PRAGMADEV STUDIO

USER MANUAL

PRAGMADEV
modeling and testing tools

Contents

1 - Introduction

1.1 - Scope

PragmaDev provides a set of modeling and testing tools that helps managing complexity inherent to developing state of the art systems. *PragmaDev Studio* integrates different tools based on international standards. The tools target architects/system engineers, developers, and testers.

- *PragmaDev Specifier* helps system engineers to unambiguously specify and verify the functionalities of the system, and define the best architecture for performance or energy efficiency. The technology used results in a graphical and executable model. Verification and validation of the dynamic of the system is done with the integrated simulator, and the best architecture is analyzed with a unique performance analyzer.
- *PragmaDev Developer* helps software designers to write maintainable and self documented code. The technology used for development describes the architecture and contains a graphical view of the main paths of execution down to the code itself.
- *PragmaDev Tester* helps testers to write validation and integration tests with an abstract dedicated language. A substantial number of test cases with this technology are published by international standardization bodies to ensure conformance to their specifications.
- *PragmaDev Tracer* is common to all other tools. It is used in the early phase to describe the expected behavior and properties of the system to be developed. It is used in the later phase to trace execution and verify it is conform to the expected properties and scenarios described earlier on.

The complete tool set *PragmaDev Studio* includes bridges from one tool to the other such as automatic test case generation out of a functional model (model based testing), a Performance simulator to find the best architecture, and a Network simulator to analyze deployment of distributed systems.

1.2 - Licensing

A Freemium version of PragmaDev Studio is freely available. It includes all modules with a project and a file size restriction except external text files and MSC traces. So that means PragmaDev Tracer is free and comes with the Freemium version of PragmaDev Studio.

The other modules Specifier, Developer, and Tester are available independently are all together with a Studio license.

Please note that if you have one of these licenses, you should start the corresponding tool. For example if you have a Specifier license, to use your unrestricted license, you should start the Specifier tool (not Studio).

PragmaDev modules are based on a floating license mechanism that can be shared over your local network. One license is one user at a time. And the same license can be used over Windows, Linux, Mac, and Solaris.

You can either:

- Rent a license for a monthly fee that includes maintenance (support and upgrades).
- Buy a license so that you own it for an unlimited amount of time, in this case upgrades and support are part of an optional annual maintenance fee.

Please find below the detailed features for each module or for the whole Studio.

Table 1: Tool feature matrix

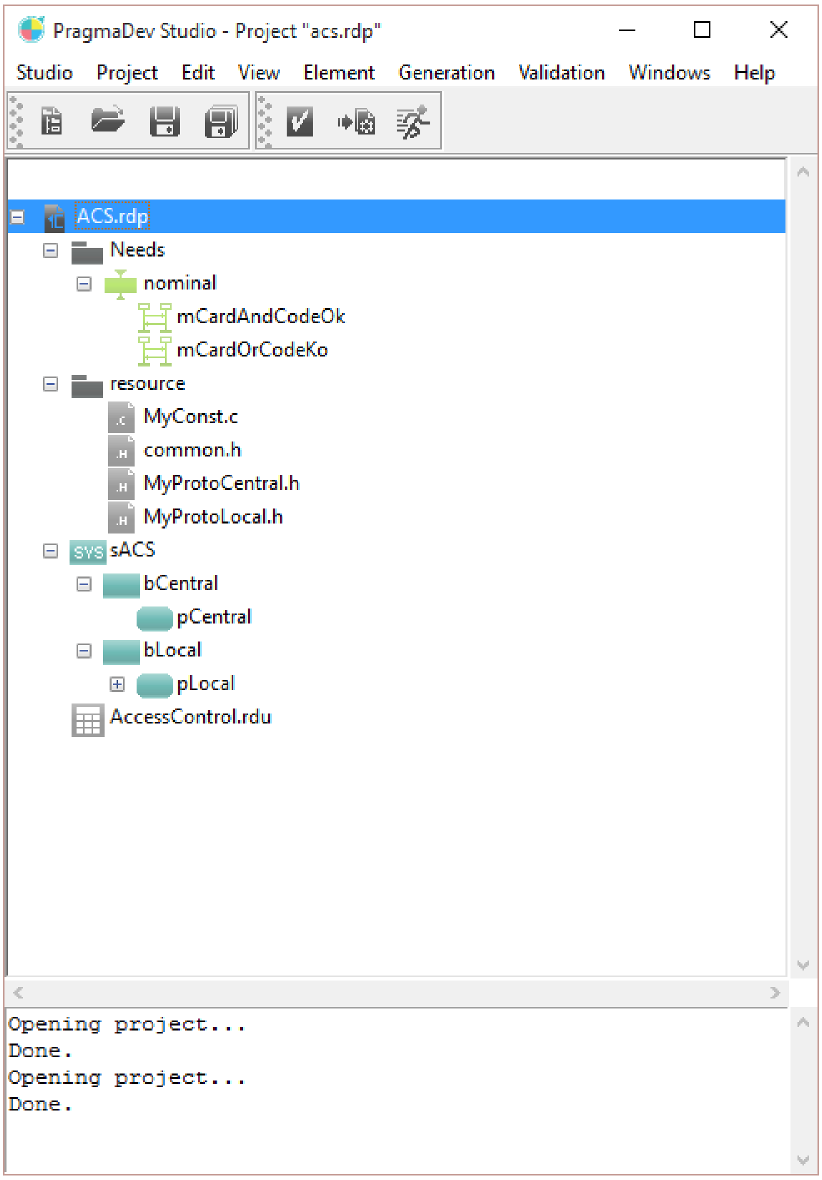
Tool Feature	PragmaDev Specifier	PragmaDev Developer	PragmaDev Tester	PragmaDev Studio	PragmaDev Tracer
MSC-PSC	X	X	X	X	X
HMSC - Use Case	X	X	X	X	
Prototyping GUI	X	X	X	X	
Documentation generator	X	X	X	X	
Diff - Merge	X	X	X	X	
ASN.1	X	X	X	X	
Traceability	X	X	X	X	
SDL model editors	X			X	
Model Simulator	X			X	
SDL-RT Model Editor		X		X	
C code generation		X		X	
MSC2TTCN			X	X	
TTCN2MSC			X	X	
Test Simulator			X	X	
Test C code generator			X	X	
Deployment simulator				X	
Performance analyzer				X	
PR/FIACRE/IF/xLIA export				X	
SDL code generator				X	
Model validation with OBP				X	

2 - Common features

This chapter includes features common to all modules within PragmaDev Studio. Specific features are described in the module's chapter.

2.1 - 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.



```
Opening project...
Done.
Opening project...
Done.
```

Project manager window

2.1.1 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 11.

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 10). 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.

2.1.2 Files and directories

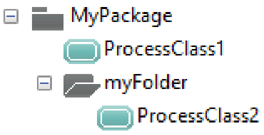
The files associated to nodes in a project tree are stored as relative paths from the directory containing the project file. This mode allows the project to be on a shared disk so that it can be used by several users on different machines. This mode is also suited for sharing a project across different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

2.1.3 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:

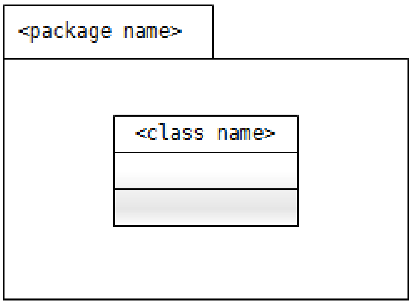


`ProcessClass1` and `ProcessClass2` are both in the same package: `MyPackage`. The folder `MyFolder` is just used for grouping and does not imply any organisation on the elements it contains. So any diagram including a "`USE MyPackage;`" in its declarations will be able to instantiate both `ProcessClass1`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

2.1.4 Supported file types

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

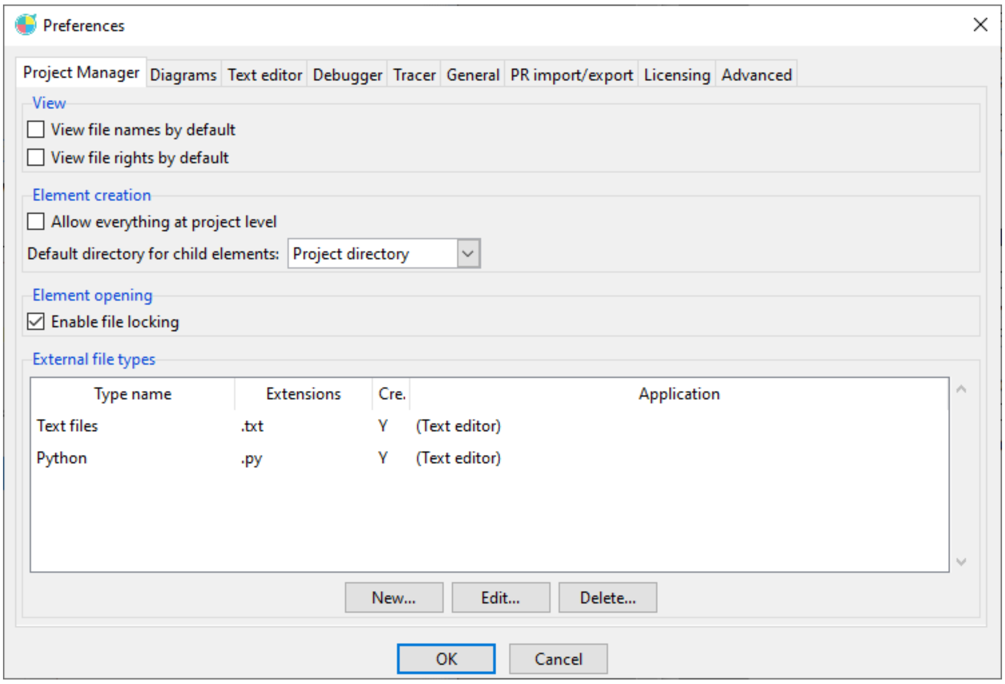
- Diagrams:
  - SDL system,
  - SDL block,
  - SDL block class,
  - SDL process,
  - SDL process class,
  - SDL procedure,
  - SDL composite state (cf. notes 1. & 2. below),
  - SDL service (cf. note 1. below),
  - SDL macro (cf. note 1. below),
  - MSC/PSC,
  - High-level MSC,
  - UML use case diagram,
  - UML class diagram,
  - UML deployment diagram,
  - IF observer diagrams.
- SDL declarations files. The standard extensions for these files is `.pr`.
- 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 (`.ttcn3`).
- 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).
- OTF (Open Trace Format) documents can be added to the project and viewed with the MSC editor.

Notes:

1. These diagram types are only available in SDL language.
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 "Project Manager" tab in the preferences dialog, opened via the "Preferences..." item in the "Studio" menu:



View

□ View file names by default

□ View file rights by default

Element creation

□ Allow everything at project level

Default directory for child elements: Project directory

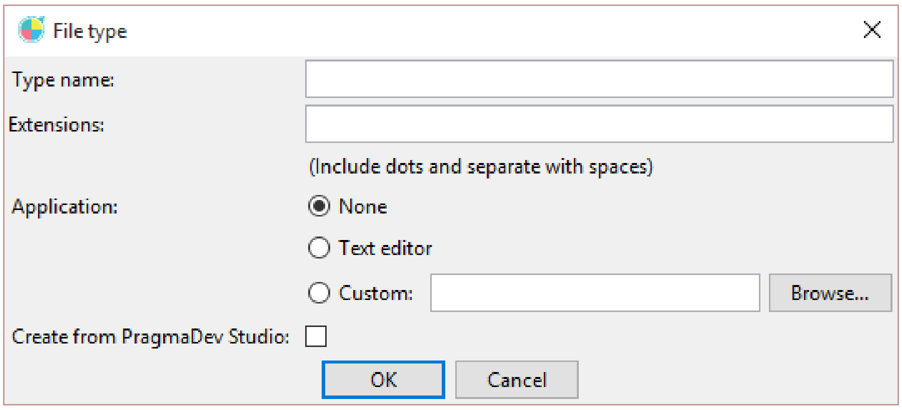
Element opening

☑ Enable file locking

External file types

Type name	Extensions	Cre.	Application
Text files	.txt	Y	(Text editor)
Python	.py	Y	(Text editor)

Creating a new custom file type opens the following window:



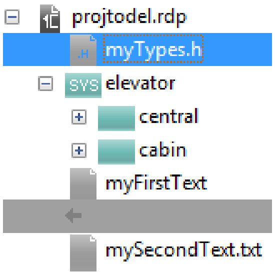
Notes:

- If the "Application" field is "None", files of this type may be included in the project, but opening them will rely on the facilities provided by the operating system. So you will have to associate an application to this file type using the services of your OS.
- Setting the application to "Text editor" allows to do text searches and replacements in the file. Otherwise, when doing a text search or 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.
- If a file type for PDF files exists (extension `.pdf` or `.PDF`) and it specifies an application, this will be the application used to view PragmaDev Studio manuals. If no such file type exists, or it has its application set to `None`, the services of the underlying OS will be used to open the PDF files.
- 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.

2.1.5 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 gree arrow indicates where the node will be pasted. In the above example, the node will be moved or pasted betwwen 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.

2.1.6 Adding components to the system

There are three ways of adding components to your project:

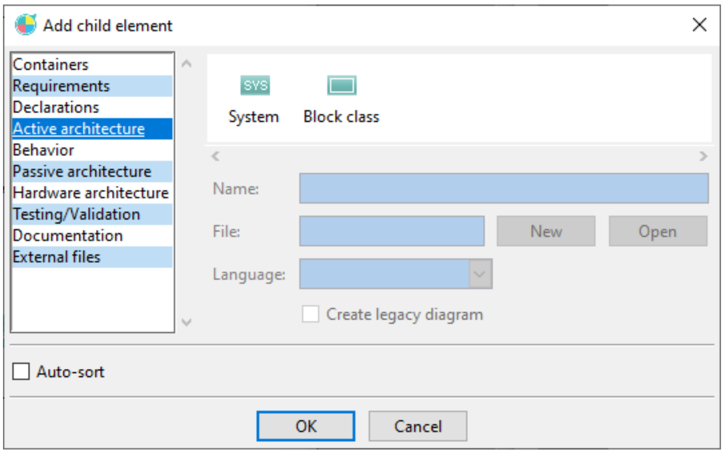
- You can directly add components one by one via the project manager in the project tree. This operation is described in “Adding a single component” on page 14.
- You can import a whole directory with all its sub-directories in the project tree. This operation is described in “Importing a directory” on page 16.
- 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 74. This way is used to create the part of the project tree that is mapped to the block/process hierarchy.

2.1.6.1 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 catogeries of element that can be added in a project.Once a category has been choosen, available node types are listed on the upper right of the window.

The other fields in this dialog are the following:

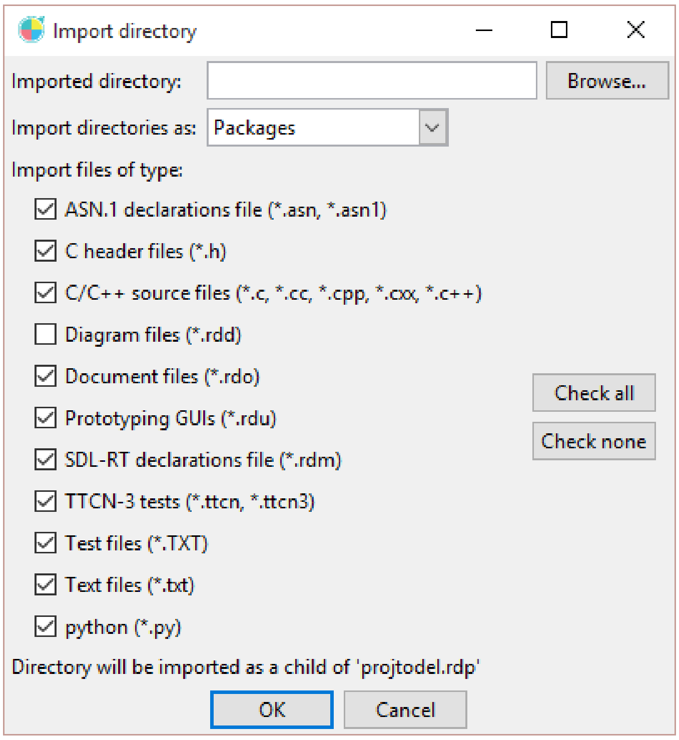
- *Name*: the name appearing for the node in the project tree. If not set, this field will be automatically set if you choose a file in the "*File*" field.
- *File*: the name of the file associated to the node. This field cannot be set if the node's type is "Package" or “Folder”, since these have no associated file. Otherwise, the field will be automatically set to a computed default file name, and can also be modified via the "*New*" or "*Open*" buttons, to create a new file and to select an existing one, respectively.
- *Language*: this field is only available for diagrams that have different variants depending on the language used, and if their parent do not already specify a language for them. So this mostly includes system, block class and process class diagrams, that can be written either in SDL, or in SDL-RT. When the filed is not greyed out, this information is mandatory: the dialog cannot be validated unless a language is specified.
- *Create legacy diagram*: this checkbox is only available for diagrams having a “new-style” and a “legacy” variant. Checking it will create a legacy diagram instead of the default, which is to create a new-style diagram. This applies to MSCs, and to all behavioral diagrams: process, process class, procedure, service and macro. Please note that the usage of legacy diagrams is deprecated, and that a lot of editing facilities will be missing in legacy diagrams. This option should only be used to include in the project diagrams made with an older version of PragmaDev Studio.
- *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.

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.

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



Import directory

Imported directory:

Import directories as: Packages

Import files of type:

☑ ASN.1 declarations file (*.asn, *.asn1)
☑ C header files (*.h)
☑ C/C++ source files (*.c, *.cc, *.cpp, *.cxx, *.c++)
□ Diagram files (*.rdd)
☑ Document files (*.rdo)
☑ Prototyping GUIs (*.rdu)
☑ SDL-RT declarations file (*.rdm)
☑ TTCN-3 tests (*.ttcn, *.ttcn3)
☑ Test files (*.TXT)
☑ Text files (*.txt)
☑ python (*.py)

Directory will be imported as a child of 'projtodel.rdp'

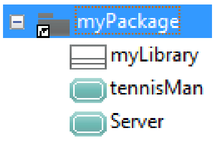
This window allows to select the directory to import and the type of the files that will actually be imported in the project tree. The file types that appear are standard PragmaDev Studio types (C source and header files, diagram files) and all the external file types (see “Supported file types” on page 11). 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.

2.1.7 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.

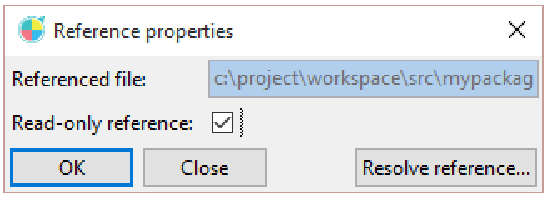
Any sub-tree in a project can be shared. To export a sub-tree, select its root node and select "Export element sub-tree" in the "Element" menu. The sub-tree is exported to a file with a `rdx` extension. Once exported, the icon for the sub-tree root will be displayed in the project manager as follows:



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:



Referenced file: c:\project\workspace\src\mypackag

Read-only reference: ☑

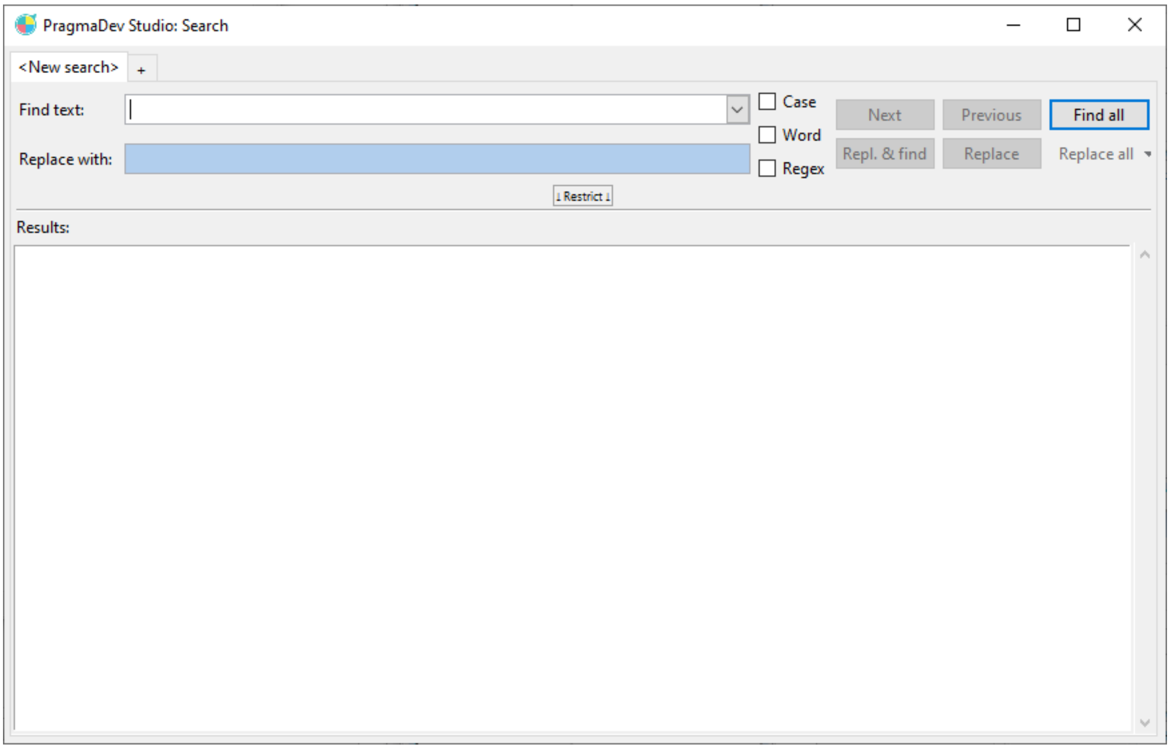
NB:

- 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.

2.1.8 Search and replace

2.1.8.1 Searching the whole project

Searching for some text in the whole project is available by selecting the “Find all...” entry in the “Edit” menu of the project manager. The find all window appears:

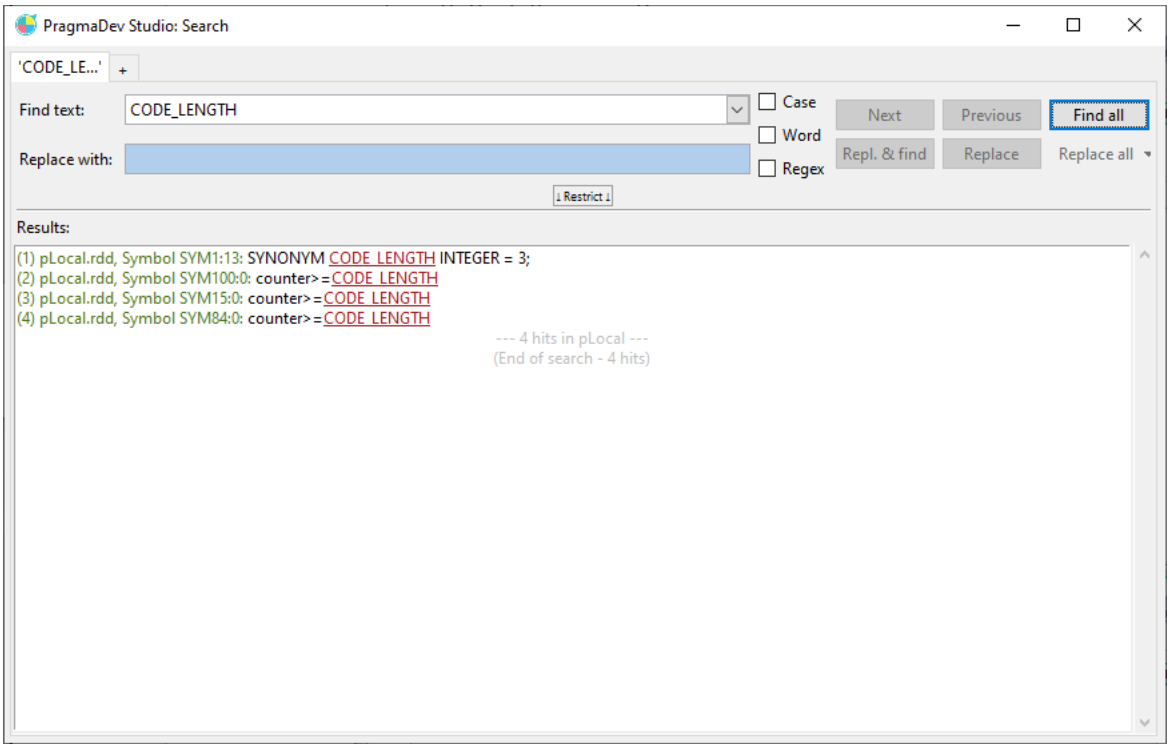


The window is organized in tabs, allowing you to keep the results of several searches. To create a new tab, just click the “+” at the end of the tab bar. Tabs cannot be closed; they will all be forgotten when you close the window as a whole.

The text to find is entered in the “Find text” field. The options are:

- “Case” makes the search case-sensitive:
- “Word” searches only for whole words;
- “Regex” searches for regular expressions instead of plain text.

Once the text to find is specified, clicking the “Find all button” will display all the matches in the lower part of the window:



Find text: CODE_LENGTH

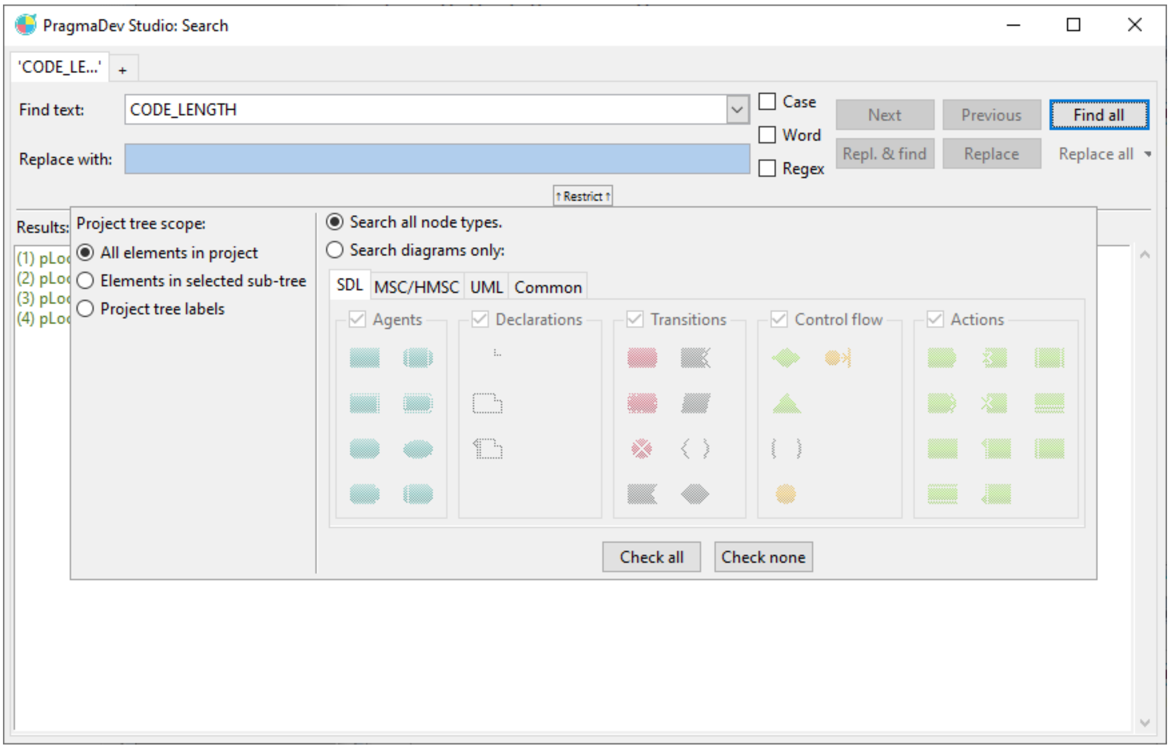
Replace with:

□ Case
□ Word
□ Regex

```
(1) pLocal.rdd, Symbol SYM1:13: SYNONYM CODE_LENGTH INTEGER = 3;
(2) pLocal.rdd, Symbol SYM100:0: counter>=CODE_LENGTH
(3) pLocal.rdd, Symbol SYM15:0: counter>=CODE_LENGTH
(4) pLocal.rdd, Symbol SYM84:0: counter>=CODE_LENGTH
--- 4 hits in pLocal ---
(End of search - 4 hits)
```

2.1.8.1.1 Restrictions

The search can also be restricted to only part of the project by clicking the “Restrict” button, which will display the restriction panel:



Find text: CODE_LENGTH

Replace with:

□ Case
□ Word
□ Regex

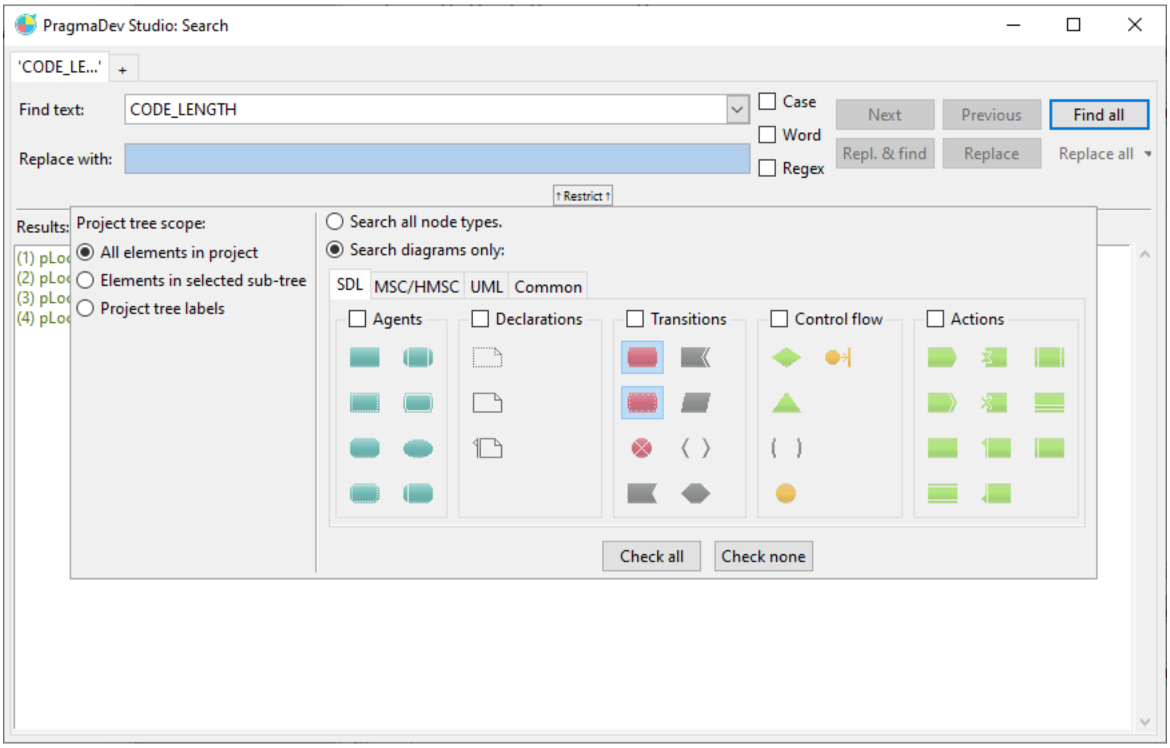
Project tree scope:
● All elements in project
○ Elements in selected sub-tree
○ Project tree labels

● Search all node types.
○ Search diagrams only:

☑ Agents
☑ Declarations
☑ Transitions
☑ Control flow
☑ Actions

- If “All elements in project” is checked on the left side of the panel, all elements in the projects are searched, considering the filters set on the right side of the panel;
- If “Elements in selected sub-tree” is checked on the left side of the panel, on the currently selected node and all its descendants will be searched, with the additional filters configured on the right side of the panel;
- If “Project tree labels” are checked on the left side of the panel, the text is searched in the element names in project tree itself, and not in the element contents. The filters on the right side of the panel are then not available and will be greyed out.

The filters on the right side of the panel allow to restrict the search to diagrams wih a given type, and also to symbols with a type within a given set within these diagrams. For example, with the following setting:



PragmaDev Studio: Search

'CODE_LE...' +

Find text: CODE_LENGTH

Replace with:

□ Case
□ Word
□ Regex

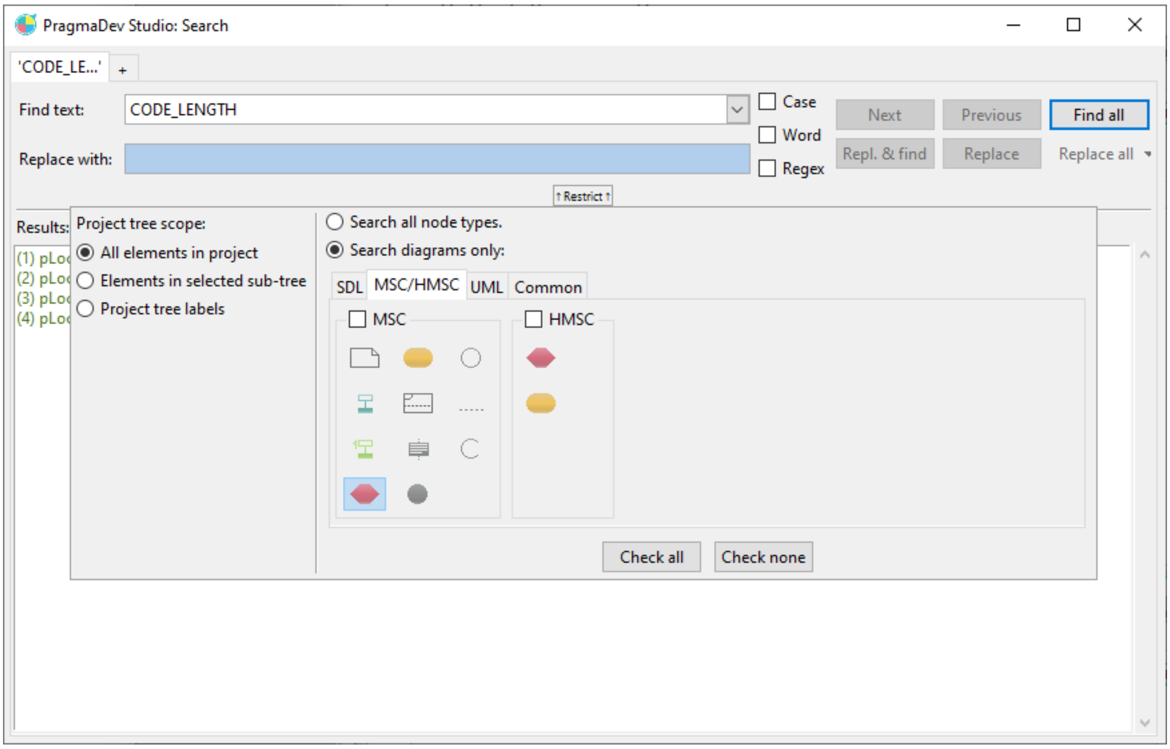
Results:

Project tree scope:
● All elements in project
○ Elements in selected sub-tree
○ Project tree labels

○ Search all node types.
● Search diagrams only:

SDL | MSC/HMSC | UML | Common

□ Agents □ Declarations □ Transitions □ Control flow □ Actions



PragmaDev Studio: Search

'CODE_LE...' +

Find text: CODE_LENGTH

Replace with:

□ Case
□ Word
□ Regex

Results:

Project tree scope:
● All elements in project
○ Elements in selected sub-tree
○ Project tree labels

○ Search all node types.
● Search diagrams only:

SDL | MSC/HMSC | UML | Common

□ MSC □ HMSC

and nothing else selected, the text would be searched only in state and composite state symbols in SDL diagrams, and in condition symbols in MSC diagrams.

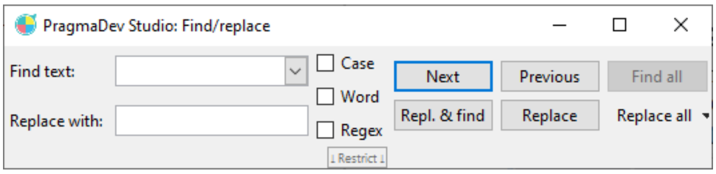
2.1.8.1.2 Regular expressions

Regular expressions are a language allowing to express more complicated searches than just text. Most characters in a regular expression match themselves, except the following ones, that have a special meaning:

- "." matches any character in the text.
- "*" matches 0 or more repetition of the part just before it in the regular expression. So for example, "a*" will match "a", "aa", "aaaaaaa", and also "".
- "+" matches 1 or more repetitions of the part just before it in the regular expression. So for example, "a+" will match "a", "aa", "aaaaaaa", but not "".
- "?" makes the part before it in the regular expression optional. So "a?" will match "a" or the empty string, and nothing else.
- A set of characters can be specified between "[" and "]" to match any of the characters in the set. So for example, "[abc]" will match "a", "b", "c", but not "d".
  A range can be specified using a "-": "[a-z]" will match any lowercase letter, and "[0-9]" will match any digit.
  If a set of characters must include the characters "-" or "]", they can be "escaped" with a "\". So "[(){}[\]\-]" will match the characters "(", ")", "{", "}", "[", "]" and "-".
  A set can also be inverted by starting it with the character "^". So "[^a-z]" will match any character, except a lowercase letter.
- Regular expressions can be grouped by enclosing them between "(" and ")". So for example, "(ab)+" will match 1 or more repetition of the group "ab", so it will match "ab", "abab", "ababab", but not "aaba".
- The "\" character can be used to reference special groups of characters:
  - "\s" matches any space character (space, tab, ...);
  - "\S" matches any character, except a space character;
  - "\b" matches the empty string but only at the beginning or end of a word;
  - "\B"matches the empty string but only when *not* at the beginning or end of a word;
  - "\d" matches any digit;
  - "\D" matches anything that is not a digit;
  - "\w" matches any letter or an underscore;
  - "\W" matches anything but letters and underscores.

2.1.8.2 Replacing text in the whole project

Replacing text in the whole project is done by selecting the "Find / replace" entry in the "Edit" menu of the project manager. It displays the following window:

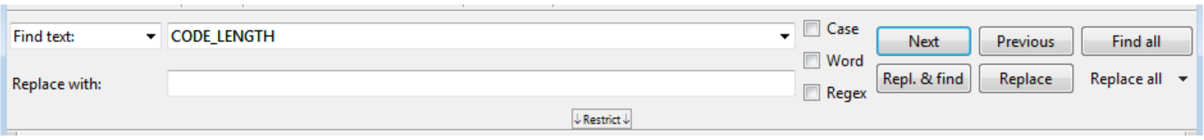


The "Next" and "Previous" buttons allow to navigate to the next and previous match respectively. "Replace" will replace the currently selected occurence; "Repl. & find" will replace the current occurence and search for the next one. The “Replace all” button displays a menu where only the entry “everywhere” is available, allowing to do the replacement everywhere in the project.

When searching for a regular expression, it is also possible to include in the replacement string references to groups in the searched expression. These references are a “\”, followed by the order number of the group in the expression. So for example, searching for “`(ab)(cd)`” and replacing it with “`\2\1`” will replace all occurences of “`abcd`” with “`cdab`”.

2.1.8.3 Searching and replacing in the editors

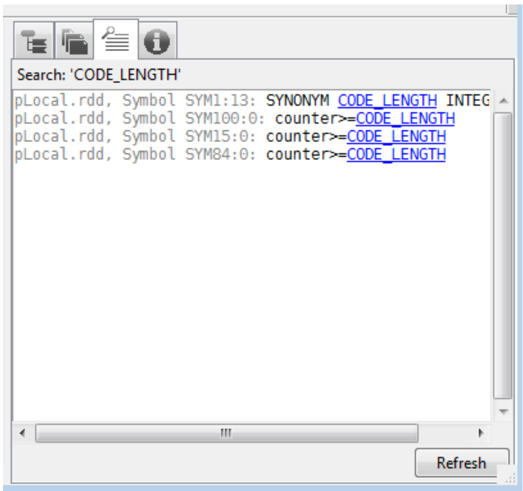
Finding and replacing text is also available in all editors by selecting the entry “Find / replace” in the “Search” menu. This will display in the window the find / replace component just under the quick buttons:



Find text: CODE_LENGTH
Replace with:
□ Case
□ Word
□ Regex

The “Next”, “Previous”, “Repl. & find” and “Replace” buttons are the same as in the find / replace dialog in the project, but are of course restricted to the current diagram or file.

The “Find all” button will display all matches in the panel on the right side of the window in the “search results” tab:



Search: 'CODE_LENGTH'

```
pLocal.rdd, Symbol SYM1:13: SYNONYM CODE_LENGTH INTEG
pLocal.rdd, Symbol SYM100:0: counter>=CODE_LENGTH
pLocal.rdd, Symbol SYM15:0: counter>=CODE_LENGTH
pLocal.rdd, Symbol SYM84:0: counter>=CODE_LENGTH
```

Clicking on a match will select the symbol or line where the match was found.

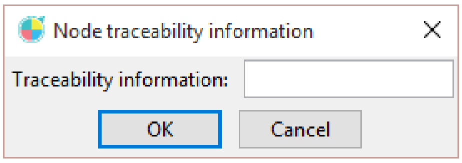
The “Replace all” menu may include the following entries:

- “in selection” will replace all matches within the current selection.
- “from here” will replace all matches from the current position until the end. This is only meaningful in text files.
- “everywhere” will replace all matches from the beginning to the end.

2.1.9 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.

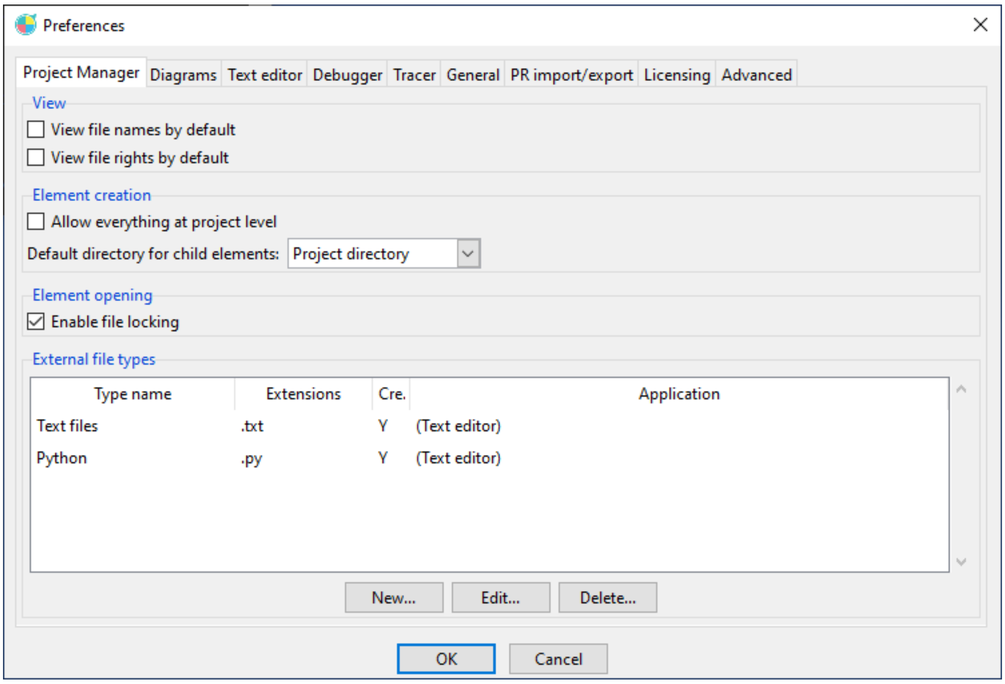
Note PragmaDev Studio doesn’t interpret the traceability information at all. It is just saved in the project XML file as an attribute for the node. The interpretation of the information is done by the requirement management tool.

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 44.

2.1.10 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.

2.1.10.1 Project manager preferences



View
☐ View file names by default
☐ View file rights by default

Element creation
☐ Allow everything at project level
Default directory for child elements: Project directory

Element opening
☑ Enable file locking

External file types

Type name	Extensions	Cre.	Application
Text files	.txt	Y	(Text editor)
Python	.py	Y	(Text editor)

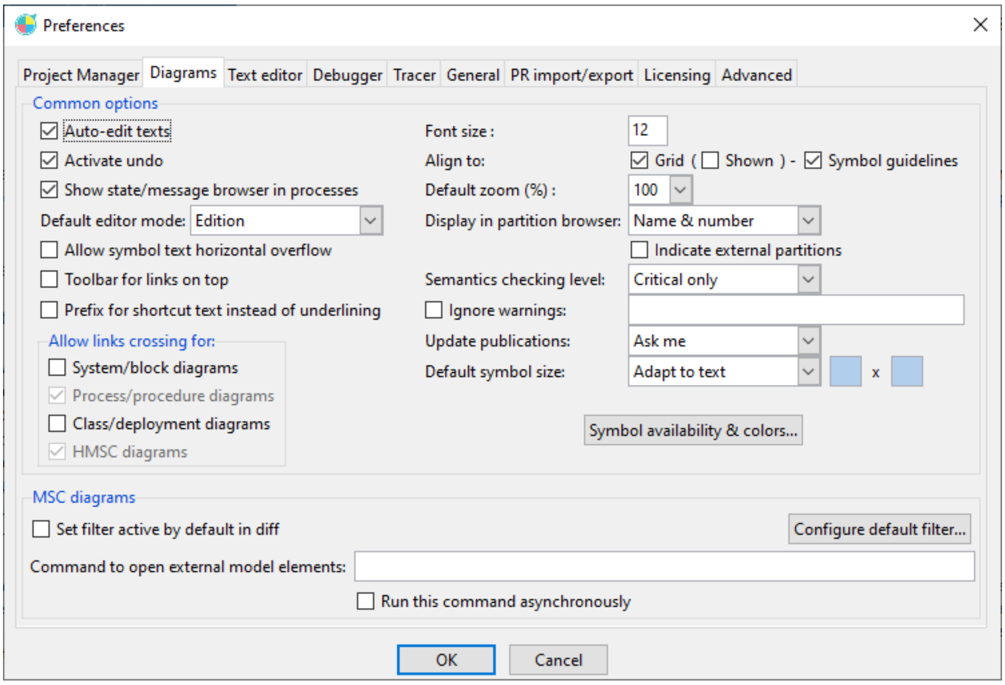
The "*View file names by default*" and "*View file rights by default*" options control whether the file names and/or access rights will be automatically displayed in the project manager window when a project is opened.

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 or agent classes 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.

The “*External file types*” have been described in “Supported file types” on page 11.

2.1.10.2 Diagram preferences



Common options

☑ Auto-edit texts

☑ Activate undo

☑ Show state/message browser in processes

Default editor mode: Edition

☐ Allow symbol text horizontal overflow

☐ Toolbar for links on top

☐ Prefix for shortcut text instead of underlining

Allow links crossing for:

☐ System/block diagrams

☑ Process/procedure diagrams

☐ Class/deployment diagrams

☑ HMSC diagrams

Font size : 12

Align to: ☑ Grid (☐ Shown) - ☑ Symbol guidelines

Default zoom (%) : 100

Display in partition browser: Name & number

☐ Indicate external partitions

Semantics checking level: Critical only

☐ Ignore warnings:

Update publications: Ask me

Default symbol size: Adapt to text x

MSC diagrams

☐ Set filter active by default in diff

Command to open external model elements:

☐ Run this command asynchronously

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 78 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 clicking on the symbol edits its text; in navigation mode, the button allows to edit the symbol’s text, clicking on a symbol just selects it, and double-clicking on it 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. Note that this option has no effect in new-style behavioral diagrams, where symbols are always adjusted to their text.

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 “*Align to*” options control what guides will be available in diagrams for symbol positionning:

- If “*Grid*” is checked, the symbol center position will be automatically aligned on a grid. This grid can be displayed by checking the “*Shown*” checkbox appearing after the “*Grid*” option.
- If “*Symbol guidelines*” is checked, lines will appear as you move the symbols around, allowing to align them to existing symbols. This allows to align symbol left sides, top sides, or centers. This feature is only available for new-style behavioral diagrams. Also note that enabling the symbol guidelines will disable the grid, as symbols cannot be aligned on both the grid and the other symbols.

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:

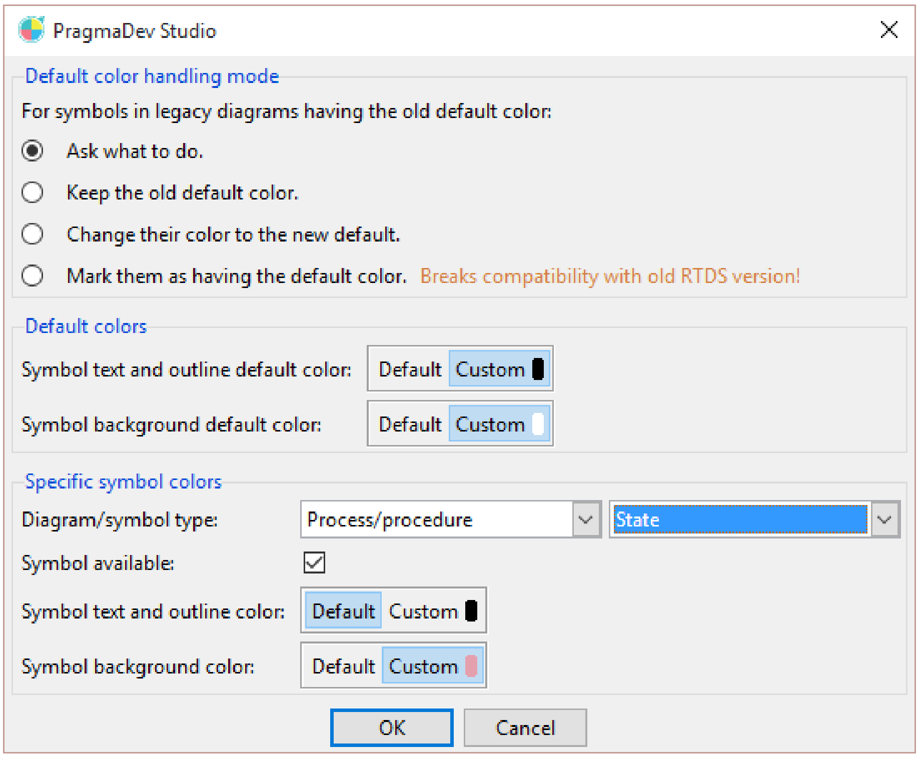
- "All" to report errors for any problem
- "Critical only" to report errors only for problems preventing the system to work and only warnings for other problems

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 68.

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). Note that this option does not apply to new-style diagrams (MSCs or behavioral).

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:



PragmaDev Studio

Default color handling mode

For symbols in legacy diagrams having the old default color:

● Ask what to do.

○ Keep the old default color.

○ Change their color to the new default.

○ Mark them as having the default color. Breaks compatibility with old RTDS version!

Default colors

Symbol text and outline default color: Default | Custom

Symbol background default color: Default | Custom

Specific symbol colors

Diagram/symbol type: Process/procedure | State

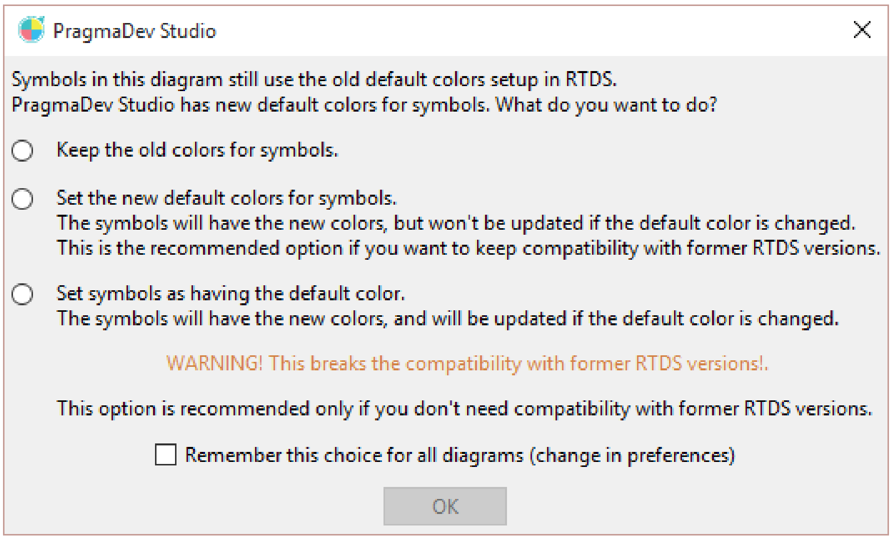
Symbol available: ☑

Symbol text and outline color: Default | Custom

Symbol background color: Default | Custom

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:



PragmaDev Studio

Symbols in this diagram still use the old default colors setup in RTDS.
PragmaDev Studio has new default colors for symbols. What do you want to do?

○ Keep the old colors for symbols.

○ Set the new default colors for symbols.
The symbols will have the new colors, but won't be updated if the default color is changed.
This is the recommended option if you want to keep compatibility with former RTDS versions.

○ Set symbols as having the default color.
The symbols will have the new colors, and will be updated if the default color is changed.

WARNING! This breaks the compatibility with former RTDS versions!.

This option is recommended only if you don't need compatibility with former RTDS versions.

☐ Remember this choice for all diagrams (change in preferences)

- *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 breaks the compatibility with former Real Time Developper Studio versions.

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.

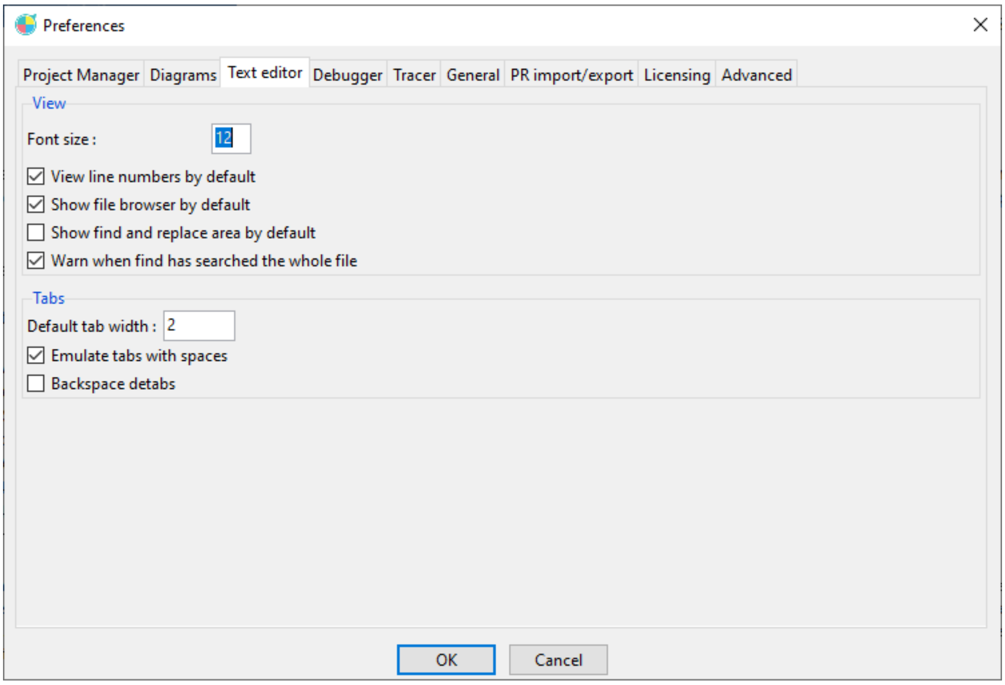
Note that making a symbol type unavailable only prevents it from appearing in the toolbars, it doesn't make it invalid: if a diagram containing such a symbol is opened, it will still be displayed correctly, and all operations will work on the symbol.

Also note that the symbol and link default colors are dynamic: changing the default color in this dialog will change all symbols or links marked as having the default color. Setting a specific color for a symbol or link, or setting it back to use the default is done via the symbol and link properties panels in the diagram editor (see “Symbol and link properties” on page 62).

The options in the “*MSC diagrams*” group apply only to MSC diagrams:

- Checking “*Set filter active by default in diff*” will automatically select the default filter options when comparing MSC diagrams. The default filter can be configured via the “*Configure default filter...*” button. A detailed description of the filter options can be found in paragraph “Conformance checking: diagram diff & property match” on page 97.
- The “Command to open external model elements” options allows to specify the executable to launch to open model elements when a model element identifier is specified for an item in a MSC diagram. If the “Run this command asynchronously” option is checked, this command will be run without waiting for it to complete before returning the control to PragmaDev Studio. Note that this prevents any error reporting if the command fails for any reason.
  Note that this command will only be used for model elements that are not recognized by PragmaDev Studio. For example, model element identifiers automatically set in traces produced during a debug session will be directly handled by PragmaDev Studio, and this command will not be used.
  For more information about model element identifiers in MSC diagrams, see “Linking with model elements” on page 371.

2.1.10.3 Text editor preferences



View

Font size : 12

☑ View line numbers by default

☑ Show file browser by default

□ Show find and replace area by default

☑ Warn when find has searched the whole file

Tabs

Default tab width : 2

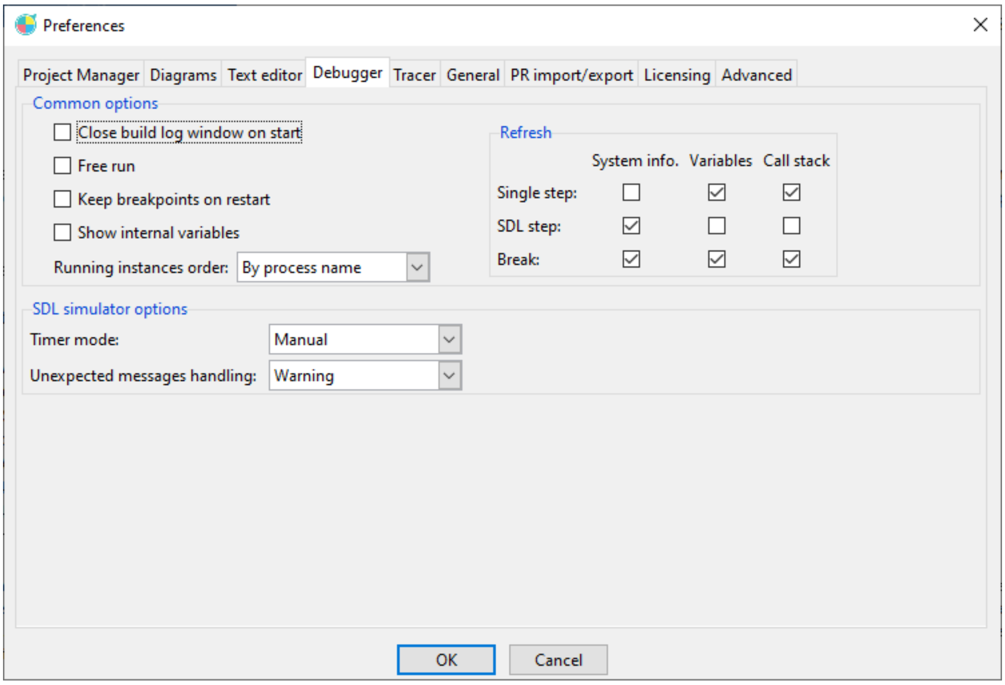
☑ Emulate tabs with spaces

□ Backspace detabs

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.

2.1.10.4 Debugger preferences



Common options

□ Close build log window on start

□ Free run

□ Keep breakpoints on restart

□ Show internal variables

Running instances order: By process name

Refresh

	System info.	Variables	Call stack
Single step:	□	☑	☑
SDL step:	☑	□	□
Break:	☑	☑	☑

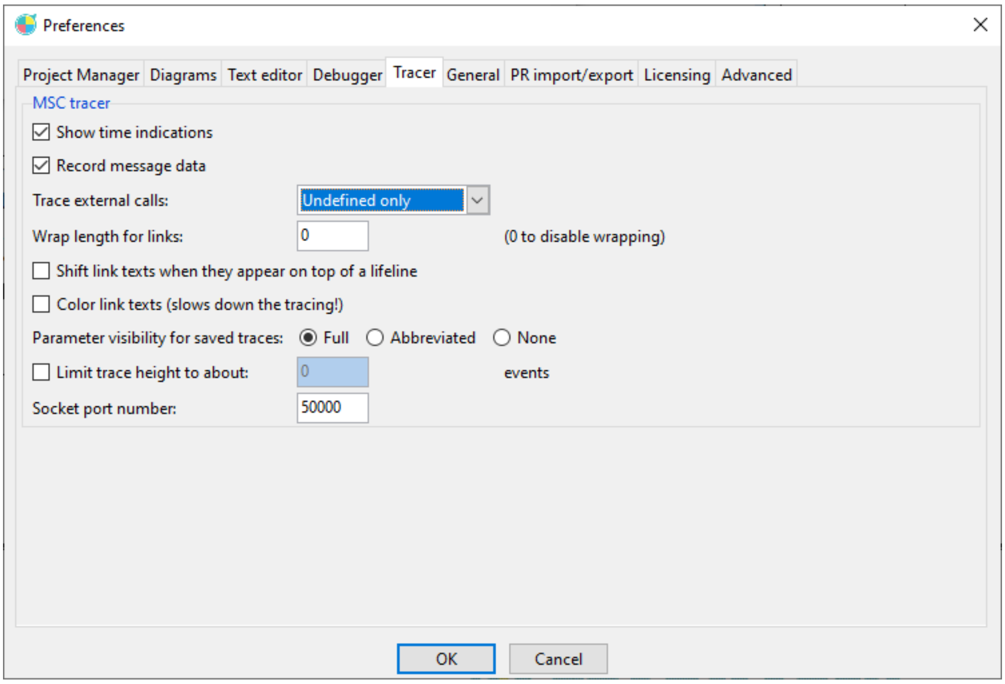
SDL simulator options

Timer mode: Manual

Unexpected messages handling: Warning

The options in the “*Common options*” group are the default values for those found in the “*Options*” menu in the debugger and simulator windows. They’re described in chapters “Model Debugger” on page 311 and “Model Simulator” on page 185. The latter also includes the description of the options in the “*SDL simulator options*” group.

2.1.10.5 Tracer preferences



MSC tracer

☑ Show time indications

☑ Record message data

Trace external calls: Undefined only

Wrap length for links: 0 (0 to disable wrapping)

□ Shift link texts when they appear on top of a lifeline

□ Color link texts (slows down the tracing!)

Parameter visibility for saved traces: ● Full ○ Abbreviated ○ None

□ Limit trace height to about: 0 events

Socket port number: 50000

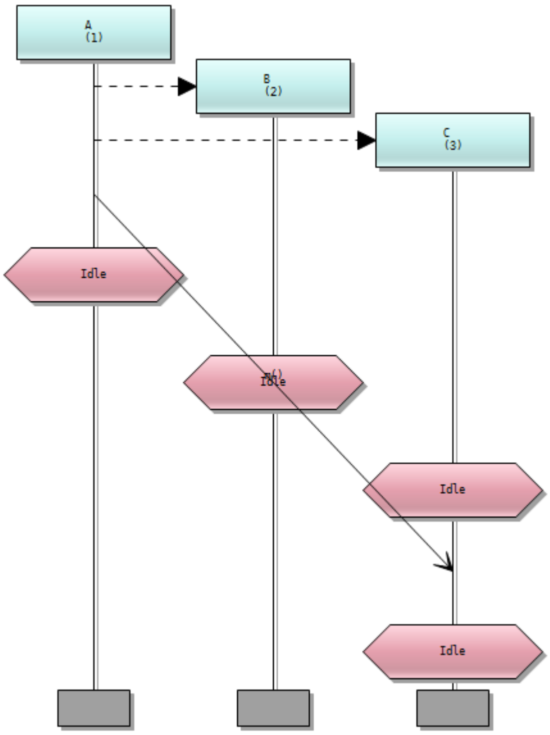
The “*Show time indications*” option controls whether the absolute times will be recorded in MSC traces.

If the “*Record message data*” option is checked, the tracer will record only the message name for each message, not its parameters.

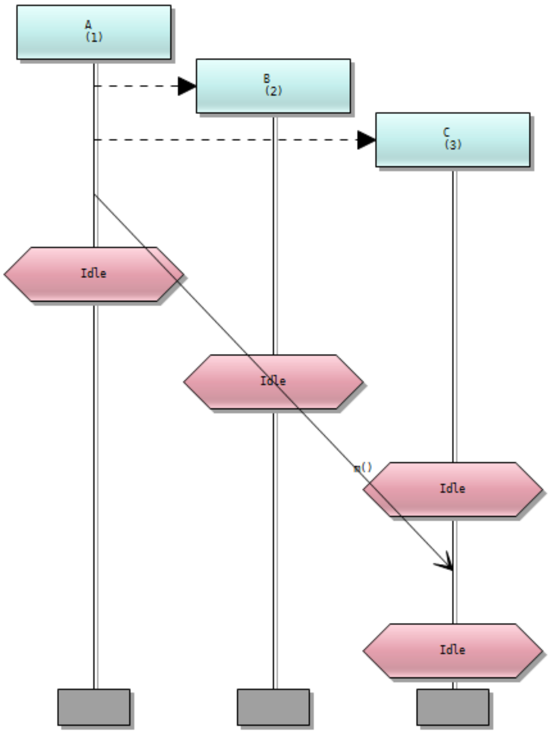
The “*Trace external calls*” option allows to capture synchronous calls from SDL operators or external procedures, to the environment. By default only the undefined ones are traced. There is a possibility to trace all of them or none of them. The point is be able to replay a scenario and for that matter to simulate the synchronous behavior of the environment.

If the “*Wrap length for links*” value is not 0, link texts will wrap when they are above this number of characters. This allows to still be able to read link texts when they are very long.

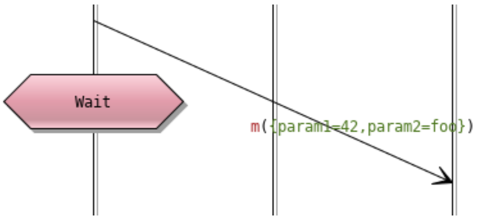
The “*Shift link text when they appear on top of a lifeline*” checkbox controls the placement of link texts when they might be confused with a text on a lifeline. For example, if without this option checked, a link text would appear like this:



checking the option would make it appear like this:



The “*Color link texts*” option colors differently the message name and its parameters in the trace:

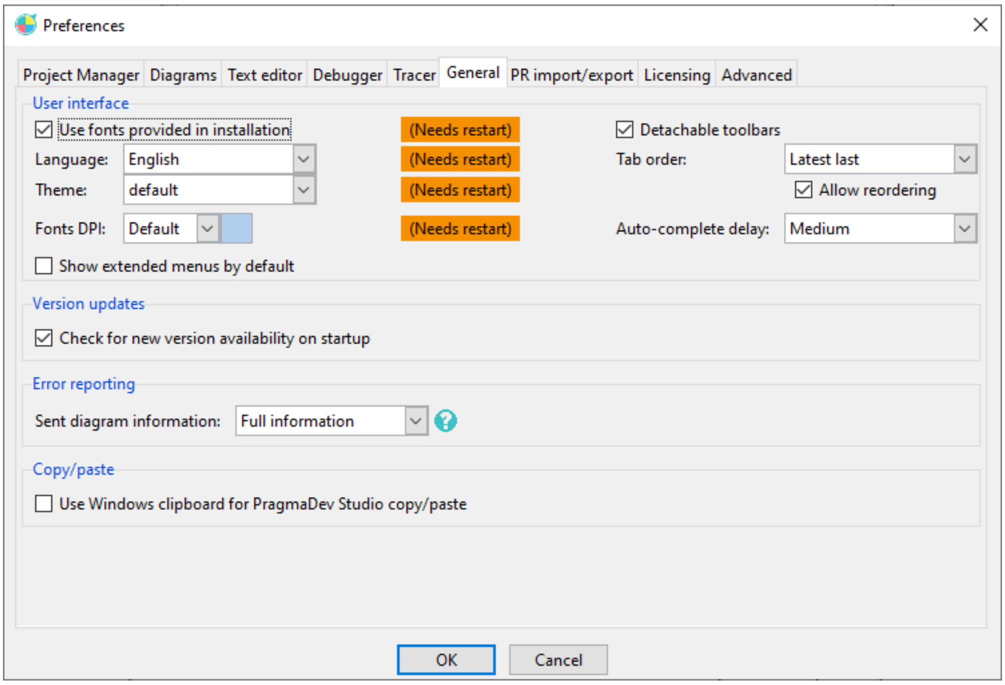


The option “*Parameter visibility for saved traces*” allows to set the visibility level for message parameters in the MSC diagrams saved from the tracer (see “Message parameters display” on page 94).

If the option “*Limit trace height to about*” is checked and a number of events is specified, the height for traces will be limited to this number of events. This allows to avoid performance issues when doing big traces. Please note that this only has an impact on the number of events displayed in the tracer window, and not on the number of events actually remembered in the trace. All events are still recorded and saving the trace will save them all.

The “*Socket port number*” option is for the socket port on which external clients can connect to generate a trace from an external program.

2.1.10.6 General preferences



User interface

☑ Use fonts provided in installation (Needs restart)

Language: English (Needs restart)

Theme: default (Needs restart)

Fonts DPI: Default (Needs restart)

□ Show extended menus by default

☑ Detachable toolbars

Tab order: Latest last

☑ Allow reordering

Auto-complete delay: Medium

Version updates

☑ Check for new version availability on startup

Error reporting

Sent diagram information: Full information

Copy/paste

□ Use Windows clipboard for PragmaDev Studio copy/paste

The “*Use fonts provided in installation*” option sets the font used 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 “*Language*” option controls the language used for PragmaDev Studio user interface; when this setting is modified, PragmaDev Studio will have to be restarted for the changes to appear.

The “*Theme*” option controls the GUI theme used for PragmaDev Studio. It will be automatically set to the native look on Windows and macOS.

Checking the option “*Show extended menus by default*” allow to always display full menus with all entries in all windows. The default is to show only menus that don’t have all their options already present as quick-buttons.

The option “*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 possible to reorder tabs in their parent window. Note that detaching tabs is always possible.

The “*Auto-complete delay*” option controls the delay between the last key press event and the display of the auto-complete list in diagam and text editors. It also allows to turn off the display. Note that even if the option is set to “*Off*”, the auto-complete list can always be displayed “manually” using the “F8” key.

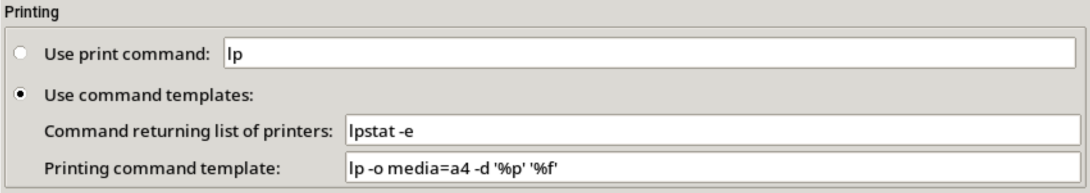
The “*Check for new version availability on startup*” checkbox allows to have PragmaDev Studio check if a newer version is available each time it is launched. If there is one, a dialog will appear offering to open a web browser on the downloads page on the PragmaDev website.

The “*Error reporting*” group allows to configure how much information is sent to PragmaDev support when an unexpected error occurs in PragmaDev Studio. The basic information that is always sent includes the traceback for the error, and a list of the last operations that were made during the editing session before the error occurred. By default, only the kind of operations and the internal identifiers for the symbols involved are sent, but the “Sent diagram information” option allows to send more than that, which can be very helpful to correct the condition that caused the error:

- “*No information*”: no information other than what is described above will be sent with the error. This means no diagram name or structure, and no texts for the symbols.
  This setting should be used when there are confidentiality issues with your diagrams, and you do not want anything about them to be transferred at all. Note that without any information on the diagram, correcting the error might be difficult in some cases.
- “*Anonymized information*”: the information transferred with the error will include a skeleton of the diagrams you were working on, giving its overall structure, but with no text for anything. The name of the diagram will not be transferred.
  This offers a trade-off between confidentiality and ease of correction by allowing to rebuild a diagram that mostly looks like the one you were working on, but that won’t include any actual text from yours.
- “*Full information*”: this will include all information available with the error, i.e the full diagrams you were working on as well as all texts for symbols and operations.
  Use this setting when you have no confidentiality issue on your diagrams. This is the setting that makes the correction of the problem the easiest.

The other options are platform-dependent:

- The “*Use Windows clipboard for PragmaDev Studio copy/paste*” option is only available on Windows. If it is checked, copying symbols from one PragmaDev Studio instance to another will be possible. Note however that it might have unexpected results if symbols are pasted in anything else than PragmaDev Studio.
- On Unix, a group of prefrences control how printing is done:



Printing

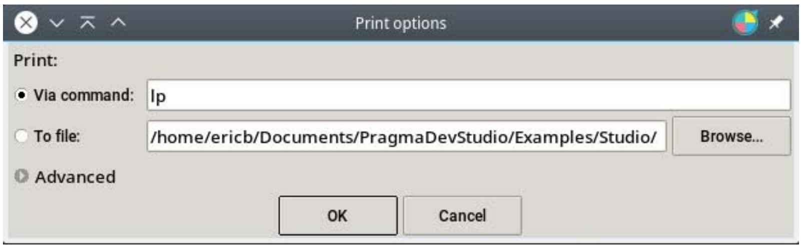
○ Use print command: lp

● Use command templates:

Command returning list of printers: lpstat -e

Printing command template: lp -o media=a4 -d '%p' '%f'

If the “*Use print command*” option is checked, printing will be done by running the given command with the file to print (in PostScript format). The typical value for this command is "`lp`", optionally with options to select the wanted printer. When this option is used, the print dialog will look like this:



Print:

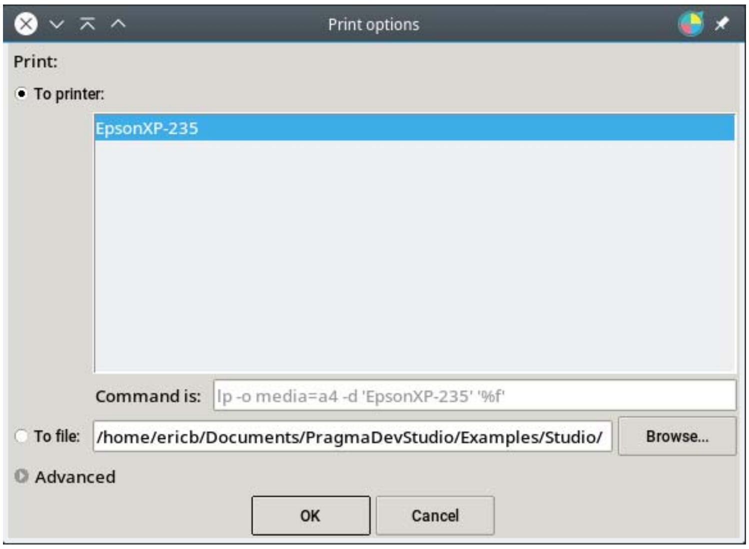
● Via command: lp

○ To file: /home/ericb/Documents/PragmaDevStudio/Examples/Studio/

If the "*Use command templates*" option is checked, the 2 following fields must be set:

- The "*Command returning list of printers*" must be set to a command sending to its standard output the names of all available printers. The typical value is "`lpstat -e`".
- The "*Printing command template*" must be set to a template for the printing command, where the string `%p` will be replaced by the printer name as printed by the "*command returning list of printers*", and `%f` will be replaced by the name of the file to print, again in PostScript format. The typical value is "`lp -d '%p' '%f'`". In the dialog above, an option has been added to specify the page size (A4).

When command templates are used, the print dialog will look like this:



Print:

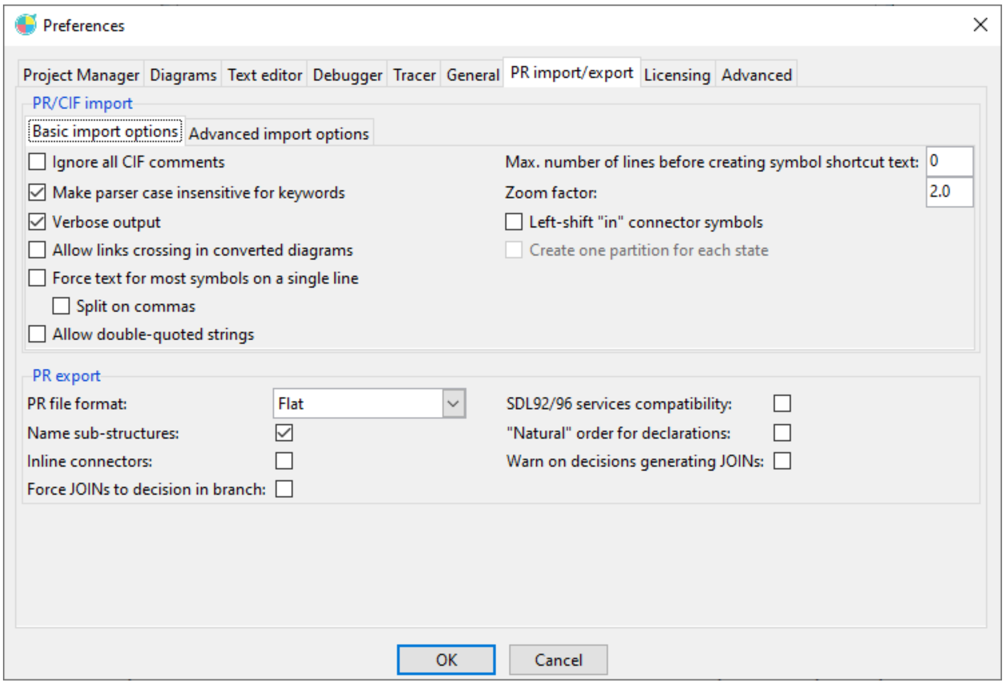
● To printer:

EpsonXP-235

Command is: lp -o media=a4 -d 'EpsonXP-235' '%f'

○ To file: /home/ericb/Documents/PragmaDevStudio/Examples/Studio/

2.1.10.7 PR import & export preferences



PR/CIF import

Basic import options | Advanced import options

- ☐ Ignore all CIF comments
- ☑ Make parser case insensitive for keywords
- ☑ Verbose output
- ☐ Allow links crossing in converted diagrams
- ☐ Force text for most symbols on a single line
  - ☐ Split on commas
- ☐ Allow double-quoted strings

Max. number of lines before creating symbol shortcut text: 0

Zoom factor: 2.0

- ☐ Left-shift "in" connector symbols
- ☐ Create one partition for each state

PR export

PR file format: Flat

Name sub-structures: ☑

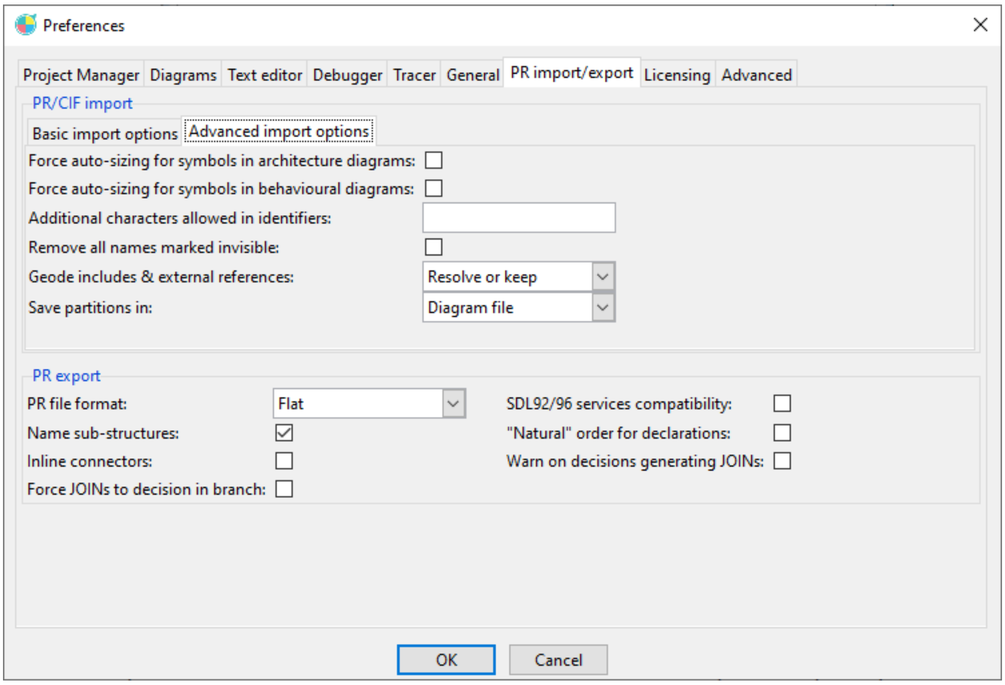
Inline connectors: ☐

Force JOINs to decision in branch: ☐

SDL92/96 services compatibility: ☐

"Natural" order for declarations: ☐

Warn on decisions generating JOINs: ☐



PR/CIF import

Basic import options | Advanced import options

Force auto-sizing for symbols in architecture diagrams: ☐

Force auto-sizing for symbols in behavioural diagrams: ☐

Additional characters allowed in identifiers:

Remove all names marked invisible: ☐

Geode includes & external references: Resolve or keep

Save partitions in: Diagram file

PR export

PR file format: Flat

Name sub-structures: ☑

Inline connectors: ☐

Force JOINs to decision in branch: ☐

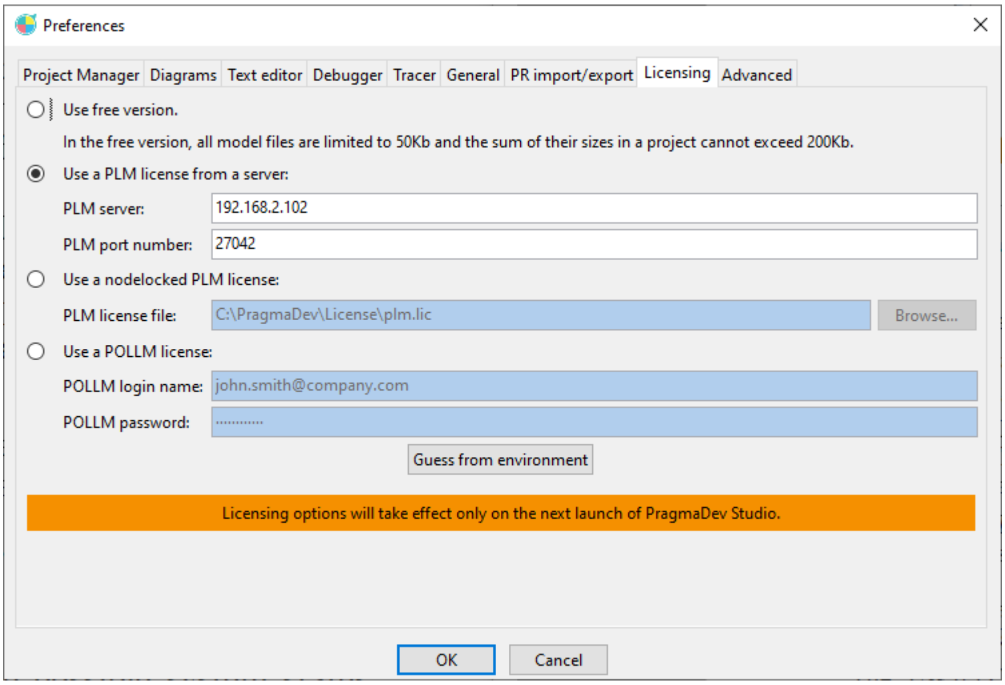
SDL92/96 services compatibility: ☐

"Natural" order for declarations: ☐

Warn on decisions generating JOINs: ☐

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 an MSC-PR file” on page 402 and “Exporting the project as an SDL/PR file” on page 464 respectively.

2.1.10.8 Licensing options



○ Use free version.

In the free version, all model files are limited to 50Kb and the sum of their sizes in a project cannot exceed 200Kb.

● Use a PLM license from a server:

PLM server: 192.168.2.102

PLM port number: 27042

○ Use a nodelocked PLM license:

PLM license file: C:\PragmaDev\License\plm.lic

○ Use a POLLM license:

POLLM login name: john.smith@company.com

POLLM password: ···········

Licensing options will take effect only on the next launch of PragmaDev Studio.

These options allow to specify which kind of license you will be using for PragmaDev Studio if you have one.

The “*Use free version*” option indicates that you will not be using any license. This allows you to use PragmaDev Studio freely on small projects.

The “*Use a PLM license from a server*” option indicates that you will be using a license managed by a PLM server. The hostname or IP address of the server and the socket port on which it listens are set via the “*PLM server*” and “*PLM port number*” options, respectively.

The “*Use a nodelocked PLM license*” option indicates that you will be using a PLM file dedicated to your computer.

The “*Use a POLLM license*” option indicates that you will be using a license managed by the PragmaDev OnLine License Manager. The login name and password specified in the dialog must be the same as the ones you use on the PragmaDev webiste for the license management interface.

The “*Guess from environment*” button allows to try to guess the settings from the environment variables set for your system. This is particularly useful if you were using an older version or PragmaDev Studio or Real Time Developer Studio, that relied on environment variables for the licensing information. Please note that this is no more supported: any change in the environment variables will have to be set back in these preferences by using this button again.

When PragmaDev Studio is launched for the first time, the licensing information will be asked interactively and stored in these preferences.

2.1.10.9 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.

2.1.11 User defined external tools

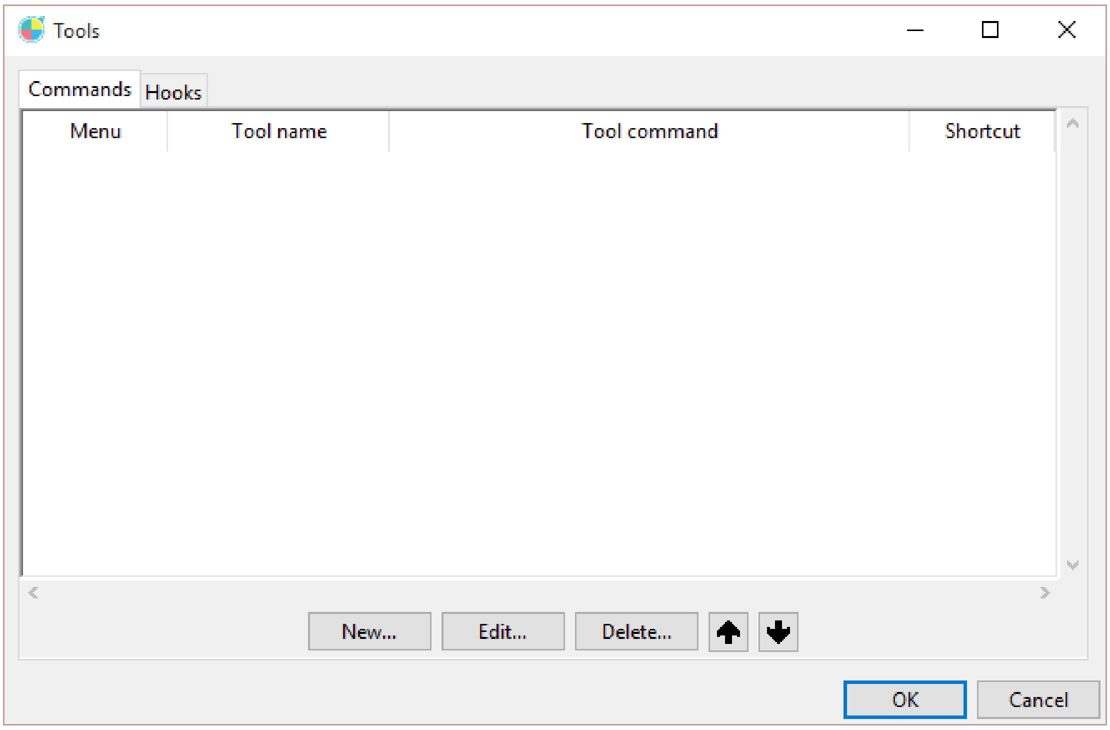
The project manager allows to define a set of external tools that may be used on the files in the project or on the project itself. A typical use of these tools is to interface Real Time Developer Studio with a configuration management system, but their use is much larger than that.

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:

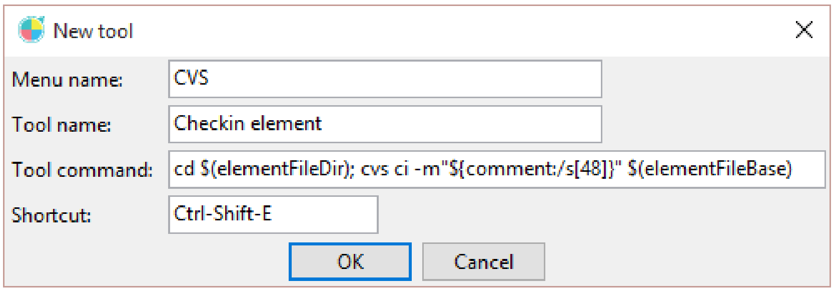
- "Configure...", allowing to manage external tools (see below),
- "Import..." and "Export...", allowing to share tool definitions between different users using a representation of these definitions in a text file.

2.1.11.1 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 41) and the shortcut for the item. Defining a new tool or editing an existing one opens the tool definition window:



New tool

Menu name: CVS

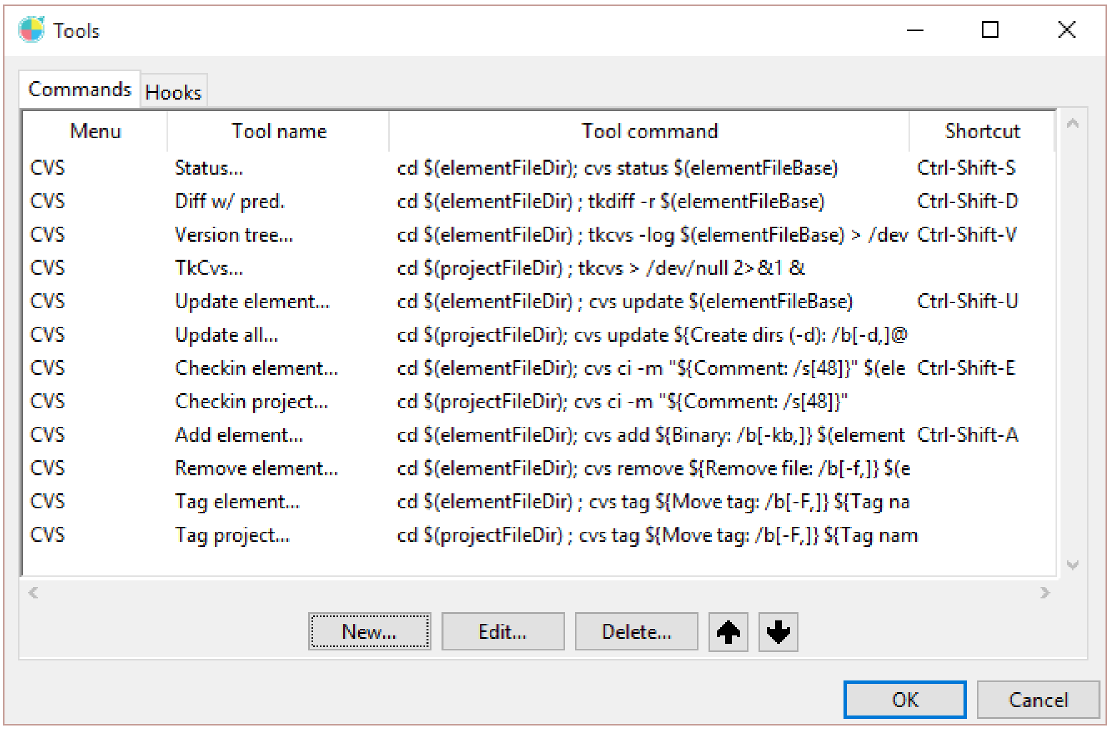
Tool name: Checkin element

Tool command: cd $(elementFileDir); cvs ci -m"${comment:/s[48]}" $(elementFileBase)

Shortcut: Ctrl-Shift-E

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.

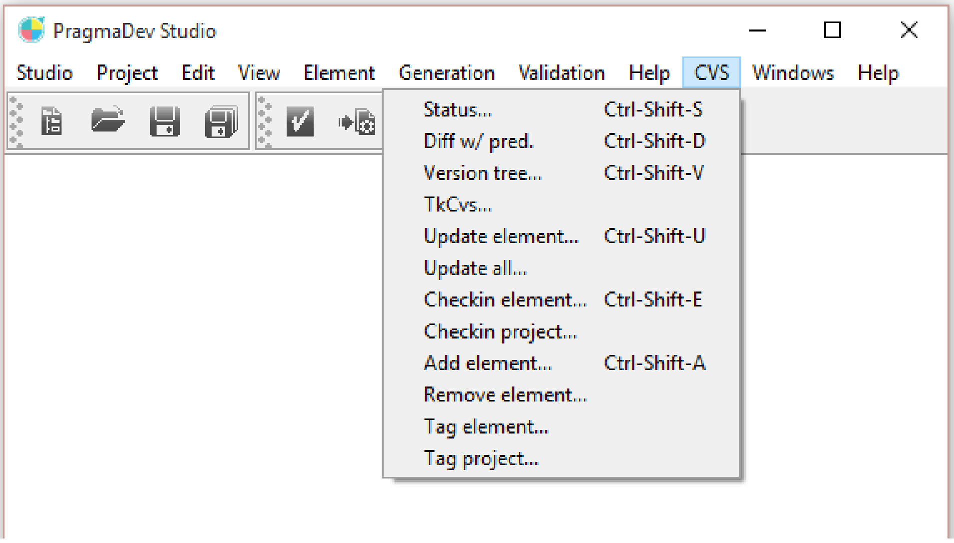
Below is an example of a typical CVS menu:



Tools

Menu	Tool name	Tool command	Shortcut
CVS	Status...	cd $(elementFileDir); cvs status $(elementFileBase)	Ctrl-Shift-S
CVS	Diff w/ pred.	cd $(elementFileDir) ; tkdiff -r $(elementFileBase)	Ctrl-Shift-D
CVS	Version tree...	cd $(elementFileDir) ; tkcvs -log $(elementFileBase) > /dev	Ctrl-Shift-V
CVS	TkCvs...	cd $(projectFileDir) ; tkcvs > /dev/null 2>&1 &	
CVS	Update element...	cd $(elementFileDir) ; cvs update $(elementFileBase)	Ctrl-Shift-U
CVS	Update all...	cd $(projectFileDir); cvs update ${Create dirs (-d): /b[-d,]@	
CVS	Checkin element...	cd $(elementFileDir); cvs ci -m "${Comment: /s[48]}" $(ele	Ctrl-Shift-E
CVS	Checkin project...	cd $(projectFileDir); cvs ci -m "${Comment: /s[48]}"	
CVS	Add element...	cd $(elementFileDir); cvs add ${Binary: /b[-kb,]} $(element	Ctrl-Shift-A
CVS	Remove element...	cd $(elementFileDir); cvs remove ${Remove file: /b[-f,]} $(e	
CVS	Tag element...	cd $(elementFileDir) ; cvs tag ${Move tag: /b[-F,]} ${Tag na	
CVS	Tag project...	cd $(projectFileDir) ; cvs tag ${Move tag: /b[-F,]} ${Tag nam	

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


PragmaDev Studio
Studio Project Edit View Element Generation Validation Help CVS Windows Help
Status... Ctrl-Shift-S
Diff w/ pred. Ctrl-Shift-D
Version tree... Ctrl-Shift-V
TkCvs...
Update element... Ctrl-Shift-U
Update all...
Checkin element... Ctrl-Shift-E
Checkin project...
Add element... Ctrl-Shift-A
Remove element...
Tag element...
Tag project...

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

2.1.11.2 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:

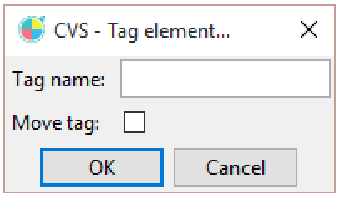
- "`$(projectFile)`": this marker will be replaced by the opened project's full file name. If no project is opened, a message will be displayed and the tool command won't be executed.
- "`$(projectFileDir)`": same as `$(projectFile)`, but for the project directory.
- "`$(projectFileBase)`": same as `$(projectFile)`, but for the project file name without its directory.
- "`$(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:

  
  CVS - Tag element...

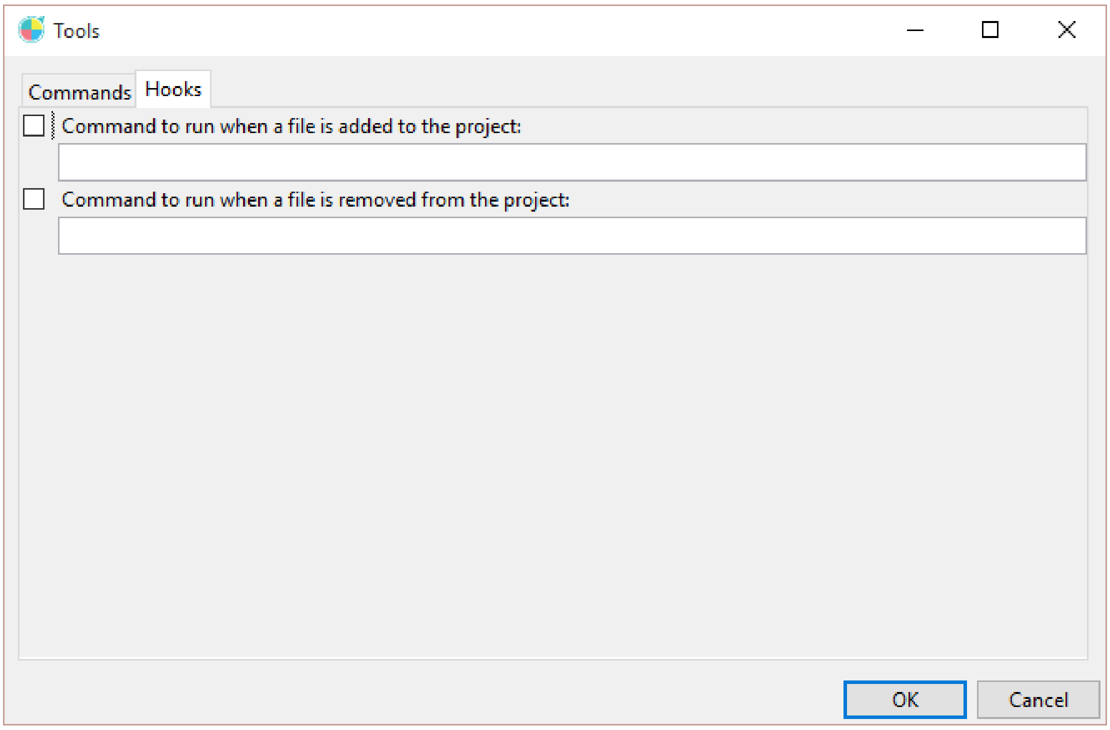
  Tag name:

  Move tag: □

  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.

