDev Studio allows two possible ways for system validation: the *IF toolbox* from Verimag and *Diversity* from the CEA. For this, a validation profile has to be created in the “Validation / Options...” menu. When creating a new profile, a pop up window will ask the language used for the verification:



10.5.2 IF Toolbox

SDL Z.100 systems can be translated to IF descriptions as specified by Verimag. The translation rules and restrictions can be found in the Reference Manual. Tools based on IF technology allow:

- Exhaustive simulation,
- Test generation.

The IFx toolbox has to be downloaded on Verimag website: <http://www-if.imag.fr/>. The Python language interpreter will be required too; it be found at <http://www.python.org/>.

Rules to be verified during exploration are described in IF observers. Each time a transition is executed in the system, the IF observer verifies its internal rules. Whenever a rule is verified or violated, it is possible to generate an MSC or a test case.

10.5.2.1 IF Observers

Rules verified during the state space exploration are described by observers.

An observer file can be directly had to a project, or inside a folder, by choosing “Add child element...” in the contextual menu. Many observers can be added, and used to test the system.

Observers are processes that can view everything that happens in the system. They are evaluated each time the system reaches a new state. They can verify:

- Static rules such as the value of variables,
- Dynamic rules such as a sequence of events.

Observers can define variables, handle timers, and evaluate expressions.

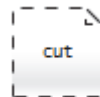
Observers, as supported in PragmaDev Studio, look like SDL processes but they are not SDL processes. The syntax of the statements is based on IF language, and it does not have any message queue by default.

10.5.2.1.1 Types of Observers

There are three types of Observers:

- **pure**
That type is the most basic one, it can not interfere with the system nor with the exploration.
- **cut**
This is the most common type of observers, it can not interfere with the system but it can interfere with the state exploration. A typical behavior is to stop exploration (cut) in a branch that is not of interest.
- **intrusive**
The observer can interfere with the state exploration and it can modify the system itself: send signals and modify variables.

The type of Observer is indicated in the declaration symbol:



10.5.2.1.2 Data types in Observers

The following basic data types are available in IF: *integer*, *real*, *boolean*, *pid*, *clock*; *character* and *charstring* are replaced by *RTDS_charstring*, which is a string of integers which holds the ASCII value of each character. To define the value of a charstring, it is required to define the ASCII code of each element and concatenate all these elements with the symbol ^.

For example, for a charstring c equal to "toto":

```
var c RTDS_charstring;
task c := RTDS_charstring(116) ^ RTDS_charstring(111) ^
RTDS_charstring(116) ^ RTDS_charstring(111);
```

The following constructs are also available:

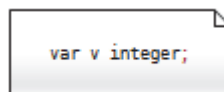
Table 5: IF constructs

Constructs	Declaration	Usage
const	const MyConst=3;	var v integer; task v := MyConst;
var	var v integer; var c clock;	var v integer; task v := 2;
enum	type MyType = enum red, green, blue endenum;	var v MyType; task v := green;

Table 5: IF constructs

Constructs	Declaration	Usage
record	<pre>type MyRecord=record FirstField integer; SecondField boolean; endrecord;</pre>	<pre>var v MyRecord; task v.FirstField := 3;</pre>
range	<pre>type MyRange=range 0..4;</pre>	
array	<pre>type MyArray=array[4] of integer;</pre>	<pre>var v MyArray; task v[3] := 5;</pre>
string	<pre>type MyString=string[5] of integer; var v MyString</pre>	<pre>var v MyString; var w MyString; var x MyString; var position integer; const value = 7; task position := length(v); task v := insert(v, position, value); task w := remove(v, position); task x := v^w; // concatenate</pre>
while	<pre>while (<condition>) do <statements>; endwhile;</pre>	<pre>while (i<4) do output sig1; task i:=i+1; endwhile</pre>
if	<pre>if (<condition>) then <statements>; else <statements>; endif;</pre>	<pre>if (i<4) then output sig1; else output sig2; endif</pre>

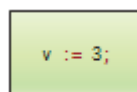
Variables are declared in the text symbol:



```
var v integer;
```

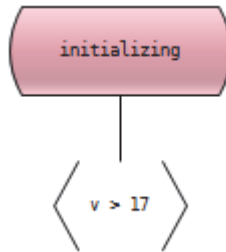
Variables are manipulated in:

- Action symbols

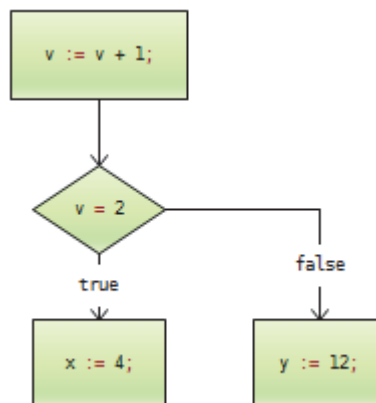


```
v := 3;
```

- Provided symbols



- Decision symbols

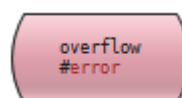


10.5.2.1.3 Action symbols in Observers

The Observer process starts with the Start symbol, can go through a number of states, and can be ended with the Stop symbol.

States

There are three types of states: ordinary, error, and success. The type of state is written below the state name with a hash:

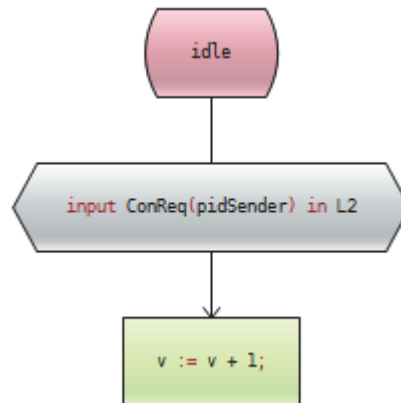


By default the state is considered ordinary.

Triggers

Three possible events can trigger the state: *match*, *provided*, *when*.

A match statement is described as a closed continuous signal SDL symbol:



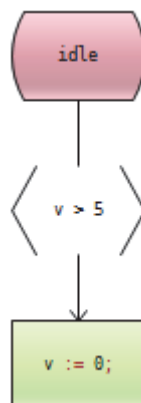
In this example, pidSender has previously been declared like a pid.

The possible match statements are:

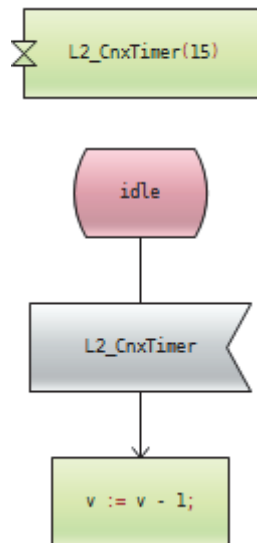
- match input <sig(sender pid, params)> in <pid>
- match output <sig(sender pid, params)> from (<pid>) via (<channel>) to (<pid>)
- match fork(<pid>) in <pid>
- match kill(<pid>) in <pid>
- match <deliver>
- match informal "my text" in (<pid>)

The match keyword is omitted in the graphical symbol.

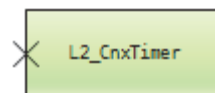
A provided statement is described with the SDL continuous signal symbol:



A when statement is described with the SDL start timer symbol to start the clock, and an input symbol:

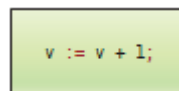


A timer can also be cancelled with the SDL cancel timer symbol:



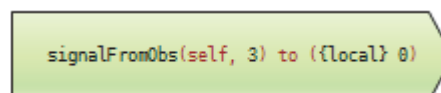
10.5.2.1.4 Statements

Any IF statement can be written in the SDL action symbol:



10.5.2.1.5 Output

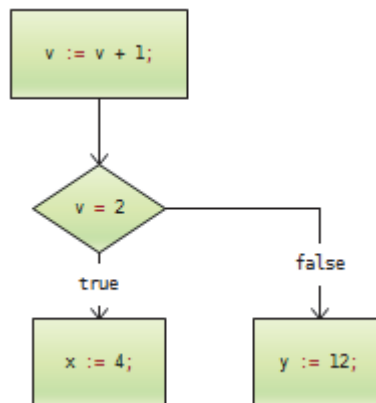
An IF intrusive observer is able to send signals to any process of the system, but can not receive any signal



self has always to be add as first parameter for an output in an observer.

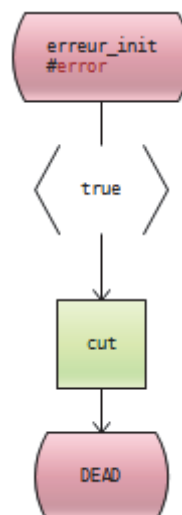
10.5.2.1.6 Decisions

Decisions are handled with unstable states in IF. It uses the SDL decision symbol:

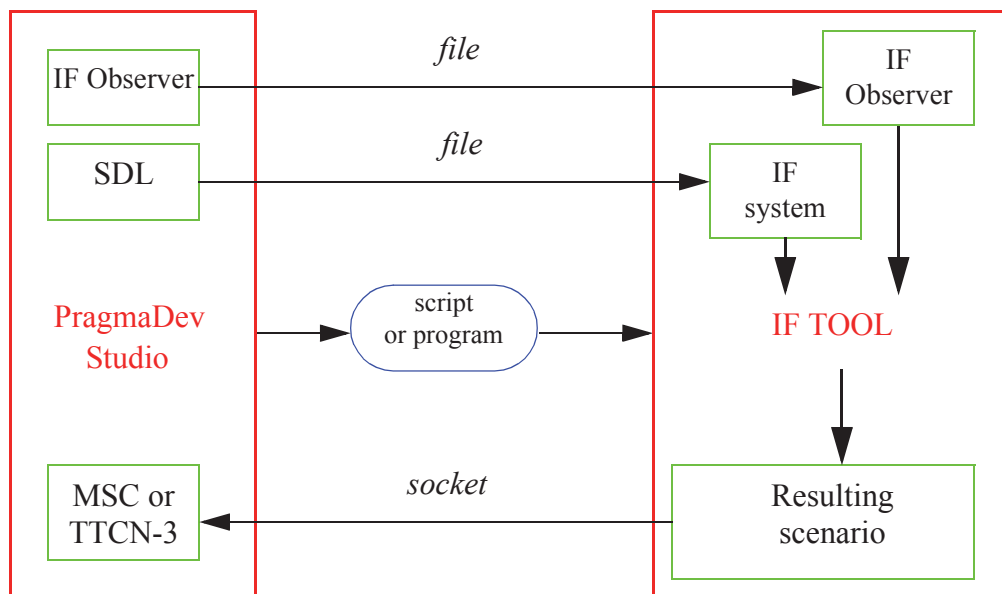


10.5.2.1.7 Reducing state space

Exploration in the current branch can be stopped with the IF *cut* statement in an action symbol. Please note the `#success` or `#error` states do not stop exploration of the system. An explicit cut action should be used afterwards in order to stop exploration.

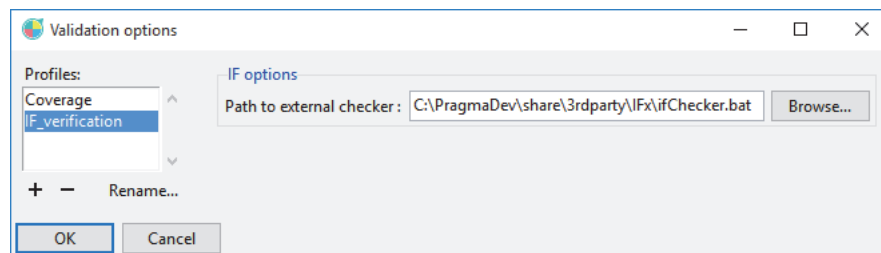


10.5.2.2 General architecture



The SDL system can simply be exported to an IF file, or a full verification process can be automated from PragmaDev Studio. In that case, after exporting an SDL model to IF language, PragmaDev Studio calls a script -which probably calls an IF tool on the IF description- and opens a socket waiting for an MSC trace file. To do so, a validation profile for IF is needed.

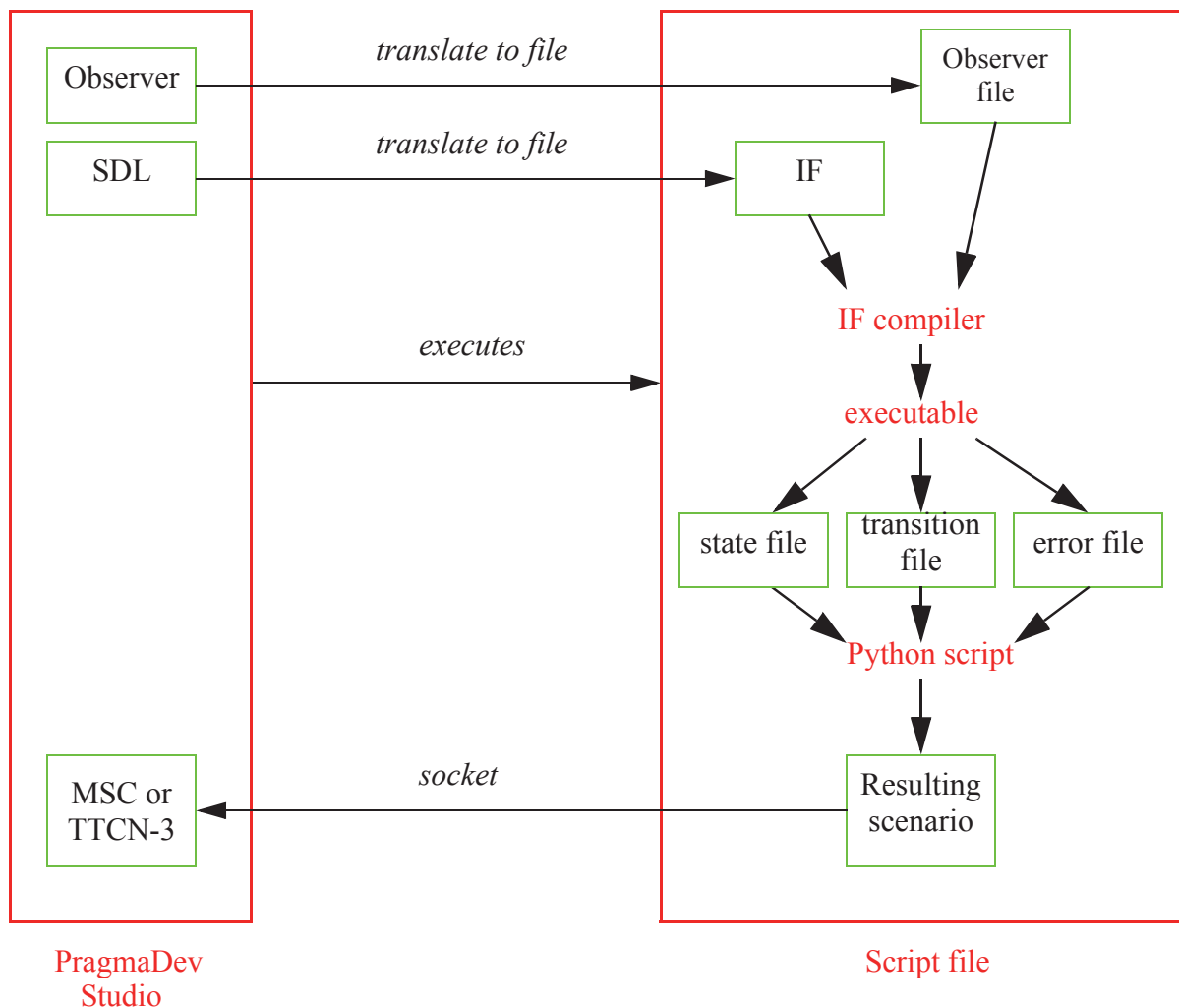
The script or the external executable can be customized via the validation options:



As the current directory is not specified, the executed script should not depend on where it is executed. The MSC trace file format is the one described in the PragmaDev Tracer documentation.

10.5.2.3 Example script for Verimag IFx toolset

An example script for the external model checker is delivered with PragmaDev Studio. This script uses the IFx toolkit from Verimag. This script works as described in the following diagram:

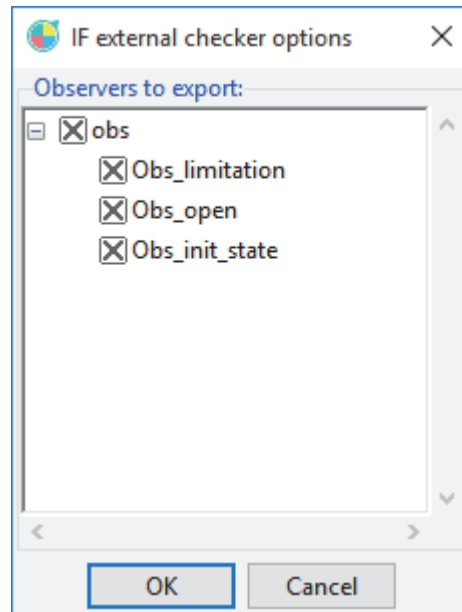


The script can be found in the sub-directory `share/3rdparty/IFx/ifChecker.sh` of the PragmaDev Studio installation directory. It requires an IF observer to specify the properties to check in the SDL system. This observer is specified via the environment variable `RTDS_IF_OBSERVER`, which must contain the full path to the observer file.

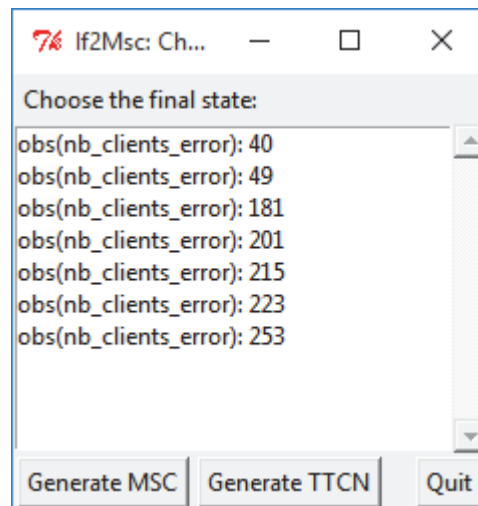
Once a validation profile for IF has been created and the environment variable has been set, call IF toolbox by selecting “Validation / Checking...”. This will successively:

- Export the current project as an IF file;

- Select the observers to test the system. If observers are inside folder, it is possible to select every observers by selecting the folder;



- Run the IFx compiler on this file and the observer file, which will produce an executable;
- Run this executable to actually perform the simulation;
- Get the output from the execution and pass it to a Python script which will analyse the errors if any. This script displays the following window if the simulation found some scenarios that invalidate the conditions:



The numbers displayed are the internal numbers for the global system states identified as errors by the IFx toolkit. For each state number are displayed the name of the observer which has stopped the system and the corresponding error state name.

Generating an MSC of the scenario leading to a given error state is done by selecting the state number in the list and pressing the *Generate MSC* button.

PragmaDev Studio will display a dialog stating an external file is to be imported in the project.

Generating TTCN test cases for one or several scenarios is done by selecting one or several state numbers and pressing the *Generate TTCN* button. PragmaDev Studio will display a dialog stating an external file is to be imported in the project.

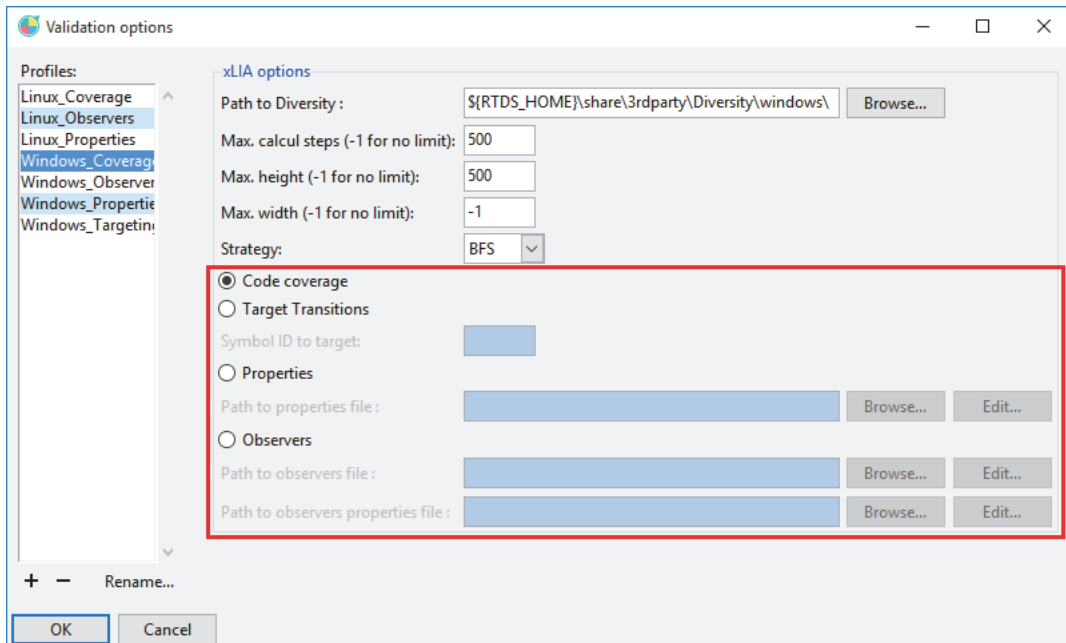
10.5.3 Diversity

PragmaDev and CEA List created a common laboratory that started in September 2013. As a result of this lab, test cases can be generated automatically out of an SDL Z.100 model (i.e., model-based testing). For that purpose the Diversity CEA tool is integrated in PragmaDev Studio environment, but requires a dedicated license. Diversity is based on a symbolic resolution engine that differs substantially from other model checking technologies. SDL Z.100 systems are automatically translated to xLIA files (proprietary CEA file format), and the xLIA files are used as inputs of Diversity. Diversity will process the xLIA description and generate TTCN-3 testcases corresponding to the verifications goals.

Verification goals can be of four different types:

- **Code coverage**
To generate the minimum number of test cases that cover all transitions.
- **Properties**
To generate the test cases verifying a static property. A static property is a verification on the process state, and variables value, and communication actions.
- **Observers**
To generate the test cases verifying a dynamic property. A dynamic property is defined as a state machine called observer. The syntax in the current version of PragmaDev Studio is xLIA. Observers are useful in the case of temporal rules or succession of actions. A property comes with the observers that specifies the targeted observer state.
- **Target transitions**
To generate a test that covers a specific transition in the SDL model.

The goals are set in the validation profile, which is defined in “Validation /Options...” menu:



10.5.3.1 Coverage

This is the simplest profile since the goal is to cover all transitions in the model.

10.5.3.2 Properties expression

Properties must be expressed in a text file. The text file will then be indicated in the *Path to properties file* field in the validation profile. Properties allows verification on variables, processes state and communication actions. Diversity will search and stop when the property is verified in the system.

Properties have to be written in xLIA. The syntax of the properties file is:

```
@stateTest = ${ Property to check };
```

10.5.3.2.1 Process State

To verify the state of a process, the syntax is:

```
schedule#in &spec::ProcessContext.StateName &spec::ProcessContext
```

with ProcessContext the architectural path to the process. For example, to verify if process P1 in block B1 in system S1 is in the state Idle, the property is:

```
schedule#in &spec::S1.B1.P1.Idle &spec::S1.B1.P1
```

10.5.3.2.2 Variable value

To verify the value of a variable, the syntax is:

```
EqualitySymbol spec::ProcessContext.VariableName value
```

with EqualitySymbol the symbol of equality verification (=,!=,<,>).