2.1.11.3 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:



Tools
Commands
Hooks
□ Command to run when a file is added to the project:
□ Command to run when a file is removed from the project:
OK
Cancel

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.

2.1.12 Traceability information

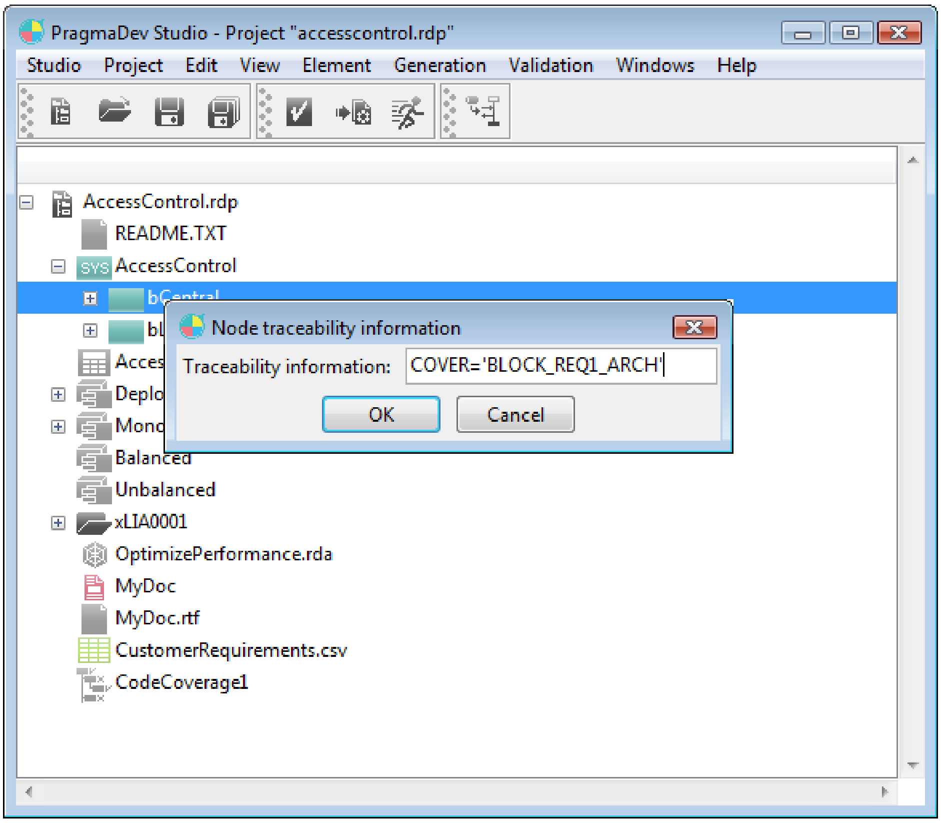
2.1.12.1 Scope

Traceability information can be defined on any element of the *Project manager*. This information can then be used in a traceabilitity information tool in order :

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

2.1.12.2 Traceability editor

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



Traceability information: COVER='BLOCK_REQ1_ARCH'

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

2.1.12.3 Integration with Reqtify

2.1.12.3.1 Organisation

An integration with the Reqtify requirment management tool is delivered with PragmaDev Studio. This integration is located in the sub-directory `share/3rdparty/Reqtify` of the installation directory, which contains the following files:

- `rtds.br`: PragmaDev Studio behavior file. This file contains the scripts allowing Reqtify to import a project file and to open a diagram or text file in it.

- rtds.types: PragmaDev Studio type file. This file describes the format for the imported file in Reqtify.
- pragma.bmp & pragmaM.bmp: PragmaDev logo and its transparency mask. This image is used to represent PragmaDev Studio projects in Reqtify.
- rtds_*.bmp & rtds_*M.bmp: Images for the nodes in a project tree, with their associated transparency mask. All nodes are represented: diagrams, textual files, packages, folders, requirements tables, prototyping GUIs, performance anaysis settings, and so on.
- pragmastudio_RDP_to_Reqtify.exe: Stand alone program that extracts information from PragmaDev Studio project file and generates an information file that conforms to the type defined in rtds.types, and therefore readable by Reqtify.

2.1.12.3.2 Installation for Reqtify

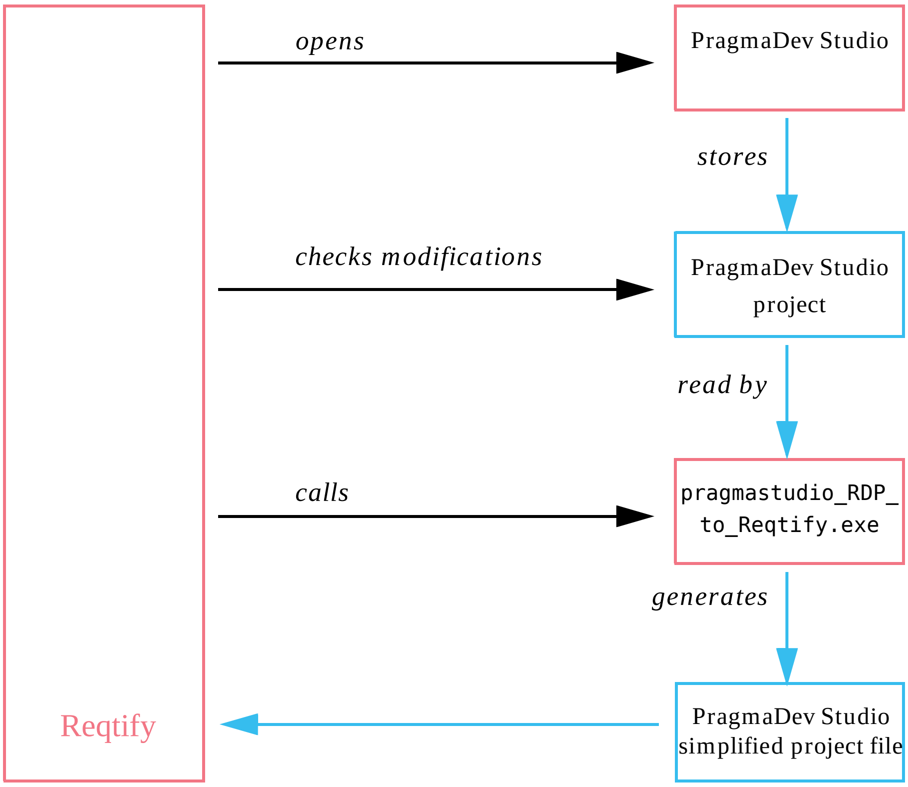
This paragraph make the assumption that the Reqtify toolset is installed in %REQTIFY_HOME% directory (usually C:\Program Files\Reqtify vXXX).. The installation of the integration simply consists in copying the following files from %RTDS_HOME%\share\3rdparty\Reqtify:

- pragmastudio_RDP_to_Reqtify.exe must be copied to:
  %REQTIFY_HOME%\bin.w32;
- rtds.br must be copied to %REQTIFY_HOME%\config\otscript\2;
- rtds.types must be copied to %REQTIFY_HOME%\config\types\Design;
- All .bmp files must be copied to %REQTIFY_HOME%\config\images\RTDS. If the RTDS subdirectory does not exist, it must be created.

Note that the integration may already be present in the Reqtify installation. If there is already a file named rtds.types in %REQTIFY_HOME%\config\types\Design, you probably don't need to do anything.

2.1.12.3.3 General architecture

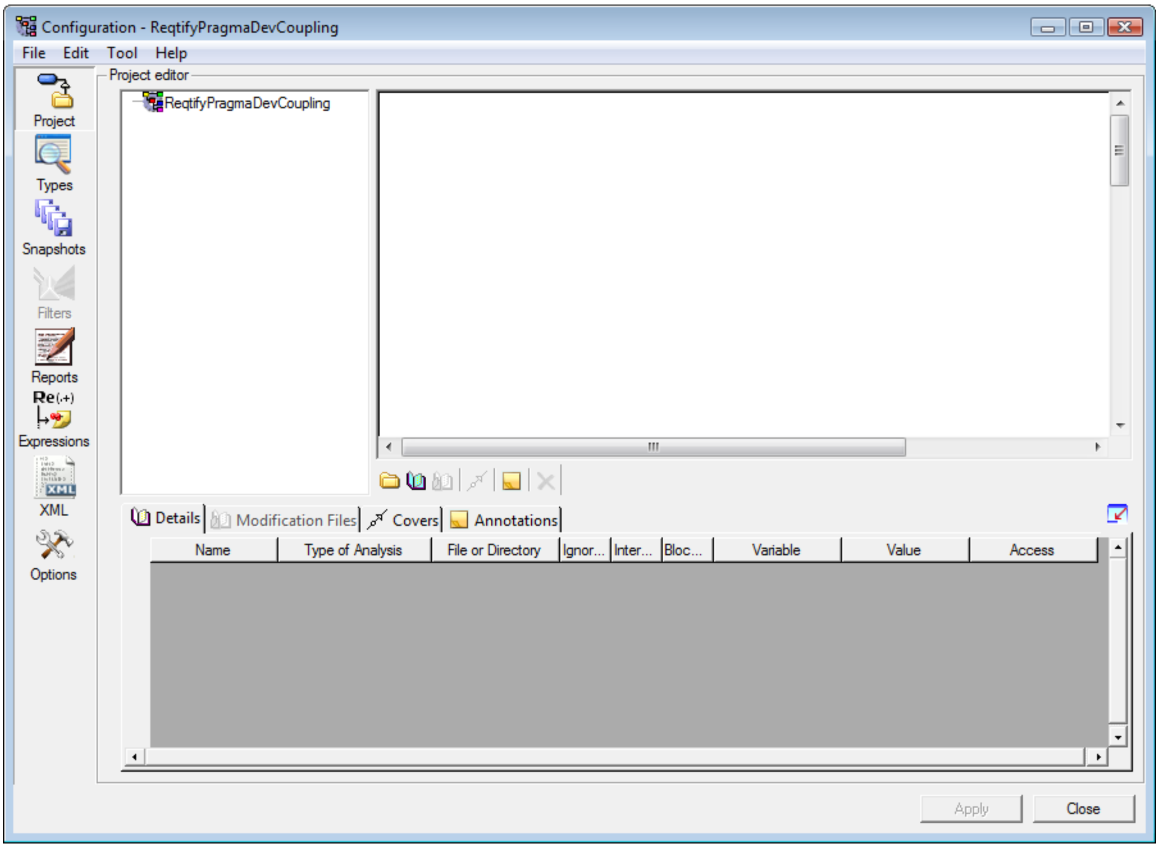
Reqtify relies on these previously installed files to work as described in the following diagram:



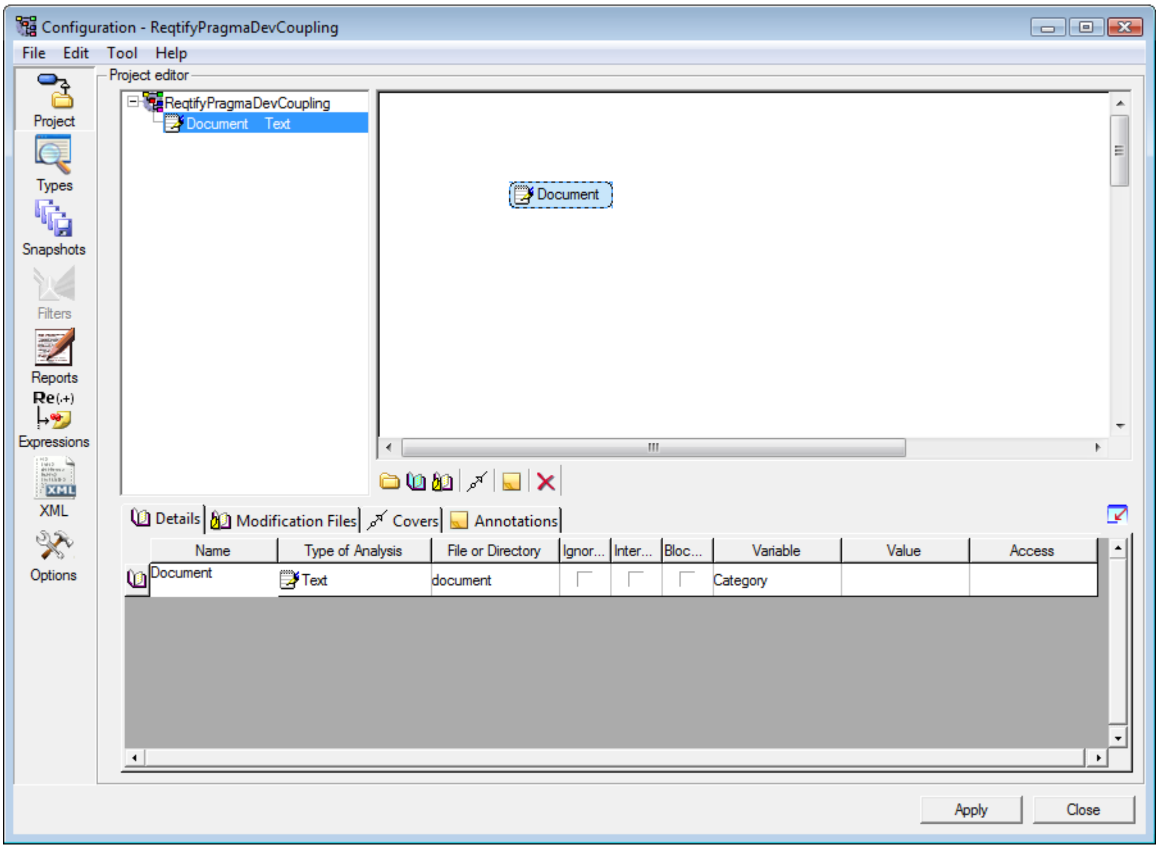
A PragmaDev Studio diagram or a text file in a project can be opened directly from Reqtify. Reqtify also checks if the PragmaDev Studio project file has been modified. If the file has been modified, it calls the `pragmastudio_RDP_to_Reqtify` utility to generates a project file understandable by Reqtify.

2.1.12.3.4 Usage

It is now possible to add a PragmaDev Studio project into Reqtify. To do so, either create a new project, or use the "Edit project..." item in the "File" menu in an existing Reqtify project. The project configuration dialog appears:

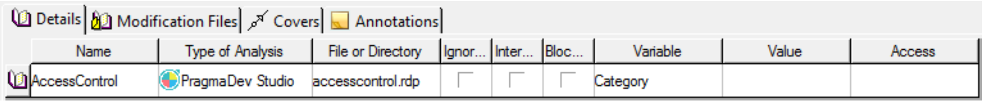


To add a PragmaDev Studio project to the Reqtify project, click the "Add a document" button (📖), then click on the project canvas above:



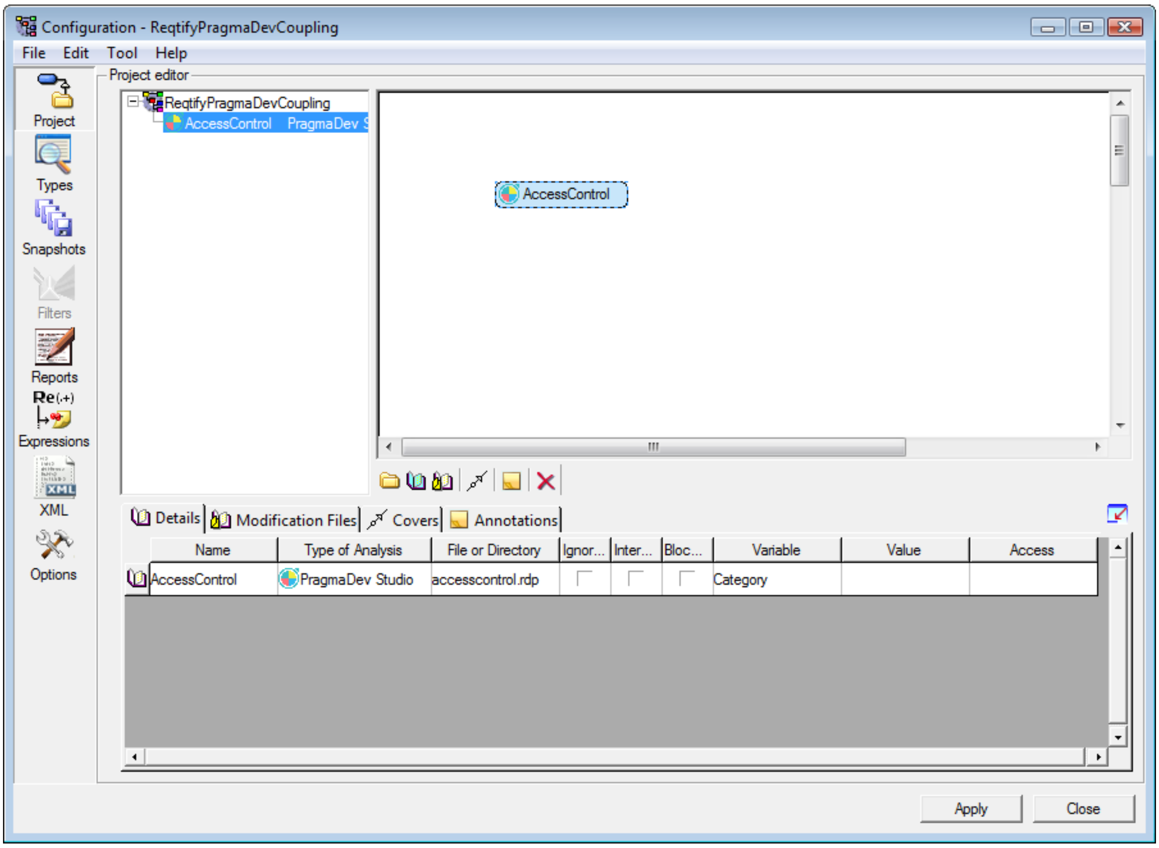
Name	Type of Analysis	File or Directory	Ignor...	Inter...	Bloc...	Variable	Value	Access
Document	Text	document	□	□	□	Category		

Keeping the newly created document selected, use the table below to select the PragmaDev Studio project: give it a name in the "Name" column, set its "Type of Analysis" to "PragmaDev Studio", then click in the "File or Directory" cell, click the "..." button that appears in the cell and select the project file (.rdp):



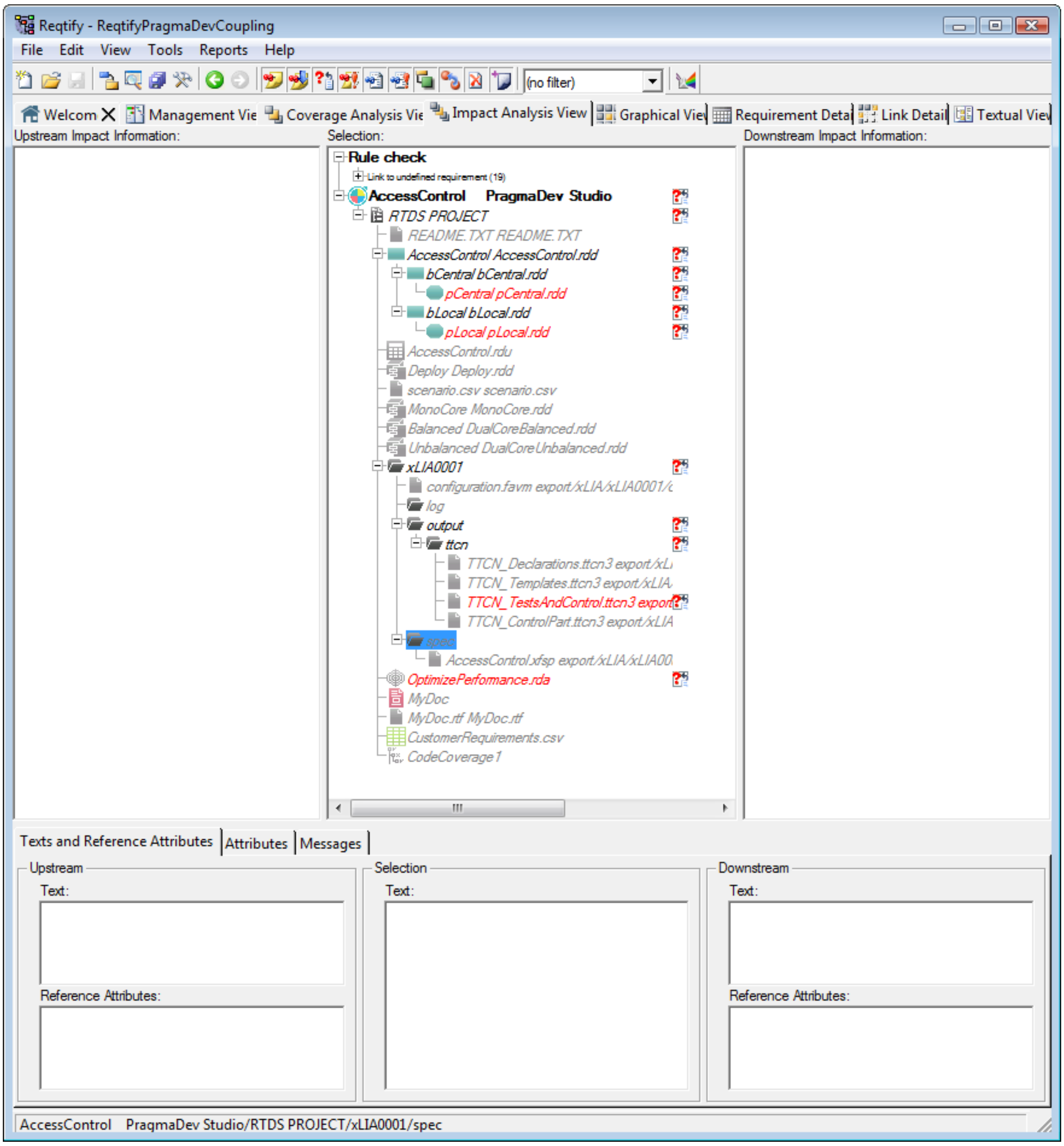
Name	Type of Analysis	File or Directory	Ignor...	Inter...	Bloc...	Variable	Value	Access
AccessControl	PragmaDev Studio	accesscontrol.rdp	□	□	□	Category		

The icon & name for the document then changes in the project canvas to show the document is now the selected PragmaDev Studio project:



Name	Type of Analysis	File or Directory	Ignor...	Inter...	Bloc...	Variable	Value	Access
AccessControl	PragmaDev Studio	accesscontrol.rdp	□	□	□	Category		

Clicking "Apply" will create or update the Reqtify project. The PragmaDev Studio project will then appear in the Reqtify project views in a very similar way as it appears in the project manager:


Reqtify - ReqtifyPragmaDevCoupling
File Edit View Tools Reports Help
(no filter)
Welcom
Management Vie
Coverage Analysis Vie
Impact Analysis View
Graphical Vie
Requirement Deta
Link Detail
Textual Vie
Upstream Impact Information:
Selection:
Downstream Impact Information:
Rule check
Link to undefined requirement (19)
AccessControl PragmaDev Studio
RTDS PROJECT
README.TXT README.TXT
AccessControl AccessControl.rdd
bCentral bCentral.rdd
pCentral pCentral.rdd
bLocal bLocal.rdd
pLocal pLocal.rdd
AccessControl.rdu
Deploy Deploy.rdd
scenario.csv scenario.csv
MonoCore MonoCore.rdd
Balanced DualCoreBalanced.rdd
Unbalanced DualCoreUnbalanced.rdd
xLIA0001
configuration.favm export/xLIA/xLIA0001/c
log
output
ttcn
TTCN_Declarations.ttcn3 export/xLI
TTCN_Templates.ttcn3 export/xLIA
TTCN_TestsAndControl.ttcn3 export
TTCN_ControlPart.ttcn3 export/xLIA
spec
AccessControl.xfsp export/xLIA/xLIA00
OptimizePerformance.rda
MyDoc
MyDoc.rtf MyDoc.rtf
CustomerRequirements.csv
CodeCoverage1
Texts and Reference Attributes
Attributes
Messages
Upstream
Text:
Reference Attributes:
Selection
Text:
Downstream
Text:
Reference Attributes:
AccessControl PragmaDev Studio/RTDS PROJECT/xLIA0001/spec

Double-clicking in Reqtify on a diagram or text file node will open the imported PragmaDev Studio project, then open the diagram or text file.

2.1.12.3.5 Format for traceability information

The traceability information for all nodes in the project tree has the following formats:

- For a node covering one or more requirements:
  `cover="<REQ. ID 1> <REQ. ID 2> ..."`
- For a node defining one or more requirements:
  `requirement="<REQ. ID 1> <REQ. ID 2> ..."`

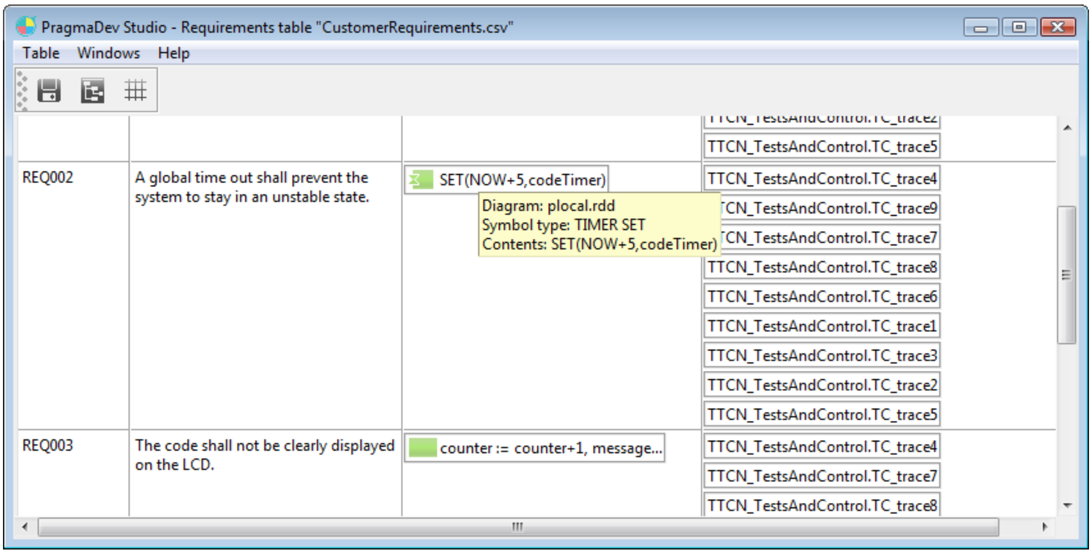
The requirement identifiers are those appearing in the other documents in the Reqtify project.

2.1.12.3.6 Information imported from requirements tables

The PragmaDev Studio / Reqtify integration also automatically uses the information stored in requirements tables in the PragmaDev Studio project to automatically create covered and/or defined requirements:

- The identifier used for the requirement is the one appearing in the first column of the requirements table.
- A covered requirement is automatically added to the parent diagram of every symbol that appears in the third column of the table.
- A covered requirement is automatically added to the parent TTCN file of every testcase that appears in the fourth column of the table.

For example, for the following line in a requirements table:


PragmaDev Studio - Requirements table "CustomerRequirements.csv"
Table
Windows
Help

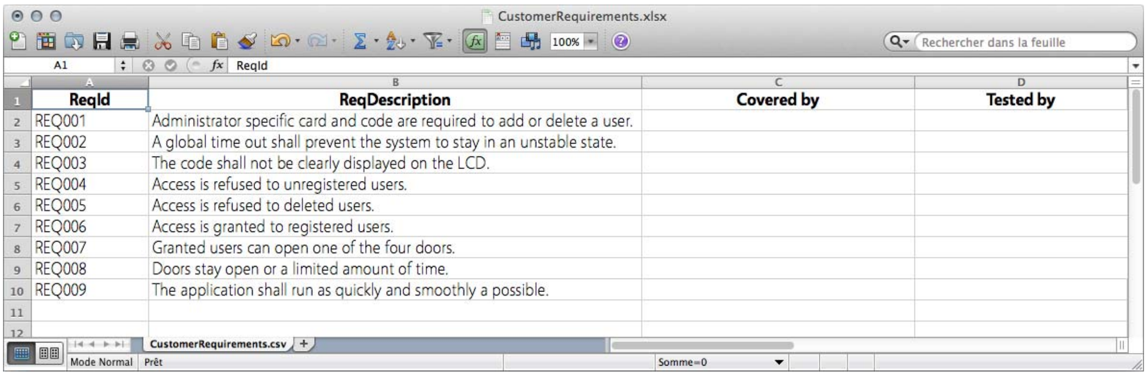
			TTCN_TestsAndControl.TC_trace2 TTCN_TestsAndControl.TC_trace5
REQ002	A global time out shall prevent the system to stay in an unstable state.	SET(NOW+5,codeTimer)	TTCN_TestsAndControl.TC_trace4 TTCN_TestsAndControl.TC_trace9 TTCN_TestsAndControl.TC_trace7 TTCN_TestsAndControl.TC_trace8 TTCN_TestsAndControl.TC_trace6 TTCN_TestsAndControl.TC_trace1 TTCN_TestsAndControl.TC_trace3 TTCN_TestsAndControl.TC_trace2 TTCN_TestsAndControl.TC_trace5
REQ003	The code shall not be clearly displayed on the LCD.	counter := counter+1, message...	TTCN_TestsAndControl.TC_trace4 TTCN_TestsAndControl.TC_trace7 TTCN_TestsAndControl.TC_trace8

Diagram: plocal.rdd
Symbol type: TIMER SET
Contents: SET(NOW+5,codeTimer)

the requirement `REQ002` will be automatically added as a covered requirement to the node for diagram `pLocal.rdd` in the project tree, and to the node for the TTCN source file `TTCN_TestsAndControl.ttcn`, since it contains all the testcases listed as testing it in the table.

2.1.12.3.7 Example

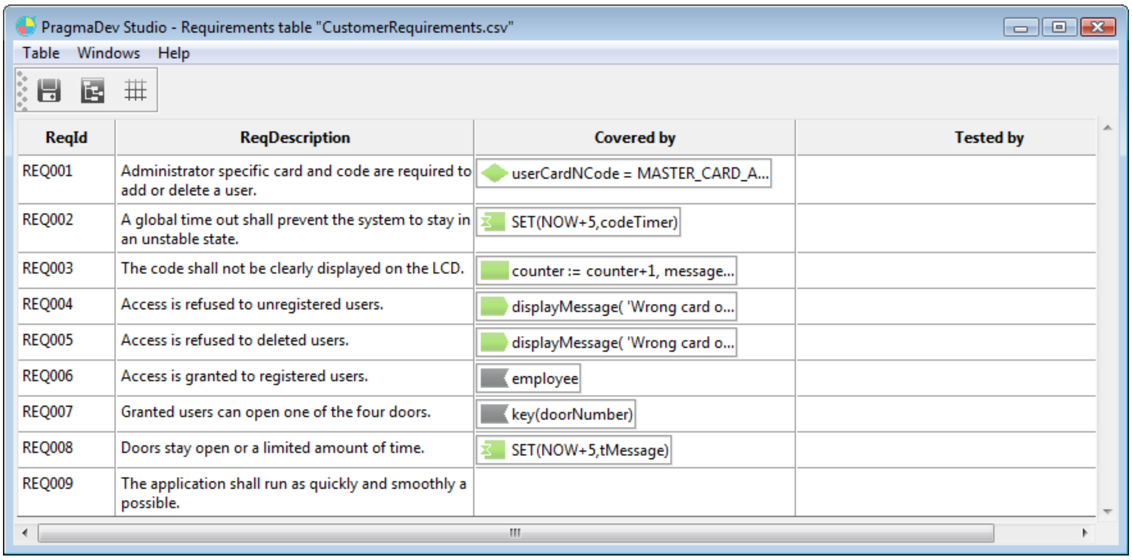
The following example creates a link from an Excel spreadsheet to a PragmaDev Studio project. The Excel spreadsheet is actually the one used to generate the requirements table in CSV format that is imported in the PragmaDev Studio project:



ReqId	ReqDescription	Covered by	Tested by
REQ001	Administrator specific card and code are required to add or delete a user.		
REQ002	A global time out shall prevent the system to stay in an unstable state.		
REQ003	The code shall not be clearly displayed on the LCD.		
REQ004	Access is refused to unregistered users.		
REQ005	Access is refused to deleted users.		
REQ006	Access is granted to registered users.		
REQ007	Granted users can open one of the four doors.		
REQ008	Doors stay open or a limited amount of time.		
REQ009	The application shall run as quickly and smoothly a possible.		

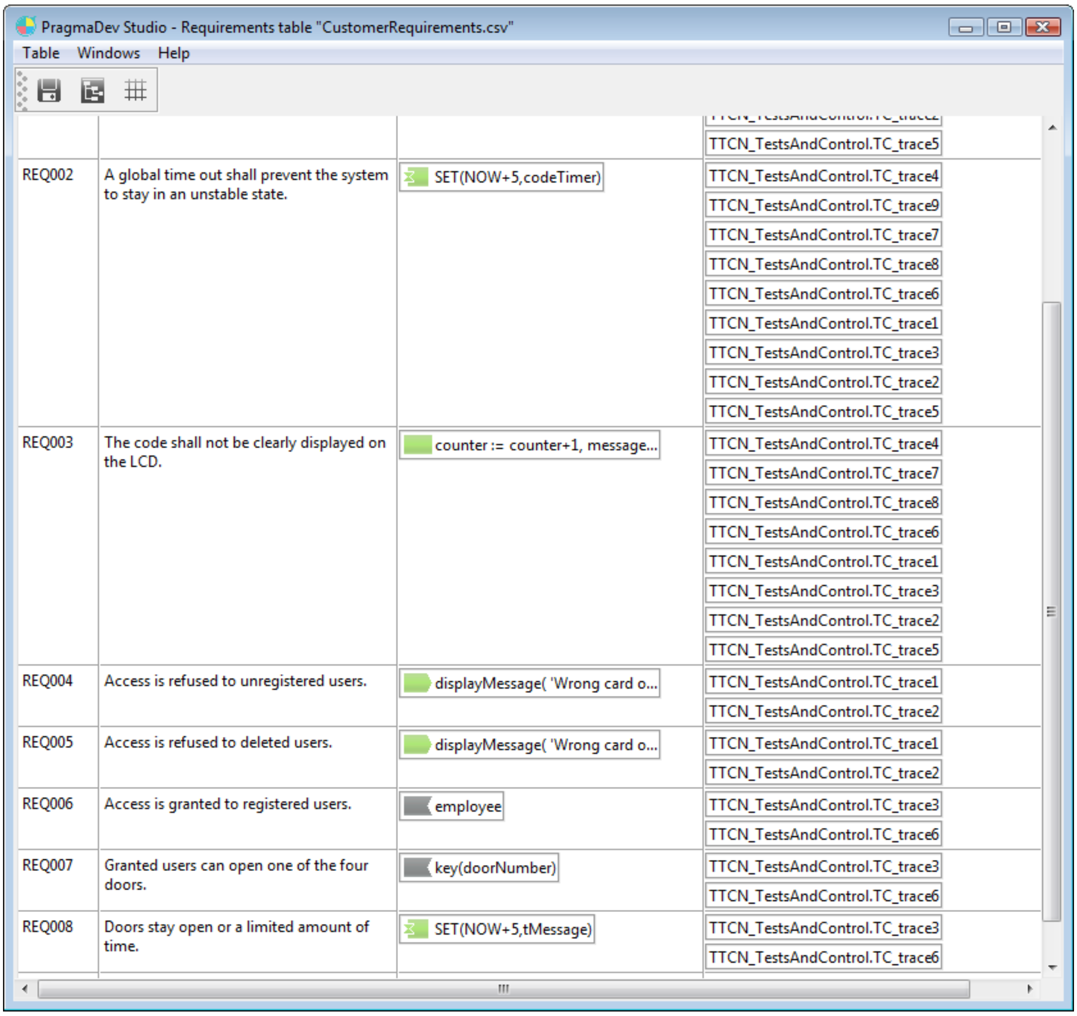
The requirements are listed the normal way: requirement identifier in the first column, requirement description in the second one, then 2 columns, one for the symbols covering the requirement and another one for the testcases testing it.

After exporting it in CSV format and importing it in PragmaDev Studio, the symbols covering the requirements can be inserted in the table as explained in “Covering symbols” on page 168:


PragmaDev Studio - Requirements table "CustomerRequirements.csv"
Table
Windows
Help

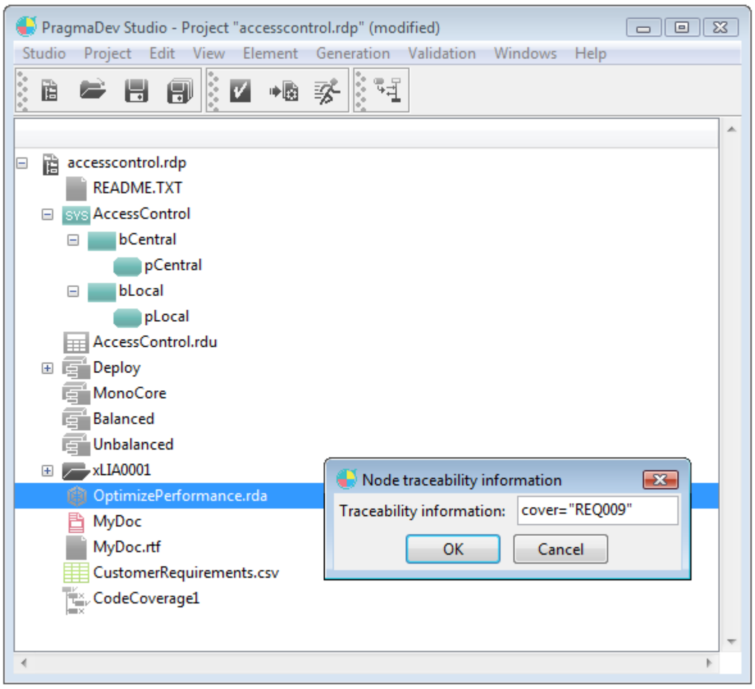
ReqId	ReqDescription	Covered by	Tested by
REQ001	Administrator specific card and code are required to add or delete a user.	userCardNCode = MASTER_CARD_A...	
REQ002	A global time out shall prevent the system to stay in an unstable state.	SET(NOW+5,codeTimer)	
REQ003	The code shall not be clearly displayed on the LCD.	counter := counter+1, message...	
REQ004	Access is refused to unregistered users.	displayMessage('Wrong card o...	
REQ005	Access is refused to deleted users.	displayMessage('Wrong card o...	
REQ006	Access is granted to registered users.	employee	
REQ007	Granted users can open one of the four doors.	key(doorNumber)	
REQ008	Doors stay open or a limited amount of time.	SET(NOW+5,tMessage)	
REQ009	The application shall run as quickly and smoothly a possible.		

After that, running a co-simulation with the testcases automatically inserts the testcases in the 4th column:


PragmaDev Studio - Requirements table "CustomerRequirements.csv"
Table
Windows
Help

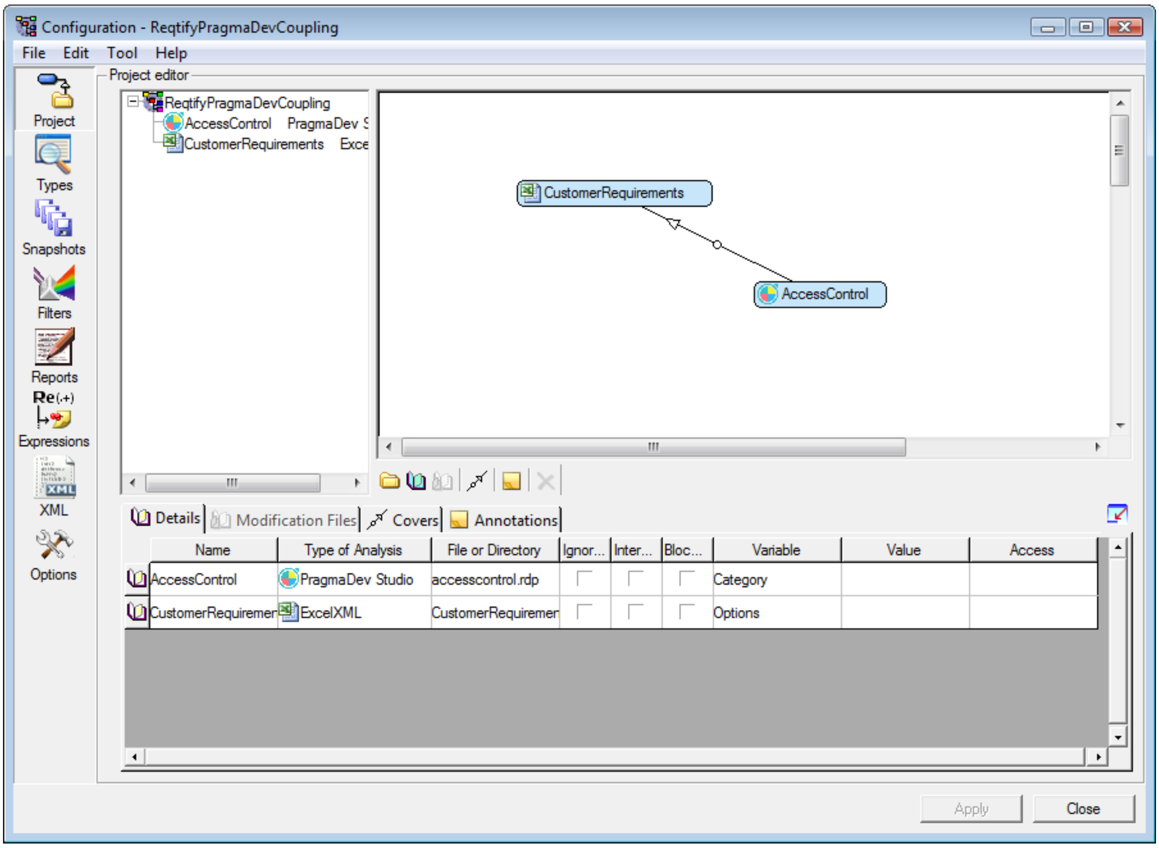
			TTCN_TestsAndControl.TC_trace2 TTCN_TestsAndControl.TC_trace5
REQ002	A global time out shall prevent the system to stay in an unstable state.	SET(NOW+5,codeTimer)	TTCN_TestsAndControl.TC_trace4 TTCN_TestsAndControl.TC_trace9 TTCN_TestsAndControl.TC_trace7 TTCN_TestsAndControl.TC_trace8 TTCN_TestsAndControl.TC_trace6 TTCN_TestsAndControl.TC_trace1 TTCN_TestsAndControl.TC_trace3 TTCN_TestsAndControl.TC_trace2 TTCN_TestsAndControl.TC_trace5
REQ003	The code shall not be clearly displayed on the LCD.	counter := counter+1, message...	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
REQ004	Access is refused to unregistered users.	displayMessage('Wrong card o...	TTCN_TestsAndControl.TC_trace1 TTCN_TestsAndControl.TC_trace2
REQ005	Access is refused to deleted users.	displayMessage('Wrong card o...	TTCN_TestsAndControl.TC_trace1 TTCN_TestsAndControl.TC_trace2
REQ006	Access is granted to registered users.	employee	TTCN_TestsAndControl.TC_trace3 TTCN_TestsAndControl.TC_trace6
REQ007	Granted users can open one of the four doors.	key(doorNumber)	TTCN_TestsAndControl.TC_trace3 TTCN_TestsAndControl.TC_trace6
REQ008	Doors stay open or a limited amount of time.	SET(NOW+5,tMessage)	TTCN_TestsAndControl.TC_trace3 TTCN_TestsAndControl.TC_trace6

One of the requirements is not covered by a symbol in a diagram, so it can be set via the traceability information on the corresponding node in the project tree:



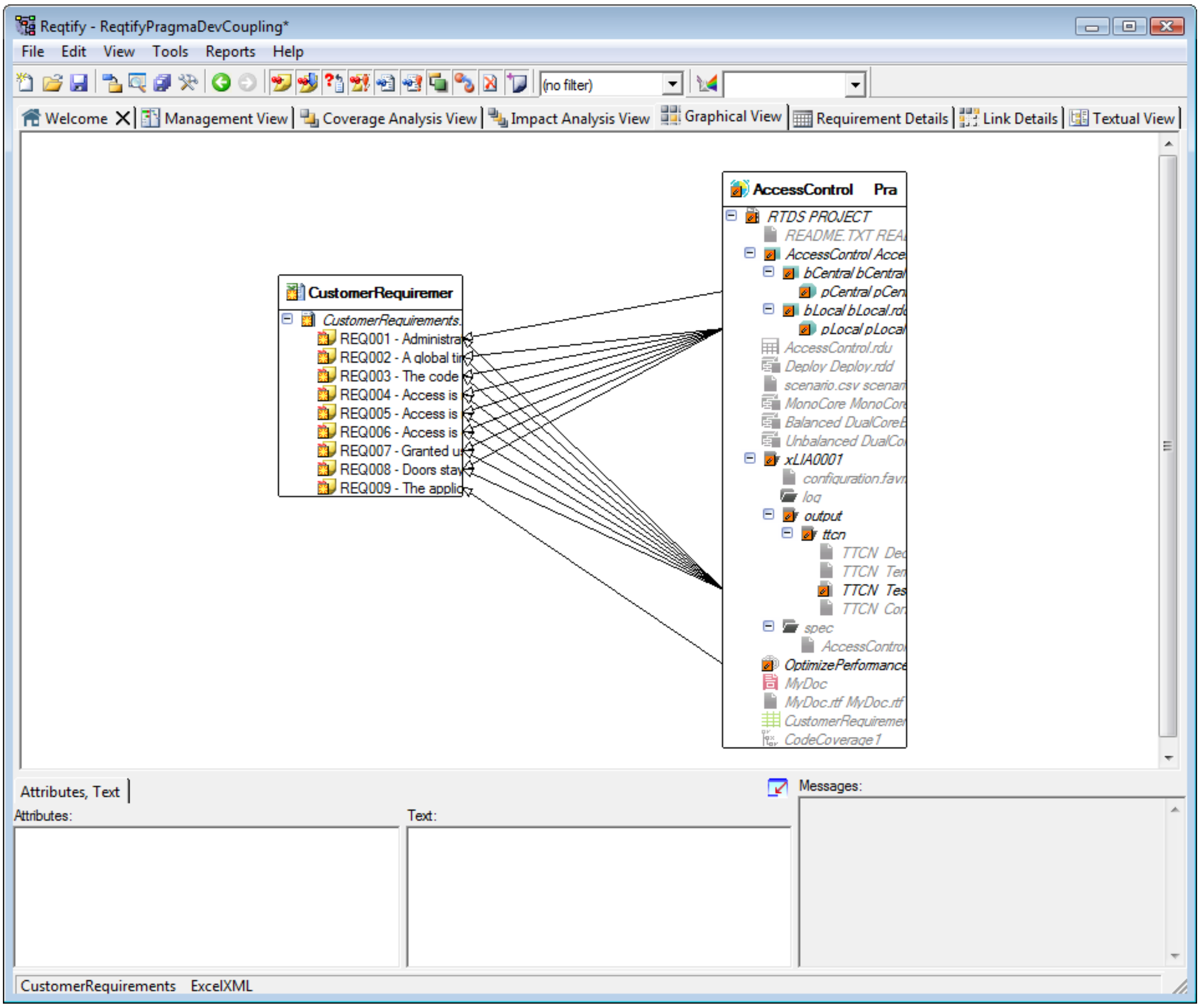
Traceability information: cover="REQ009"

Both the Excel spreadsheet and the PragmaDev Studio project can then be added in a Reqtify project, with a cover link between the two:



Name	Type of Analysis	File or Directory	Ignor...	Inter...	Bloc...	Variable	Value	Access
AccessControl	PragmaDev Studio	accesscontrol.rdp	□	□	□	Category		
CustomerRequiremer	ExcelXML	CustomerRequiremer	□	□	□	Options		

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

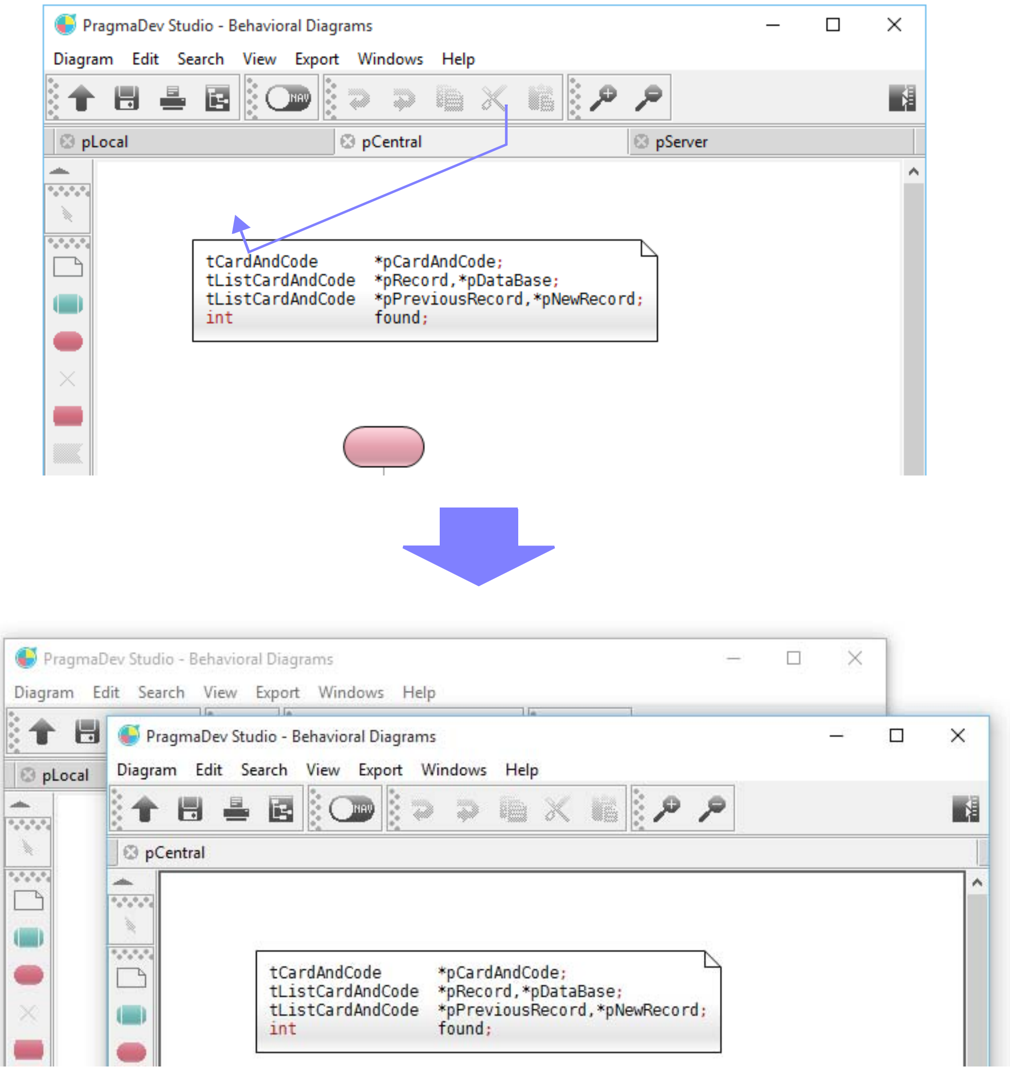


2.2 - Editor windows

2.2.1 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.

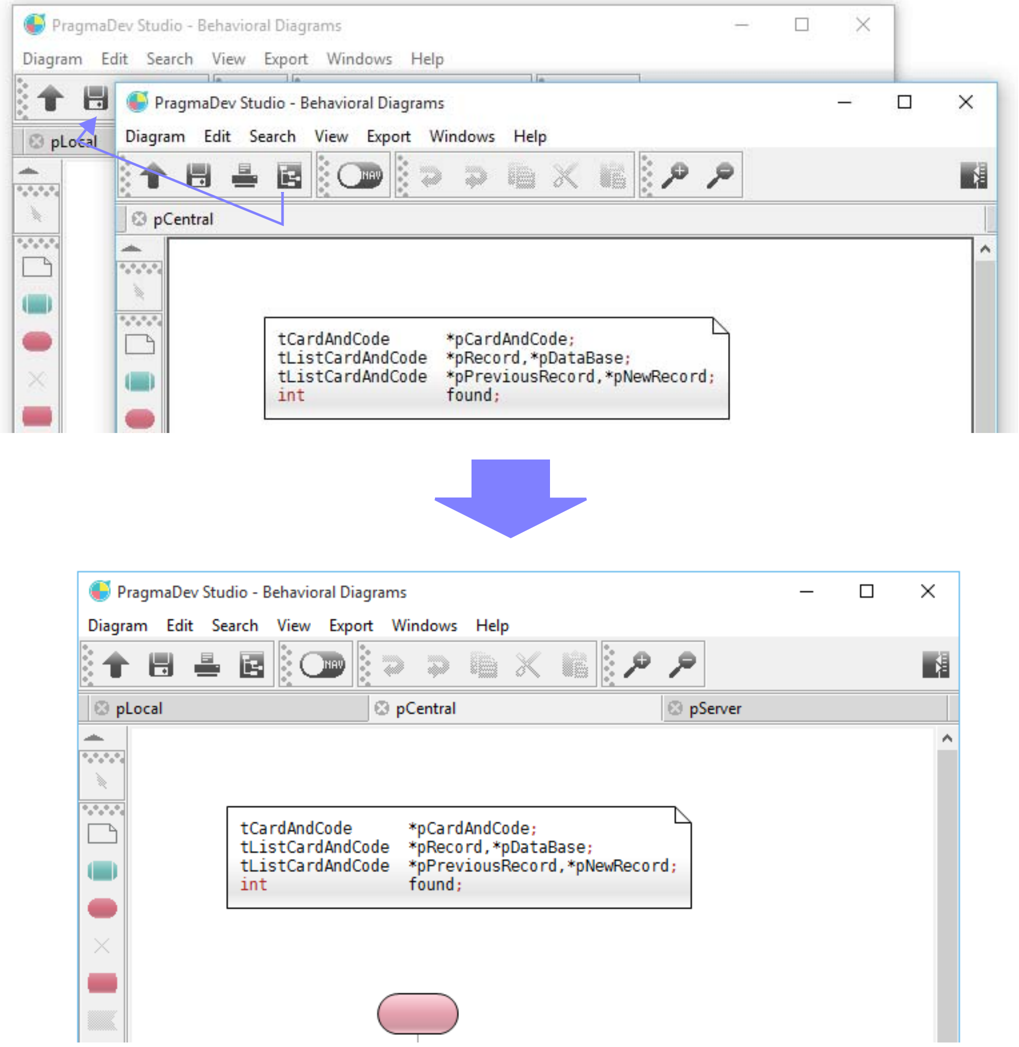
If needed, a tab can be dragged out of its parent window to create a new window:



```
tCardAndCode      *pCardAndCode;
tListCardAndCode  *pRecord,*pDataBase;
tListCardAndCode  *pPreviousRecord,*pNewRecord;
int               found;
```

```
tCardAndCode      *pCardAndCode;
tListCardAndCode  *pRecord,*pDataBase;
tListCardAndCode  *pPreviousRecord,*pNewRecord;
int               found;
```

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:



```
tCardAndCode       *pCardAndCode;
tListCardAndCode   *pRecord,*pDataBase;
tListCardAndCode   *pPreviousRecord,*pNewRecord;
int                found;
```

```
tCardAndCode       *pCardAndCode;
tListCardAndCode   *pRecord,*pDataBase;
tListCardAndCode   *pPreviousRecord,*pNewRecord;
int                found;
```

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 34). 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 ne element is opened in the same window.

2.2.2 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 size and position...*" records the position and size of the current window and applies both automatically for all new windows of the same kind;

- "*Forget recorded size & pos...*" 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.
- "*Restore recorded size & pos...*" will put back the window to the recorded size and position if any. This can be useful when the window has been moved or resized.

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 window kinds in the current session:

- If a window of this kind has tabs, the menu entry is the window kind (e.g "Diagram editor"), with an associated cascade menu having one entry for each tab in any window of this kind. The entry for the tab is the file name for the opened diagram. Selecting this entry will raise the corresponding editor window, and select the specified tab.
- If a window of this kind 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.

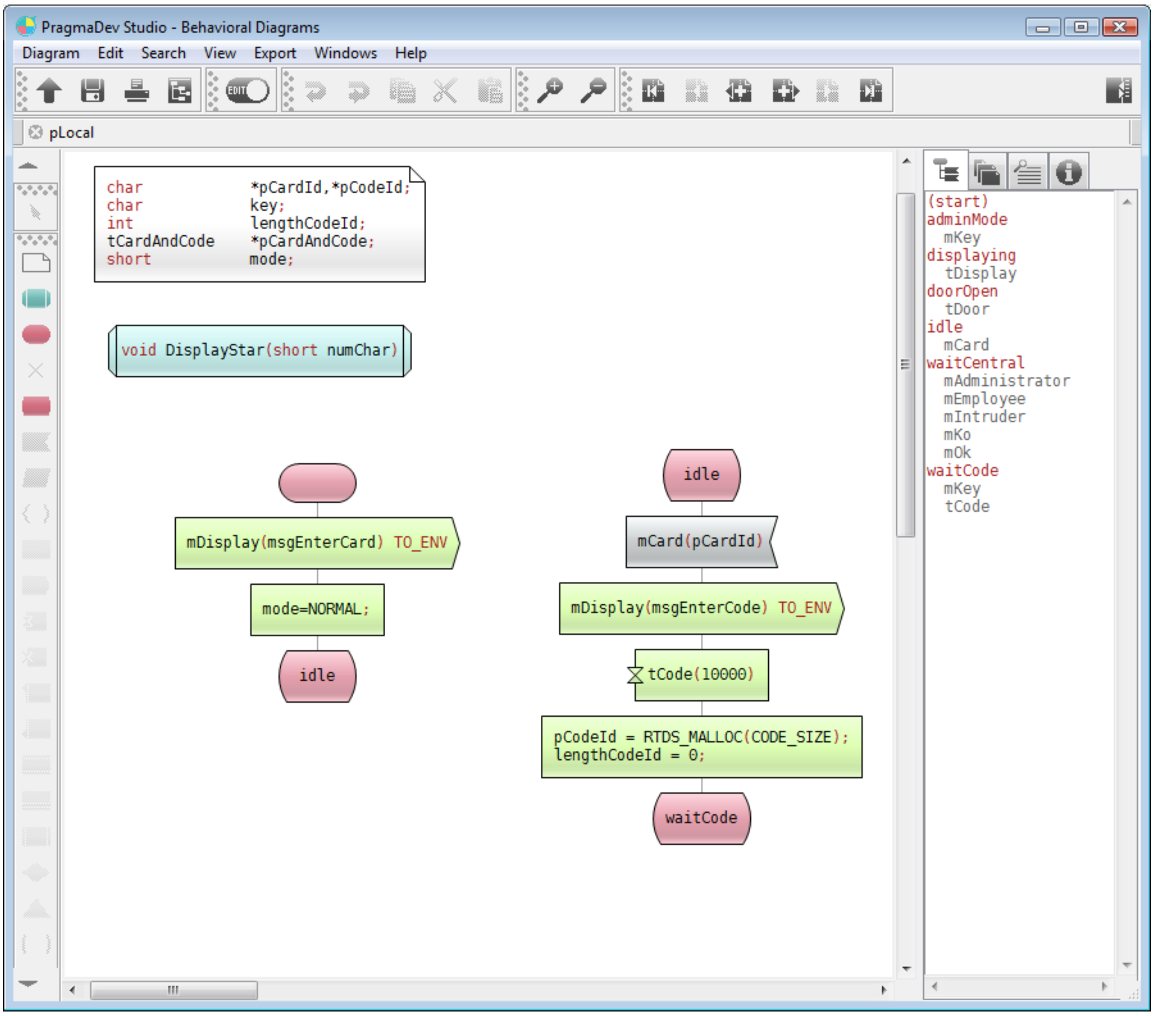
Note that the project manager also always has an entry in this menu, which is always the first and is named "Project", followed by the name of the currently opened project.

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.

2.3 - Diagram editor

Two types of diagrams exist in PragmaDev Studio. Normal mode diagrams are diagrams actually made with PragmaDev Studio, and legacy mode diagrams are diagrams made with the older version of PragmaDev Studio, called Real Time Developer 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.



```
char            *pCardId,*pCodeId;
char            key;
int             lengthCodeId;
tCardAndCode    *pCardAndCode;
short           mode;
```

```
void DisplayStar(short numChar)
```

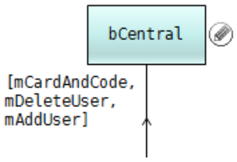
Diagram editor window

2.3.1 Common features

2.3.1.1 Editor modes

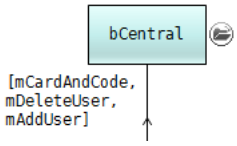
Most editors have two modes, that can be switched either via two options in the “View” menu, or via the EDIT / MAN button in the toolbar:

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



  Clicking on a symbol selects it, and double-clicking anywhere in it opens the diagram associated to it if any.

- In edition mode, symbol texts are editable directly by clicking on the text. To select it, the click has to happen on the symbol, but outside its 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:

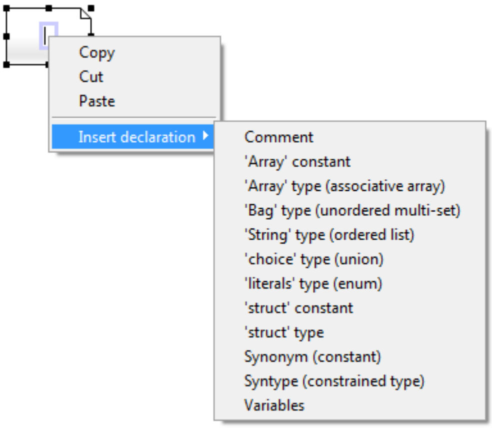


  The button will only appear if there is an associated diagram to the symbol, or if there can be one, even if it doesn't exist yet.
  As clicking between a symbol'stext and its border can sometimes be tricky, there's also another way to select a symbol: when nothing is selected, shift-clicking on it anywhere will select it, even if the click is done in the text.

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 clicking on them.

2.3.1.2 Contextual help for declarations

For symbols containing declarations, help is available via the contextual menu triggered by right clicking in the symbol text while it is opened for modification:



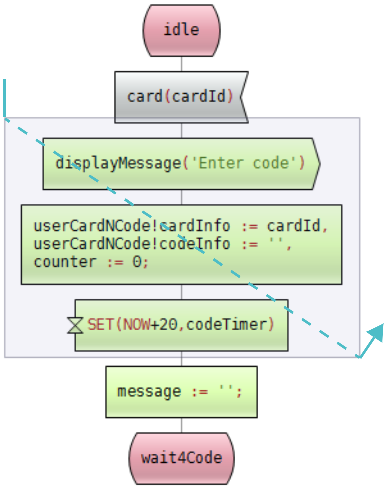
For each declaration kind is given its name in the current language, as well as its equivalent in C when applicable, or a short explanation of what the type is. Selecting an entry in the menu will insert a skeleton for this kind of declaration at the current insertion position.

Note that todaay, this menu is only available in SDL declaration symbols.

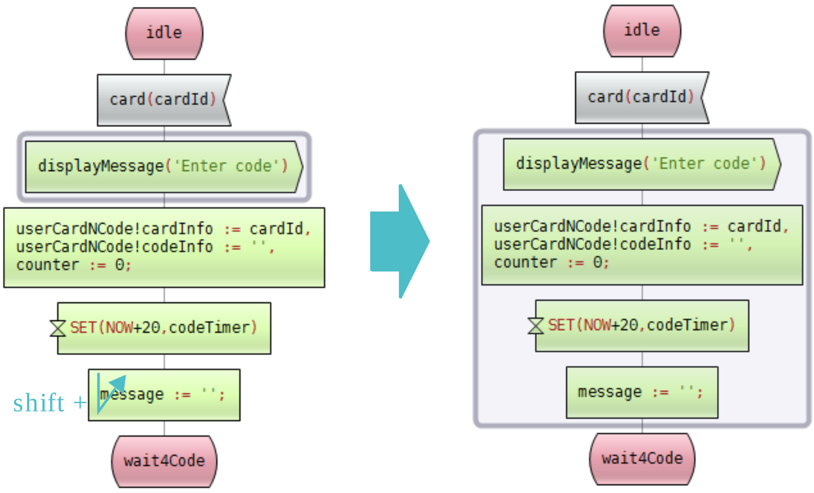
2.3.1.3 Selecting multiple symbols

Multiple selection is available in most diagrams. Note however that the selection can be restrained to something meaningful. For example, in a behavioral diagram, you will only be able to select symbols that are in the same transition or decision branch and that follow each other. This is because no operation could be performed on a random set of symbols.

Multiple selection is done by surrounding the symbols to select with the selection rectangle:


idle
card(cardId)
displayMessage('Enter code')
userCardNCode!cardInfo := cardId,
userCardNCode!codeInfo := '',
counter := 0;
SET(NOW+20,codeTimer)
message := '';
wait4Code

or by extending the selection by shift-clicking on other symbols. Note that in behavioral diagrams, this will select all symbols between the selected one and the shift-clicked one:


idle
card(cardId)
displayMessage('Enter code')
userCardNCode!cardInfo := cardId,
userCardNCode!codeInfo := '',
counter := 0;
SET(NOW+20,codeTimer)
shift +
message := '';
wait4Code
idle
card(cardId)
displayMessage('Enter code')
userCardNCode!cardInfo := cardId,
userCardNCode!codeInfo := '',
counter := 0;
SET(NOW+20,codeTimer)
message := '';
wait4Code

2.3.1.4 Frame concept

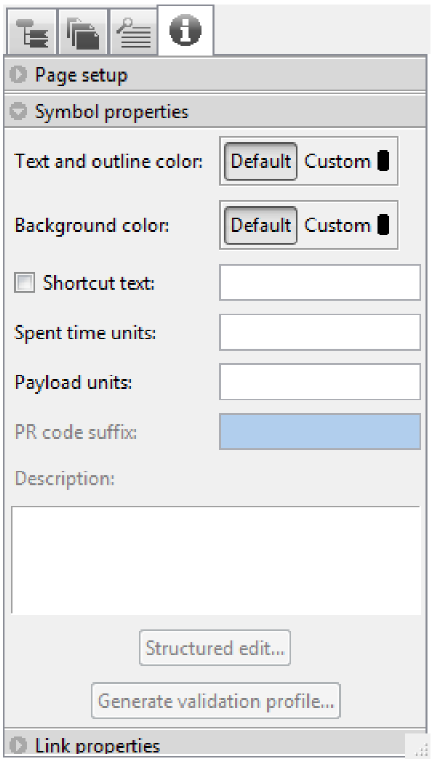
All legacy diagrams have a surrounding frame, containing all the symbols in the diagram. You can't put symbols outside that frame. For SDL system and block diagrams, the frame also represents the external boundary of the agent, and channels may be connected to it.

In the new editors for MSC and behavioral diagrams introduced in PragmaDev Studio V5, there is no surrounding frame, as it was not used for these kind of diagrams.

2.3.1.5 Symbol and link properties

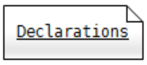
Each symbol or link has a property sheet allowing to enter all its features in a guided way. The actual properties depend on the type of the symbol or link. It may be opened by selecting a symbol or link and choose "Properties..." in the contextual menu; it will open in the browser zone on the right side of the diagram editor window.

The basic property sheet for a symbol is the following:

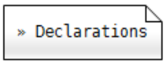


- Text and outline color allows to select the color of the text in the symbol and its outline.
- Background color allows to select the background color of the symbol.
- The symbol shortcut text may be used to specify an alternate text to display in the symbol and to open its "real" text in an external 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:



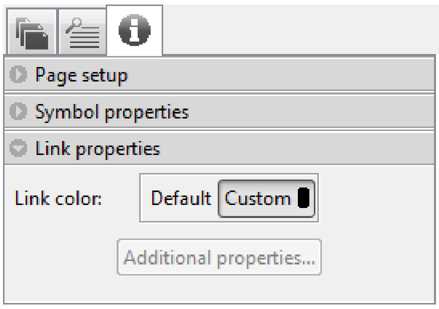
- If the "*Prefix for shortcut text instead of underlining*" is checked, it will appear with a prefix:



Double-clicking on the symbol shortcut text will open a text editor showing the symbol's 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 405).

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



Link color: Default [✓ Custom]

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

2.3.1.6 Moving symbols

Symbols in a diagram may be moved by using the mouse. Some diagrams also allow to select a symbol or a group of symbols and to move them with the arrow keys:

- Pressing the arrow keys alone will move the symbol or group by one grid cell;
- Pressing the arrow keys with the control key depressed will move the symbol or group by one point.

When possible, the symbol moves are constrained by the diagram logic:

- In MSC diagrams, lifelines can only be moved horizontally, and all the events happening on them move with the lifeline. Also, condition, MSC references and inline expression symbols can only be moved vertically so that the lifelines they concern stay the same.
- In behavioral diagrams, moving a start, state or label symbol will move the entire transition or block of transitions attached to it. The moves of transitions in a

state block are also constrained: they can only be moved horizontally to reorder them.

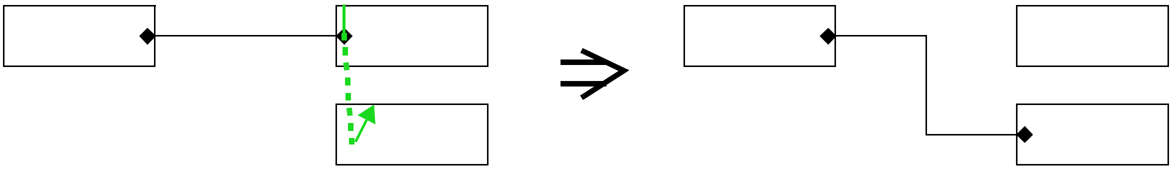
NB: MSC or behavioral diagrams in legacy form actually allow to move any symbol in every direction.

2.3.1.7 Modifying links

There are 2 kinds of links that can appear in diagrams:

- Straight links appear only in MSC diagrams: the link goes in a straight line from its starting point to its ending point, the line possibly being horizontal or oblique. These links cannot be directly edited: to modify the link, the end points have to be moved and the link follows.
- Broken links are present in most other diagrams: they are made of several segments that can be either vertical or horizontal. There are no oblique lines in a broken link. These also have end points that can be moved.

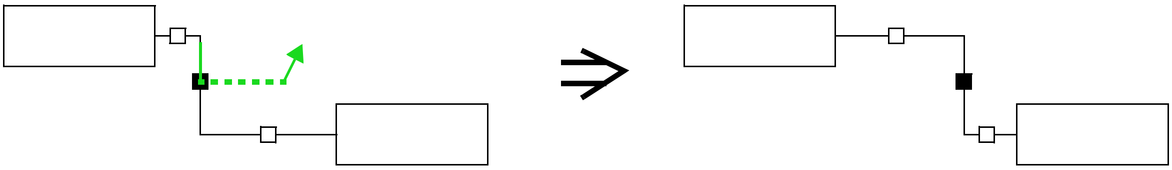
In the diagram editor, the end points of a link can be moved via the diamond-shaped handles appearing when the link is selected:



press on handle, then drag and release

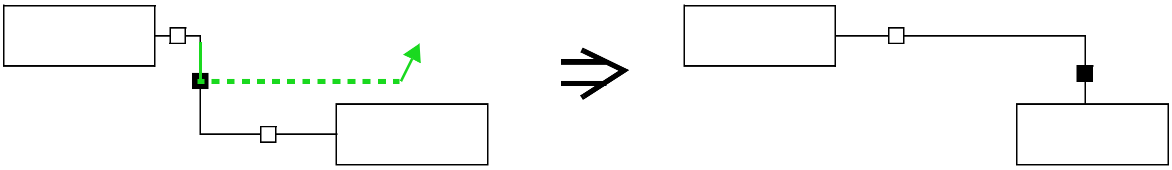
These handles can be moved at another spot on the same symbol (e.g, moving a link end along a lifeline in a MSC), or from one symbol to another.

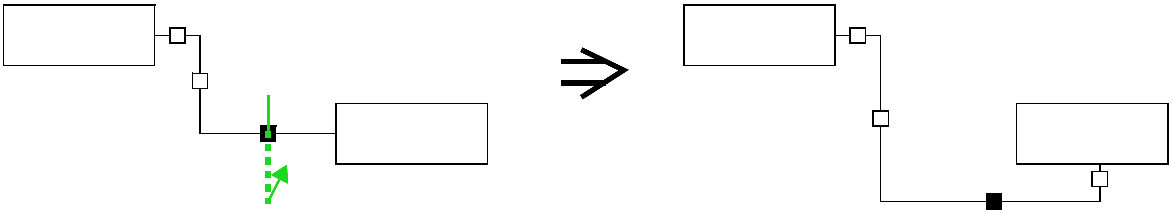
If a link have segments, a segment may be moved via the handle that appears in the the its middle when the link is selected:



press on handle, then drag and release

If needed, segments will be automatically created or deleted:





NB: in behavioral diagrams, the links are only a representation of the symbol sequence and can neither be selected, nor modified.

2.3.1.8 Button and tool bars

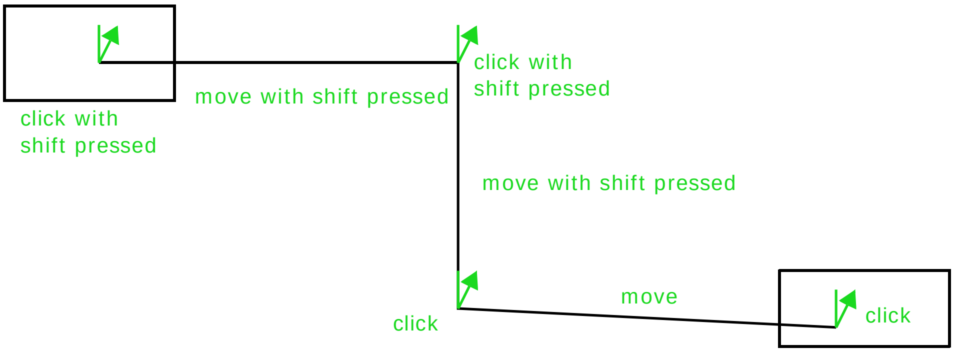
The diagram editor window has several button bars and tool bars:

- The button bars are horizontal and are placed at the top of the window, just above the tabs. They give access to all usual operations.
- The tool bars are vertical and are placed on the window's left side. They allow to insert elements in the diagram: symbols, links, or other items, depending on the kind of diagram. An additional tool bar - usually the first one in the column - allows to switch back to selection mode.

Insertion of symbols is usually done by clicking on the symbol insertion tool, then clicking on where the symbol should be created in the diagram zone. In behavioral diagrams, though, when a symbol must be created after another one in a transition, the predecessor symbol must be selected, and clicking on the symbol insertion button will create the new symbol as a successor of the selected one automatically. Note that only valid symbol types for successors of the selected one will be active.

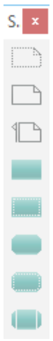
Depending on the diagram type, a link insertion tool bar may also be present. This tool bar will contain one button for each link type allowed in the diagram. Make sure you select the right type for the symbols you want to link, as all symbols do not accept all kinds of links between them.

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:



Other tool bars may be present, as for example in the MSC diagram where there is a specific tool bar for lifeline components: timer starts, time-out and ends, time constraints, liefeline segments (method, suspended), action symbols, and so on.

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 34). Button bars will however always be detachable.

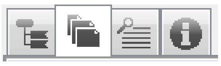
Notes:

- Some diagrams have a tiny lock icon looking like this: 🔒 at the top of the tool bars zone. This allows to turn on or off the “sticky” mode for symbol and link insertion:
    - The default behavior is to insert a single symbol or link: once one symbol or link has been inserted, the editor returns to selection mode.
    - Clicking once on the icon turns on the “sticky” mode: the editor will no more return to selection mode after the first symbol or link insertion, and 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 legacy form of the MSC diagrams have specific tools that appear in the same tool bar as the selection tool. See “Specific tools” on page 84.

2.3.1.9 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 “🗂” tab at the top of the browsers:



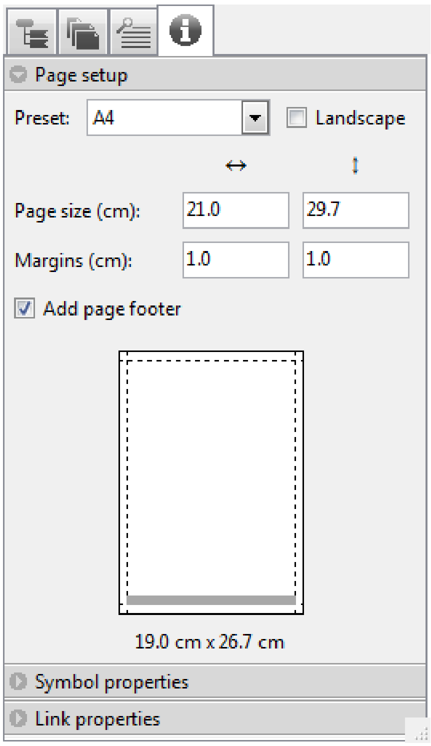
Notes:

- A partition can only be deleted if it doesn’t contain any symbol or link.

- Partitions are not available in MSC diagrams.

2.3.1.10 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. It will appear in the browser zone on the right side of the diagram window:



Preset: A4 □ Landscape

Page size (cm): 21.0 29.7

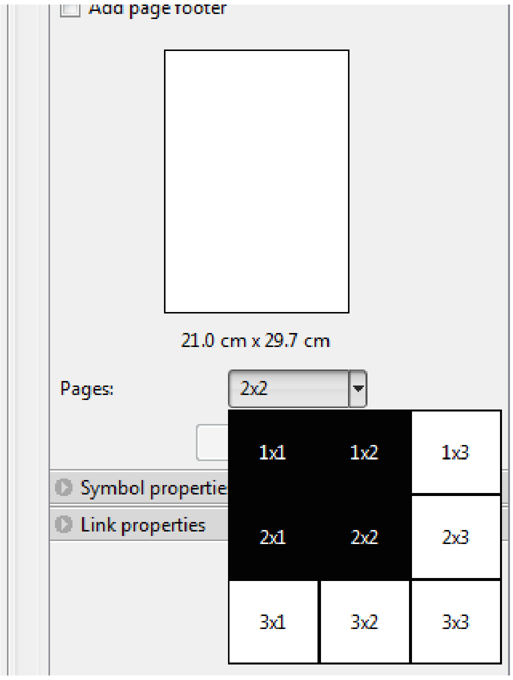
Margins (cm): 1.0 1.0

☑ Add page footer

19.0 cm x 26.7 cm

The fields “*Preset*”, “*Landscape*”, and “*Page size (cm)*” are used to specify the paper size. The “*Margins (cm)*” 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 bottom part.

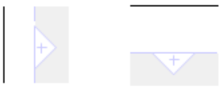
The number of pages in MSC and behavioral diagrams will grow as needed as symbols are added to them. For other kinds of diagrams, there is a fixed number of pages that has to be defined via an additional menu:



The “*Pages*” field allows to specify the number of rows and columns of pages in the partition. Note that for diagrams showing this field, the new page setup will have to be explicitely applied in order to be taken into account: since the configured zone can be too small to contain all the existing diagram contents, the change may fail, so it cannot be applied automatically.

A default page setup can be configured in the project manager (menu "Studio", "Default page setup..."). This default will be used for all elements in the project that cannot have their own, for example textual files, or the project itself.

For diagram having a fixed preset page number, there is a shortcut to add a row or column of pages on every partition: along each side of the edge pages, the following buttons appear:



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

2.3.1.11 Publications

2.3.1.11.1 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 26).

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:

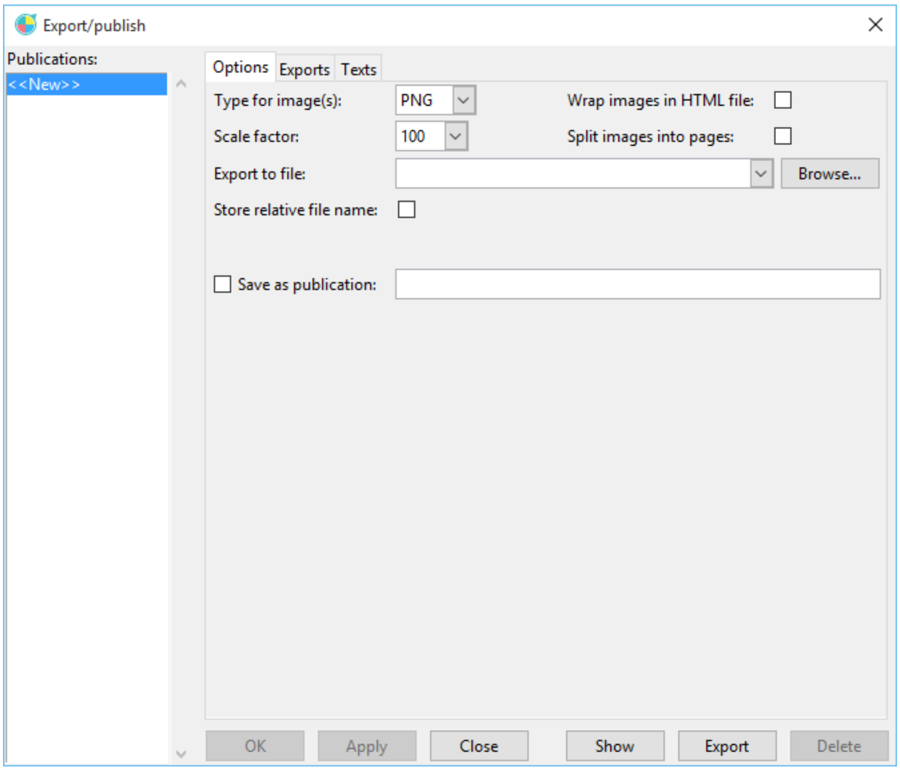
- When saving a diagram having publication, a dialog may appear asking whether to update the publications. This behavior is controlled by the "Update publication" option in the diagram preferences (see “Diagram preferences” on page 26).
- On Windows or macOS, the selected symbols may also be directly 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 other 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 `<img src="">` 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.

2.3.1.11.2 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:



Export/publish

Publications:

<<New>>

Options | Exports | Texts

Type for image(s): PNG

Wrap images in HTML file: □

Scale factor: 100

Split images into pages: □

Export to file:

Store relative file name: □

□ Save as publication:

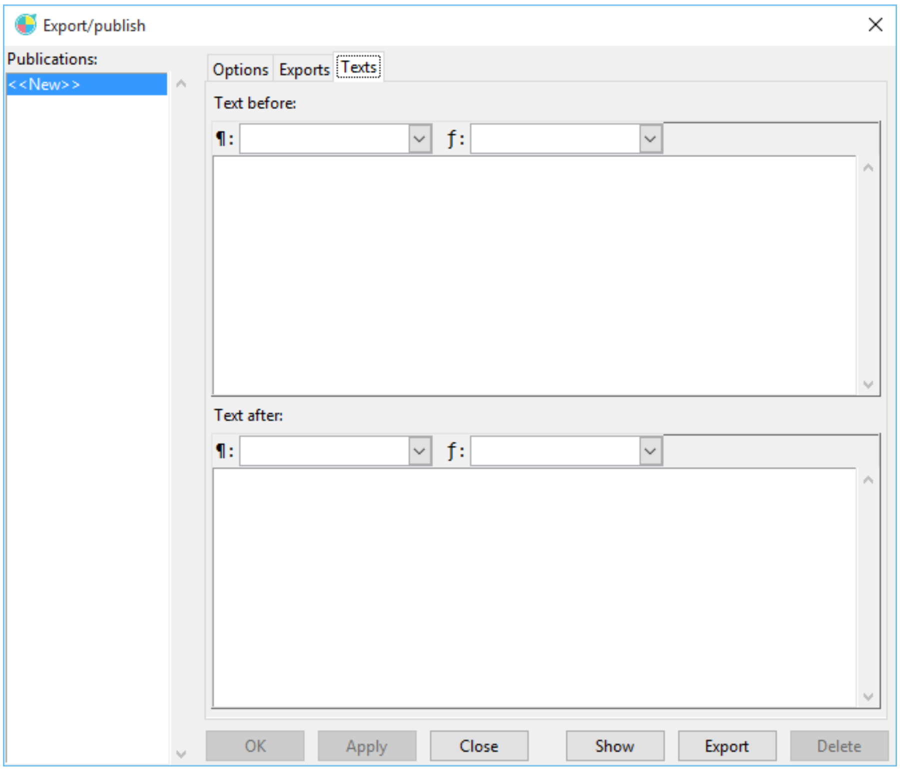
The dialog allows to set:

- 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 126.
- 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 contain 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.

2.3.1.11.3 Documenting a publication

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

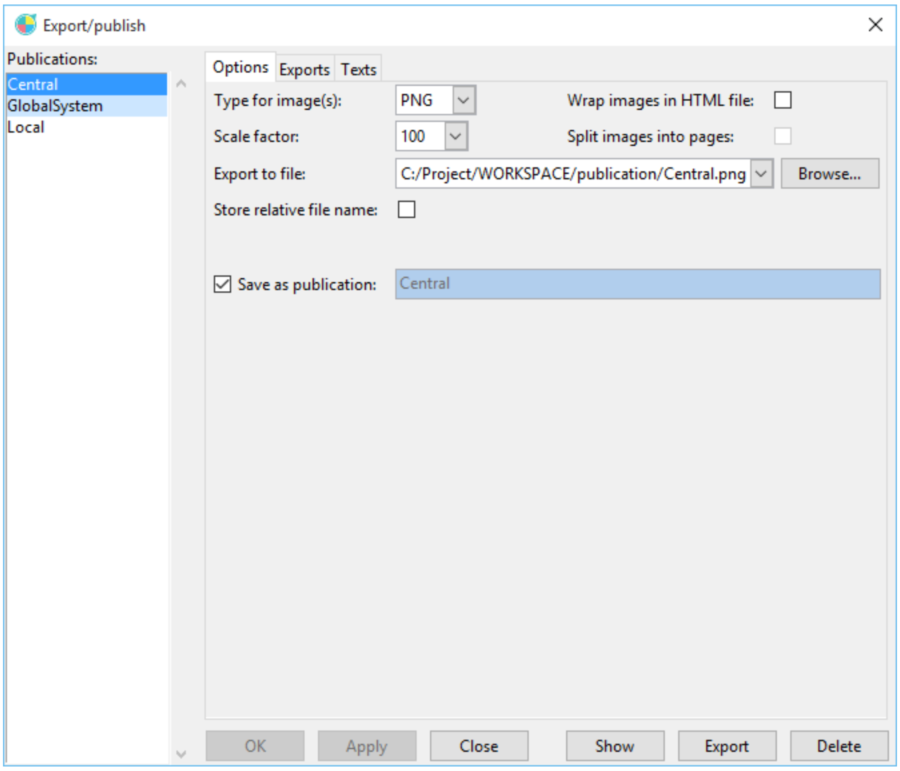


These editors are used to create or edit the styled texts that will automatically appear before and after the publication image when included in a document. For more information, see “Styled text editor” on page 140.

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’.

2.3.1.11.4 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:



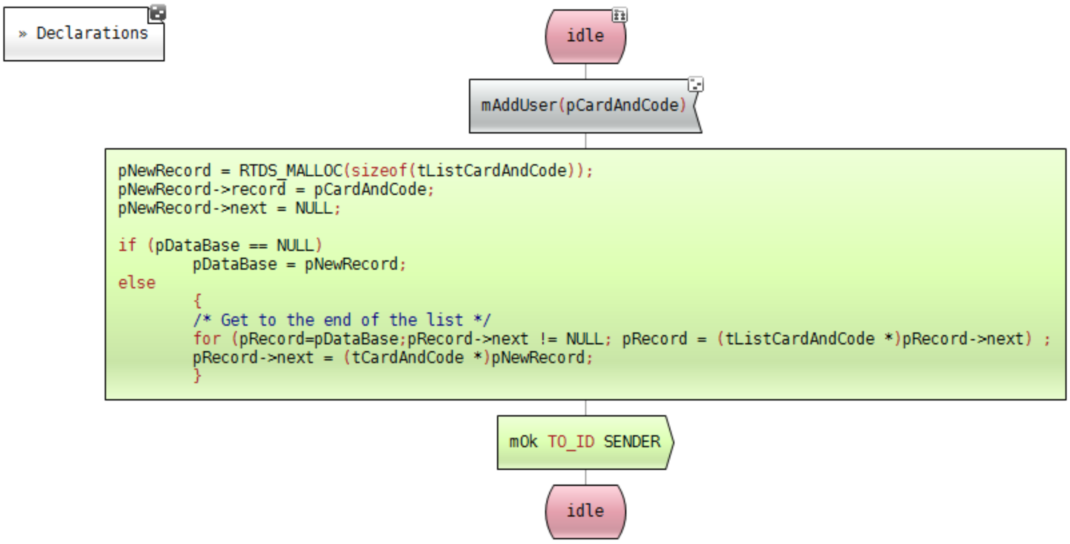
Publications:
Central
GlobalSystem
Local

Type for image(s): PNG
Wrap images in HTML file: □
Scale factor: 100
Split images into pages: □
Export to file: C:/Project/WORKSPACE/publication/Central.png
Store relative file name: □
☑ Save as publication: Central

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.

2.3.1.11.5 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:



```
pNewRecord = RTDS_MALLOC(sizeof(tListCardAndCode));
pNewRecord->record = pCardAndCode;
pNewRecord->next = NULL;

if (pDataBase == NULL)
        pDataBase = pNewRecord;
else
        {
        /* Get to the end of the list */
        for (pRecord=pDataBase;pRecord->next != NULL; pRecord = (tListCardAndCode *)pRecord->next) ;
        pRecord->next = (tCardAndCode *)pNewRecord;
        }
```

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 wether 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 130). 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 72).

2.3.2 SDL editor features

2.3.2.1 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 10).

The definition diagrams for symbols are not added to the project as described in the paragraph “Adding components to the system” on page 14, 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 14 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.

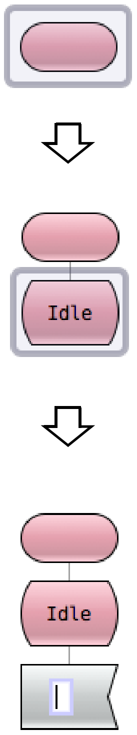
Please note:

- 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.

2.3.2.2 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:



Manually inserted "Start" symbol
Make sure it is selected

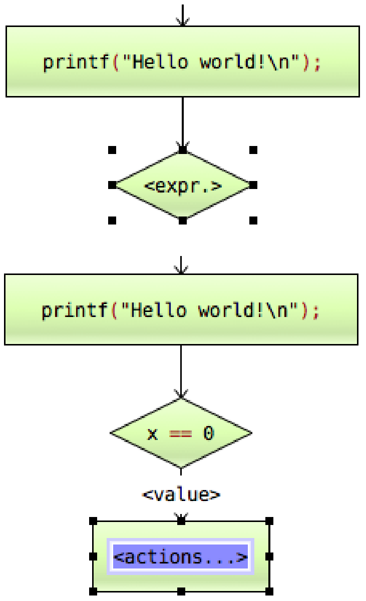
Double-click(legacy) or click (normal) on "State" symbol;

"State" symbol automatically inserted
below current symbol.

Double-click(legacy) or click (normal) on "Signal input" symbol;

"Signal input" symbol is automatically inserted
below current symbol.

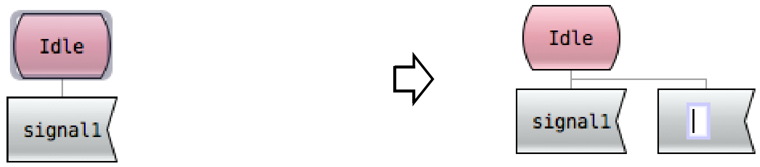
The automatic insertion will also choose automatically the "best" link type for linking the two symbols:

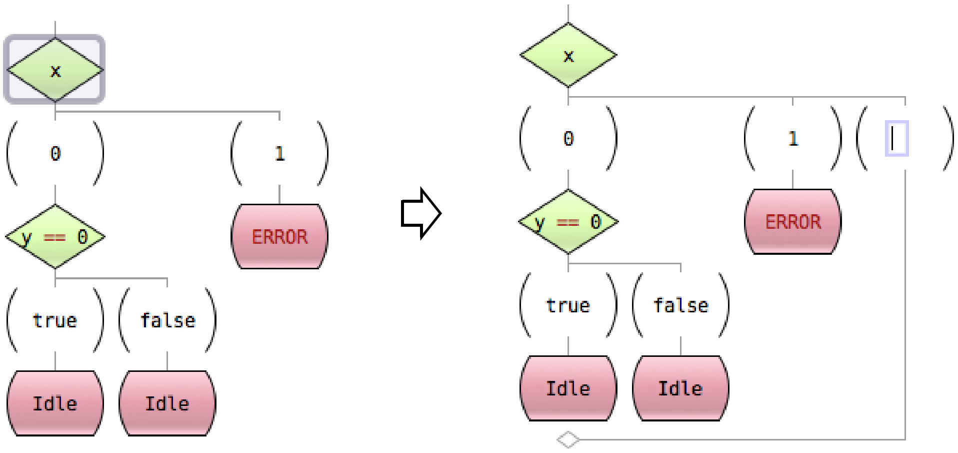


Automatic insertion of a condition after a task block

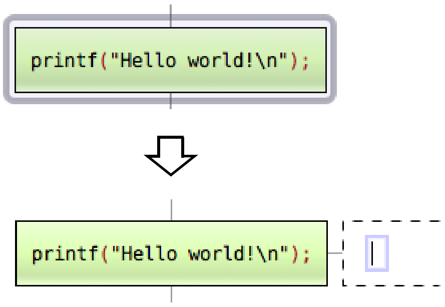
Automatic insertion of a task block after a condition

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





Auto-insertion is also provided for comment and text extension symbols where they are supported. These symbols will be inserted at the right of the selected one:

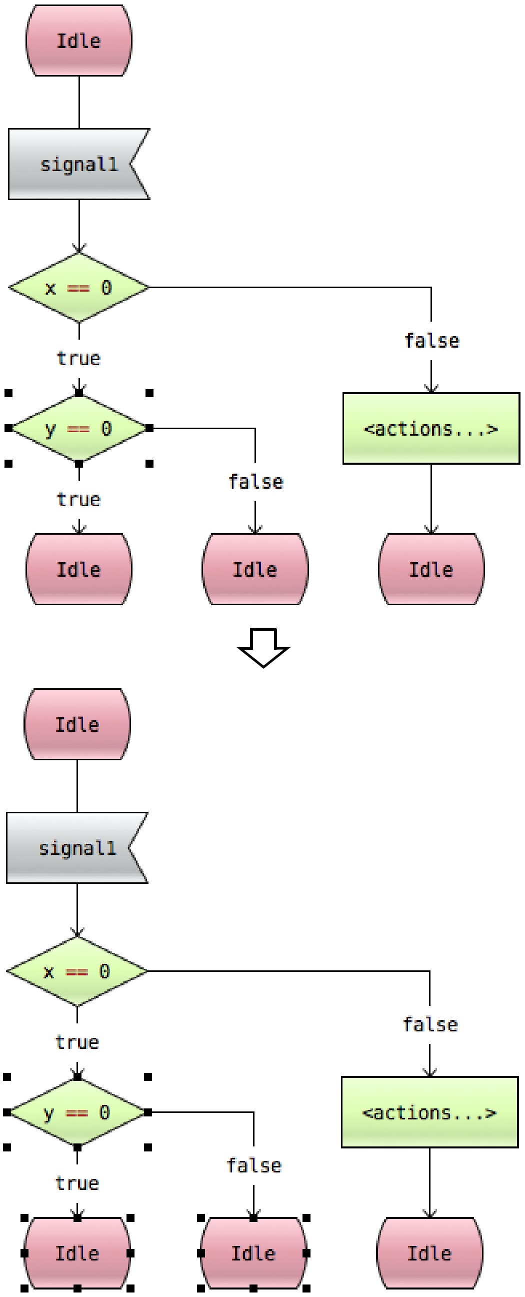


```
printf("Hello world!\n");
```

```
printf("Hello world!\n");
```

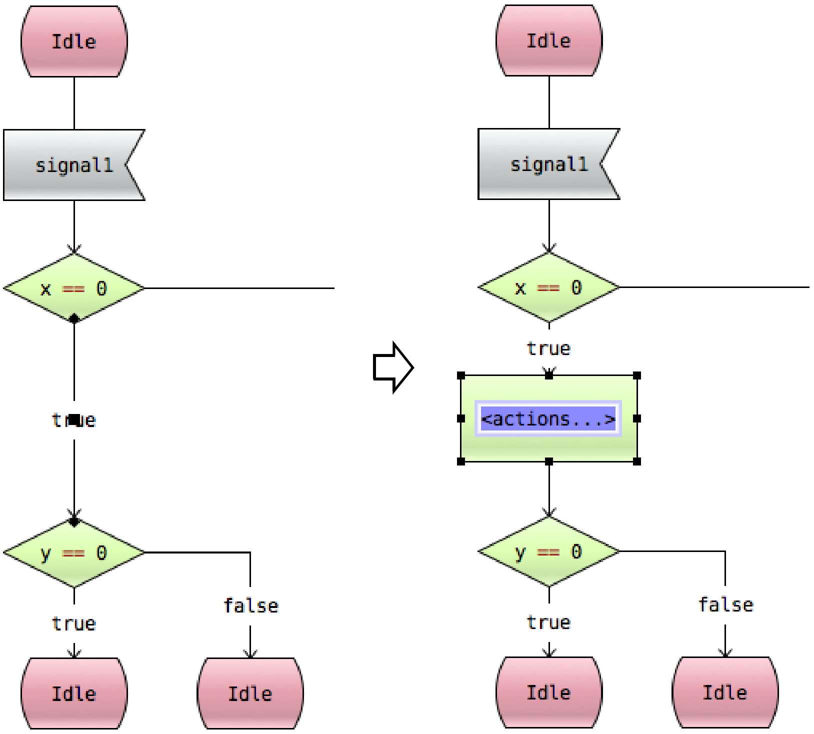
2.3.2.3 Automatic transition selection in legacy mode

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:



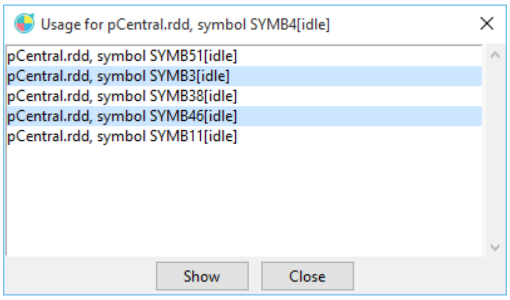
2.3.2.4 "View" / "Go to" menu and state / message browser

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.

2.3.2.5 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:


Usage for pCentral.rdd, symbol SYMB4[idle]

pCentral.rdd, symbol SYMB51[idle]
pCentral.rdd, symbol SYMB3[idle]
pCentral.rdd, symbol SYMB38[idle]
pCentral.rdd, symbol SYMB46[idle]
pCentral.rdd, symbol SYMB11[idle]

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.

2.3.2.6 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:

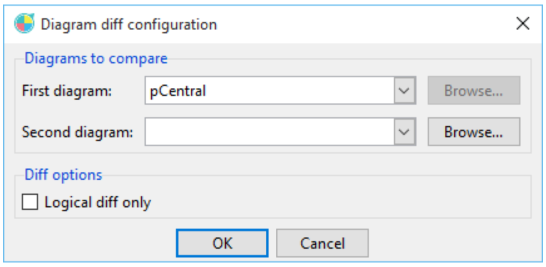


Diagram diff configuration

Diagrams to compare

First diagram: pCentral

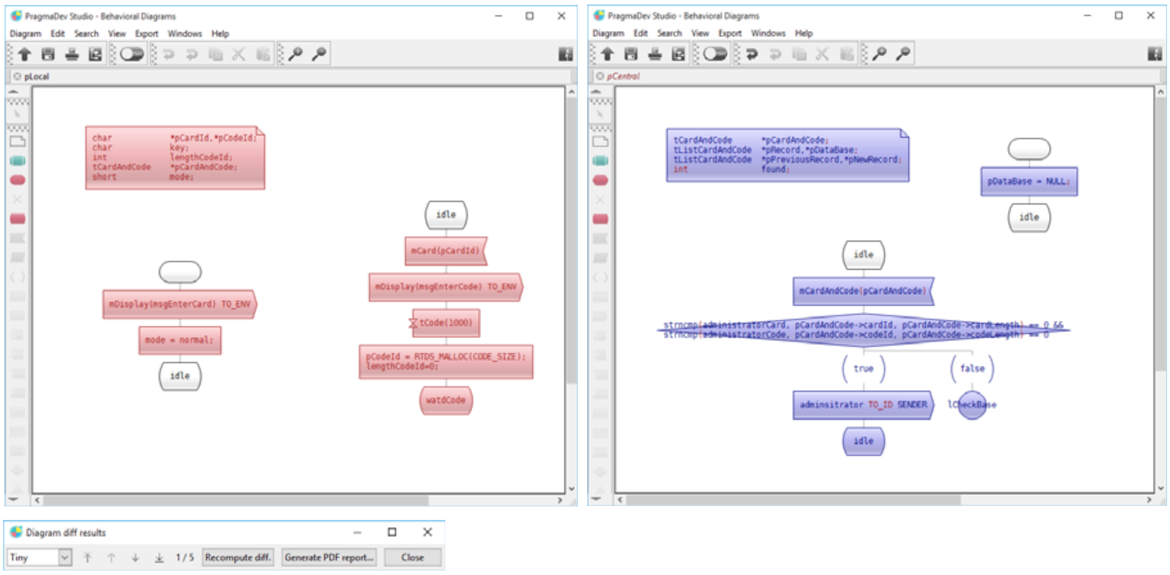
Second diagram:

Diff options

☐ Logical diff only

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.

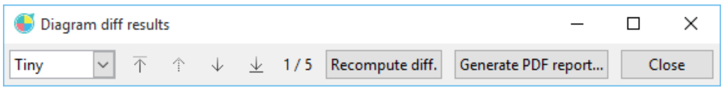
The results for the diff is displayed as follows:


PragmaDev Studio - Behavioral Diagrams
Diagram Edit Search View Export Windows Help
pLocal
pCentral
idle
mCard(pCardId)
mDisplay(msgEnterCode) TO_ENV
tCode(1000)
pCodeId = RTDS_MALLOC(CODE_SIZE); lengthCodeId=0;
waitCode
mDisplay(msgEnterCard) TO_ENV
mode = normal;
mCardAndCode(pCardAndCode)
pDataBase = NULL;
true
false
adminsitrator TO_ID SENDER
idle
Diagram diff results
Tiny
1 / 5
Recompute diff.
Generate PDF report...
Close

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).

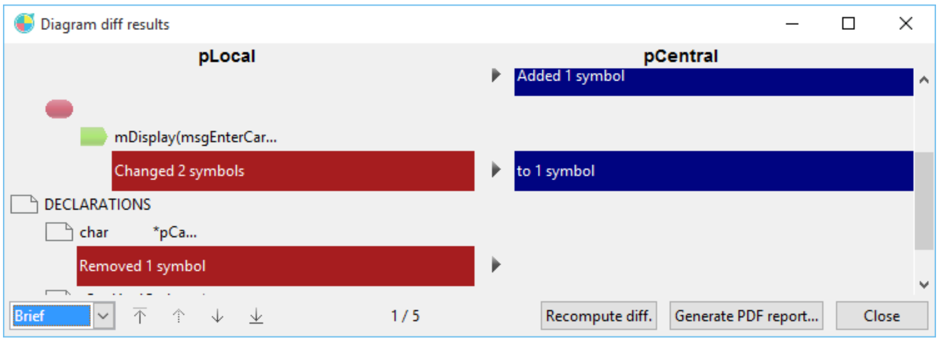
The window below the editors allow to navigate through the differences. This window has 3 possibles layouts:

- 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:



Added 1 symbol

mDisplay(msgEnterCar...

Changed 2 symbols

to 1 symbol

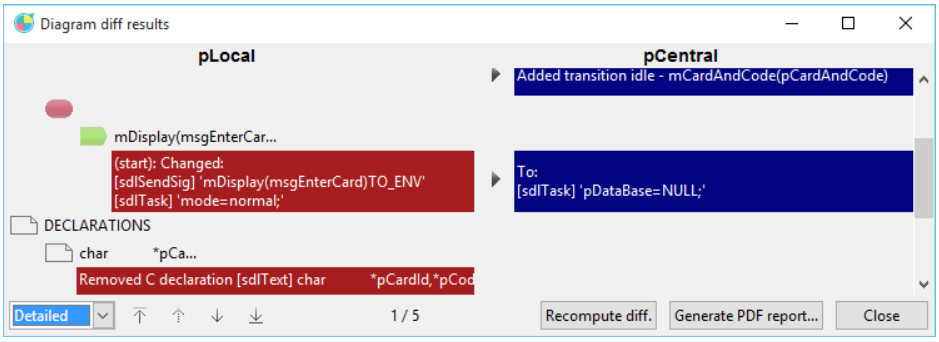
DECLARATIONS

char *pCa...

Removed 1 symbol

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:



Added transition idle - mCardAndCode(pCardAndCode)

(start): Changed:
[sdlSendSig] 'mDisplay(msgEnterCard)TO_ENV'
[sdlTask] 'mode=normal;'

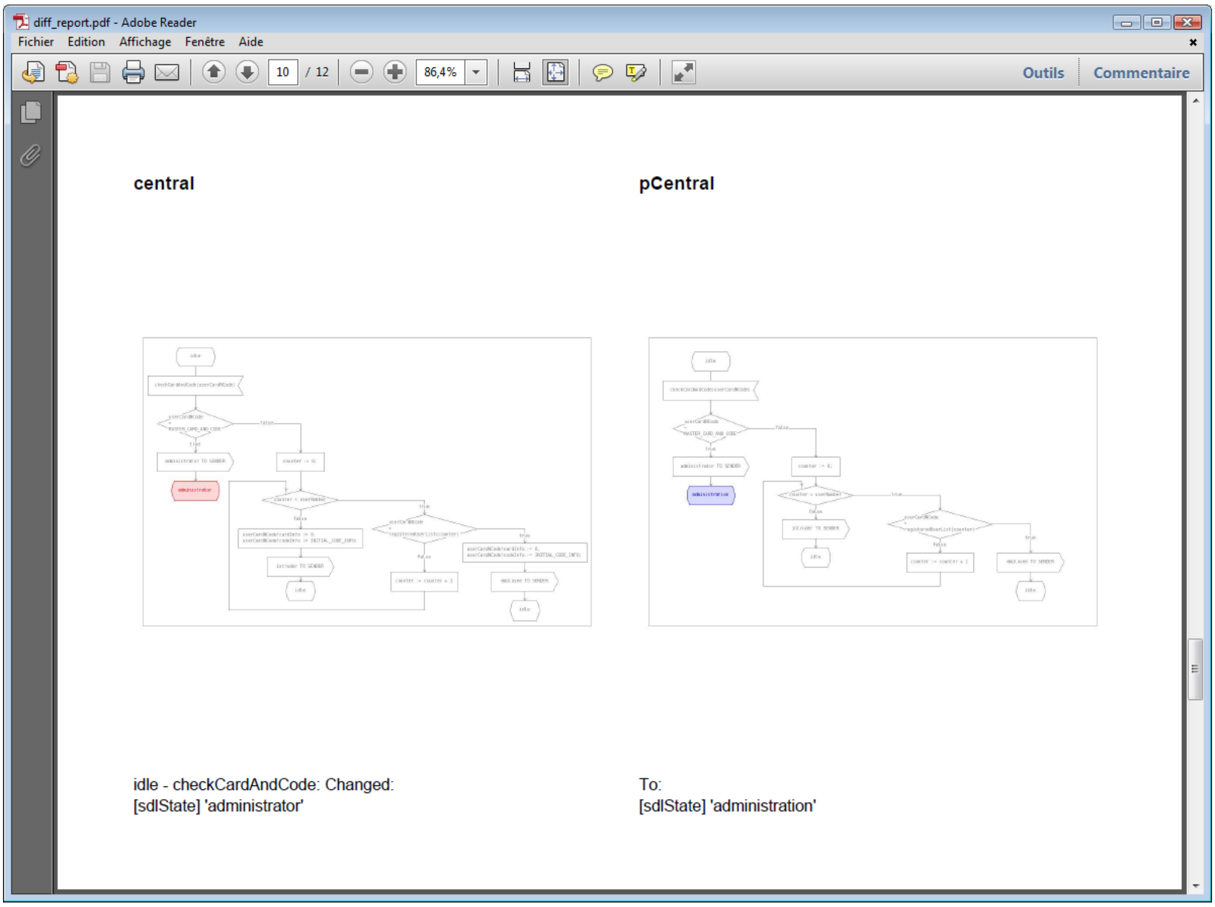
To:
[sdlTask] 'pDataBase=NULL;'

Removed C declaration [sdlText] char *pCardId,*pCod

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:


diff_report.pdf - Adobe Reader
Fichier Edition Affichage Fenêtre Aide
10 / 12
86,4%
Outils
Commentaire
central
pCentral

idle - checkCardAndCode: Changed:
[sdlState] 'administrator'

To:
[sdlState] 'administration'

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.

2.3.3 MSC editor

This kind of editor is used for Message Sequence Charts. A MSC diagram describes a sequence of events happening in a system, with a set of “lifelines” represented as vertical lines, with symbols representing events attached to them.

There are 3 main kinds of MSC diagrams, which are all recognized by PragmaDev Studio:

- Basic MSCs represent a sequence of events that have actually happened during a system execution. They will contain a lifeline for each process or block in the system, and events will be message exchanges, message saves, timer starts, timer resets, etc... They are typically obtained by using the MSC tracing functionality in the simulator or the debugger (see “MSC trace” on page 193 and “MSC trace” on page 315).
- Specification MSCs will contain the same kind of events, but can group them within other symbols with attached semantics. For example, a sequence of events can be isolated in another MSC diagram that will be referenced via a “*MSC reference*” symbol. Or a sequence of events can be enclosed in an “*inline expression*”, allowing to specify this sequence is optional, or can be repeated several times.
- Property Sequence Charts are another kind of specification MSCs that are used to describe “if/then” conditions: *if* a given sequence of events appear in a diagram, *then* another sequence must appear behind it, or must not appear behind it.

The whole format for MSCs - basic & specification - and PSCs is described in “MSC & PSC reference guide” on page 363. Note that there is no specific editor for each kind of diagrams: all symbols are available in the editor, and the kind of diagram is recognized automatically from what it contains.

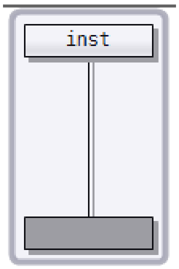
The MSC editor itself is mainly the same as the editor for other SDL diagrams. However, there are a few extra features that may be available depending on the type of MSC diagrams:

- *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).

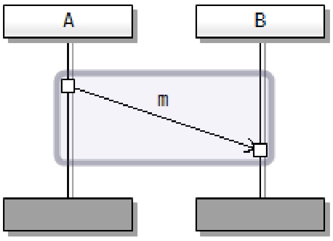
The features of the MSC editor and their availability are described in the following paragraphs.

2.3.3.1 Specific tools

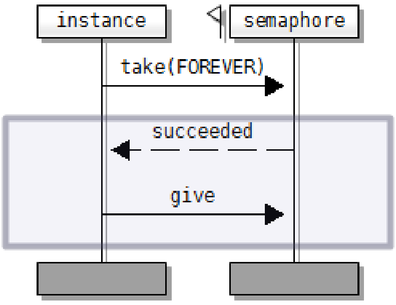
The selection tool in MSC diagrams allows to select symbols and links, just as in other editors. It also allows to select a rectangular zones, that can be copied and pasted and exported as image files:



Selected lifeline



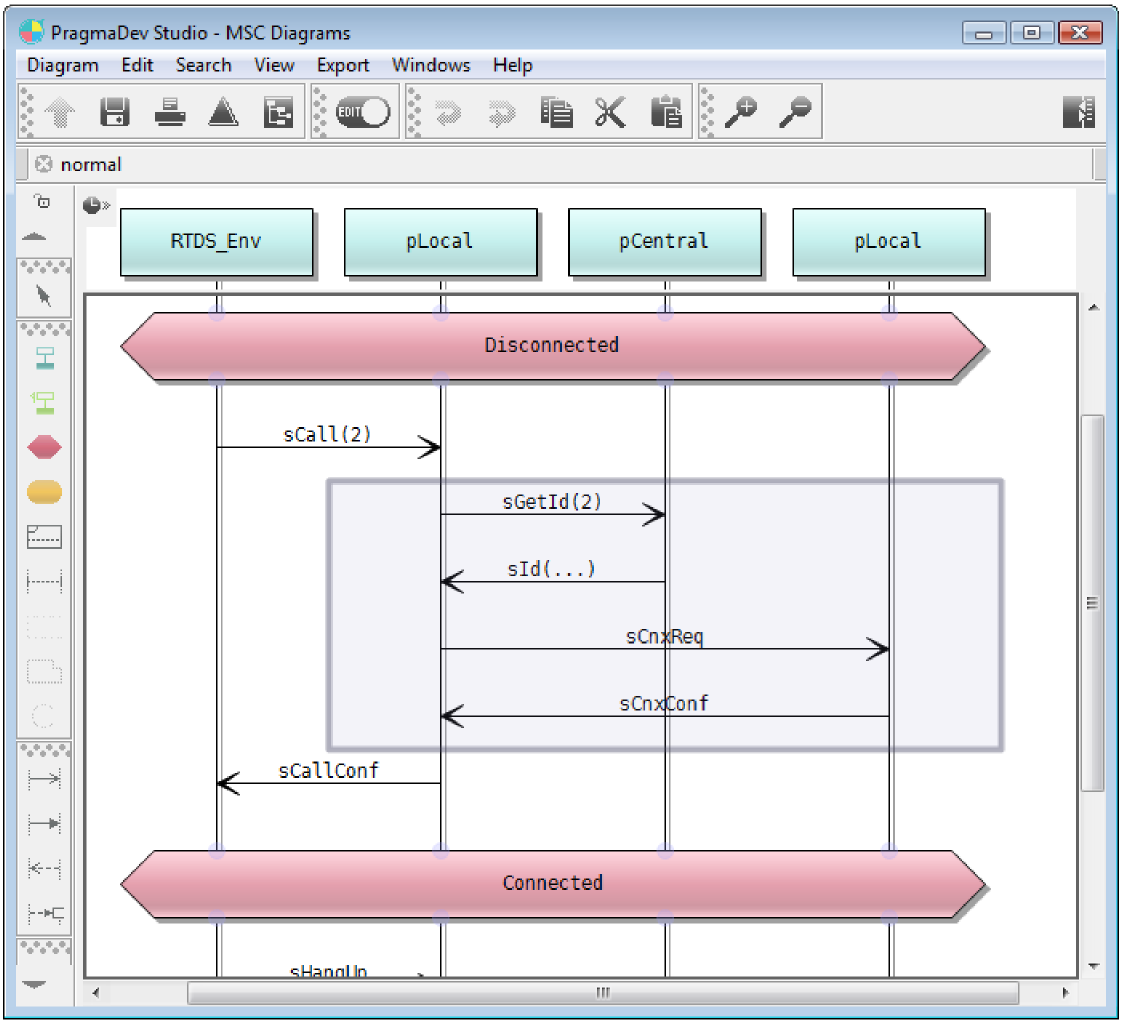
Selected message link



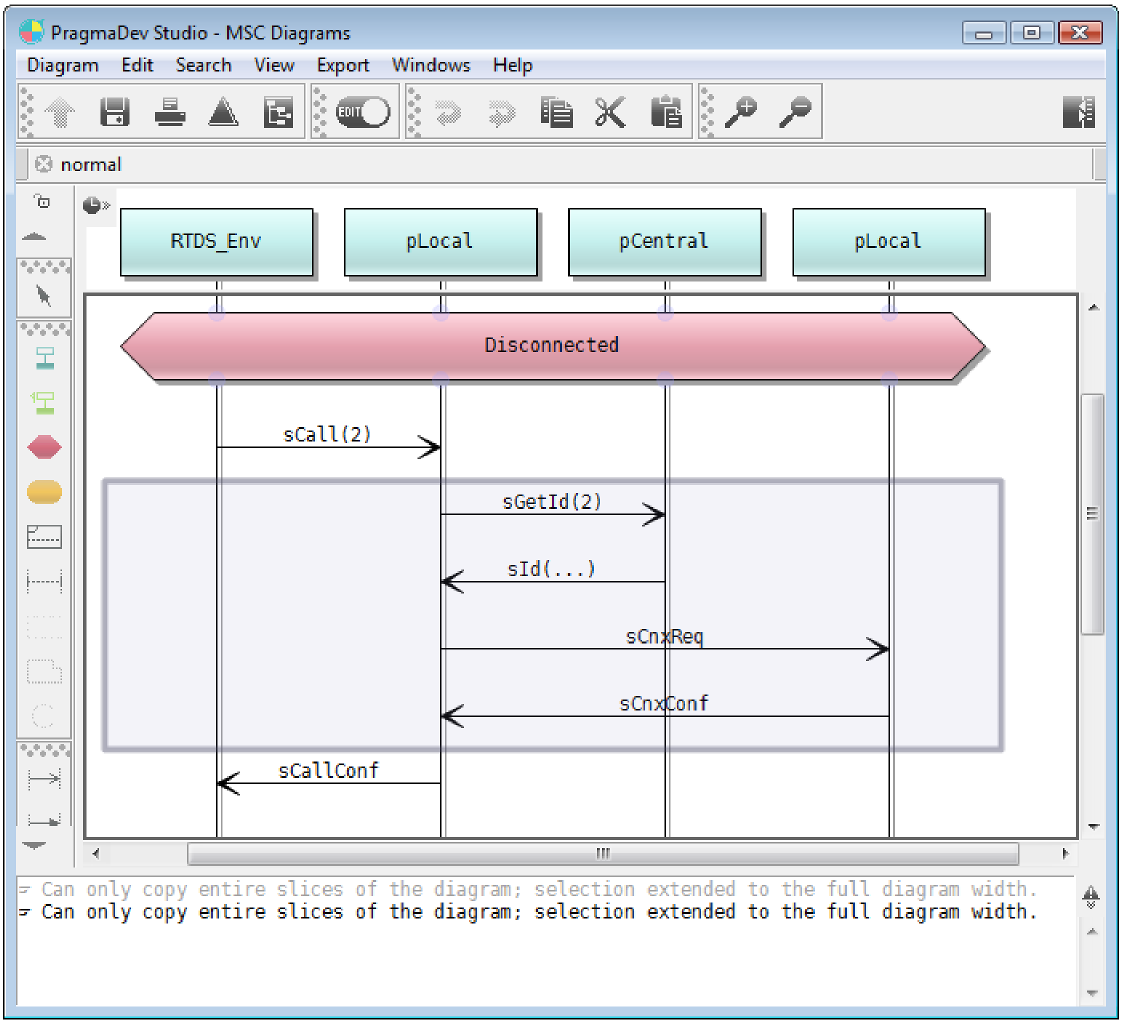
Rectangular selection

Note that copying and pasting rectangular zones will work only for full “horizontal slices” of the diagram: if a rectangular zone is selected, but does not span the full diagram width, it will be automatically extended when copied or cut.

For example, if this zone is selected:

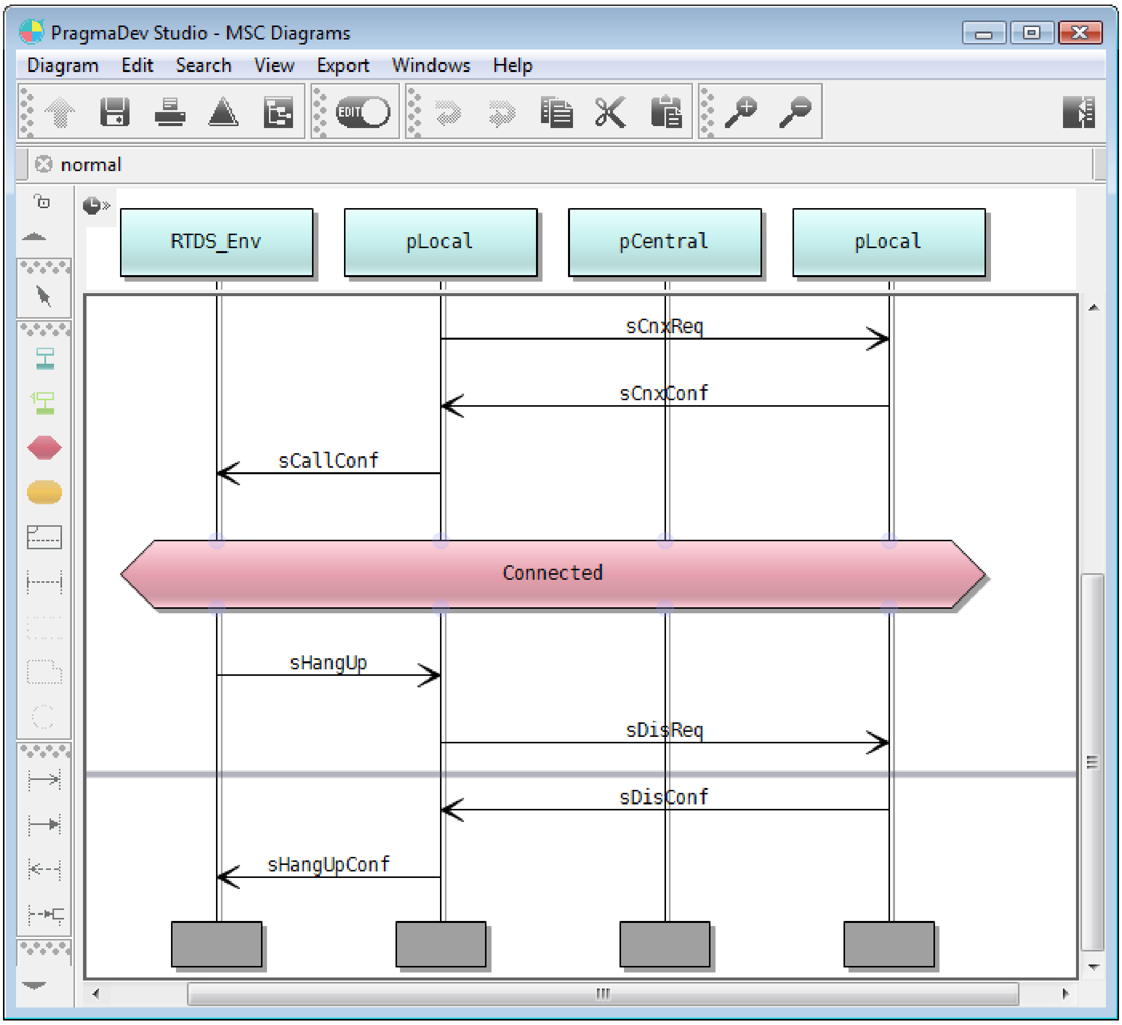


copying it will automatically extend the zone to the full width of the diagram and display a warning:

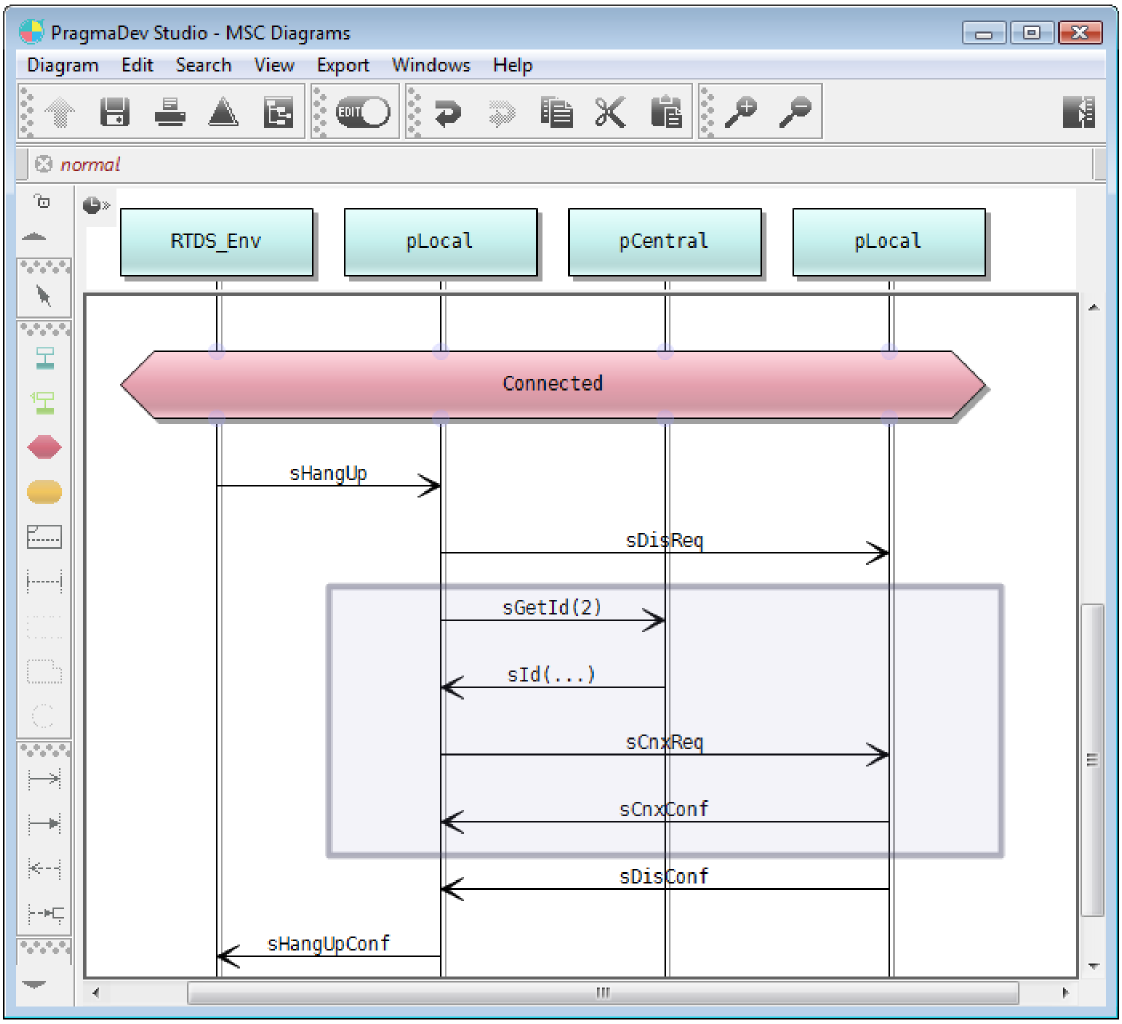


```
Can only copy entire slices of the diagram; selection extended to the full diagram width.
Can only copy entire slices of the diagram; selection extended to the full diagram width.
```

When pasting a rectangular zone, a horizontal insertion line will be displayed:



Clicking in the diagram while the insertion line is displayed will paste the copied zone at this position:



Note that the copy will fail if any object has an end within the slice but the other end outside it, such as a lifeline starting before the slice and ending in it. The paste will fail if one of the copied lifelines does not exist at the paste position.

Rectangular selections will not work in legacy diagrams. Those have 2 other tools in addition to the selection arrow:

- 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.

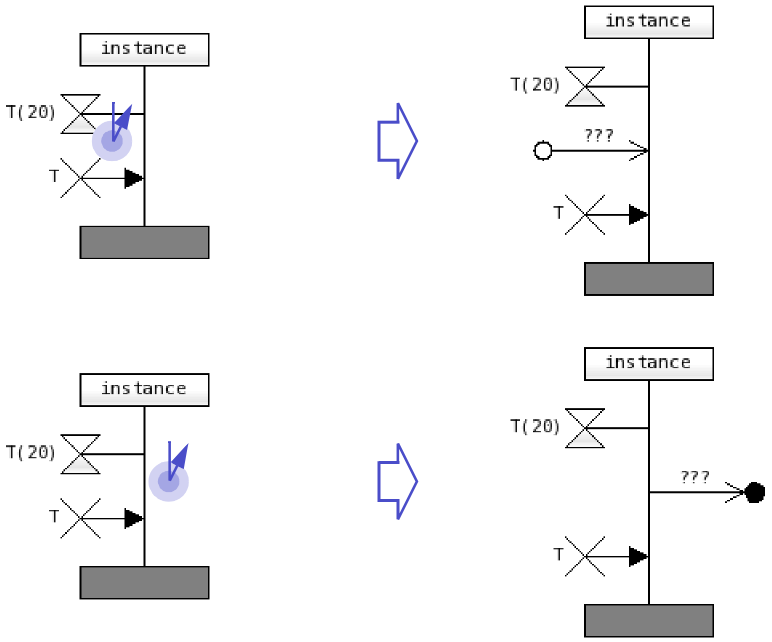
Note: these tools are not available in *normal* MSC diagrams. However:

- 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, but most insertions will actually push down everything below the inserted item, making such a feature unnecessary.

2.3.3.2 Symbol creation

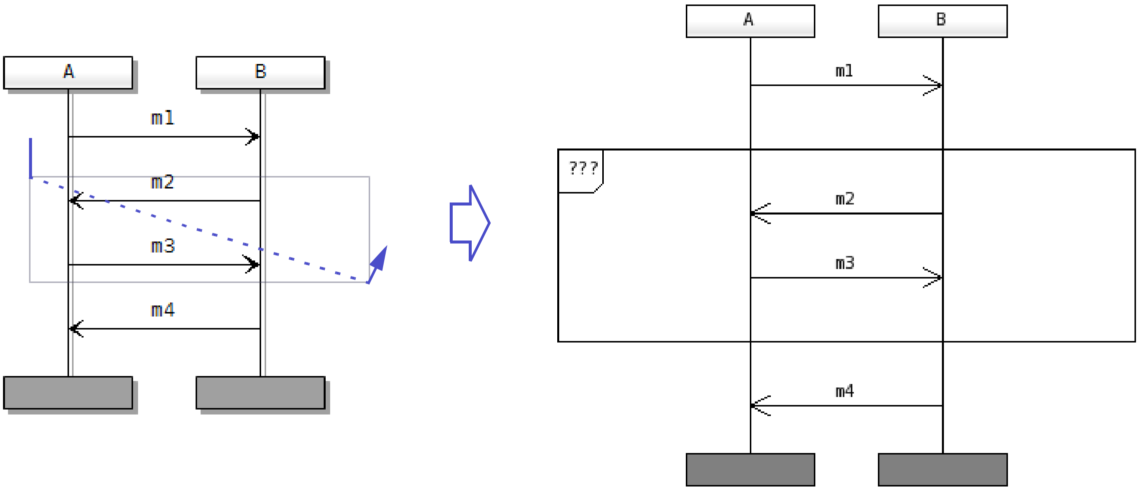
Creating symbols in MSC diagrams works mostly the same way as for other diagrams, as described in “Button and tool bars” on page 65. Some of these tools have a special behavior, mostly because of the nature of the MSC diagrams, which describes mostly a sequence of events, and not individual symbols:

- When creating lifelines, only the horizontal position will be considered: lifelines are always created starting from the top of the diagram and going to the bottom. The only exception are for lifelines created dynamically by another one. For these, an instance creation link must be created, staring from its creator. The lifeline head will then go down to the position where it starts.
- The creation of a lost (resp. found) message is done by selecting the message creation tool and clicking on the right side (resp. left side) of the lifeline sending it (resp. receiving it). For example:

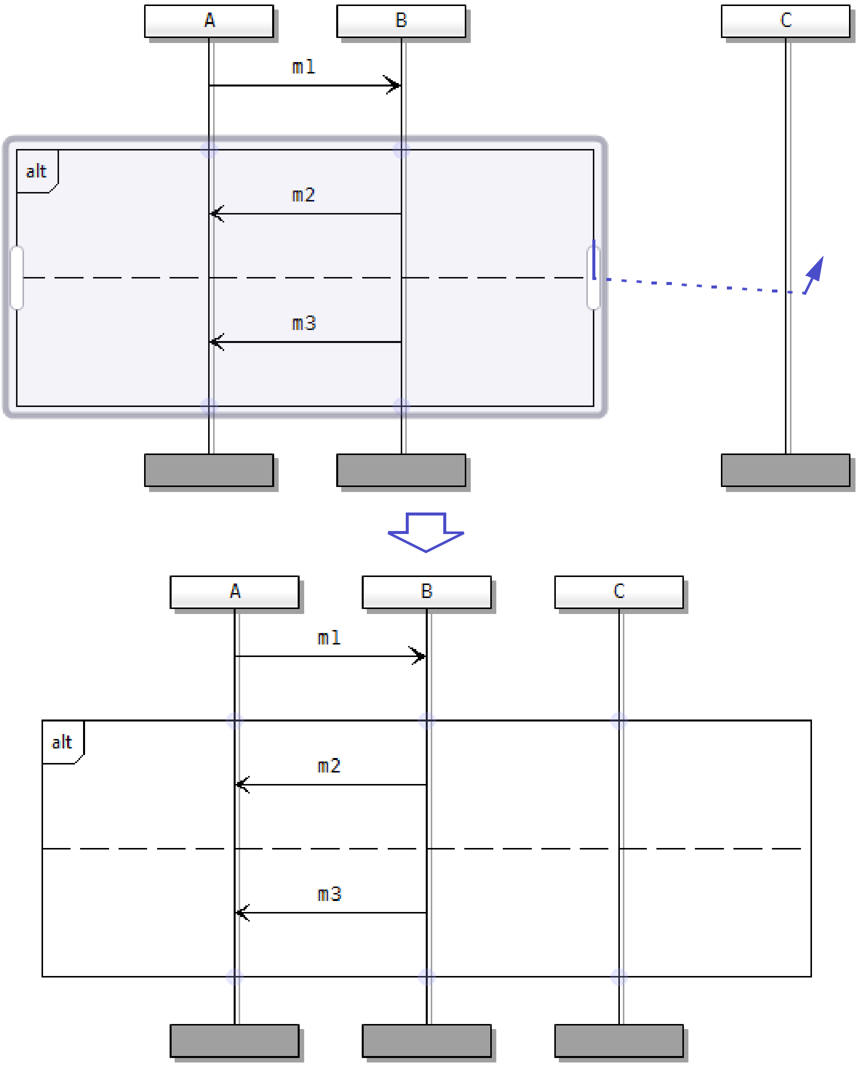


Note that in legacy diagrams, lost and found messages have their own specific symbol that must be created the usualy way, and a message link has to be created between the lost (resp. found) message symbol and its sender (resp. receiver) lifeline.

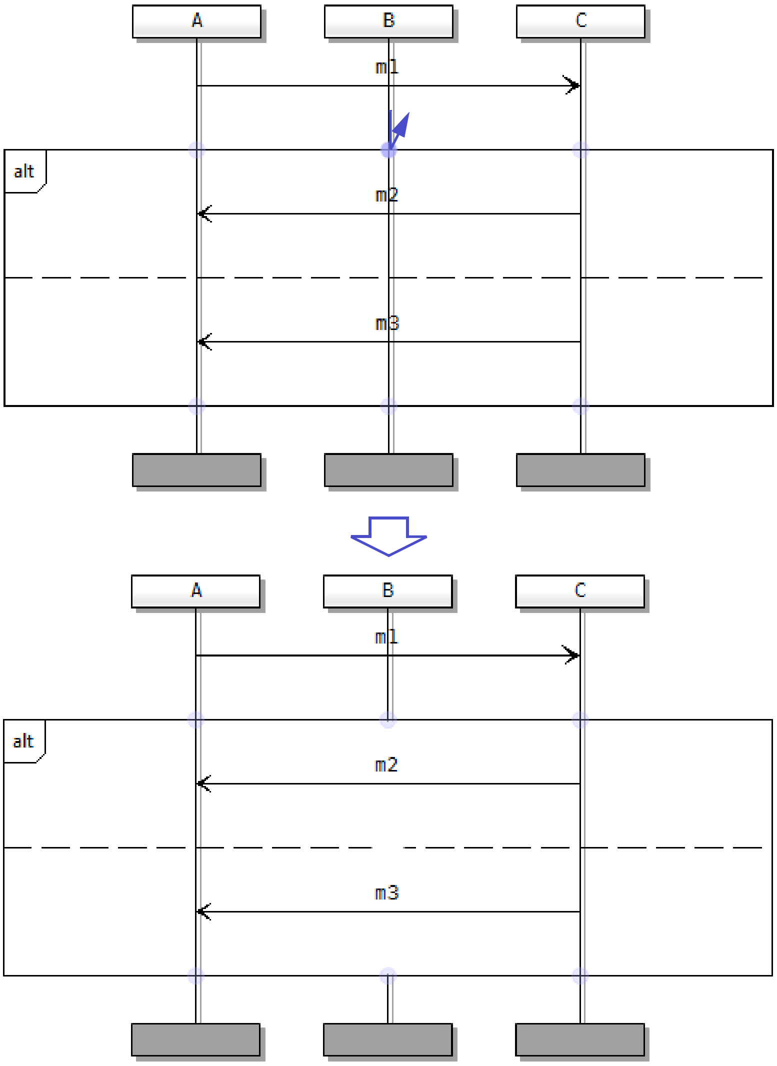
- For conditions, MSC references and inline expressions, they must be created over the lifelines they impact. This is done simply by making them span these lifelines, and optionally all the events that must be included in them:



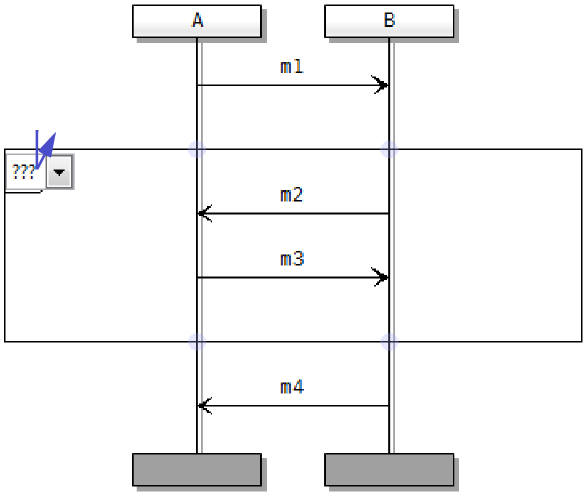
Once created, this symbols can be moved up or down by dragging them, and resized horizontally via the handles appearing on their sides when they are selected, which is the way to make them impact other lifelines than the ones setup at their creation:



Excluding from the symbol a lifeline included in it can be done via the circular handles appearing at the connection points between the symbols and the lifelines:



- For inline expressions, their kind can then be changed by selecting it in the box appearing when the mouse cursor is over its text:

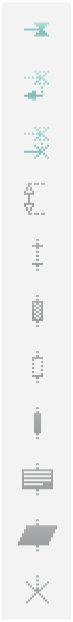


NB: in legacy diagrams, lifelines can be included or excluded from a condition or MSC reference by selecting both and selecting “*[Un]span lifeline(s)*” in the contextual menu (opened via a right-click). However, there is no visual indication about lifelines impacted by a given symbol. The lifelines are always behind the symbol, whether they are impacted or not.

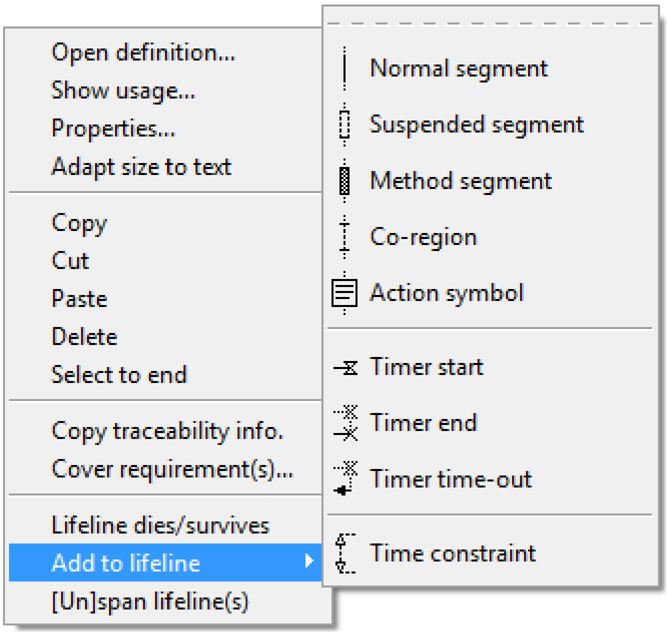
2.3.3.3 Manipulating components in lifelines

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):



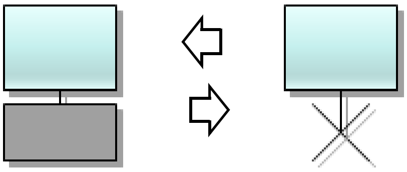
Toolbar for normal diagrams



Context menu for legacy diagrams

The items (button for *normal*, 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 ↖.

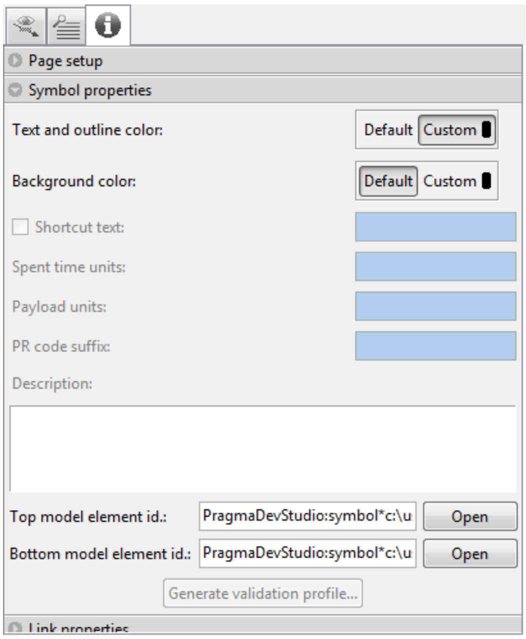
  Note: A *Message save* can be inserted via the ▱ button in *normal* diagrams or ▱ in *legacy* diagrams.

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.

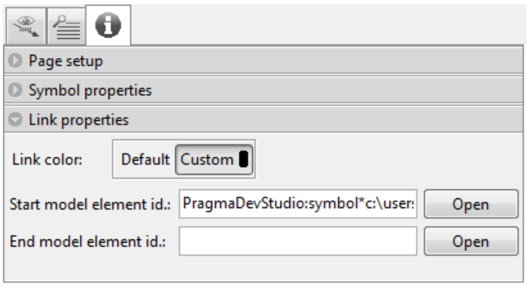
Note: In *normal* diagrams these actions are distinct, i.e., “Delete symbol” and “Delete lifeline” depending on where the contextual menu was triggered.

2.3.3.4 MSC symbol and link properties

Sybmols and links in MSC diagrams mostly use the same property sheet as regular symbols and links, as described in “Symbol and link properties” on page 62. The only addition is the identifiers for the model elements attached to the symbol or link:



Page setup
Symbol properties
Text and outline color: Default [✓ Custom]
Background color: [✓ Default] Custom
□ Shortcut text:
Spent time units:
Payload units:
PR code suffix:
Description:
Top model element id.: PragmaDevStudio:symbol*c:\u Open
Bottom model element id.: PragmaDevStudio:symbol*c:\u Open
Generate validation profile...
Link properties



Page setup
Symbol properties
Link properties
Link color: Default [✓ Custom]
Start model element id.: PragmaDevStudio:symbol*c:\user: Open
End model element id.: Open

These will be recorded automatically if the MSC diagram is a trace from a simulation, or has been created via PragmaDev Tracer commands including the appropriate options (see “Common options and arguments” on page 387). They can also be specified “manually” via the symbol or link properties.

PragmaDev Studio will open itself the model elements that it has recorded in traces. For other ones, the command specified in the preferences will be used (see “Diagram preferences” on page 26).

2.3.3.5 Message parameters display

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

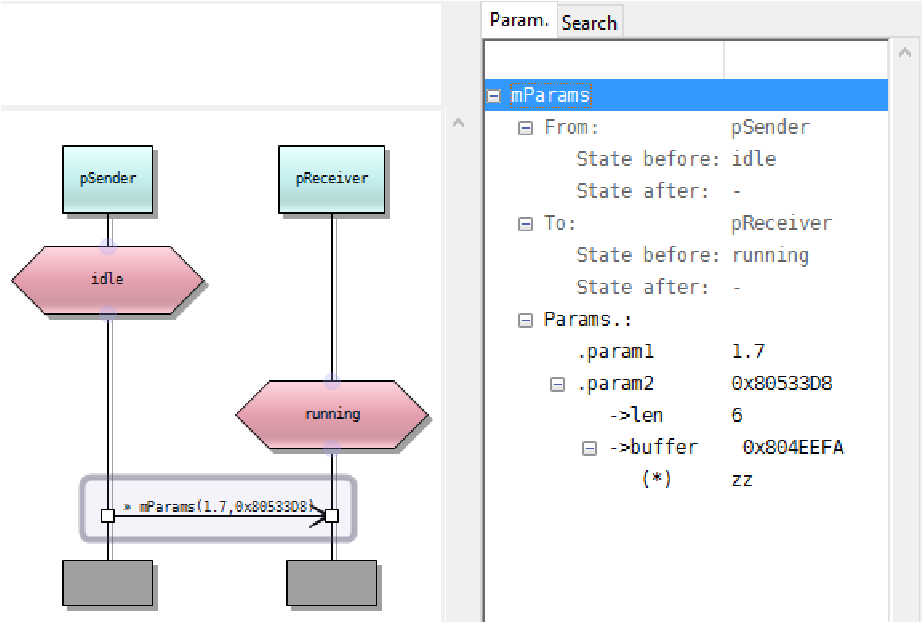
- A visibility set to “Full” displays the full text for the message parameters as it is recorded in the diagram file. The parameters for structured messages are then displayed in the format described in paragraph “Sending SDL messages to the running system” on page 334. This can make the diagram quite 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:

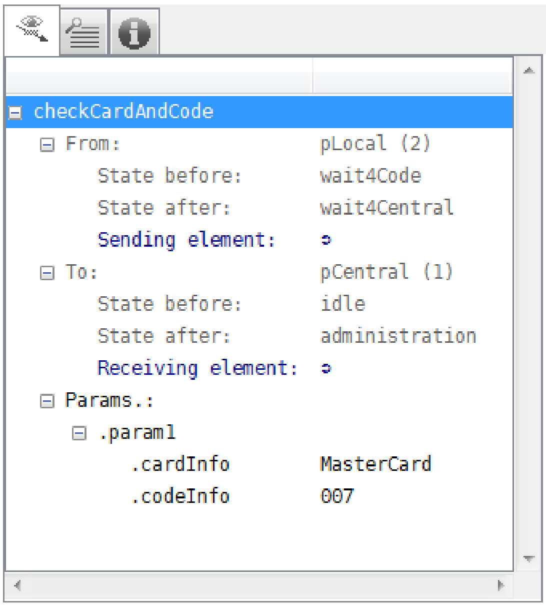


```
⊟ mParams
   ⊟ From:              pSender
        State before:   idle
        State after:    -
   ⊟ To:                pReceiver
        State before:   running
        State after:    -
   ⊟ Params.:
        .param1         1.7
     ⊟ .param2          0x80533D8
          ->len         6
       ⊟ ->buffer        0x804EEFA
             (*)        zz
```

Other information is also displayed in the panel:

- The sender and receiver process;
- The states of the sender and receiver processes before and after they sent or received the message;

- If model element identifiers are specified for the message send or the message receive, their presence will also be notified in the panel:



```
⊟ checkCardAndCode
   ⊟ From:                    pLocal (2)
        State before:         wait4Code
        State after:          wait4Central
        Sending element:      ➲
   ⊟ To:                      pCentral (1)
        State before:         idle
        State after:          administration
        Receiving element:    ➲
   ⊟ Params.:
      ⊟ .param1
           .cardInfo          MasterCard
           .codeInfo          007
```

Double-clicking on the “*Sending element*” or “*Receiving element*” tree node will open the corresponding element, either with PragmaDev Studio if it’s an element it can handle, or via the command to open external model element specified in the preferences (see “Diagram preferences” on page 26).

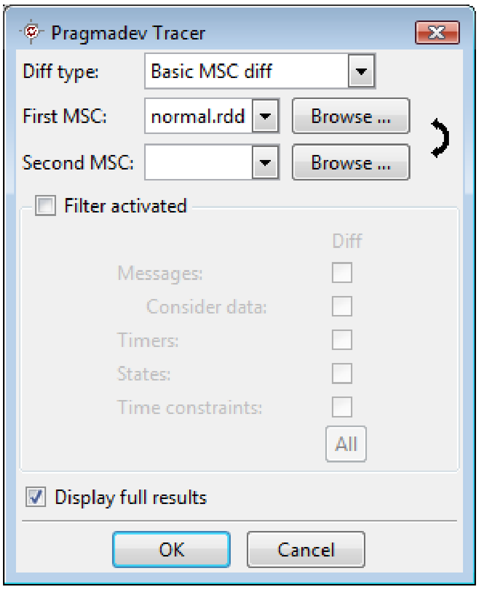
2.3.3.6 Conformance checking: diagram diff & property match

PragmaDev Studio offers 3 levels of conformance checking:

- A MSC trace can be compared to another MSC trace, used as a reference. This can typically be used for regression testing, the reference trace giving the wanted behavior, and being compared to a newly obtained trace.
  In this kind of comparison, all events in both diagrams are compared one by one without any interpretation of any kind. This is mainly intended for trace comparisons, but it also works on other diagram kinds, as items normally only present in specification or PSC diagrams are taken into account too, e.g inline expressions or relative time constraints.
- A MSC trace can be compared to a specification diagram. For this comparison, the semantics in the specification is taken into account. For example, if there is an ‘`opt`’ inline expression in the specification containing a sequence of message exchanges, the comparison will interpret it, and consider that the diagrams are matching if the sequence is there, or if it is not there at all.
  This allows to describe expected scenarios in a powerful way via specification MSC diagrams and match the execution traces against them later.
- Occurrences of a property described in a PSC diagram can be found in a MSC trace. In this case, the semantics are considered in the PSC diagram, as well as the PSC specific elements. Note that this is different from a specification vs. trace comparison, as properties describe a small part of a scenario that can actually match several times in a trace. MSC specification diagrams describe a whole scenario, and will be matched entirely on the trace.
  Properties are a good and powerful way to specify wanted and unwanted behavior in the designed system.

2.3.3.6.1 Basic MSC diff: trace vs. trace, spec. vs. spec., ...

The basic MSC diff just compare two diagrams events by events and reports the found differences. This kind of comparison is launched by selecting ‘Compare diagrams...’ in the ‘Diagram’ menu, or by clicking the △ button in the toolbar. The following dialog then appears:



Pragmadev Tracer

Diff type: Basic MSC diff

First MSC: normal.rdd

Second MSC:

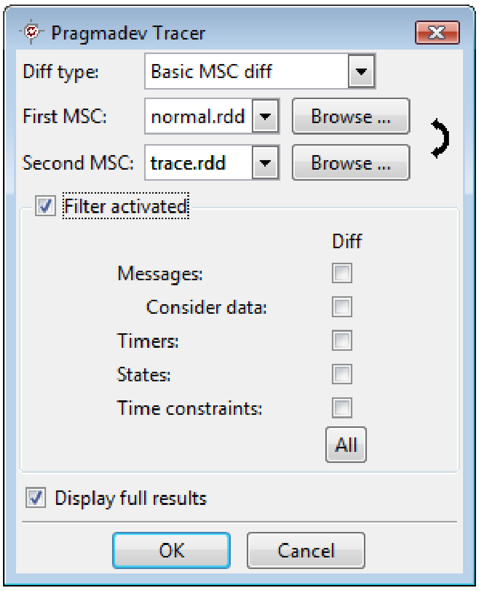
□ Filter activated

	Diff
Messages:	□
Consider data:	□
Timers:	□
States:	□
Time constraints:	□

☑ Display full results

Selecting the basic MSC diff is done by selecting the corresponding value in the ‘Diff type’ field. The name for first MSC will be automatically set to the name of the currently displayed diagram. For the MSC to compare, it can be either selected in the list attached to the ‘Second MSC’ field, or loaded from a file via its ‘Browse...’ button. Once selected, the arrow in the right part of the dialog allows to exchange the two MSCs if the comparison must be done the opposite way.

PragmaDev Studio allows to exclude some elements from the comparison based on their type. This is done by checking the ‘Filter activated’ option:



Pragmadev Tracer

Diff type: Basic MSC diff

First MSC: normal.rdd

Second MSC: trace.rdd

☑ Filter activated

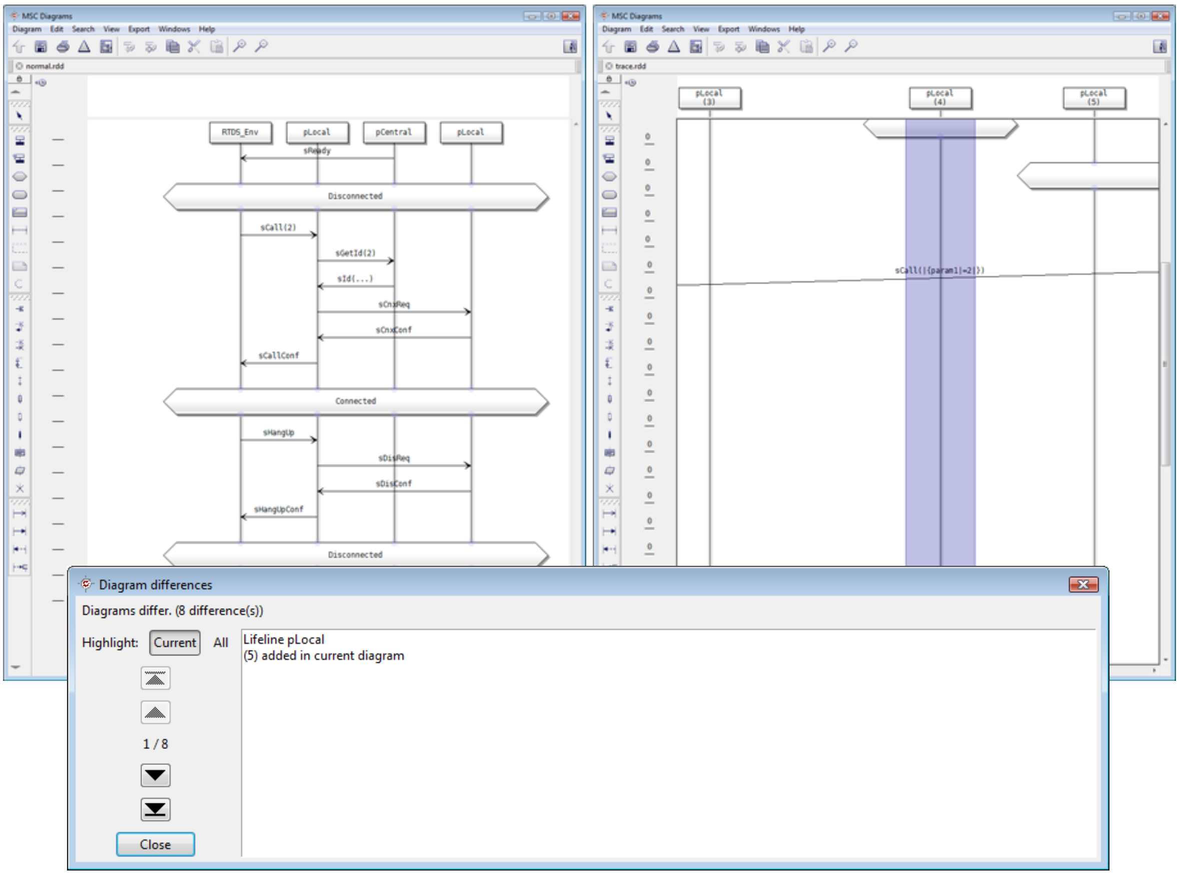
	Diff
Messages:	□
Consider data:	□
Timers:	□
States:	□
Time constraints:	□

☑ Display full results

All the shown element types can be included or excluded from the comparison. The ‘All’ button will check all the boxes if any of them is unchecked, and uncheck them if all are checked. The ‘Consider data’ option allows to compare messages without looking at their actual parameters. If the option is unchecked, only the names of the message are compared and nothing else, even if it is present.

The option ‘Display full results’ at the bottom of the dialog allows to display only a summary of the comparison results instead of the full set of differences. To display the summary, just uncheck the box.

If this option is checked and after validating the dialog, PragmaDev Studio puts each diagram in its own window and displays them side by side. A dialog also appears at the bottom of the screen, allowing to browse through the found differences:



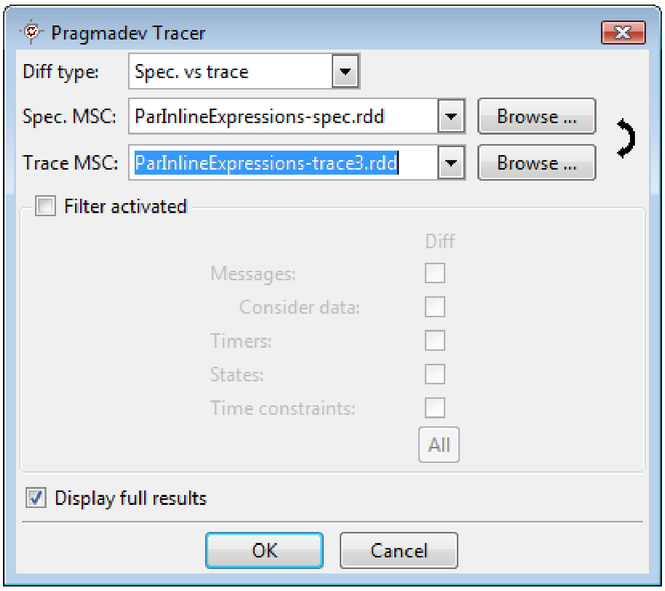
Diagrams differ. (8 difference(s))

Lifeline pLocal
(5) added in current diagram

A summary of the differences is displayed at the top. Each difference will be highlighted in red in the diagram displayed on the left, and in blue in the diagram displayed on the right. The text in the dialog gives a short description of the identified difference. The arrows allow to browse through the differences. The option ‘Highlight’ allows to highlight all differences in both diagrams to get a quick view of what differs without having to browse through all the differences.

2.3.3.6.2 Spec vs. trace comparison

Comparing a specification diagram to an actual trace is done the same way as for a basic MSC diff, except the diff type has to be set to 'Spec. vs. trace' in the dialog:



Pragmadev Tracer

Diff type: Spec. vs trace

Spec. MSC: ParInlineExpressions-spec.rdd

Trace MSC: ParInlineExpressions-trace3.rdd

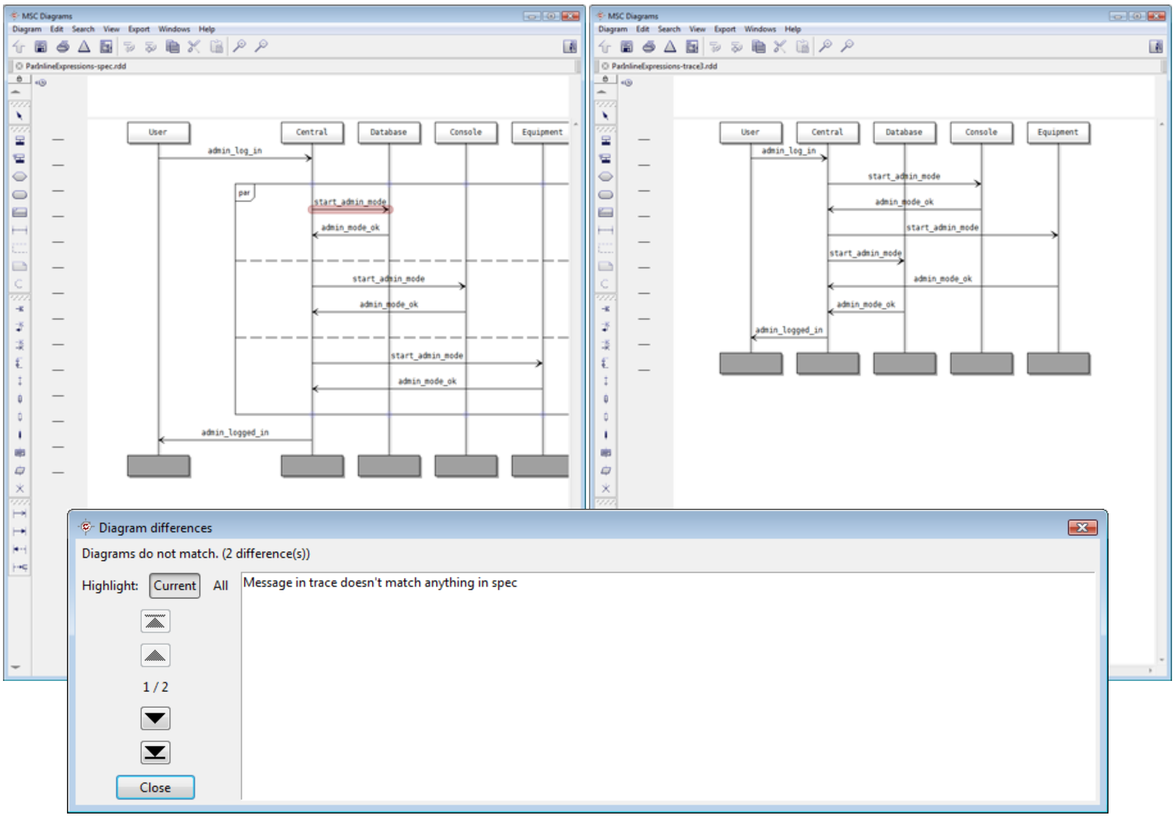
□ Filter activated

	Diff
Messages:	□
Consider data:	□
Timers:	□
States:	□
Time constraints:	□

☑ Display full results

Note also that the specification diagram must be the one specified in the field 'Spec MSC' in the dialog, which is always the first one. If needed, the diagrams can be swapped by using the arrow button on the dialog's right side. The same filters are provided as for a basic MSC diff.

Once validated, the found differences are displayed in the same way as for a basic MSC diff; only the way to perform the comparison changes, as semantics in the specification is taken into account where it wouldn't be in a basic MSC diff:

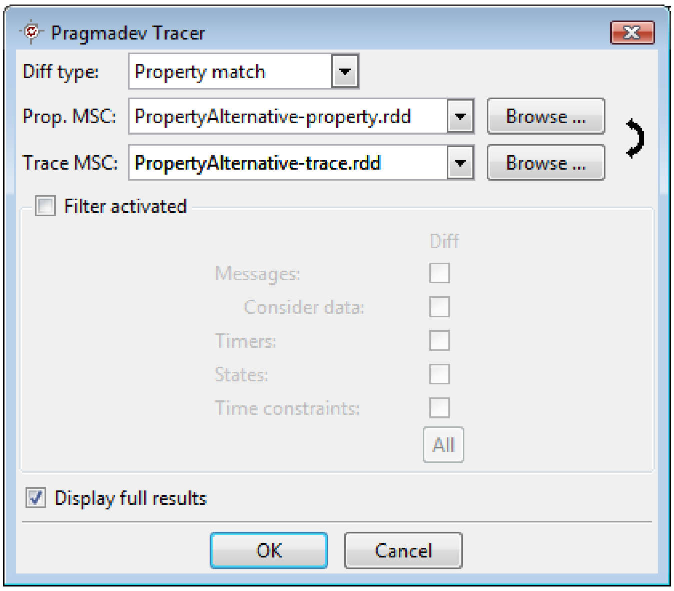


Diagrams do not match. (2 difference(s))

Message in trace doesn't match anything in spec

2.3.3.6.3 Property match

Matching a PSC diagram against a MSC trace is done the same way as for the other kinds of comparisons, except the diff type has to be set to 'Property match':



Diff type: Property match

Prop. MSC: PropertyAlternative-property.rdd

Trace MSC: PropertyAlternative-trace.rdd

□ Filter activated

Messages: □

Consider data: □

Timers: □

States: □

Time constraints: □

☑ Display full results

Note that the PSC diagram has to be the one specified in the 'Prop. MSC' field, which is always the first one. If needed, the diagrams can be swapped with the arrow button on the right side of the dialog. The same comparison options are provided as for basic MSC and specification vs. trace comparisons, but they are less significant here, as a property diagram is always partial.

Once validated, the property matches and violations are displayed in a similar way to the display of differences in the other kinds of comparisons. Mostly the colors and the difference descriptions differ: each matched element in the property or the MSC diagram will be displayed as green, and each unmatched one as red. The difference description will be:

- ‘Property match’ if the property matches:

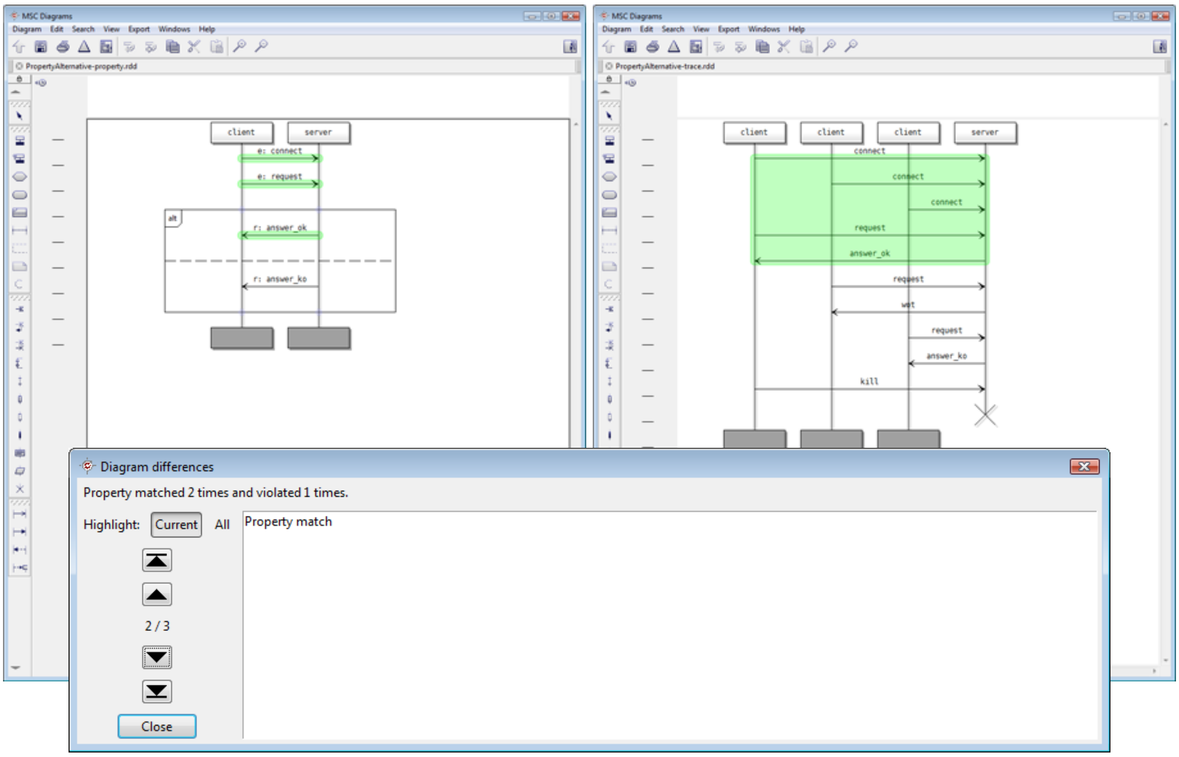


Diagram differences

Property matched 2 times and violated 1 times.

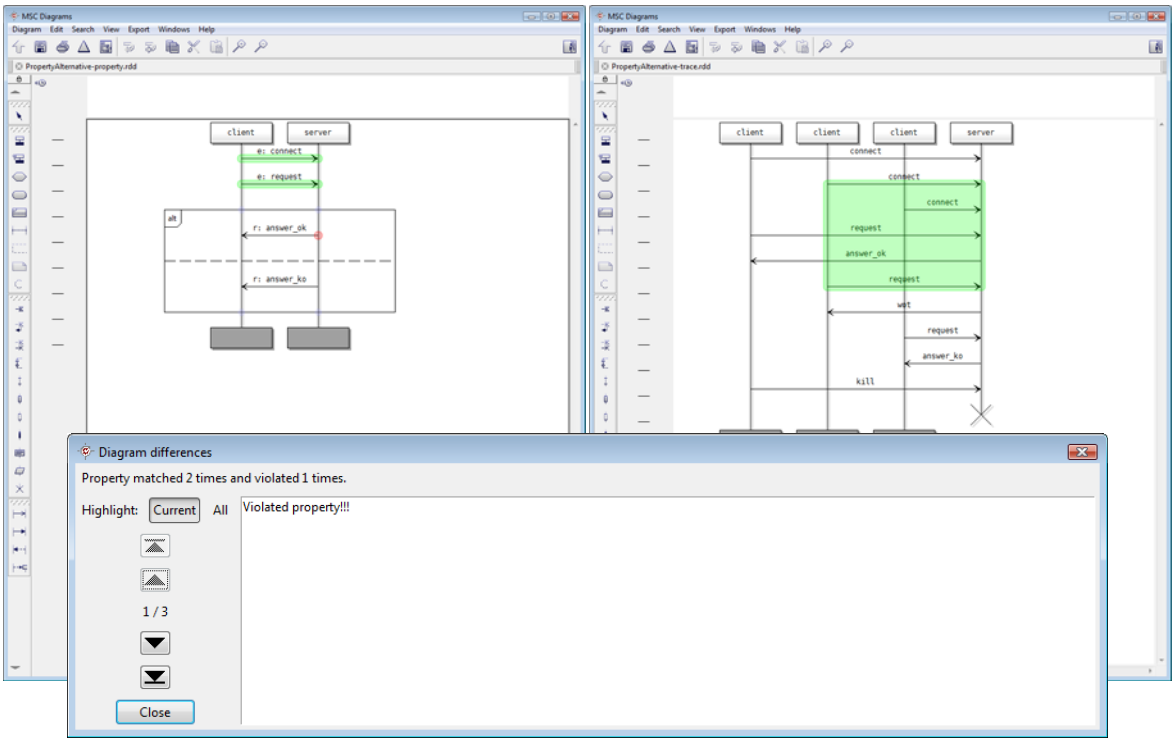
Highlight: Current All

Property match

2 / 3

Close

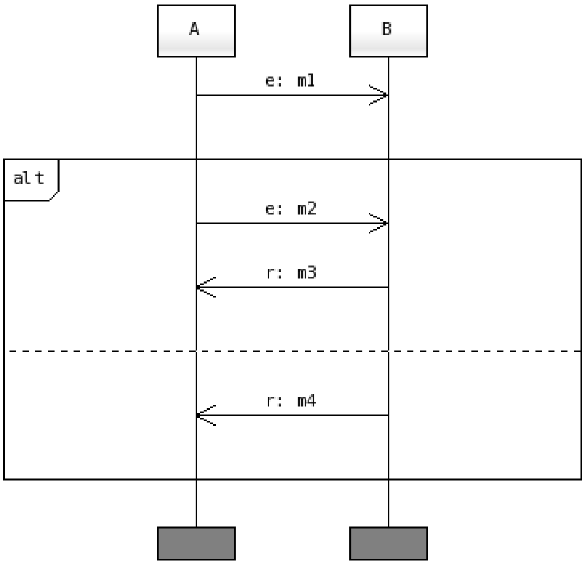
- 'Violated property!' if the property does not match:



Property matched 2 times and violated 1 times.

Violated property!!!

- 'Possibly violated property' in some very specific cases where it is impossible to tell if the property is matched or not. A typical example where this case happens is the following:



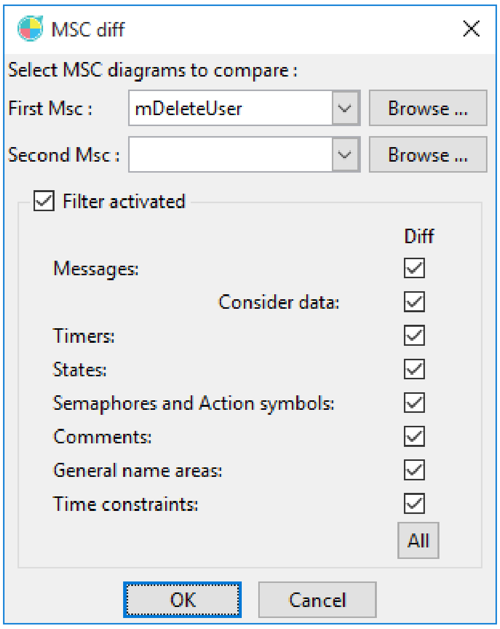
If the trace contains a message m1 from A to B, followed neither by a message m2 from A to B, nor by a message m4 from B to A, there's no way to know which part of the alternative should have matched. But if it was the first part, the message m2 is not there, so the property doesn't apply, and if it was the second one, the required message m4 is not there either, so the property is violated. In this case, a possible property violation will be reported.

Note that a property is not necessarily violated if something doesn't match in it. Typically, an unmatched fail message means the property is matched.

2.3.3.6.4 Legacy diagram diff

The comparison of legacy MSC diagrams is done mostly the same way as for other diagrams, but the results are displayed in quite a different way.

To run a comparison on two legacy MSC diagrams, use the "*Compare with other diagram...*" entry in the editor's "*Diagram*" menu. The following dialog appears:



MSC diff

Select MSC diagrams to compare :

First Msc : mDeleteUser

Second Msc :

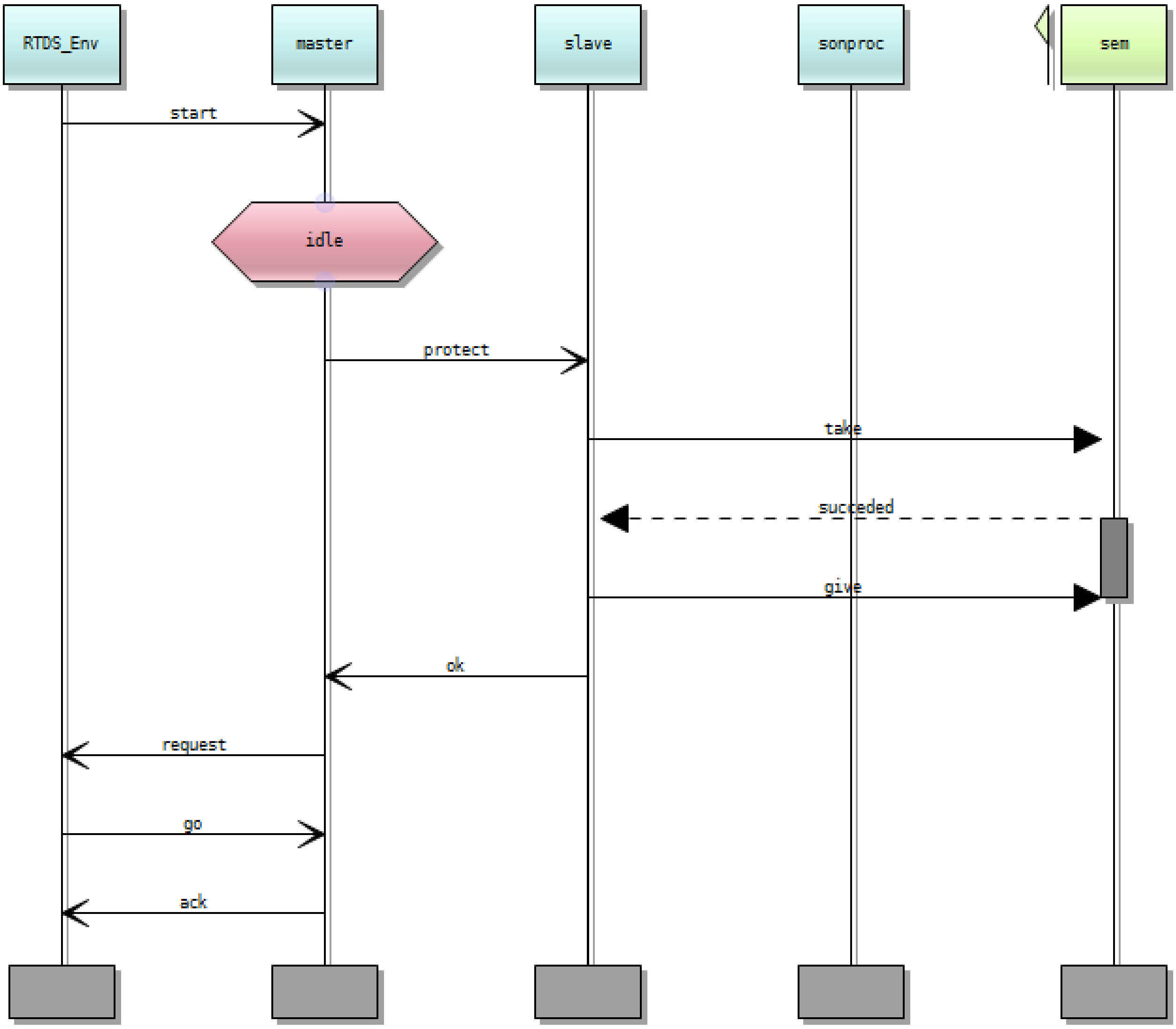
☑ Filter activated

	Diff
Messages:	☑
Consider data:	☑
Timers:	☑
States:	☑
Semaphores and Action symbols:	☑
Comments:	☑
General name areas:	☑
Time constraints:	☑

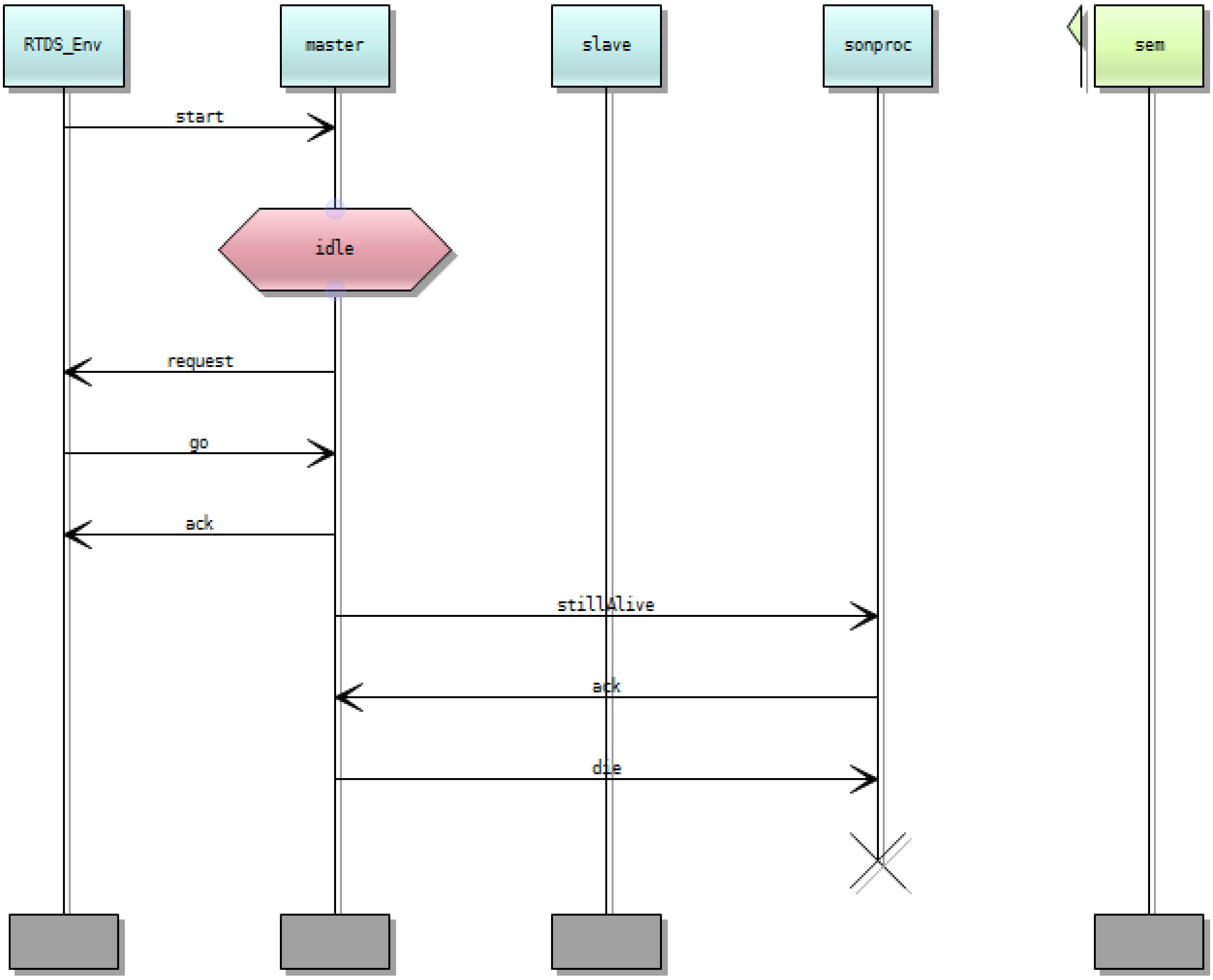
Here the two MSC diagrams can be selected from the current project or from the file system. Filters can also be applied on the result of the MSC comparison.

Once validated, the result of the comparison will appear 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.

For example, if the comparison is made on the two following diagrams:



First MSC diagram in comparison

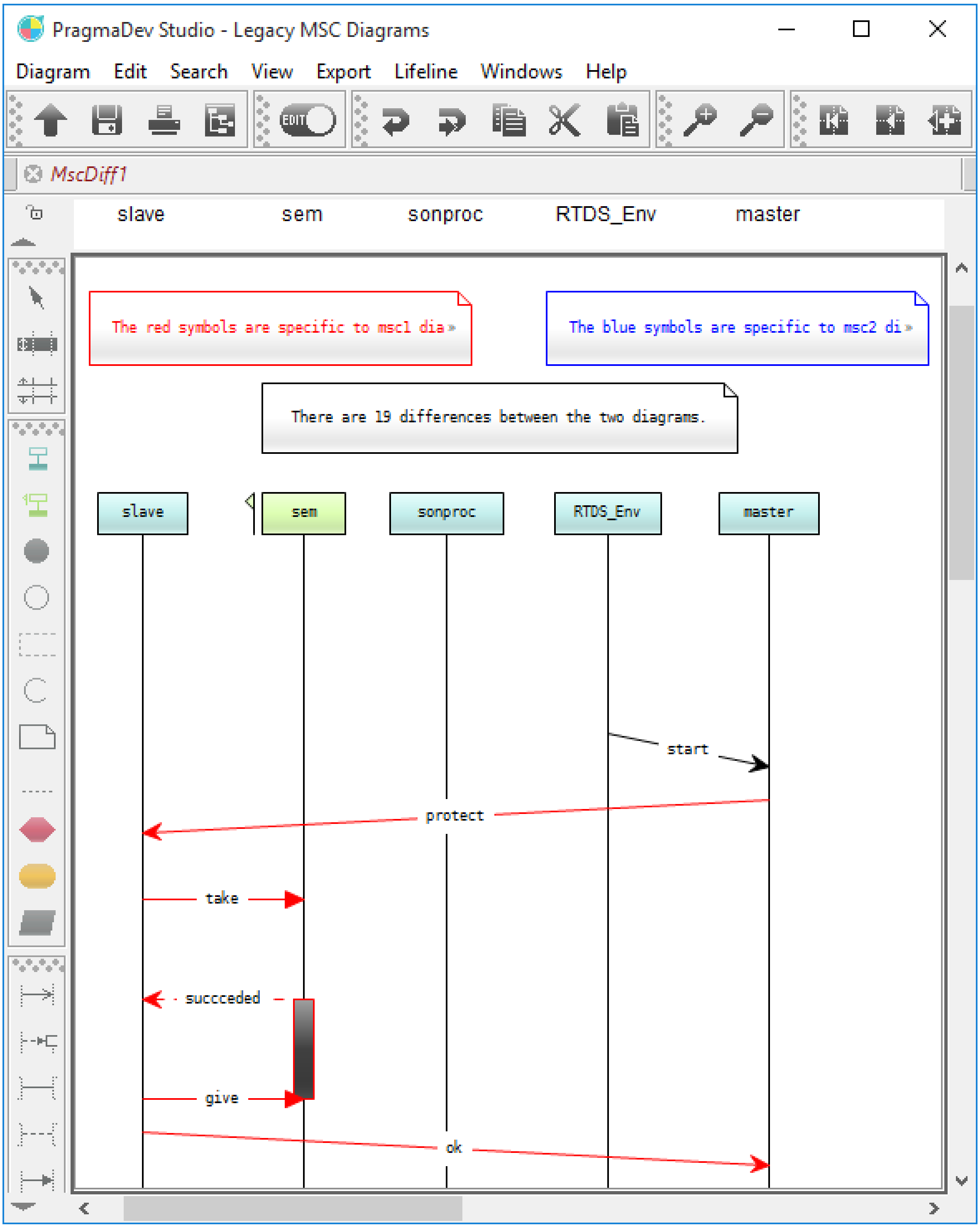


Second MSC diagram in comparison

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.

The results of the comparison will be display as follows:



PragmaDev Studio - Legacy MSC Diagrams
Diagram Edit Search View Export Lifeline Windows Help
EDIT
MscDiff1
slave sem sonproc RTDS_Env master
slave
sem
sonproc
RTDS_Env
master
start
protect
take
succceded
give
ok

The red symbols are specific to msc1 dia»

The blue symbols are specific to msc2 di»

There are 19 differences between the two diagrams.

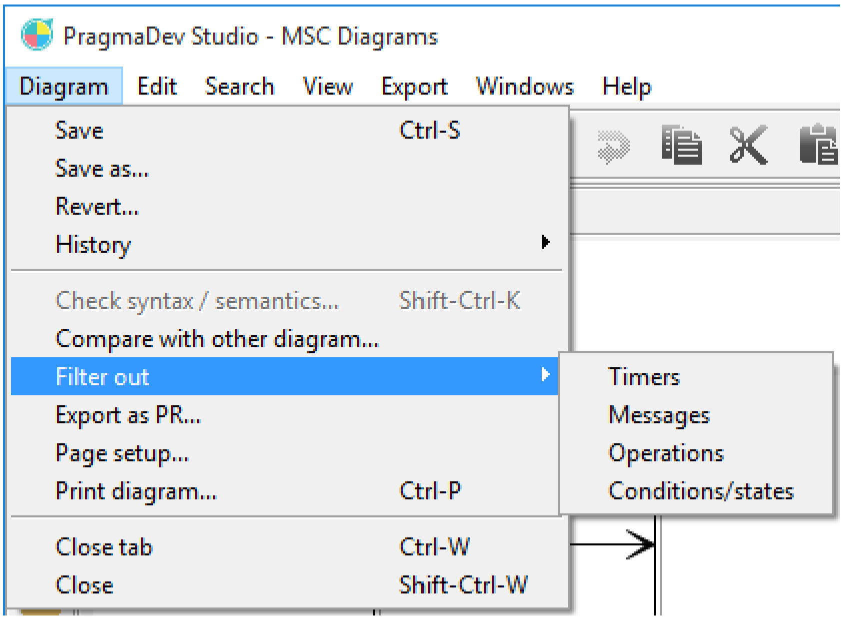
The differences are identified by colors: in the comparison results diagram, the events specific to the first diagram are represented in red, and the events specific to the second diagram are represented in blue.

2.3.3.7 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.

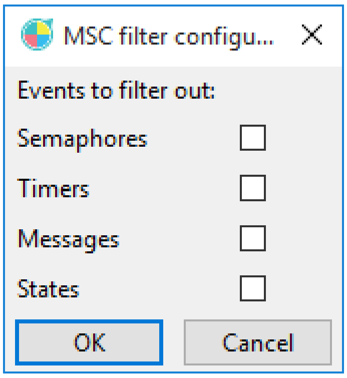
This functionality is accessed via the “Diagram” menu:

- 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.

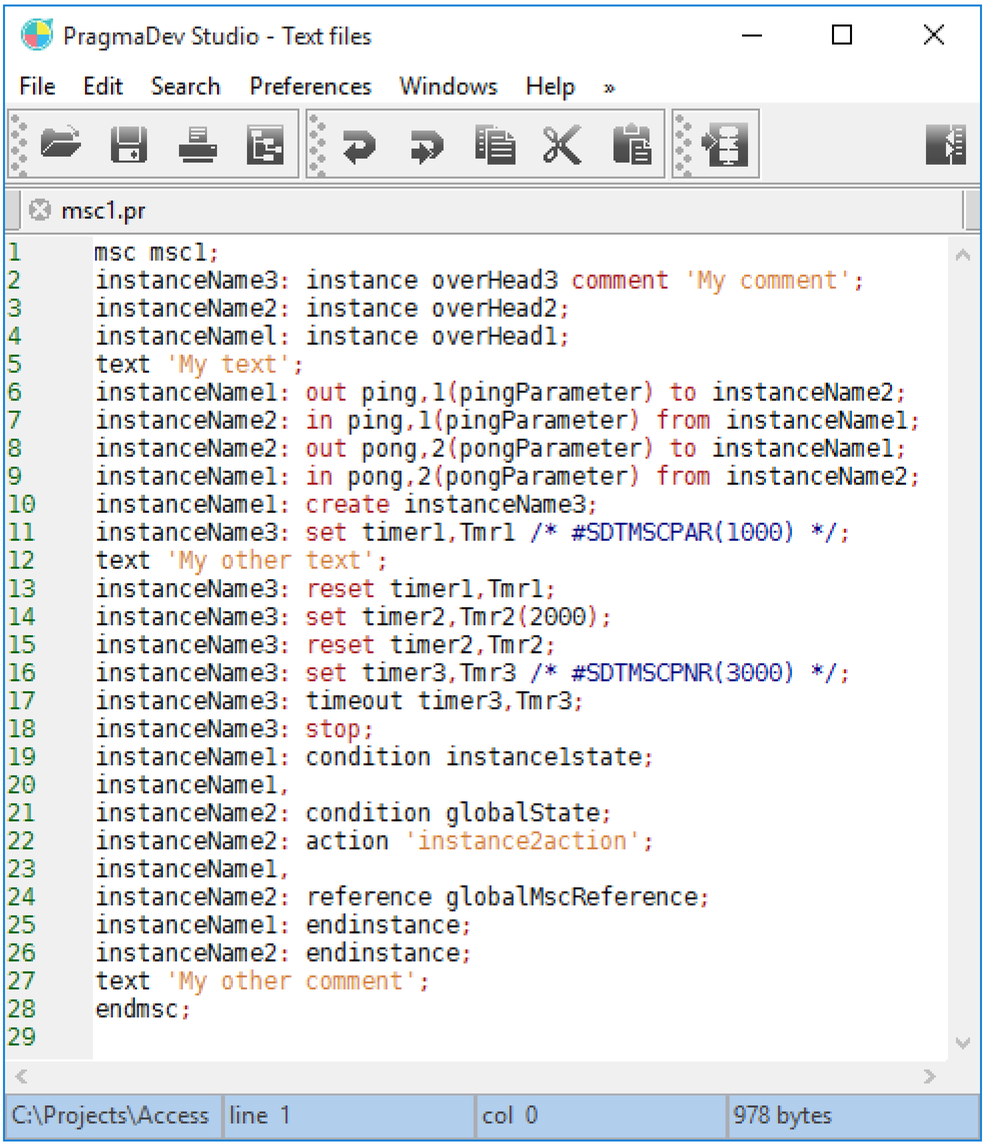
2.3.3.8 Lifeline collapsing and expanding

As filtering allows to hide specific information, it is sometimes needed to hide what’s happening between specific lifelines. For example, several instances can be tasks in the same entity, and it can be practical to see what's happening to this entity rather than to each individual instance.

PragmaDev Tracer allows this kind of operation via *lifeline collapsing*: several lifelines can be collapsed in a single one, all events happening on these lifelines disappearing, as well as the events happening between the collapsed lifelines. An example of collapsing is given in paragraph “Collapsed lifelines” on page 374. To get the second diagram, the lifelines B and C must be selected, then collapsed via the ‘Collapse lifelines’ operation in the contextual menu opened by a right click.

2.3.3.9 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.