10.5.3.2.3 Communication action

It is also possible to check a process state or a variable value after a communication action append. The syntax is:

```
$obs
${ CommunicationAction &spec::SystemName.MessageName }
${ Property to check }
```

with CommunicationAction the input or output keyword. For example, to verify that the variable Count in process P1 is at one after message Increment has been received will be written:

```
$obs
${ input &spec::SystemName.increment }
${ = spec::SystemName.P1.Count 1}
```

10.5.3.2.4 Multi Properties

It is also possible to verify many properties in the same time by using &&, || or ! before properties expression:

```
@stateTest = ${ &&
${Property 1}
${Property 2}
};
```

By using &&, Diversity will check if the system can be in a state where all properties are verified at the same moment. With ||, Diversity will check if the system can be in a state where at least one property is verified and with !, only one property must be checked.

10.5.3.3 Observers

Diversity allows to write observers to check more complex properties. Observer are processes that can view everything that happens in the system.

To do so, two files have to be written, one for the observer itself to indicate in *Path to observers file* field, an other for the properties to check on this observer to indicate in *Path to observes properties file* field.

10.5.3.3.1 Observers syntax

Observers have to be written in xLIA. The general syntax is:

```
statemachine< or > ObserverName {
    @machine:
    ...
    observer behaviour
    ...
}
```

The behaviour of the observers are described as state machine. These state machines are a combination of states and transitions to go from one state to another.

The first state of an observer is always the initial state:

```
state< initial > #init {  
    transition {  
        } --> FirstStateName;  
}
```

Then for each state, the syntax is:

```
state StateName {  
    transition TransitionName1 {  
        TransitionTrigger1  
    } --> NextStateName1  
    transition TransitionName2 {  
        TransitionTrigger2  
    } --> NextStateName2  
    ...  
}
```

TransitionTrigger is the action happening in the system which will leads the observer to an other state. Transition triggers can be communication action or guard on state.

The syntax for an input trigger is:

```
:> obs { input MessageName; } [ true ];
```

The syntax for an output trigger is:

```
:> obs { output MessageName; } #provided true;
```

The syntax for a guard on state is:

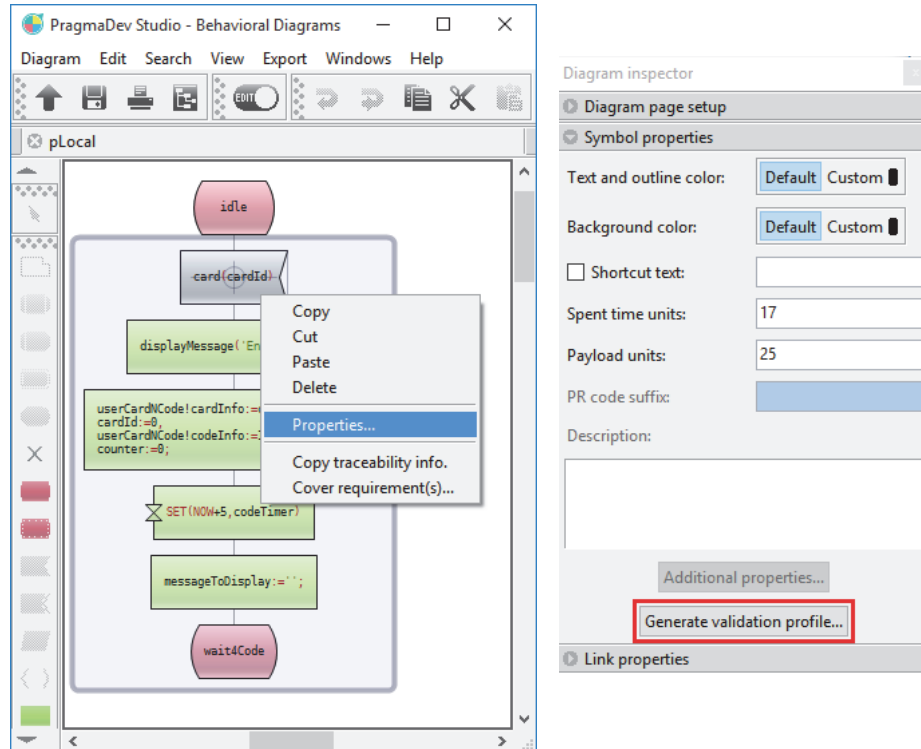
```
guard (: ProcessContext.StateName schedule#in ProcessContext);
```

Because during exploration of the system, observers are considered to be inside the system, system name does not have to appear in **ProcessContext**.

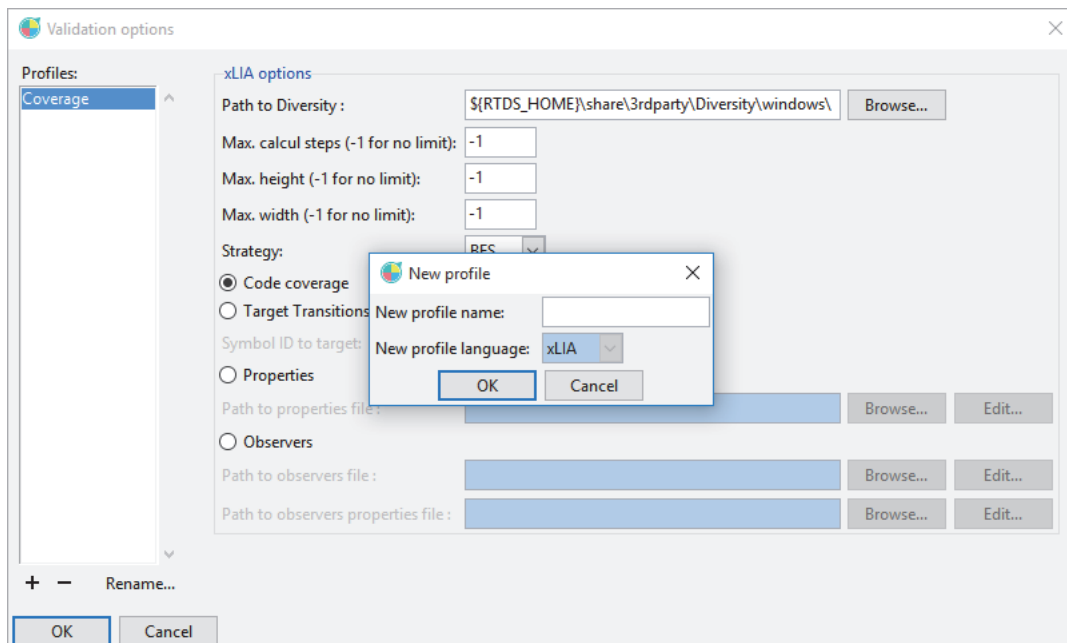
As for properties on system, it is now possible to write properties to test observers.

10.5.3.4 Transition targeting

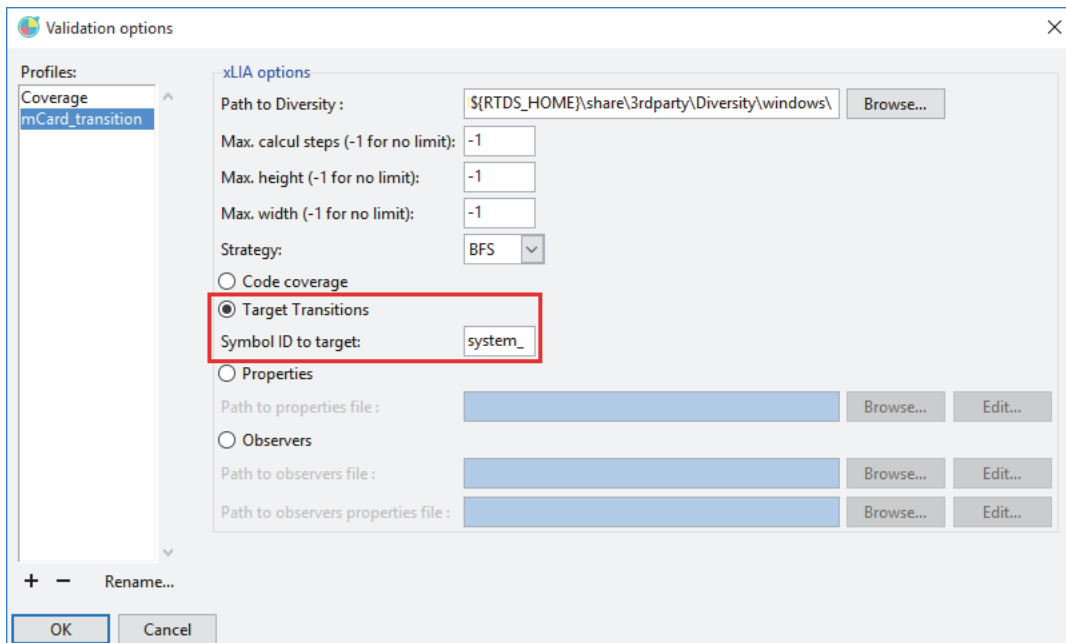
It is possible to ask to Diversity to generate automatically a testcase which will lead to a specific transition in a process. This can be done via “Generate validation profile...” in the properties of the message input symbol starting the transition in the process diagram.



Doing so will trigger the creation on a new validation profile:

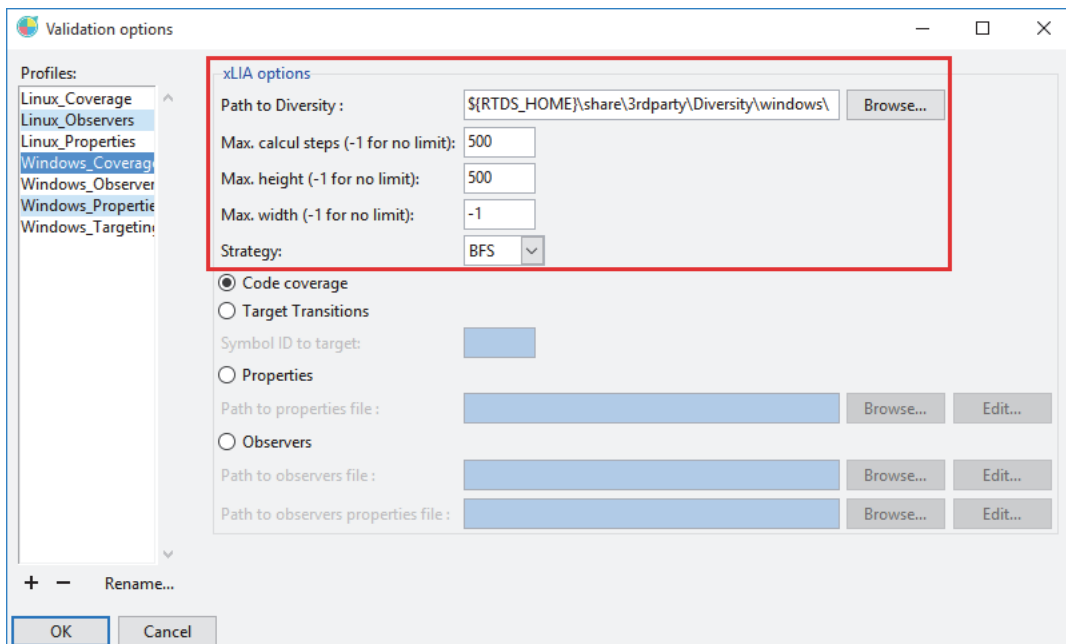


The Symbol ID to target will be filled automatically:



10.5.3.5 General validation properties

There are other options in xLIA profile:



Path to Diversity is the path to diversity executable file.

Max. calcul steps, *Max. height* and *Max. width* are used by Diversity to limit exploration. A value of -1 for these information means there are no limit of exploration.