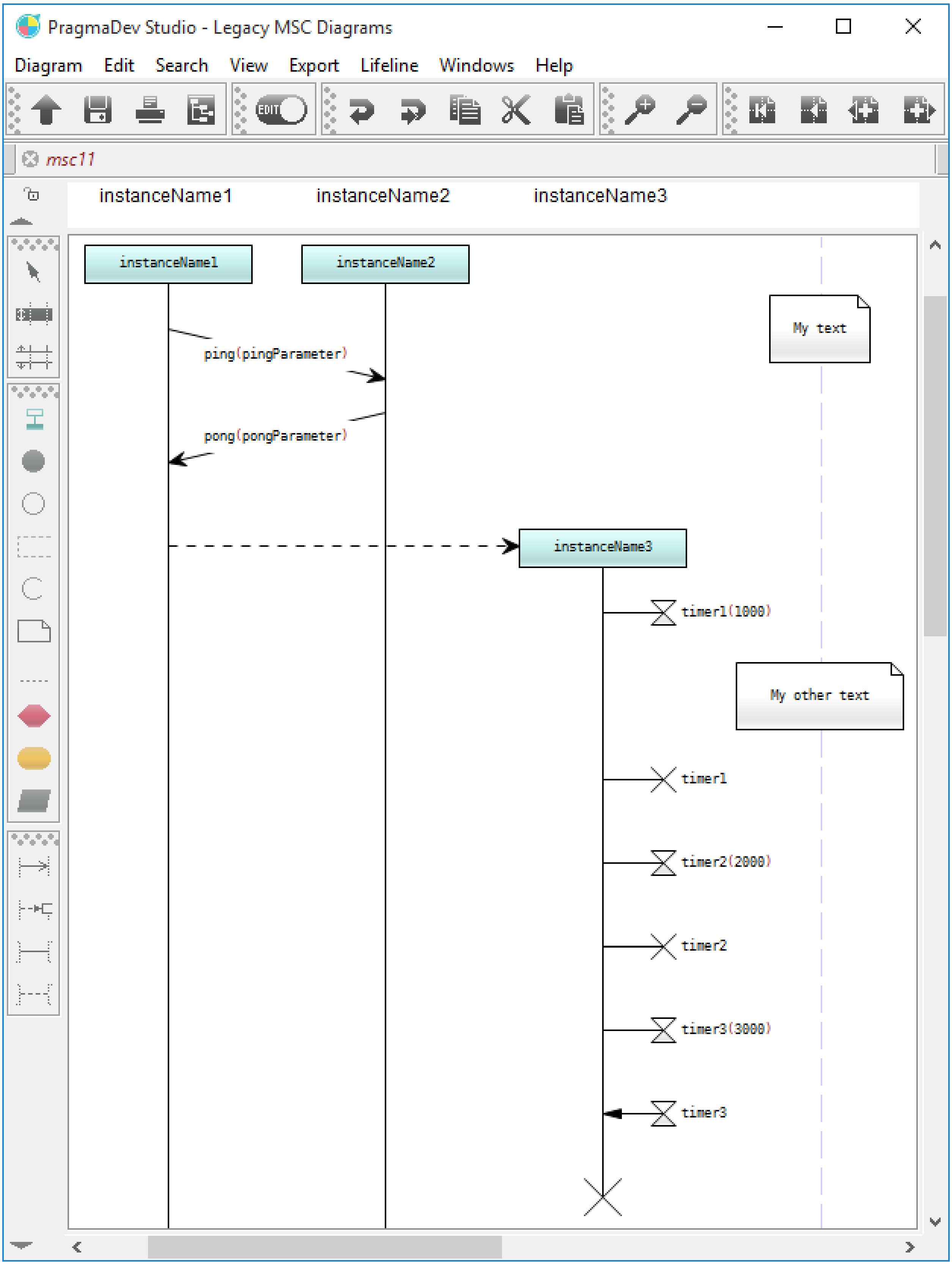


```
msc msc1;
instanceName3: instance overHead3 comment 'My comment';
instanceName2: instance overHead2;
instanceName1: instance overHead1;
text 'My text';
instanceName1: out ping,1(pingParameter) to instanceName2;
instanceName2: in ping,1(pingParameter) from instanceName1;
instanceName2: out pong,2(pongParameter) to instanceName1;
instanceName1: in pong,2(pongParameter) from instanceName2;
instanceName1: create instanceName3;
instanceName3: set timer1,Tmr1 /* #SDTMSCPAR(1000) */;
text 'My other text';
instanceName3: reset timer1,Tmr1;
instanceName3: set timer2,Tmr2(2000);
instanceName3: reset timer2,Tmr2;
instanceName3: set timer3,Tmr3 /* #SDTMSCPNR(3000) */;
instanceName3: timeout timer3,Tmr3;
instanceName3: stop;
instanceName1: condition instancelstate;
instanceName1,
instanceName2: condition globalState;
instanceName2: action 'instance2action';
instanceName1,
instanceName2: reference globalMscReference;
instanceName1: endinstance;
instanceName2: endinstance;
text 'My other comment';
endmsc;
```

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:

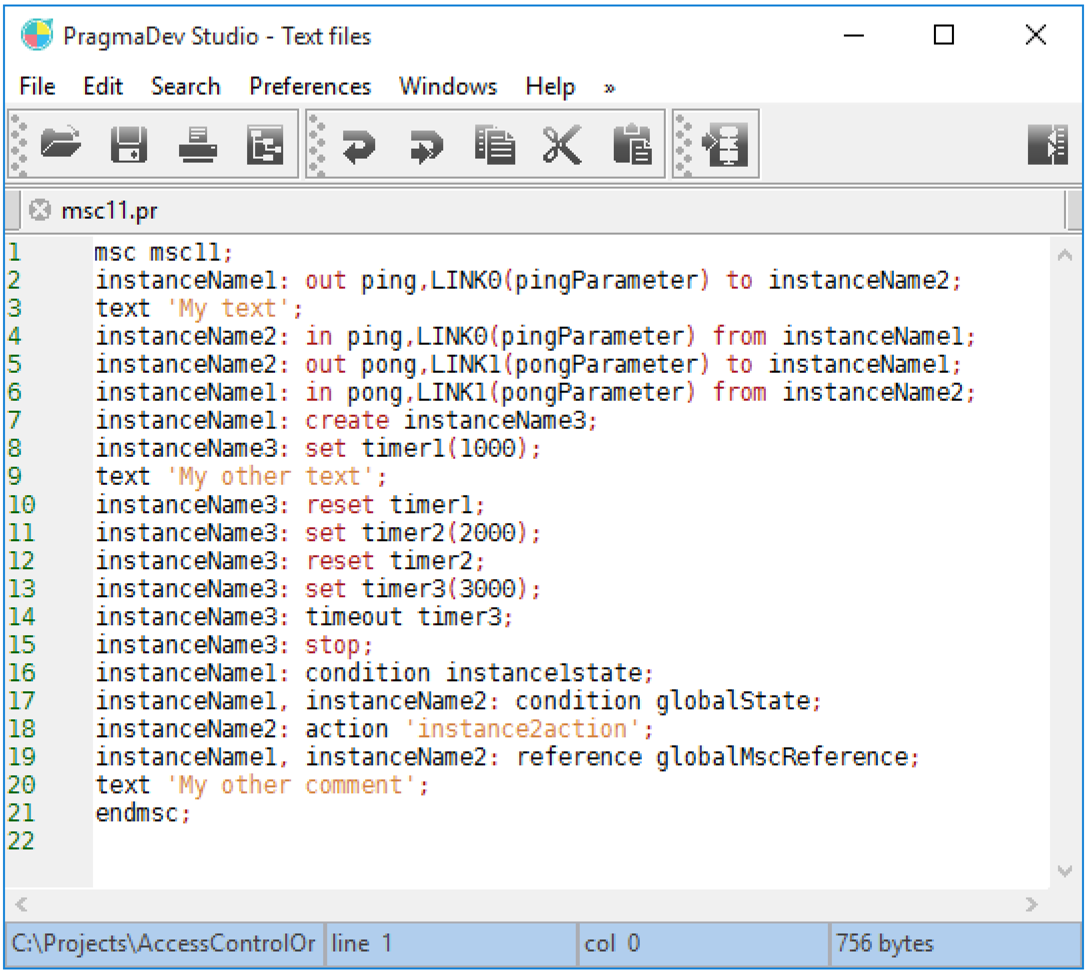


Imported MSC

2.3.3.10 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:

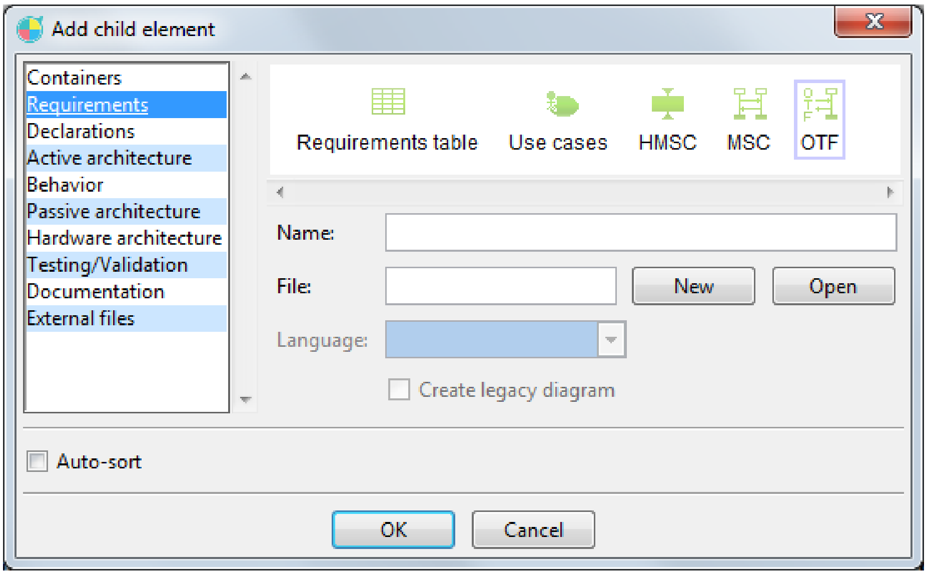

PragmaDev Studio - Text files
File Edit Search Preferences Windows Help »
msc11.pr
C:\Projects\AccessControlOr
line 1
col 0
756 bytes

```
msc msc11;
instanceName1: out ping,LINK0(pingParameter) to instanceName2;
text 'My text';
instanceName2: in ping,LINK0(pingParameter) from instanceName1;
instanceName2: out pong,LINK1(pongParameter) to instanceName1;
instanceName1: in pong,LINK1(pongParameter) from instanceName2;
instanceName1: create instanceName3;
instanceName3: set timer1(1000);
text 'My other text';
instanceName3: reset timer1;
instanceName3: set timer2(2000);
instanceName3: reset timer2;
instanceName3: set timer3(3000);
instanceName3: timeout timer3;
instanceName3: stop;
instanceName1: condition instance1state;
instanceName1, instanceName2: condition globalState;
instanceName2: action 'instance2action';
instanceName1, instanceName2: reference globalMscReference;
text 'My other comment';
endmsc;
```

Export MSC PR example

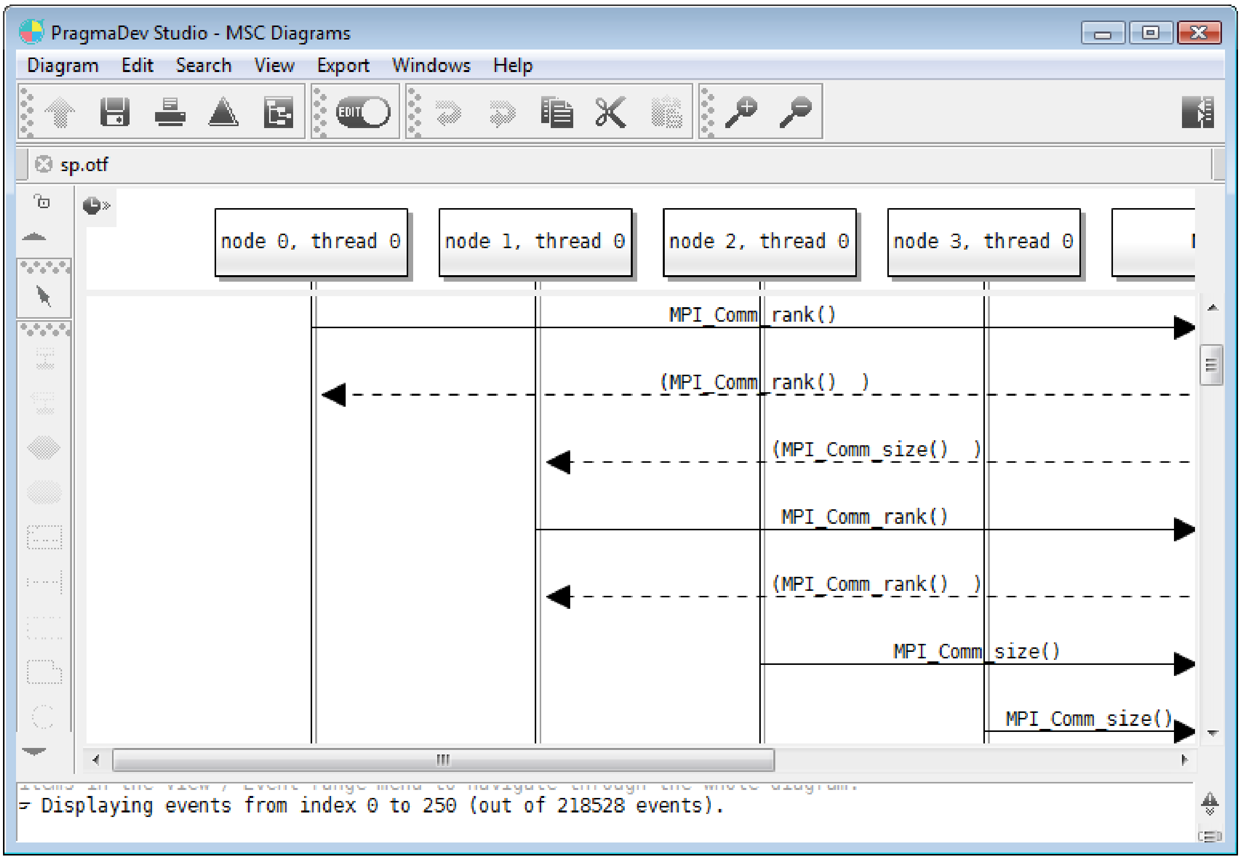
2.3.3.11 Open Trace Format (OTF) support

Traces in OTF support can be viewed in PragmaDev Studio in the MSC editor. To add an OTF trace right click on the project and *Add a child element... / Requirements* to the project. The following window will pop up:



This format is read-only: the traces cannot be modified, and the diagram cannot be saved back in OTF format.

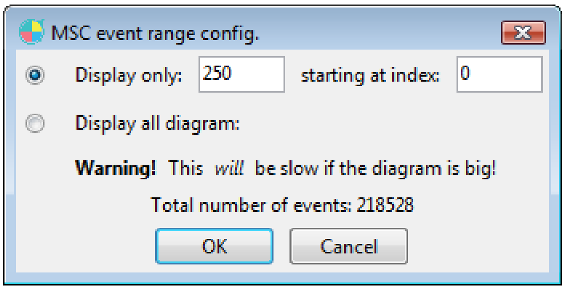
As OTF traces are often very low-level and include a very big number of events, they are often truncated when opening them to avoid displaying a huge diagram that would be very difficult to work with. If they are, a warning is displayed in the editor's notification zone:



```
items in the view / Event range menu to navigate through the whole diagram.
Displaying events from index 0 to 250 (out of 218528 events).
```

The range of displayed events can be changed by using the entries of the “*Event range*” sub-menu in the “*View*” menu:

- “*Next event range*” will move the “event window” forward, displaying the same amount of events, but starting just after the last event that was previously displayed.
- “*Previous event range*” will move the “event window” backward, displaying the same amount of events, but ending just before the first event that was previously displayed.
- “*Custom event range...*” will display the following dialog:



MSC event range config.

● Display only: 250 starting at index: 0

○ Display all diagram:

Warning! This *will* be slow if the diagram is big!

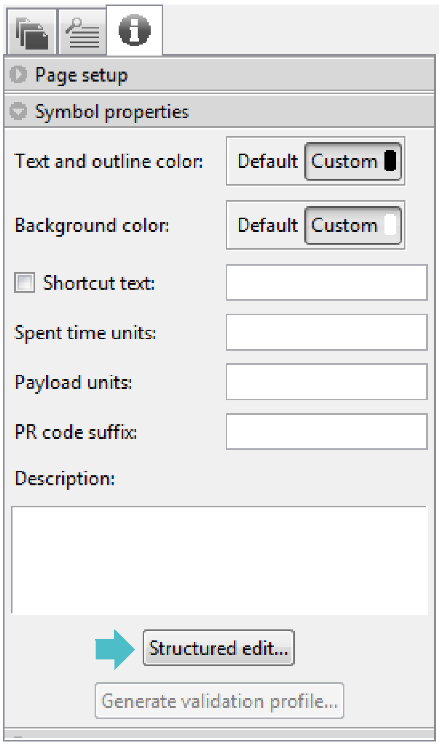
Total number of events: 218528

Note that displaying a big number of events is highly likely to have an impact on the performance of the editor.

2.3.4 UML diagrams

2.3.4.1 Symbol properties

For symbols that may show attributes and/or operations (class symbols in class diagrams, nodes and components in deployment diagram), it is possible to edit these attributes and operations in a guided and structured way, without any need to remember the syntax for the texts themselves. This can be done by using the "*Structured edit...*" button in the symbol's properties:



Text and outline color: Default Custom

Background color: Default Custom

□ Shortcut text:

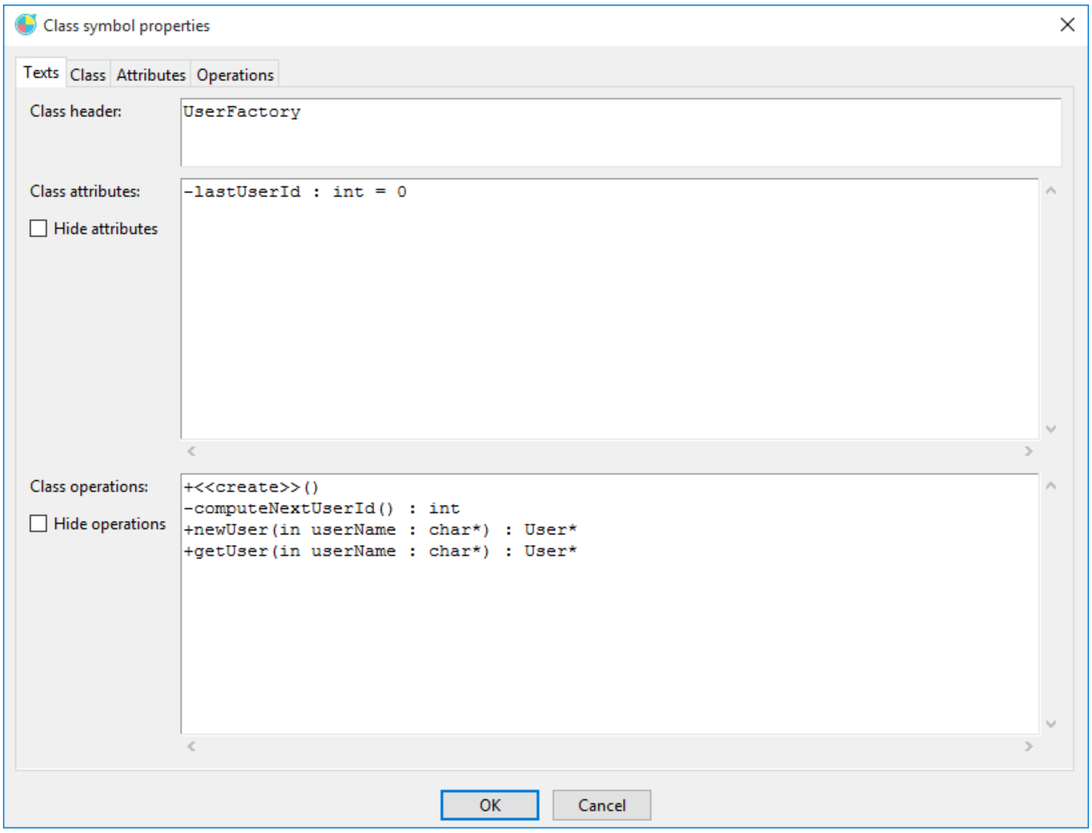
Spent time units:

Payload units:

PR code suffix:

Description:

The symbol's texts can then be edited in the following dialog:



Class symbol properties

Class header: UserFactory

Class attributes:

```
-lastUserId : int = 0
```

□ Hide attributes

Class operations:

```
+<<create>>()
-computeNextUserId() : int
+newUser(in userName : char*) : User*
+getUser(in userName : char*) : User*
```

□ Hide operations

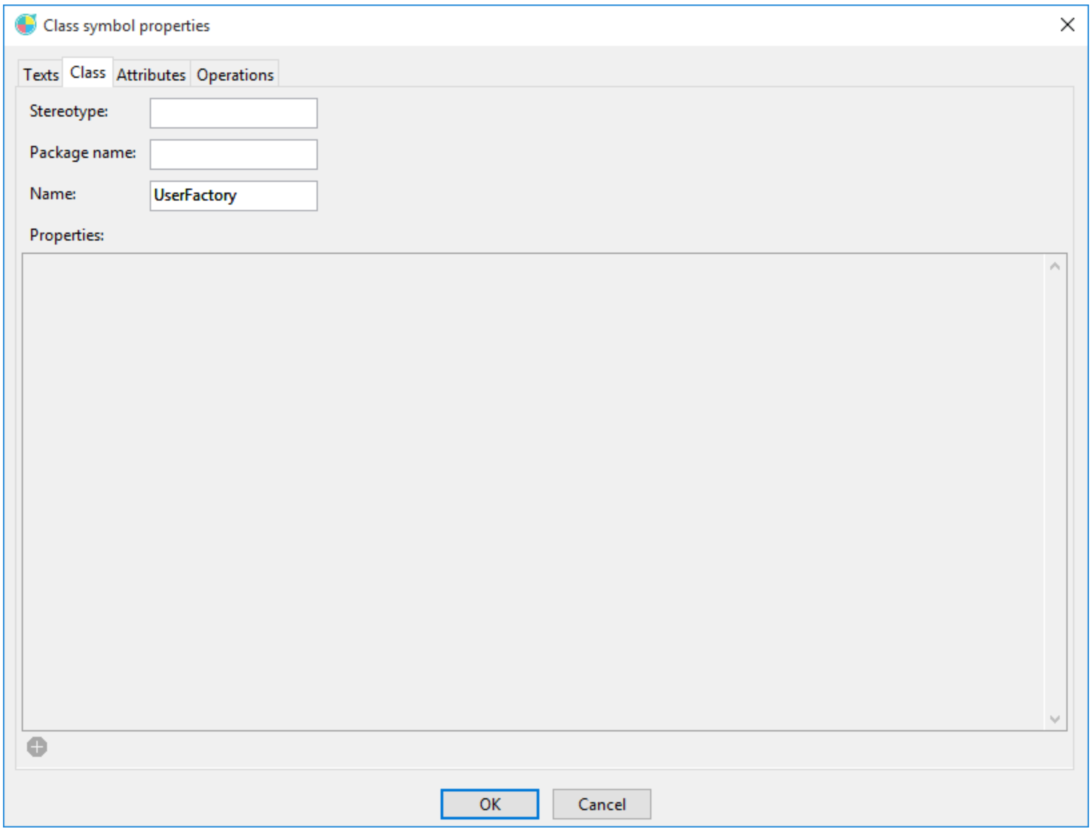
This property sheet is divided into 3 or 4 tabs depending on the selected symbol. These tabs are described in the following paragraphs.

2.3.4.1.1 “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.

2.3.4.1.2 “Class” tab

This tab is the second one in the dialog:



Stereotype:

Package name:

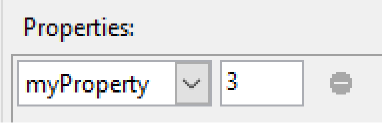
Name: UserFactory

Properties:

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:

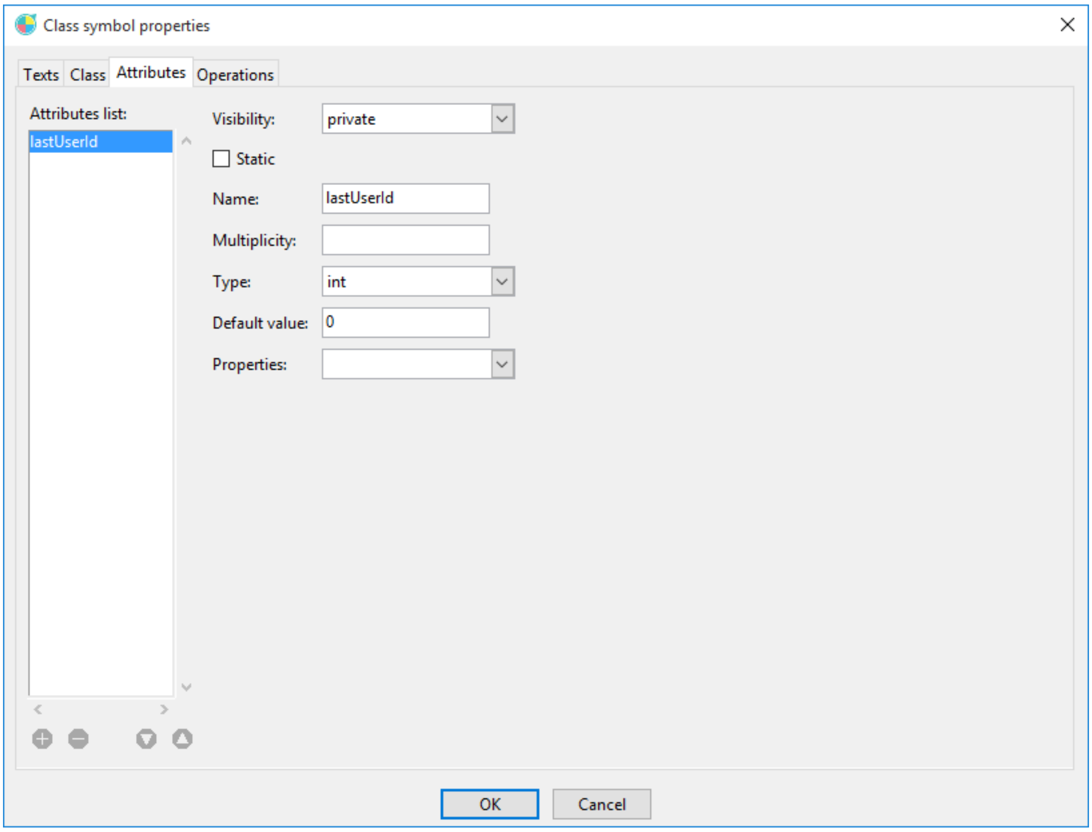


Properties:

myProperty 3

2.3.4.1.3 “Attributes” tab

This tab is the third in the dialog:



Attributes list:
lastUserId

Visibility: private

□ Static

Name: lastUserId

Multiplicity:

Type: int

Default value: 0

Properties:

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:

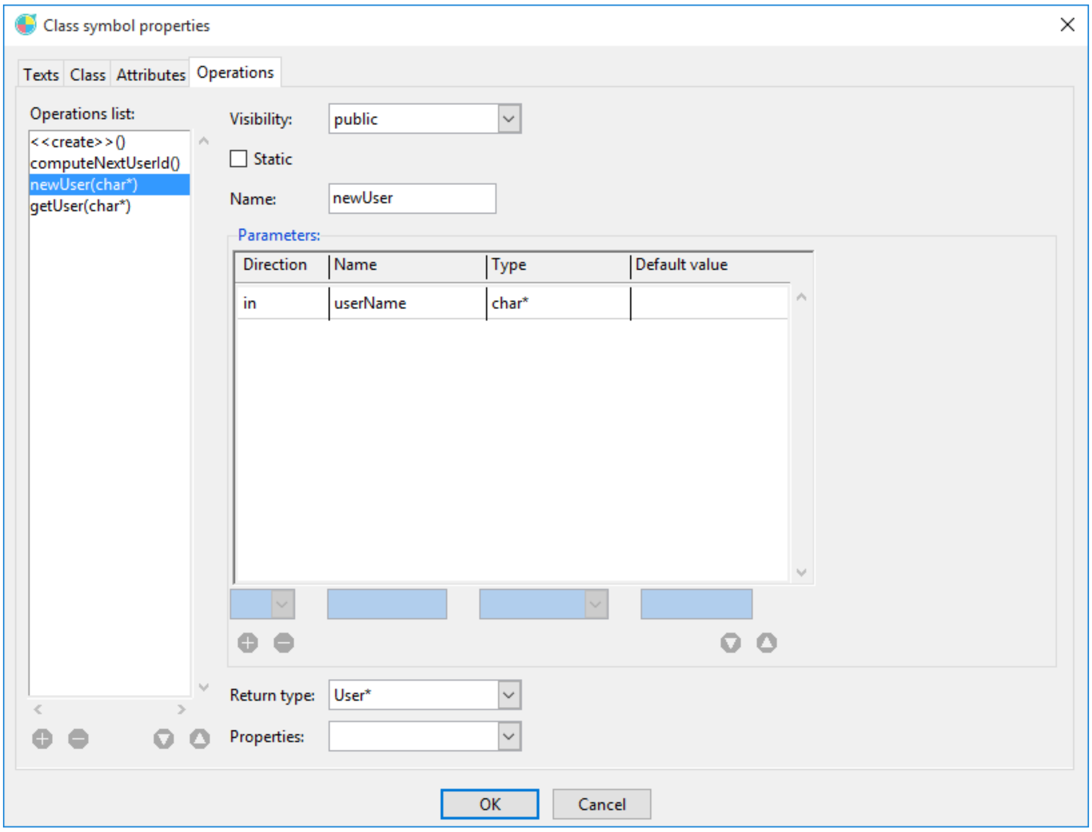
```
visibility name[multiplicity] : type = default-value {property-string}
```

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.

2.3.4.1.4 “Operations” tab

This tab is the fourth in the dialog:



Operations list:
<<create>>()
computeNextUserId()
newUser(char*)
getUser(char*)

Visibility: public
□ Static
Name: newUser

Parameters:

Direction	Name	Type	Default value
in	userName	char*	

Return type: User*
Properties:

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
```

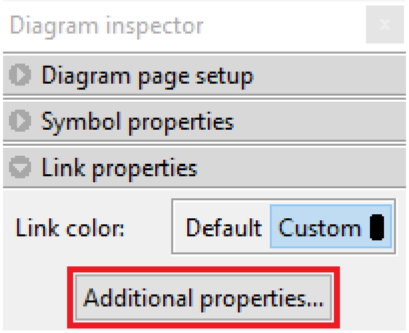
The visibility is rendered as the standard UML character (’+’ for public, ’#’ for protected, and ’-’ for private). The return type and property string for the operation and the type and default value for the parameters are included only if they are not empty.

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

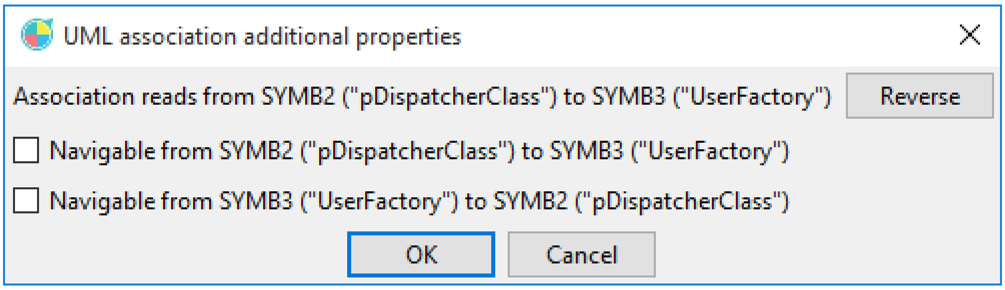
Any modified operation will be automatically updated in the operations text in the “Text” tab.

2.3.4.2 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:



Association reads from SYMB2 ("pDispatcherClass") to SYMB3 ("UserFactory")

□ Navigable from SYMB2 ("pDispatcherClass") to SYMB3 ("UserFactory")

□ Navigable from SYMB3 ("UserFactory") to SYMB2 ("pDispatcherClass")

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.

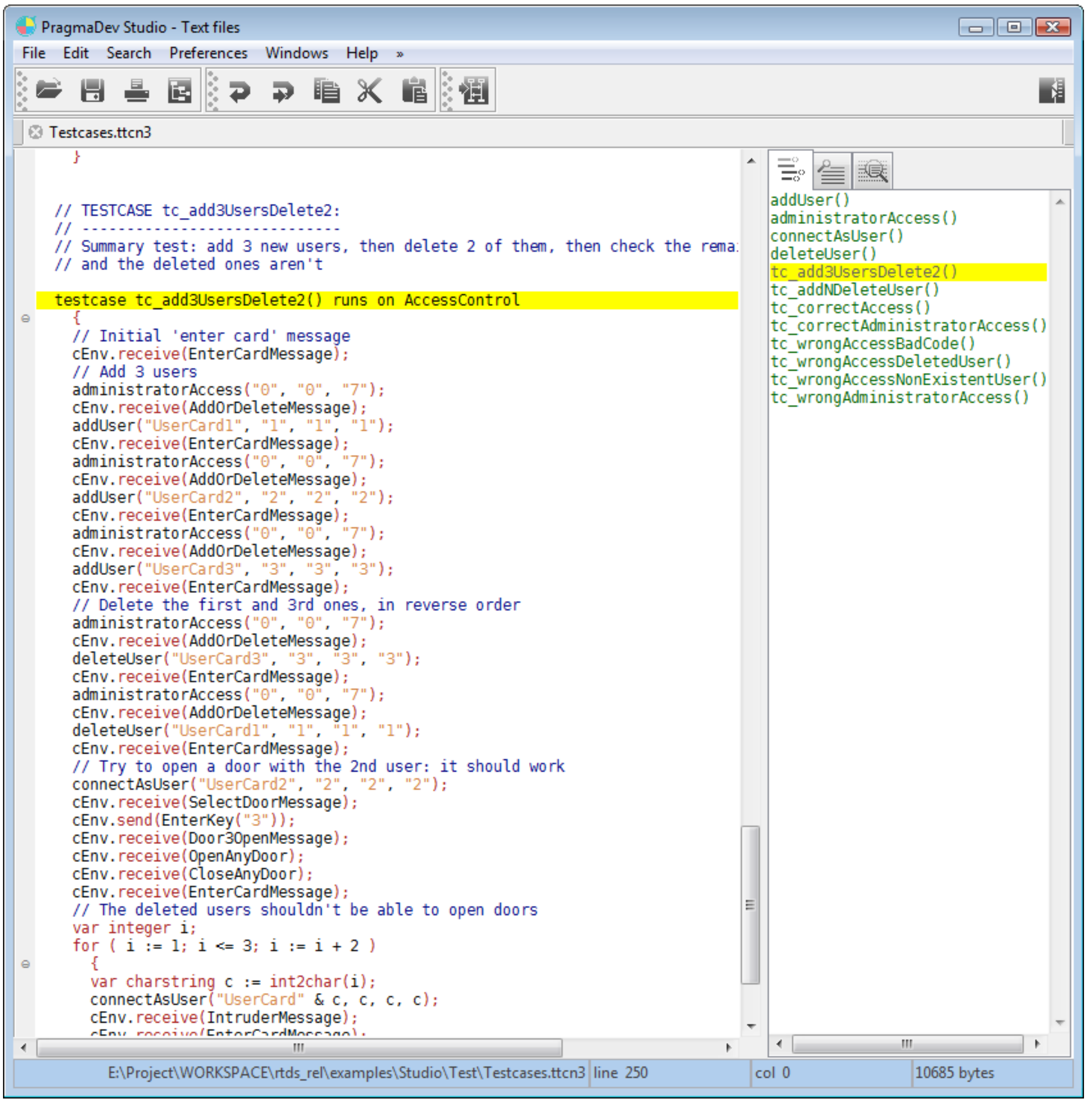
2.3.4.3 Access to generated C++ files

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.

2.4 - 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++, SDL-PR (declarations only), SDL-RT declarations, ASN.1, and TTCN-3. A browser at its right side also lists all the functions, classes and methods in the file, and allows to jump to any of these by clicking on their name. This browser is available for C/C++ and TTCN-3:


PragmaDev Studio - Text files
File Edit Search Preferences Windows Help »
Testcases.ttcn3

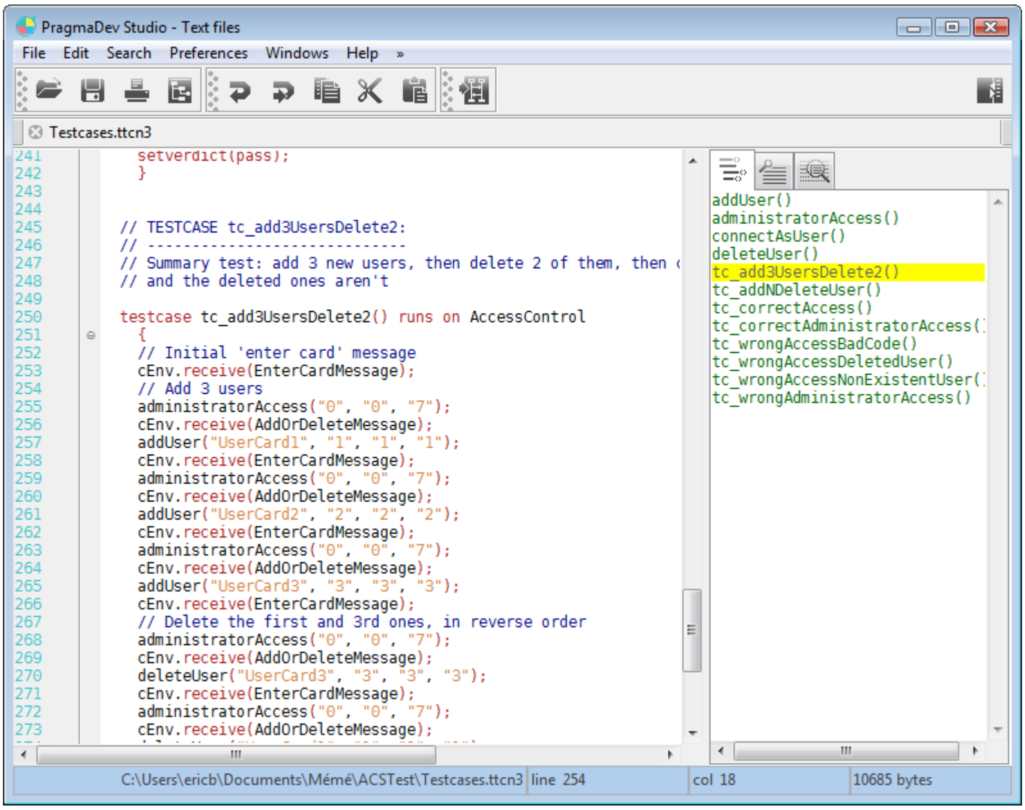
```
    }


  // TESTCASE tc_add3UsersDelete2:
  // ----------------------------
  // Summary test: add 3 new users, then delete 2 of them, then check the rema
  // and the deleted ones aren't

  testcase tc_add3UsersDelete2() runs on AccessControl
    {
    // Initial 'enter card' message
    cEnv.receive(EnterCardMessage);
    // Add 3 users
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    addUser("UserCard1", "1", "1", "1");
    cEnv.receive(EnterCardMessage);
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    addUser("UserCard2", "2", "2", "2");
    cEnv.receive(EnterCardMessage);
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    addUser("UserCard3", "3", "3", "3");
    cEnv.receive(EnterCardMessage);
    // Delete the first and 3rd ones, in reverse order
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    deleteUser("UserCard3", "3", "3", "3");
    cEnv.receive(EnterCardMessage);
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    deleteUser("UserCard1", "1", "1", "1");
    cEnv.receive(EnterCardMessage);
    // Try to open a door with the 2nd user: it should work
    connectAsUser("UserCard2", "2", "2", "2");
    cEnv.receive(SelectDoorMessage);
    cEnv.send(EnterKey("3"));
    cEnv.receive(Door3OpenMessage);
    cEnv.receive(OpenAnyDoor);
    cEnv.receive(CloseAnyDoor);
    cEnv.receive(EnterCardMessage);
    // The deleted users shouldn't be able to open doors
    var integer i;
    for ( i := 1; i <= 3; i := i + 2 )
      {
      var charstring c := int2char(i);
      connectAsUser("UserCard" & c, c, c, c);
      cEnv.receive(IntruderMessage);
      cEnv.receive(EnterCardMessage);
```

```
addUser()
administratorAccess()
connectAsUser()
deleteUser()
tc_add3UsersDelete2()
tc_addNDeleteUser()
tc_correctAccess()
tc_correctAdministratorAccess()
tc_wrongAccessBadCode()
tc_wrongAccessDeletedUser()
tc_wrongAccessNonExistentUser()
tc_wrongAdministratorAccess()
```

E:\Project\WORKSPACE\rtds_rel\examples\Studio\Test\Testcases.ttcn3 | line 250 | col 0 | 10685 bytes

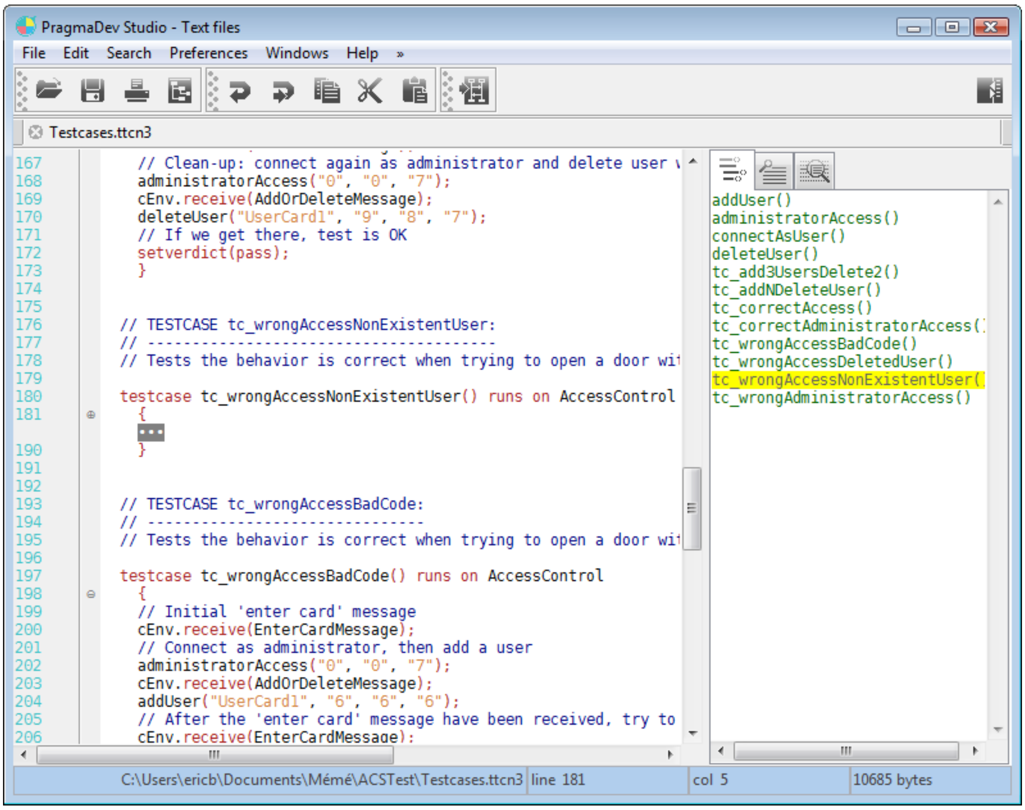
Syntax highlighting and function browsing is also available for other languages such as Python, Common Lisp, XML DTDs and CORBA IDL. This support may be partial only.

The current line and column numbers are indicated in the zone at the bottom of the editor. Line numbers can also be displayed on the left side of the window by selecting the “*Line numbers*” entry in the “*Preferences*” menu:



```
    setverdict(pass);
    }


  // TESTCASE tc_add3UsersDelete2:
  // -----------------------------
  // Summary test: add 3 new users, then delete 2 of them, then (
  // and the deleted ones aren't

  testcase tc_add3UsersDelete2() runs on AccessControl
    {
    // Initial 'enter card' message
    cEnv.receive(EnterCardMessage);
    // Add 3 users
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    addUser("UserCard1", "1", "1", "1");
    cEnv.receive(EnterCardMessage);
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    addUser("UserCard2", "2", "2", "2");
    cEnv.receive(EnterCardMessage);
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    addUser("UserCard3", "3", "3", "3");
    cEnv.receive(EnterCardMessage);
    // Delete the first and 3rd ones, in reverse order
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    deleteUser("UserCard3", "3", "3", "3");
    cEnv.receive(EnterCardMessage);
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
```

Between the line numbers and the text itself is the zone for code-folding buttons. These appear on the first line of each block of code. Clicking on this button allows to “fold” the block:


PragmaDev Studio - Text files
File Edit Search Preferences Windows Help »
Testcases.ttcn3

```
    // Clean-up: connect again as administrator and delete user \
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    deleteUser("UserCard1", "9", "8", "7");
    // If we get there, test is OK
    setverdict(pass);
    }


  // TESTCASE tc_wrongAccessNonExistentUser:
  // -------------------------------------
  // Tests the behavior is correct when trying to open a door wi

  testcase tc_wrongAccessNonExistentUser() runs on AccessControl
    {
    ...
    }


  // TESTCASE tc_wrongAccessBadCode:
  // -----------------------------
  // Tests the behavior is correct when trying to open a door wi

  testcase tc_wrongAccessBadCode() runs on AccessControl
    {
    // Initial 'enter card' message
    cEnv.receive(EnterCardMessage);
    // Connect as administrator, then add a user
    administratorAccess("0", "0", "7");
    cEnv.receive(AddOrDeleteMessage);
    addUser("UserCard1", "6", "6", "6");
    // After the 'enter card' message have been received, try to
    cEnv.receive(EnterCardMessage);
```

addUser()
administratorAccess()
connectAsUser()
deleteUser()
tc_add3UsersDelete2()
tc_addNDeleteUser()
tc_correctAccess()
tc_correctAdministratorAccess(
tc_wrongAccessBadCode()
tc_wrongAccessDeletedUser()
tc_wrongAccessNonExistentUser(
tc_wrongAdministratorAccess()

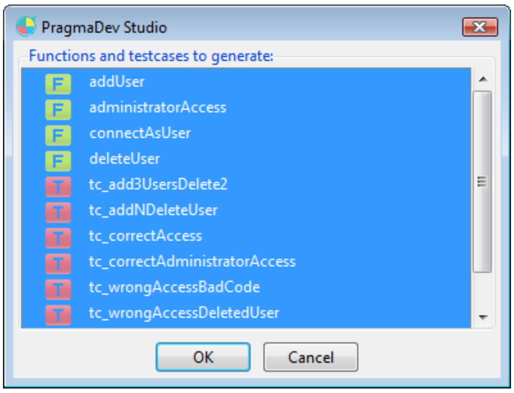
C:\Users\ericb\Documents\Mémé\ACSTest\Testcases.ttcn3 | line 181 | col 5 | 10685 bytes

Clicking again on the button allows to unfold a previously folded block.

Code-folding is available for C/C++, TTCN-3, ASN.1 and Python files.

2.4.1 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 . A dialog will open, looking like follows:



Functions and testcases to generate:

F addUser
F administratorAccess
F connectAsUser
F deleteUser
T tc_add3UsersDelete2
T tc_addNDeleteUser
T tc_correctAccess
T tc_correctAdministratorAccess
T tc_wrongAccessBadCode
T tc_wrongAccessDeletedUser

This dialog allows to select the functions and testcases in the current TTCN module that will be represented as MSC diagrams. By default, all are selected.

Once the dialog is validated, a MSC diagram editor will appear for each selected function or testcase, containing its MSC representation. Note that these diagrams are temporary: they are generated in temporary files which will be deleted as soon as the editor is closed. To keep the MSC representation, it is possible to use “Save as...” in the “Diagram” menu.

2.4.2 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.

2.5 - Documentation generation

PragmaDev Studio offers three ways to document your project: export whole or a part of the project as an html document, export the publications as graphics, organize the publication in a full document orgainized in chapters.

2.5.1 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.

2.5.1.1 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.

2.5.1.2 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.

2.5.2 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 68.

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.

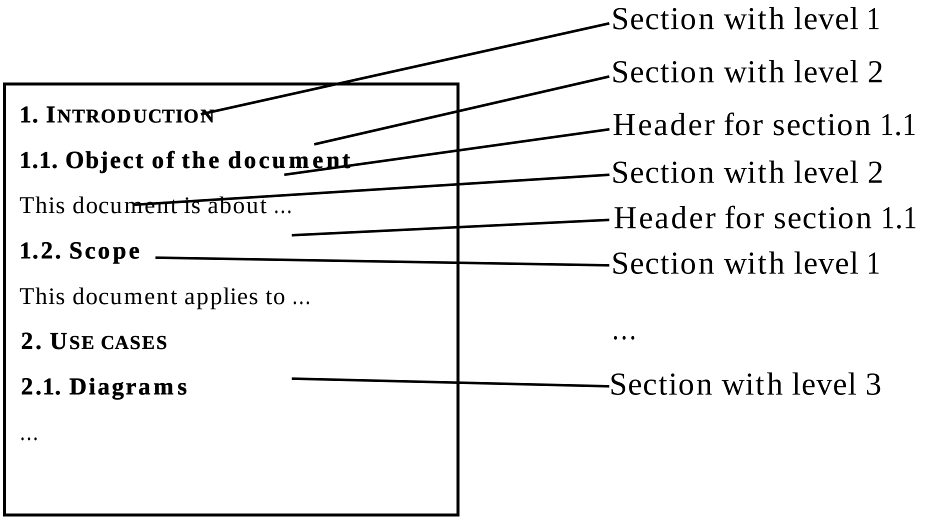
2.5.3 Document editor

2.5.3.1 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.



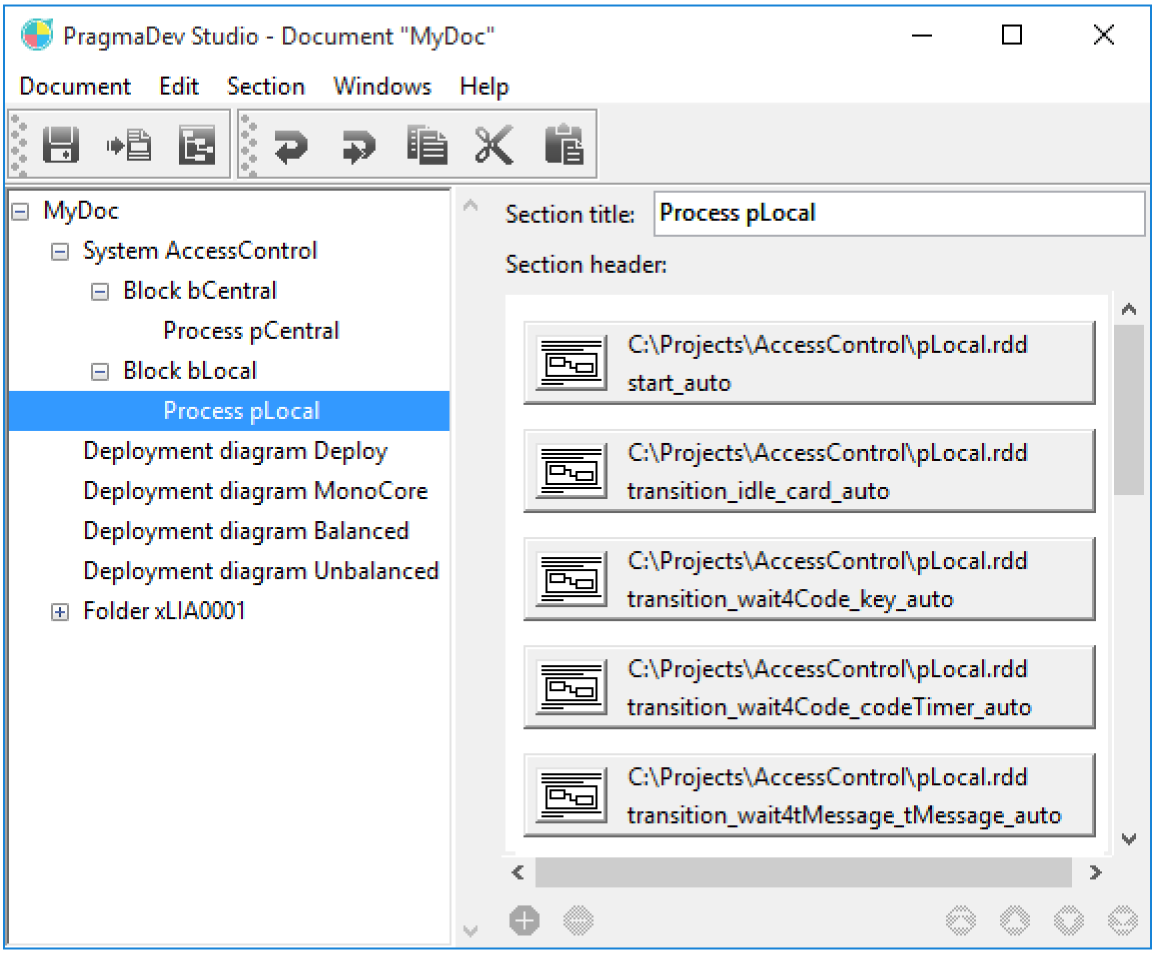
A section header is composed of header items, which can be:

- Text items: these items just contain styled text.
- Publication items: these items are references to publications in diagrams, as described in "Publications" on page 68. 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:



- MyDoc
  - System AccessControl
    - Block bCentral
      - Process pCentral
    - Block bLocal
      - Process pLocal
    - Deployment diagram Deploy
    - Deployment diagram MonoCore
    - Deployment diagram Balanced
    - Deployment diagram Unbalanced
    - Folder xLIA0001

Section title: Process pLocal

Section header:

C:\Projects\AccessControl\pLocal.rdd
start_auto

C:\Projects\AccessControl\pLocal.rdd
transition_idle_card_auto

C:\Projects\AccessControl\pLocal.rdd
transition_wait4Code_key_auto

C:\Projects\AccessControl\pLocal.rdd
transition_wait4Code_codeTimer_auto

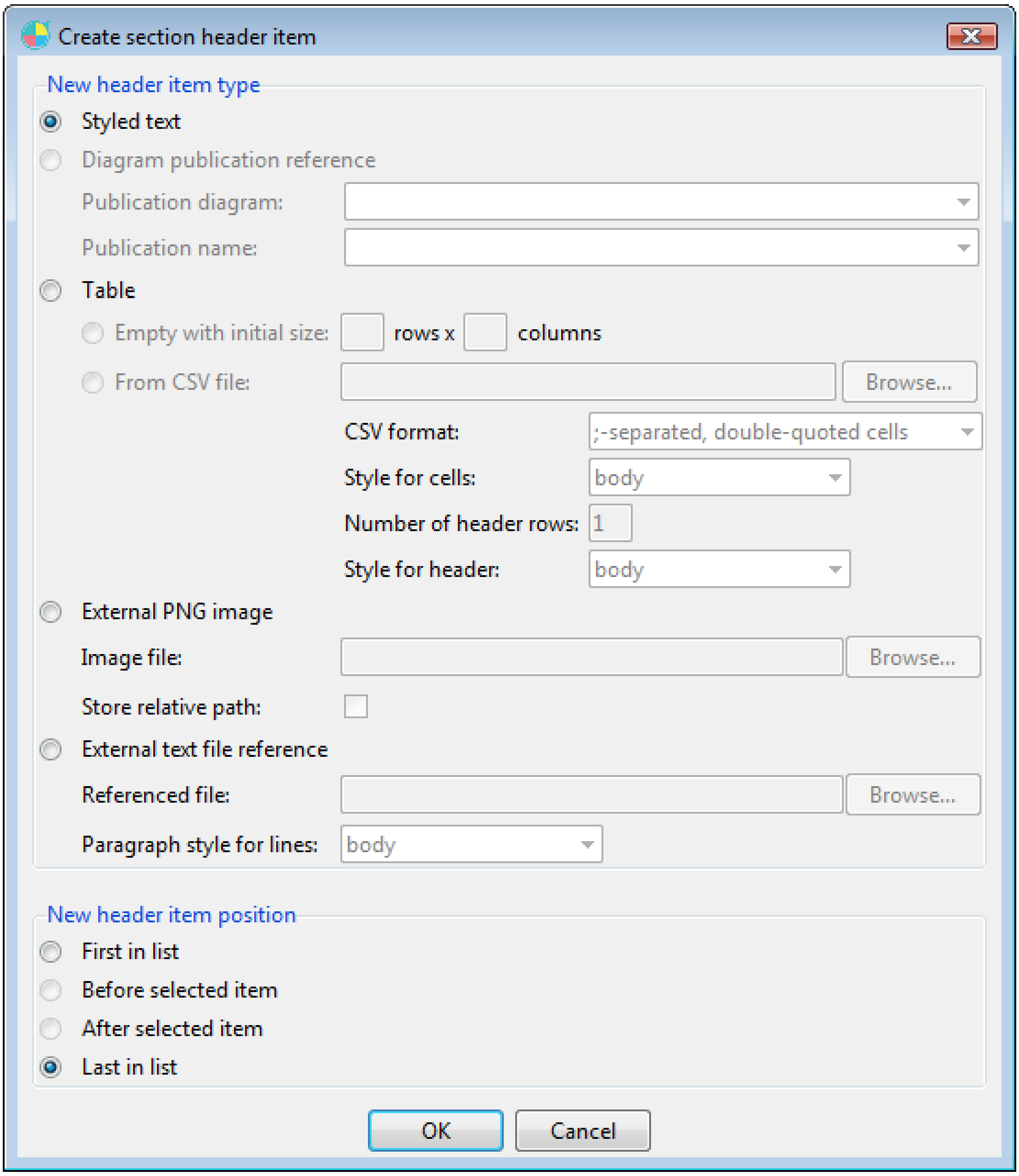
C:\Projects\AccessControl\pLocal.rdd
transition_wait4tMessage_tMessage_auto

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:



Create section header item

New header item type

● Styled text

○ Diagram publication reference

Publication diagram:

Publication name:

○ Table

○ Empty with initial size: rows x columns

○ From CSV file: Browse...

CSV format: ;-separated, double-quoted cells

Style for cells: body

Number of header rows: 1

Style for header: body

○ External PNG image

Image file: Browse...

Store relative path: □

○ External text file reference

Referenced file: Browse...

Paragraph style for lines: body

New header item position

○ First in list

○ Before selected item

○ After selected item

● Last in list

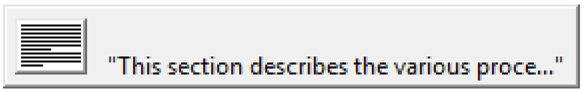
The upper zone allows to choose the type of header item to insert - text, publication, table, external image or external text file - and all required information for the header item:

- 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.

- For table items, either the initial number of rows, or a CSV file with its format information, the style to give to cells - normal and header -, and the number of rows in the CSV file to use as header rows.
- 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.

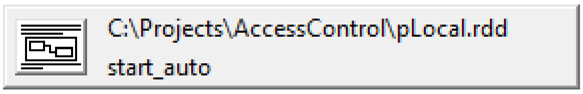
Once created, a text header item will appear like this:



"This section describes the various proce..."

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

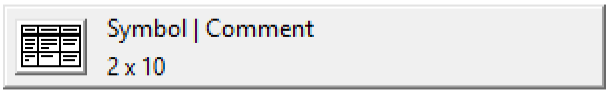
Publication header items will appear like this:



C:\Projects\AccessControl\pLocal.rdd
start_auto

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

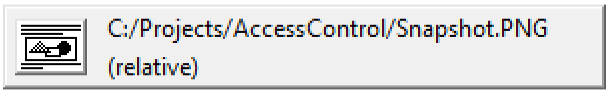
Table items will appear like this:



Symbol | Comment
2 x 10

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:



C:/Projects/AccessControl/Snapshot.PNG
(relative)

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 directly to a PDF file, or to a format that a word processor can handle. The formats available today are:

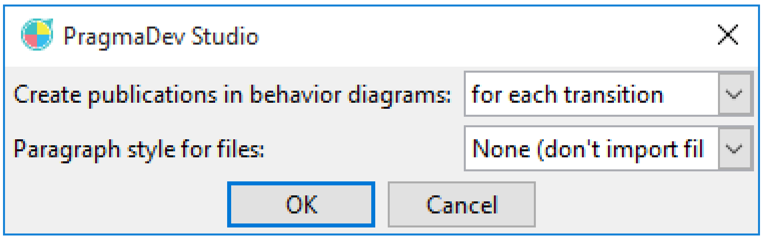
- RTF, which can be used with all the major word processors;
- OpenDocument format, which is an ISO-standard for text documents, mainly used by OpenOffice.org and other open-source word processors;
- SGML; this format is for advanced users and won't be described further in this document. Please refer 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 defining styles and setting up options for the documentation, as described in the sections “Documentation styles & options” on page 131.

2.5.3.2 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 publications 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:



Create publications in behavior diagrams: for each transition

Paragraph style for files: None (don't import fil

The options in the dialog are the following:

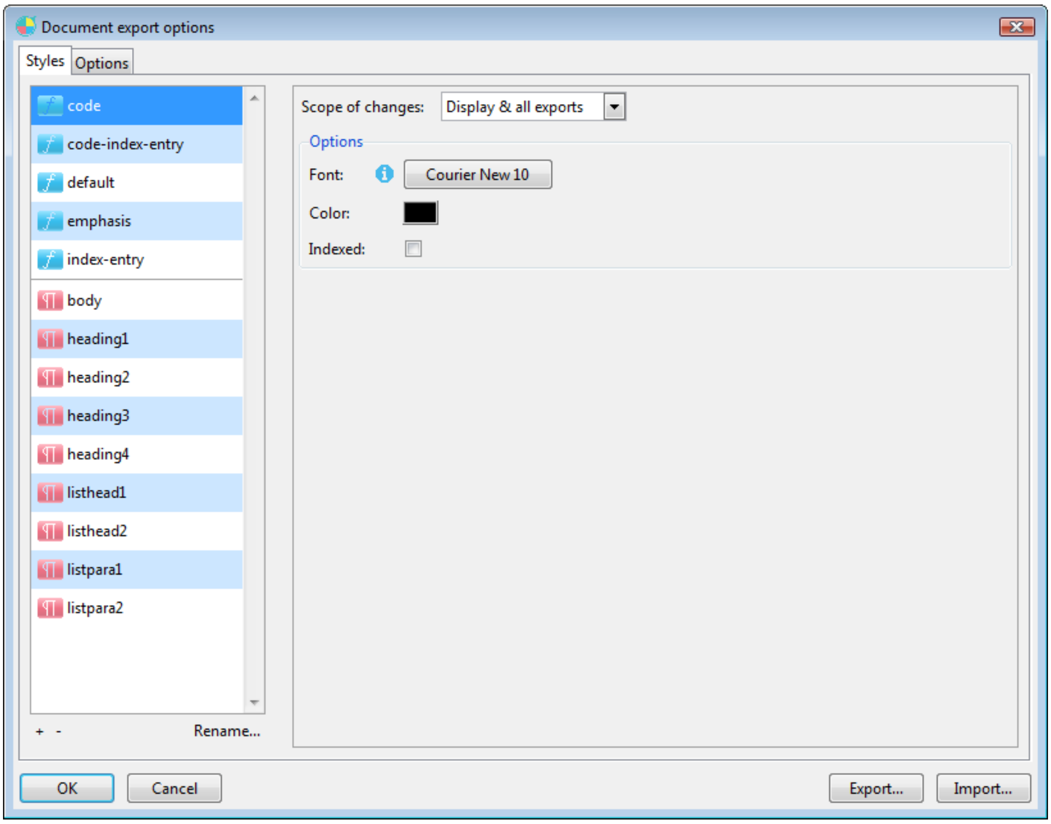
- *Create publications in behavior diagrams*
  Allows to specify the level at which the publications are created in behavioral diagrams such as processes or procedures. The available choices are:
  - *For each partition*
    A publication will be created for each partition in the diagram, exporting all 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 72.

2.5.3.3 Documentation styles & options

Documentation styles & options are configured at project level by selecting the entry ‘*Documentation styles & options*’ in the ‘*Project*’ menu in the project manager. It allows to define all the styles that can be used in documents, setup how they will appear on screen and in all available export formats, and to define the general options for documentation exports. This is done in the following dialog:



Document export options

Scope of changes: Display & all exports

Options

Font: Courier New 10

Color: ■

Indexed: □

The two tabs in the dialog allow to define the styles and export options. The styles tab displays a list of existing styles on the left side, with buttons allowing to create, delete or rename a style. There are 2 kinds of styles:

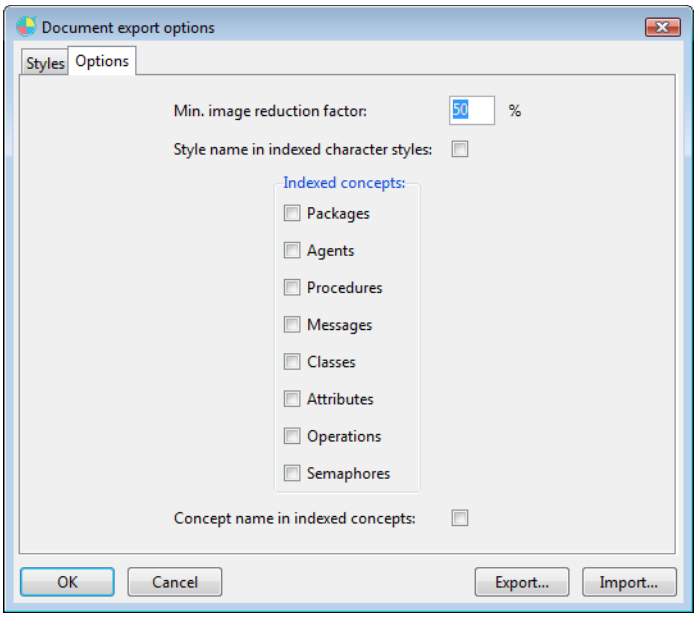
- Paragraph styles are identified by a ¶ in front of their names; they define the appearance for whole paragraphs. This concept is the same as the usual styles found in word processors. Typical options for paragraphs include the font for the paragraph text, its margins, its alignment, and so on...
  Paragraph styles are described in detail in subsection “Paragraph styles” on page 137.

- Character styles are identified by a f in front of their names; they define the appearance for range of characters within paragraphs. Options for these styles are just a font (with its family, its size and its style), a color, and whether the text with this style should appear in the document index.
  Character styles are described in detail in subsection “Character styles” on page 136.

Projects created with older version of PragmaDev Studio may have missing styles, especially for exports. In this case, a warning will be displayed when the project is opened saying that the missing styles have been created and should be reviewed. It is advised to do so, as the values for the options will be default values and some styles might even end up being invalid. See the description of indicators in “Style definitions” on page 133 for a way to spot quickly which options can cause issues.

Also note that styles for the OpenDocument export format are actually not used when exporting. Today, this kind of export is 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.

The documentation options appear like follows:



Min. image reduction factor: 50 %

Style name in indexed character styles: □

Indexed concepts:

□ Packages
□ Agents
□ Procedures
□ Messages
□ Classes
□ Attributes
□ Operations
□ Semaphores

Concept name in indexed concepts: □

- The option ‘*Min. image reduction factor*’ defines how images 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 at least by this factor before doing any other adaptation.
- Selecting the ‘*Style name in indexed character styles*’ checkbox allows to indicate that index entries must appear as ‘<text> (<character style name>)’ in the generated index. The default is to include only the text without any prefix.
  For example, if the text “engine” appears anywhere with the character style “External element” defined as an indexed style (cf. “Character styles” on page 136 for this option), it will appear in the generated index for the document as:
  - “engine” if the option ‘*Style name in indexed character styles*’ is not checked;

  - “engine (External element)” if the option ‘*Style name in indexed character styles*’ is checked.
- The options in the ‘*Indexed concepts*’ group define which elements will appear automatically in the index. For example, if ‘*Packages*’ is checked, all package names will appear in the index, wherever they are found: in “USE” clauses, in UML class diagrams, and so on.
- The ‘*Concept name in indexed concepts*’ checkbox allows to include the concept name in all generated index entries specified in the ‘*Indexed concepts*’. For example, a package named ‘Classes’ would appear as “Classes (package)” in the index with this option checked. The default is to use only “Classes” as the index entry text.

2.5.3.3.1 Style definitions

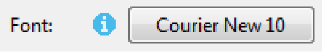
The ‘Documentation styles & options’ dialog allows to setup all options that will be used to display the various styles as well as when they will be exported to any of the supported document formats.

To indicate the scope of the changes, a selector named ‘Scope of changes’ is always present on the top of the options for the style. It can have the following values:

- ‘*Display & all exports*’: with this value, everything configured in the dialog will impact how the style is displayed and how it will be exported in all export formats. For example, if the font for the style is set to ‘Times’, this font will be used when displaying the text and when exporting in OpenDocument, RTF, HTML, PDF and SGML formats.
  This is the easiest and fastest way to define options for the documents, as every option needs to be setup only once. Be aware though that it can have unwanted side-effects: some option values are not available in all export formats, and having the same value for all of them is sometimes simply impossible. So once option values have been defined for everything, they may have to be adjusted for specific export formats.
  Indicators in front of option values will help you figure out which ones need adjusting. See below.
- ‘*Display*’: with this value, everything configured in the dialog will only impact how the style is displayed. Note that some options are either meaningless or not available when displaying text, so they will be greyed out if this scope is selected.
- ‘<export format> *export*’: with one of these values, everything configured in the dialog will only impact the style used when exporting a document to the specified format. Available export formats are OpenDocument, RTF, HTML, PDF and SGML.

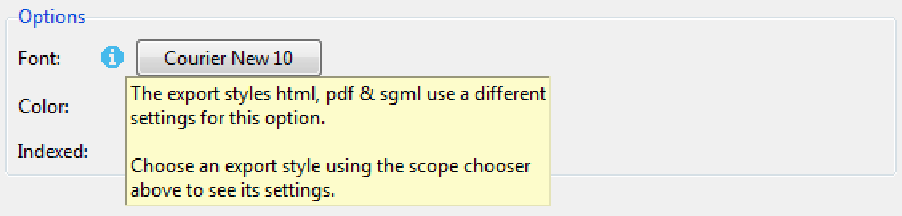
After configuration, indicators may appear in front of option values in special conditions:

- If the option does not have the same value for all styles, it will have an “information”indicator in front of it:



Font: ℹ Courier New 10

Hovering over the indicator with the mouse pointer will display a tooltip explaining why it is present:



Options

Font: ⓘ Courier New 10

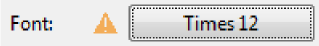
Color:

Indexed:

The export styles html, pdf & sgml use a different settings for this option.

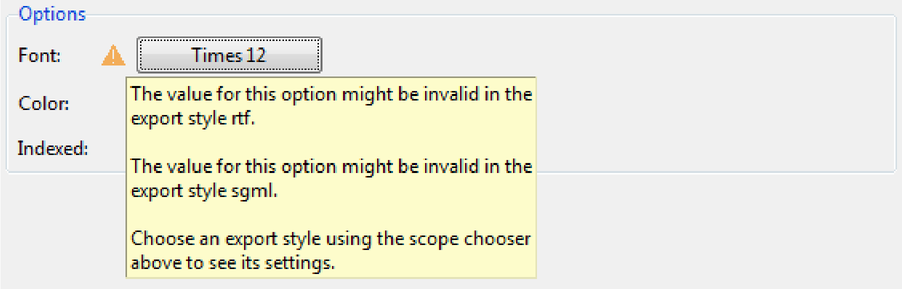
Choose an export style using the scope chooser above to see its settings.

- If the value for this option for one of the styles might be invalid, it will have a “warning” indicator in front of it:



Font: ⚠ Times 12

Again, a tooltip displayed when hovering over the indicator will explain its presence:



Options

Font: ⚠ Times 12

Color:

Indexed:

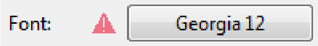
The value for this option might be invalid in the export style rtf.

The value for this option might be invalid in the export style sgml.

Choose an export style using the scope chooser above to see its settings.

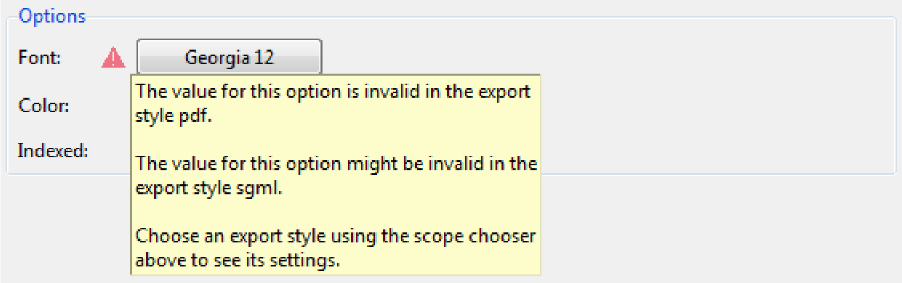
The problem will typically happen for the text font, as some export styles have a limited set of fonts that will always be available, when any other one will depend on what’s installed on the system where the document will be opened.

- If the value for this option for one of the styles is invalid, it will have an “error” indicator in front of it:



Font: ⚠ Georgia 12

Again, the tooltip will give the reason for the error:



Options

Font: ⚠ Georgia 12

Color:

Indexed:

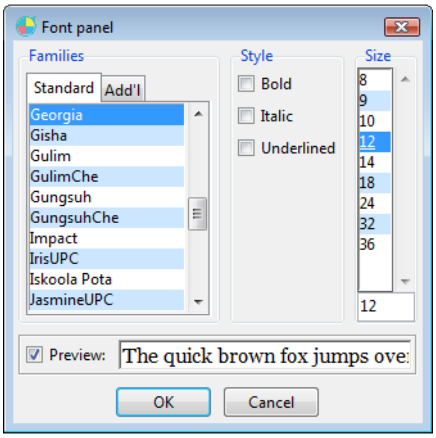
The value for this option is invalid in the export style pdf.

The value for this option might be invalid in the export style sgml.

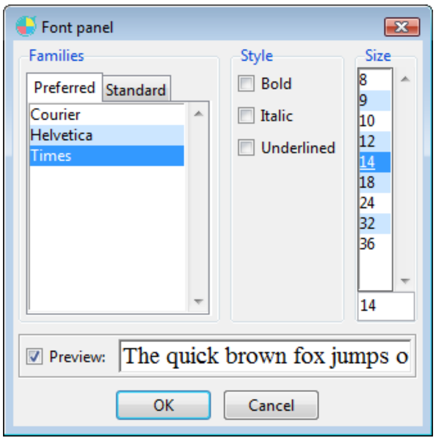
Choose an export style using the scope chooser above to see its settings.

This problem occurs for styles having only a defined set of fonts that are available, and no other will work.

Since most issues happen with the text font, a special selector for fonts is used for documentation options. It looks like follows:



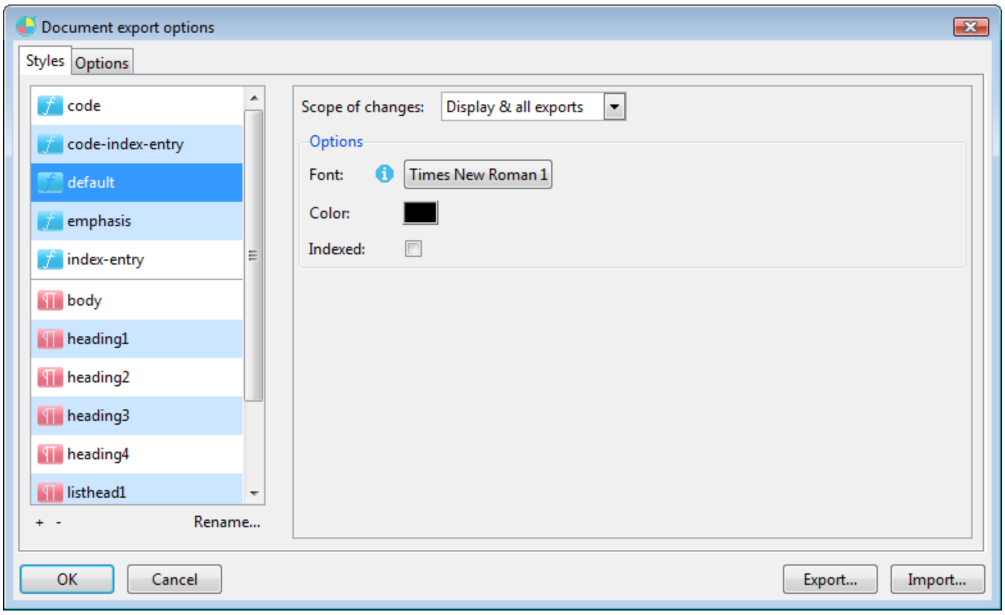
In addition to all the fonts that could be identified on the current system and that will appear in the “Standard” tab, there is a specific set of fonts in the “Preferred” tab that will list the fonts that are guaranteed to work for this export styles, whether they actually are installed on the current system or not. For example, for the PDF export style, the “Preferred” tab looks like this:



This indicates that only the “Courier”, “Helvetica” and “Times” fonts will always work when exporting PDF.

2.5.3.3.2 Character styles

Character styles appear as follows in the ‘*Documentation styles & options*’ dialog:



Scope of changes: Display & all exports

Font: Times New Roman 1

Color:

Indexed: □

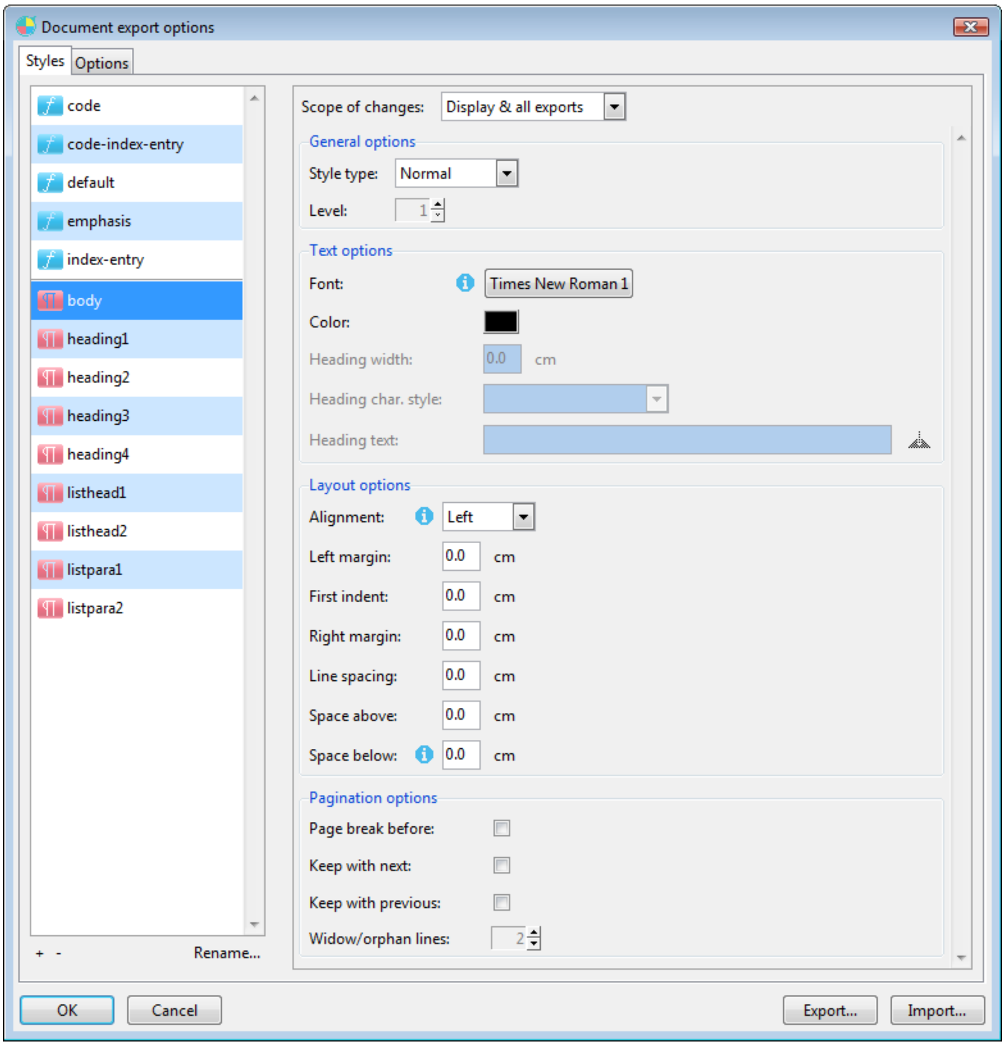
The “Scope of changes” option is explained in “Style definitions” on page 133. The other options are:

- ‘*Font*’: the font for the text with its size and style (normal, italic, bold, etc...). Clicking on the button containing the font name will open a font selector, which is the one described in “Style definitions” on page 133.
- ‘*Color*’: color for the text. Clicking on the color will open a color selector.
- ‘*Indexed*’: if checked, texts with this style will automatically appear in the index in the generated document.

This last option is actually the only way to define “custom” index entries today. Note that 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.

2.5.3.3.3 Paragraph styles

Paragraph styles appear as follows in the ‘*Documentation styles & options*’ dialog:



Scope of changes: Display & all exports

Style type: Normal

Level: 1

Font: Times New Roman 1

Color:

Heading width: 0.0 cm

Heading char. style:

Heading text:

Alignment: Left

Left margin: 0.0 cm

First indent: 0.0 cm

Right margin: 0.0 cm

Line spacing: 0.0 cm

Space above: 0.0 cm

Space below: 0.0 cm

Page break before: □

Keep with next: □

Keep with previous: □

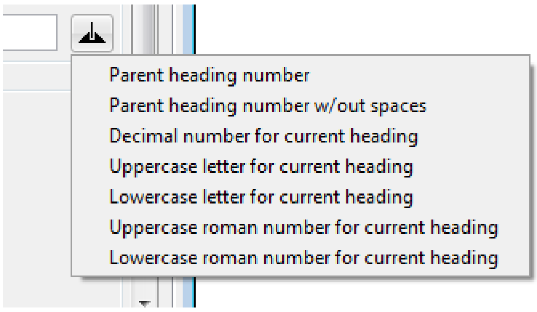
Widow/orphan lines: 2

The ‘Scope of changes’ options is described in “Style definitions” on page 133. The other options are displayed in groups:

- ‘*General options*’: these options apply to the style as a whole and are not specific to the display or any export format. They will only be modifiable if the selected scope is “*Display & all exports*”.
  - ‘*Style type*’: can be “*Normal*”, “*Heading*”, “*List header*” or “*List para.*”.
    A “Normal” style is for paragraphs within the text with no particular attributes. They are typically used for the body of the text, or in some special styles such as paragraphs containing code.
    A “Heading” style is for section headers. They usually include a section number. Note that these paragraphs do not appear in the document editor, as the sections are only displayed in the section tree with no particular style.
    A “List header” style is for bulleted or enumerated lists. These paragraphs are the first one in list items and include the bullet or number.
    A “List para.” style is for paragraph within a bulleted or enumerated list, but that are continuation of the previous item. They usually have the same margins as the list header paragraph, but do not include a bullet or number.

    - '*Level*' is only available for '*Heading*', '*List header*' or '*List para.*' styles. It gives the nesting level of the section or list.

- '*Text options*': these options control the appearence of the text as well as the text appearing in th header if any (secion header or list bullet or number).
    - '*Font*': the font for the paragraph text, with its size and style (normal, italic, bold, etc...). Clicking on the button containing the font name will open a font selector, which is the one described in "Style definitions" on page 133.
    - '*Color*': the color for the text. Clicking on the color button will open a color selector.
    - '*Heading width*': the width for the paragraph heading if applicable, in centimeters. This option is only available for paragraph having a heading, i.e section headers and list headers. See the description for the '*Layout options*' below for a detailed explanation of this option and how it relates to the margins.
    - '*Heading char. style*': the character style for the heading if applicable. This option is only available for paragraph having a heading, i.e section headers and list headers. The value is the name of a character style. If left blank, the heading text will have the same style as the paragraph text.
    - '*Heading text*': text for the heading if applicable. This option is only available for paragraph having a heading, i.e section headers and list headers. 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:


Parent heading number
Parent heading number w/out spaces
Decimal number for current heading
Uppercase letter for current heading
Lowercase letter for current heading
Uppercase roman number for current heading
Lowercase roman number for current heading

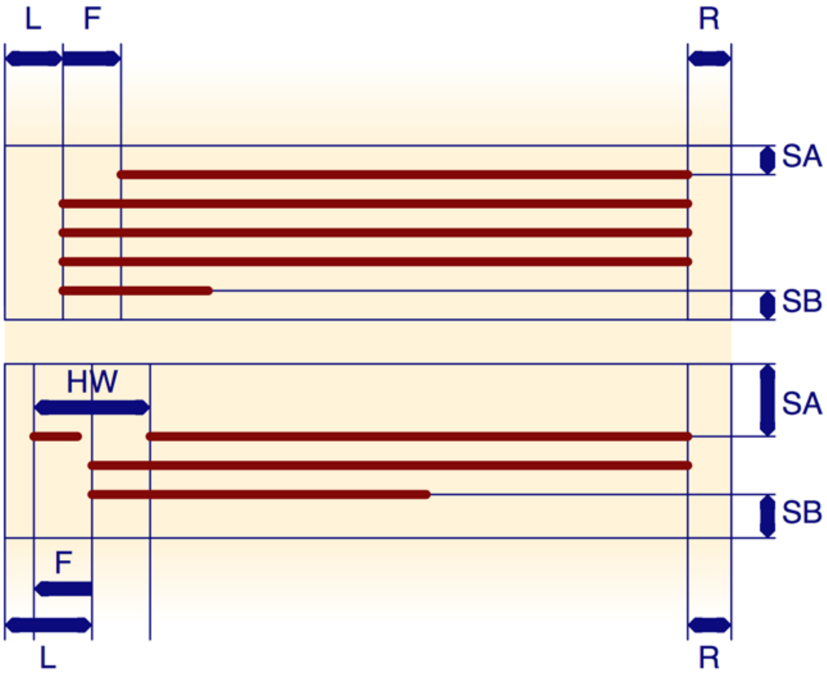
    For example, for a classic section numbering with "1. " for the first level, "1.1. " for the second one, "1.1.1. " for the third one, and so on, the heading texts would be:
        - "*<Decimal number for current heading>*. " for the heading at level 1;
        - "*<Parent heading number w/out spaces>.<Decimal number for current heading>*. " for headings at all other levels.

- '*Layout options*': these options concern the general layout of the text on the page, i.e the various margins.
    - '*Alignment*': text alignment, which can be '*Left*', '*Right*', '*Center*' or '*Full*' for full justification on the left & right borders.
    - '*Left margin*': the distance between the page border and the main paragraph text, in centimeters. See below for a more detailed explanation.
    - '*First indent*': the distance between the border of the main paragraph text and the position where the first paragraph line starts, in centimeters. This value

can be negative if the first line of the paragraph starts before the main text. See below.

- ‘*Right margin*’: the distance from the right border of the main paragraph text to the right side of the page, in centimeters. See below.
- ‘*Line spacing*’: the space between the lines of the paragraph, in centimeters. The value 0 means standard spacing. Use a negative value to bring lines closer to each other, and a positive value to increase the spacing.
- ‘*Space above*’: the minimum space above the paragraph, in centimeters. See below.
- ‘*Space below*’: the minimum space below the paragrph, in centimeters. See below.

Here is a graphical explanation of the various margins and spacings, with L = left margin, R = right margin, F = first indent, HW = heading width, SA = space above and SB = space below:



The paragraph at the top is a “normal” paragaph with a first line starting a bit to the right of the main paragraph text, so the first indent is positive. The paragraph at the bottom is a heading with its first line starting to the left of the main paragraph text, a heading having the specified width and more space above and below than a normal paragraph.

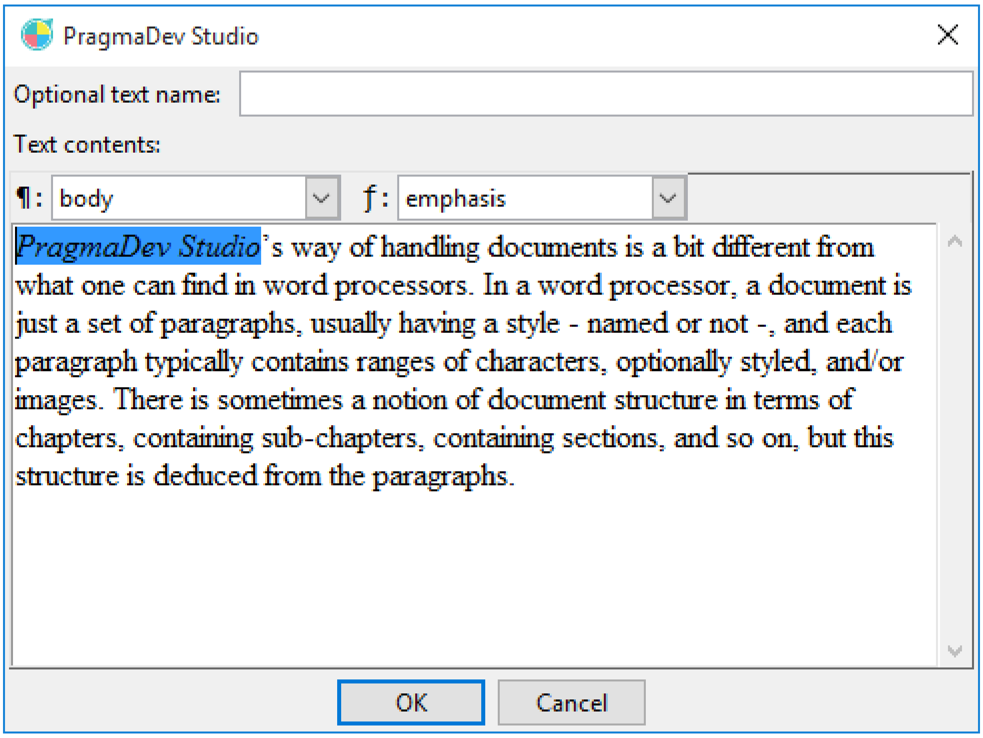
Note that actual space between 2 paragraphs is the maximum value of the space below set for the first one and the sapce above set for the second one.

- *Pagination options*’: these options control where the page breaks can and cannot happen.
  - ‘*Page break before*’: forces a page break at the beginning of all paragraph with this style. Note that this means the ‘*Space above*’ value is never used.
  - ‘*Keep with next*’: forces the paragraph to stay on the same page as the one following it if possible.
  - ‘*Keep with previous*’: forces the paragraph to stay on the same page as the one preceding it if possible.

- ‘*Widow/orphan lines*’: controls the number of lines in the paragraph that can end up alone at the top or bottom of a page. For example, if set to 2, if the paragraph ends up at the bottom of the page and there isn’t enough space to write 2 lines, the paragraph will be moved to the next page.

2.5.3.4 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:



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.

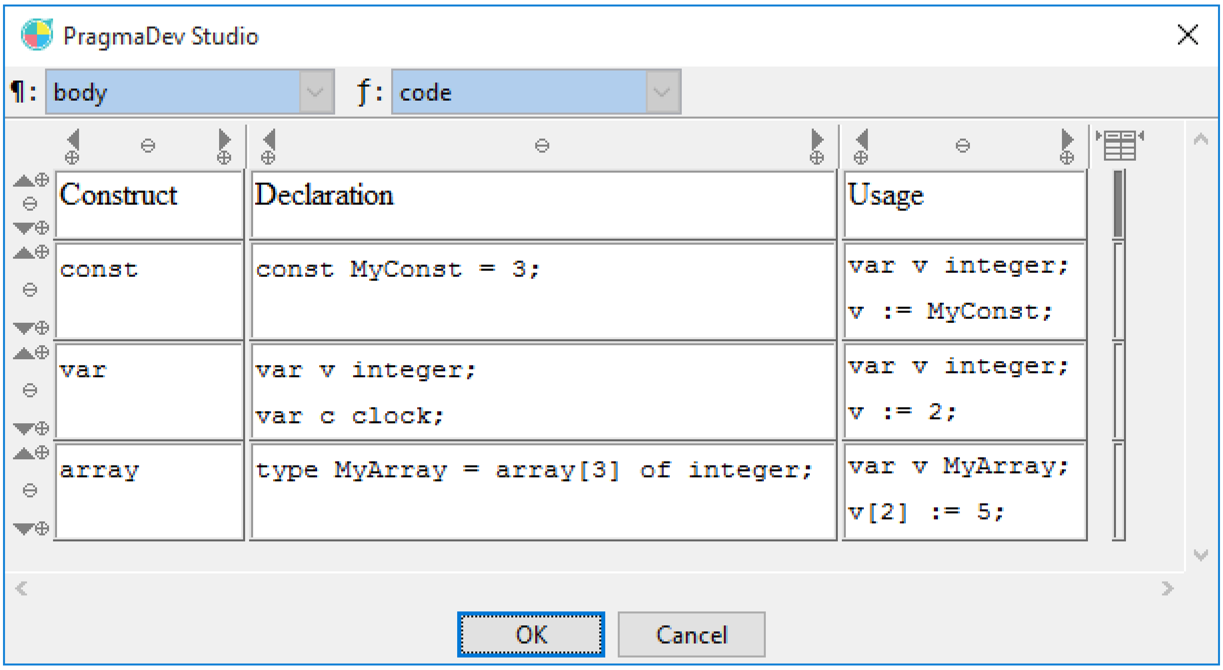
The bottom zone contains the actual text. The menus at the top allow to select:

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

Note that not all paragraph styles are available in all contexts: as said in “Documentation styles & options” on page 131, 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.

2.5.3.5 Table editor

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



¶: body   ƒ: code

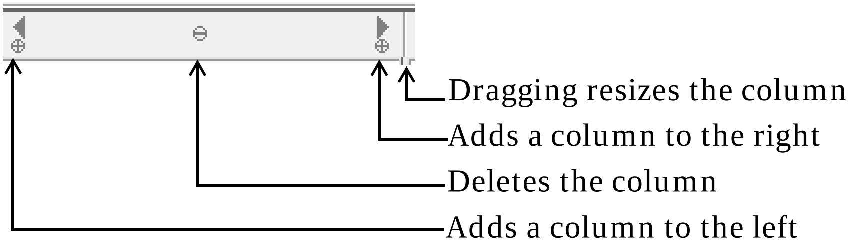
Construct	Declaration	Usage

const	const MyConst = 3;	var v integer;
		v := MyConst;

var	var v integer;
	var c clock;
		var v integer;
		v := 2;

array	type MyArray = array[3] of integer;	var v MyArray;
		v[2] := 5;

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:


Dragging resizes the column
Adds a column to the right
Deletes the column
Adds a column to the left

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.

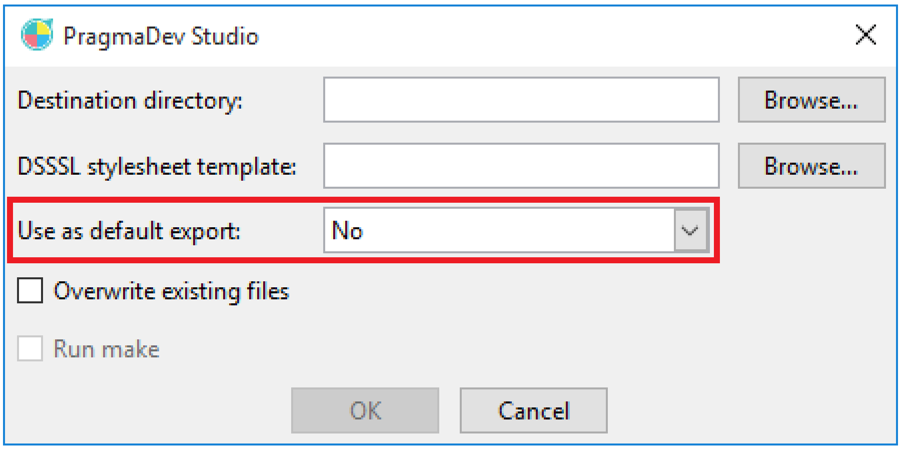
2.5.3.6 Exporting documents

Documents can be exported to 6 types of documents:

- Documents in PDF formats. This produces a directly readable and printable document. This kind of export is completely handled by PragmaDev Studio. Note however that this export format won't allow to integrate the exported document in a larger one, and that the output is hardly configurable at all.

- 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 LaTeX format, i.e. a text file containing standard LaTeX markup. This export can be "raw", i.e. contain only the body of the document, or use a template, which can specify the document header with its class, the packages it uses, and so on.
- 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:



PragmaDev Studio

Destination directory:

DSSSL stylesheet template:

Use as default export: No

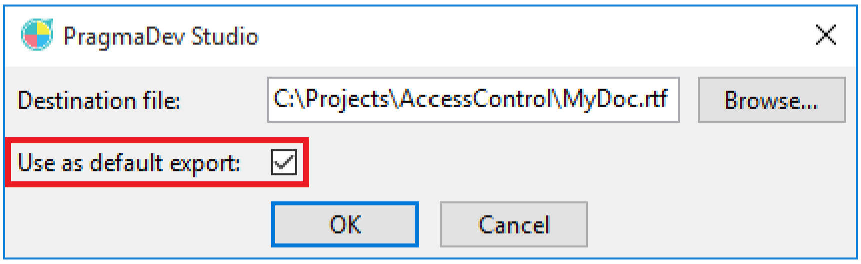
□ Overwrite existing files

□ Run make

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.

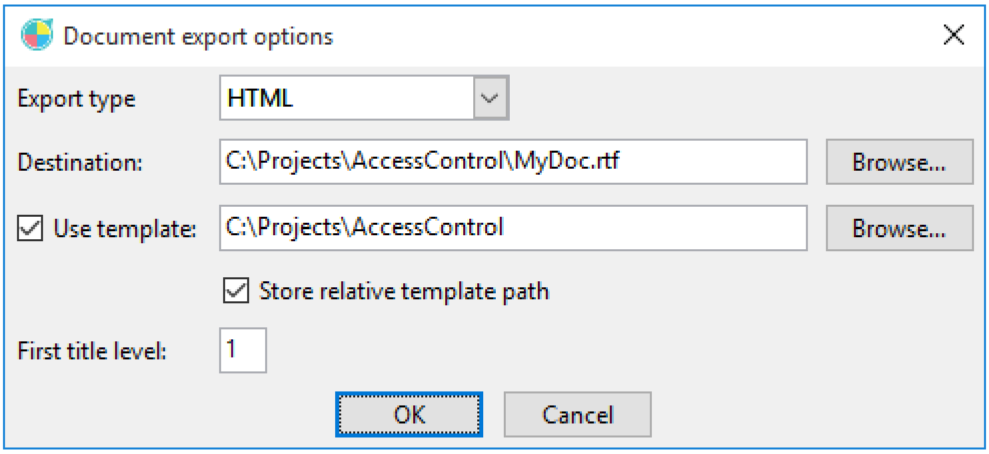
Note: this choice is only a checkbox for exports not supporting templates, such as RTF:



Destination file: C:\Projects\AccessControl\MyDoc.rtf

Use as default export: ☑

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



Export type: HTML

Destination: C:\Projects\AccessControl\MyDoc.rtf

☑ Use template: C:\Projects\AccessControl

☑ Store relative template path

First title level: 1

The available options are:

- *The export type*
  HTML, Open Document, RTF, PDF, LaTeX 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 or PDF exports, required for Open Document and SGML exports, and optional for HTML and LaTeX exports.
- *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.

2.5.3.6.1 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.

2.5.3.6.2 Exporting as PDF

Exporting as PDF is quite straightforward too: selecting "Export as", then "PDF" in the "Document" menu will just ask the file to export to, and export the PDF to the given file.

2.5.3.6.3 Exporting as OpenDocument format

As said in "Documentation styles & options" on page 131, 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.

2.5.3.6.4 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 %REFFILE 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.

2.5.3.6.5 Exporting as LaTeX

There are two types of LaTeX exports for PragmaDev Studio documents:

- The first export kind is the “raw” export, which is selected when no template is used. The exported file will only contain the body of the document without any header. This can be used in the document must be included in a bigger one.
- The second export kind uses a template. This template can be used to turn the export into a full LaTeX document by specifying a header with the document class, the used packages, and so on. The format for the template is described below.

A template for LaTeX export is itself a LaTeX file containing standardized comments, that must appear alone on a line. These comments are:

- %%%CONTENTS%%%
  This comment will be replaced by the actual contents of the document, which will contain only the markup for the document sections and text.
- %%%USEEPSIMAGES%%%
  If present, this comment has only an effect if the document includes external PNG images. In this case, if an image in /path/to/image.png is included in the document, and there is an encapsulated Postscript version of the image in /path/to/image.eps, then the generated LaTeX document will use the encapsulated Postscript version. If this line is not present, or if no EPS version of the image exists, the PNG is converted to encapsulated Postscript.

Note that the styles definition for the LaTeX export format are very partially used. The appearence of the exported document will depend mostly on the LaTeX document class and the packages it uses. The only impact of these styles will be the following:

- If a paragraph or character style uses an italic font, the text in the exported LaTeX document will be tagged with \textit{…};
- If a paragraph or character style uses a bold font, the text in the exported LaTeX document will be tagged with \textbf{…}, unless it appears in a section title;

- If a paragraph or character style uses a non-proportional font, i.e. a font having the same width for all characters, the text in the exported LaTeX document will be tagged with `\texttt{…}`. Non-proportional fonts are recognized by their family name. The following standard fonts are recognized as non-proportional:
  - Courier and its variants (Courier New or Courier-New);
  - DejaVu Sans Mono;
  - Bitstream Vera Sans Mono;
  - Lucida Console;
  - Typewriter;
  - computer-modern-typewriter.
- If a paragraph or character style uses a color for the text that is not black, the text in the exported LaTeX document will be tagged with `\textcolor[rgb]{…}{…}` to display it in the proper color.
- If a paragraph has an alignment that is not full justification, the corresponding LaTeX environment will be used wherever it is possible. For example, a centered paragraph will be enclosed with `\begin{center}` and `\end{center}`.

All other display attributes are ignored.

Heading styles are also mapped to the corresponding LaTeX command:

- 1: `\part{…}`
- 2: `\chapter{…}`
- 3: `\section{…}`
- 4: `\subsection{…}`
- 5: `\subsubsection{…}`
- 6: `\paragraph{…}`
- 7: `\subparagraph{…}`

Heading styles with a level greater than 7 cannot be used. For a document that should not include any parts, the first title level can be set to 2 in the document export options. If the exported LaTeX document is supposed to be an article which does not have the ‘chapter’ level, the first title level can be set to 3.

2.5.3.7 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.

2.5.3.7.1 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.

2.5.3.7.2 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.

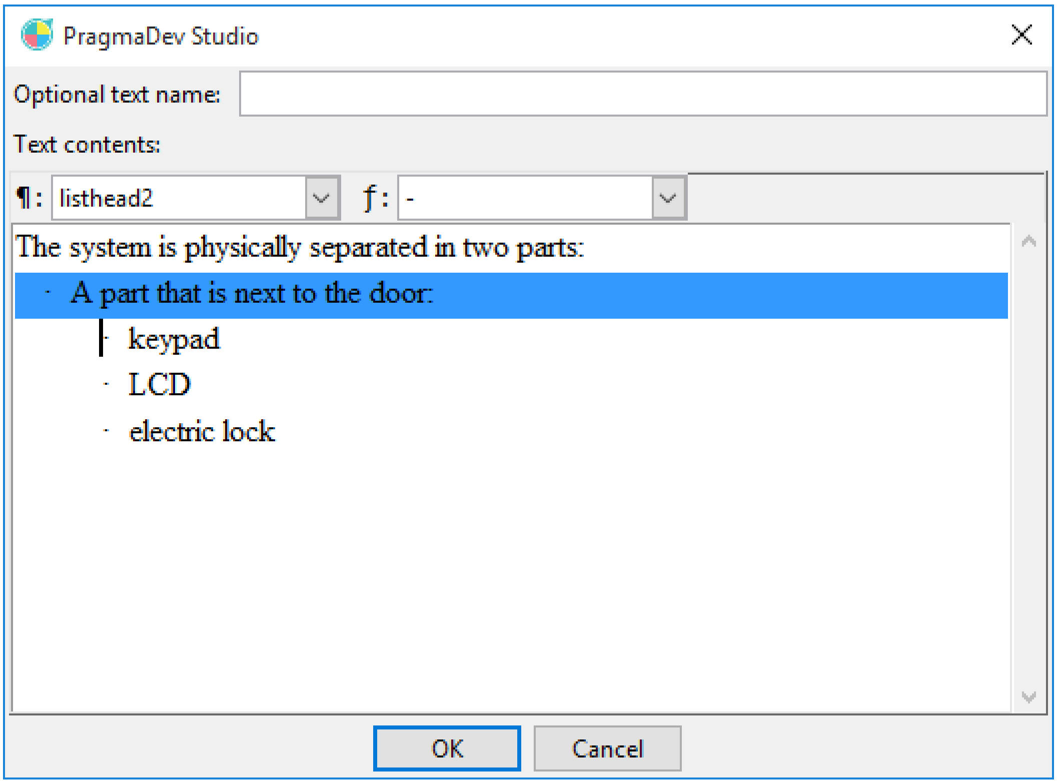
Note: Once created, the master document will include a copy of all character and paragraph styles that were in the document exported by PragmaDev Studio when it was inserted. Any following changes will only have an impact in the exported document, and *not* in the master document including it.

2.5.3.8 Questions and answers

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

- **How can I create a “bold” or “italic” character style as in a word processor?**
  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.
- **Several paragraph styles I’ve defined in the documentation display options do not appear in the styled text editors. Why?**
  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.
- **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!**
  As said in “Documentation styles & options” on page 131, 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:



The system is physically separated in two parts:
- A part that is next to the door:
  - keypad
  - LCD
  - electric lock

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.

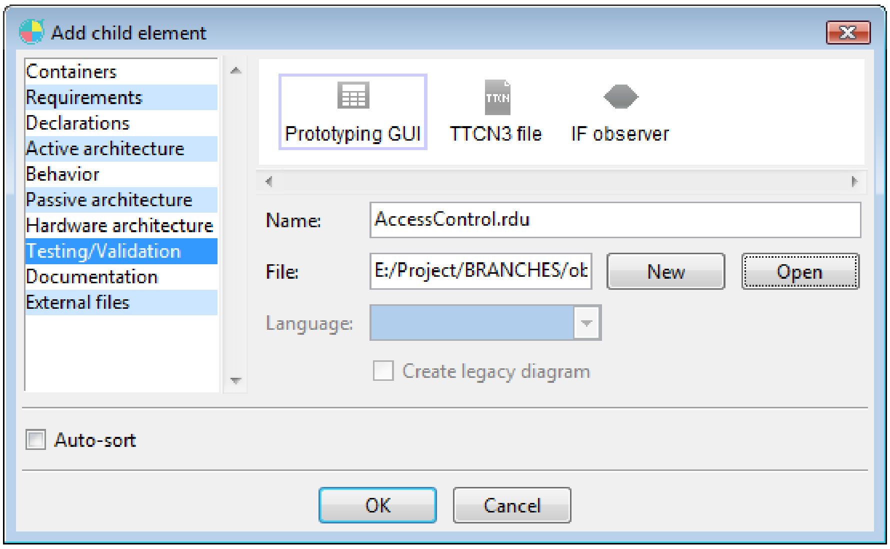
To be able to delete these paragraphs, you first have to change the styles for the paragraphs after the selection or before it to ensure that list nesting will be consistent after the deletion.

2.6 - 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.

2.6.1 Prototyping GUI editor

Insert a *Prototyping GUI* element in the project:



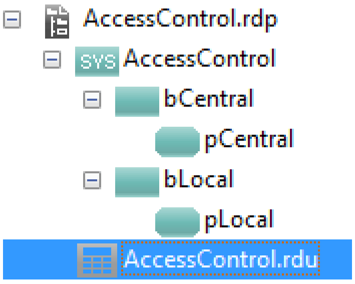
Name: AccessControl.rdu

File: E:/Project/BRANCHES/ob

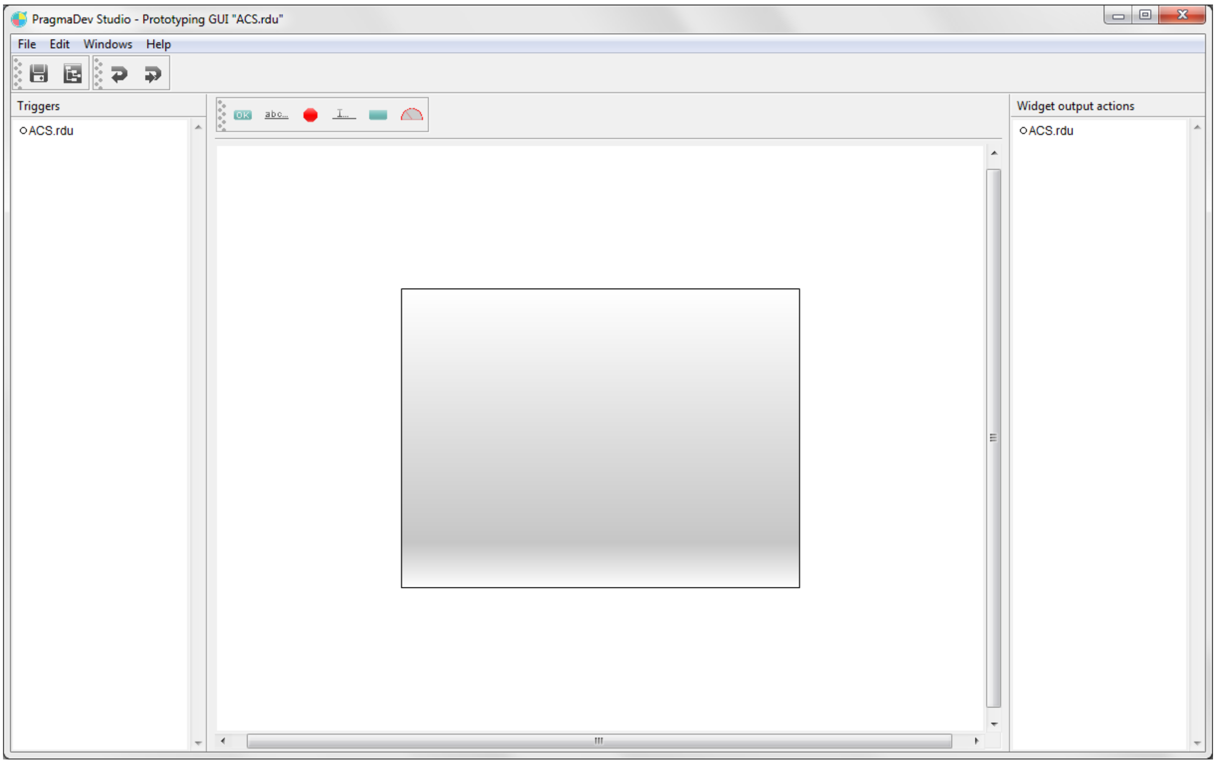
Language:

□ Create legacy diagram

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



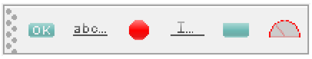
Double-clicking on this component will open a new empty GUI editor:



The editor is divided in three parts:

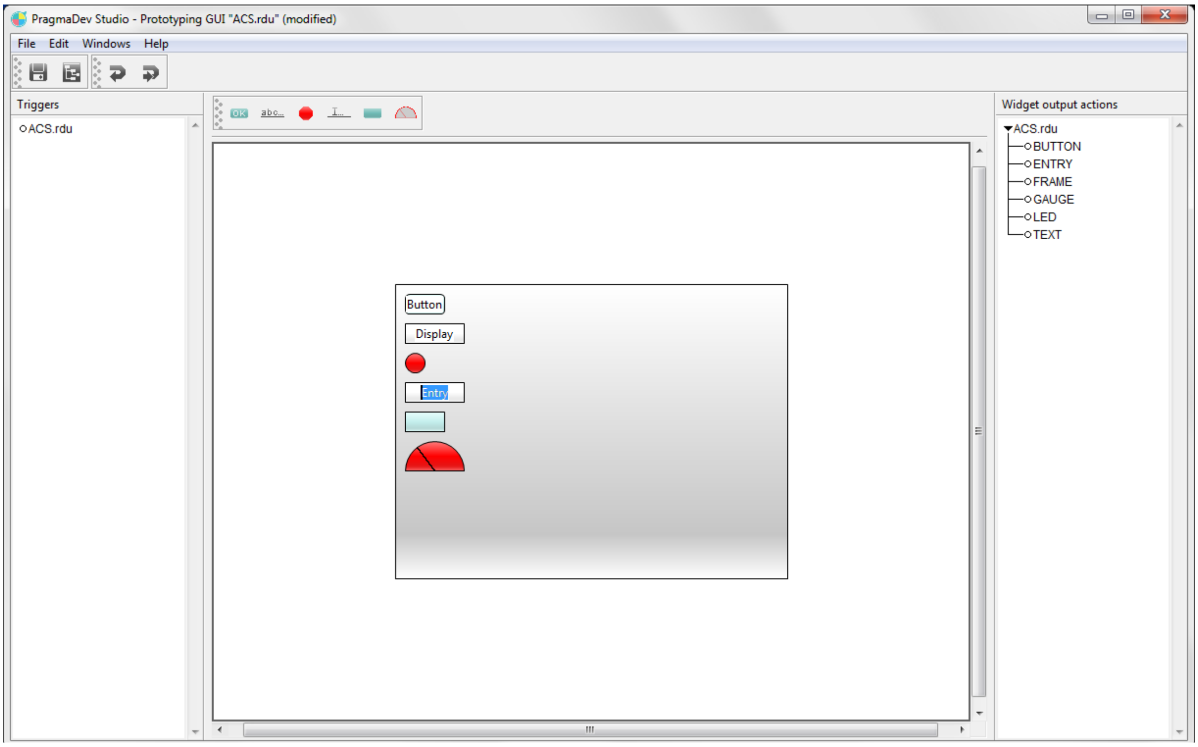
- The trigger tree on the left to describe which events will trigger actions on the display.
- The graphical layout in the middle to design what the GUI will look like.
- The output tree on the right to describe the actions to be executed when the GUI is stimulated.

In the middle area six types of graphical elements can be inserted: buttons, text displays, and LEDs, text inputs, frames, and gauges:

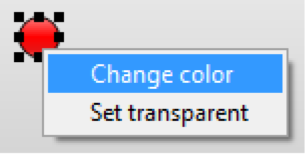


Each time one of these elements is inserted, a corresponding tree item is added in the right area.

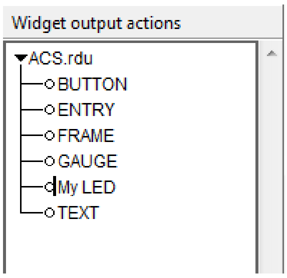
Let’s consider for example we are adding one of each widget in the display:



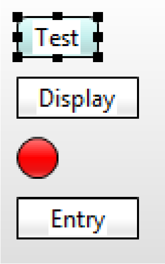
Right click on the widget to change its color:



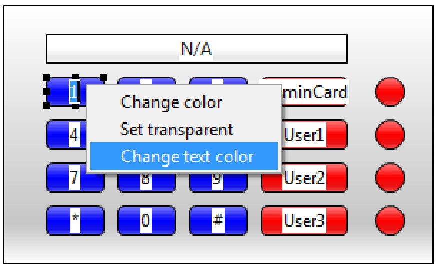
Double-click on the name in the tree to edit the name of the widget:



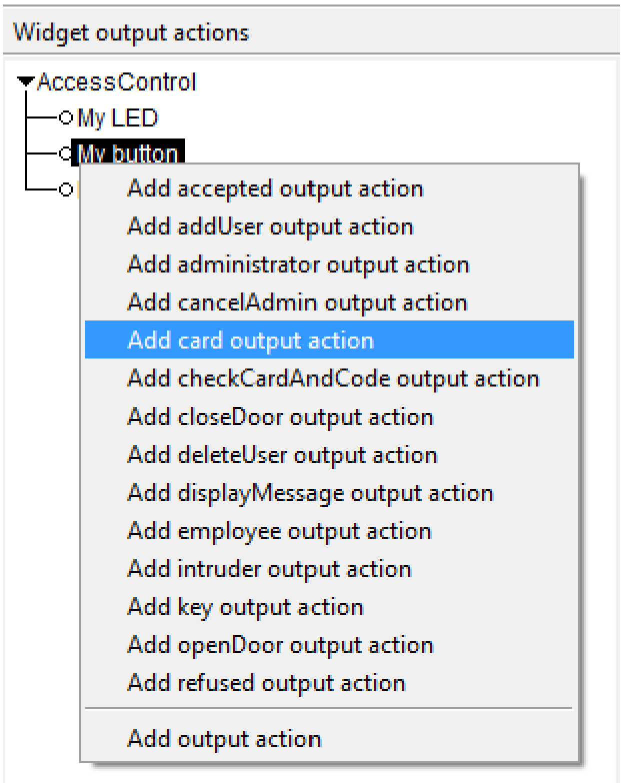
Click on the widget text to edit it:



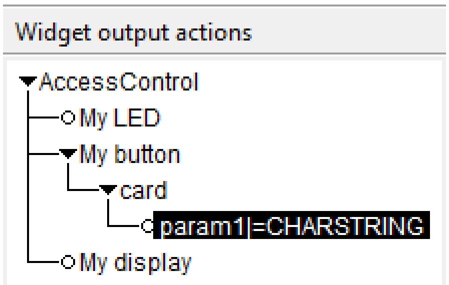
Right click on widget containing text to change the text color:



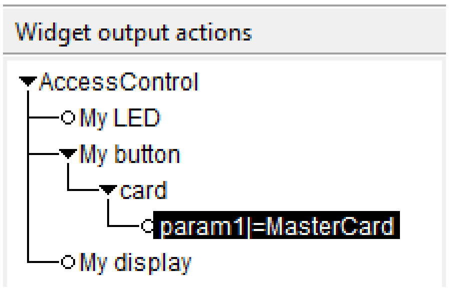
A text display and a LED generally do not generate any output action. Let's add an action to the Admin button: right click on the corresponding element in the output tree and all the available message in the system will be displayed:


Widget output actions
AccessControl
My LED
My button
Add accepted output action
Add addUser output action
Add administrator output action
Add cancelAdmin output action
Add card output action
Add checkCardAndCode output action
Add closeDoor output action
Add deleteUser output action
Add displayMessage output action
Add employee output action
Add intruder output action
Add key output action
Add openDoor output action
Add refused output action
Add output action

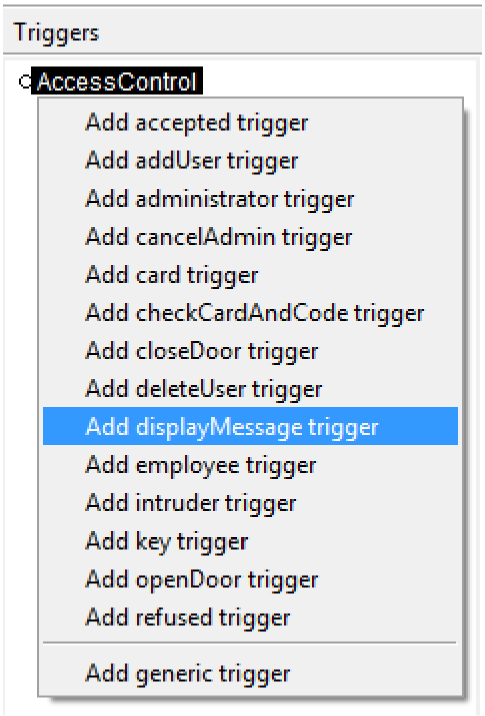
In this example, we are using the AccessControl SDL Z.100 example. Let's say the 'card' message will be sent to the system when the user clicks on 'My button'. The parameters associated with the message will be automatically displayed and the expected type is diplayed to ease editing:



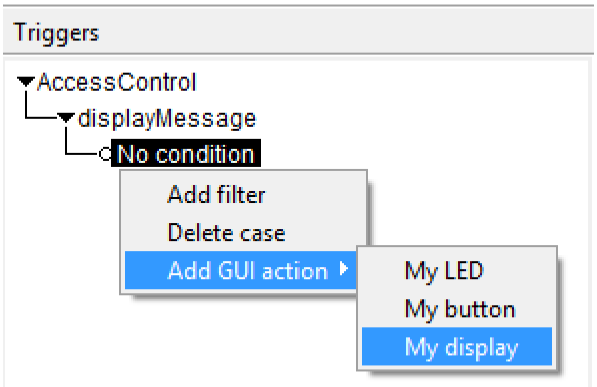
In that example the administrator card should have 'MasterCard' as its parameter. Double click on the type and enter the desired parameter value:



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:

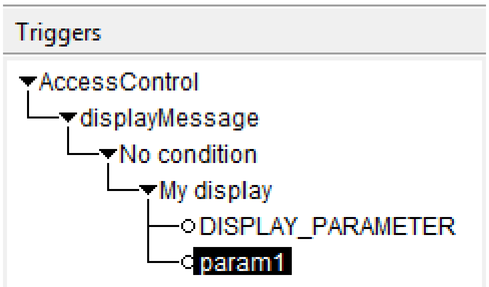


In the new element, it is now possible to add cases. Each case is a set of filters to be verified and actions to do when the filters are verified. In our case, we will not have any filter since we want to display the parameter value all the time.



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:

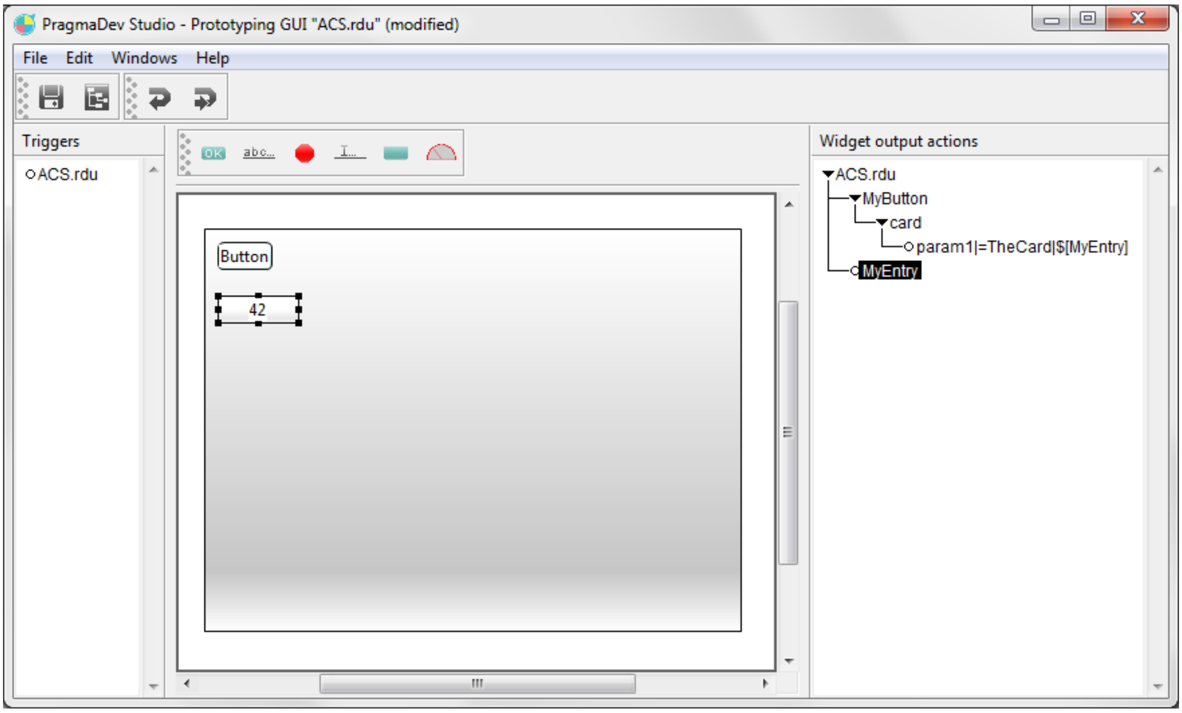


We could also use the DISPLAY type of action to display a specific text that has nothing to do with the parameters value.

To access an Entry value use the following syntax:

```
        |$[<entry name>]
```

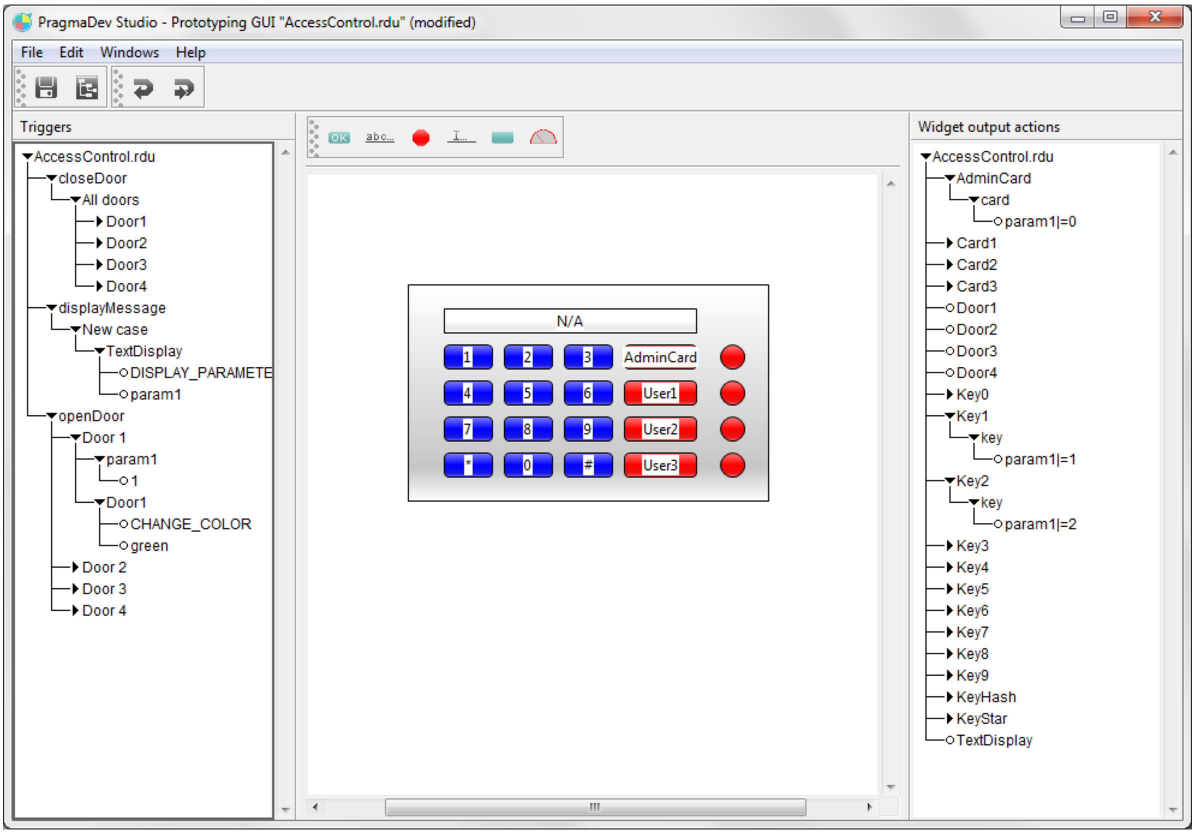
In the example below:



param1|=TheCard|$[MyEntry]

Pressing *MyButton* will send the message *card* and the value of its first parameter will be the concatenation of the string *TheCard* and of the value in the entry *MyEntry*. For example *TheCard42*.

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:

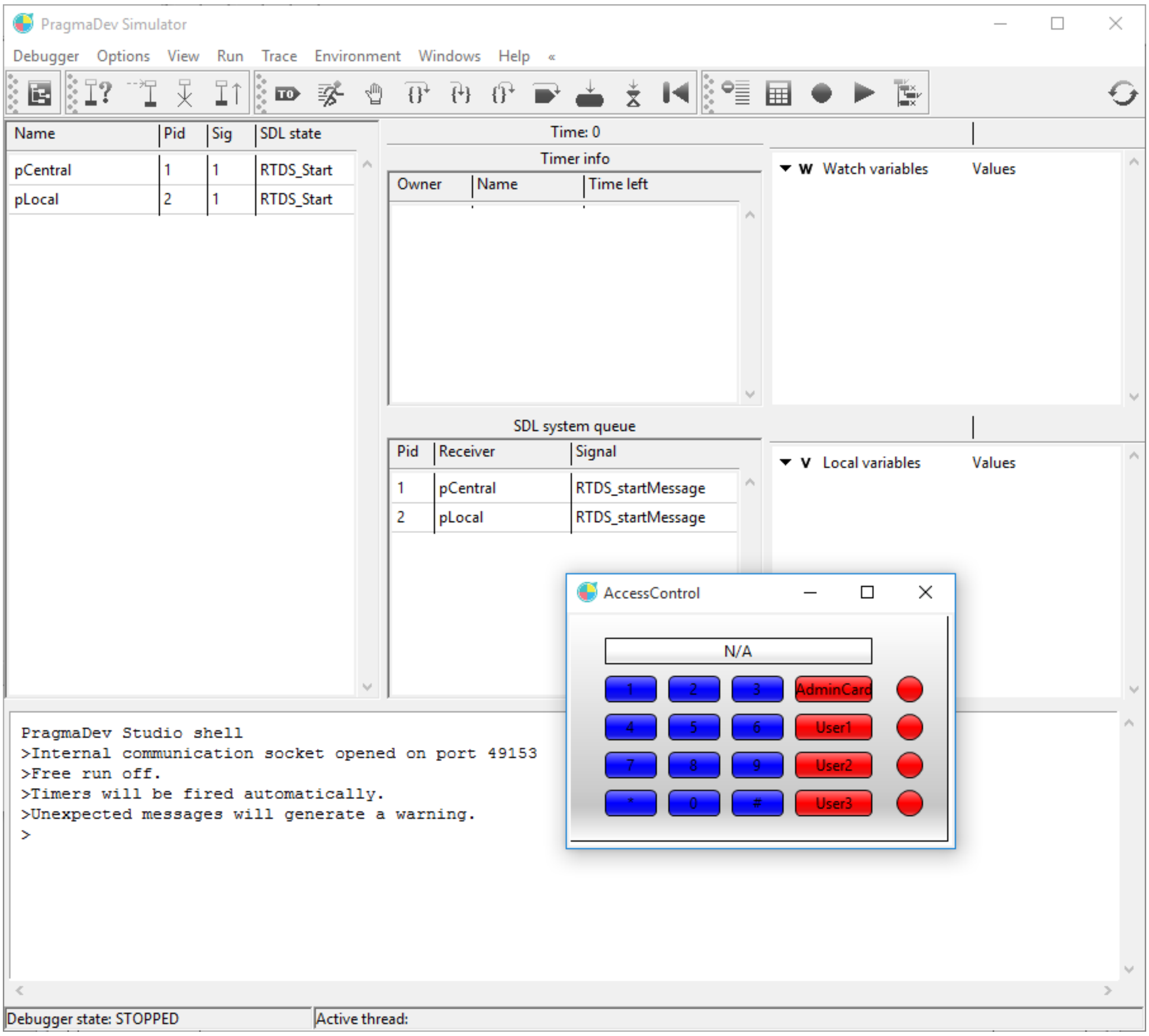
- `Key1`
  When this button is pressed, the `key` message is sent to the system with its first parameter set to 1
- `Key2`
  When this button is pressed, the `key` message is sent to the system with its first parameter set to 2
- `AdminCard`

When this button is pressed, the `card` message is sent to the system with its first parameters set to `MasterCard`.

2.6.2 Prototyping GUI runner

2.6.2.1 Within the simulator or debugger

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



Name	Pid	Sig	SDL state
pCentral	1	1	RTDS_Start
pLocal	2	1	RTDS_Start

Time: 0

Timer info

Owner	Name	Time left

SDL system queue

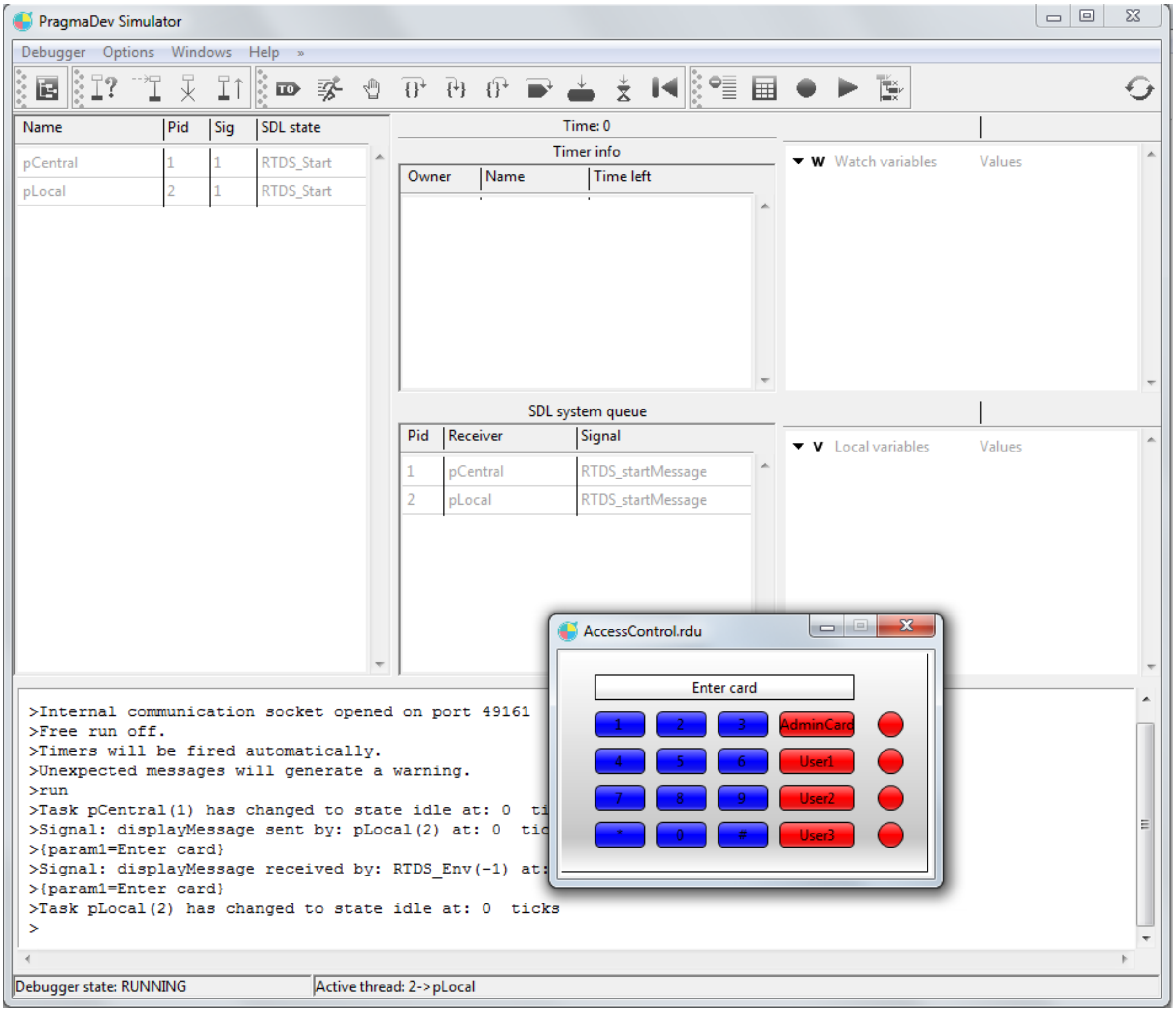
Pid	Receiver	Signal
1	pCentral	RTDS_startMessage
2	pLocal	RTDS_startMessage

W Watch variables	Values

V Local variables	Values

```
PragmaDev Studio shell
>Internal communication socket opened on port 49153
>Free run off.
>Timers will be fired automatically.
>Unexpected messages will generate a warning.
>
```

Debugger state: STOPPED | Active thread:



Name	Pid	Sig	SDL state
pCentral	1	1	RTDS_Start
pLocal	2	1	RTDS_Start

Owner	Name	Time left

SDL system queue

Pid	Receiver	Signal
1	pCentral	RTDS_startMessage
2	pLocal	RTDS_startMessage

```
>Internal communication socket opened on port 49161
>Free run off.
>Timers will be fired automatically.
>Unexpected messages will generate a warning.
>run
>Task pCentral(1) has changed to state idle at: 0  ti
>Signal: displayMessage sent by: pLocal(2) at: 0  tic
>{param1=Enter card}
>Signal: displayMessage received by: RTDS_Env(-1) at:
>{param1=Enter card}
>Task pLocal(2) has changed to state idle at: 0  ticks
>
```

Debugger state: RUNNING | Active thread: 2->pLocal

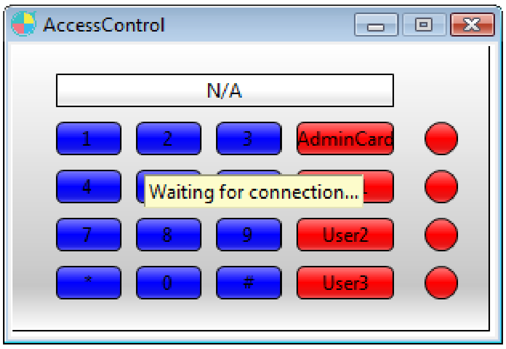
The GUI will connect automatically to the running system. That’s it.

2.6.2.2 Standalone prototyping GUI with external executable

If an executable is generated by PragmaDev Developer or PragmaDev Studio with standalone prototyping GUI runner support (see “Debug and trace options” on page 252), the prototyping GUI can be run by itself:

- Build the executable for your system with the appropriate profile.

- Once the executable is built, select the prototyping GUI in the project and click the quickbutton in the toolbar. The prototyping GUI runner appears, displaying a ‘Waiting for connection’ message:



Waiting for connection...

- Run the generated executable. It will connect to the prototyping GUI running in PragmaDev Studio and will allow you to interact with it just as you would do within the debugger or simulator.

Also note that when building with standalone protoyping GUI support, a file with the name of the executable and the extension .rdy will be generated in the code generation directory. This file can be distributed to anyone having a PragmaDev Studio installation, even in free mode, and running the file will run the prototyping GUI and the generated executable, allowing anyone to test it.

2.7 - Code coverage results

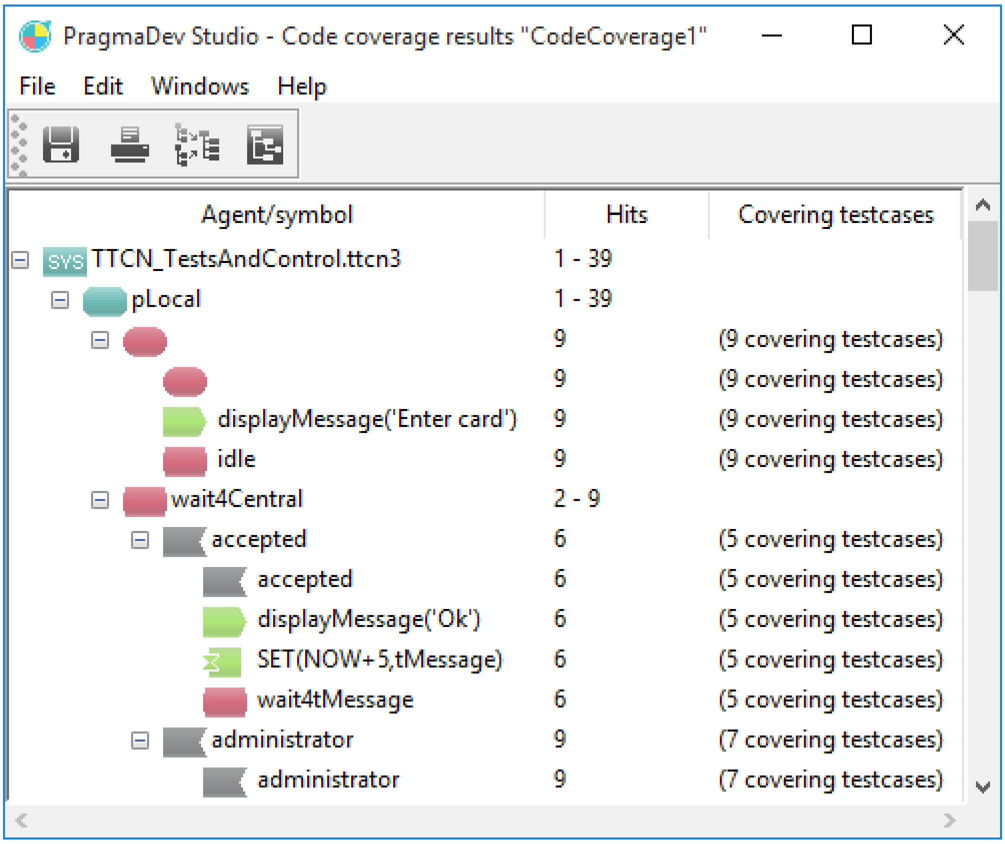
2.7.1 Generating code coverage results

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 “Model coverage” on page 336 and “Model coverage” on page 214). 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 248 and “Main simulator options” on page 187.

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 windows is closed without saving, the code coverage results node will disappear from the project.

2.7.2 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 171. 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.

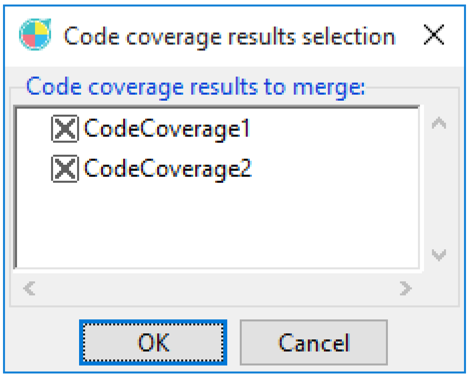
The tree may be expanded, collapsed and sorted using the “Edit” menu. Double-clicking on a node will display the corresponding symbol if any.

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.

2.7.3 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:


Code coverage results selection

Code coverage results to merge:

☒ CodeCoverage1

☒ CodeCoverage2

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.

2.8 - Requirements Table Editor

2.8.1 Principles

There are two ways of creating requirements in PragmaDev Studio:

- A requirement table can be created and filled in the application.
- All requirements can 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. This is probably the general usage, as requirements are often defined outside of the application.

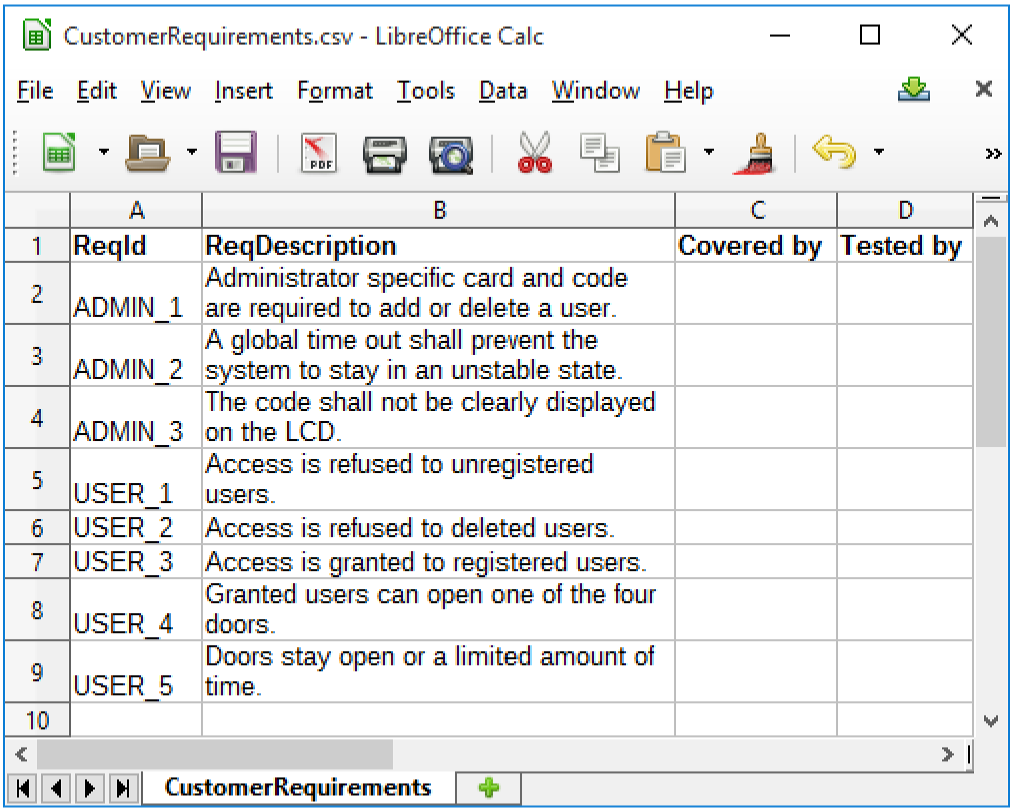
The input formats for requirement tables supported by PragmaDev Studio are the text formats supported by Microsoft Excel:

- 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 seperated 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.

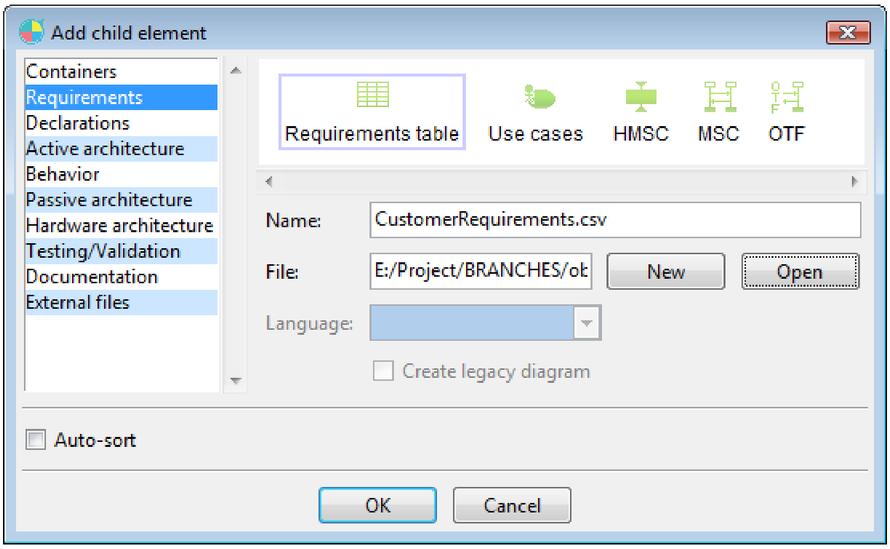
The requirements table must have at least 2 columns and at most 4. The first row is considered by PragmaDev Studio as the title row, all others describing one requirement. The first column is the requirement identifier, and the second one its description. Only the title is considered for the 2 last columns: PragmaDev Studio will use them to store its own information: the symbols and testcases covering the requirement; see “Covering symbols” on page 168 and “Covering testcases” on page 170.

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


CustomerRequirements.csv - LibreOffice Calc
File Edit View Insert Format Tools Data Window Help
CustomerRequirements

	A	B	C	D
1	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:



Name: CustomerRequirements.csv

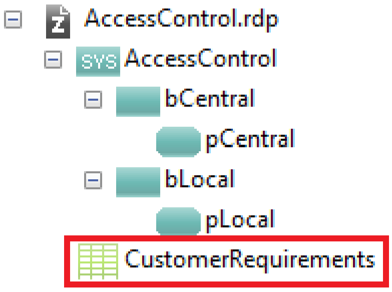
File: E:/Project/BRANCHES/ol

Language:

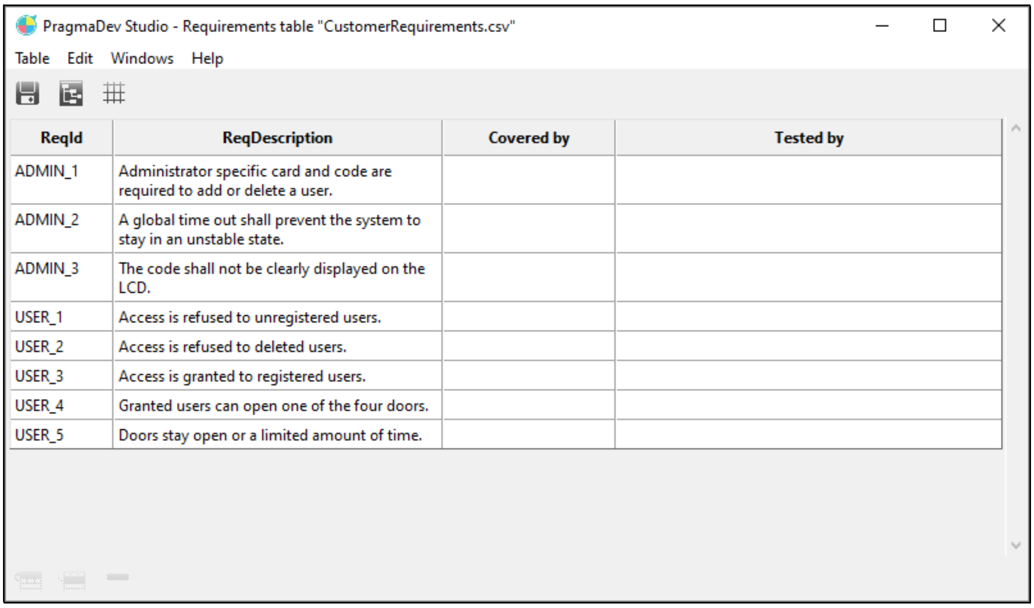
□ Create legacy diagram

□ Auto-sort

Once added, the table will appear in the project:



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

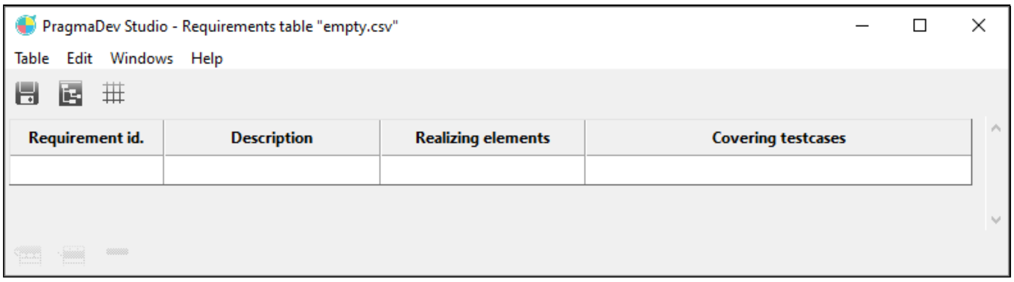

PragmaDev Studio - Requirements table "CustomerRequirements.csv"
Table
Edit
Windows
Help

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.		

Requirements can also beadded to or deleted from the table within the editor. When a row is selected, this is done via the buttons at the bottom of the editor window, or via the entries in the “Edit” menu:

- Clicking on ▤, selecting “Insert row above” or pressing Control+R will insert a requirement row above the selected one.
- Clicking on ▤, selecting “Insert row below” or pressing Control+Shift+R will insert a requirement row below the selected one.
- Clicking on ▤, selecting “Delete row” or pressing Control+Delete will delete the selected row.

This can be used to create a requirements table from scratch: if the CSV file selected when inserting the table in the project does not exist, opening the table will actually always display a single empty row:



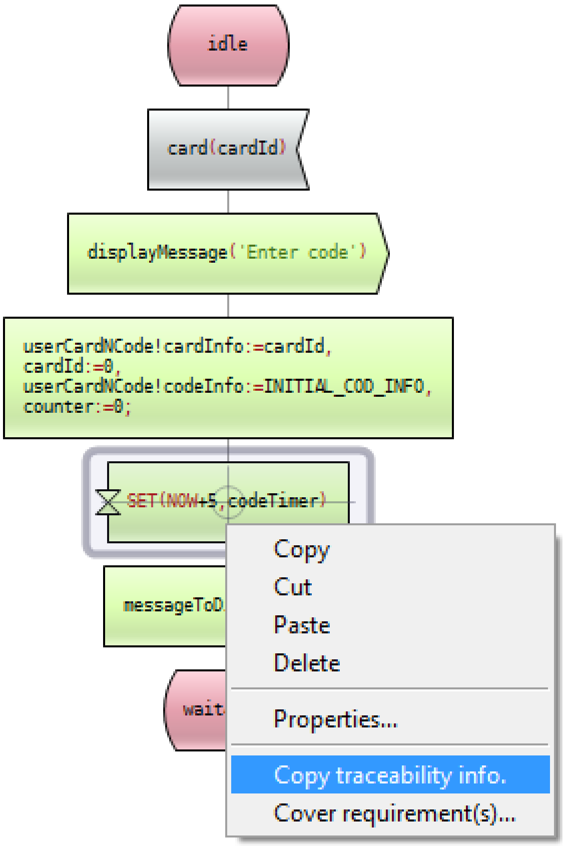
Requirement id.	Description	Realizing elements	Covering testcases

Double-clicking on a “Requirement id.” or “Description” cell will open it for edition and allow to specify the identifier and the description for the requirement. Adding new requirement rows allows to create the table completely.

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.

2.8.2 Covering symbols

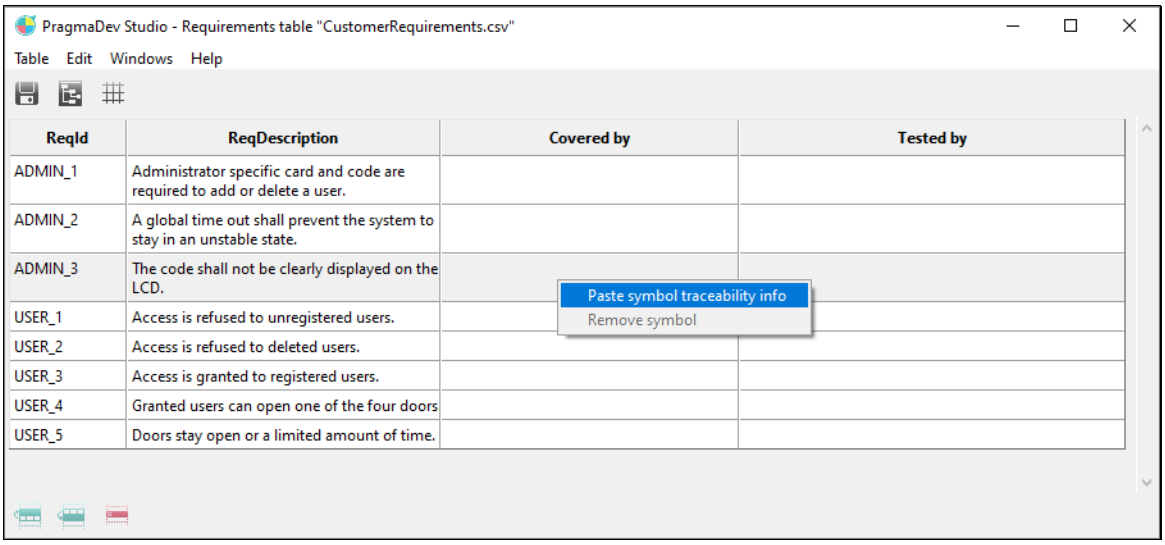
Specifying covering symbols for requirements is done via a copy/paste operation: the symbol must be selected in its parent diagram, its traceability information copied, and then pasted in the requirements table. Here are the steps in detail:



```
userCardNCode!cardInfo:=cardId,
cardId:=0,
userCardNCode!codeInfo:=INITIAL_COD_INFO,
counter:=0;
```

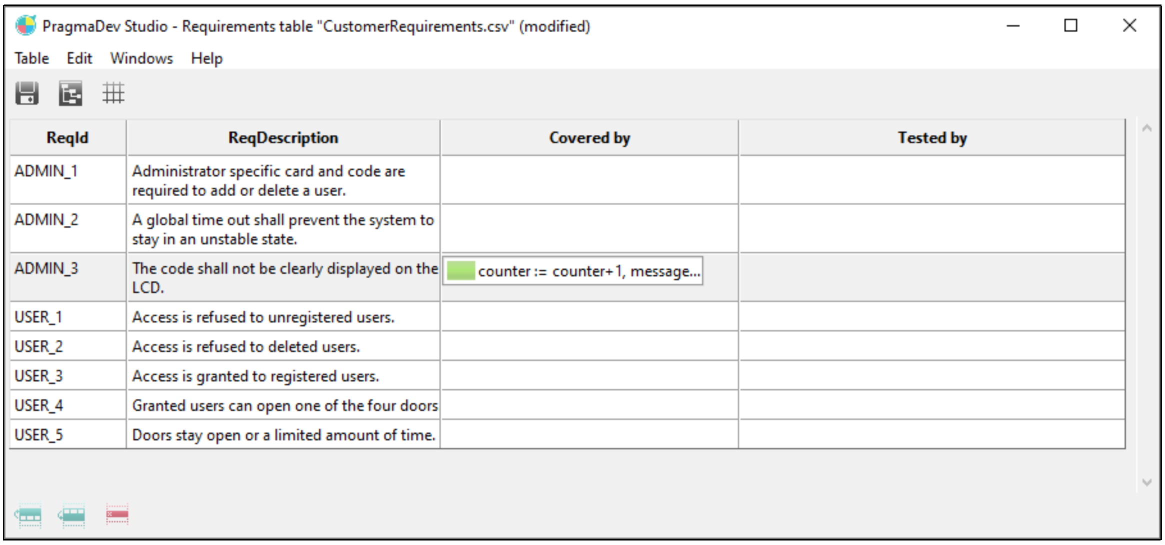
In an editor diagram, when a symbol is selected, an item “Copy traceability info.” is available in the contextual menu as well as in the “Edit” menu. Note that the traceability information for a symbol has no actual representation and cannot be pasted anywhere else than in a requirements table. The paste 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.csv"
Table
Edit
Windows
Help
Paste symbol traceability info
Remove symbol

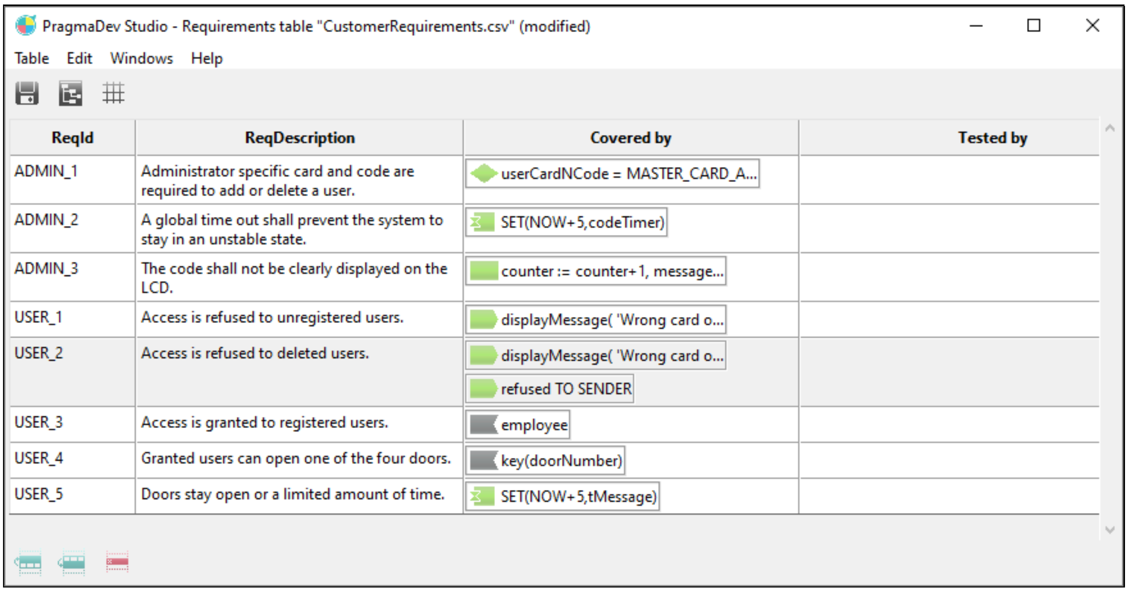
ReqId	ReqDescription	Covered by	Tested by
ADMIN_1	Administrator specific card and code are required to add or delete a user.		
ADMIN_2	A global time out shall prevent the system to stay in an unstable state.		
ADMIN_3	The code shall not be clearly displayed on the LCD.		
USER_1	Access is refused to unregistered 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.		

Once pasted, the covering symbol will appear as an icon giving its type followed by the first characters in its text:


PragmaDev Studio - Requirements table "CustomerRequirements.csv" (modified)
Table
Edit
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.	🟩 counter := counter+1, message...	
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, all the symbols will all appear in the third table column:

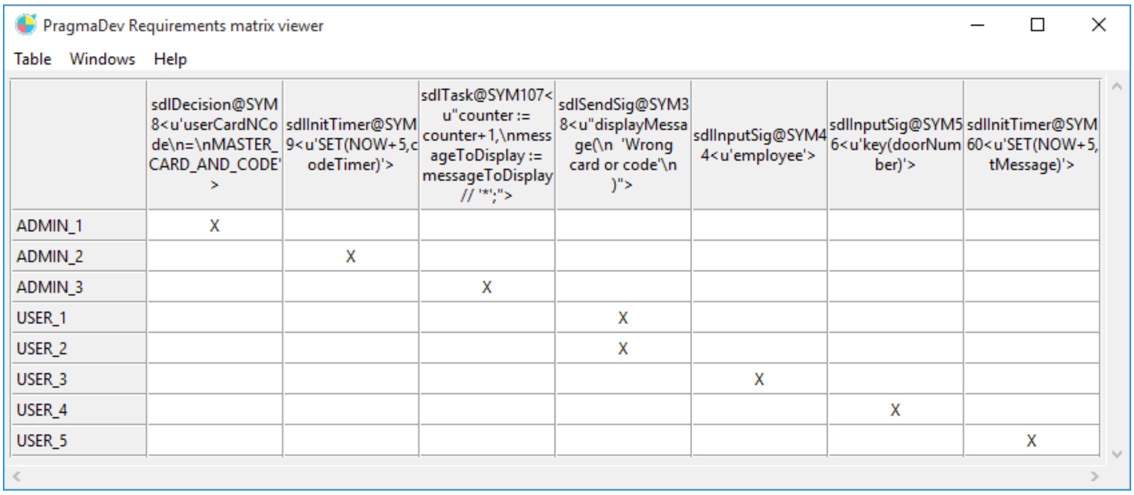


PragmaDev Studio - Requirements table "CustomerRequirements.csv" (modified)
Table Edit Windows Help

ReqId	ReqDescription	Covered by	Tested by
ADMIN_1	Administrator specific card and code are required to add or delete a user.	userCardNCode = MASTER_CARD_A...	
ADMIN_2	A global time out shall prevent the system to stay in an unstable state.	SET(NOW+5,codeTimer)	
ADMIN_3	The code shall not be clearly displayed on the LCD.	counter := counter+1, message...	
USER_1	Access is refused to unregistered users.	displayMessage('Wrong card o...	
USER_2	Access is refused to deleted users.	displayMessage('Wrong card o... refused TO SENDER	
USER_3	Access is granted to registered users.	employee	
USER_4	Granted users can open one of the four doors.	key(doorNumber)	
USER_5	Doors stay open or a limited amount of time.	SET(NOW+5,tMessage)	

Double-clicking on one of the symbol representations in the requirements table will open its parent diagram and select the symbol.

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.

2.8.3 Covering testcases

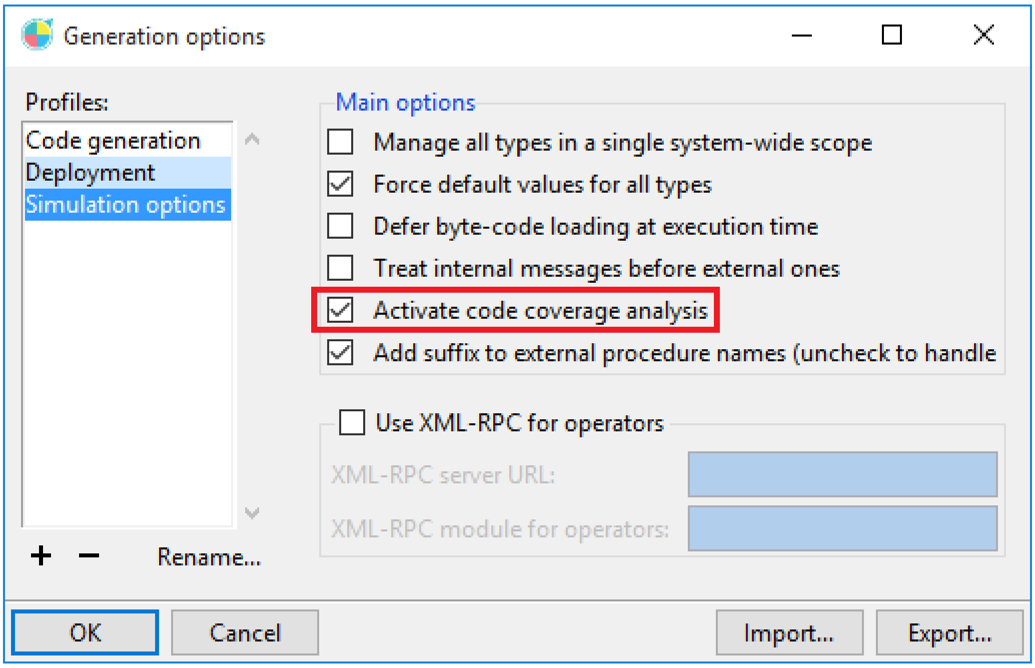
Once the symbols covering the requirements have been specified in the requirements table, PragmaDev Studio can automatically extract the information of the covering testcases from a cosimulation of SDL and TTCN. This extraction is done in 2 steps:

- 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 161. 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.

2.8.3.1 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:



Main options

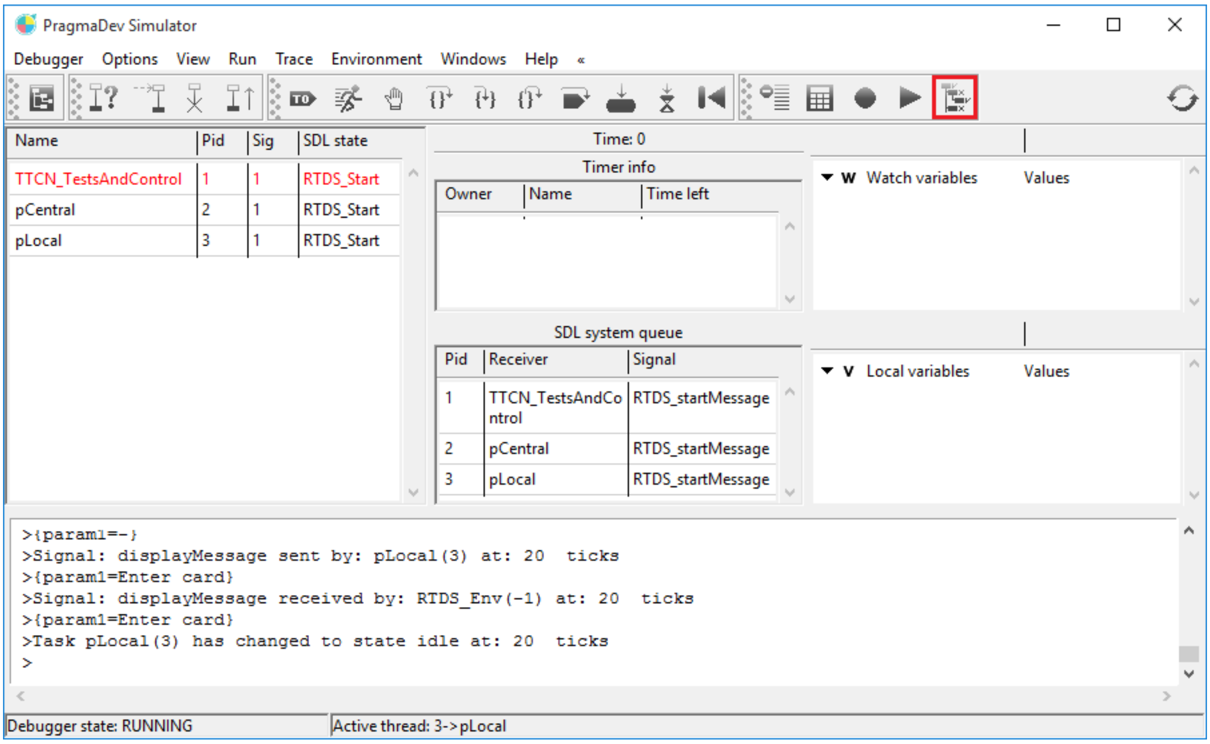
☐ Manage all types in a single system-wide scope
☑ Force default values for all types
☐ Defer byte-code loading at execution time
☐ Treat internal messages before external ones
☑ Activate code coverage analysis
☑ Add suffix to external procedure names (uncheck to handle

☐ Use XML-RPC for operators

XML-RPC server URL:

XML-RPC module for operators:

Then, the co-simulation of TTCN and SDL must be launched as explained in “TTCN-3 co-simulation” on page 343: 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 346. At the end of the simulation session, the code coverage can be extracted by using the button:



Name	Pid	Sig	SDL state
TTCN_TestsAndControl	1	1	RTDS_Start
pCentral	2	1	RTDS_Start
pLocal	3	1	RTDS_Start

Owner	Name	Time left

SDL system queue

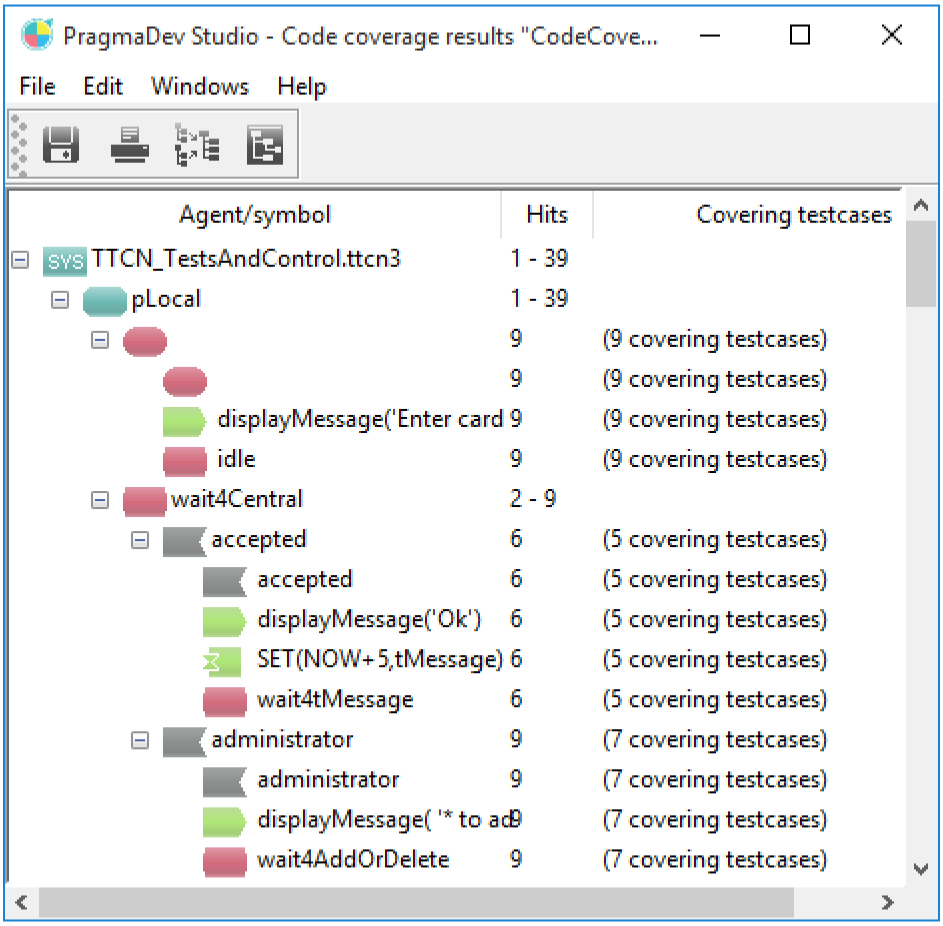
Pid	Receiver	Signal
1	TTCN_TestsAndControl	RTDS_startMessage
2	pCentral	RTDS_startMessage
3	pLocal	RTDS_startMessage

```
>{param1=-}
>Signal: displayMessage sent by: pLocal(3) at: 20  ticks
>{param1=Enter card}
>Signal: displayMessage received by: RTDS_Env(-1) at: 20  ticks
>{param1=Enter card}
>Task pLocal(3) has changed to state idle at: 20  ticks
>
```

Debugger state: RUNNING

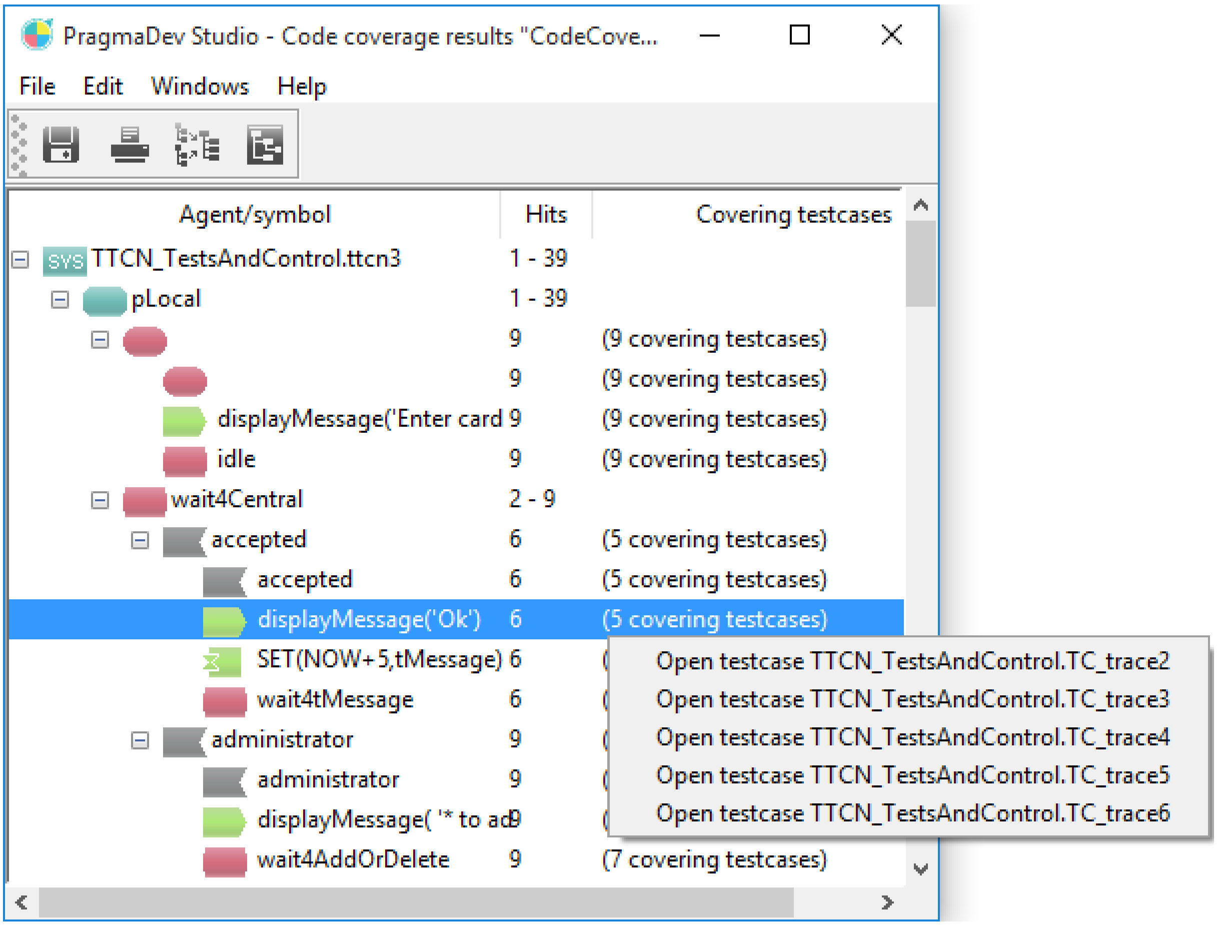
Active thread: 3->pLocal

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	
	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)
displayMessage('* to ad	9	(7 covering testcases)
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:


PragmaDev Studio - Code coverage results "CodeCove...
File
Edit
Windows
Help
Open testcase TTCN_TestsAndControl.TC_trace2
Open testcase TTCN_TestsAndControl.TC_trace3
Open testcase TTCN_TestsAndControl.TC_trace4
Open testcase TTCN_TestsAndControl.TC_trace5
Open testcase TTCN_TestsAndControl.TC_trace6

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	(
wait4tMessage	6	(
administrator	9	(
administrator	9	(
displayMessage('* to ad	9	(
wait4AddOrDelete	9	(7 covering testcases)

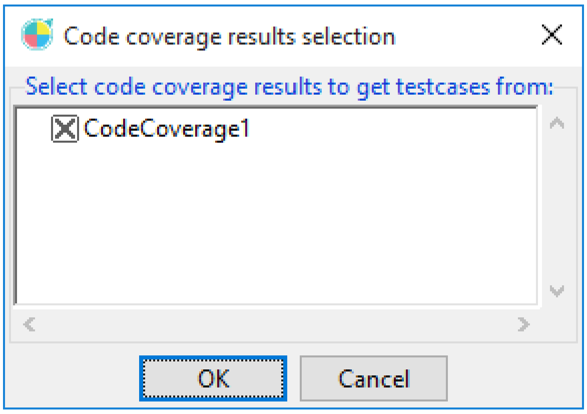
The contextual menu displays the names of the testcases that actually covered the symbols. Selecting one of the items will open its parent module in the text file editor, and position the insertion point directly on the declaration of the testcase.

2.8.3.2 Importing testcase coverage information in requirements

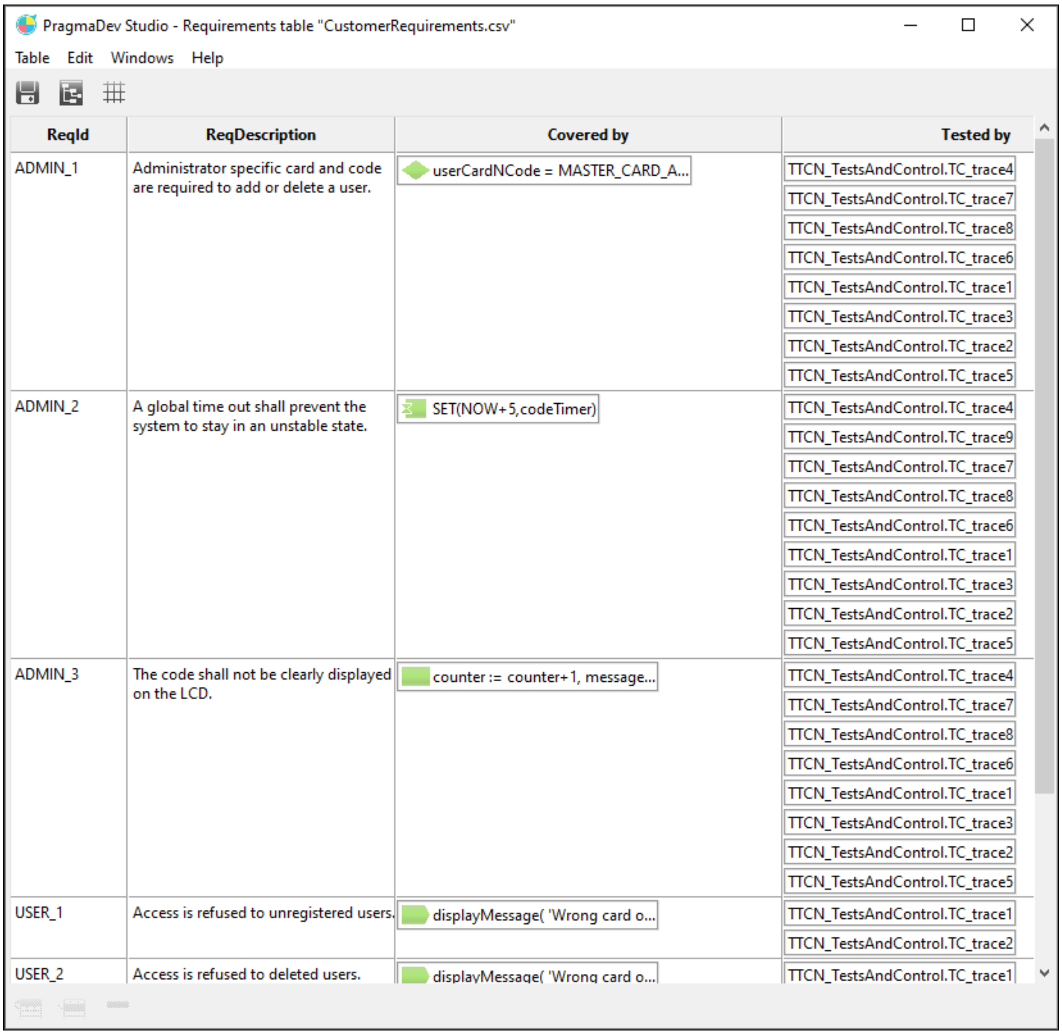
To be able to import testcase coverage information in a requirements table, the following prerequisites must be met:

- The requirements table must include symbols covering the requirements, entered as explained in “Covering symbols” on page 168.
- 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 171.

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:


Code coverage results selection
Select code coverage results to get testcases from:
☒ CodeCoverage1
OK
Cancel

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:


PragmaDev Studio - Requirements table "CustomerRequirements.csv"
Table
Edit
Windows
Help

ReqId	ReqDescription	Covered by	Tested by
ADMIN_1	Administrator specific card and code are required to add or delete a user.	userCardNCode = MASTER_CARD_A...	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.	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.	counter := counter+1, message...	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
USER_1	Access is refused to unregistered users.	displayMessage('Wrong card o...	TTCN_TestsAndControl.TC_trace1 TTCN_TestsAndControl.TC_trace2
USER_2	Access is refused to deleted users.	displayMessage('Wrong card o...	TTCN_TestsAndControl.TC_trace1

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.

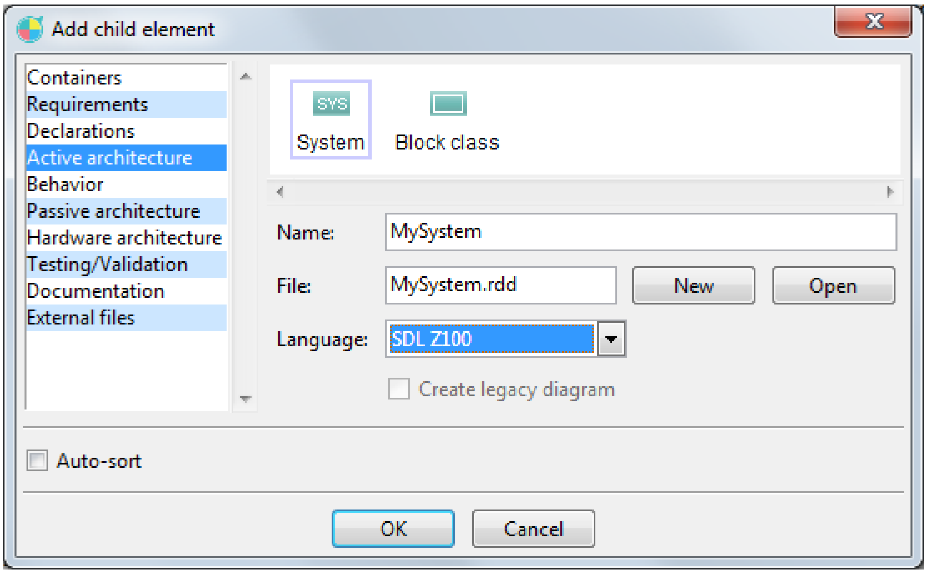
Note that these testcases are not modifiable, except by clearing all of them (item “Clear covering testcases...” in the “Table” menu), or by importing coverage information from another results set, which will replace the current covering information.

3 - PragmaDev Specifier

3.1 - SDL Z.100 project

PragmaDev Specifier helps system engineers to unambiguously specify and verify the functionalities of the system, and define the best architecture for performance or energy efficiency. The technology used results in a graphical and executable model. Verification and validation of the dynamic of the system is done with the integrated simulator, and the best architecture is analyzed with a unique performance analyzer.

PragmaDev Specifier is based on SDL models. For information on the language itself, please refer to the language reference documents. PragmaDev projects are generic. In order to work with PragmaDev Specifier an SDL system must be created. For that matter start a new project, right click on the project and select addc hild element. Select active architecture and system in the window:



Name: MySystem

File: MySystem.rdd

Language: SDL Z100

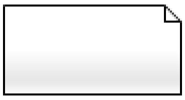
□ Create legacy diagram

□ Auto-sort

3.2 - 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.

3.2.1 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.

3.2.2 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`

3.2.3 NEWTYPE declarations

The standard SDL-92 NEWTYPE declaration is supported, with the following additions and restrictions:

- 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 wether 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.
- Only basic types for indices in `ARRAY` types are fully supported. Indices of complex types such as `STRUCT` or `CHOICE` will not work everywhere (e.g they will work as expected in simulation, but not in code generation).
- For simulation, it is possible to make the scope for `NEWTYPE`'s global: having two types with the same name in two different agents will then not work. This is done by checking the option "Manage all types in a single system-wide scope" in the project generation options. This option is unchecked by default.

3.2.4 SYNTYPE declarations

The standard SDL-92 `SYNTYPE` declaration is supported, with the following additions and restrictions:

- 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.
- As for `NEWTYPE`'s, the scope for `SYNTYPE`'s can be made global. See "NEWTYPE declarations" on page 179.

3.2.5 SYNONYM declarations

The standard SDL-92 `SYNONYM` declaration is supported, with the following restrictions:

- 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.

- As for NEWTYPE's, the scope for SYNONYM's can be made global. See "NEWTYPE declarations" on page 179.

3.2.6 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.

3.2.7 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.

3.2.8 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.

3.2.9 SIGNALSET declarations

SIGNALSET declarations are not needed and not supported in PragmaDev Studio.

3.2.10 USE declarations

The standard USE declarations are supported with the following additions and restrictions:

- USE clauses may appear at all levels in the architecture.
- 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.

3.2.11 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.

3.2.12 Data declarations (DCL)

The standard SDL-92 `DCL` declaration is supported with no restrictions. Remote variables are available, so the `REMOTE`, `IMPORTED`, `EXPORT` and `IMPORT` declarations are supported as well.

3.2.13 Structural element declarations

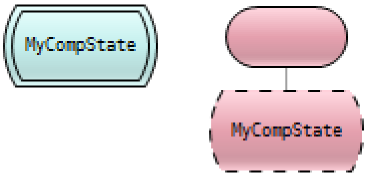
All standard SDL-92 structural element declarations are supported, except system types, procedure types, and service types, and with the addition of the SDL-2000 composite state:

- System;
- Block;
- Process;
- Procedure, both regular and remote;
- Package;
- Block type (named *block class* in PragmaDev Studio);
- Process type (named *process class* in PragmaDev Studio);
- Composite state;
- Service;
- Macro.

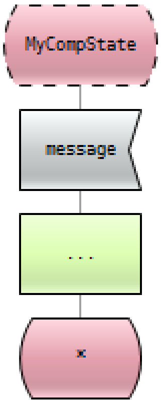
The following restrictions apply:

- 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.
- 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 (`.pr`) is used.
  - For agent classes declarations, a class diagram is used (see “Class description” on page 233). Note that passive classes cannot be associated in any way to SDL agents, as SDL does not have the necessary syntax elements to manipulate them.

- Gates in process and block classes are declared 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:

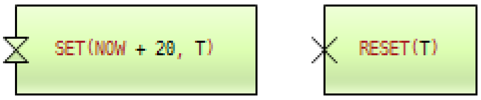


3.3 - SDL symbols syntax

The syntax for all symbols is compliant with the SDL-92 Z100 recommendation, with the following exceptions:

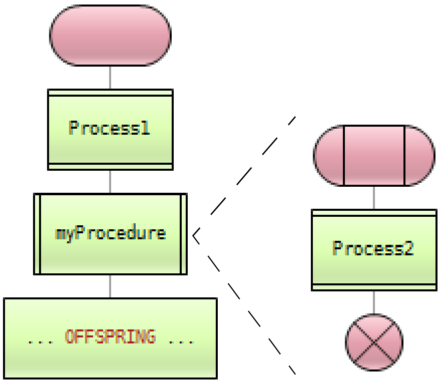
- Virtualities (`VIRTUAL`, `REDEFINED`, `FINALIZED`) in start, input, save and continuous signal symbols are valid, but ignored: all transitions are considered virtual by default.

- For timers, SET and RESET symbols have a specific shape:



- 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 TO 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 only partially supported: it will work in simulation if sort is a basic type with a discrete and finite set of values (e.g Boolean, or a SYNTYPE based on Integer with a min:max CONSTANTS clause).

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.

3.4 - Model Simulator

Two ways of debugging an 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.

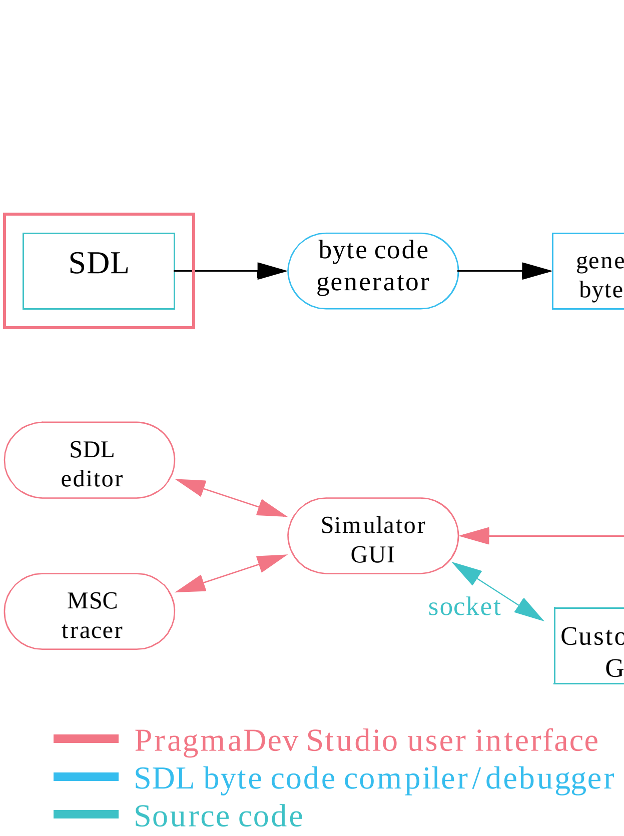
Both approaches have their advantages and drawbacks:

- 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 246 and “Model Debugger” on page 311. This section describes the internal Model Simulator.

3.4.1 Simulator architecture

The Model 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.

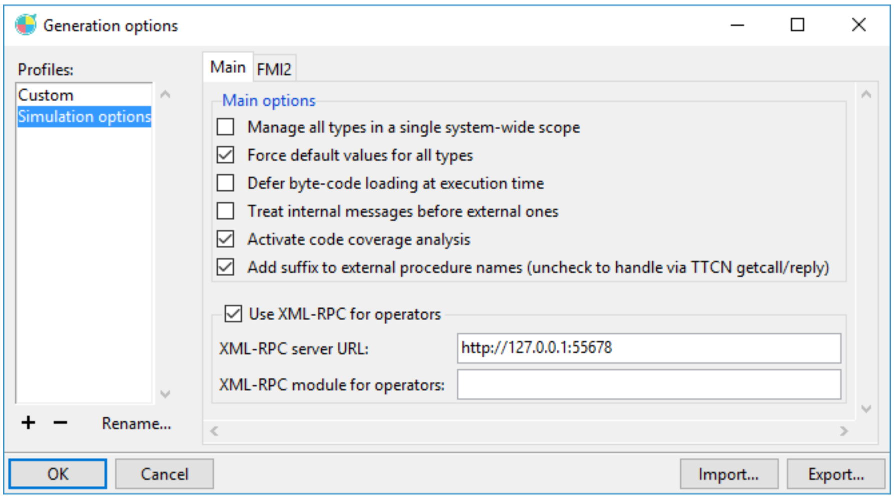


The Model Simulator has all the expected features of a debugger. It allows you to:

- 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.

3.4.2 Main simulator options

A few options are available when simulating the model. They can be configured in the “Generation / Options...” menu:



☐ Manage all types in a single system-wide scope
☑ Force default values for all types
☐ Defer byte-code loading at execution time
☐ Treat internal messages before external ones
☑ Activate code coverage analysis
☑ Add suffix to external procedure names (uncheck to handle via TTCN getcall/reply)

☑ Use XML-RPC for operators
XML-RPC server URL: http://127.0.0.1:55678
XML-RPC module for operators:

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 model coverage analysis*

In order to optimize performance and memory consumption, model 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 “User defined external operators and procedures” on page 218.

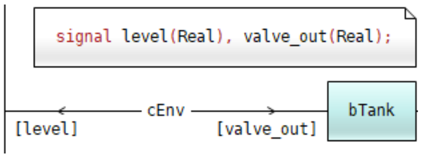
3.4.3 Co-simulation with FMI

The Model Simulator supports FMI 2.0 (Functional Mock-up Interface v2.0) for *Model-Exchange* (import only) and *CoSimulation* (master only).

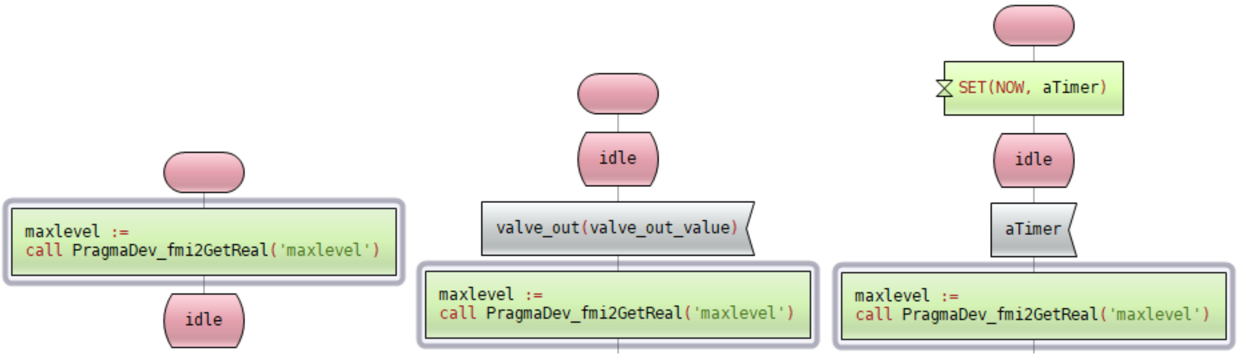
3.4.3.1 SDL system for co-simulation with FMI

To co-simulate an SDL system with an FMU (Functional Mock-up Unit) the following should be considered while modeling the system:

- Communication with the FMU is done only via the *environment*:
  - FMU *input variables* are SDL messages send from the system to the environment.
  - FMU *output variables* are SDL messages sent from the environment to the system.
- SDL messages used to communicate with the FMU should have a single parameter of one of the following types:
  - Boolean
  - Integer
  - Real
  - CharString



- FMU variables can be queried in SDL behavior diagrams using the predefined external procedures found in PragmaLib. Care should be taken when calling these procedures to make sure the returned value is valid and the intended one during FMU execution. As a general rule these procedures (a) should not be called during the start transition, and (b) it is advisable to call them during transitions triggered by FMU outputs. If a variable needs to be queried before any FMU output is received, then (c) a timer should be used to ensure the validity of the returned value.


maxlevel := call PragmaDev_fmi2GetReal('maxlevel')
idle
idle
valve_out(valve_out_value)
maxlevel := call PragmaDev_fmi2GetReal('maxlevel')
SET(NOW, aTimer)
idle
aTimer
maxlevel := call PragmaDev_fmi2GetReal('maxlevel')

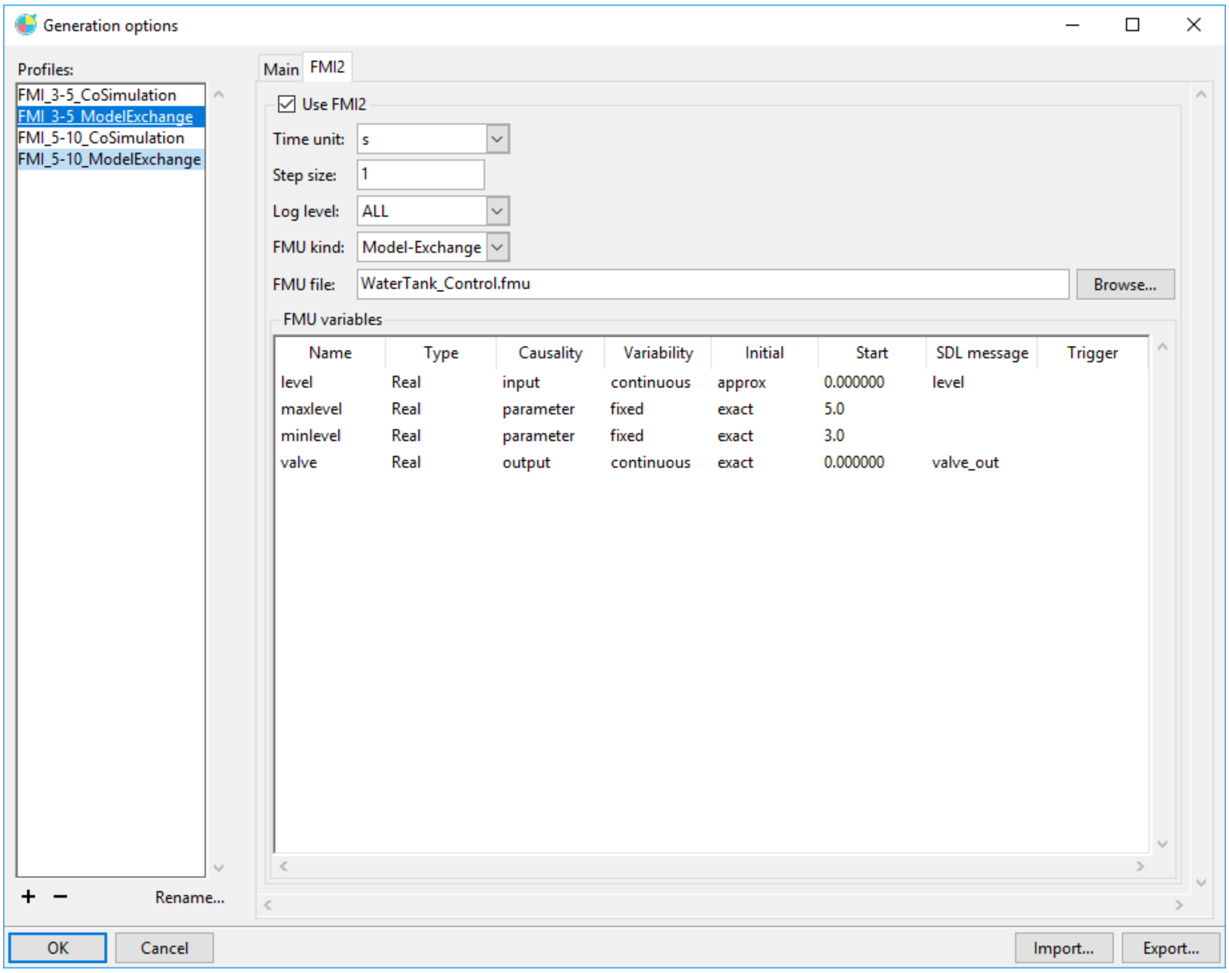
a) not valid b) valid c) valid

The available procedures for querying FMU variables are (see “Querying FMI2 variables” on page 217):

- `PragmaDev_fmi2GetBoolean(<variable name>)`
- `PragmaDev_fmi2GetInteger(<variable name>)`
- `PragmaDev_fmi2GetReal(<variable name>)`
- `PragmaDev_fmi2GetString(<variable name>)`

3.4.3.2 Simulator options for co-simulation with FMI

These options can be configured via the “Generation / Options...” menu, in the “FMI2” tab:



☑ Use FMI2

Time unit: s

Step size: 1

Log level: ALL

FMU kind: Model-Exchange

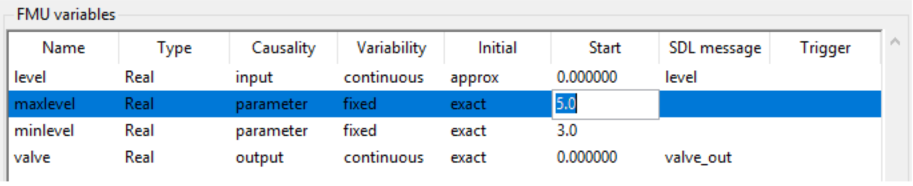
FMU file: WaterTank_Control.fmu

Name	Type	Causality	Variability	Initial	Start	SDL message	Trigger
level	Real	input	continuous	approx	0.000000	level	
maxlevel	Real	parameter	fixed	exact	5.0		
minlevel	Real	parameter	fixed	exact	3.0		
valve	Real	output	continuous	exact	0.000000	valve_out	

The available options are:

- *Use FMI2*
  Enables co-simulation with FMI in the Model Simulator.
- *Time unit*
  Unit of time to be used in SDL for co-simulation with FMI. For example, if a timer is started in SDL like `SET(NOW+10, aTimer)` and the time unit is set to ‘ms’, then the timer will fire after 10 milliseconds when co-simulating with an FMU.
- *Step size*
  This is the time interval (in *time units*) at which the `DoStep` (in FMI2 CoSimulation) or `SetTime` (in FMI2 ModelExchange) functions will be called by the Model Simulator during co-simulation.
- *Log level*
  Level of log messages generated by the FMU during co-simulation. These messages will be displayed in the Model Simulator window.

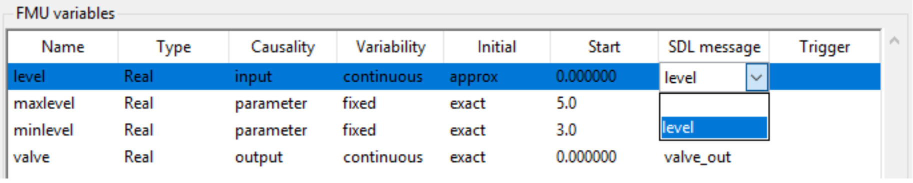
- *FMU kind*
  The FMU kind to use in co-simulation (either CoSimulation or ModelExchange) if both of them are supported by the given FMU.
- *FMU file*
  The FMU to use for co-simulation.
- *FMU variables*
  A table of all *FMI2 variables* found in the FMU file.
  - The “Start” value can be edited when applicable by double-clicking the corresponding cell in the table and entering a new value.



FMU variables

Name	Type	Causality	Variability	Initial	Start	SDL message	Trigger
level	Real	input	continuous	approx	0.000000	level	
maxlevel	Real	parameter	fixed	exact	5.0		
minlevel	Real	parameter	fixed	exact	3.0		
valve	Real	output	continuous	exact	0.000000	valve_out	

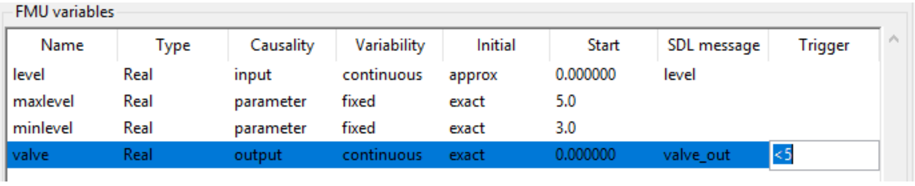
  - An “SDL message” can be mapped to an FMU *input* or *output* by double-clicking the corresponding cell in the table and choosing one of the available messages in the list.



FMU variables

Name	Type	Causality	Variability	Initial	Start	SDL message	Trigger
level	Real	input	continuous	approx	0.000000	level	
maxlevel	Real	parameter	fixed	exact	5.0		
minlevel	Real	parameter	fixed	exact	3.0		
valve	Real	output	continuous	exact	0.000000	valve_out	

  - The “Trigger” for an FMU *output* can be set by double-clicking the corresponding cell in the table and entering a new value.



FMU variables

Name	Type	Causality	Variability	Initial	Start	SDL message	Trigger
level	Real	input	continuous	approx	0.000000	level	
maxlevel	Real	parameter	fixed	exact	5.0		
minlevel	Real	parameter	fixed	exact	3.0		
valve	Real	output	continuous	exact	0.000000	valve_out	<5

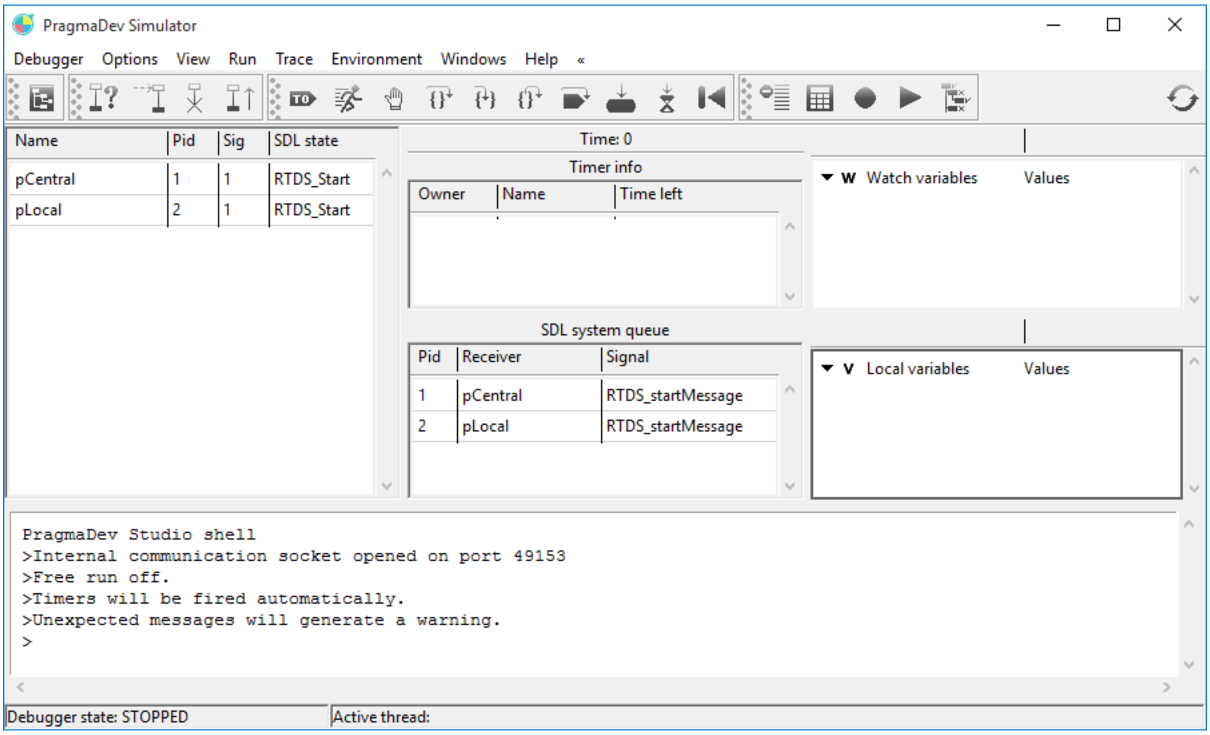
  A trigger is a comma separated list of conditions. The mapped SDL message will be sent to the SDL system only if one of the conditions is fulfilled. By default (no trigger set) the SDL message will be sent every *step size*. A condition is defined using one of the SDL boolean operators (‘=’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘/=’) followed by a value of the same type as the output variable. For Integer and Real types all operators can be used in conditions, while for Boolean and String types only the ‘=’ and ‘/=’ operators are allowed. A list of conditions is checked from left-to-right every *step size*. If a condition is fulfilled, then an SDL message is sent, and the condition is marked as checked. A checked condition will not be evaluated in the next step. A condition check-mark is removed when another condition is fulfilled and thus checked. If the trigger consists of a single condition then the message can be sent only once during the co-simulation, i.e., when the condition if fulfilled.

An example for co-simulation with FMI can be found in PragmaDev Studio example projects, in `$RTDS_HOME/examples/Specifier/WaterTank_FMI`.

3.4.4 Launching the Model Simulator

The Model 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 Model Simulator window is started automatically and you are ready to debug your system.



Name	Pid	Sig	SDL state
pCentral	1	1	RTDS_Start
pLocal	2	1	RTDS_Start

Time: 0

Timer info

Owner	Name	Time left

W Watch variables	Values

SDL system queue

Pid	Receiver	Signal
1	pCentral	RTDS_startMessage
2	pLocal	RTDS_startMessage

V Local variables	Values

```
PragmaDev Studio shell
>Internal communication socket opened on port 49153
>Free run off.
>Timers will be fired automatically.
>Unexpected messages will generate a warning.
>
```

Debugger state: STOPPED

Active thread:

The Model 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 Model Simulator can be restarted at any time with the reset button or shell command. The underlying simulation environment is restarted and cleaned up.

3.4.5 Stepping levels

Since your source code is a composite of graphical SDL symbols and textual SDL lines of code, the Model 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 Model 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,

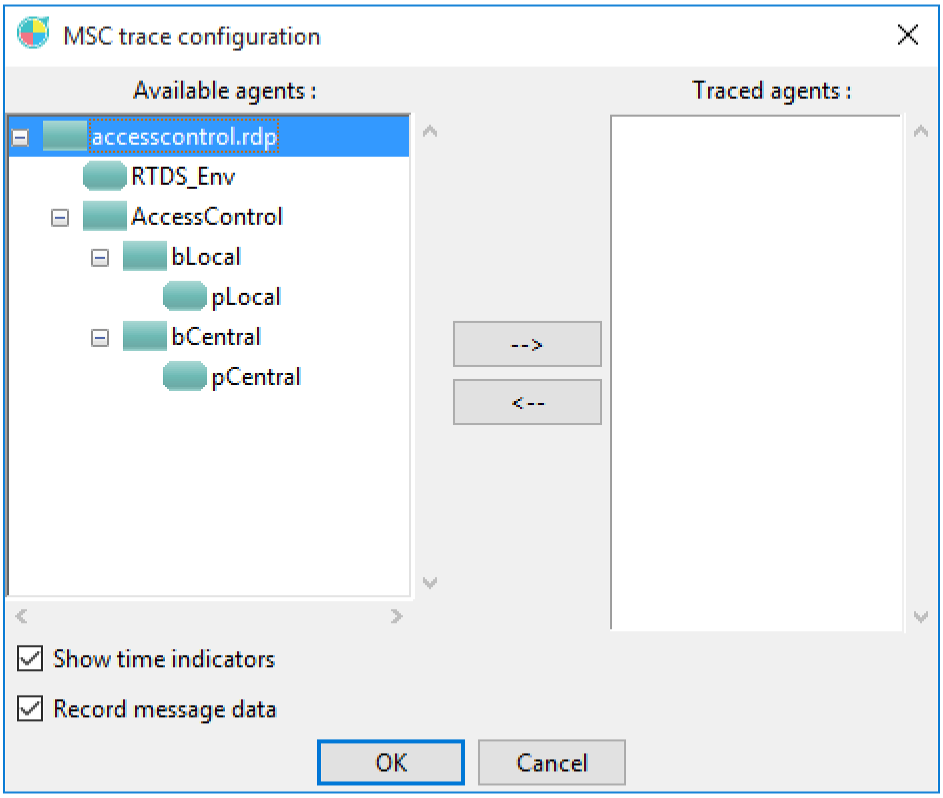
- Run until all signals are consumed except timers.

3.4.6 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.

The tracer window itself is described in “Tracer window” on page 383.

3.4.7 Displayed information

The Model 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.

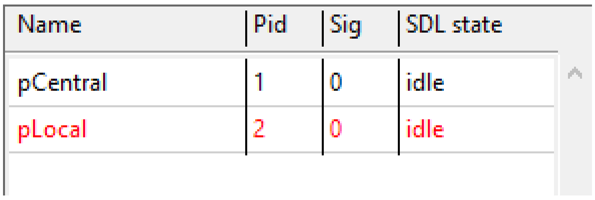
The information to refresh can be setup in the “Options / Refresh options...” menu as explained in “Refresh options” on page 201.

3.4.7.1 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.

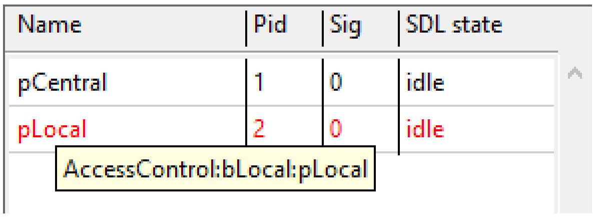
When the system is running the active process line is printed in red.



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:

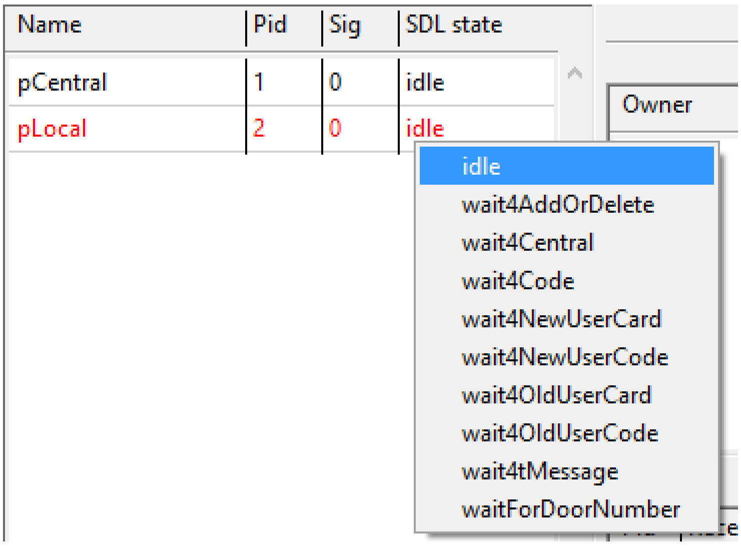


Name	Pid	Sig	SDL state
pCentral	1	0	idle
pLocal	2	0	idle

AccessControl:bLocal:pLocal

Double-clicking on a process name will also open the corresponding diagram in an editor window.

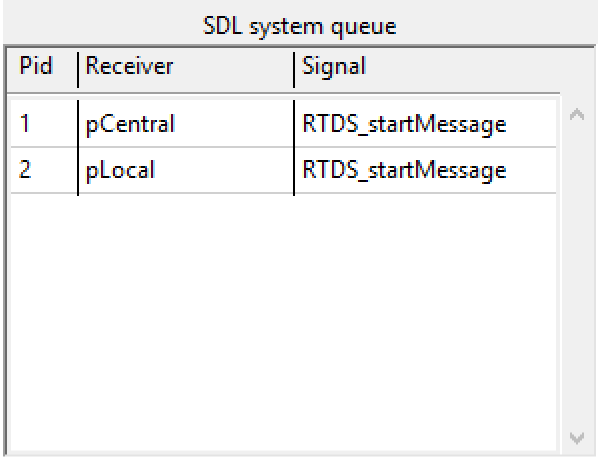
The *Process information* window also allows to modify the SDL state of a process. To do so right click on the SDL state column of the process line. A pop up menu will list all the available SDL state that have been defined in the system. Select one and the SDL state is modified.



Name	Pid	Sig	SDL state
pCentral	1	0	idle
pLocal	2	0	idle

3.4.7.2 System queue

Model 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.

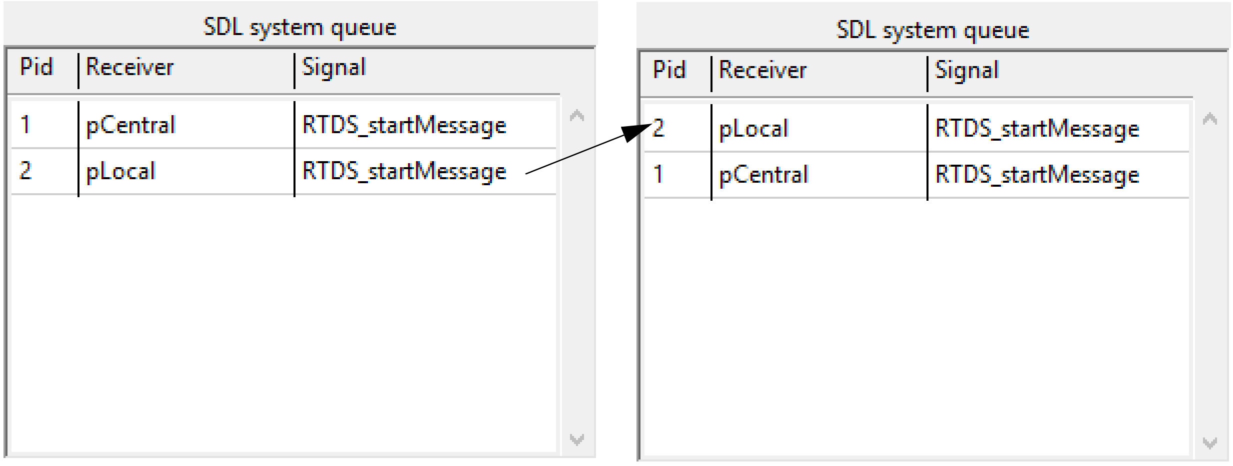
That allows to:

- Execute the signal inputs in the order the signals have been sent.

- Re-order the pending signals.

That is a key feature of the Model 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

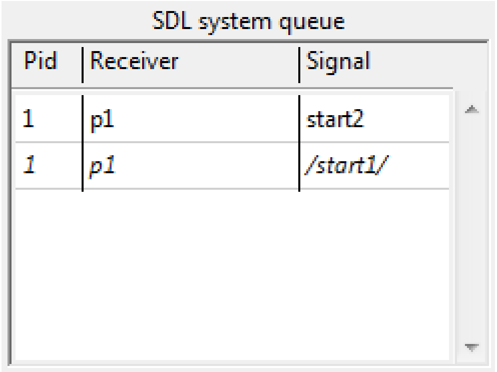
Pid	Receiver	Signal
1	pCentral	RTDS_startMessage
2	pLocal	RTDS_startMessage

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.

3.4.7.3 Timers

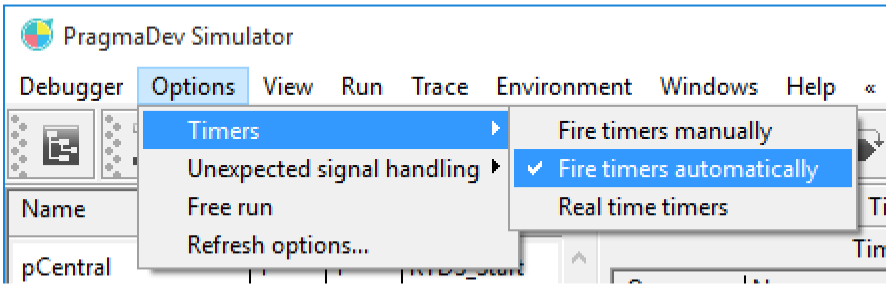
The *Timer info* part displays all on-going timers started from the SDL design. The displayed fields are:

- 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 415).

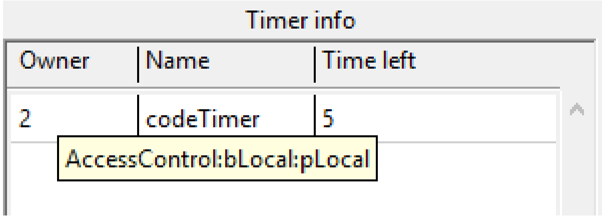
There are 3 ways to manipulate timers in the Model Simulator. Selection is done through the “Options / Timers” menu:



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

3.4.7.4 Watch

There are several ways to add a variable in the *Watch window*:

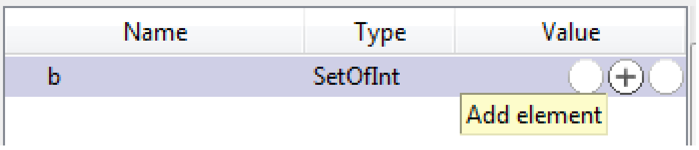
- From the shell
  Type the following command in the shell:
  `watch add <variable name>`
- From the SDL editor
  Select an expression in the SDL editor and go to the “Debug / Add watch” menu to add the expression in the *Watch window*.

An existing watch can also be removed from the list of watched variables by either:

- Selecting it in the tree and press the ‘Del’ or ‘Backspace’ key;
- Running in the simulator shell the command:
  `watch del <variable name>`
  Note that for this command to work, the variable must be in the current scope.

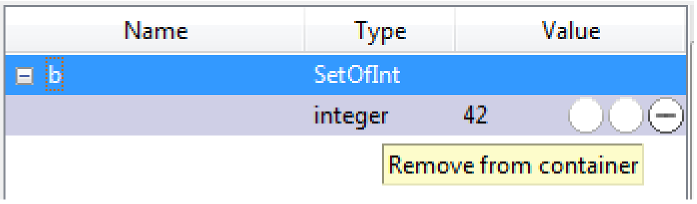
The *Watch window* also allows to modify the value of variables. To do so:

- If the variable has a single value, double click on it, change it, then press `<Return>` to update it.
- If the variable has several values - which is true for SDL arrays, sequences and bags -, buttons will appear when the mouse pointer is over the variable or one of its elements:
  - For containers, a button will appear allowing to add an element to it:



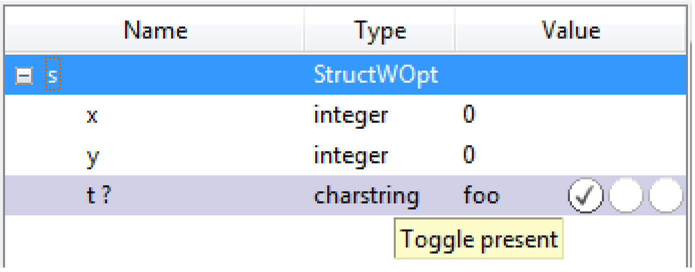
Name	Type	Value
b	SetOfInt	

  - For elements, a button will appear allowing to remove it from its container:



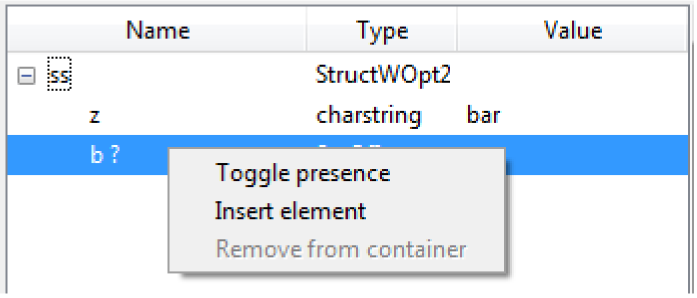
Name	Type	Value
⊟ b	SetOfInt	
	integer	42

- If the variable value is optional such as for an optional field in a struct, a button will appear allowing to toggle the variable presence:



Name	Type	Value
⊟ s	StructWOpt	
x	integer	0
y	integer	0
t ?	charstring	foo

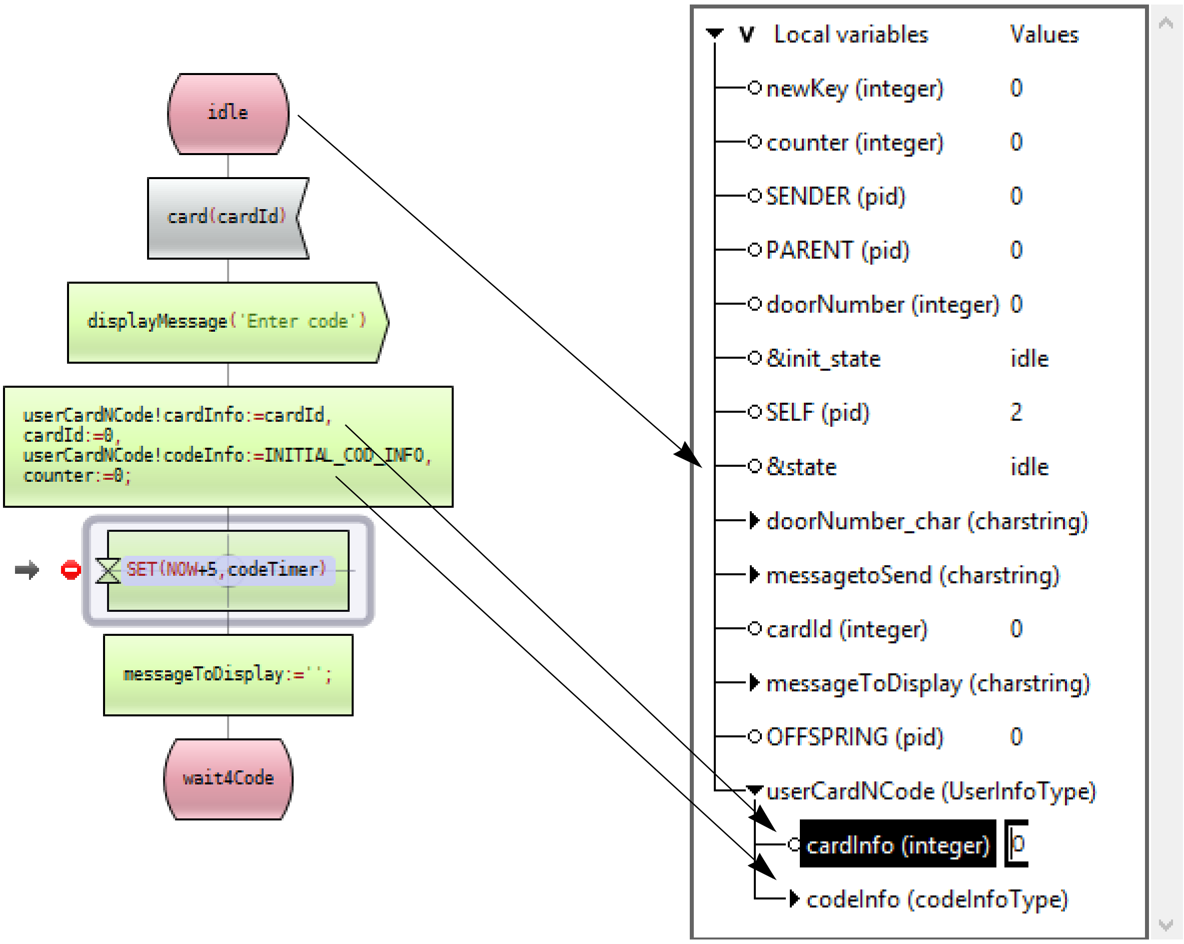
Adding an element to a container, removing an element from its container and toggling a variable presence can also be done via the contextual menu:



Name	Type	Value
⊟ ss	StructWOpt2	
z	charstring	bar
b ?		

3.4.7.5 Local variables

When stepping through the code the Model Simulator automatically displays the local variables of the current process:



idle
card(cardId)
displayMessage('Enter code')
userCardNCode!cardInfo:=cardId,
cardId:=0,
userCardNCode!codeInfo:=INITIAL_COD_INFO,
counter:=0;
SET(NOW+5,codeTimer)
messageToDisplay:='';
wait4Code

Local variables	Values
newKey (integer)	0
counter (integer)	0
SENDER (pid)	0
PARENT (pid)	0
doorNumber (integer)	0
&init_state	idle
SELF (pid)	2
&state	idle
doorNumber_char (charstring)	
messagetoSend (charstring)	
cardId (integer)	0
messageToDisplay (charstring)	
OFFSPRING (pid)	0
userCardNCode (UserInfoType)	
cardInfo (integer)	0
codeInfo (codeInfoType)	

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 in the same way as for the watched variables. See “Watch” on page 198.

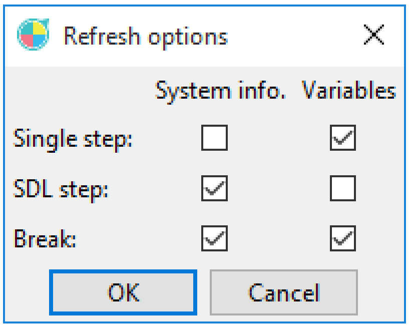
3.4.7.6 Refresh options

The information displayed in the Model 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:



	System info.	Variables
Single step:	☐	☑
SDL step:	☑	☐
Break:	☑	☑

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: ⟳

3.4.8 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:
|$(<os environment variable>) to acces an operating system environment variable
|${<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.

3.4.8.1 shell commands

To list all the available commands in this category, type:
`help shell`

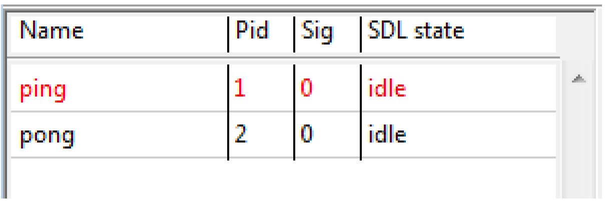
It will list the following commands:

```
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`

3.4.8.2 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.

3.4.8.3 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 `systemQueueSetNextReceiverName` 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 Model Simulator.
- `extractCoverage`
  Extracts the model 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 model coverage results stored in it.

3.4.8.4 variables commands

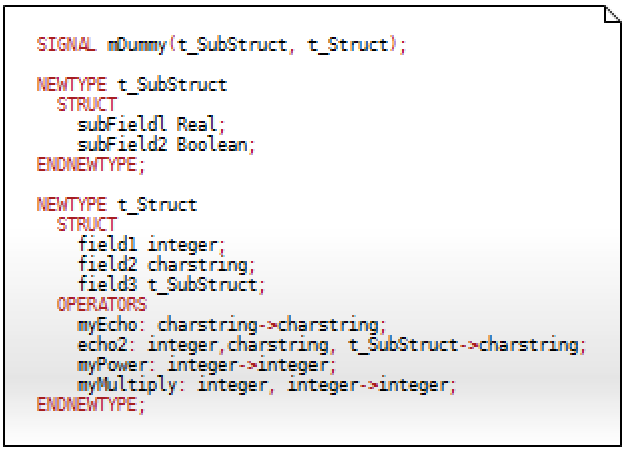
To list all the available commands in this category, type:
`help variables`

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 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:

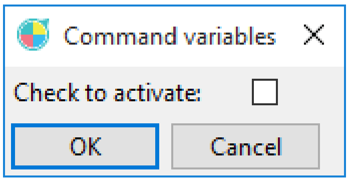
```
>varFromValue c=|{subField1|=1.23|,subField2|=0|}
>varFromType b t_Struct
>varFieldSet b.field1=666
>varFieldSet b.field2=Hello world
>varFieldSet b.field3.subField1=6.55957
>varFieldSet b.field3.subField2=0
>print b
|{field1|=666|,field2|=Hello world|,field3|=|{subField1|=6.55957|,subField2|=0|}|}
>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.

3.4.8.5 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 Model Simulator. This feature should be used to increase simulation speed and when internal information is not needed.

3.4.8.6 customization commands

To list all the available commands in this category, type:
`help customization`

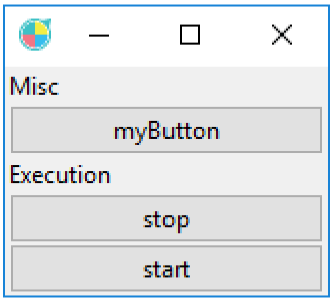
It will list the following commands:

```
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:

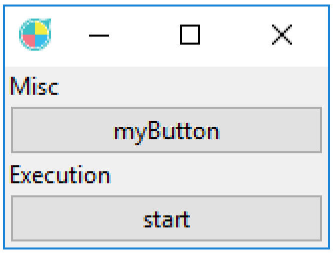
```
>buttonWindowCreate myWindow
>buttonWindowLabelAdd myWindow Misc
>buttonWindowAdd myWindow myButton = help
>buttonWindowLabelAdd myWindow Execution
>buttonWindowAdd myWindow stop = send2name pPing normal mStop
```

```
>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.

3.4.9 Status bar

The status bar is divided in two parts:

- The Model Simulator internal state
  The Model 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 2: Model 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 Model Simulator is stuck. Restart the Model Simulator.

Table 2: Model Simulator internal states

- The active thread
  The active thread is displayed in the right part of the status bar when known.

3.4.10 Breakpoints

3.4.10.1 Setting breakpoints

Breakpoints are set in the SDL editor. Select an SDL symbol and click on STOP 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
```

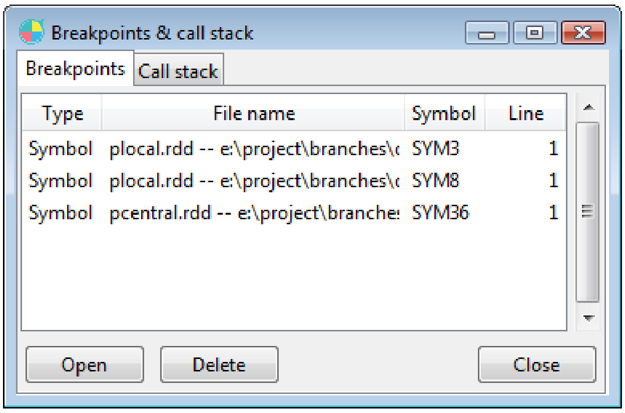
Since symbol identifiers are not directly visible in the diagram editor, the best way to get the command is to set the breakpoint interactively, which will record the corresponding command in the shell history. Note that symbol identifiers never change, so it is safe to put such a command in a scenario file that will be executed several times.

3.4.10.2 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:



Type	File name	Symbol	Line
Symbol	plocal.rdd -- e:\project\branches\(	SYM3	1
Symbol	plocal.rdd -- e:\project\branches\(	SYM8	1
Symbol	pcentral.rdd -- e:\project\branche:	SYM36	1

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.

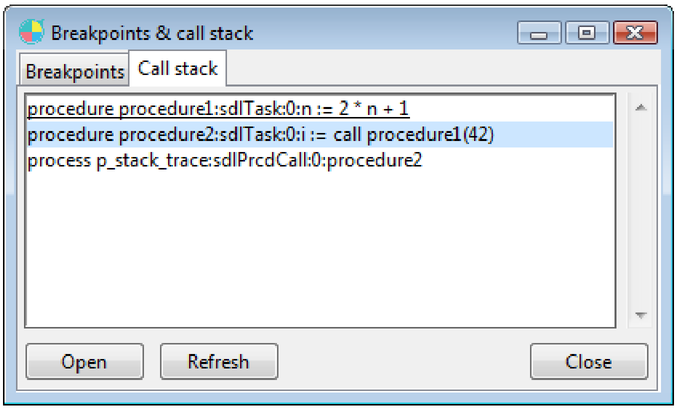
3.4.10.3 Deleting breakpoints

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 211.
- 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.

3.4.10.4 Call stack

When a breakpoint is hit or whenever the execution stops in the middle of some code, the call stack can be displayed via the same button as the brekapoint list ( ), and selecting the “*Call stack*” tab:

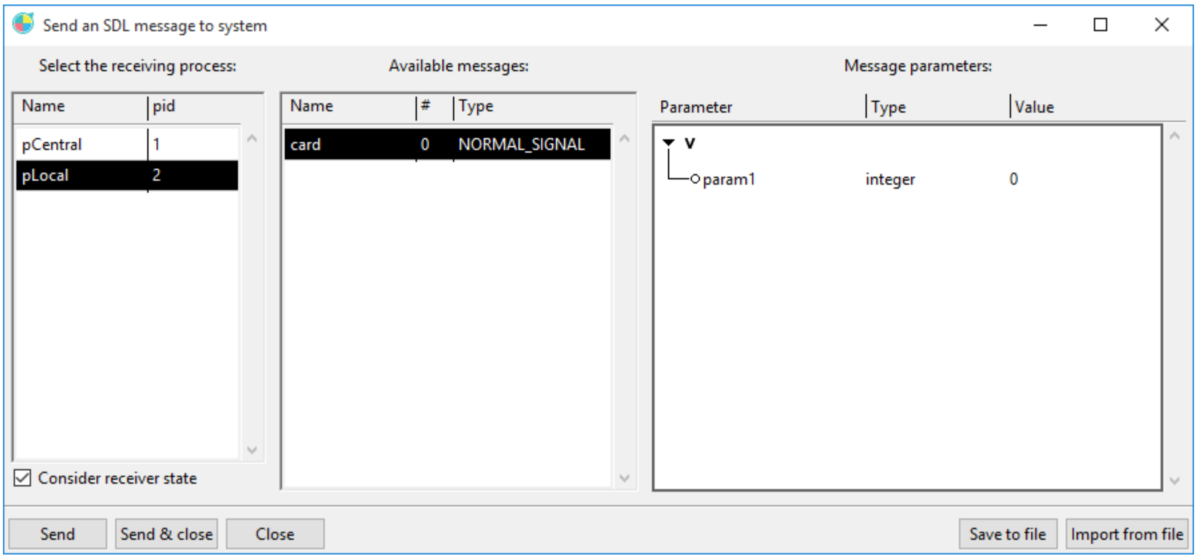


```
procedure procedure1:sdlTask:0:n := 2 * n + 1
procedure procedure2:sdlTask:0:i := call procedure1(42)
process p_stack_trace:sdlPrcdCall:0:procedure2
```

This can be particularly useful when a system contains a lot of procedures calling each other. Double-clicking on any item in the call stack will open the corresponding diagram and show the symbol where the execution has stopped.

3.4.11 Sending SDL messages to the running system

The Model 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:



Name	pid
pCentral	1
pLocal	2

Name	#	Type
card	0	NORMAL_SIGNAL

Parameter	Type	Value
v		
param1	integer	0

<u>*The send an SDL message to system Window*</u>

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.

3.4.12 Model coverage

The Model Simulator's *Get model coverage* button gets the model coverage analysis results for the running system so far. This feature is available only if the *Activate model coverage analysis* is checked in the simulation options (see "Main simulator options" on page 187).

For more details on model coverage results, see "Code coverage results" on page 161.

3.4.13 Provided external procedures

A number of built-in procedures are available in Model Simulator. To have access to these procedures, the library must be imported in the model by "using" a built-in package:
`use PragmaLib;`

3.4.13.1 Formatted output

The following procedures are available for formatted output:

- `PragmaDev_b4sprintf(<arg>) -> <sprintf arg>`
  - <arg>: boolean.
  - <sprintf arg>: PragmaDev_arg4sprintf. Built-in wrapper type for <arg>.

  This procedure is declared as:
  ```
  PROCEDURE PragmaDev_b4sprintf(boolean_arg BOOLEAN) ->
  PragmaDev_arg4sprintf EXTERNAL;
  ```
- `PragmaDev_i4sprintf(<arg>) -> <sprintf arg>`
  - <arg>: integer.
  - <sprintf arg>: PragmaDev_arg4sprintf. Built-in wrapper type for <arg>.

  This procedure is declared as:
  ```
  PROCEDURE PragmaDev_i4sprintf(integer_arg INTEGER) ->
  PragmaDev_arg4sprintf EXTERNAL;
  ```
- `PragmaDev_f4sprintf(<arg>) -> <sprintf arg>`
  - <arg>: real.
  - <sprintf arg>: PragmaDev_arg4sprintf. Built-in wrapper type for <arg>.

This procedure is declared as:
`PROCEDURE PragmaDev_f4sprintf(real_arg REAL) -> PragmaDev_arg4sprintf EXTERNAL;`

- `PragmaDev_s4sprintf(<arg>) -> <sprintf arg>`
  - <arg>: charstring.
  - <sprintf arg>: PragmaDev_arg4sprintf. Built-in wrapper type for <arg>.

  This procedure is declared as:
  `PROCEDURE PragmaDev_s4sprintf(charstring_arg CHARSTRING) -> PragmaDev_arg4sprintf EXTERNAL;`
- `PragmaDev_sprintf(<format>, <args>) -> <formatted string>`
  - <format>: charstring. Based on C sprintf format specifiers (%d, %f, %s, ...).
  - <args>: PragmaDev_arg4sprintf. Built-in wrapper type for boolean, integer, real, and charstring. This should be a concatenation of the procedures described above (PragmaDev_<specifier>4sprintf where <specifier> is either b, i, f, or s).
  - <formatted string>: charstring.

  This procedure is declared as:
  `PROCEDURE PragmaDev_sprintf(format CHARSTRING, args PragmaDev_arg4sprintf) -> CHARSTRING EXTERNAL;`
  Usage example:
  `CHARSTRING output := PragmaDev_sprintf('This is %s %d', PragmaDev_s4sprintf('number') // PragmaDev_i4sprintf(42));`

3.4.13.2 File manipulation

The following procedures are available for file manipulation:

- `PragmaDev_FileOpen(<file name>, <open mode>) -> <file id.>`
  - <file id.>: integer.
  - <file name>: charstring. Path is relative to the project.
  - <open mode>: charstring. Based on C fopen file manipulation modes (‘w’, ‘r’, ‘a’, ‘r+’,’a+’...)

  This procedure is declared as:
  `PROCEDURE PragmaDev_FileOpen(file_name CHARSTRING, file_mode CHARSTRING) -> INTEGER EXTERNAL;`
- `PragmaDev_FileClose(<file id.>) -> <success>`
  - <success>: boolean.
  - <file id.>: integer. Value given by the PragmaDev_FileOpen.

  This procedure is declared as:
  `PROCEDURE PragmaDev_FileClose(file_id INTEGER) -> BOOLEAN EXTERNAL;`
- `PragmaDev_FileReadLine(<file id.>) -> <read line>`
  - <read line>: charstring. The line read in the file. It will always include at least the ending new line character. If the returned string is empty, it means the end of the file has been reached.
  - <file id.>: integer. Value given by the PragmaDev_FileOpen.

  This procedure is declared as:
  `PROCEDURE PragmaDev_FileReadLine(file_id INTEGER) -> CHARSTRING EXTERNAL;`

- `PragmaDev_FileWriteLine(<line>, <file id.>) -> <success>`
  - <success>: boolean.
  - <line>: charstring. Line to write in the file.
  - <file id.>: integer. Value given by the PragmaDev_FileOpen.

  This procedure is declared as:
  ```
  PROCEDURE PragmaDev_FileWriteLine(string_to_write CHARSTRING,
  file_id INTEGER) -> BOOLEAN EXTERNAL;
  ```

3.4.13.3 Radar graph

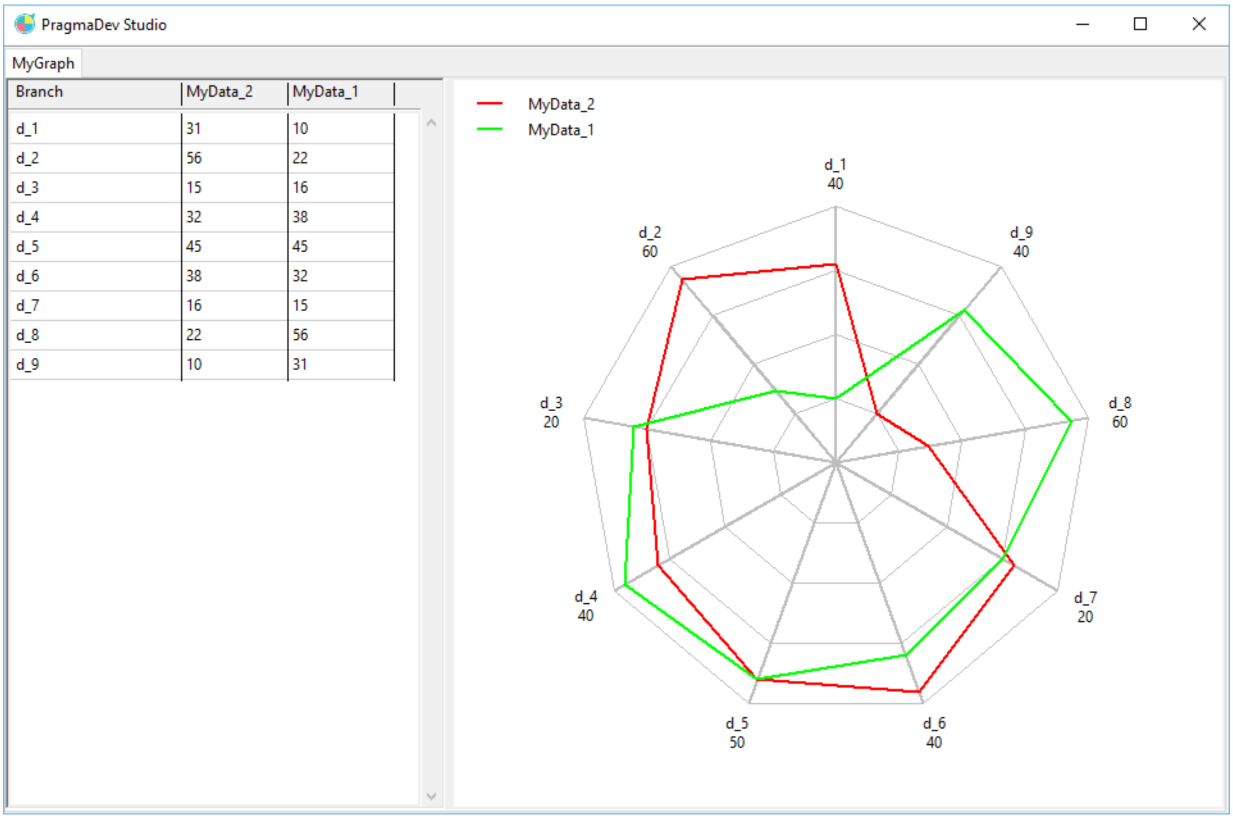
The following procedures are available to generate radar graphs. Several graphs can be generated at the same time. The resulting window will organize them in tabs. The scale on the branches is automatically adjusted.

- `PragmaDev_RadarGraphCreate(<graph name>, <branch labels>) -> <graph id.>`
  - <graph id.>: integer.
  - <graph name>: charsting. The name will be displayed in the window tab.
  - <branch labels>: charstring. Semi-column separated list of branch names.

  This procedure is declared as:
  ```
  PROCEDURE PragmaDev_RadarGraphCreate(graph_name CHARSTRING,
  branch_labels CHARSTRING) -> INTEGER EXTERNAL;
  ```
- `PragmaDev_RadarGraphAddLine(<graph id.>, <line label>, <line values>) -> <status>`
  - <status>: boolean.
  - <graph id.>: integer. Value given by the PragmaDev_RadarGraphCreate.
  - <line label>: charstring. Label of the line in the graph.
  - <line values>: charstring. Semicolon separated list of values for each branch.

  This procedure is declared as:

```
PROCEDURE PragmaDev_RadarGraphAddLine(graph_id INTEGER,
line_label CHARSTRING, line_branch_values CHARSTRING) -> BOOLEAN
EXTERNAL;
```



Branch	MyData_2	MyData_1
d_1	31	10
d_2	56	22
d_3	15	16
d_4	32	38
d_5	45	45
d_6	38	32
d_7	16	15
d_8	22	56
d_9	10	31

Radar graph example

3.4.13.4 Querying FMI2 variables

The following procedures are available for querying FMU variable values in FMI2 co-simulation:

- `PragmaDev_fmi2GetBoolean(<variable name>) -> <variable value>`
  - <variable name>: charstring.
  - <variable value>: boolean.

  This procedure is declared as:
  ```
  PROCEDURE PragmaDev_fmi2GetBoolean(variable_name CHARSTRING) ->
  BOOLEAN EXTERNAL;
  ```
- `PragmaDev_fmi2GetInteger(<variable name>) -> <variable value>`
  - <variable name>: charstring.
  - <variable value>: integer.

  This procedure is declared as:
  ```
  PROCEDURE PragmaDev_fmi2GetInteger(variable_name CHARSTRING) ->
  INTEGER EXTERNAL;
  ```
- `PragmaDev_fmi2GetReal(<variable name>) -> <variable value>`
  - <variable name>: charstring.
  - <variable value>: real.

  This procedure is declared as:

```
PROCEDURE PragmaDev_fmi2GetReal(variable_name CHARSTRING) ->
REAL EXTERNAL;
```

- `PragmaDev_fmi2GetString(<variable name>) -> <variable value>`
  - <variable name>: charstring.
  - <variable value>: charstring.

  This procedure is declared as:
  ```
  PROCEDURE PragmaDev_fmi2GetString(variable_name CHARSTRING) ->
  CHARSTRING EXTERNAL;
  ```

3.4.14 User defined external operators and procedures

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 “Main simulator options” on page 187.

An operator call in the Model Simulator will result in a call to:

```
[<module name>.]<operator name>
```

on the server.

An external procedure call in the Model 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`	*Not available.*
`OctetString`	*Not available.*
`PID`	`<int>`
`Duration`	`<int>`
`Time`	`<int>`
`STRUCT`	`<struct>` with the same fields as the SDL `STRUCT` in the same order
`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`* `)`	`<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 *`ElementSort`*. The array may also contain an additional single `<struct>` with a single field called '`default`', its type also being the XML-RPC representation of *`ElementSort`*. 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: `array := (. … .)`
`String(`*`ElementSort`*`)`	*Not available.*
`Bag(`*`ElementSort`*`)`	*Not available.*

Table 3: XML-RPC representation of SDL data types

An example is available in our distribution.

3.4.15 Connecting an external tool

3.4.15.1 Normal mode

It is possible to connect an external tool to the Model Simulator through a socket. To allow connections to the Model 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 Model Simulator is running. Only the port number can be configured.

The Model 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 Model Simulator `help` and display it in the shell.

3.4.15.2 XML mode

If a more structured way to communicate with the simulator is needed, it is also possible to connect an external tool in XML mode with the command:

```
connectxml <port number>
```

The connection works exactly the same way, but the commands sent to the simulator as well as the answers received from it are encapsulated in XML tags. The disconnection commnd is the same as in normal mode (`disconnect`).

For a more precise description on the format of the commands and answers, please refer to the corresponding section in the Reference Manual.

3.4.16 Command line simulation

The Simulator can be started from a shell or a DOS console and run an execution script automatically with the `simulate` sub-command of the PragmaDev Studio command line interface tool pragmastudiocommand. Check the Reference Manual for more information.

3.4.17 Raspberry Pi GPIO

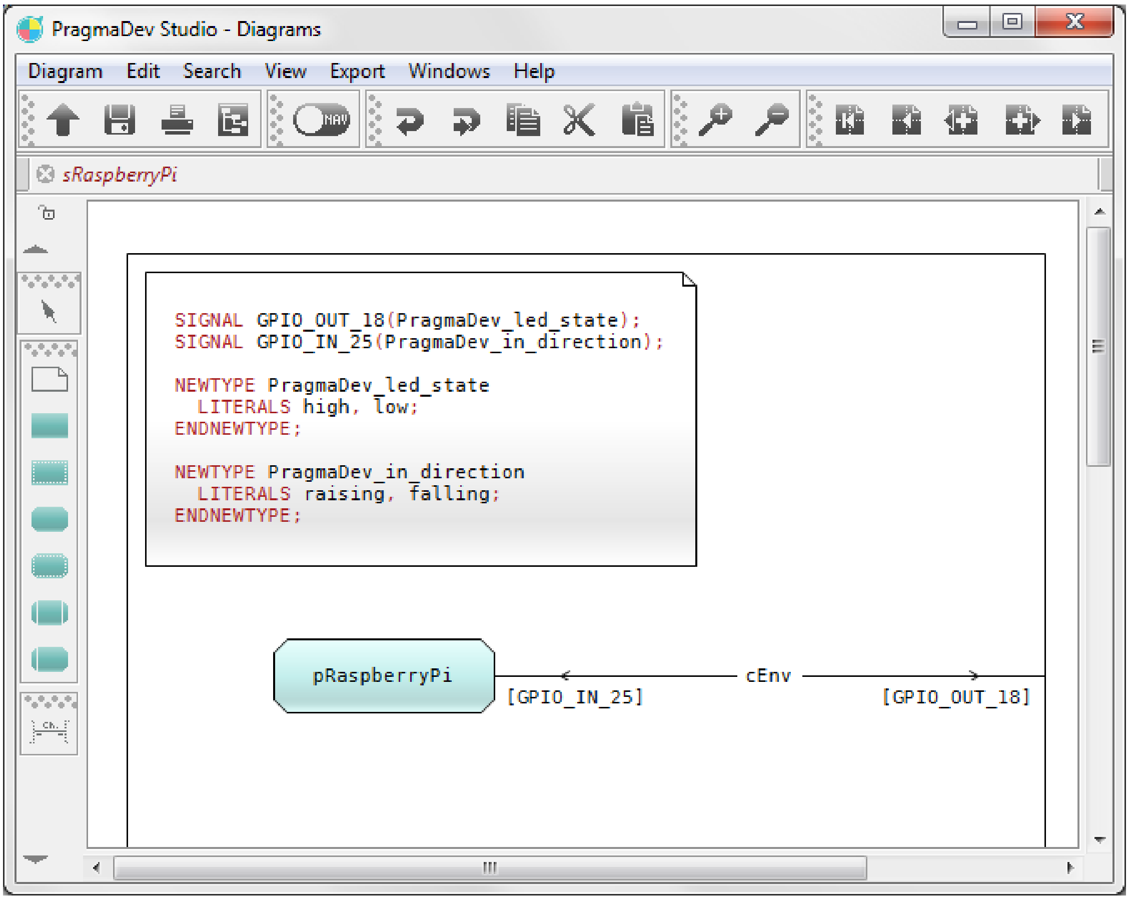
When running on a Raspberry Pi the Model Simulator can directly interact with the GPIO of the Raspberry board. For that matter, outgoing messsages to the environment must be named GPIO_OUT_XX where XX is the GPIO number, and incoming messages from the environment must be named GPIO_IN_YY where YY is the GPIO number. Please note the GPIO numbering goes from 1 to 27 and a GPIO can not be outgoing and incoming at the same time.

The GPIO_OUT_XX outgoing signal comes with a parameter that is a LITERALS which possible values are: high or low. The GPIO_IN_YY incoming signals comes with a parameter that is a LITERALS which possible values are: raising or falling.

The declaration in the system must be of that form:

```
SIGNAL GPIO_OUT_XX(PragmaDev_led_state);
SIGNAL GPIO_IN_YY(PragmaDev_in_direction);

NEWTYPE PragmaDev_led_state
  LITERALS high, low;
ENDNEWTYPE;

NEWTYPE PragmaDev_in_direction
  LITERALS raising, falling;
ENDNEWTYPE;
```

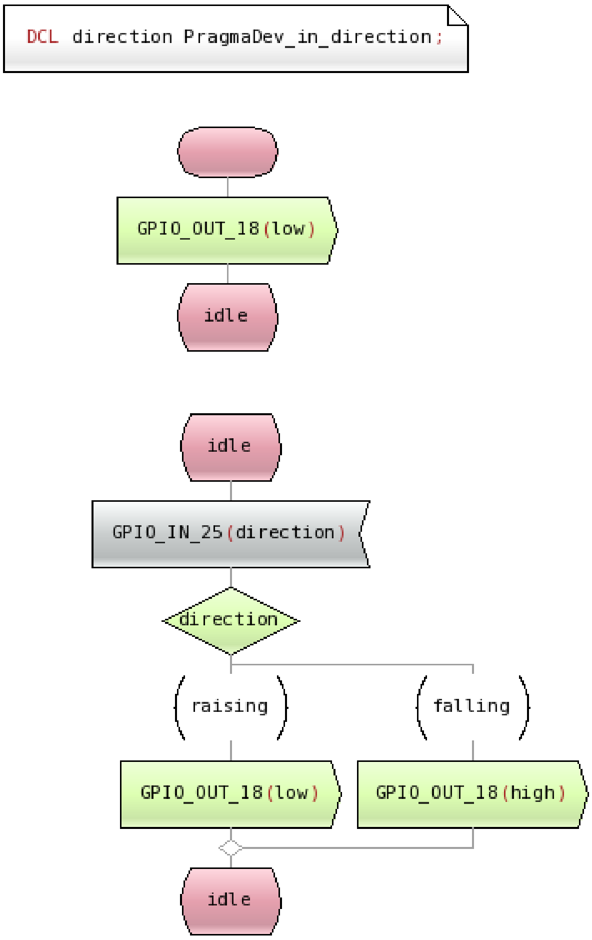
In the example provided in the distribution, the system lights up a LED connected to GPIO 18 when pressing a button connected to GPIO 25:

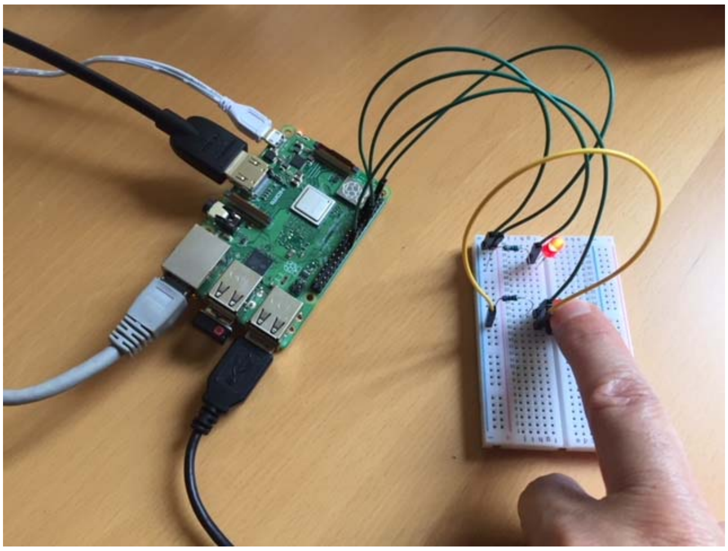

PragmaDev Studio - Diagrams
Diagram Edit Search View Export Windows Help
sRaspberryPi
pRaspberryPi
cEnv
[GPIO_IN_25]
[GPIO_OUT_18]

```
SIGNAL GPIO_OUT_18(PragmaDev_led_state);
SIGNAL GPIO_IN_25(PragmaDev_in_direction);

NEWTYPE PragmaDev_led_state
  LITERALS high, low;
ENDNEWTYPE;

NEWTYPE PragmaDev_in_direction
  LITERALS raising, falling;
ENDNEWTYPE;
```

On the channel connected to the environment, declaring an outgoing message called GPIO_OUT_18 sets GPIO 18 as an output on the board automatically. And declaring an incoming message called GPIO_IN_25 sets the GPIO 25 as an input automatically.

The start transition of the state machine initially turns off the LED. Then when the button is pressed the LED is lit up and when the button is released it is turned off.





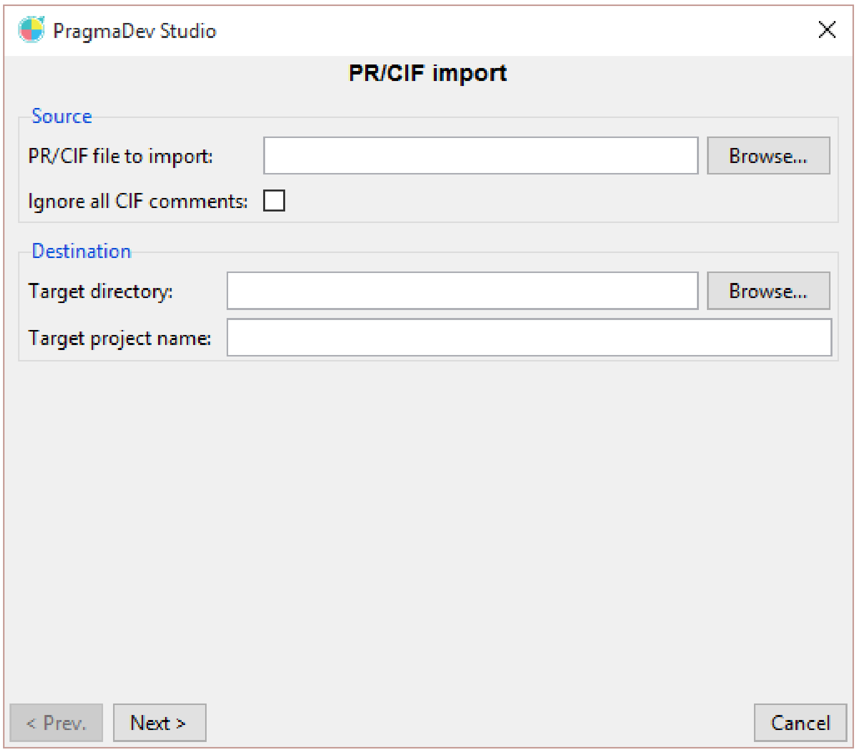
3.5 - Importing a PR/CIF file

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

To export a model as a PR file please refer to “Exporting the project as an SDL/PR file” on page 464.

3.5.1 Source & destination panel

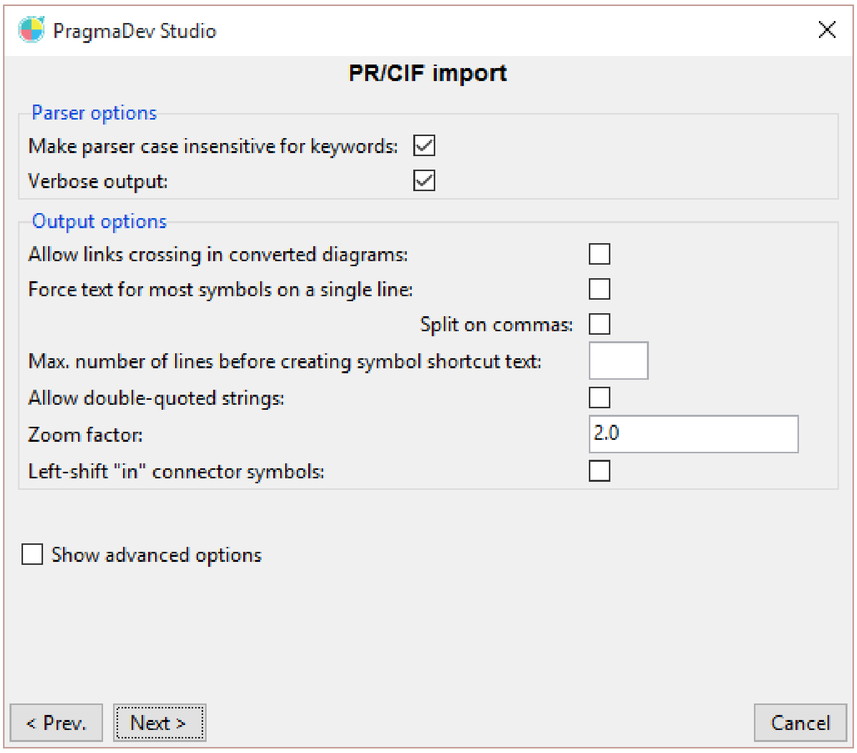
The first panel in the PR/CIF import wizard is the following:



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 place and size the symbols automatically.

3.5.2 Basic options panel

The next panel are the basic import options:



PragmaDev Studio

PR/CIF import

Parser options

Make parser case insensitive for keywords: ☑

Verbose output: ☑

Output options

Allow links crossing in converted diagrams: □

Force text for most symbols on a single line: □

Split on commas: □

Max. number of lines before creating symbol shortcut text:

Allow double-quoted strings: □

Zoom factor: 2.0

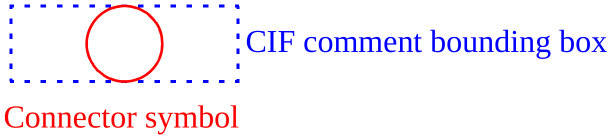
Left-shift "in" connector symbols: □

□ Show advanced options

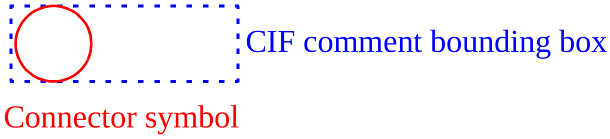
The options in this panel are:

- *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:

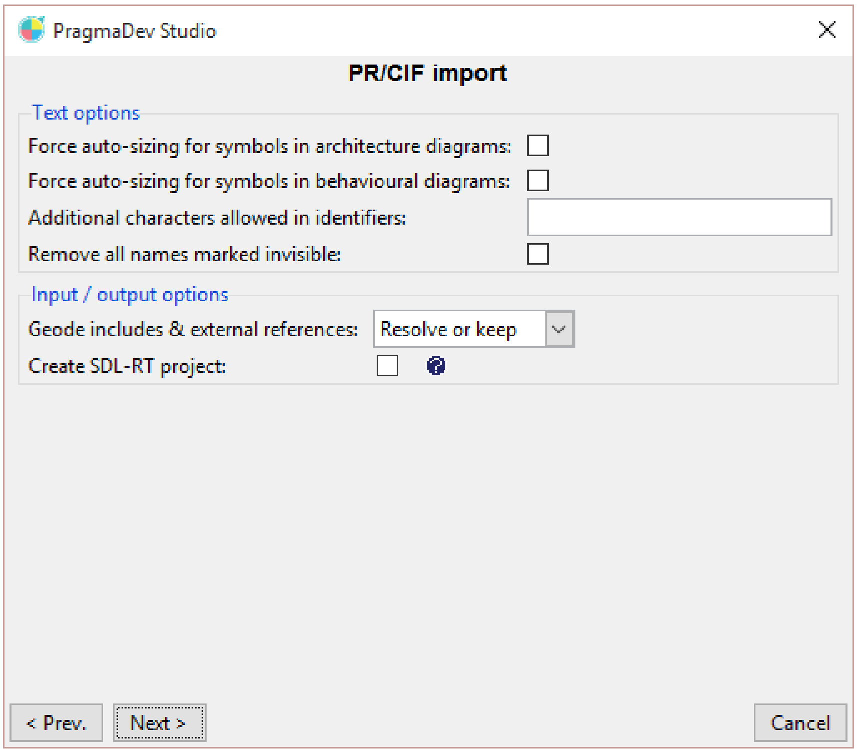


  This option should be checked when importing files from Geode / ObjectGeode.
- *Create one partition for each state*: Only available when CIF comments are ignored. This option will automatically create a new partition for each STATE encountered in the imported PR file.

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 229).

3.5.3 Advanced options panels

The first advanced options panel is the following:



PragmaDev Studio

PR/CIF import

Text options

Force auto-sizing for symbols in architecture diagrams: □

Force auto-sizing for symbols in behavioural diagrams: □

Additional characters allowed in identifiers:

Remove all names marked invisible: □

Input / output options

Geode includes & external references: Resolve or keep

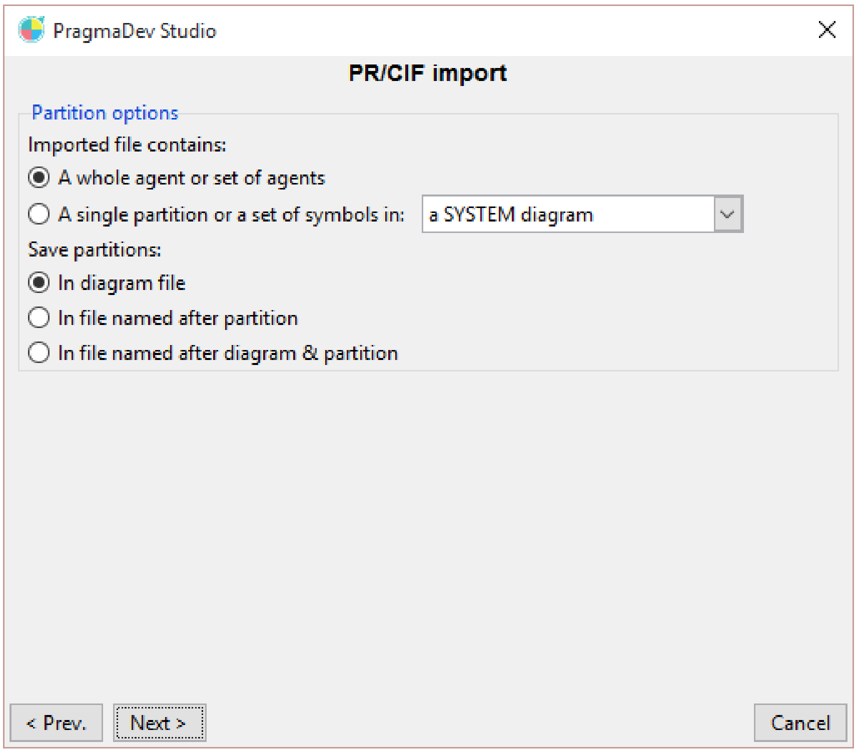
Create SDL-RT project: □

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 62).

  - 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:



PragmaDev Studio

PR/CIF import

Partition options

Imported file contains:

● A whole agent or set of agents

○ A single partition or a set of symbols in: a SYSTEM diagram

Save partitions:

● In diagram file

○ In file named after partition

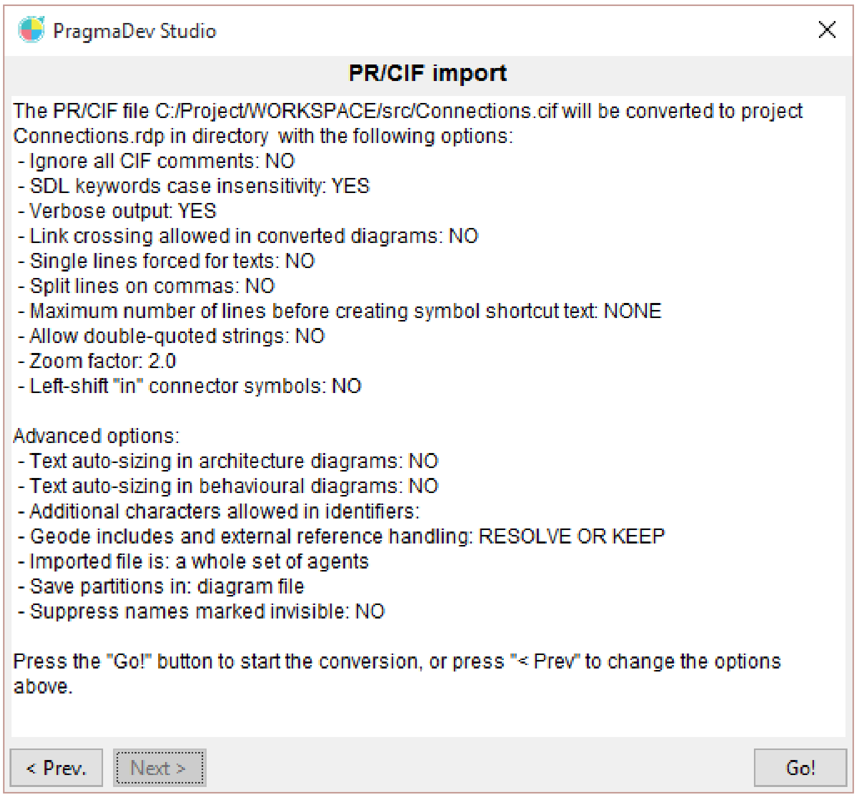
○ In file named after diagram & partition

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.

3.5.4 Summary panel

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



The PR/CIF file C:/Project/WORKSPACE/src/Connections.cif will be converted to project Connections.rdp in directory with the following options:
- Ignore all CIF comments: NO
- SDL keywords case insensitivity: YES
- Verbose output: YES
- Link crossing allowed in converted diagrams: NO
- Single lines forced for texts: NO
- Split lines on commas: NO
- Maximum number of lines before creating symbol shortcut text: NONE
- Allow double-quoted strings: NO
- Zoom factor: 2.0
- Left-shift "in" connector symbols: NO

Advanced options:
- Text auto-sizing in architecture diagrams: NO
- Text auto-sizing in behavioural diagrams: NO
- Additional characters allowed in identifiers:
- Geode includes and external reference handling: RESOLVE OR KEEP
- Imported file is: a whole set of agents
- Save partitions in: diagram file
- Suppress names marked invisible: NO

Press the "Go!" button to start the conversion, or press "< Prev" to change the options above.

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

3.5.5 PR/CIF import progress and result

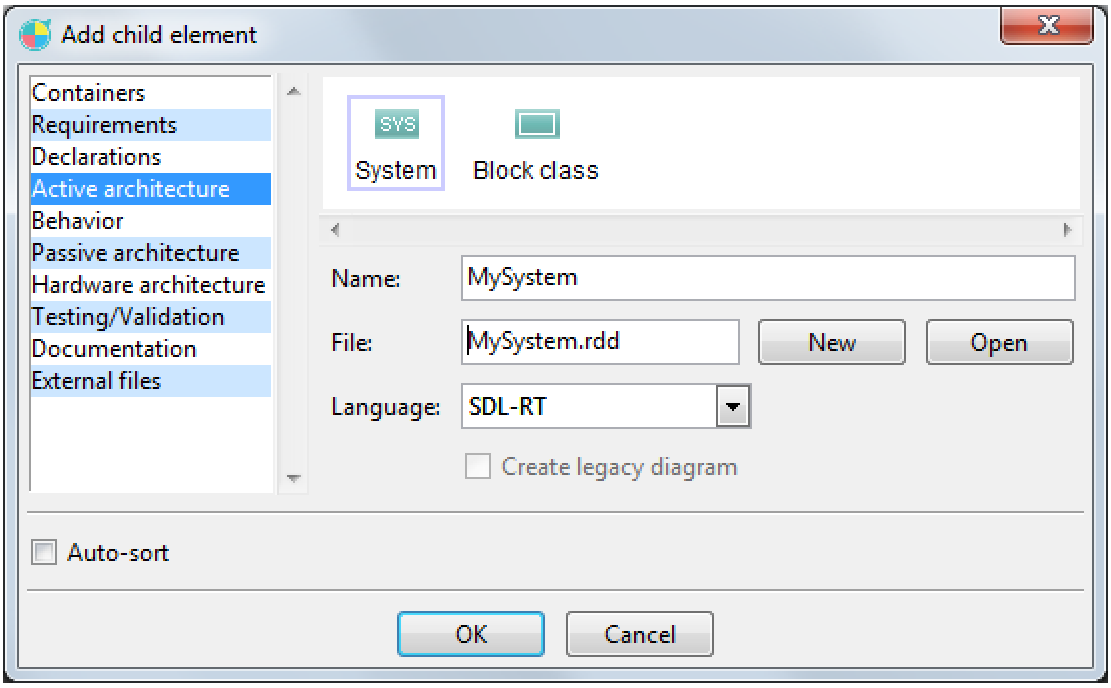
Once the import has started, a dialog will appear displaying all messages returned by the conversion, including warning and error messages if any and progress messages if the "*Verbose*" option was checked. All warnings and errors will also be saved in the converted project and displayed each time the 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.

4 - PragmaDev Developer

4.1 - SDL-RT project

PragmaDev Developer helps software designers to write maintainable and self documented code. The technology used for development describes the architecture and contains a graphical view of the main paths of execution down to the code itself.

PragmaDev Developer is based on SDL-RT technology which combine SDL and C/C++ code in one consistent model. For information on the language itself, please refer to the languages reference documents. PragmaDev projects are generic. In order to work with PragmaDev Developer an SDL-RT system must be created. For that matter start a new project, right click on the project and select addc hild element. Select active architecture and system in the window:



Name: MySystem

File: MySystem.rdd

Language: SDL-RT

□ Create legacy diagram

□ Auto-sort

4.2 - Data and SDL-RT types declarations

4.2.1 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-RT text block; C code can be typed in directly. The C code covers different aspects depending on the SDL-RT 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-RT architecture (blocks, process, and proce-

dures). 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-RT 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-RT procedure will be inserted at the procedure function declaration level. It therefore contains local variables to the procedure.

4.2.2 SDL-RT messages and message lists declaration

SDL-RT 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), addlMsg;
```

4.2.3 SDL-RT timer declaration

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

4.2.4 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 pended tasks on the basis of their priority
- FIFO
  Queue pended 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.

4.2.5 Process declaration

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

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.

4.2.6 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);
```

4.2.7 Class description

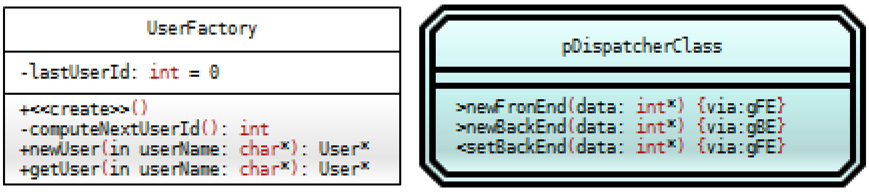
A class is described graphically with a class symbol in a class diagram. The syntax used is the UML syntax:

- 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>`}).

<u>Example:</u>

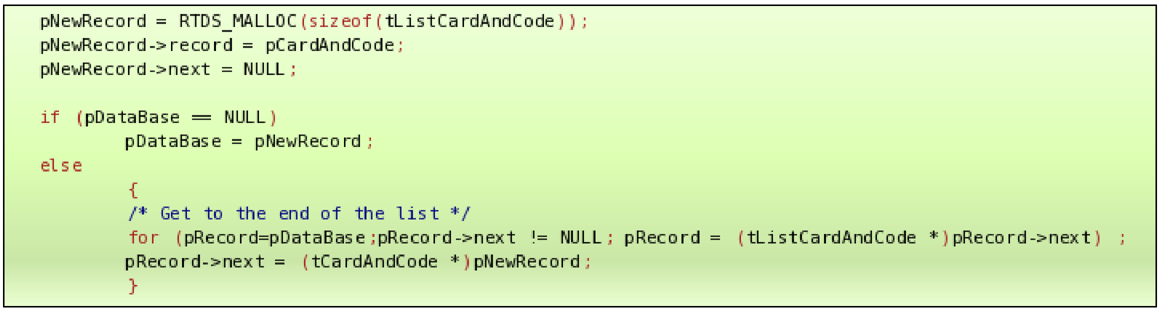

UserFactory
-lastUserId: int = 0
+<<create>>()
-computeNextUserId(): int
+newUser(in userName: char*): User*
+getUser(in userName: char*): User*
pDispatcherClass
>newFronEnd(data: int*) {via:gFE}
>newBackEnd(data: int*) {via:gBE}
<setBackEnd(data: int*) {via:gFE}

4.3 - SDL-RT symbols syntax

4.3.1 Task block

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

<u>Example:</u>



```
pNewRecord = RTDS_MALLOC(sizeof(tListCardAndCode));
pNewRecord->record = pCardAndCode;
pNewRecord->next = NULL;

if (pDataBase == NULL)
        pDataBase = pNewRecord;
else
         {
         /* Get to the end of the list */
         for (pRecord=pDataBase;pRecord->next != NULL; pRecord = (tListCardAndCode *)pRecord->next) ;
        pRecord->next = (tCardAndCode *)pNewRecord;
         }
```

4.3.2 Next state

The syntax in the next state SDL-RT graphical symbol is:

```
<new SDL-RT state>
```

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

It also can be “-”, meaning the state is not changed. This can be particularly useful in transitions attached to a “*” state symbol.

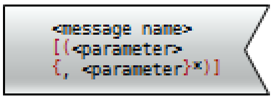
4.3.3 Continuous signals

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-RT state can contain several continuous signal a priority level needs to be defined with the `PRIO` keyword. Lower values correspond to higher priorities. The syntax is:

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

Example:
```
( a > 5 )
PRIO 3
```

4.3.4 Message input



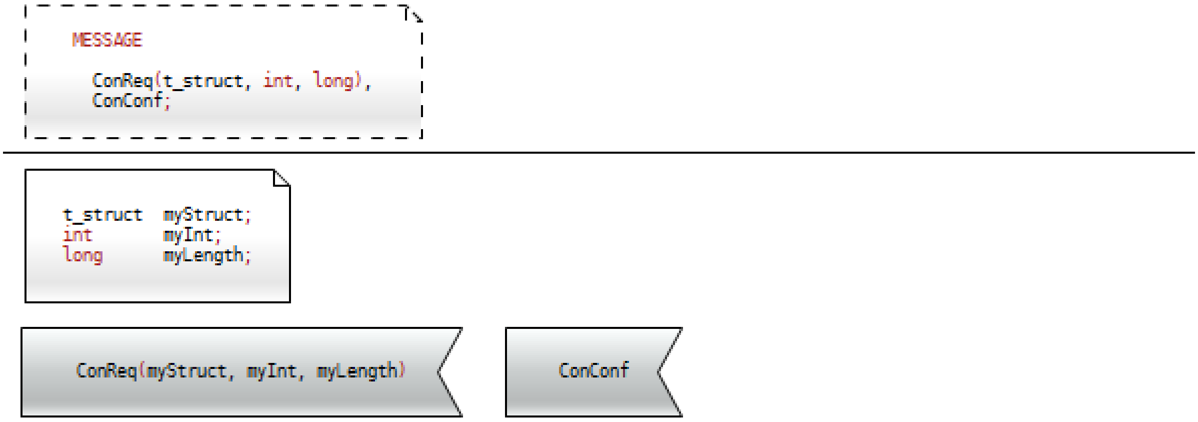
```
<message name>
[(<parameter>
{, <parameter>}*)]
```

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:



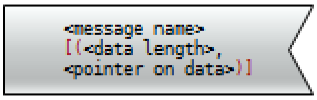
```
MESSAGE
  ConReq(t_struct, int, long),
  ConConf;
```

```
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>)]
```

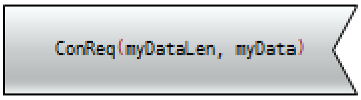
Even though it is not recommended, if a message is declared without any parameter it is possible to transmit undefined parameters with a length and a pointer on the parameter data. In that case it is necessary to declare 2 variables that will be the parameter length and the pointer on the parameters.

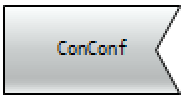
The syntax in the message input symbol is the following:

```
<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.

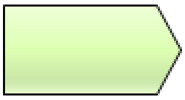
Examples:





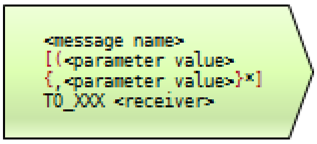
4.3.5 Message output

4.3.5.1 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.

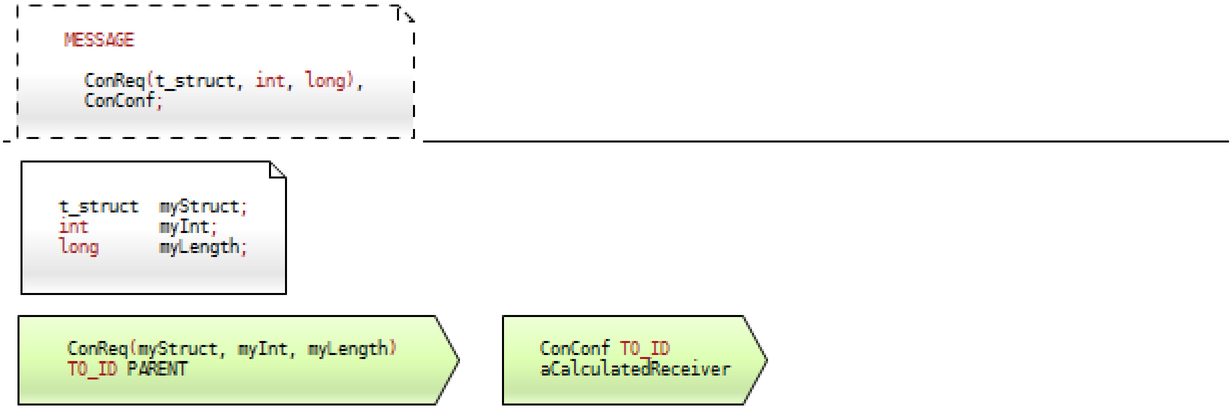


```
<message name>
[(<parameter value>
{,<parameter value>}*]
TO_XXX <receiver>
```

`<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 responsability to free the associated memory if needed.

Examples:



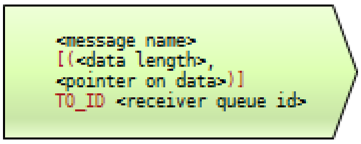
```
MESSAGE
  ConReq(t_struct, int, long),
  ConConf;
```

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

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

```
ConConf TO_ID
aCalculatedReceiver
```

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



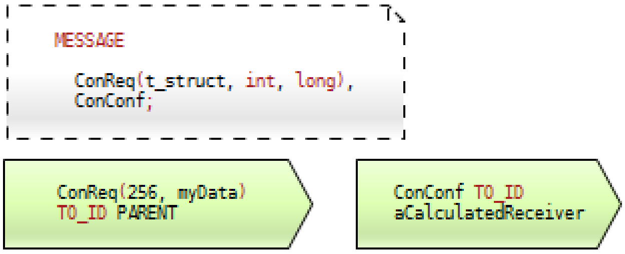
```
<message name>
[(<data length>,
<pointer on data>)]
TO_ID <receiver queue id>
```

`<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:

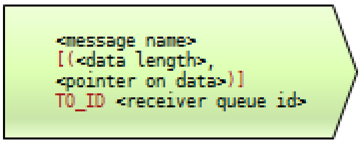


```
MESSAGE
  ConReq(t_struct, int, long),
  ConConf;
```

```
ConReq(256, myData)
TO_ID PARENT
```

```
ConConf TO_ID
aCalculatedReceiver
```

4.3.5.2 Queue Id



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

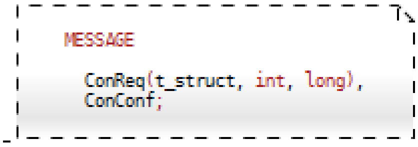
`<message name>[(<parameter value> {,<parameter value>}*] TO_ID <receiver queue id>`

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

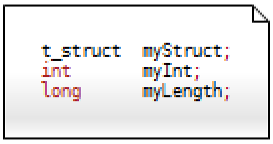
- `<receiver queue id>` is of type `RTDS_QueueId.`
  It can take the value given by the SDL-RT keywords:

`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.

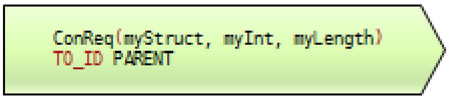
Examples:



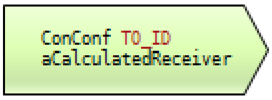
```
MESSAGE
  ConReq(t_struct, int, long),
  ConConf;
```



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

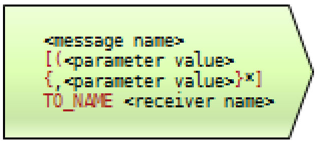


```
ConReq(myStruct, myInt, myLength)
TO_ID PARENT
```



```
ConConf TO_ID
aCalculatedReceiver
```

4.3.5.3 Process name

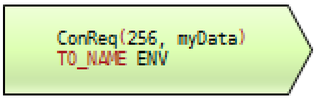


```
<message name>
[(<parameter value>
{,<parameter value>}*]
TO_NAME <receiver name>
```

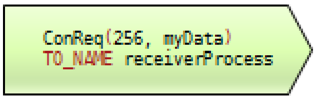
`<message name> [(<parameter value> {,<parameter value>}*] TO_NAME <receiver name>`

`<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-RT system.

Examplees:



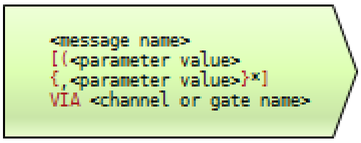
```
ConReq(256, myData)
TO_NAME ENV
```



```
ConReq(256, myData)
TO_NAME receiverProcess
```

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.

4.3.5.4 Via a channel or gate

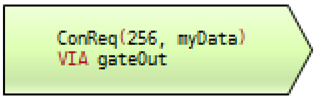


```
<message name>
[(<parameter value>
{,<parameter value>}*]
VIA <channel or gate name>
```

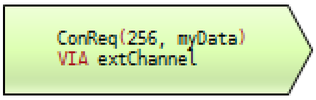
<message name> [(<parameter value> {,<parameter value>}*] VIA <channel or gate name>

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

Examples:



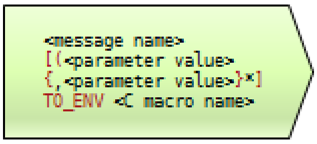
```
ConReq(256, myData)
VIA gateOut
```



```
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 4.3.5.3 applies.