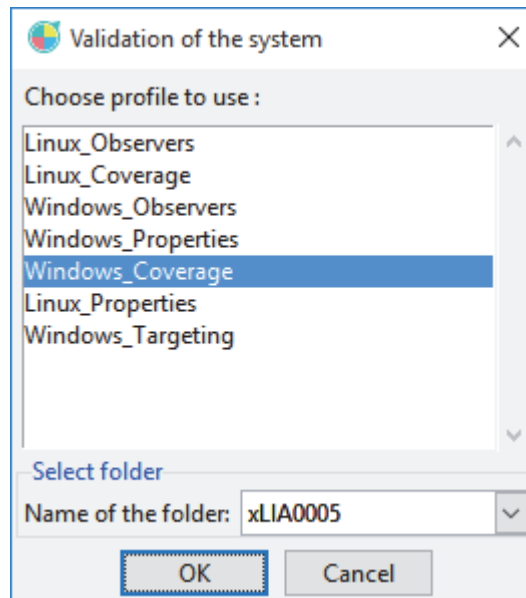
Strategy is for the exploration strategy type:

- DFS: Depth First Search
- BFS: Breadth First Search

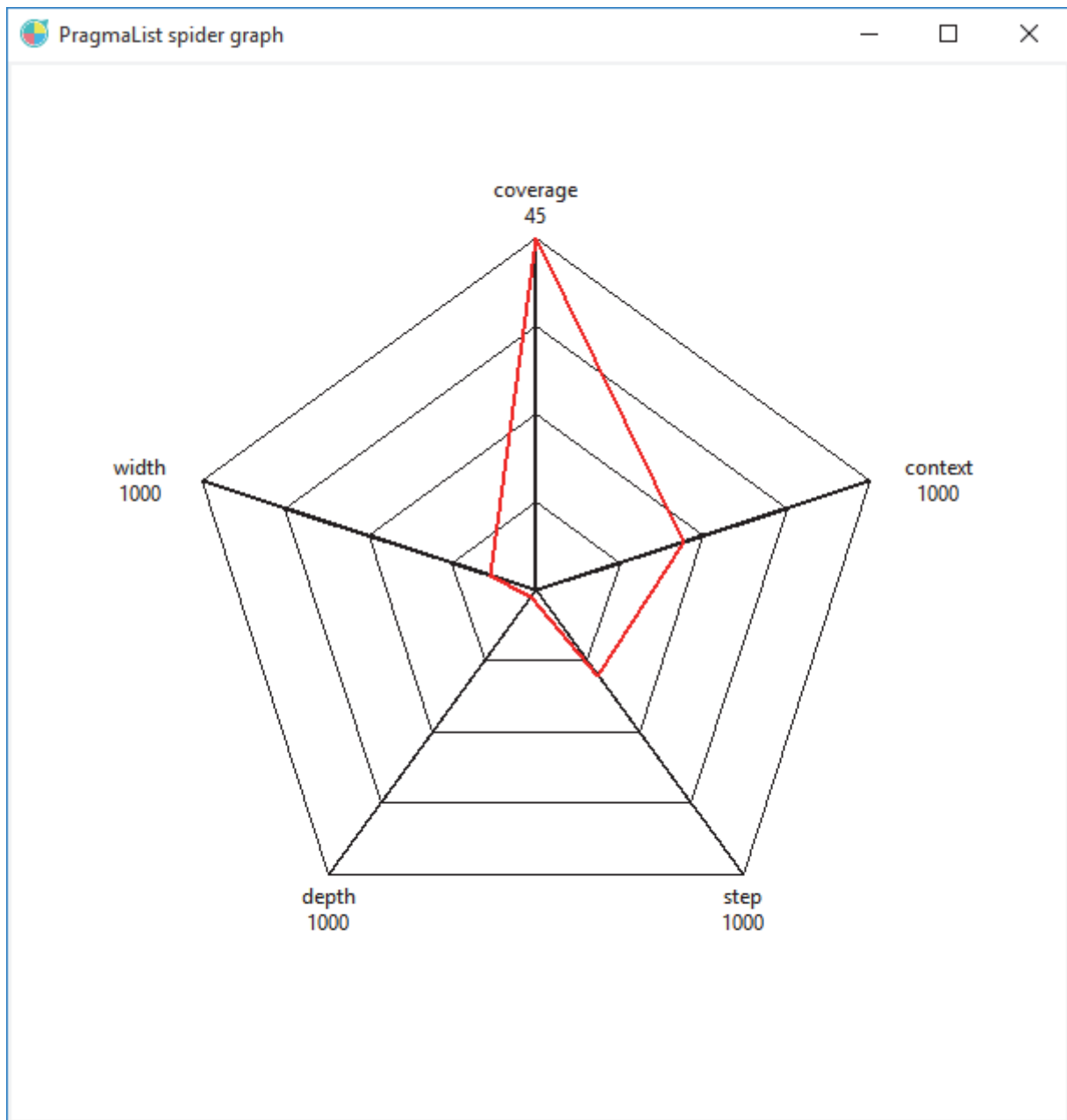
- RFS: Random First Search

10.5.3.6 Results of verification

When validation profile is defined, run the verification by selecting “Validation / Checking...”. It will ask which validation profile has to be used for verification and in which folder TTCN-3 files have to be generated:



PragmaDev Studio will automatically generate xLIA code from the system, and run Diversity to verify the system. During Diversity execution, a graphical view of the system exploration is shown:



This graph shows the progress of Diversity in relation to the options set in the profile. The coverage branch shows how far the exploration is from the goals (in the case of model coverage for example, the number below coverage is the number of transitions in the model). The exploration will end when coverage is reached, or when maximum calcul step, maximum height or maximum width is reached.

The following information appears in a rapport at the end of execution:

- The CONTEXT count
The number of evaluation contexts created (they may not all be in the final sym-

bolic execution graph because of the cut-back which eliminates useless context executions at the end).

- The EVAL count
The number of calculation steps performed.
- The Max HEIGHT reaching
The maximum depth recorded during the symbolic execution.
- The Max WIDTH reaching
The effective max width recognized in respect with contexts effectively evaluated.
- The DEADLOCK found
The number of deadlocks recorded during the symbolic execution.

Followed by the report on the transition coverage and possibly the list of not covered transitions.

In a case of success:

PROGRAM COVERAGE PROCESSOR

All the << 46 >> transitions are covered!

If the objectives are not complete:

PROGRAM COVERAGE PROCESSOR

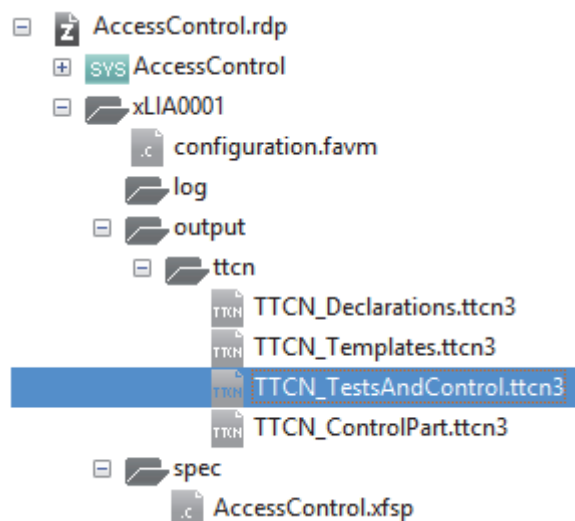
Warning: all the programs are not covered!

Results: << 52 on 76 >> are covered!

List of the << 24 >> transitions none covered:

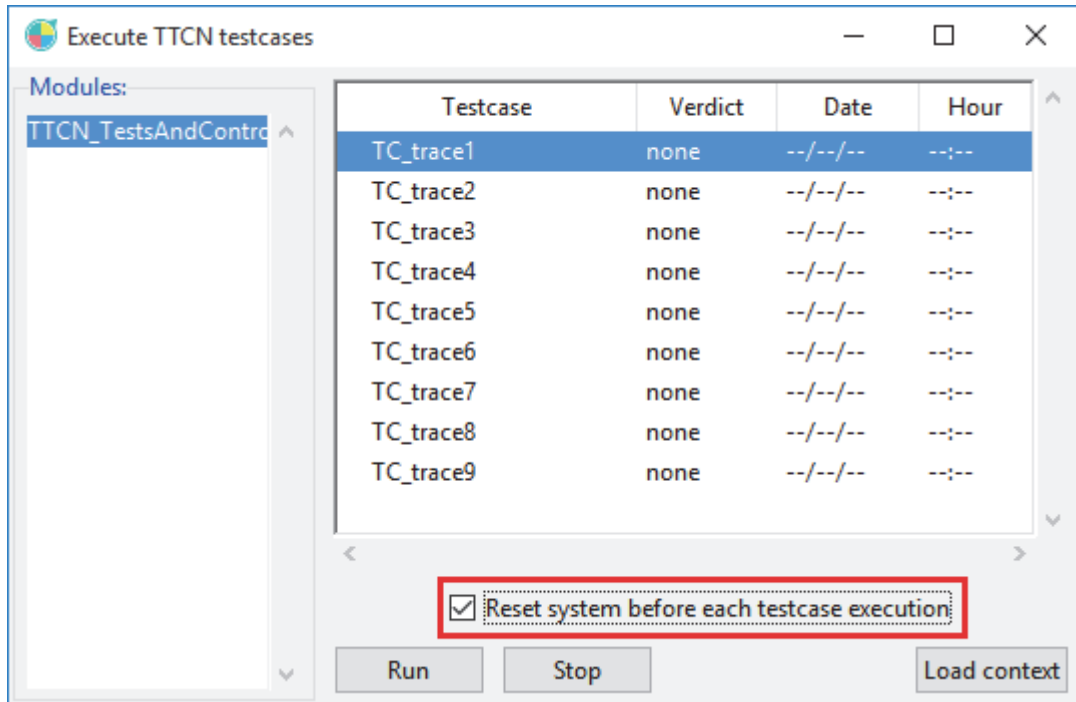
Finally, as a result, many files are generated in the specified folder:

- *configuration.favm*, needed by Diversity to check properties.
- An *.xsfp* file named as the system name in *spec* sub-folder: this file is the xLIA translation of the SDL system.
- Four *ttn* files in *output/ttn* sub-folders, results of Diversity exploration. The testcases are in the *TTCN_TestsAndControl.ttn3* file.




In the case of property check, observer verification or transition targeting, the testcase will lead the system to the wanted state.

In the case of model coverage, several test cases are generated. Please note each test case expect the system to be in its initial state. For that reason when simulating the test cases against the system, check *Reset system before each testcase execution* before running the testcases to restart the system after each testcase execution.



11 - Code coverage results

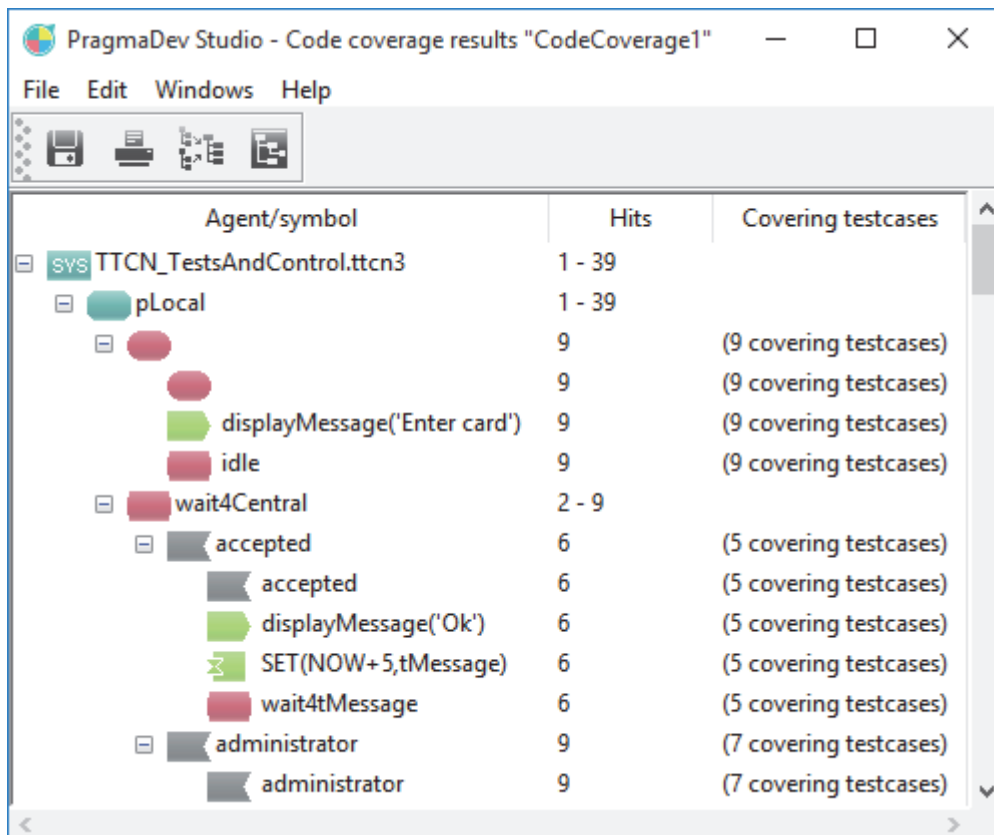
11.1 - Generating code coverage results

Code coverage results cannot be created directly in the project manager. They are always obtained via a debug or simulation session, by clicking on the  button in the debugger or simulator window (see “Code coverage” on page 254 and “Code coverage” on page 290). Note that the option activating code coverage analysis must be checked in the code generation or simulation options for this feature to be available; see “Profiles” on page 157 and “Simulator options” on page 267.

Generating the code coverage results for a debug or simulation session will automatically create a code coverage results node in the project manager and open it. Note that the results set has to be saved in order to be kept. If the code coverage results viewer window is closed without saving, the code coverage results node will disappear from the project.

11.2 - Code coverage results viewer window

The window allowing to view the results of a code coverage analysis looks like follows:



Agent/symbol	Hits	Covering testcases
sys TTCN_TestsAndControl.ttcn3	1 - 39	
pLocal	1 - 39	
	9	(9 covering testcases)
	9	(9 covering testcases)
displayMessage('Enter card')	9	(9 covering testcases)
idle	9	(9 covering testcases)
wait4Central	2 - 9	
accepted	6	(5 covering testcases)
accepted	6	(5 covering testcases)
displayMessage('Ok')	6	(5 covering testcases)
SET(NOW+5,tMessage)	6	(5 covering testcases)
wait4tMessage	6	(5 covering testcases)
administrator	9	(7 covering testcases)
administrator	9	(7 covering testcases)

Code coverage analysis result window

The tree shows:

- At its first level, all processes in the system;
- At its second level all states, connector entries and start symbols in each process;
- At its third level, all message inputs, message save and continuous signals for each state;
- At its last level, all symbols in the transition.

For each node is displayed the minimum and maximum number of hits for the symbol or transition or process. If the minimum and maximum are the same, a single number is displayed. Colors allow to identify at once non-covered symbols or groups: if it has not been covered at all, it will appear as red; if it has only been partially covered, it will appear as orange.

A third column in the viewer gives the testcases covering the symbol on the line if any and available. Information about testcase coverage is only available after a co-simulation of a set of TTCN testcases and a SDL system. See “Co-simulation code coverage analysis” on page 127. Only the number of covering testcases is displayed in the viewer. The full list of testcases is displayed in a popup menu when right clicking on the cell. The menu items allow to display the testcases themselves in a text file editor.

The tree may be expanded, collapsed and sorted using the “Edit” menu. Double-clicking on a node will display the corresponding symbol if any.

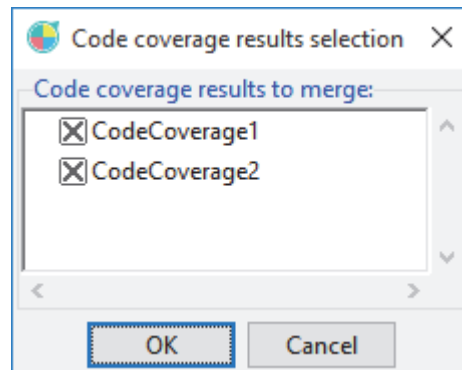
NB: the states displayed at the second level of the tree are just a way to summarize the information for all transitions for this state. Therefore:

- The numbers displayed for the node are not the number of times the process went into that state. It's the number of transitions with this state as initial state executed by the process.
- There is no symbol corresponding to such a node: for a given state, each transition may be represented with a specific state symbol in the diagram, but only one state symbol will appear in the code coverage results. So it can't be associated to a single symbol.

11.3 - Merging code coverage results

PragmaDev Studio allows to merge the different code coverage results sets obtained in several debug or simulation sessions. To do so, open one of the sets to merge and select

“File / Merge...” in the menu, or click on the  button. A dialog appears, allowing to select the other sets to merge with the current one:



The hierarchy is the one in the project manager, showing only the code coverage results. Each set can be selected by checking the box in front of it. The checkbox in front of a package or folder name allows to select all the code coverage results sets in this package or folder.

Once validated, all the selected sets are merged with the current one, and a new results set is created in the project for the merge results and opened immediately. As code coverage results created during debug or simulation, this set has to be saved to be actually inserted in the project.

12 - TTCN-3 support

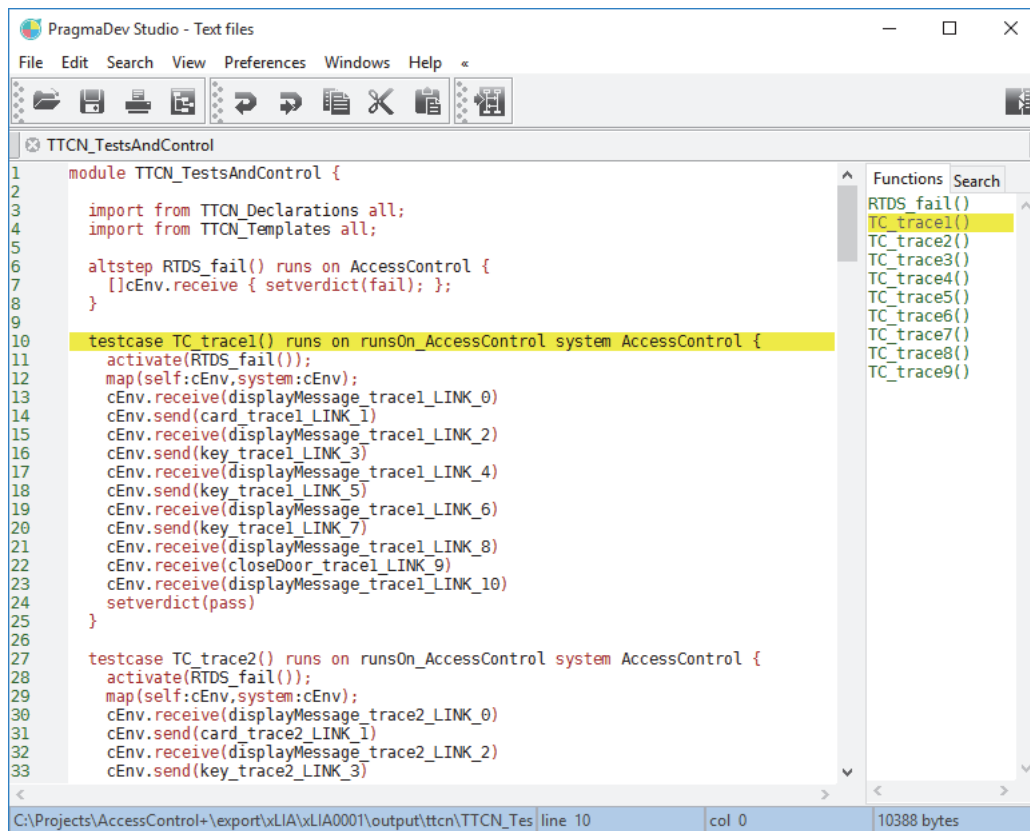
12.1 - Levels of support

PragmaDev Studio supports TTCN-3 test suites in both the editors and the simulator:

- Source files in TTCN-3 core language can be included in a project.
- Full syntax coloring and checking is available for these files.
- TTCN-3 test suites can be simulated in the PragmaDev Simulator along with the system they test.

12.2 - TTCN-3 core language file editor

TTCN-3 core language source files can be included in PragmaDev Studio projects and edited with the included text editor:

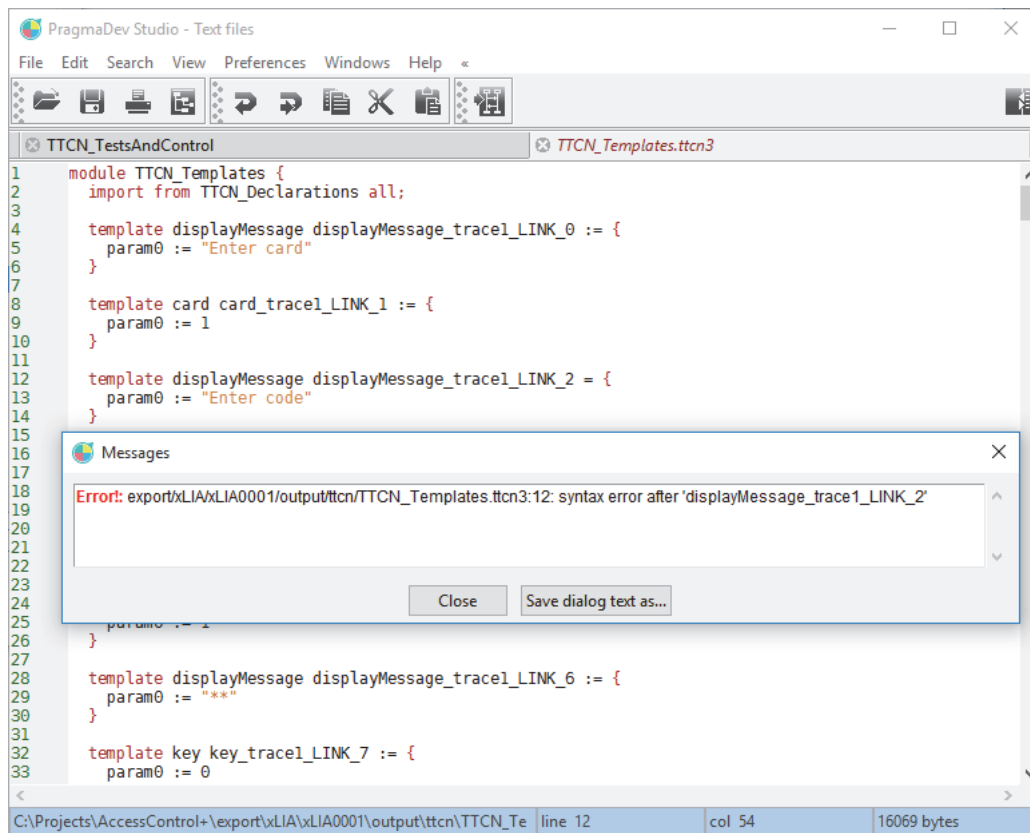


```
1 module TTCN_TestsAndControl {
2
3   import from TTCN_Declarations all;
4   import from TTCN_Templates all;
5
6   altstep RTDS_fail() runs on AccessControl {
7     [!cEnv.receive { setverdict(fail); };
8   }
9
10  testcase TC_trace1() runs on runsOn_AccessControl system AccessControl {
11    activate(RTDS_fail());
12    map(self:cEnv.system:cEnv);
13    cEnv.receive(displayMessage_trace1_LINK_0)
14    cEnv.send(card_trace1_LINK_1)
15    cEnv.receive(displayMessage_trace1_LINK_2)
16    cEnv.send(key_trace1_LINK_3)
17    cEnv.receive(displayMessage_trace1_LINK_4)
18    cEnv.send(key_trace1_LINK_5)
19    cEnv.receive(displayMessage_trace1_LINK_6)
20    cEnv.send(key_trace1_LINK_7)
21    cEnv.receive(displayMessage_trace1_LINK_8)
22    cEnv.receive(closeDoor_trace1_LINK_9)
23    cEnv.receive(displayMessage_trace1_LINK_10)
24    setverdict(pass)
25  }
26
27  testcase TC_trace2() runs on runsOn_AccessControl system AccessControl {
28    activate(RTDS_fail());
29    map(self:cEnv.system:cEnv);
30    cEnv.receive(displayMessage_trace2_LINK_0)
31    cEnv.send(card_trace2_LINK_1)
32    cEnv.receive(displayMessage_trace2_LINK_2)
33    cEnv.send(key_trace2_LINK_3)
```

Each TTCN-3 source file should contain a single module, which must have the same name as the file itself. For example, a TTCN-3 file named `TTCN_TestsAndControl.ttcn3` must contain exactly one TTCN module, named `TTCN_TestsAndControl`.