4.3.5.5 Environment

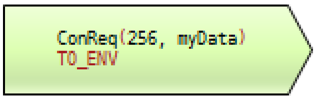


```
<message name>
[(<parameter value>
{,<parameter value>}*]
TO_ENV <C macro name>
```

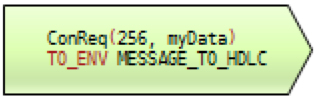
<message name> [(<parameter value> {,<parameter value>}*] TO_ENV <C macro name>

<C macro name> is the name of the macro that will be called when this SDL-RT 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
```



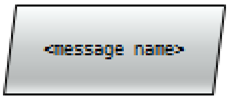
```
ConReq(256, myData)
TO_ENV MESSAGE_TO_HDLC
```

In this second example the generated code will be:

MESSAGE_TO_HDLC(ConReq,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.

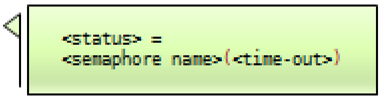
4.3.6 Saved message



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

```
<message name>
```

4.3.7 Semaphore take



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

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

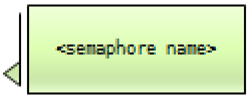
where `<timeout option>` is:

- `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.

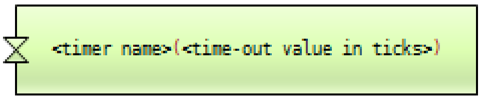
4.3.8 Semaphore give



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

```
<semaphore name>
```

4.3.9 Timer start

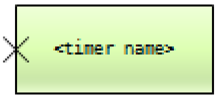


To start a timer the syntax in the ‘start timer SDL-RT graphical symbol’ is :

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

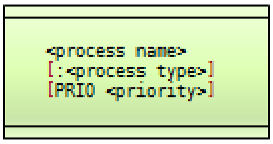
`<time value in tick counts>` is usually an ‘int’ but is RTOS and target dependant.

4.3.10 Timer stop



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

4.3.11 Process



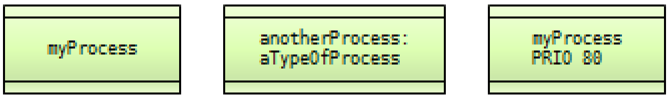
To create a process the syntax in the ‘create process SDL-RT graphical symbol’ is:

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

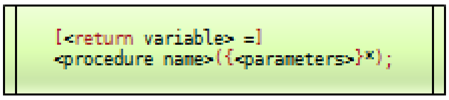
to create one instance of `<process type>` named `<process name>` with priority `<priority>`.

The default priority will be 150.

Examples:



4.3.11.1 Procedure call

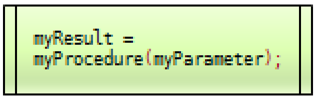


The procedure call symbol is used to call an SDL-RT procedure (see “Procedure declaration” on page 233). Since it is possible to call any C function in an SDL-RT task block it is important to note SDL-RT procedures are different because they know the calling process context, e.g. SDL-RT keywords such as `SENDER`, `OFFSPRING`, `PARENT` are the ones of the calling process.

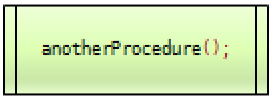
The syntax in the procedure call SDL-RT graphical symbol is the standard C syntax:

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

Examples:



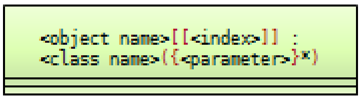
```
myResult =
myProcedure(myParameter);
```



```
anotherProcedure();
```

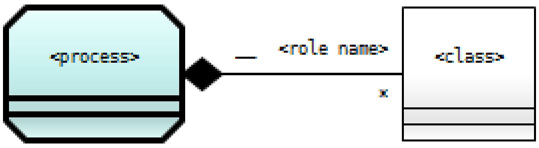
Note: A procedure defined in SDL-RT can not be called directly from a C statement. It has to be called from the procedure call graphical symbol. This is due to the fact that the procedure needs to know the process context so the generated code adds a parameter to the procedure definition and call.

4.3.12 Object initialization



```
<object name>[[<index>]] :
<class name>({<parameter>}*)
```

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:

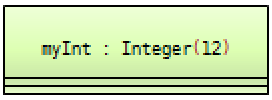


In this case, one or several instances of `<class>` are part of the process `<process>`, depending on the composition's cardinality. These instances are known in the process via a variable named `<role name>`.

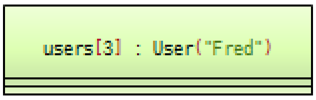
The object initialization symbol may be used like follows:

- 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:



```
myInt : Integer(12)
```



```
users[3] : User("Fred")
```

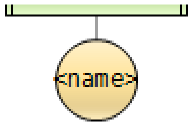
Note: In the generated code, using a task block containing:

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

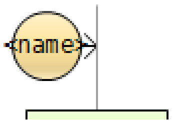
is exactly the same than using the object initialization symbol, except for the added semantics checking in the case of compositions with a maximum cardinality of 1.

4.3.13 Connectors

Connectors are a way to make the execution continue at another spot in the diagram. A connector out is also called a `JOIN`, and a connector in a `LABEL`:



Connector out



Connector in

Connectors are used to:

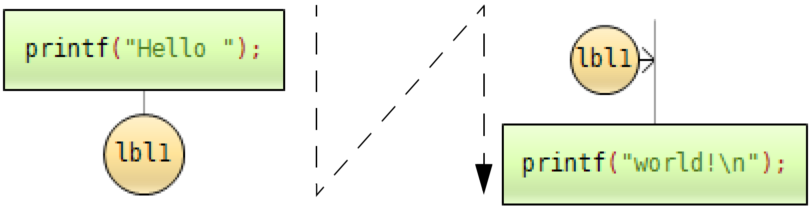
- split a transition into several pieces so that the diagram stays legible,
- to gather different branches to a same point.

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:



4.3.14 Decision

The expression to evaluate in the symbol can contain:

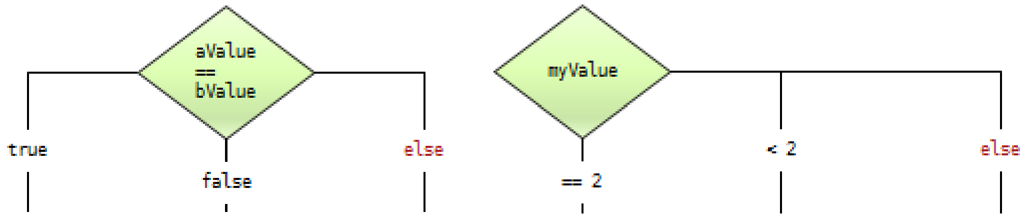
- any standard C expression that returns a C true/false expression,
- an expression that will be evaluated against the values in the decision branches.

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:



4.3.15 SDL-RT keywords

4.3.15.1 Global keywords

The following SDL-RT keywords are defined and can be used in all 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.

4.3.15.2 Local keywords

The following keywords are dedicated to specific symbols :

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

Table 4: Keywords in symbols

keywords	concerned symbols
USE SDL_MESSAGE_LIST	text symbol

Table 4: Keywords in symbols

4.4 - 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.

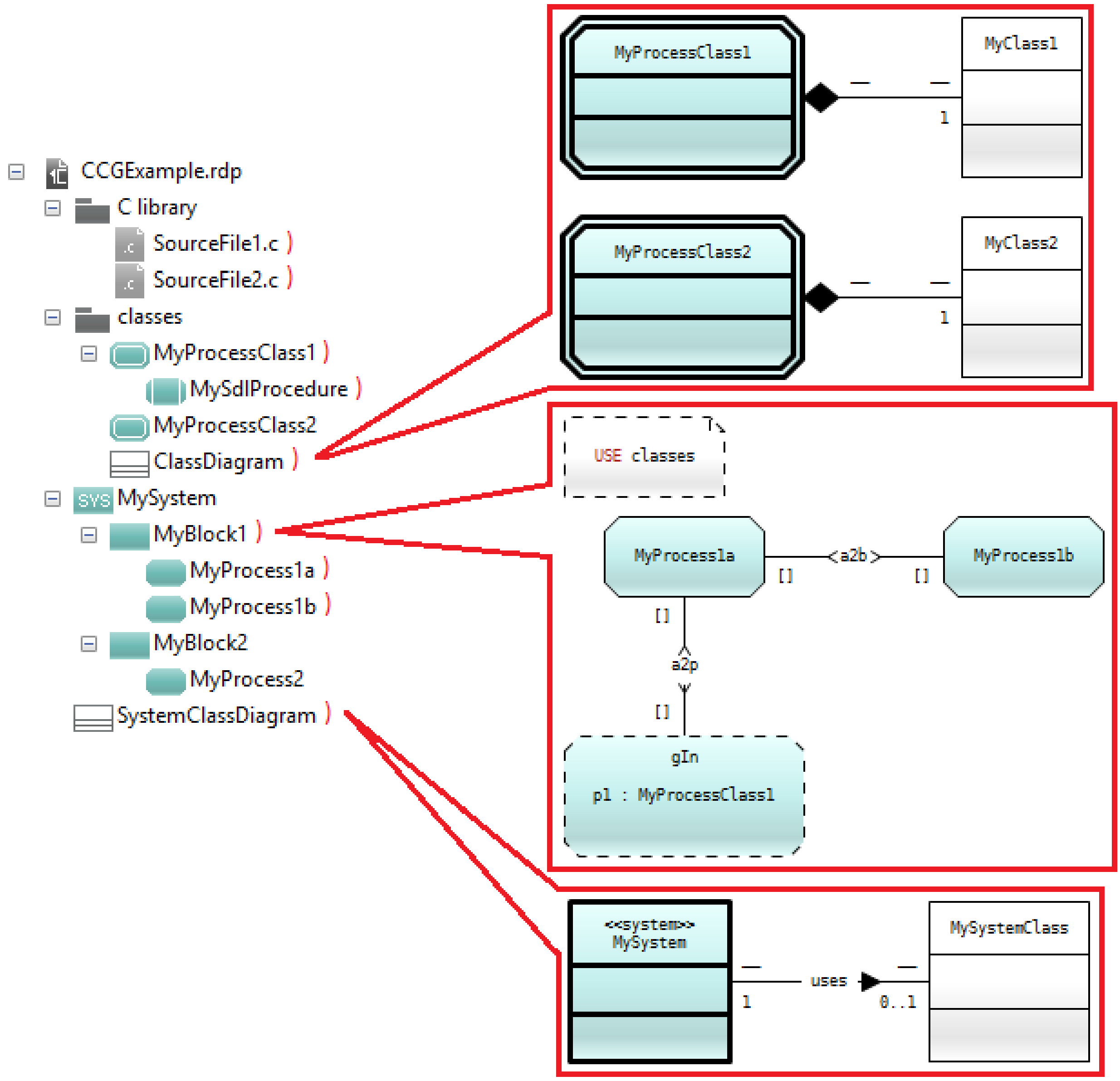
Clicking on the button or selecting the “Generate code if needed...” or “Force code generation...” in the “Generation / Build” menu generates the code, but also runs the whole build process if enabled in the generation options (see “Profiles” on page 248).

4.4.1 Concerned elements

The elements concerned by the code generation are the following:

- All elements in the agent sub-tree, including SDL-RT 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-RT 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-RT diagrams;
- The class `MyClass1`, since it is linked to `MyProcessClass1` in diagram `ClassDiagram`;

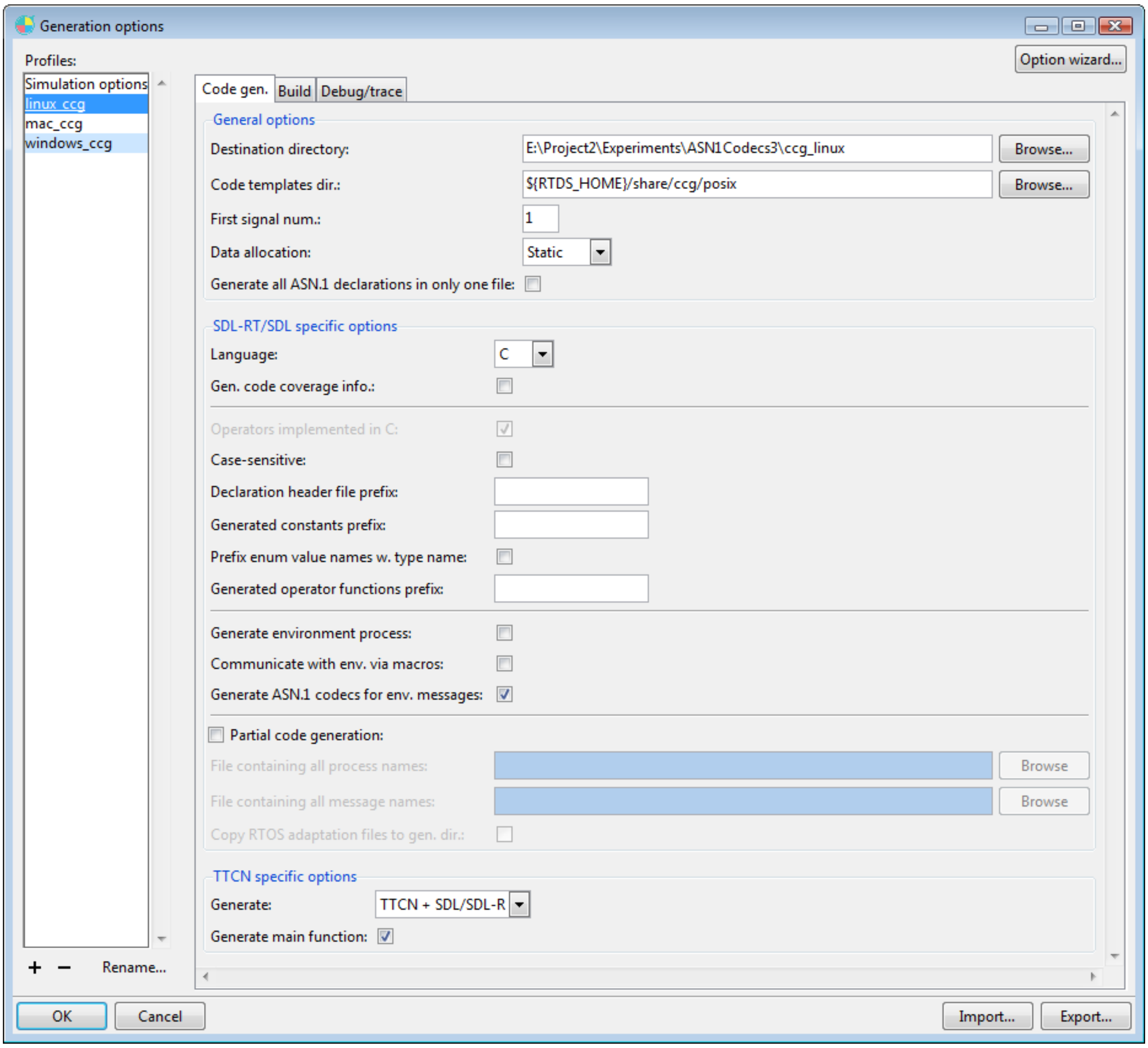
- The class `MySystemClass`, since it is linked to `MySystem` in diagram `SystemClassDiagram` and `MySystem` is a parent of `MyBlock1`.

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.

4.4.2 Profiles

4.4.2.1 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.



General options

Destination directory: E:\Project2\Experiments\ASN1Codecs3\ccg_linux

Code templates dir.: ${RTDS_HOME}/share/ccg/posix

First signal num.: 1

Data allocation: Static

Generate all ASN.1 declarations in only one file: □

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: □

Operators implemented in C: ☑

Case-sensitive: □

Declaration header file prefix:

Generated constants prefix:

Prefix enum value names w. type name: □

Generated operator functions prefix:

Generate environment process: □

Communicate with env. via macros: □

Generate ASN.1 codecs for env. messages: ☑

□ Partial code generation:

File containing all process names:

File containing all message names:

Copy RTOS adaptation files to gen. dir.: □

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑

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 254.

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. Please note some of these options are only meaningful for C code generation from SDL systems, which is only available in PragmaDev Studio, and not in PragmaDev Developer. Please refer to section “SDL C code generation” on page 403 for more details.

4.4.2.1.1 Code generation options

This tab is displayed above (paragraph “Description” on page 248). 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.
  - *Data allocation* can be static or dynamic. This option has impact for `String` and `Bag` types only. In static mode, the types are generated as array of static size. This size if defined in the RTDS_Set.h file with the macro `RTDS_MAX_RECORD_OF` set to 256 by default. In dynamic mode, the size of string and bag are dynamically handled, memory is reallocated for each new element add the the bag or the string.
  - *Generate all ASN.1 declarations in only one file.* By default, each ASN.1 file present in a project will be generated as a unique header file. If this option is checked, all ASN.1 declarations will be generated in only one header file. This is to avoid loop dependencies issue which are supported in ASN.1.
- *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 “Model coverage” on page 336.
  - 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.
- The *Communicate with env. via macros* option is only meaningful for a SDL-RT system. If it is checked, message output with the `TO_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 SDL-RT system, code generation will fail.
- The *Generate ASN.1 codecs for env. messages* option is only meaningful for SDL systems. If it is checked, communication with the environment is assumed to be done via ASN.1 encoded data, and PragmaDev Studio will automatically generate the encoders and decoders for the incoming and outgoing messages for the system, as well as the code needed to send an ASN.1 encoded message from the system to the environment, and from the environment to the system. Refer to “ASN.1 codecs for environment messages” on page 403 for more details. If both this option and *Generate environment process* are unchecked, and the SDL system doesn’t include an explicit environment process, 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`).

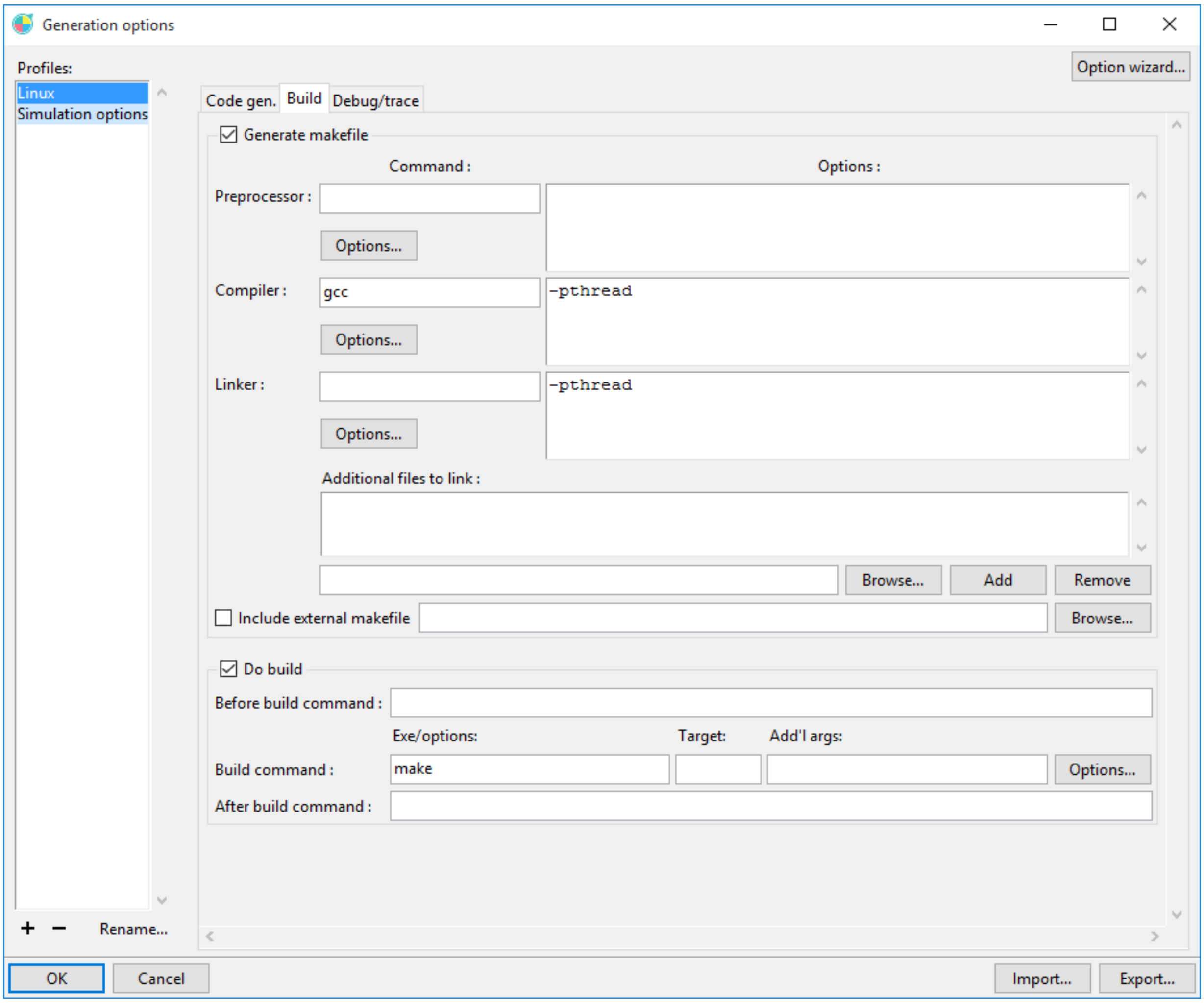
  If the *Copy RTOS adaptation files to gen. dir.* option is checked, all files in the profile directory used in the build will be copied to the generation directory, and the build process will use these files and not those in the PragmaDev Studio installation directory. This allows to have a completely 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 307.

- *TTCN specific options* group:
  These options are specific to TTCN C++ code generation (see “C++ code generation” on page 351).

4.4.2.1.2 Build options

The tab for build options looks like follows:



Generation options

Profiles: Linux, Simulation options

☑ Generate makefile

Preprocessor :

Compiler : gcc — Options : -pthread

Linker : — Options : -pthread

Additional files to link :

☐ Include external makefile

☑ Do build

Before build command :

Build command : make

After build command :

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.

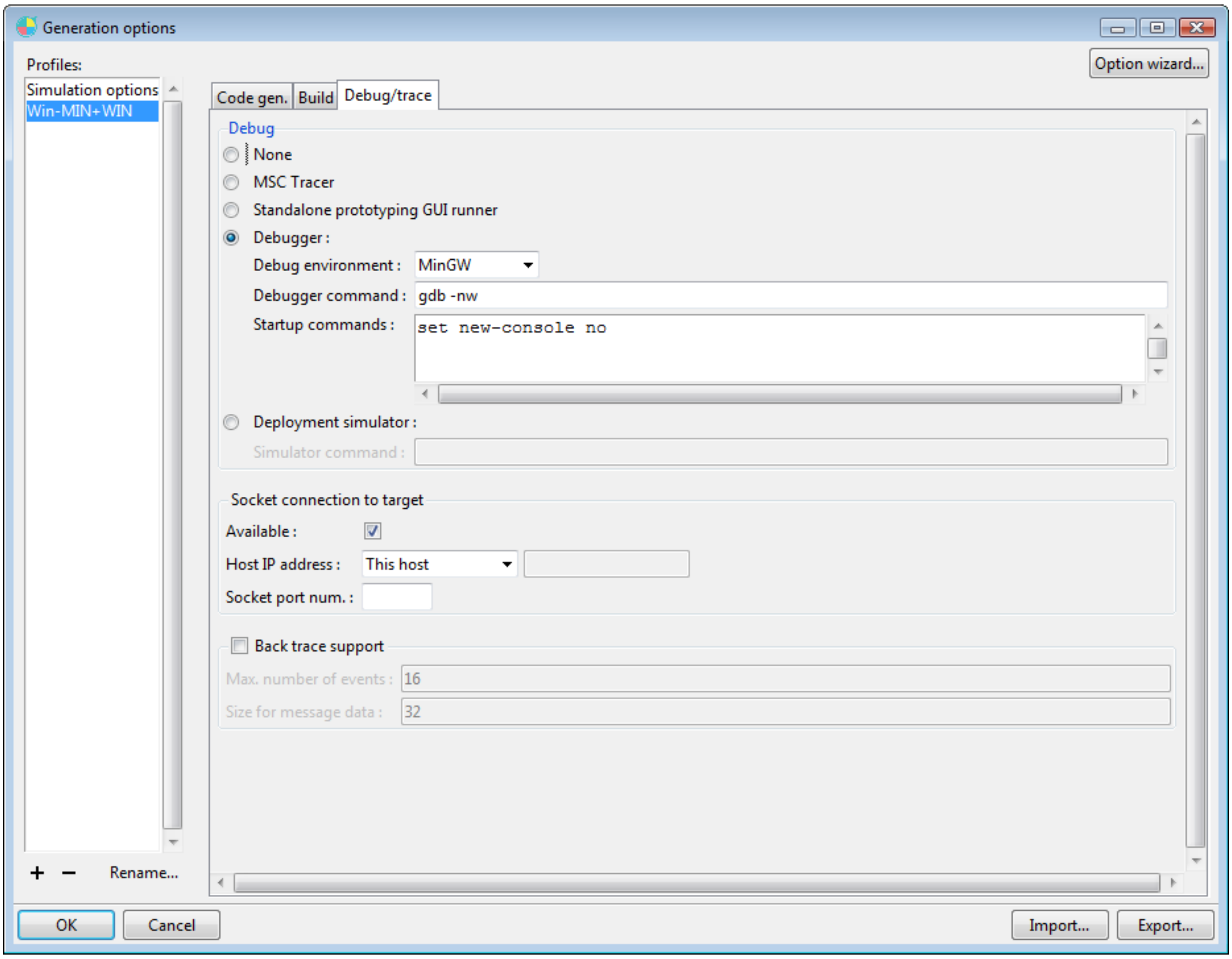
- If the *Do build* check box is set, the actual compilation will also be run by the code generation operation. The command run for the compilation itself is splitted in 3 parts:
  - The executable name with the options, e.g "`make`", or "`make -f MyMakefile`";
  - The target name, usually empty or "`RTDS_ALL`";
  - 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`

4.4.2.1.3 Debug and trace options

The tab for debug and trace options looks like follows:



Generation options

Profiles:
Simulation options
Win-MIN+WIN

Debug
○ None
○ MSC Tracer
○ Standalone prototyping GUI runner
● Debugger :
Debug environment : MinGW
Debugger command : gdb -nw
Startup commands :

```
set new-console no
```

○ Deployment simulator :
Simulator command :

Socket connection to target
Available : ☑
Host IP address : This host
Socket port num. :

□ Back trace support
Max. number of events : 16
Size for message data : 32

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 *Standalone Prototyping GUI Runner*, the profile is also for generating the final code, but the generated executable will be able to connect to a prototyping GUI running within PragmaDev Developer or PragmaDev Studio.
- 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.

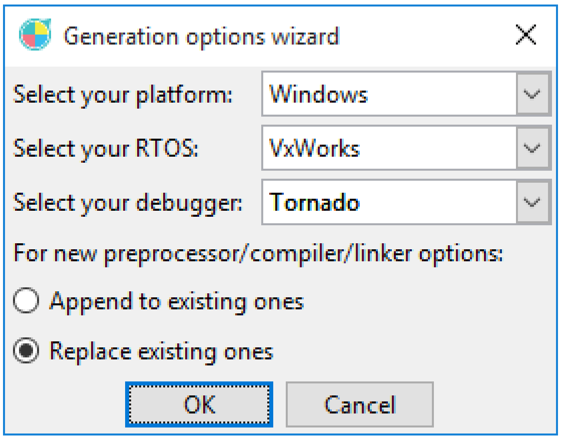
The options in the *Socket connection to target* group configure the host to target communication. They are used by the target program to send trace information:

- to PragmaDev Studio graphical debugger during a debug session;
- to PragmaDev MSC Tracer if its support has been activated;
- to the prototyping GUI running in PragmaDev Developer or PragmaDev Studio if the generation is for the standalone prototyping GUI runner.

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.

4.4.2.2 Option wizard

Almost all of the options described in paragraph “Description” on page 248 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:



Generation options wizard

Select your platform: Windows

Select your RTOS: VxWorks

Select your debugger: Tornado

For new preprocessor/compiler/linker options:

○ Append to existing ones

● Replace existing ones

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.

4.4.2.3 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:

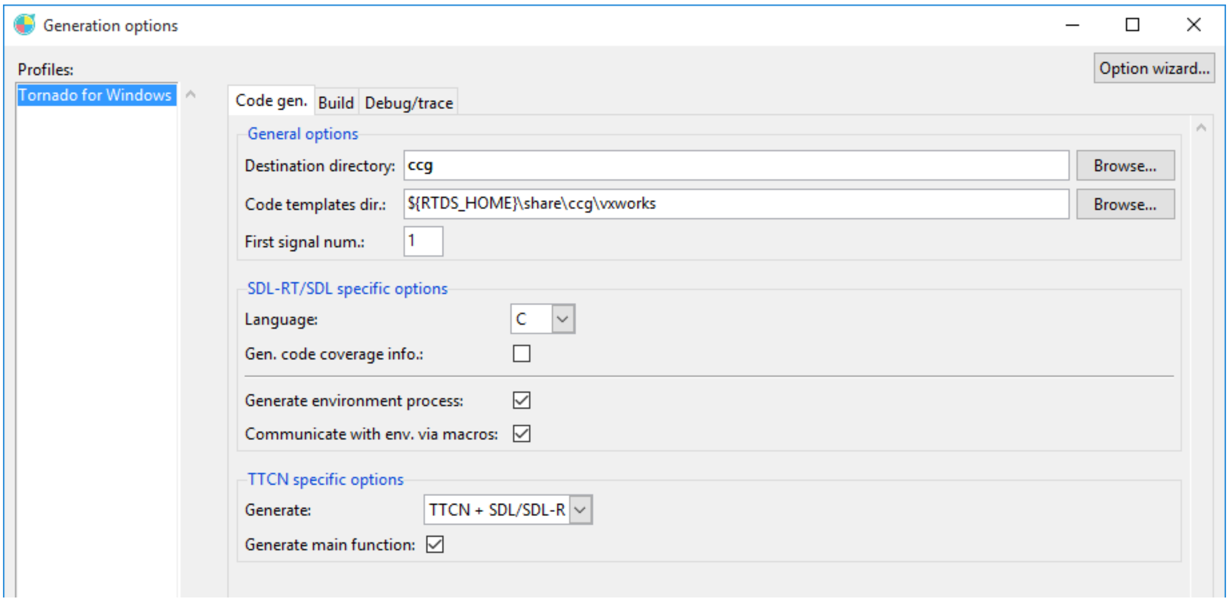
- `gdbsimnt`
  When debugging on Windows host with VxSim
- `gdbsimso`
  When debugging on Solaris host with VxSim
- `gdbi86`
  When debugging on Pentium based targets whatever the communication link is (serial or ethernet)
- `gdbppc`
  When debugging on PowerPC based targets whatever the communication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

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

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: □

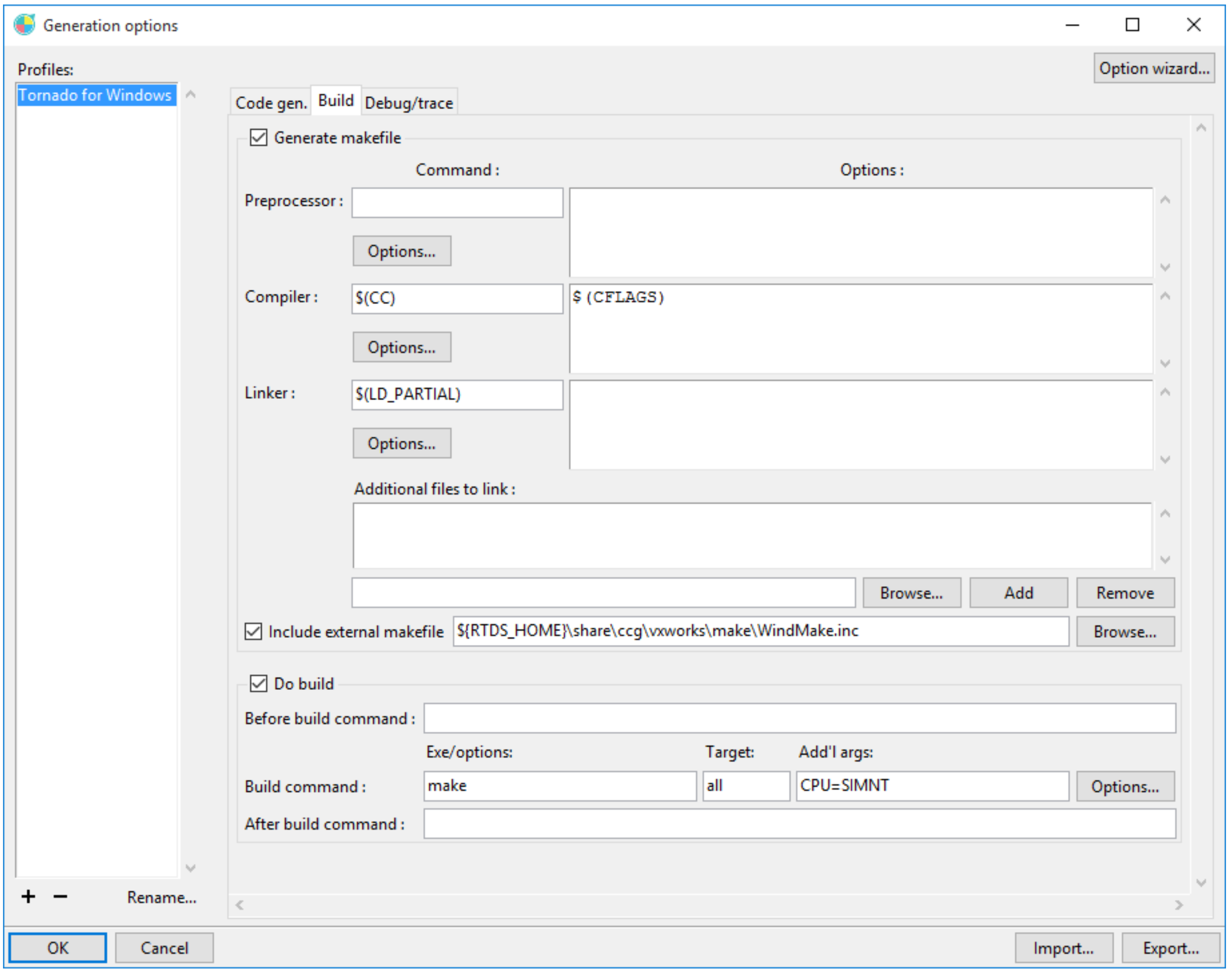
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑



Generation options

Profiles: Tornado for Windows

Code gen. | Build | Debug/trace

☑ Generate makefile

	Command :	Options :
Preprocessor :		
Compiler :	$(CC)	$(CFLAGS)
Linker :	$(LD_PARTIAL)	

Additional files to link :

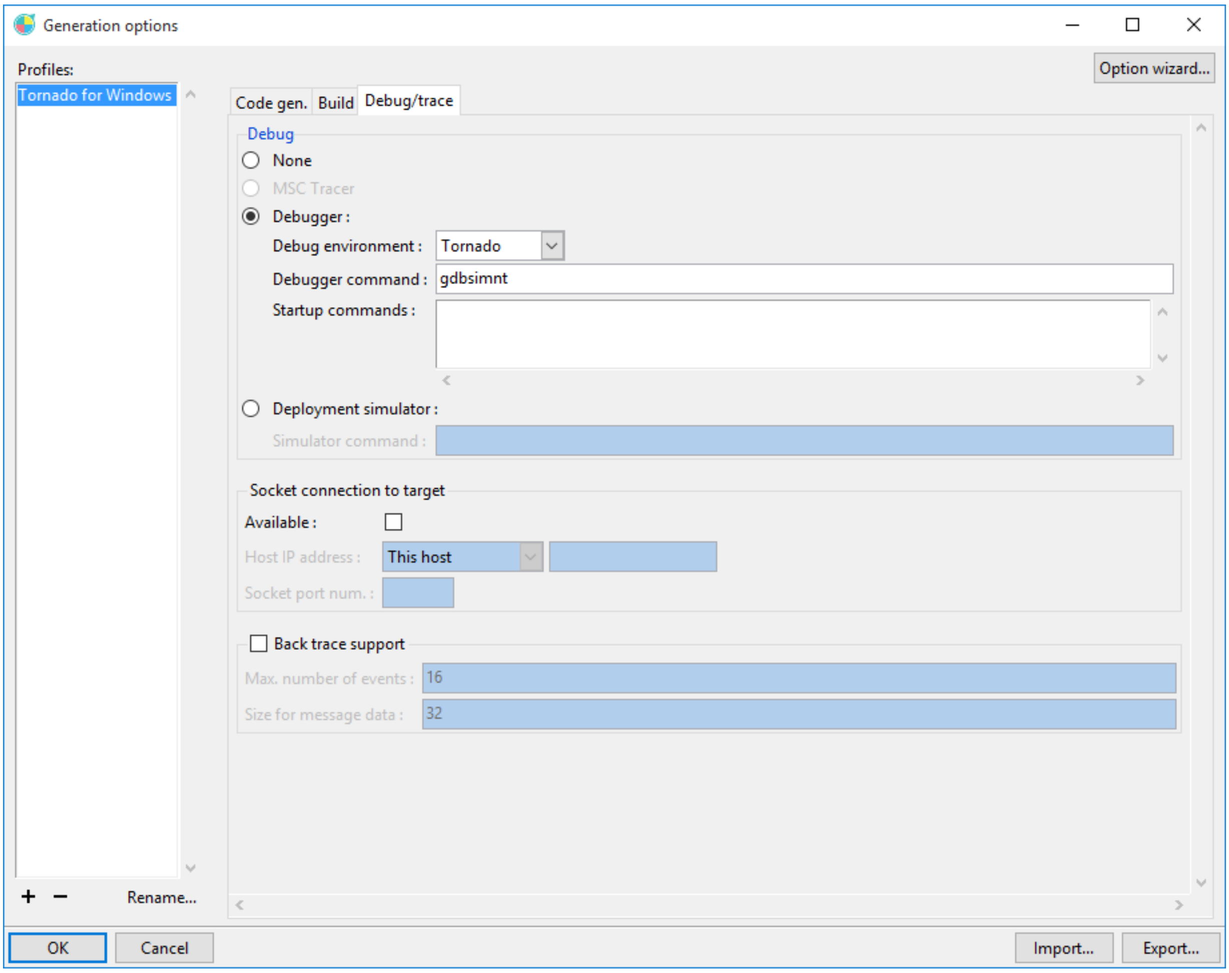
☑ Include external makefile ${RTDS_HOME}\share\ccg\vxworks\make\WindMake.inc

☑ Do build

Before build command :

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

After build command :



Generation options

Profiles:

Tornado for Windows

Code gen. Build Debug/trace

Option wizard...

Debug

○ None

○ MSC Tracer

● Debugger :

Debug environment : Tornado

Debugger command : gdbsimnt

Startup commands :

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : □

Host IP address : This host

Socket port num. :

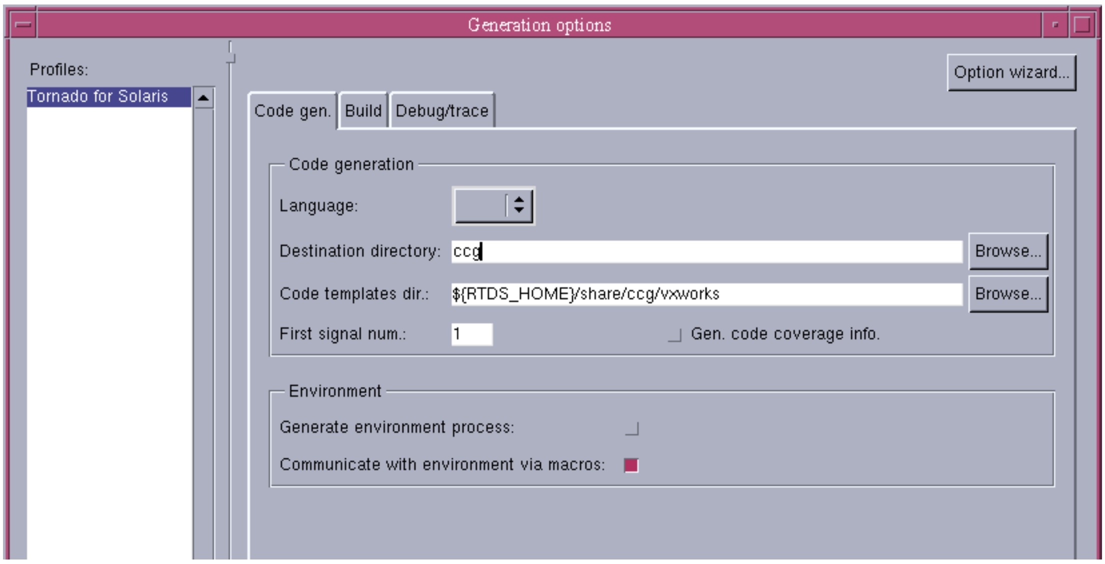
□ Back trace support

Max. number of events : 16

Size for message data : 32

Typical generation profile to debug with VxSim on Windows

On Solaris:



Profiles:
Tornado for Solaris

Code generation

Language:

Destination directory: ccg

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

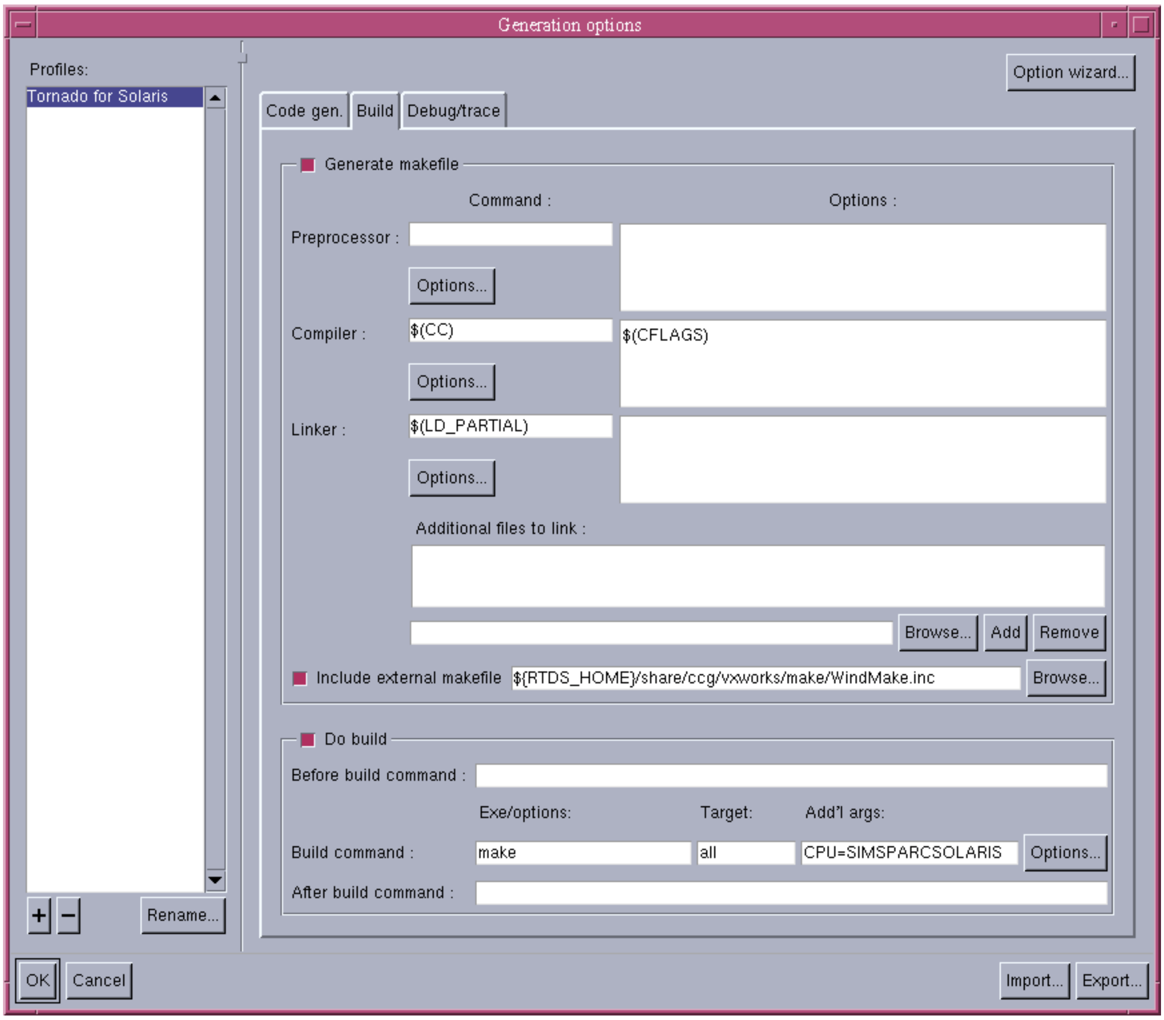
First signal num.: 1

☐ Gen. code coverage info.

Environment

Generate environment process: ☐

Communicate with environment via macros: ☑



Profiles:
Tornado for Solaris

☑ Generate makefile

	Command :	Options :
Preprocessor :		
Compiler :	$(CC)	$(CFLAGS)
Linker :	$(LD_PARTIAL)	

Additional files to link :

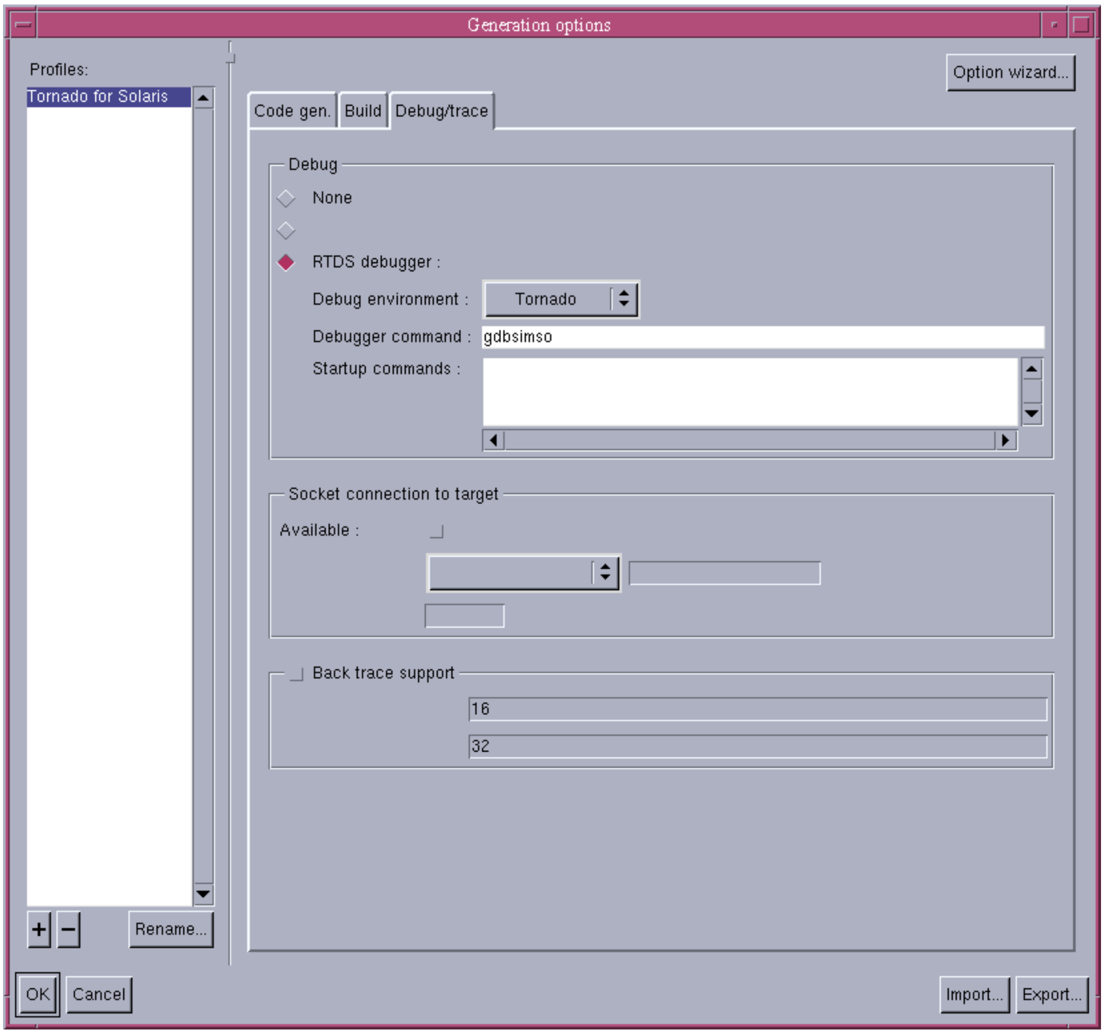
☑ Include external makefile ${RTDS_HOME}/share/ccg/vxworks/make/WindMake.inc

☑ Do build

Before build command :

	Exe/options:	Target:	Add'l args:
Build command :	make	all	CPU=SIMSPARCSOLARIS

After build command :



Profiles:

Tornado for Solaris

Debug

○ None

○

● RTDS debugger :

Debug environment : Tornado

Debugger command : gdbsimso

Startup commands :

Socket connection to target

Available : □

□ Back trace support

16

32

Typical generation profile to debug with VxSim on Solaris

4.4.2.4 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:

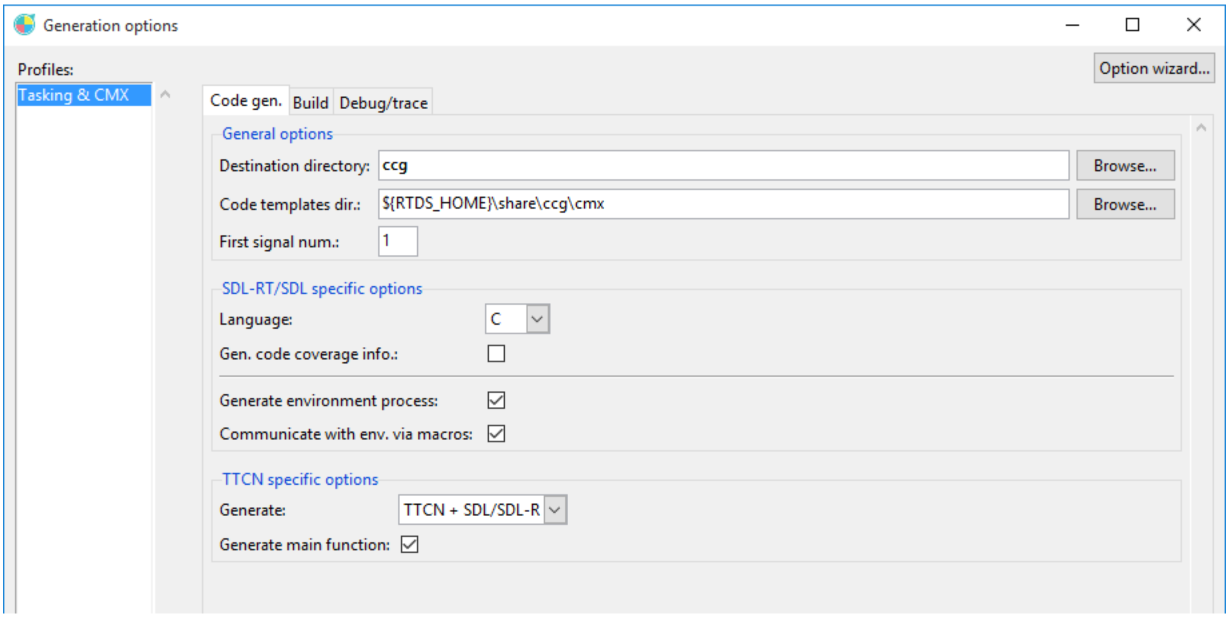
- Timer values are set in number of system ticks,
- CMX supports only counting semaphores with a FIFO queueing mechanism. SDL-RT mutex and binary semaphores have been mapped to counting semaphore,
- The SDL-RT process priorities are the ones of CMX. The default value is 150.
- When an SDL-RT task is 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.



Profiles: Tasking & CMX

General options

Destination directory: ccg

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

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: □

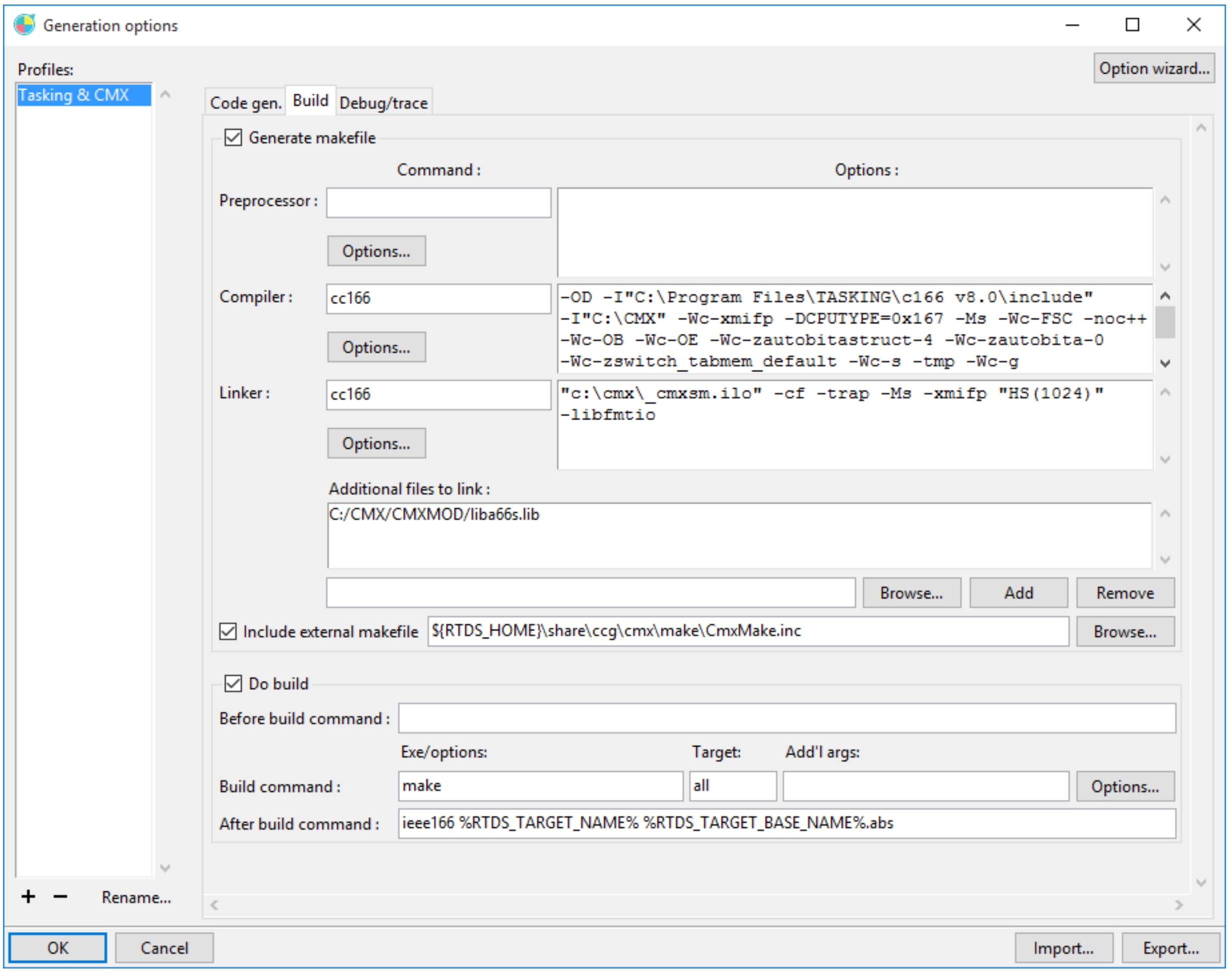
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑



Profiles: Tasking & CMX

☑ Generate makefile

Command : / Options :

Preprocessor :

Compiler : cc166

```
-OD -I"C:\Program Files\TASKING\c166 v8.0\include"
-I"C:\CMX" -Wc-xmifp -DCPUTYPE=0x167 -Ms -Wc-FSC -noc++
-Wc-OB -Wc-OE -Wc-zautobitastruct-4 -Wc-zautobita-0
-Wc-zswitch_tabmem_default -Wc-s -tmp -Wc-g
```

Linker : cc166

```
"c:\cmx\_cmxsm.ilo" -cf -trap -Ms -xmifp "HS(1024)"
-libfmtio
```

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

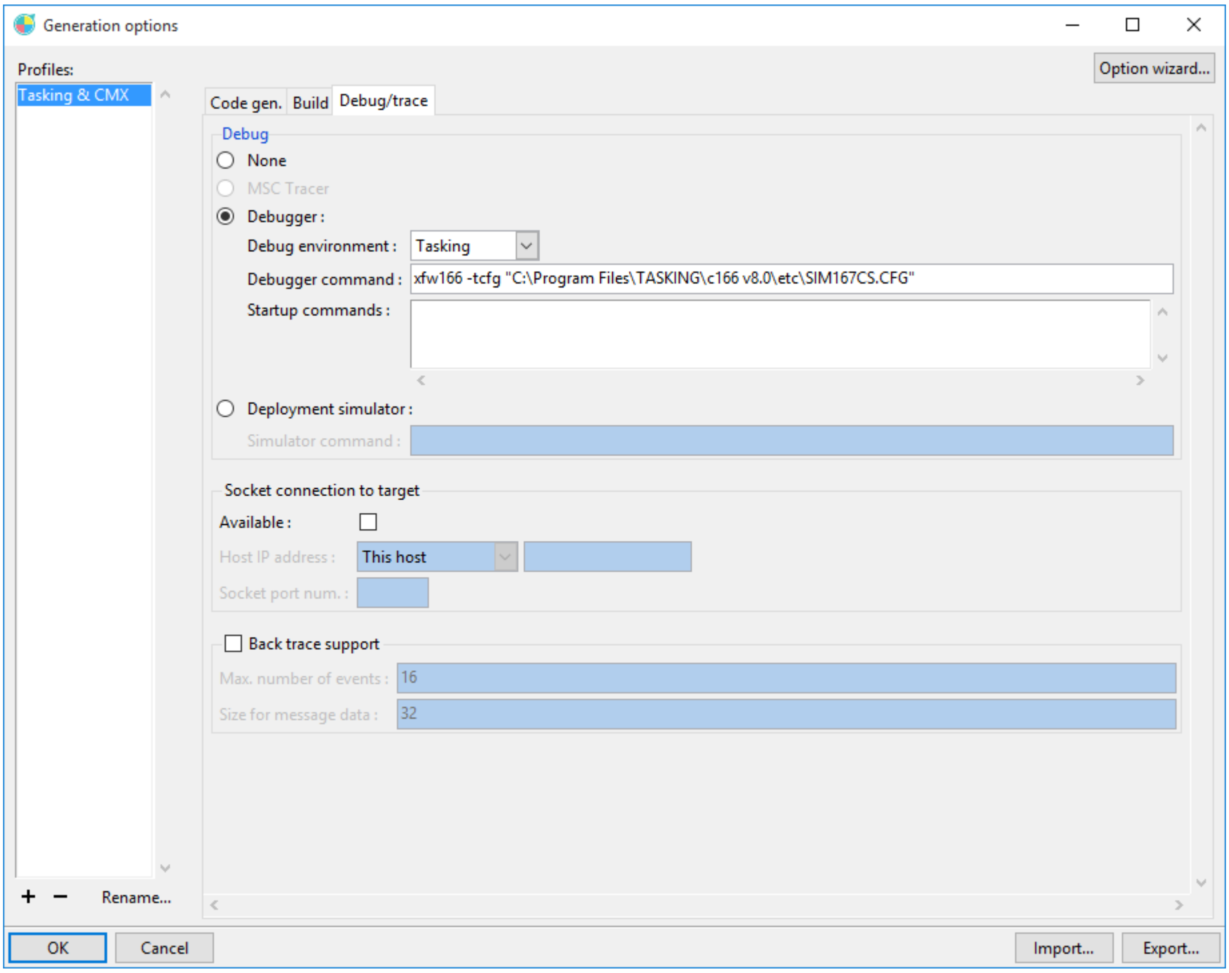
☑ Include external makefile ${RTDS_HOME}\share\ccg\cmx\make\CmxMake.inc

☑ Do build

Before build command :

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

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



Generation options

Profiles: Tasking & CMX

Debug

○ None

○ MSC Tracer

● Debugger :

Debug environment : Tasking

Debugger command : xfw166 -tcfg "C:\Program Files\TASKING\c166 v8.0\etc\SIM167CS.CFG"

Startup commands :

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : □

Host IP address : This host

Socket port num. :

□ Back trace support

Max. number of events : 16

Size for message data : 32

Typical generation profile to debug with Tasking a CMX application

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

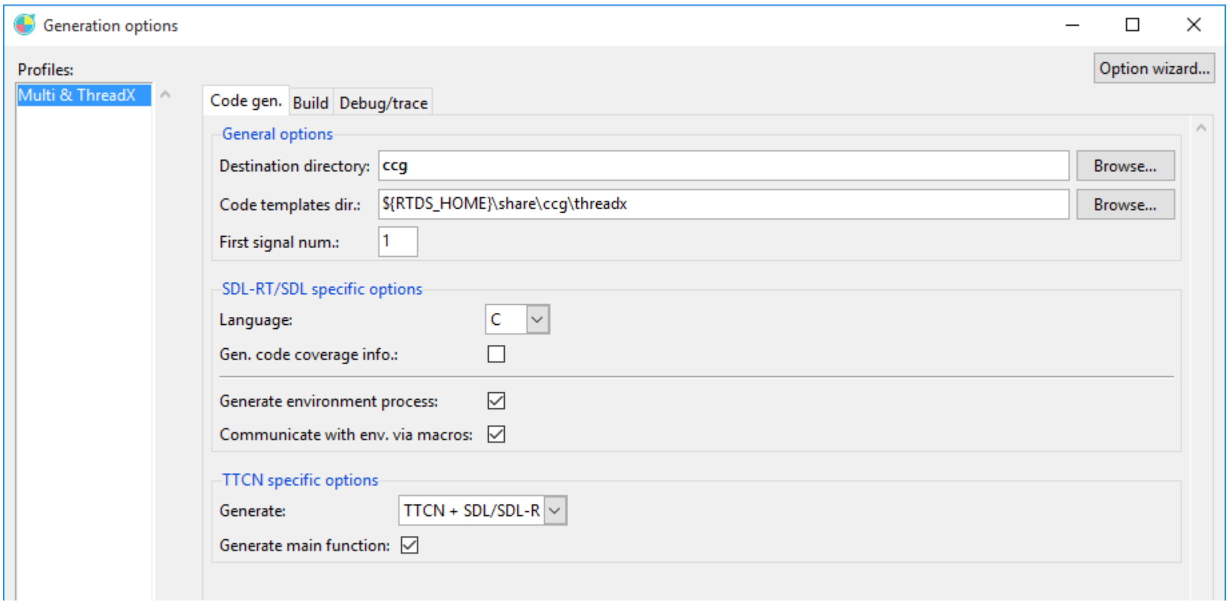
4.4.2.5 ThreadX profile

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:



Generation options

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.: ☐

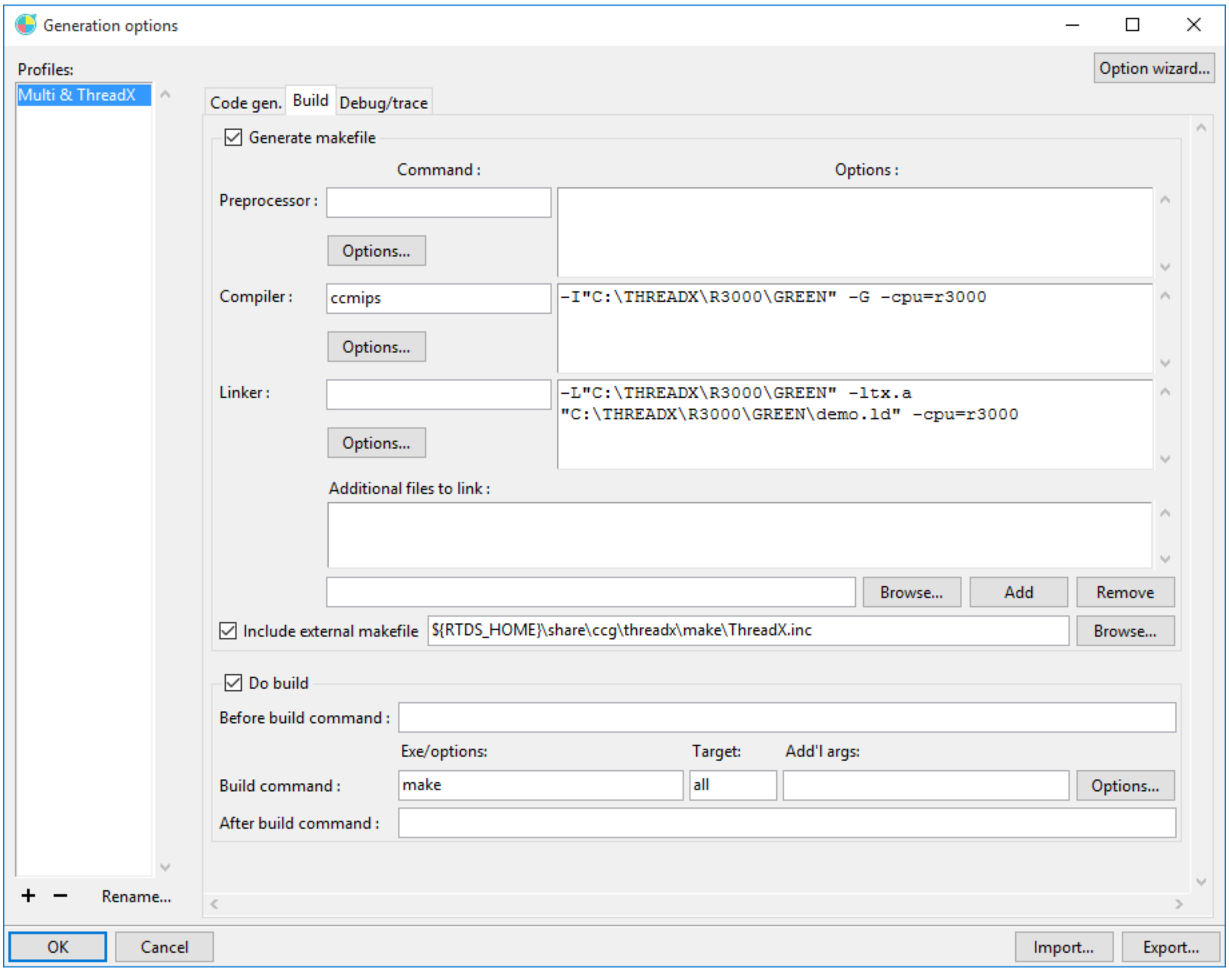
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑



Generation options

Profiles: Multi & ThreadX

Code gen. | Build | Debug/trace

☑ Generate makefile

Command : / Options :

Preprocessor :

Compiler : ccmips

```
-I"C:\THREADX\R3000\GREEN" -G -cpu=r3000
```

Linker :

```
-L"C:\THREADX\R3000\GREEN" -ltx.a
"C:\THREADX\R3000\GREEN\demo.ld" -cpu=r3000
```

Additional files to link :

☑ Include external makefile ${RTDS_HOME}\share\ccg\threadx\make\ThreadX.inc

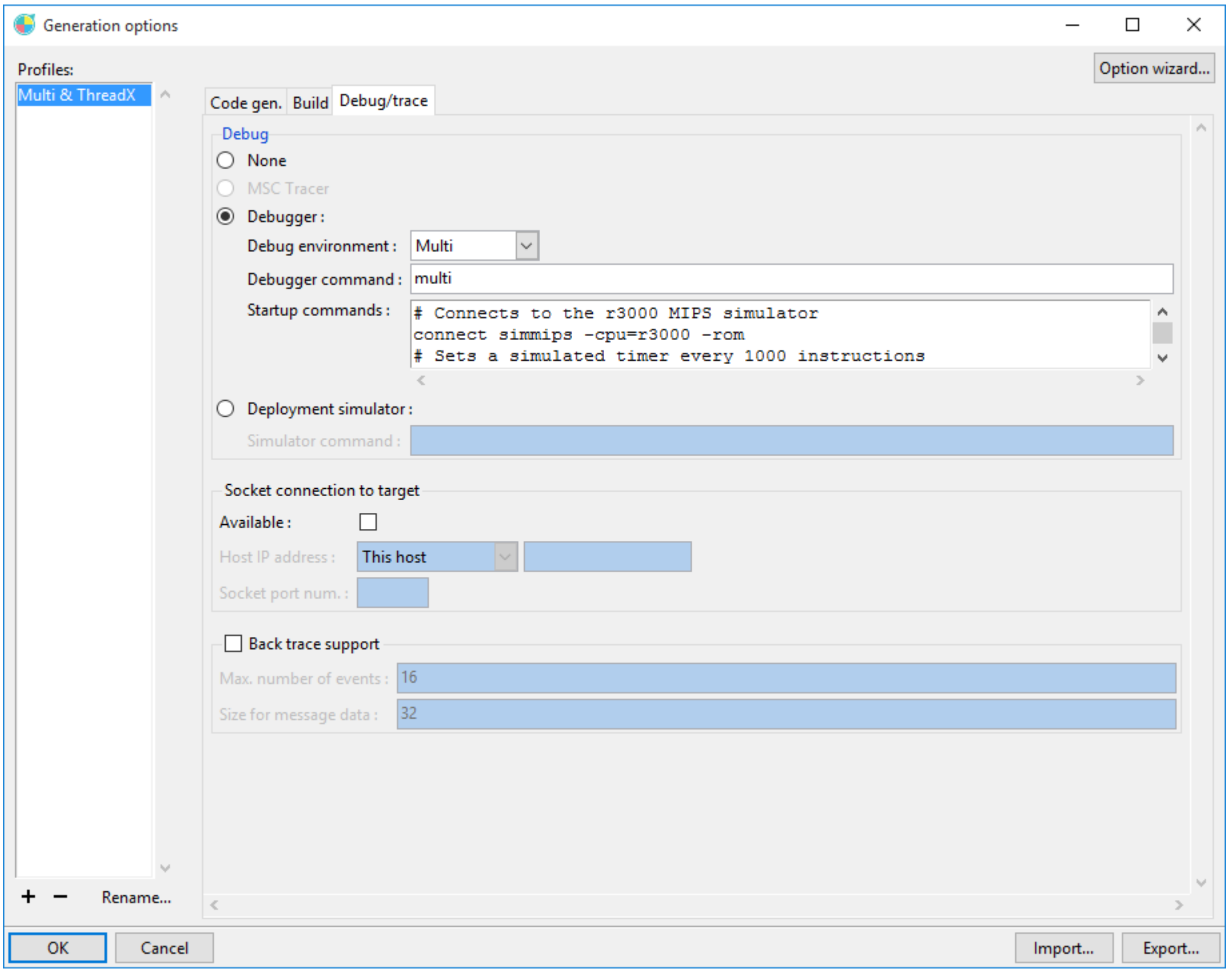
☑ Do build

Before build command :

Exe/options: / Target: / Add'l args:

Build command : make / all

After build command :



Multi & ThreadX

Debug

○ None

○ MSC Tracer

● Debugger :

Debug environment : Multi

Debugger command : multi

Startup commands :

```
# Connects to the r3000 MIPS simulator
connect simmips -cpu=r3000 -rom
# Sets a simulated timer every 1000 instructions
```

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : □

Host IP address : This host

Socket port num. :

□ Back trace support

Max. number of events : 16

Size for message data : 32

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

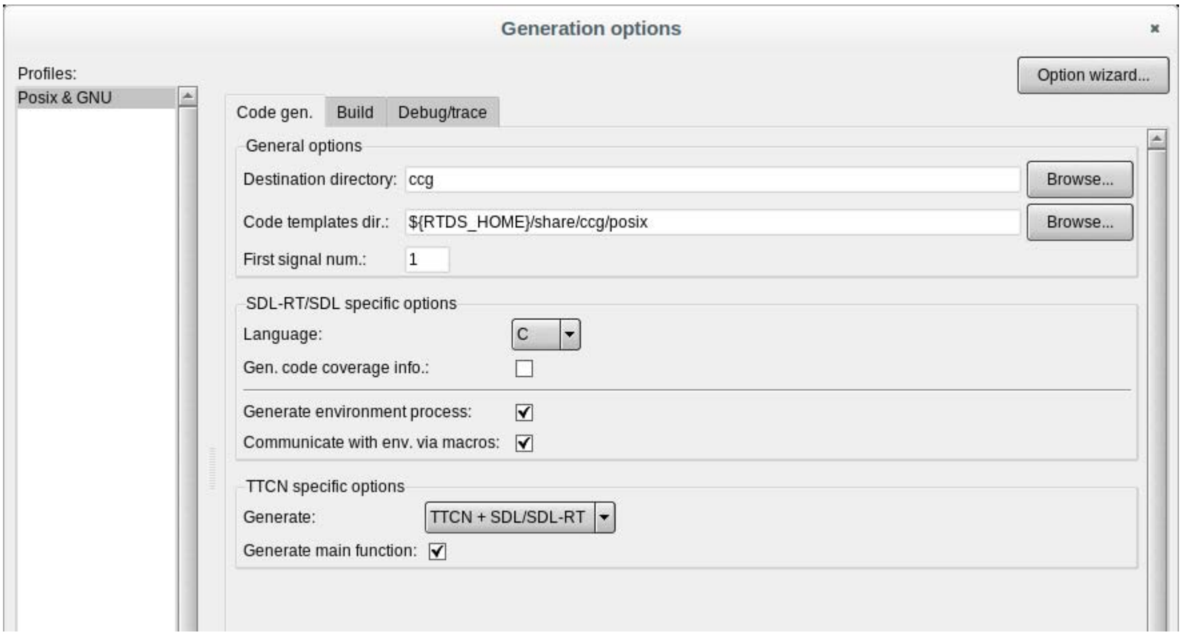
4.4.2.6 Posix profile

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 `PRIO` 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:



Generation options

Profiles: Posix & GNU

Code gen. | Build | Debug/trace

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.: □

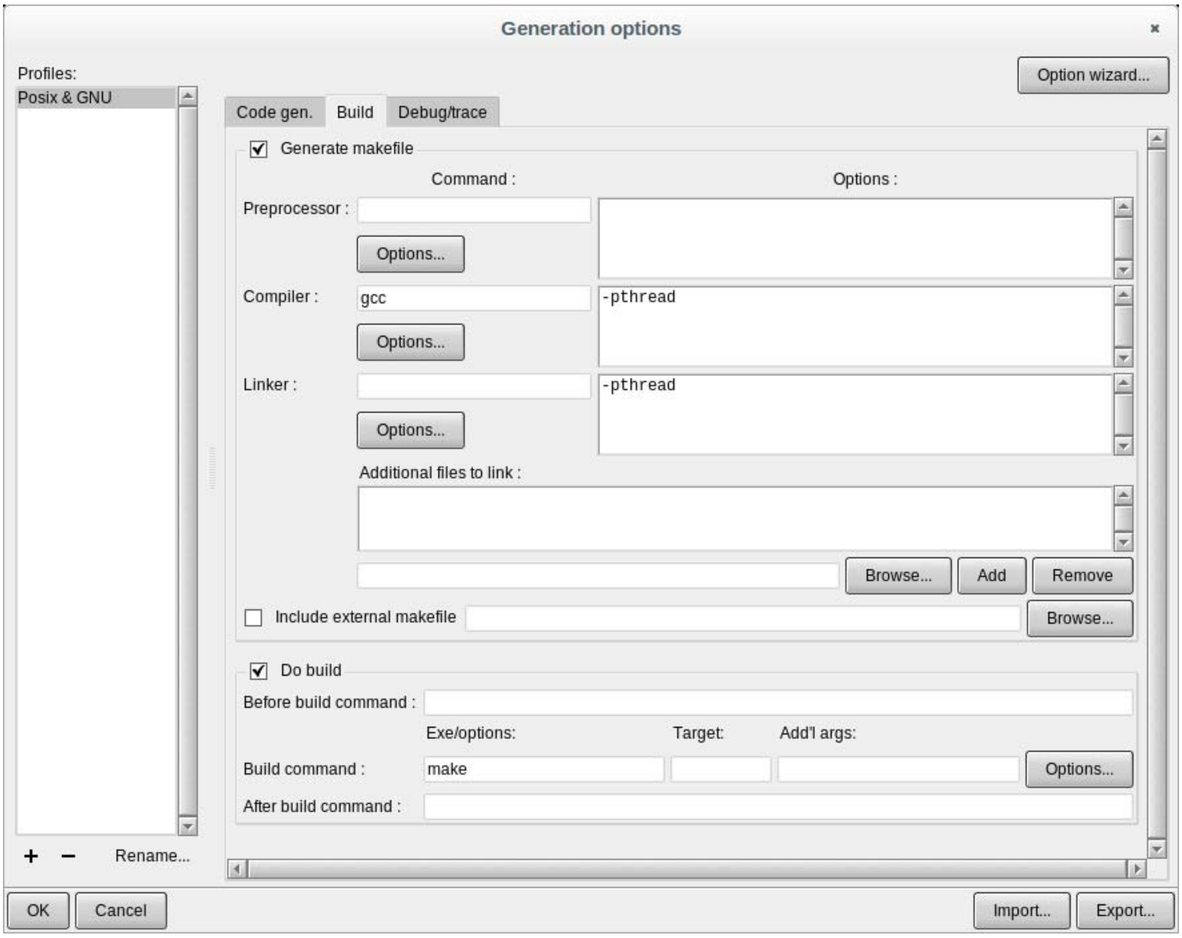
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-RT

Generate main function: ☑



Generation options

Profiles: Posix & GNU

Code gen. | Build | Debug/trace

☑ Generate makefile

	Command :	Options :
Preprocessor :		
Compiler :	gcc	-pthread
Linker :		-pthread

Additional files to link :

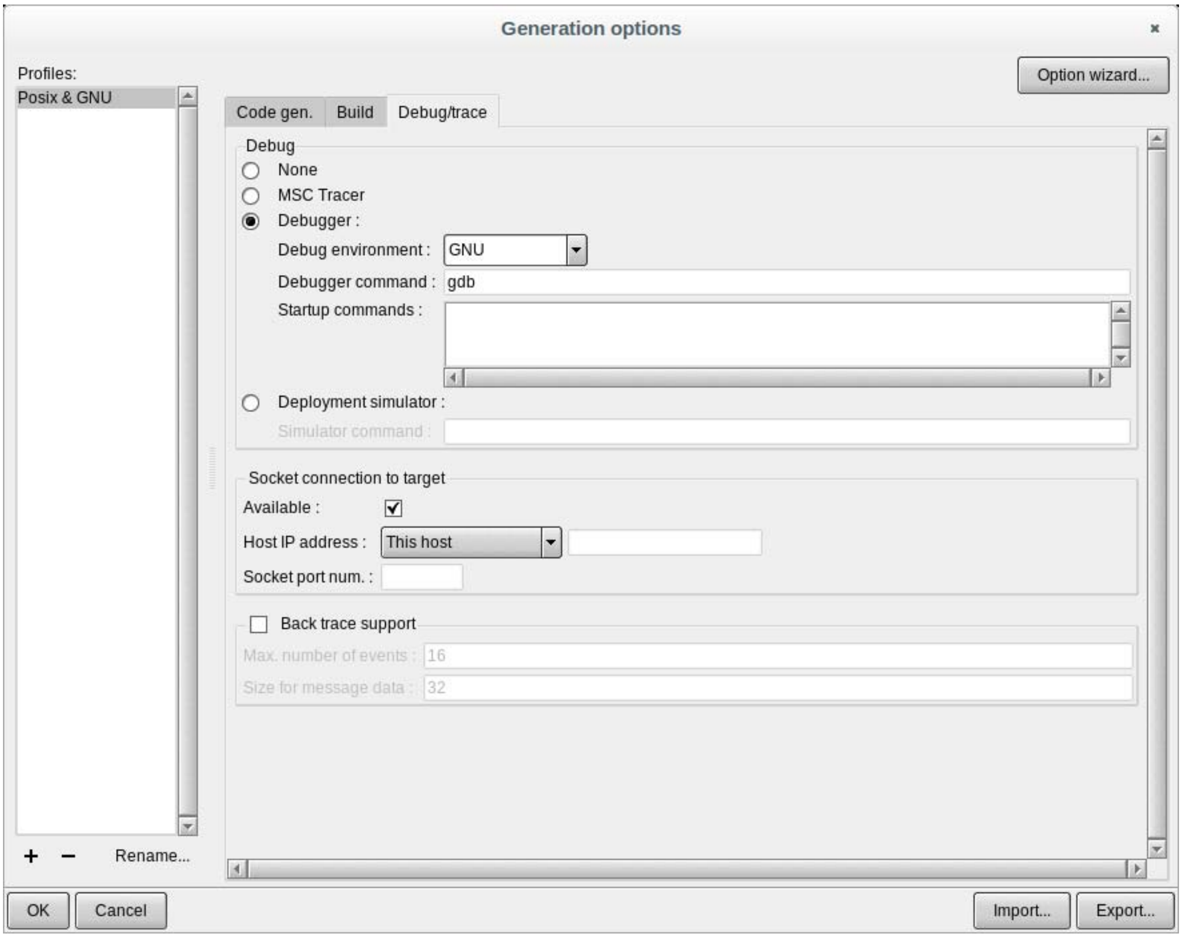
□ Include external makefile

☑ Do build

Before build command :

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

After build command :



Profiles: Posix & GNU

Debug

○ None

○ MSC Tracer

● Debugger :

Debug environment : GNU

Debugger command : gdb

Startup commands :

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : ☑

Host IP address : This host

Socket port num. :

☐ Back trace support

Max. number of events : 16

Size for message data : 32

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

On Linux, the compiler option `-pthread` is required. The linker options depend on the target platform:

- Solaris
  `-lpthread -lrt`
- Linux
  `-pthread`

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

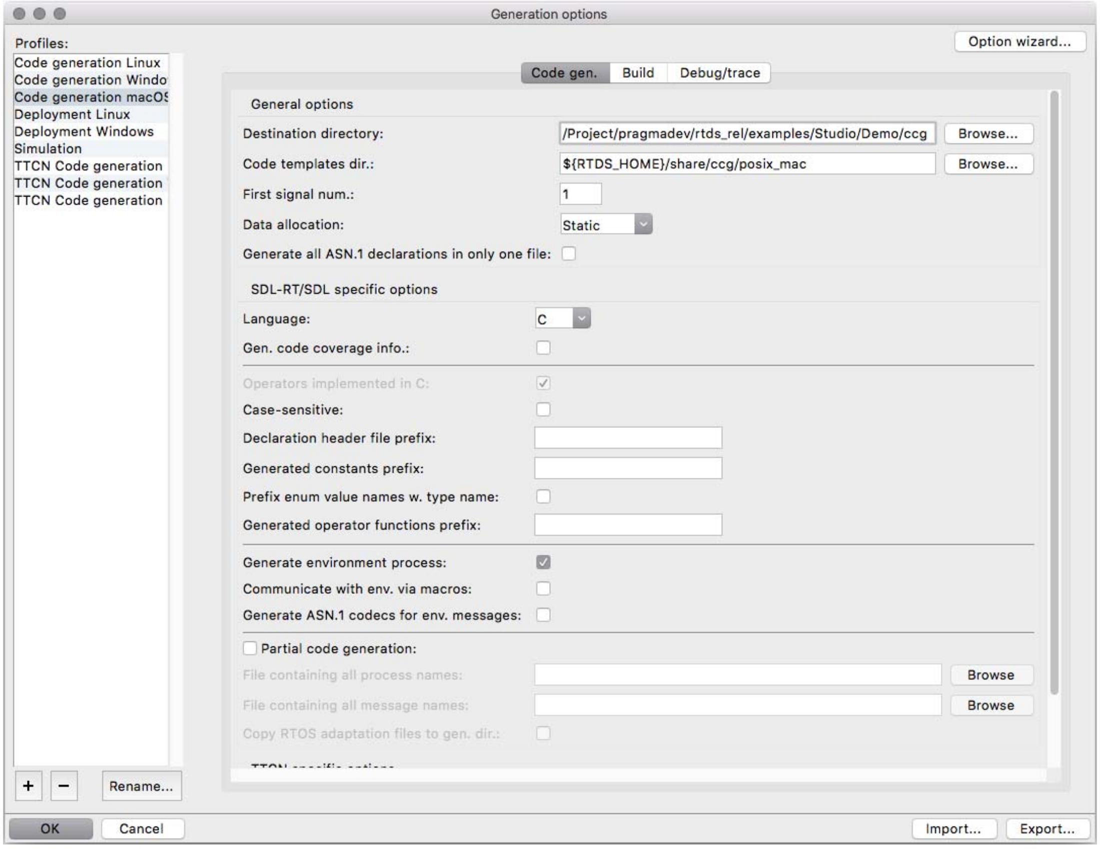
`-lpthread -lrt -lsocket -lnsl`

4.4.2.7 Posix profile for macOS

This profile is a specific version of the Posix profile (see page 266) for macOS. It mostly has the same characteristics, except for the following:

- The default compiler is `clang` and not `gcc` (which is an alias to `clang` on recent versions of macOS);
- The default debugger is `lldb`, and not `gdb` which is not available by default on macOS anymore.

Here is an example of generation options for Posix on macOS:



Generation options

Profiles:
Code generation Linux
Code generation Windo
Code generation macOS
Deployment Linux
Deployment Windows
Simulation
TTCN Code generation
TTCN Code generation
TTCN Code generation

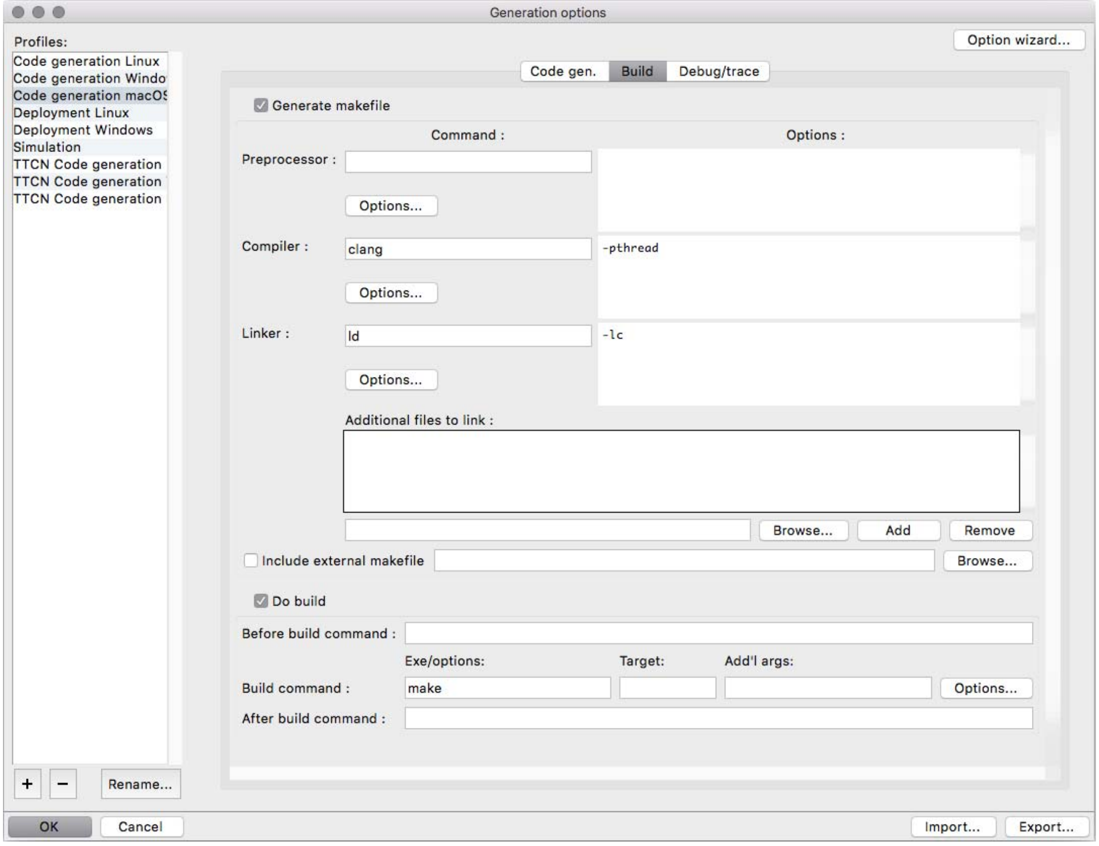
Code gen. | Build | Debug/trace

General options

Destination directory: /Project/pragmadev/rtds_rel/examples/Studio/Demo/ccg
Code templates dir.: ${RTDS_HOME}/share/ccg/posix_mac
First signal num.: 1
Data allocation: Static
Generate all ASN.1 declarations in only one file: □

SDL-RT/SDL specific options

Language: C
Gen. code coverage info.: □
Operators implemented in C: ☑
Case-sensitive: □
Declaration header file prefix:
Generated constants prefix:
Prefix enum value names w. type name: □
Generated operator functions prefix:
Generate environment process: ☑
Communicate with env. via macros: □
Generate ASN.1 codecs for env. messages: □
□ Partial code generation:
File containing all process names:
File containing all message names:
Copy RTOS adaptation files to gen. dir.: □



☑ Generate makefile

	Command :	Options :
Preprocessor :		
Compiler :	clang	-pthread
Linker :	ld	-lc

Additional files to link :

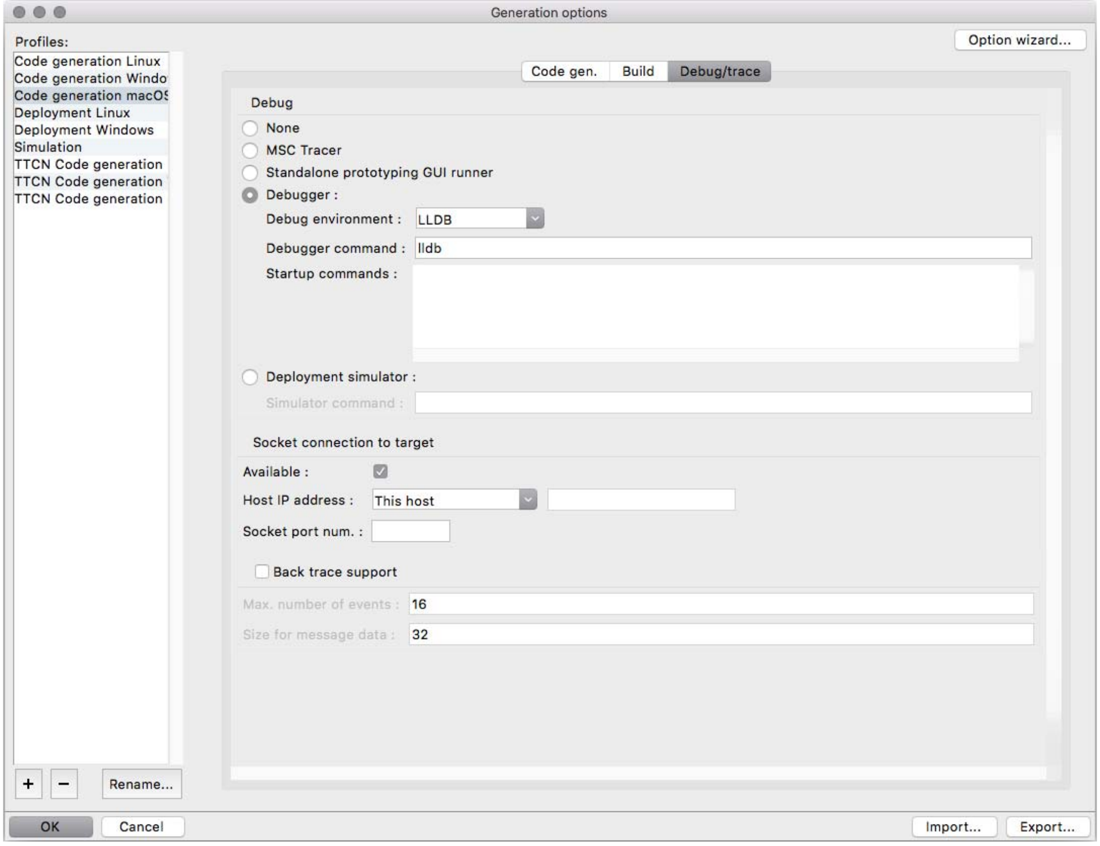
□ Include external makefile

☑ Do build

Before build command :

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

After build command :



Debug

○ None

○ MSC Tracer

○ Standalone prototyping GUI runner

● Debugger :

Debug environment : LLDB

Debugger command : lldb

Startup commands :

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : ☑

Host IP address : This host

Socket port num. :

□ Back trace support

Max. number of events : 16

Size for message data : 32

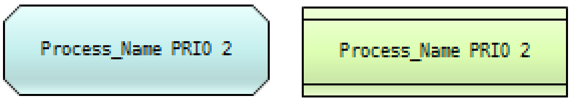
The code template directory is: `${RTDS_HOME}/share/ccg/posix_mac`

4.4.2.8 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`

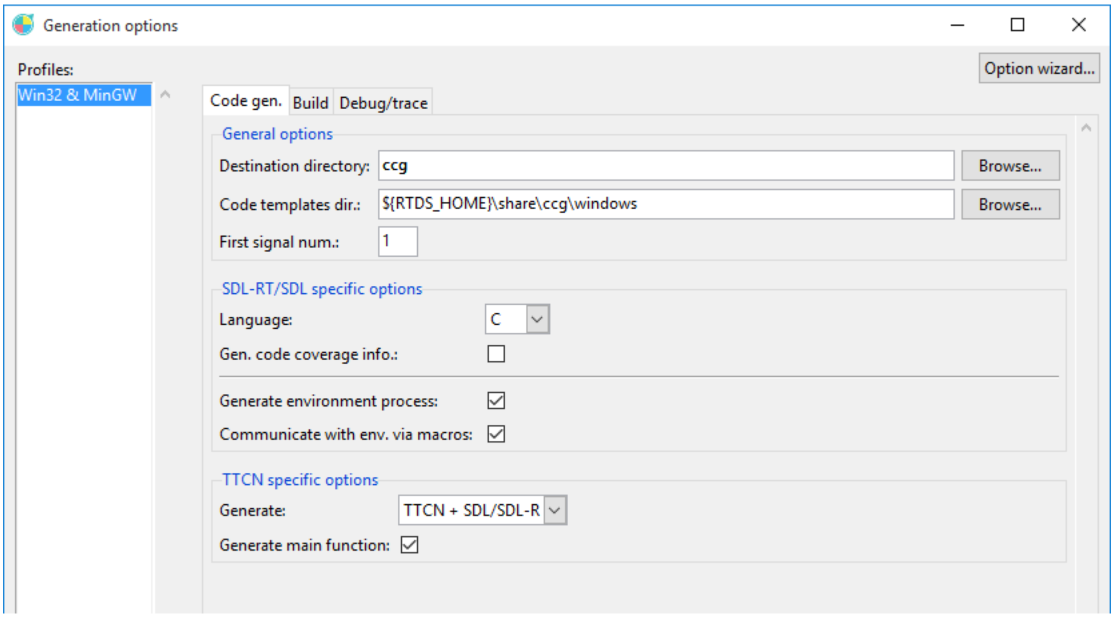
  Default SDL-RT priority is 0 (zero)

  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.: □

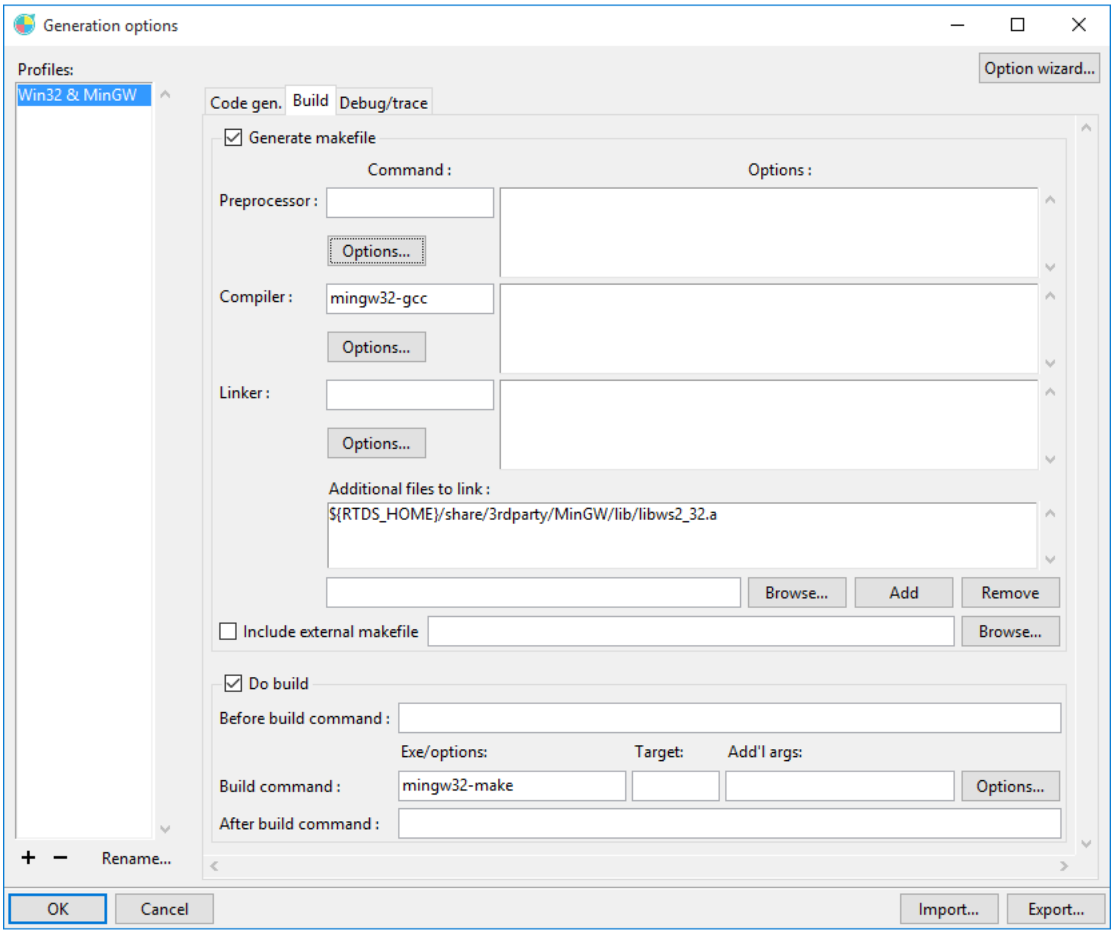
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑



Generation options

Profiles: Win32 & MinGW

Code gen. | Build | Debug/trace

☑ Generate makefile

	Command :	Options :
Preprocessor :		
Compiler :	mingw32-gcc	
Linker :		

Additional files to link :

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

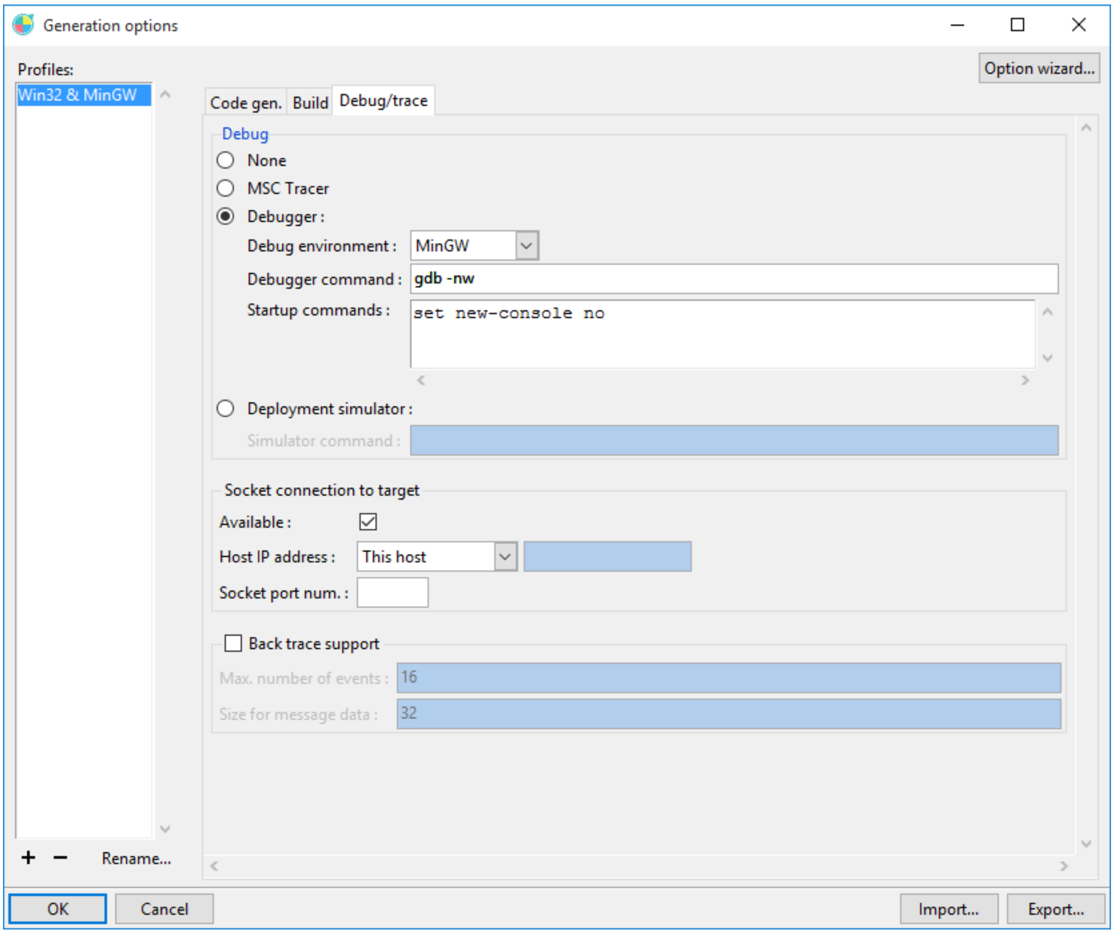
□ Include external makefile

☑ Do build

Before build command :

	Exe/options:	Target:	Add'l args:
Build command :	mingw32-make		

After build command :



Generation options

Profiles: Win32 & MinGW

Code gen. | Build | Debug/trace

Debug

○ None

○ MSC Tracer

● Debugger :

Debug environment : MinGW

Debugger command : gdb -nw

Startup commands :

```
set new-console no
```

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : ☑

Host IP address : This host

Socket port num. :

□ Back trace support

Max. number of events : 16

Size for message data : 32

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`.

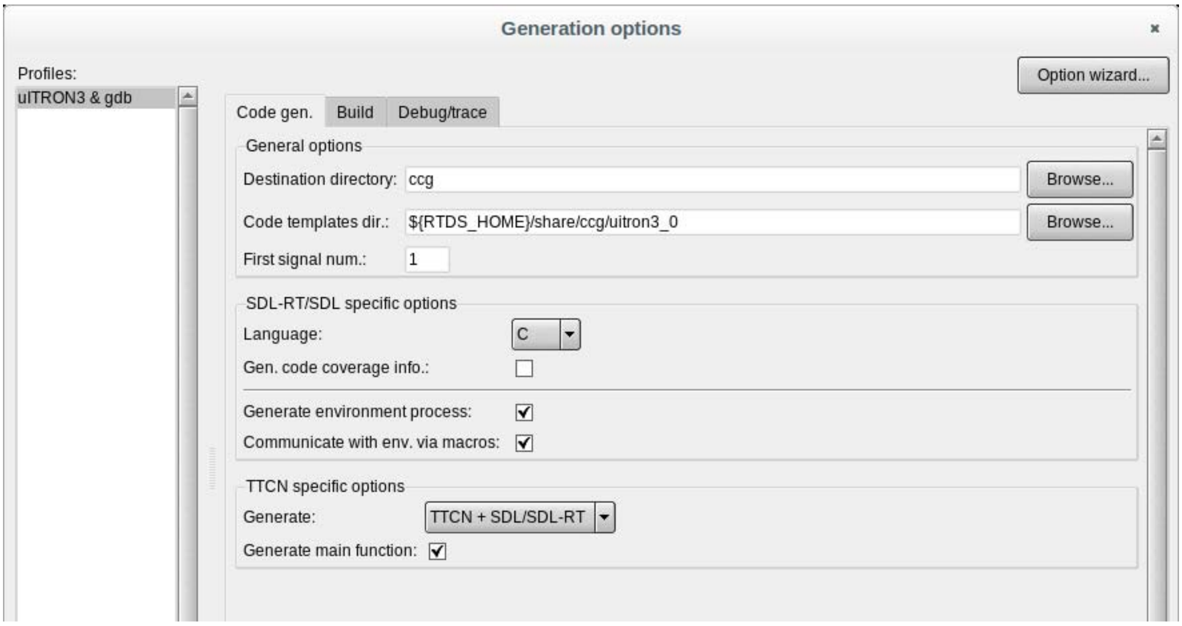
4.4.2.9 uITRON 3.0 profile

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:



Generation options

Profiles: uITRON3 & gdb

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.: □

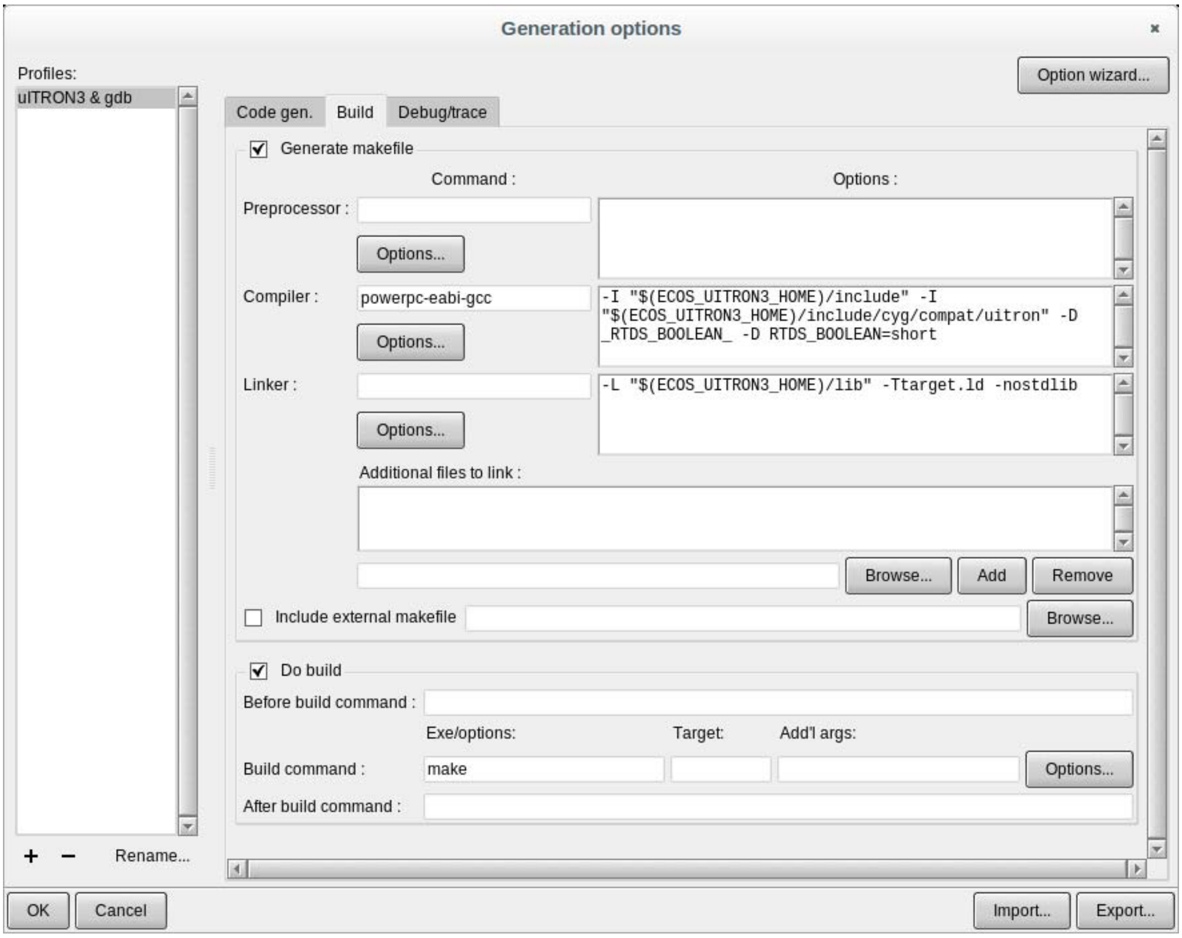
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-RT

Generate main function: ☑



Generation options

Profiles: uITRON3 & gdb

☑ Generate makefile

	Command :	Options :
Preprocessor :		
Compiler :	powerpc-eabi-gcc	-I "$(ECOS_UITRON3_HOME)/include" -I "$(ECOS_UITRON3_HOME)/include/cyg/compat/uitron" -D _RTDS_BOOLEAN_ -D RTDS_BOOLEAN=short
Linker :		-L "$(ECOS_UITRON3_HOME)/lib" -Ttarget.ld -nostdlib

Additional files to link :

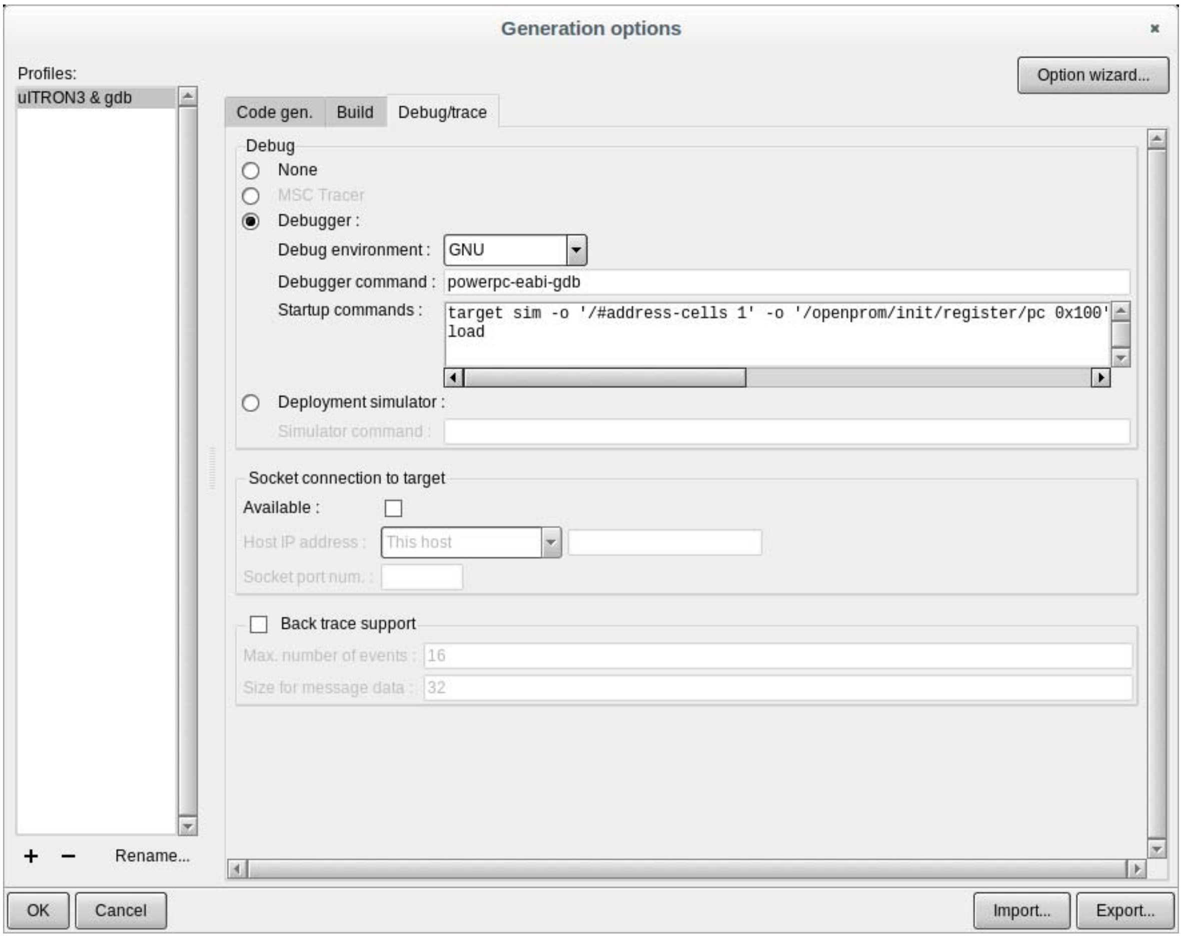
□ Include external makefile

☑ Do build

Before build command :

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

After build command :



Profiles: uITRON3 & gdb

Debug

○ None

○ MSC Tracer

● Debugger :

Debug environment : GNU

Debugger command : powerpc-eabi-gdb

Startup commands :