Since there is no notion of package in TTCN-3, all modules in a test suite should be put in the same package to be able to import each other.

RTDS supports full syntax check of the TTCN-3 files via the menu “File / Check syntax semantics...” in the source file editor:



12.3 - TTCN-3 co-simulation

TTCN-3 test suites in SDL projects can be simulated along with the system they test. To do so, the test itself must be selected and the simulator must be run as described in “Launching the PragmaDev Simulator” on page 268. The system under test described in the test will also be run automatically. The test suite will be seen as a process in the SDL simulator, and all features will be available, such as MSC tracing, breakpoints, local variables display, etc.

There are a few conventions to follow in the test to connect it to the system, and also some restrictions on the TTCN-3 features that can be simulated. These are detailed in the next paragraphs.

12.3.1 Conventions

For TTCN-3 tests to be able to communicate with the system under test, some naming conventions must be followed in the test itself:

- All test cases must run on a component which has a type with the same name as the SUT. So if your system is named `AccessControl`, you must have a component type named `AccessControl` in your test suite, and all test cases must run on it.

- Ports defined in the TTCN-3 component must have the name of the channels connected to the environment in the SUT. So if you have a single channel named `cEnv` in your system connected to the environment, your component type definition must include a port definition with the name `cEnv` for the port. The name for the port type is not significant and can be anything.
- Messages in the SDL system are represented by types in the TTCN-3 test suite. The name for the TTCN-3 type must be the name for the message as declared in the SDL system. For messages with parameters, the TTCN-3 type must be a record, with as many fields as there are parameters in the message. The names for the fields are not significant, but their order is: the first message parameter will be the first field in the type, the second parameter the second field, and so on... The type mapping between SDL and TTCN-3 is described in the reference manual.

For example, if a message named `close` with a single `Integer` parameter is declared as going out of the system via channel `cEnv`, the TTCN-3 test suite must define the following type:

```
type record close
{
  integer param1
}
```

and the type `close` must be declared as incoming in the port type for the port `cEnv`.

Messages without parameters are a bit trickier to handle, as there is no empty type in TTCN-3. The convention is then to create an enum type with the name of the message, which has a single possible value with different name than the type. For example, if a message `refused` with no parameter is declared as incoming via channel `cEnv` in the SDL system, the TTCN-3 test suite can define the following type:

```
type enum refused
{
  e_refused
}
```

and the type `refused` must be declared as outgoing in the port type for the port `cEnv`.

- TTCN-3 and SDL co-simulation also allows test suites to get operator calls from the SDL system via the `getcall/reply` operations in test cases. The signatures declared in the test suite must have the same name as the operators in the SDL system, with the same parameters in the same order and with the same name and type.

As SDL does not support `in/out` or `out` parameters in operators, they are also not supported in signatures for simulation. All signatures must also be declared as incoming in port types, since there is no way to make synchronous calls to the SDL system. So the `call/getreply` operations are not supported in TTCN-3 test suites for simulation.

Even if `getcall/reply` operations are made on ports in TTCN-3, the port is actually ignored during simulation: all operator calls can be received on any port, and a `reply` operation will always answer to the last received `getcall` operation.

This is due to the fact that operator calls have no connection with channels in the SDL system, and are therefore not connected to the test in any way. So all operator calls will actually be sent to the test, which will receive it if there is any pending `getcall` operation.

Please note that there is no need to handle operator calls in the test suite: if no procedure or mixed port type is defined in the test, the operator calls will be handled the “normal” way (ask answer from user or XML-RPC call).

12.3.2 Restrictions

The following features in TTCN-3 test suites are supported for simulation:

- Modules are supported, but not module parameters (`modulepar`). The only language supported for external modules (`language` clause) is ASN.1.
- Imports are supported, but restriction clauses are ignored: The import is accepted but will import the full module (a warning is issued during the byte-code generation). If present, the language clause has to specify ASN.1 as the language, or the import will fail. The exact syntax for ASN.1 in the language clause is “ASN.1:<year>”, where <year> is a 4-digit number. The specified year has no effect.
- Groups are supported, but do nothing.
- The notation `module-name.identifier` for imported identifiers is not supported. There must be no ambiguity in imported names.
- Component types are supported, as well as all declarations within them.
- Port types with any type are supported. However, signatures in procedure or mixed port types can only be declared as incoming. The keyword `all` for incoming or outgoing messages or signatures is also not supported.
- All basic types are supported except `anytype`, `address`, `default`, and `objid`. Subtypes of basic types with restrictions are also supported, as in:
`type integer index_type (1 .. 16);`
The special value `infinity` is only valid in constraints, not as variable value. Precision for `bitstring` and `hexstring` types are not yet handled, so operations on these strings won’t produce the expected result (concatenation, indexed access, `lengthof`, shifting, ...).
- All complex types are supported: `record`, `record of`, `set`, `set of`, `union`, `enumerated` and arrays. Optional fields in `record` or `set` types are supported, and so is the special value `omit` and the predefined function `ispresent`. However, the special values `*` and `?` in templates are equivalent, and `ifpresent` is not supported.
Recursive types are not supported.
Type compatibility is strict, meaning that it is impossible to assign a value to another one if both are not declared with the same type, or compatible subtypes of the same type.
- Signatures are supported, but with no `out` or `inout` parameters. Non-blocking signatures and exceptions are also not supported.

- Templates are supported, including modifies clause and template parameters. Template operations match and valueof are also supported, as well as the special type-name:{...} notation.
However, constraints in templates set via complex values are not supported. For example:

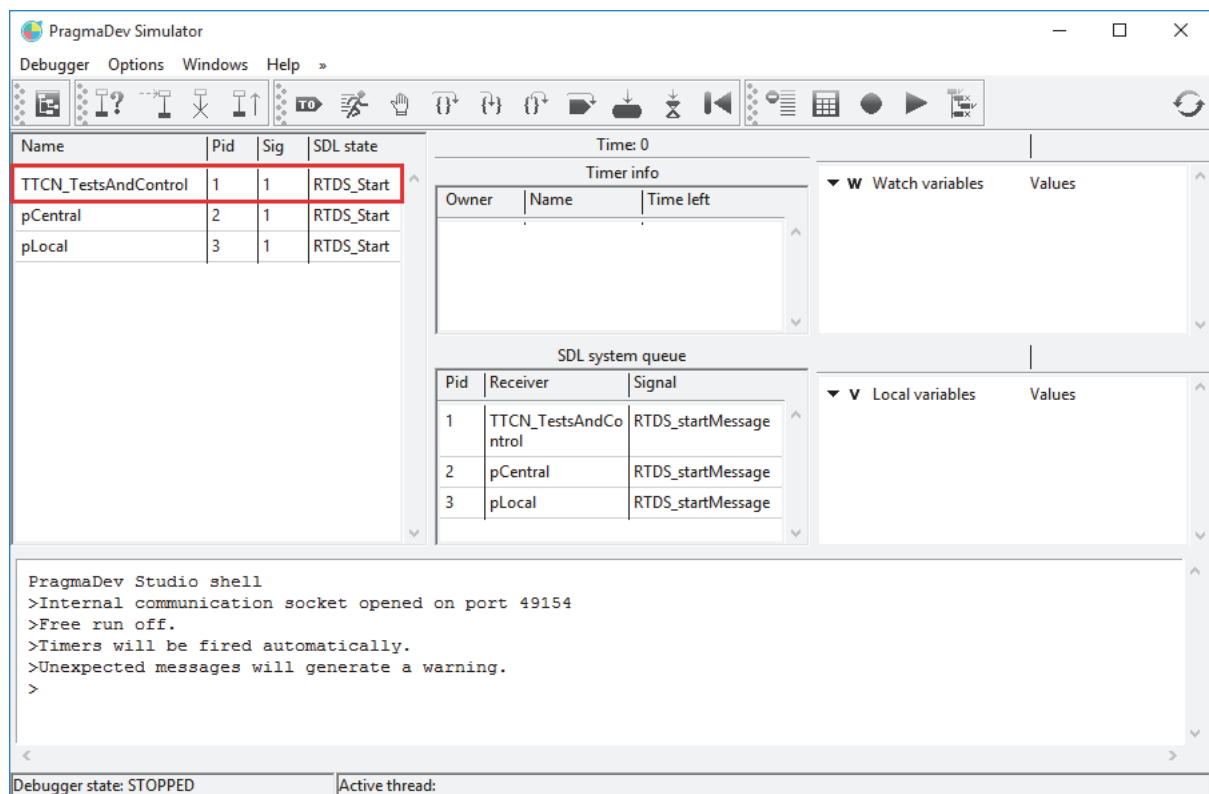
```
type record Point { integer x, integer y };  
type record Segment { Point p1, Point p2 };  
const Point origin := { 0, 0 };  
template Segment origin_segment := {p1 := origin, p2 := ?};
```



will generate an error, as constraint on p1 in origin_segment is specified via the complex value origin.
Simple constraints on strings specified with pattern are not supported, but the regexp predefined function is not.
Complex constraints of record of or set of types are not supported (subset, superset, complement, ...).
Passing templates as parameters to test cases or functions is also not supported.
- Constants are supported.
- Module control parts are supported.
- Test case invocation is supported, but specifying a time-out will not work.
- Test cases are supported, including test case parameters, runs on and system clauses
- All verdict handling is supported, including automatic verdict setting to error on runtime errors in test cases.
- Functions and altsteps are supported
- Variables are supported with no restriction.
- Timers are supported, as well as all operations on timers, except read.
- Port operations are supported, except stop, start, check, catch, call & getreply. A port must always be specified for these operations; specifying any port and all port instead of a port name is not supported.
- All statements in test cases, functions and control parts are supported: if, while, do/while, for...
- Most predefined functions are supported, including all conversion functions.
- Logging is supported but not configurable: Today, it will always print a message in the simulator shell.
- Alternatives are supported, including guard conditions on triggers, but [else] alternatives are not. repeat statements within alt are supported.
- Concurrent testing is partially supported: Components can be created and started and port connections can be established (map/unmap, connect/disconnect). Test verdicts will be handled correctly when there are several test components. However:
 - Both ports have to be specified for disconnection operations;

- The status for test components cannot be tested (running, done and killed operations on components);
- Components cannot be stopped or killed from the outside (stop and kill operations);
- Components created as alive are not supported;
- sender, from & to clauses in port operations are not supported.
- with clauses are ignored.

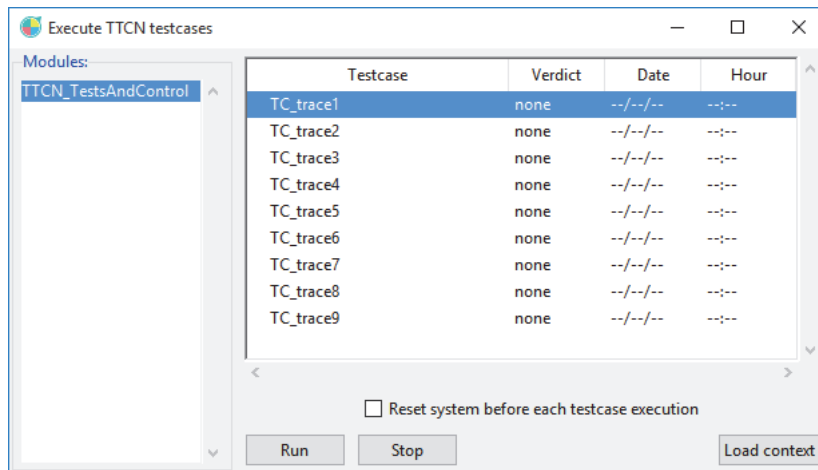
12.3.3 Simulation

As said above, running a test suite along with the system it tests is simply done from the project manager by selecting the main test module and running the simulator on it. The simulator window is the standard one and its behavior is exactly the same as the one used for SDL simulation. The test suite will simply appear as a single entry in the list of running instances:



To start the simulation, hit *Run the system*  and the control part of the selected TTCN module will be executed against the system under test.

Also, a specific window is started when a simulation is launched on a TTCN-3 module.



This window will show all available testcases in the module selected for the simulation and all its imported modules. Also if the selected module has a control part it will be shown.

Through this interface, the control part can be executed as with the simulator window, but a specific testcase can also be executed. To execute a testcase, select the module in which it is, then select the testcase and *Run* it.

Load context is used to load the context of a specific testcase. For example, if a testcase is selected and its context loaded, it is then possible to make a step-by-step execution into the simulator window.

After execution of any element, its verdict and date of execution are shown into this interface. `TTCNexecution.log` file, in the project directory, will save all information of the TTCN execution.

All features in the SDL simulator such as stepping, breakpoints, local variables display or MSC tracing are then available for TTCN code:

The screenshot displays the PragmaDev Studio interface with three main windows:

- MSC Tracer:** Shows a sequence diagram with three lifelines: pCentral (2), pLocal (3), and TTCN_TestAndControl (1). The diagram includes messages like `displayMessage({param1=enter code})`, `wait4Code`, `key({param1=1})`, and `displayMessage({param1=**})`. The time axis on the left shows ticks from 0 to 25.
- Debugger:** Contains a table of components and their states:

Name	Pid	Sig	SDL state
TTCN_TestAndControl	1	1	RTDS_Start
pCentral	2	1	RTDS_Start
pLocal	3	1	RTDS_Start

 Below the table, there are sections for "Timer info", "Watch variables", "SDL system queue", and "Local variables".
- Text files:** Displays the source code for `TTCN_TestAndControl.ttcn3`. The code includes a `testcase` block with various `activate`, `map`, `cEnv.receive`, `cEnv.send`, and `setverdict` statements. The current line of execution is highlighted at line 20: `cEnv.send(key_trace1_LINK_7)`.

As TTCN test cases do not have states, state change symbols in MSC traces are used to display test case verdicts.

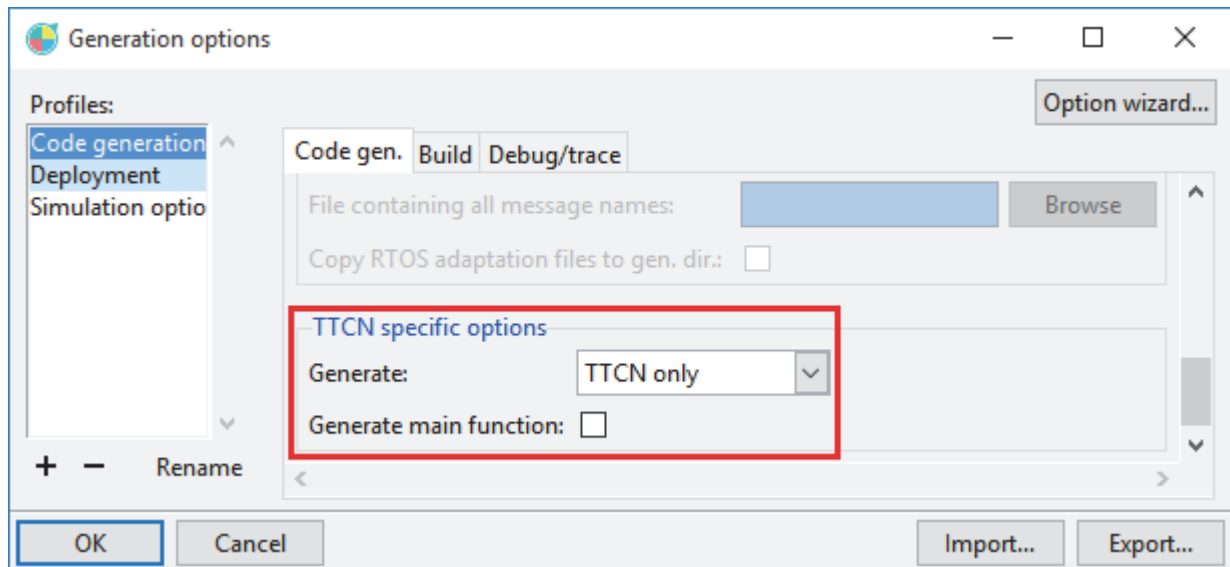
12.4 - C++ code generation

TTCN-3 modules can be generated as C++ code. It is possible to generate only the C++ code for TTCN-3 modules, or generate the modules with the corresponding SDL/SDL-RT system.

To generate C++ code from a TTCN-3 file, select the TTCN-3 file to generate and then choose either "If needed..." or "Force..." from the "Generation / Generate code" menu.

12.4.1 Stand alone

To generate TTCN-3 only, in the generation options select *TTCN only* in the Generate of the *TTCN specifics options*:



There is then some adaptations to do to adapt the tests with the SUT:

- For the communication from the tests to the SUT, macro based on the name of the TSI ports are used and need to be declared by user.
- For the communication from the SUT to the tests, messages have to be stored in a global variable.
- In the generation option, the macro `RTDS_TTCN_SUT_INIT` specifies the start function of the SUT.

For further details, see the PragmaDev Studio Reference Manual.

12.4.2 Combined with SDL

To generate SDL/SDL-RT system with the TTCN-3 file, select *TTCN + SDL/SDL-RT* in *TTCN specifics options*. Adaptation between TTCN and the SUT is automatically done.

12.4.3 Combined with SDL-RT

To generate C++ code for TTCN and SDL-RT system, all types declaration have to be done in a ASN.1 file. An import must be done in the TTCN-3 file, for example:

```
import from ASN1FileName language "ASN.1:2002" all;
```

All types declared in ASN.1 can be used with the same name in TTCN-3. The only exception is that each dash character (“-”) presents in ASN.1 has to be replaced by an underscore (“_”) in TTCN-3 module.

12.4.4 Generate the main function

PragmaDev Studio allows to automatically generate main function in `RTDS_TTCN_main.c`. If using this main function, generated code can be interactively executed. To automatically generate

main function, check *Generate main function* in the generation options. For further details, see the PragmaDev Studio Reference Manual.

12.4.5 RTOS integration

Only generation for windows *win32* and *posix* are supported.

12.4.6 Conventions.

To generate code with the associated system, some conventions must be followed:

- The name of the type for the TSI component must be the same than the SDL system.
- Ports defined in the TTCN-3 component must have the name of the channels connected to the environment in the SUT.
- Messages in the SDL system are represented by types in the TTCN-3 test suite. The name for the TTCN-3 type must be the name for the message as declared in the SDL system.

12.5 - TTCN-3 automatic generation

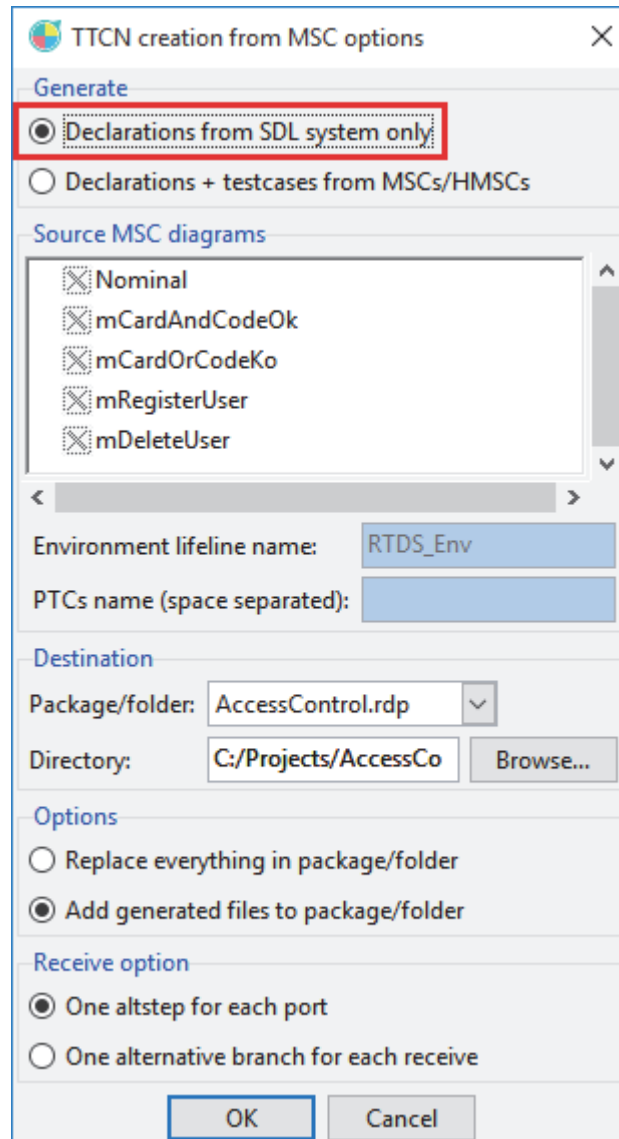
PragmaDev Studio allows to automatically generate TTCN-3 from an SDL/SDL-RT system or from MSCs and/or HMSCs.

12.5.1 From an SDL/SDL-RT architecture

The data types defined in the SDL system and used at system level when communicating with the environment can be automatically translated to TTCN-3 data types. In the case of ASN.1 data types the same definition file will be used in SDL, SDL-RT, and TTCN-3.

Please note that in order to test an SDL-RT system with TTCN-3, the data types used for external message parameters must be declared in ASN.1.

Go to the “Project / Generate TTCN...” menu and select *Declarations from SDL systems only*:



One file will be generated:

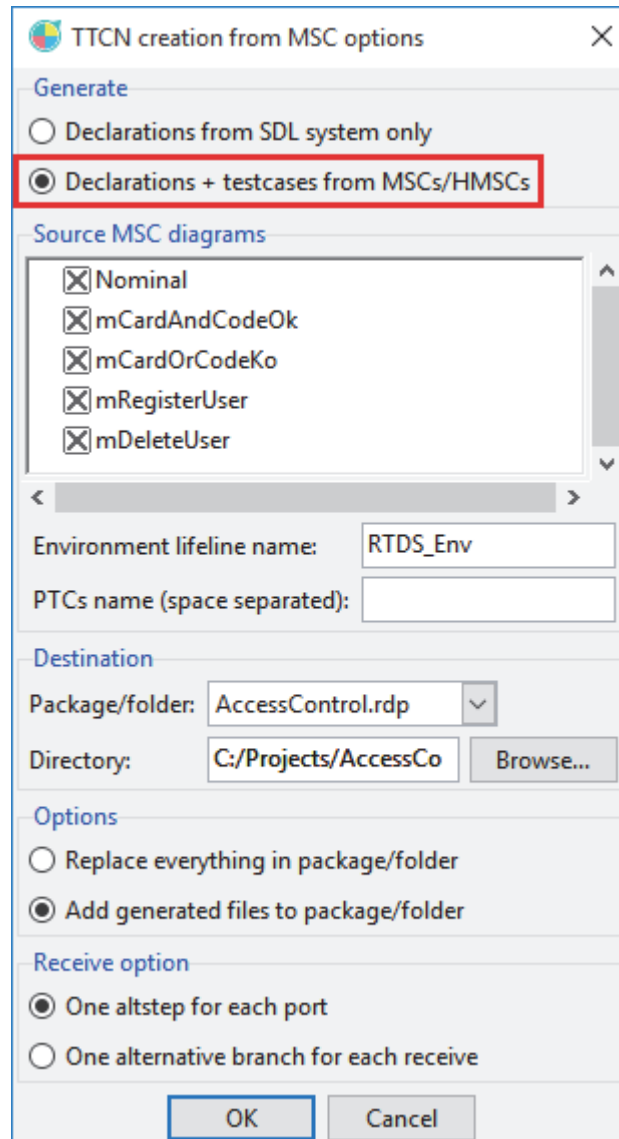
- TTCN_Declarations.ttcn
 - declaration of the necessary data types for message parameters,
 - declaration of the record types for all SDL messages,
 - declaration of ports (one for each SDL/SDL-RT channel connected to the environment),
 - declaration of TSI component type.

This file is overwritten for each TTCN-3 generation.

12.5.2 From MSCs and/or HMSCs

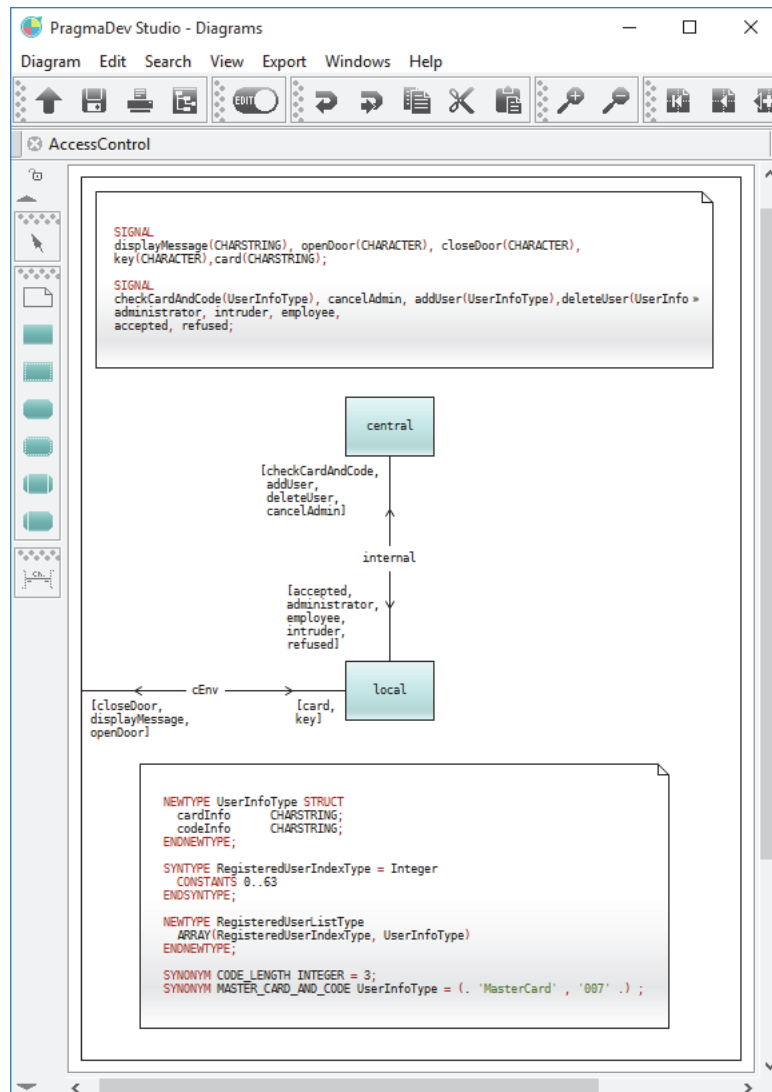
From MSCs or HMSCs, it is allowed to generate complete TTCN-3 testcases. A system level SDL architecture defining the messages and their parameters is necessary.

Go to the “Project / Generate TTCN...” menu and select *Declarations + testcases form MSCs/HMSCs*:

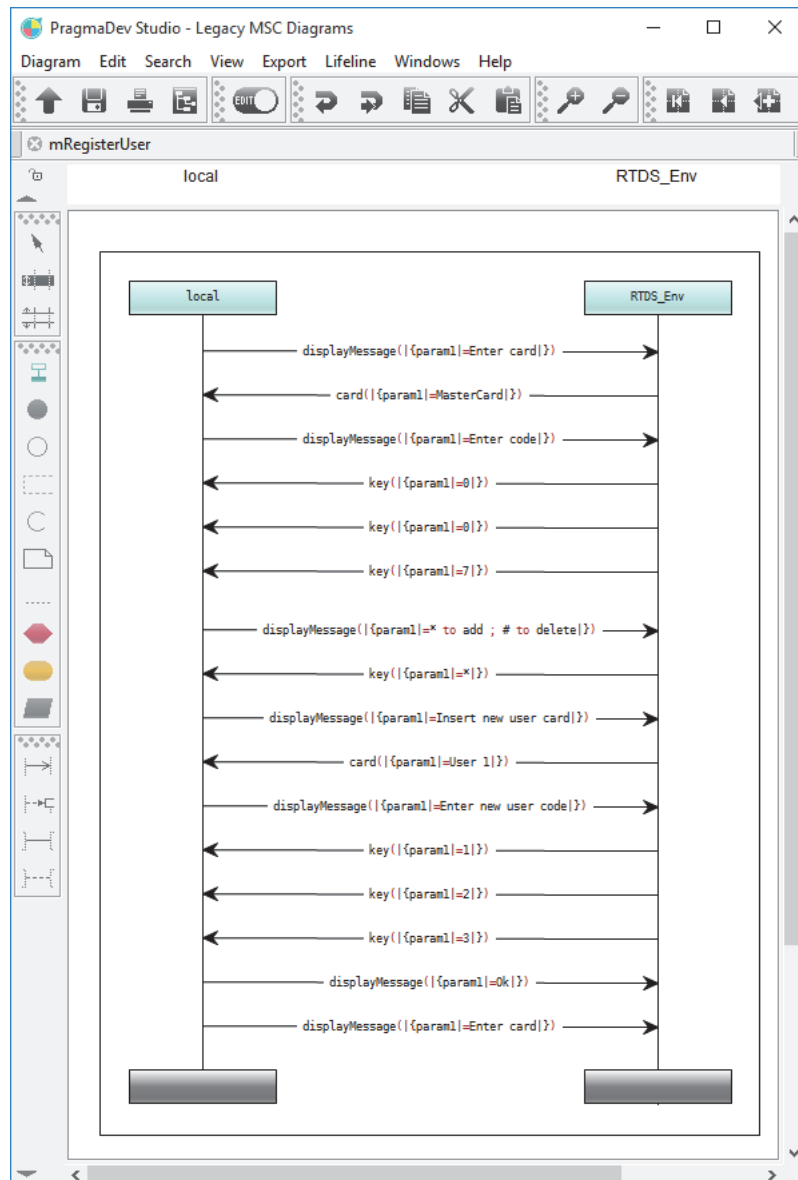


Select the MSCs/HMSCs you want to generate as TTCN-3 testcases. Inform in *Environment lifeline name* the name of the lifeline representing the environment into the MSCs. It is also possible to generate TTCN-3 not just as a unique component running against the system, but as different components. To do so, inform in *PTCs name* the name of the lifeline in the MSC that must be generate as TTCN parallels components.

For example, for the current system:



with the current MSC:



It is possible to generate a TTCN-3 simple testcase by informing only the environment lifeline named `RTDS_Env`. By doing so, a testcase simulating the `RTDS_Env` behavior will be generated and will be able to communicate with the entire SDL system (entry point is in our example channel `cEnv`).

But it is also possible to generate other components in our TTCN-3 testcases. For example, if only `local` has to be tested, input `local` in *PTCs name*. By doing so, the generated testcase will not only simulate `RTDS_Env` behavior, but also `local` behavior as a parallel component.

In every case, four files will be generated:

- `TTCN_Declarations.ttcn`
As described above.
- `TTCN_Templates.ttcn`
Declaration of all needed templates for testcases execution.

- `TTCN_TestsAndControl.ttcn`
Test cases and control part.
- `TTCN_CControlPart.ttcn`
If several generations are done, this file will contain a control part executing all generated testcases.

The remaining options of the dialog are:

- Options
 - *Replace everything in package/folder* will delete all TTCN-3 files in destination package/folder and add new generated files.
 - *Add generated files to package/folder* will only add new generated files without affecting other files, except `TTCN_Declarations.ttcn` that will always be re-generated, and `TTCN_CControlPart.ttcn` that will be consolidated.
- Receive option
 - *One altstep for each port* will generate a single `altstep` for handling all non-matching cases.
 - *One alternative branch for each receive* will put each receive in its own `alt` for handling all non-matching cases.

12.5.3 From a complete SDL system via model checking technology

It is possible to generate TTCN out of an SDL system. The basic process is to translate the SDL system to IF, to define test objectives as IF observers, and run the Verimag IFx tool to generate all possible paths to fulfill the test objective. The example script file described in “Example script for Verimag IFx toolset” on page 313 can generate TTCN out of the resulting paths.

13 - Index

B

Button bars 59
 detaching 60

C

Class

 attribute definitions 94
 constructor definition 95
 operation definitions 95

CMX RTX profile 169

Code coverage

 generation option 158
 getting 254, 290

Code generation 153

Connector

 syntax 152

Continuous signals

 syntax 144

D

Diagram

 editor 55
 frame 56
 inserting links 59
 inserting symbols 59
 page setup 61

Directory

 importing in project 13

E

External tools

 commands 37
 definition 36

F

File types

 custom 10
 supported 9

FreeRTOS profile 200

G

Generation profile

 CMX RTX 169
 FreeRTOS 200
 Nucleus 196
 OSE Delta 189
 OSE Epsilon 193
 Posix 175
 ThreadX 172
 uITRON 181, 184
 VxWorks 164
 Windows 178

H

HTML export

 project 26
 single element 26

M

Memory

 good coding practise 227

Message

 declaration 140
 syntax 144

MSC

 Insert / Remove time 77
 Timers 77

N

Next state

- syntax 143

Nucleus profile 196

O

OFFSPRING 153**OSE Delta 4.5.2 profile** 189**OSE Epsilon profile** 193

P

Package 8**PARENT** 153**Partitions** 60**Posix profile** 175**Preferences** 16**Procedure**

- declaration 142

- syntax 150

Process

- declaration 141

- syntax 150

Project 7

- exporting to HTML 26

- manager 7

- tree

 - adding nodes 12

 - rearranging 11

Prototyping GUI 132**Publications**

- options 29

R

Refresh

- options 239

- SDL simulator options 278

Reqtify 40

S

Save

- syntax 149

Scheduler 209, 215**SDL keywords** 153**SELF** 153**Semaphore**

- declaration 141

- syntax 149

SENDER 153**Simulator**

- Options 267

T

Task block

- syntax 143

Tasking integration 169**ThreadX integration** 172**Timer**

- syntax 149

Timers

- declaration 140

Tool bars 59

- detaching 60

- sticky mode 60

Tornado integration 164**Traceability** 40

U

uITRON profile 181, 184**UML**

- code generation 203

- diagrams 92

V

VxWorks profile 164

W

Windows profile 178

X

XML-RPC

operators and external procedures 268