```
target sim -o '/#address-cells 1' -o '/openprom/init/register/pc 0x100'
load
```

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : □

Host IP address : This host

Socket port num. :

□ Back trace support

Max. number of events : 16

Size for message data : 32

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
```

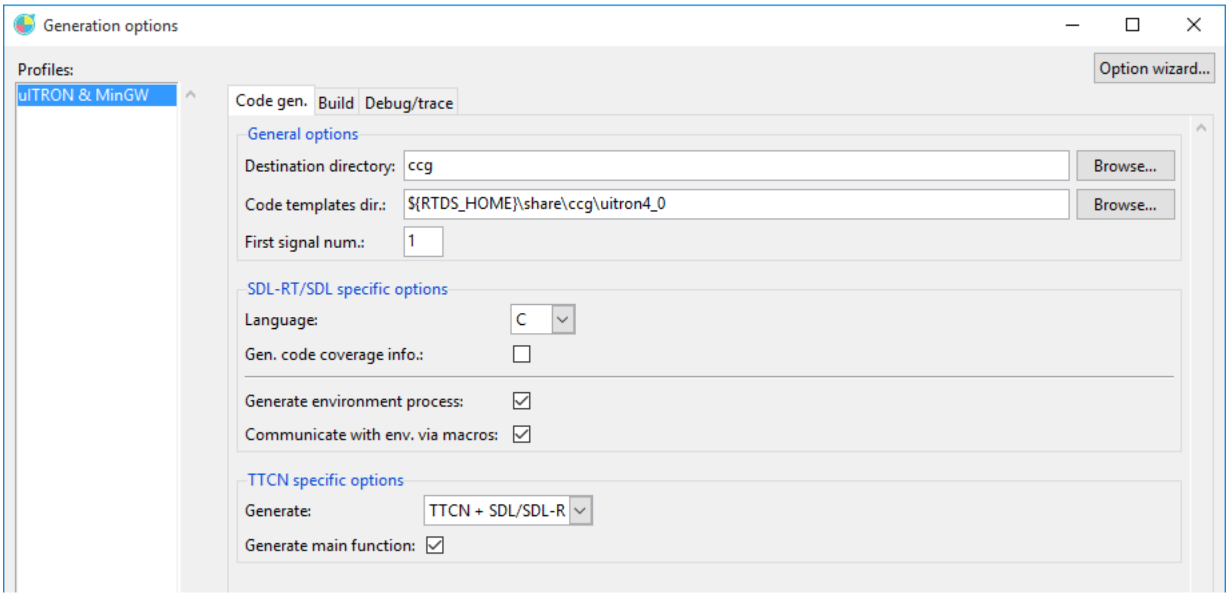
4.4.2.10 uITRON 4.0 profile

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 uITRON as implemented in NUCLEUS:



Profiles: uITRON & MinGW

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.: □

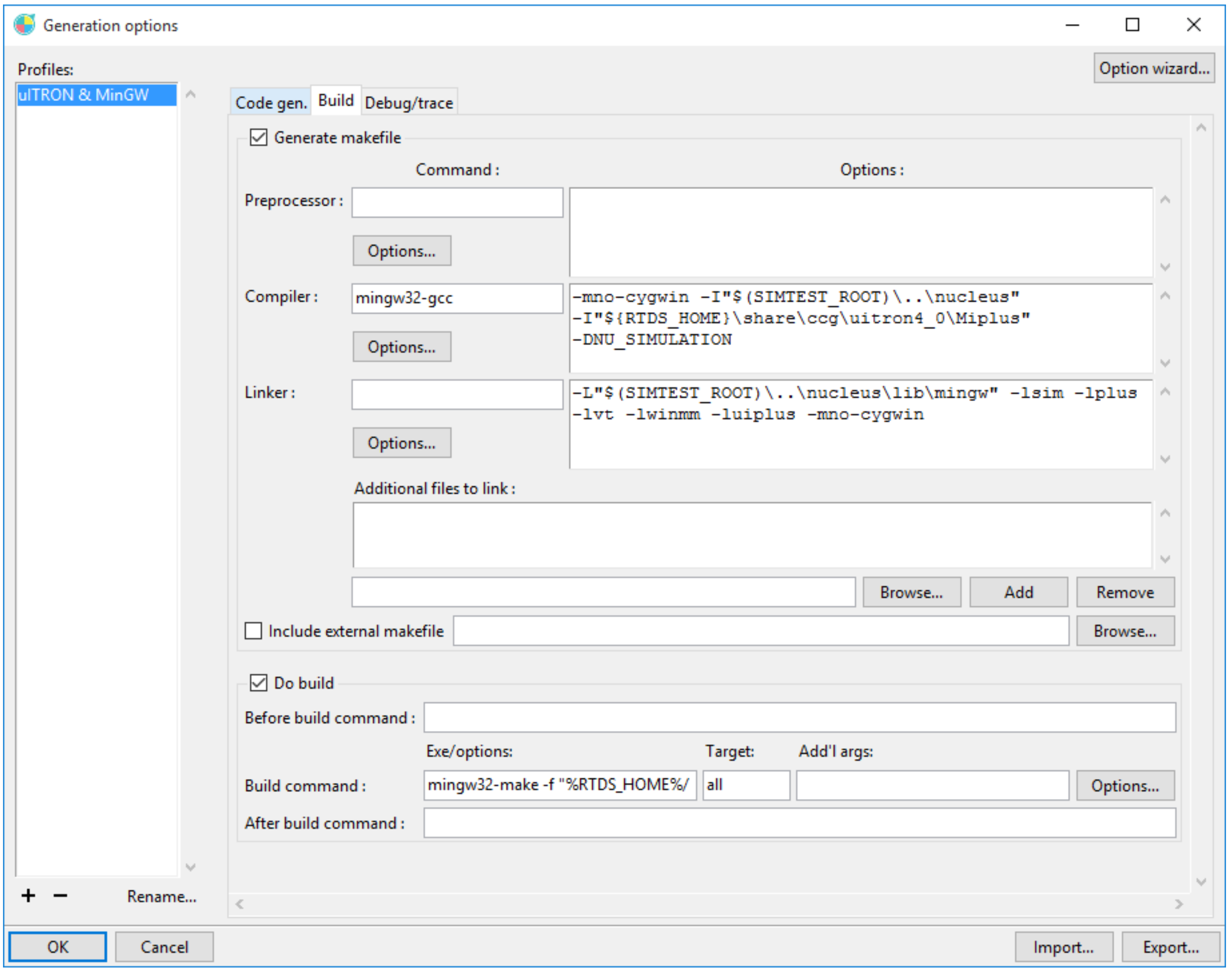
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑



Profiles: uITRON & MinGW

☑ Generate makefile

Preprocessor:

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 -luiplus -mno-cygwin
```

Additional files to link :

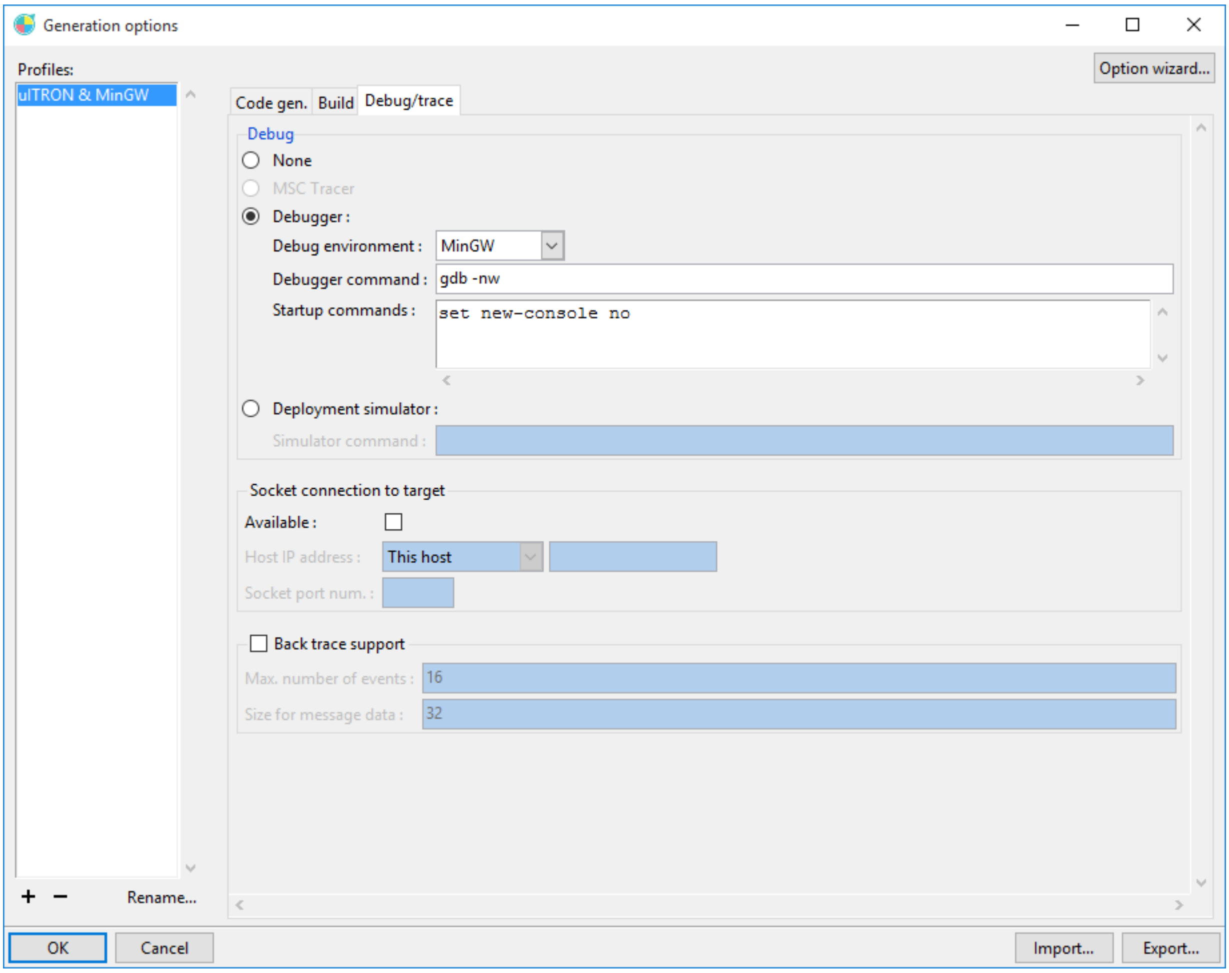
□ Include external makefile

☑ Do build

Before build command :

	Exe/options:	Target:	Add'l args:
Build command :	mingw32-make -f "%RTDS_HOME%/	all	

After build command :



Generation options

Profiles: uITRON & MinGW

Debug

○ None
○ MSC Tracer
● Debugger :
Debug environment : MinGW
Debugger command : gdb -nw
Startup commands :

```
set new-console no
```

○ Deployment simulator :
Simulator command :

Socket connection to target
Available : □
Host IP address : This host
Socket port num. :

□ Back trace support
Max. number of events : 16
Size for message data : 32

The code template directory is:

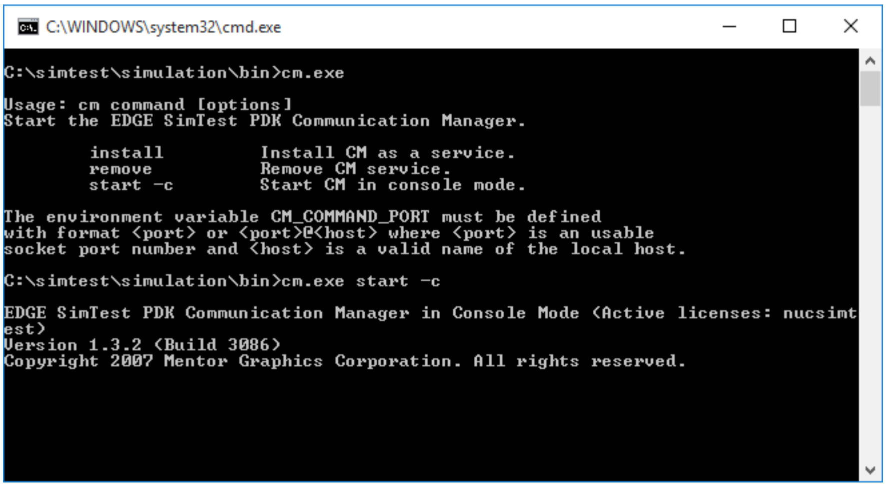
```
${RTDS_HOME}/share/ccg/uitron4_0
```

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
```

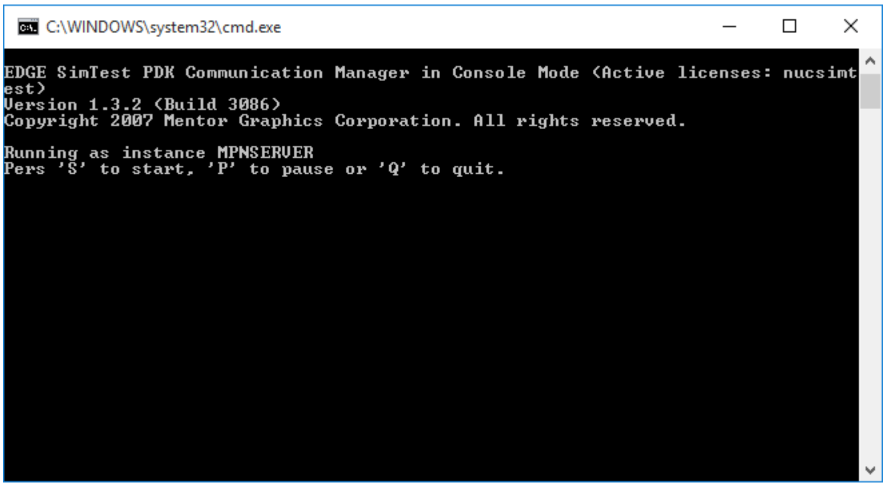
In order to debug a uITRON4 application with SimTest kernel simulator, some manual operations are required:

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



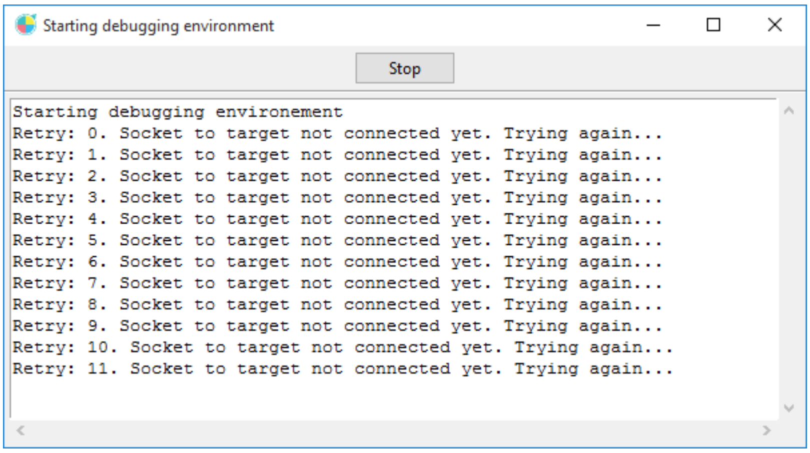
```
C:\simtest\simulation\bin>cm.exe

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: nucsimt
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`



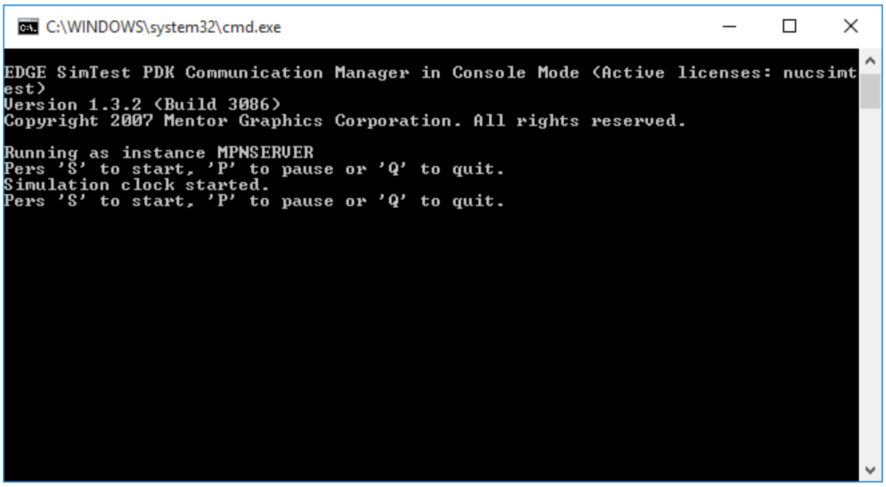
```
EDGE SimTest PDK Communication Manager in Console Mode (Active licenses: nucsimt
est)
Version 1.3.2 (Build 3086)
Copyright 2007 Mentor Graphics Corporation. All rights reserved.

Running as instance MPNSERVER
Pers 'S' to start, 'P' to pause or 'Q' to quit.
```

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



```
Starting debugging environement
Retry: 0. Socket to target not connected yet. Trying again...
Retry: 1. Socket to target not connected yet. Trying again...
Retry: 2. Socket to target not connected yet. Trying again...
Retry: 3. Socket to target not connected yet. Trying again...
Retry: 4. Socket to target not connected yet. Trying again...
Retry: 5. Socket to target not connected yet. Trying again...
Retry: 6. Socket to target not connected yet. Trying again...
Retry: 7. Socket to target not connected yet. Trying again...
Retry: 8. Socket to target not connected yet. Trying again...
Retry: 9. Socket to target not connected yet. Trying again...
Retry: 10. Socket to target not connected yet. Trying again...
Retry: 11. Socket to target not connected yet. Trying again...
```

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



```
EDGE SimTest PDK Communication Manager in Console Mode (Active licenses: nucsimt
est)
Version 1.3.2 (Build 3086)
Copyright 2007 Mentor Graphics Corporation. All rights reserved.

Running as instance MPNSERVER
Pers 'S' to start, 'P' to pause or 'Q' to quit.
Simulation clock started.
Pers 'S' to start, 'P' to pause or 'Q' to quit.
```

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

4.4.2.11 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:

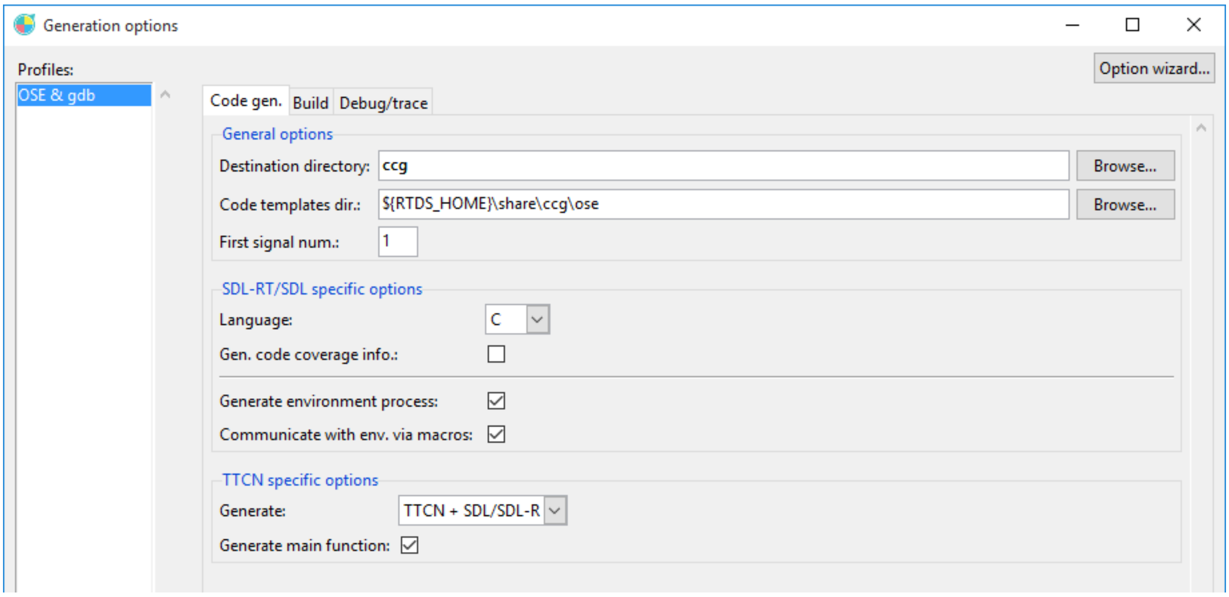
- Build process
  The OSE build process is based on dmake utility and makefile.mk and userconf.mk makefiles. When using OSE code generation, PragmaDev Studio generates pragmadev.mk makefile in the code generation directory. The makefile.mk and userconf.mk should be put somewhere else because they are 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) /* !-SIGNO(struct RTDS_MessageHeader)-! */
#define tWait (2) /* !-SIGNO(struct RTDS_MessageHeader)-! */
#define pong  (3) /* !-SIGNO(struct RTDS_MessageHeader)-! */
#define begin (4) /* !-SIGNO(struct RTDS_MessageHeader)-! */
#define myStart (5) /* !-SIGNO(struct RTDS_MessageHeader)-! */
#define myStop  (6) /* !-SIGNO(struct RTDS_MessageHeader)-! */
```

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



Profiles: OSE & gdb

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.: □

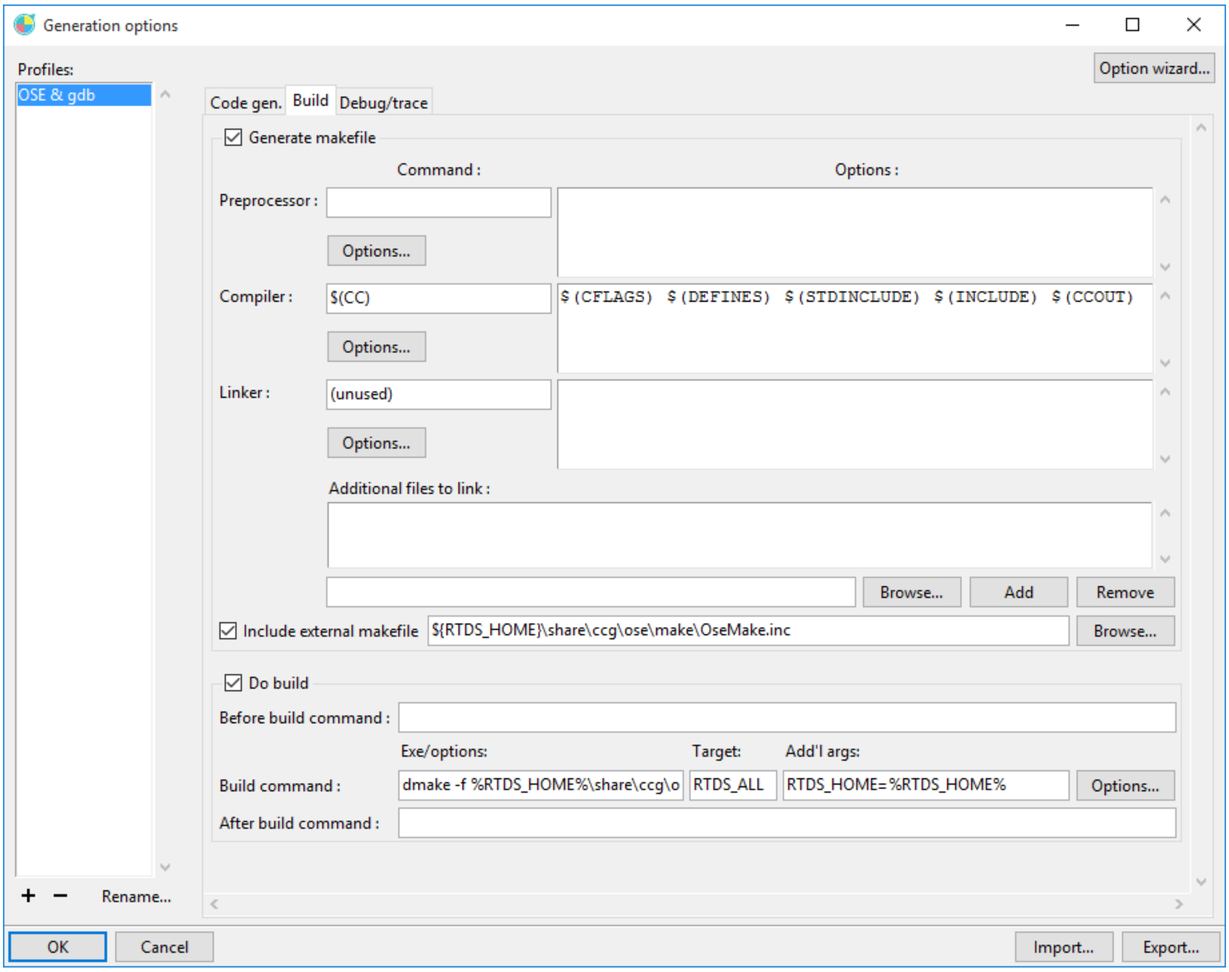
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑



Profiles: OSE & gdb

☑ Generate makefile

Command : / Options :

Preprocessor :

Compiler : $(CC) — $(CFLAGS) $(DEFINES) $(STDINCLUDE) $(INCLUDE) $(CCOUT)

Linker : (unused)

Additional files to link :

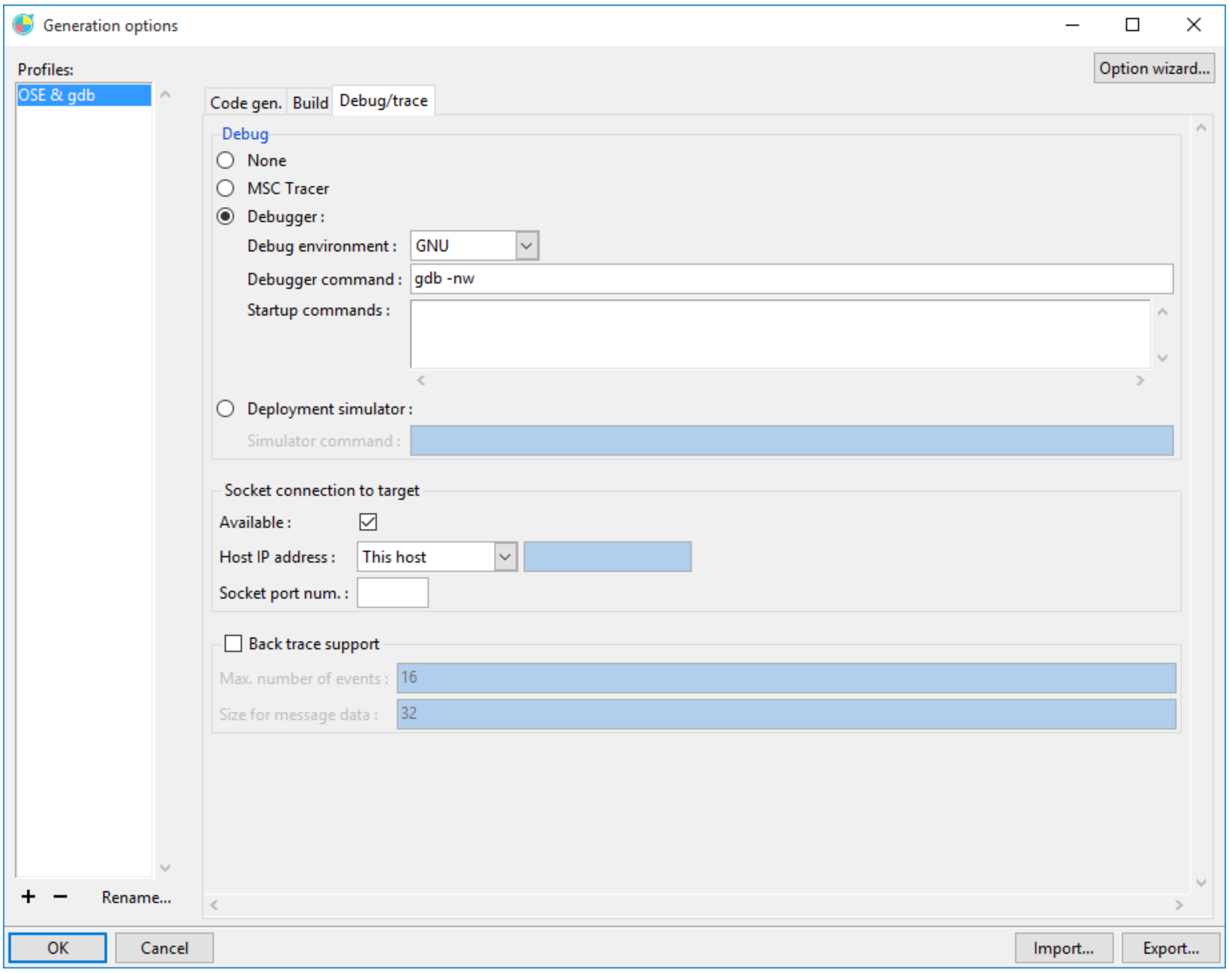
☑ Include external makefile ${RTDS_HOME}\share\ccg\ose\make\OseMake.inc

☑ Do build

Before build command :

Build command : Exe/options: dmake -f %RTDS_HOME%\share\ccg\o Target: RTDS_ALL Add'l args: RTDS_HOME=%RTDS_HOME%

After build command :



Generation options

Profiles:

OSE & gdb

Code gen. Build Debug/trace

Debug

○ None

○ MSC Tracer

● Debugger :

Debug environment : GNU

Debugger command : gdb -nw

Startup commands :

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : ☑

Host IP address : This host

Socket port num. :

□ Back trace support

Max. number of events : 16

Size for message data : 32

Typical generation profile to debug an OSE application with gdb

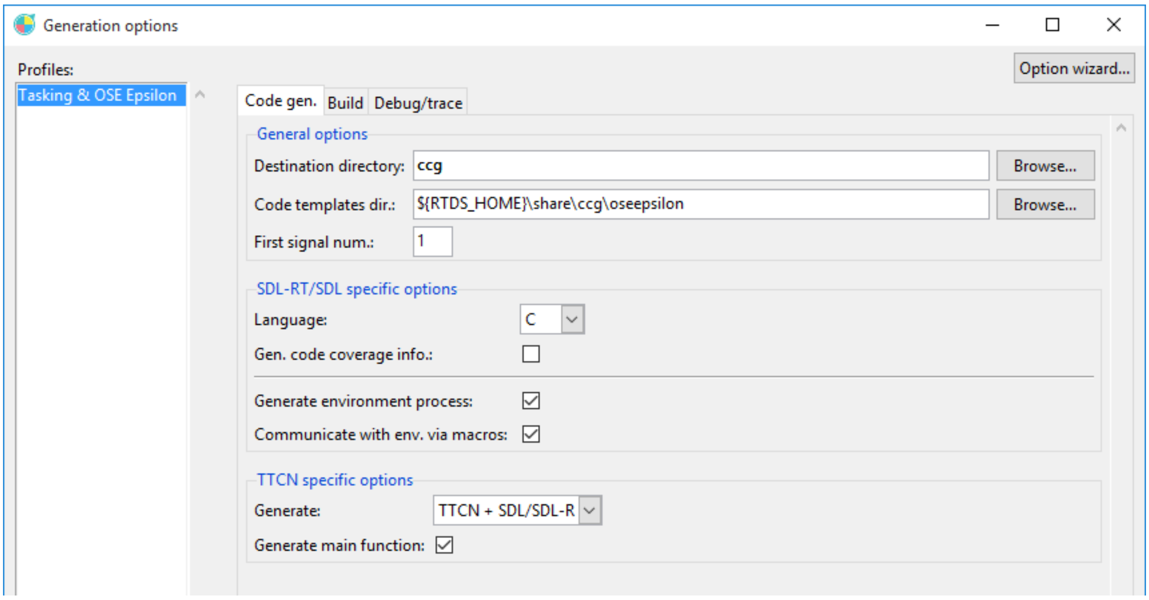
4.4.2.12 OSE Epsilon profile

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

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

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: □

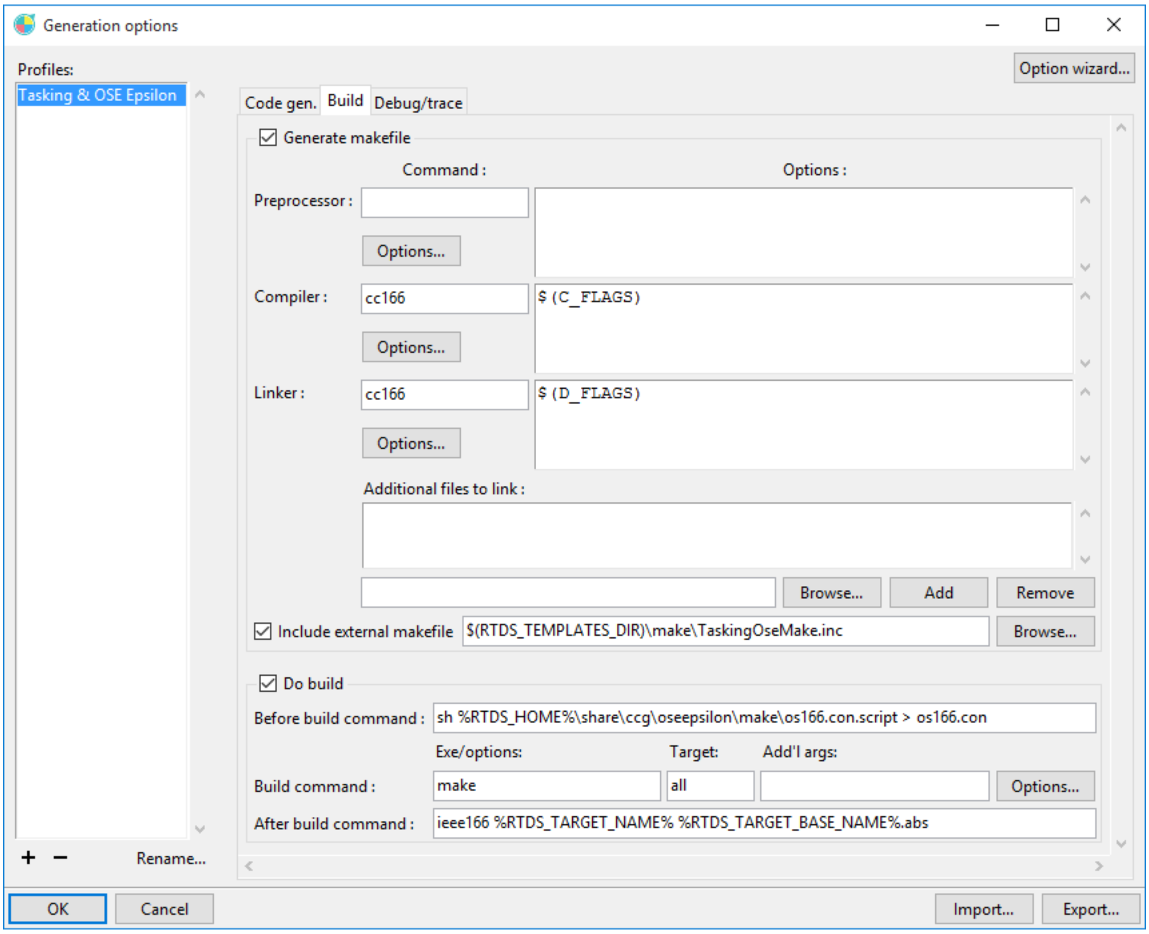
Generate environment process: ☑

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 :		
Compiler :	cc166	$(C_FLAGS)
Linker :	cc166	$(D_FLAGS)

Additional files to link :

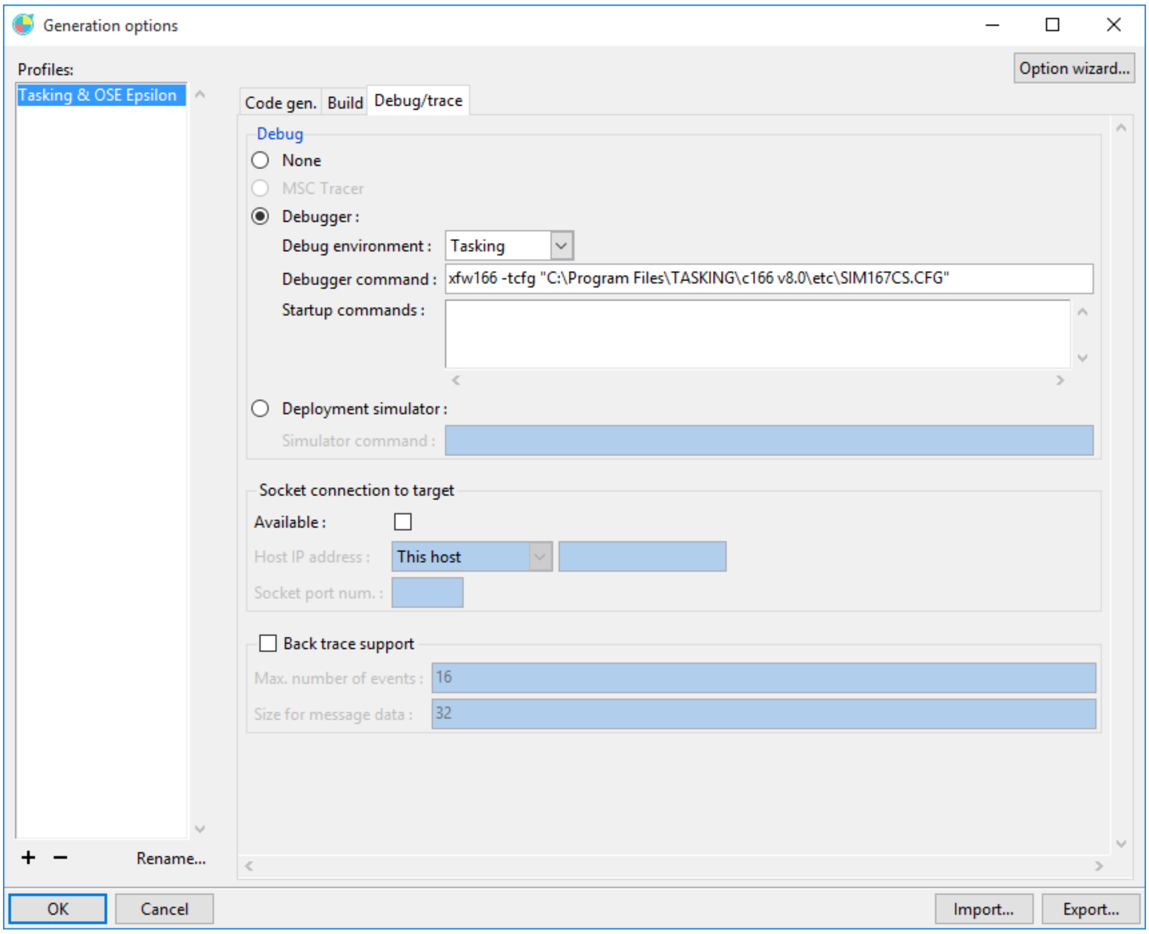
☑ Include external makefile $(RTDS_TEMPLATES_DIR)\make\TaskingOseMake.inc

☑ Do build

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

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

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



Profiles: Tasking & OSE Epsilon

Debug

○ None

○ MSC Tracer

● Debugger :

Debug environment : Tasking

Debugger command : xfw166 -tcfg "C:\Program Files\TASKING\c166 v8.0\etc\SIM167CS.CFG"

Startup commands :

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : □

Host IP address : This host

Socket port num. :

□ Back trace support

Max. number of events : 16

Size for message data : 32

Typical generation profile to debug an OSE Epsilon application with Tasking

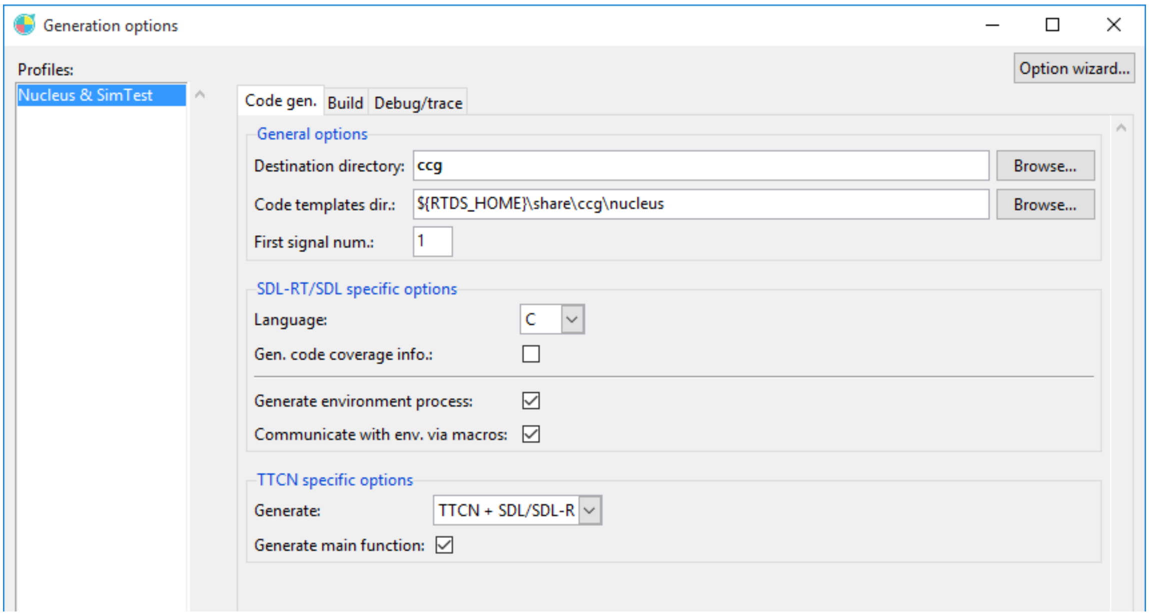
4.4.2.13 Nucleus profile

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:



Profiles: Nucleus & SimTest

General options

Destination directory: ccg

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

First signal num.: 1

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: □

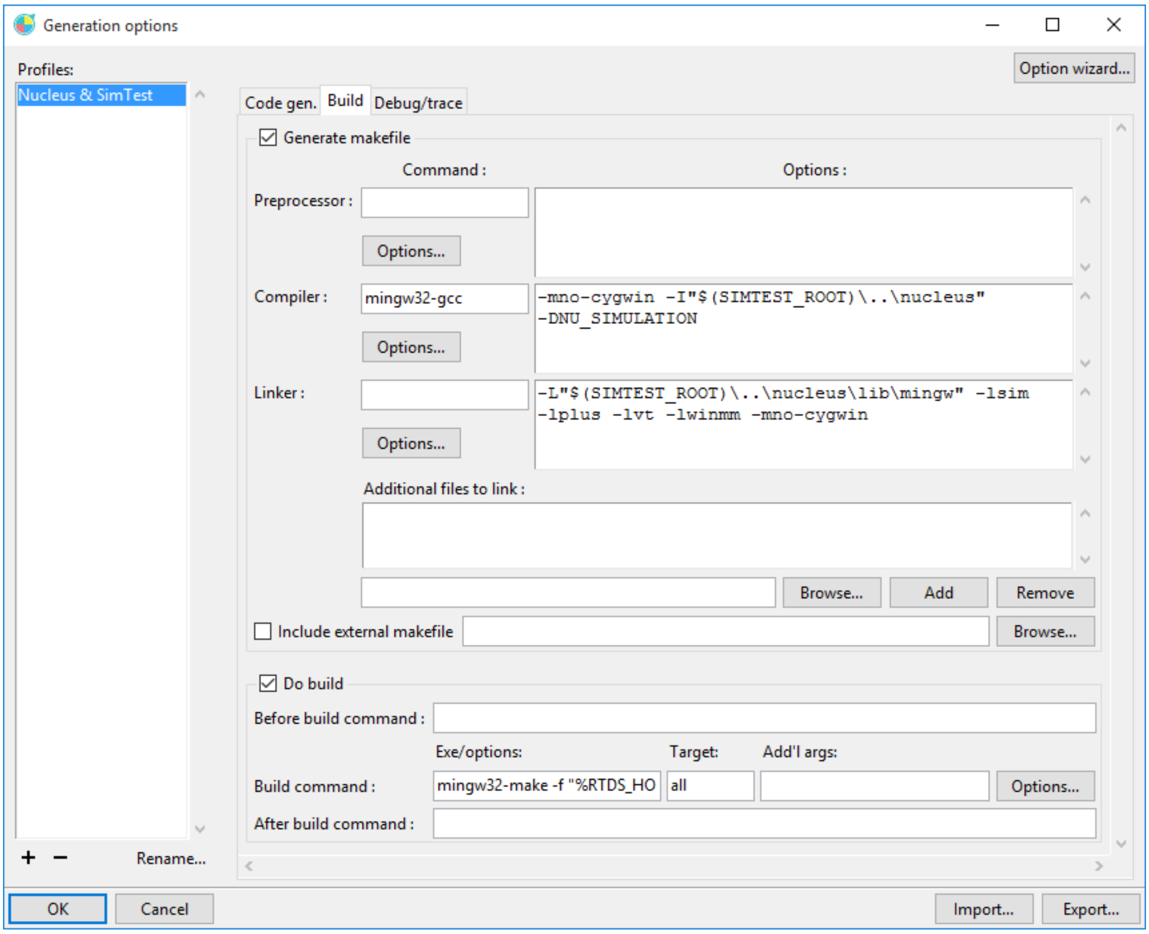
Generate environment process: ☑

Communicate with env. via macros: ☑

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑



Profiles: Nucleus & SimTest

☑ Generate makefile

Command : Options :

Preprocessor :

Compiler : mingw32-gcc

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

Linker :

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

Additional files to link :

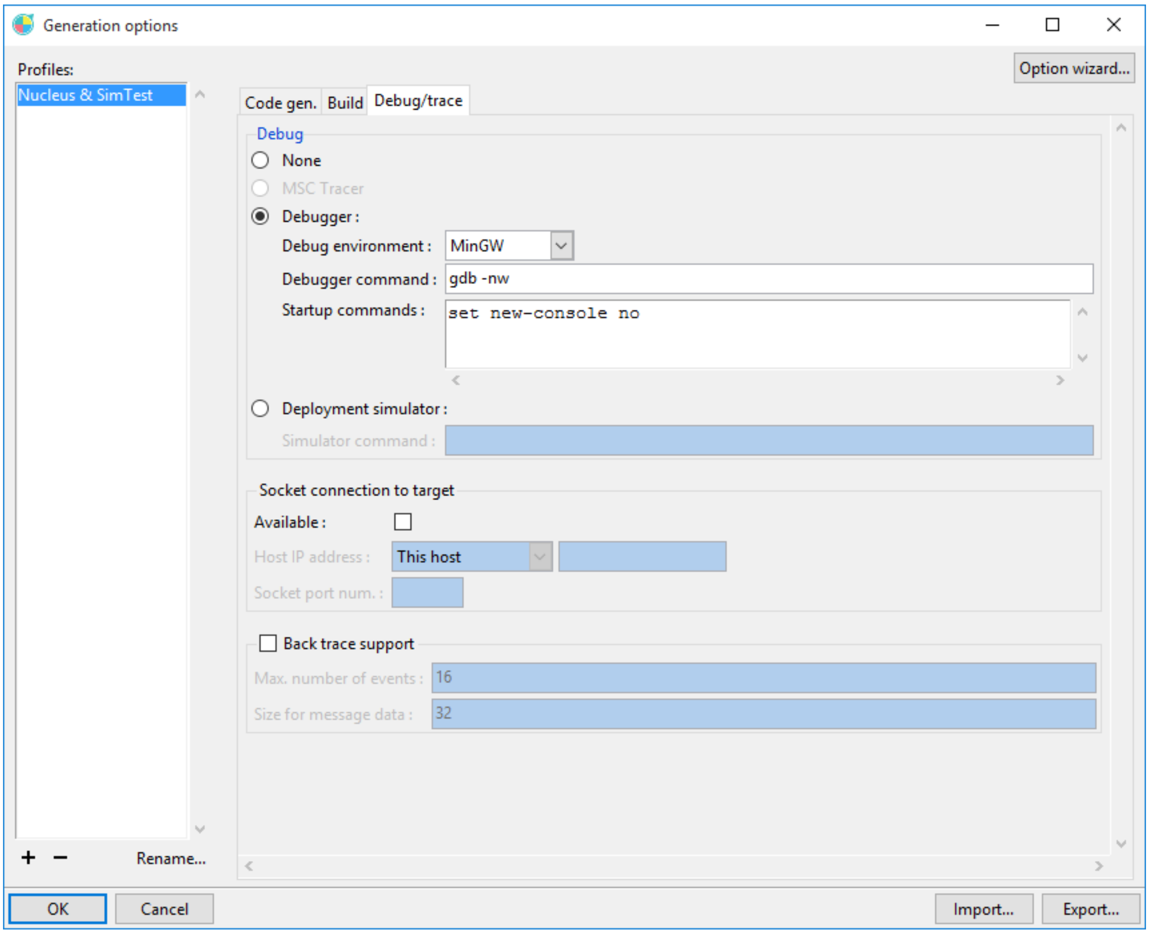
□ Include external makefile

☑ Do build

Before build command :

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

After build command :



Profiles: Nucleus & SimTest

Debug
○ None
○ MSC Tracer
● Debugger :
Debug environment : MinGW
Debugger command : gdb -nw
Startup commands : `set new-console no`
○ Deployment simulator :
Simulator command :

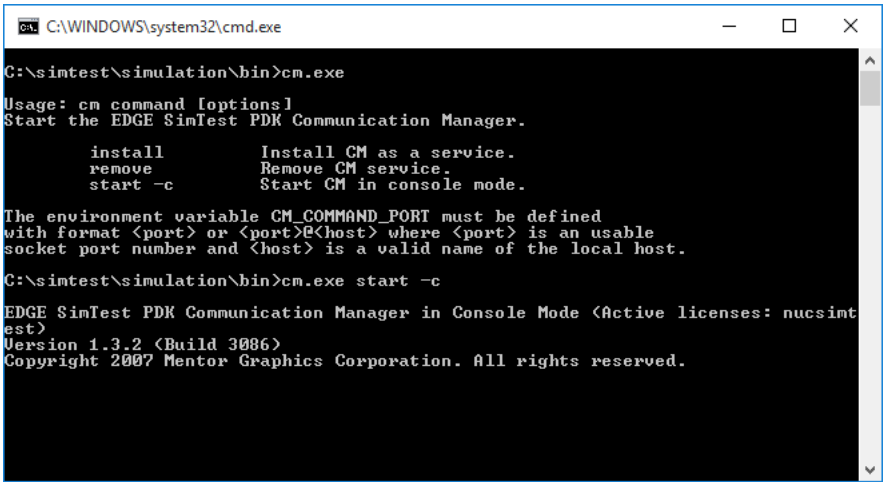
Socket connection to target
Available : □
Host IP address : This host
Socket port num. :

□ Back trace support
Max. number of events : 16
Size for message data : 32

Typical generation profile to debug a Nucleus application with gdb

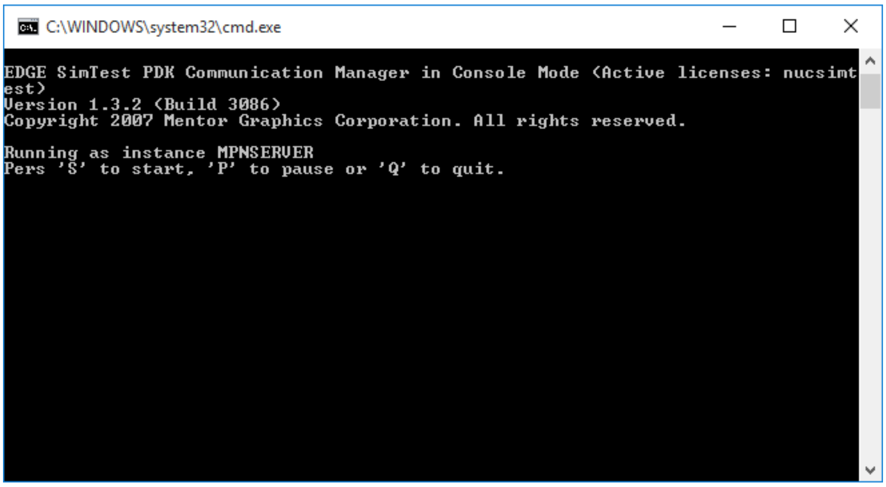
In order to debug a Nucleus application with SimTest kernel simulator, some manual operations are required:

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



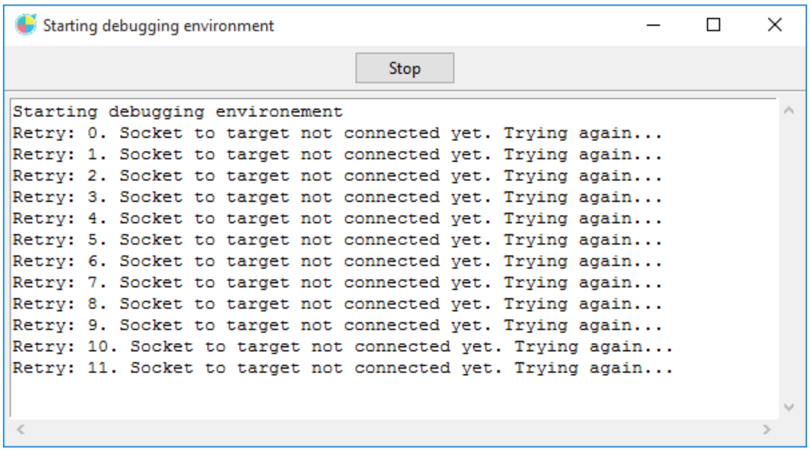
```
C:\simtest\simulation\bin>cm.exe

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: nucsimt
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`



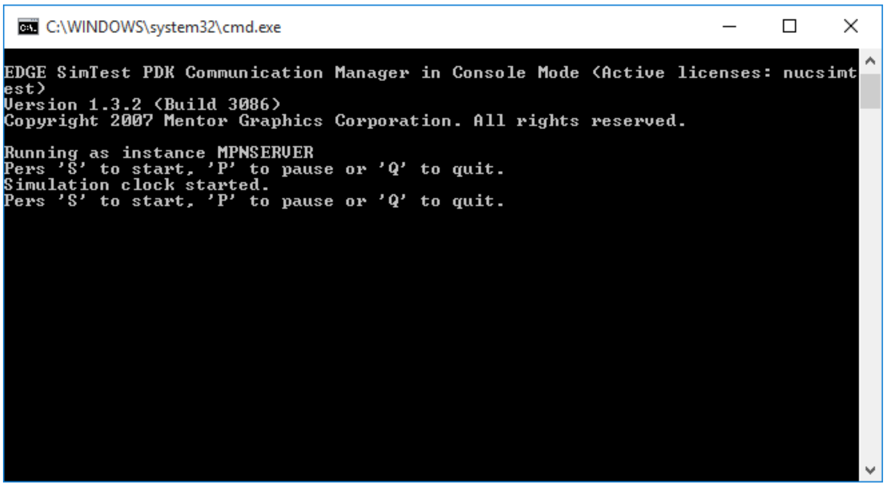
```
EDGE SimTest PDK Communication Manager in Console Mode (Active licenses: nucsimt
est)
Version 1.3.2 (Build 3086)
Copyright 2007 Mentor Graphics Corporation. All rights reserved.

Running as instance MPNSERVER
Pers 'S' to start, 'P' to pause or 'Q' to quit.
```

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



```
Starting debugging environement
Retry: 0. Socket to target not connected yet. Trying again...
Retry: 1. Socket to target not connected yet. Trying again...
Retry: 2. Socket to target not connected yet. Trying again...
Retry: 3. Socket to target not connected yet. Trying again...
Retry: 4. Socket to target not connected yet. Trying again...
Retry: 5. Socket to target not connected yet. Trying again...
Retry: 6. Socket to target not connected yet. Trying again...
Retry: 7. Socket to target not connected yet. Trying again...
Retry: 8. Socket to target not connected yet. Trying again...
Retry: 9. Socket to target not connected yet. Trying again...
Retry: 10. Socket to target not connected yet. Trying again...
Retry: 11. Socket to target not connected yet. Trying again...
```

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



```
EDGE SimTest PDK Communication Manager in Console Mode (Active licenses: nucsimt
est)
Version 1.3.2 (Build 3086)
Copyright 2007 Mentor Graphics Corporation. All rights reserved.

Running as instance MPNSERVER
Pers 'S' to start, 'P' to pause or 'Q' to quit.
Simulation clock started.
Pers 'S' to start, 'P' to pause or 'Q' to quit.
```

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

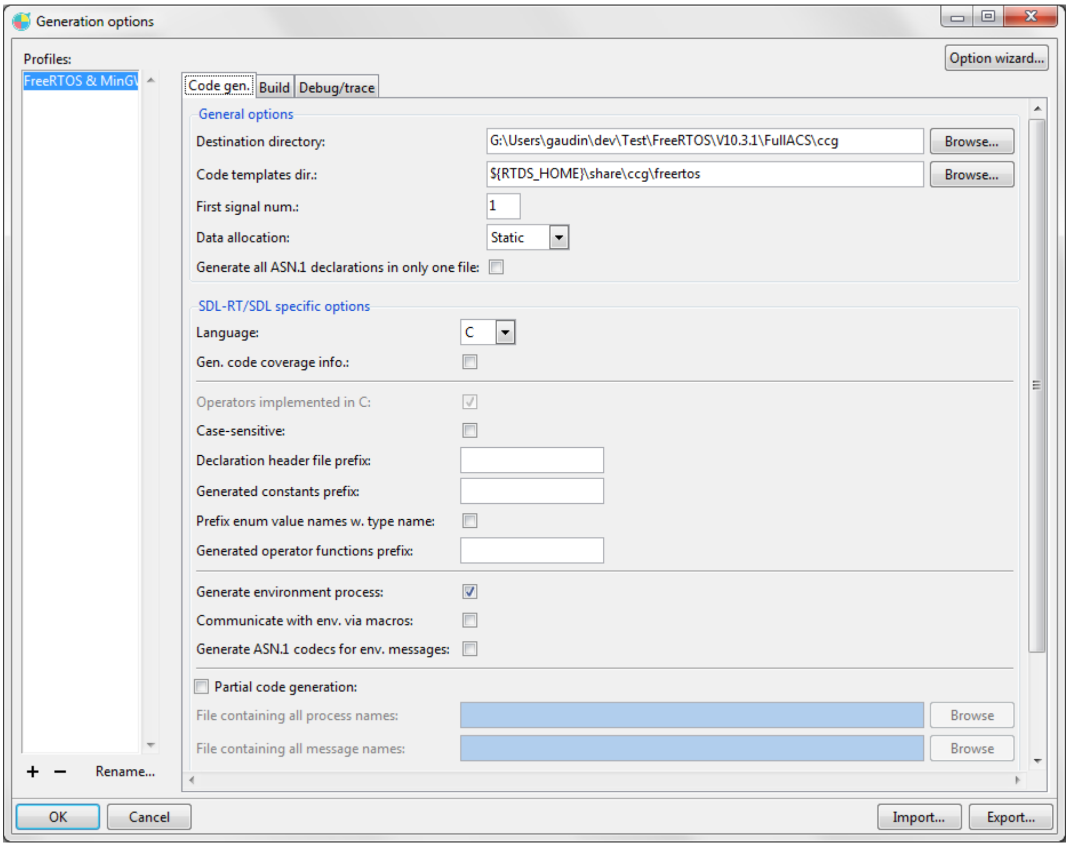
4.4.2.14 FreeRTOS profile

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 V10.3.1 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\FreeRTOSV10.3.1`. 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 FreeRTOS task called RTDS_Socket that is created before any other task. We tried to gather all theses 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.

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



Generation options

Profiles: FreeRTOS & MinGW

General options

Destination directory: G:\Users\gaudin\dev\Test\FreeRTOS\V10.3.1\FullACS\ccg

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

First signal num.: 1

Data allocation: Static

Generate all ASN.1 declarations in only one file: □

SDL-RT/SDL specific options

Language: C

Gen. code coverage info.: □

Operators implemented in C: ☑

Case-sensitive: □

Declaration header file prefix:

Generated constants prefix:

Prefix enum value names w. type name: □

Generated operator functions prefix:

Generate environment process: ☑

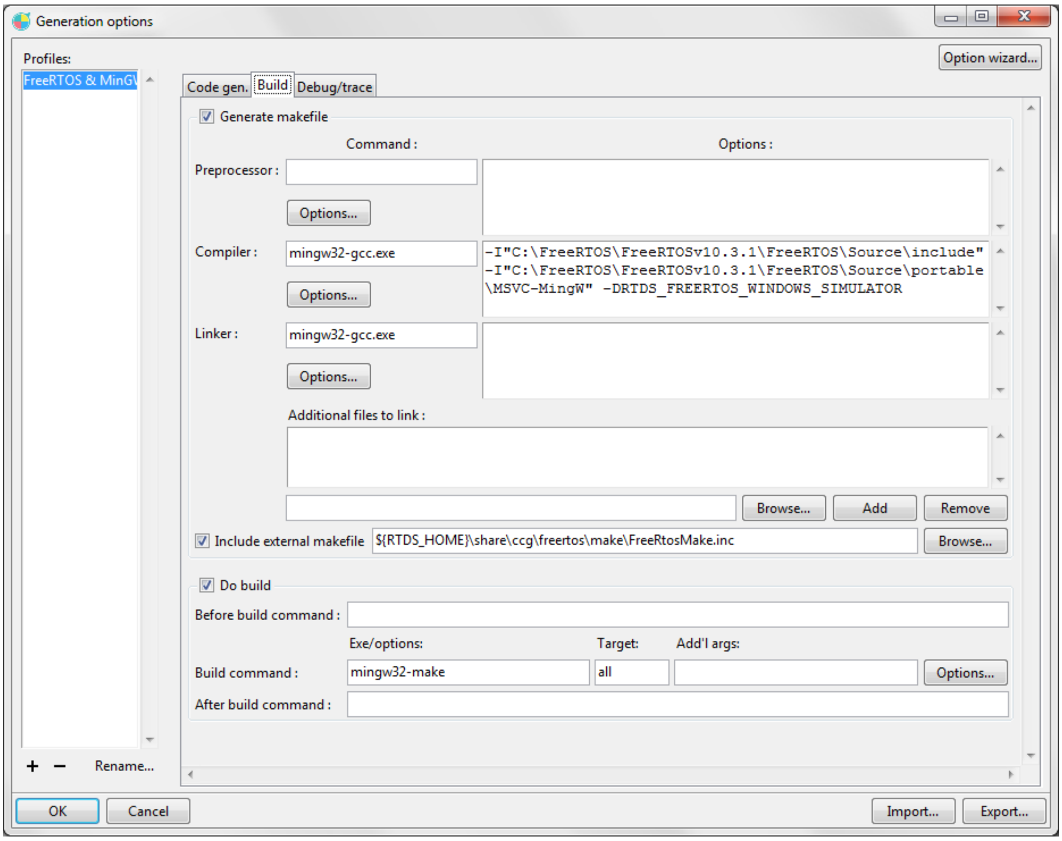
Communicate with env. via macros: □

Generate ASN.1 codecs for env. messages: □

□ Partial code generation:

File containing all process names:

File containing all message names:



Generation options

Profiles: FreeRTOS & MinGW

☑ Generate makefile

	Command :	Options :
Preprocessor :		
Compiler :	mingw32-gcc.exe	-I"C:\FreeRTOS\FreeRTOSv10.3.1\FreeRTOS\Source\include" -I"C:\FreeRTOS\FreeRTOSv10.3.1\FreeRTOS\Source\portable\MSVC-MingW" -DRTDS_FREERTOS_WINDOWS_SIMULATOR
Linker :	mingw32-gcc.exe	

Additional files to link :

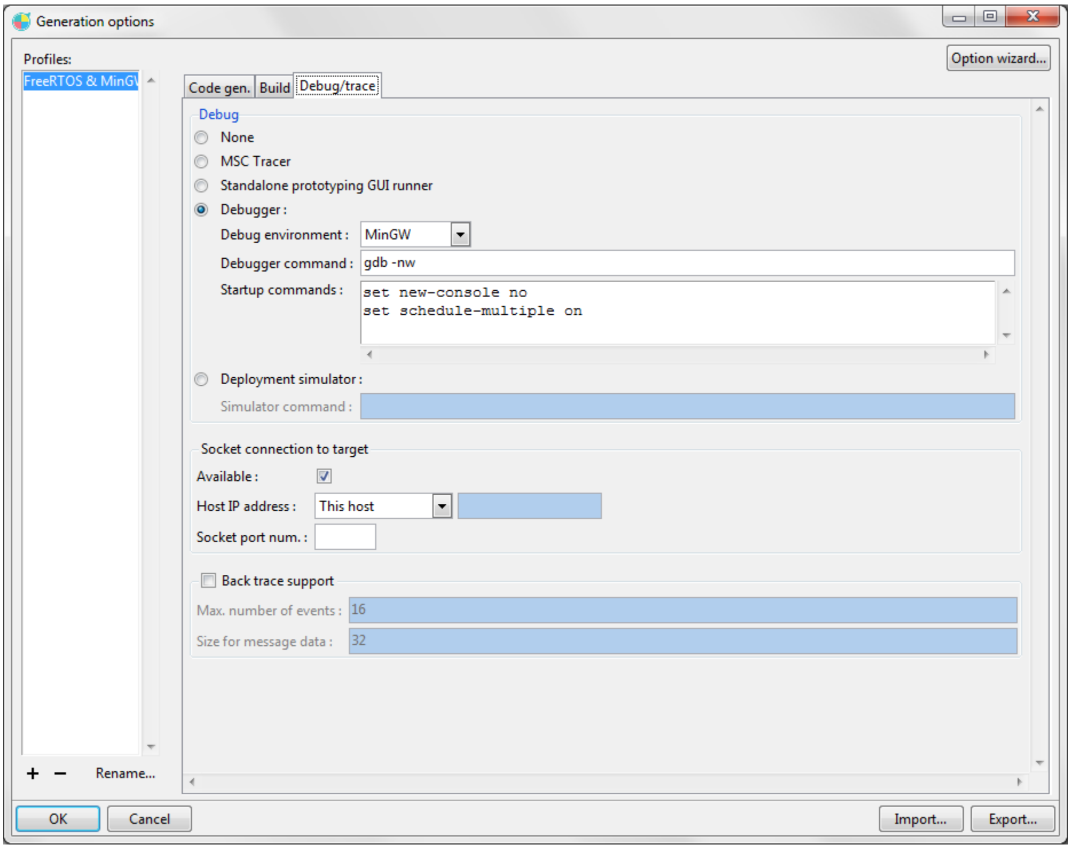
☑ Include external makefile ${RTDS_HOME}\share\ccg\freertos\make\FreeRtosMake.inc

☑ Do build

Before build command :

	Exe/options:	Target:	Add'l args:
Build command :	mingw32-make	all	

After build command :



Debug

○ None

○ MSC Tracer

○ Standalone prototyping GUI runner

● Debugger :

Debug environment : MinGW

Debugger command : gdb -nw

Startup commands :

```
set new-console no
set schedule-multiple on
```

○ Deployment simulator :

Simulator command :

Socket connection to target

Available : ☑

Host IP address : This host

Socket port num. :

□ Back trace support

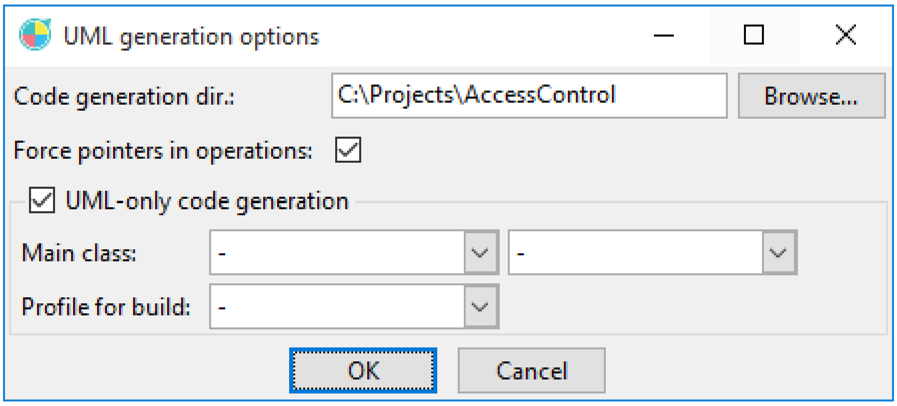
Max. number of events : 16

Size for message data : 32

Typical generation profile to debug a FreeRTOS application with gdb

4.4.3 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:



Code generation dir.: C:\Projects\AccessControl

Force pointers in operations: ☑

☑ UML-only code generation

Main class: - | -

Profile for build: -

The available options are:

- The directory where the files for the UML classes must be generated;
- Wether 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.

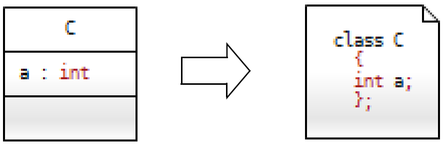
However, added operations in the diagram will be added to the C++ source file, and warnings will be issued for each operation found in the file that is not in the-diagrams. But nothing will ever be removed from the C++ source file.

4.4.4 Generated C++ code

4.4.4.1 Attributes and operations

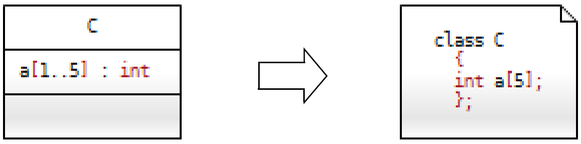
For any class involved in a code generation process, the following attributes and operations are generated in the class header file:

- For an attribute described as "*name : type*" (no multiplicity), an attribute with this name and type is generated.
  Example:



```
class C
{
int a;
};
```

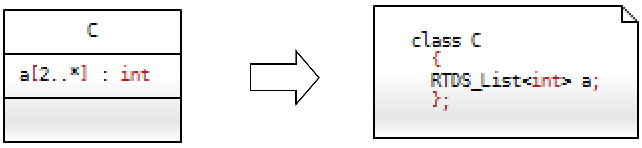
- For an attribute described as "*name [mult] : type*" with a finite multiplicity (no '*'), an attribute with this name and the type "*type [max]*" is generated, with *max* being the maximum multiplicity found in *mult*.
  Example:



```
class C
{
int a[5];
};
```

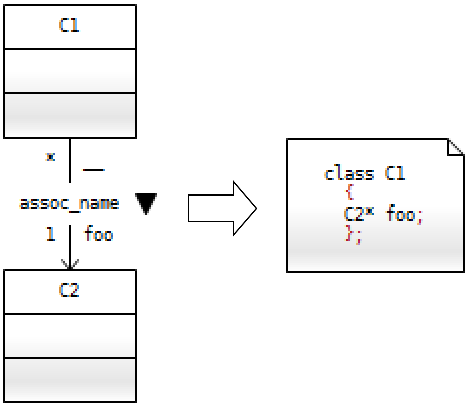
Maximum multiplicity for attribute a is 5
=> array of 5 ints

- 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 299).
  Example:



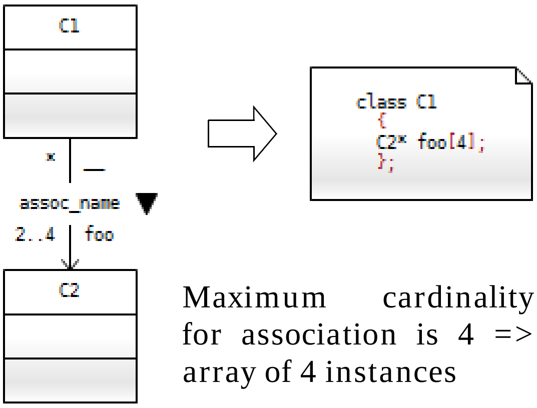
```
class C
{
RTDS_List<int> a;
};
```

- 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:



```
class C1
{
  C2* foo;
};
```

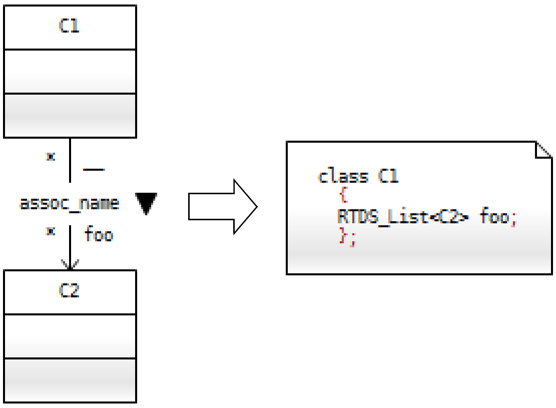
- For a navigable association to class *Class* with a multiple finite cardinality (not 1, not 0..1 and no '*' in cardinality) and a role name *role*, an attribute named role with type *Class* [max]* is generated, where *max* is the maximum cardinality for the association.
  Example:



```
class C1
{
  C2* foo[4];
};
```

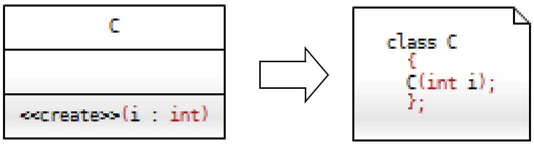
Maximum cardinality for association is 4 => array of 4 instances

- 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 299).
  Example:



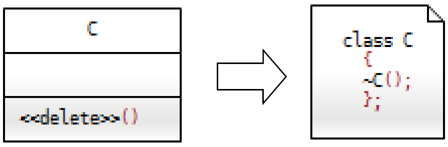
```
class C1
{
  RTDS_List<C2> foo;
};
```

- For an operation named `<<create>>`, the corresponding C++ constructor is generated with the same parameters and no return type.
  Example:



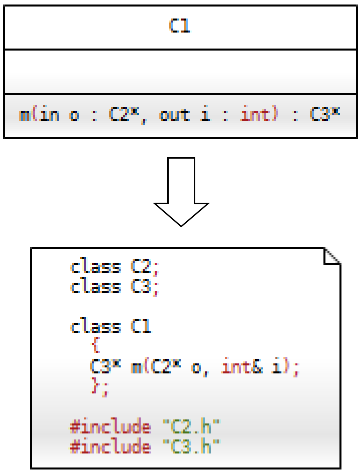
```
class C
  {
  C(int i);
  };
```

- For an operation named `<<delete>>`, the corresponding C++ destructor is generated with no return type. This operation must not have any parameters.
  Example:



```
class C
  {
  ~C();
  };
```

- 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:



```
class C2;
class C3;

class C1
  {
  C3* m(C2* o, int& i);
  };

#include "C2.h"
#include "C3.h"
```

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/cpptemplates`. 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;`).

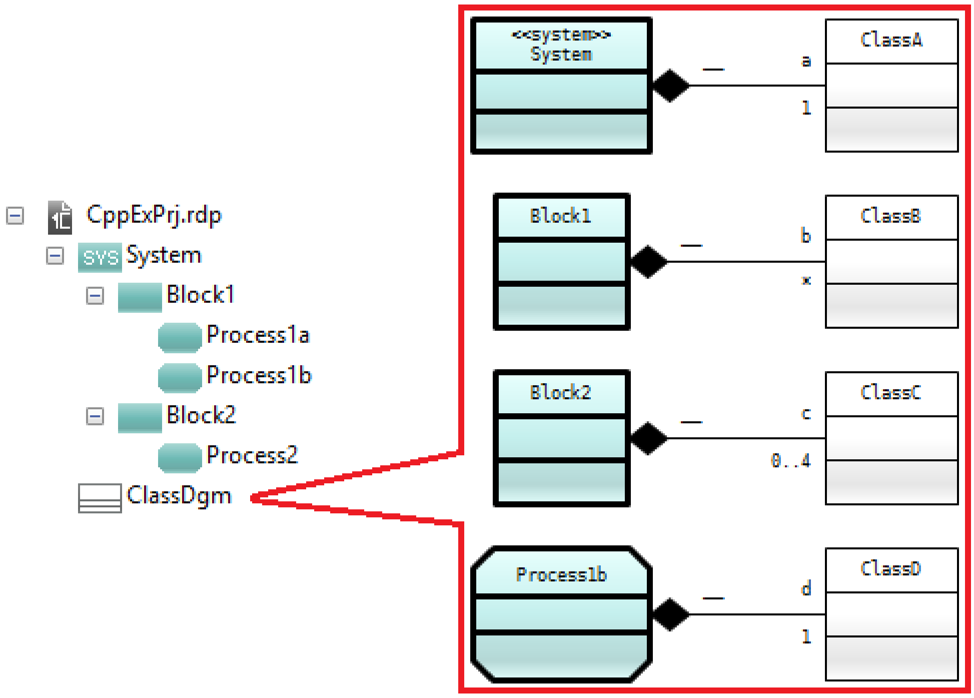
- No attribute should be declared with a type being a class defined in a diagram. If it is, the class won't be recognized and the class will be undeclared when it is used. To define such an attribute, it is mandatory to use an association.

- 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.

4.4.4.2 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`.

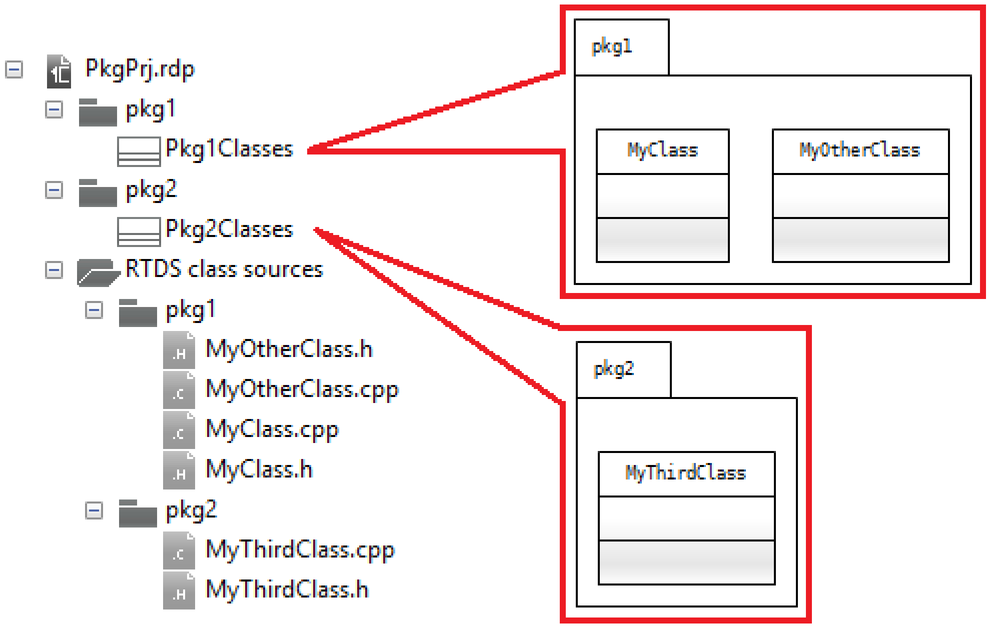
This is achieved by generating the following files:

- 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`.

4.4.4.3 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.

4.4.5 Built in scheduler

The code generation described in “Code generation” on page 246 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.

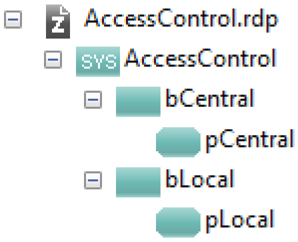
Generating scheduled code also allows integration in an external scheduler.

The following paragraphs describe these points in detail.

4.4.5.1 Deployment diagram for scheduling policy

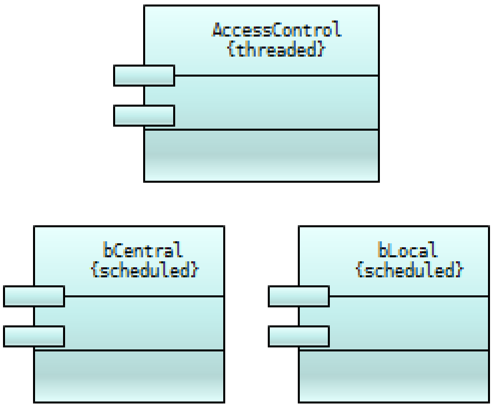
To indicate which processes will have their instances scheduled and which will map to their own task, a UML deployment diagram has to be defined. In this diagram, agents will be represented as components, and the component properties will indicate if the agent is scheduled or threaded. The code generation should then be run on this diagram to actually turn scheduling on in the generated code.

For example, for the following system:



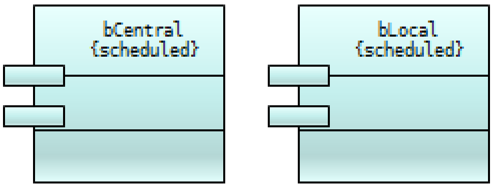
AccessControl.rdp
SYS AccessControl
bCentral
pCentral
bLocal
pLocal

the components in the deployment diagram may be:

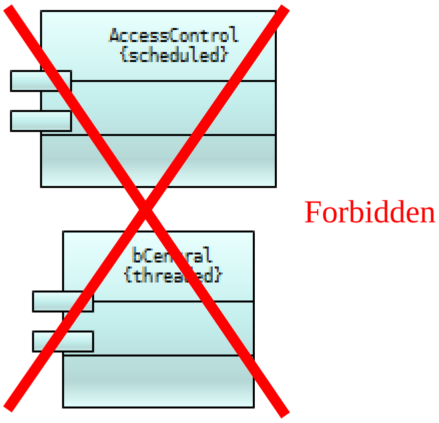


This means that all process instances within blocks `bCentral` will be scheduled together, as well as all process instances within `bLocal`. As the whole system is “threaded”, a RTOS task will be created for each block `bCentral` and `bLocal`.

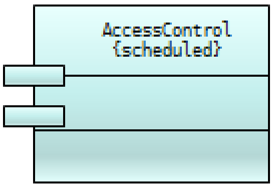
The default policy for agents is actually “threaded”, so the diagram above is in fact equivalent to this one:



Of course, if an agent is scheduled, all the agents it contains will be “scheduled” too, so the following deployment diagram is illegal:

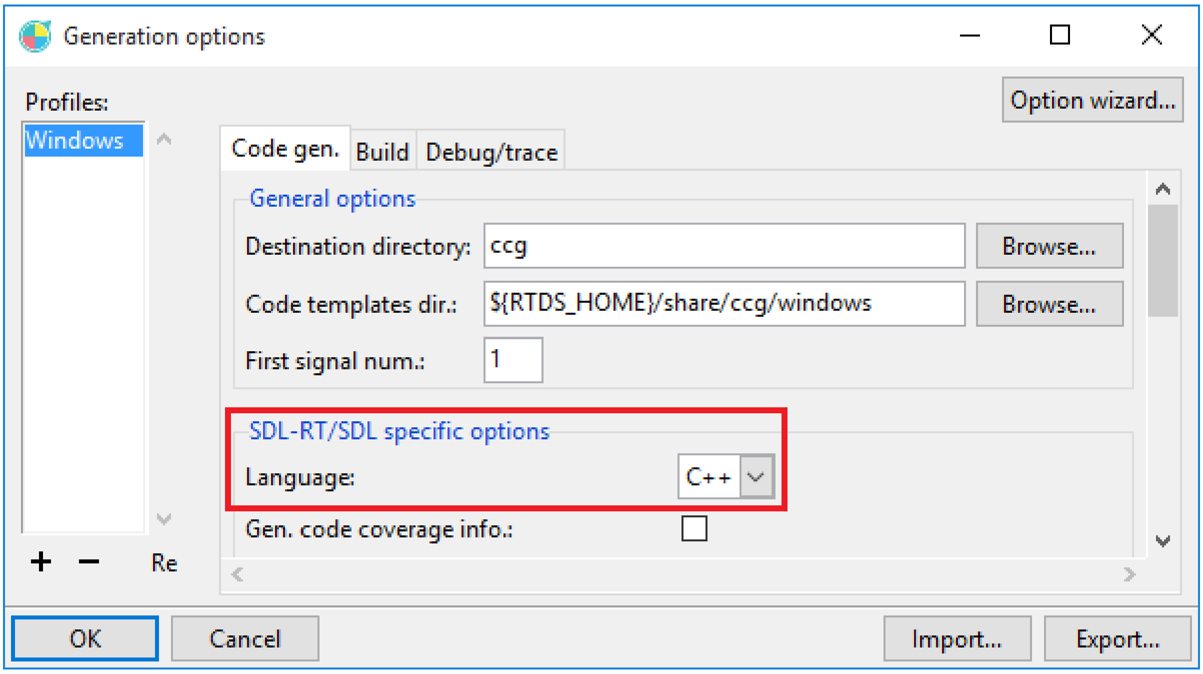


To schedule the whole system, allowing it to be executed without any RTOS, only the following component is necessary:



4.4.5.2 Profiles for scheduling

To be able to use all features with scheduling, the best way is to set the language to C++ in the code generation profile, in the “Code gen.” tab, options “Language”:



General options

Destination directory: ccg

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

First signal num.: 1

SDL-RT/SDL specific options

Language: C++

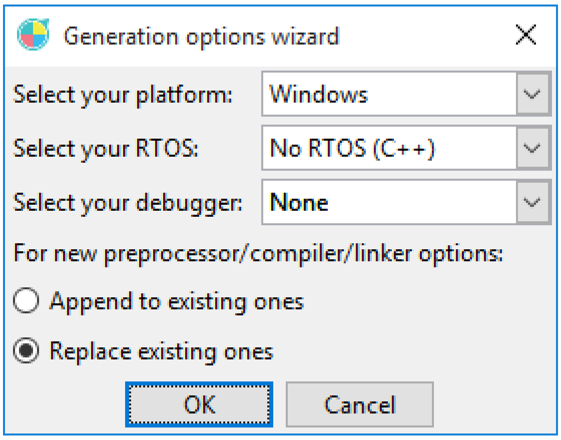
Gen. code coverage info.: □

Generating C code is also possible, but some features such as process classes will not be available.

There are also limitations when generating from SDL-RT diagrams:

- Some constructs cannot be used in variable initializations. This is due to the fact that declarations are simply copied from the declaration symbol to either the attributes part of a C++ class when generating C++, or within a `typedef` for a `struct` type when generating C. Therefore, plain variable definitions such as “`int i;`” will work, but even specifying a default value as in “`int i = 0;`” won’t, as it isn’t a valid attribute or field definition.
- When generating C code, a function is generated for each transition in a process. The context management is done by copying the fields of a `struct` for a given process instance to local variables in the transition function, and to copy back the local variables to the `struct` in the end. This allows to keep the generated code exactly as it is written in the diagram.
  However, this forbids to keep any reference on a local variable for the process. For example, keeping the address of a process variable in another one will not work: when the address is assigned, it would take the address of the local variable in the transition function and point to a wrong location.

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:



Generation options wizard

Select your platform: Windows

Select your RTOS: No RTOS (C++)

Select your debugger: None

For new preprocessor/compiler/linker options:

○ Append to existing ones

● Replace existing ones

Setting the RTOS to “No RTOS (C++)” or “No RTOS (C)” will use a specific integration in `<PragmaDev Studio installation dir.>/share/ccg/rtosless` or `<PragmaDev Studio installation dir.>/share/ccg/crtosless` that does not require any service from the OS. In some cases, dynamic memory allocation may be required - see paragraph “Memory management” on page 305.

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.

4.4.5.3 Memory management

The built-in scheduler uses by default dynamic memory allocation, which is required for features such as dynamic instance creation. However, the crtosless integration allows to prevent dynamic memory allocation for all or part of the generated code. This is done via a set of C preprocessor constants, giving the maximum number of descriptors that can be allocated at the same time:

- `RTDS_MAX_INSTANCES` is for process instance context descriptors;
- `RTDS_MAX_MESSAGES` is for message descriptors;
- `RTDS_MAX_TIMERS` is for timer descriptors;
- `RTDS_MAX_SEMAPHORES` is for semaphore descriptors;

- RTDS_MAX_WAITING_INSTANCES is for descriptors for instances waiting on a semaphores.

All these constants should be passed on the compiler command-line via the -D option or its equivalent. Note that these constants are for descriptors for and not for the element itself. For any given element, there might be other descriptors that need to be allocated temporarily during the life of the system. So for example, if you know that you'll only have 4 process instances in your system, it's safer to set RTDS_MAX_INSTANCES to 6 or above.

If one of the constants RTDS_MAX_TIMERS or RTDS_MAX_SEMAPHORES is defined and set to 0, this will also disable completely all the functions handling timers or semaphores, respectively. This means that if the system actually uses timers or semaphores, the generated code will not compile, since it will use functions that are not defined.

When one of these constants is defined and not 0, the given number of descriptors is allocated in a static array, and "allocating" an element actually uses the first unused descriptor in the array instead of using RTDS_MALLOC. So if all the RTDS_MAX_… constants are defined, it is safe to undefine the RTDS_MALLOC macro as it will never be used. This can be done in a copy of the crtosless profile if dynamic memory allocation is completely forbidden. Note however that the "allocation" from the static array behaves the same way as RTDS_MALLOC: if the number of descriptors is exhausted, the allocation will return a NULL pointer, which will make your system exit in error.

4.4.5.4 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.

4.4.5.5 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.

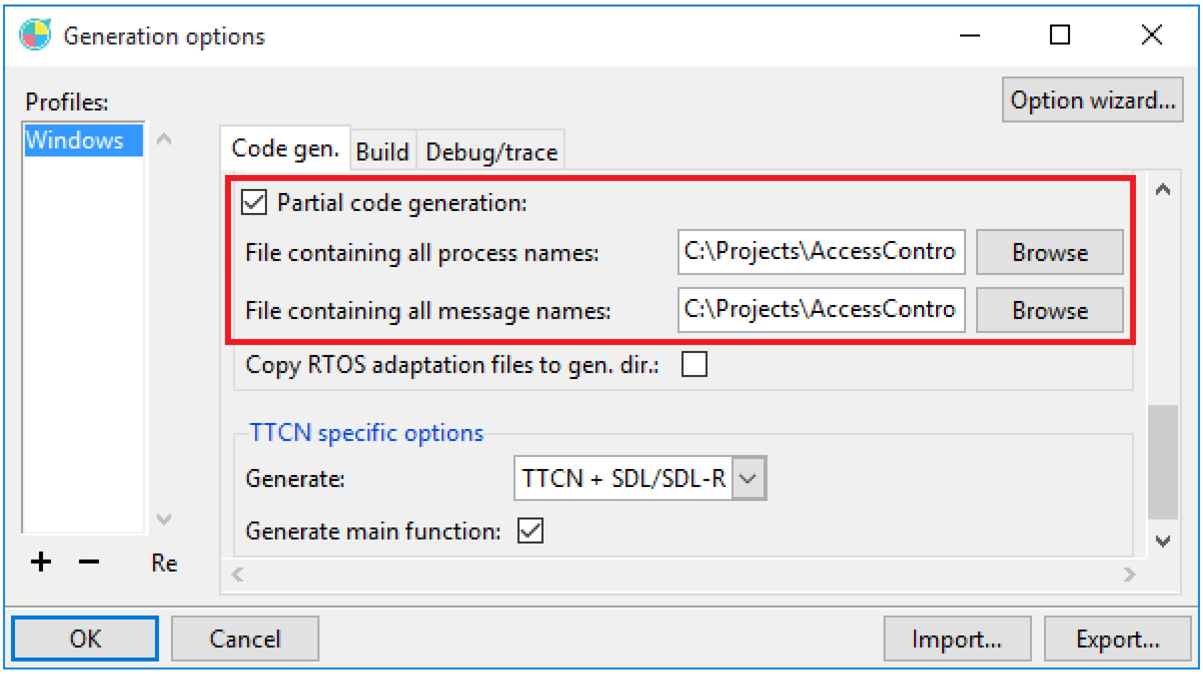
In this case, the handling of time is done via 2 specific functions, called `RTDS_incomingSdlEvent` and `RTDS_SystemTick`:

- `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.

4.4.5.6 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):



Profiles: Windows

☑ Partial code generation:

File containing all process names: C:\Projects\AccessContro

File containing all message names: C:\Projects\AccessContro

Copy RTOS adaptation files to gen. dir.: □

TTCN specific options

Generate: TTCN + SDL/SDL-R

Generate main function: ☑

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`.

4.5 - Good coding practise

4.5.1 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.

4.5.2 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.

4.5.3 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.

4.6 - Model Debugger

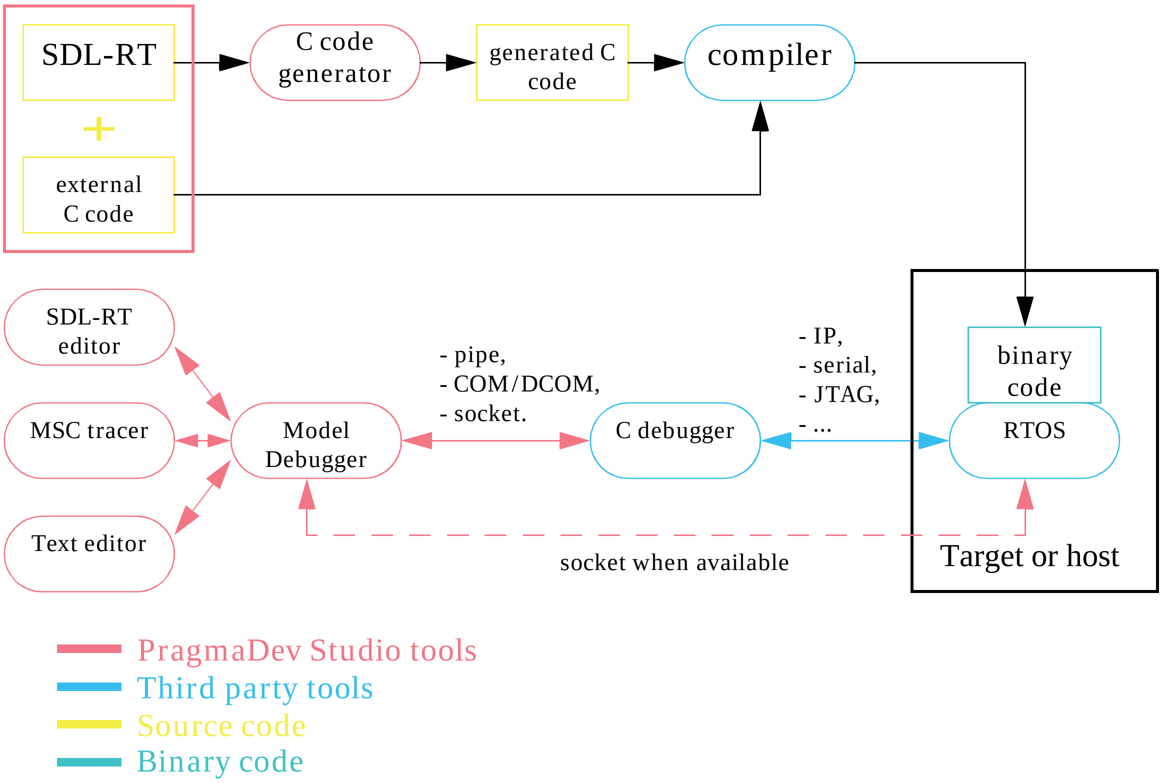
The Model Debugger relies on classical C debuggers or cross debuggers to allow graphical debugging and SDL-RT oriented information.

Currently supported debuggers are:

- Tornado
- gdb
- MinGW
- Tasking Cross View Pro C166/ST10
- XRAY
- Multi 2000

4.6.1 Debugger architecture

The Model 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 Model 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.

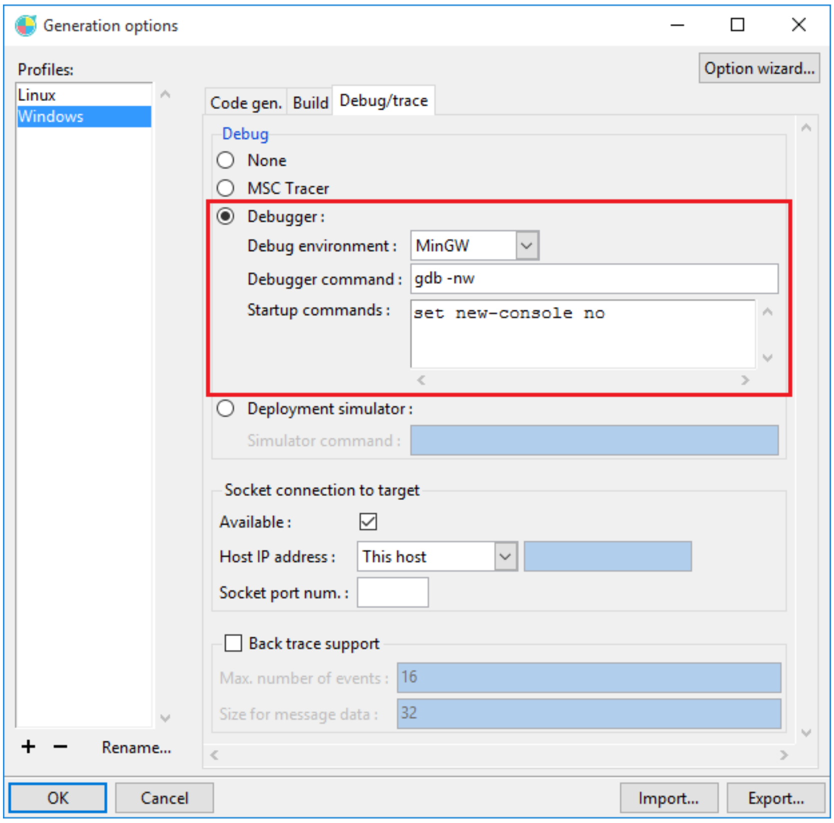
4.6.2 Launching the Model Debugger

Before starting the Model 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:



Debug
○ None
○ MSC Tracer
● Debugger :
Debug environment : MinGW
Debugger command : gdb -nw
Startup commands : set new-console no
○ Deployment simulator :
Simulator command :

Socket connection to target
Available : ☑
Host IP address : This host
Socket port num. :

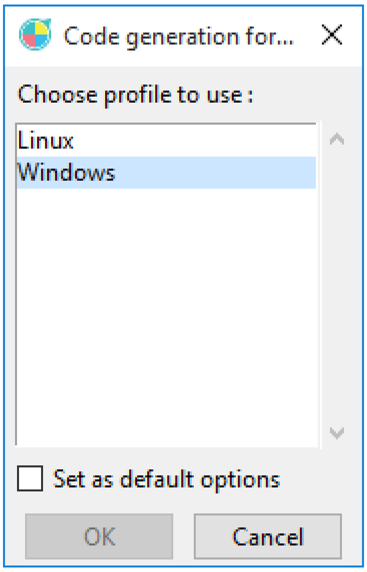
□ Back trace support
Max. number of events : 16
Size for message data : 32

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 Model 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:



Code generation for...

Choose profile to use :

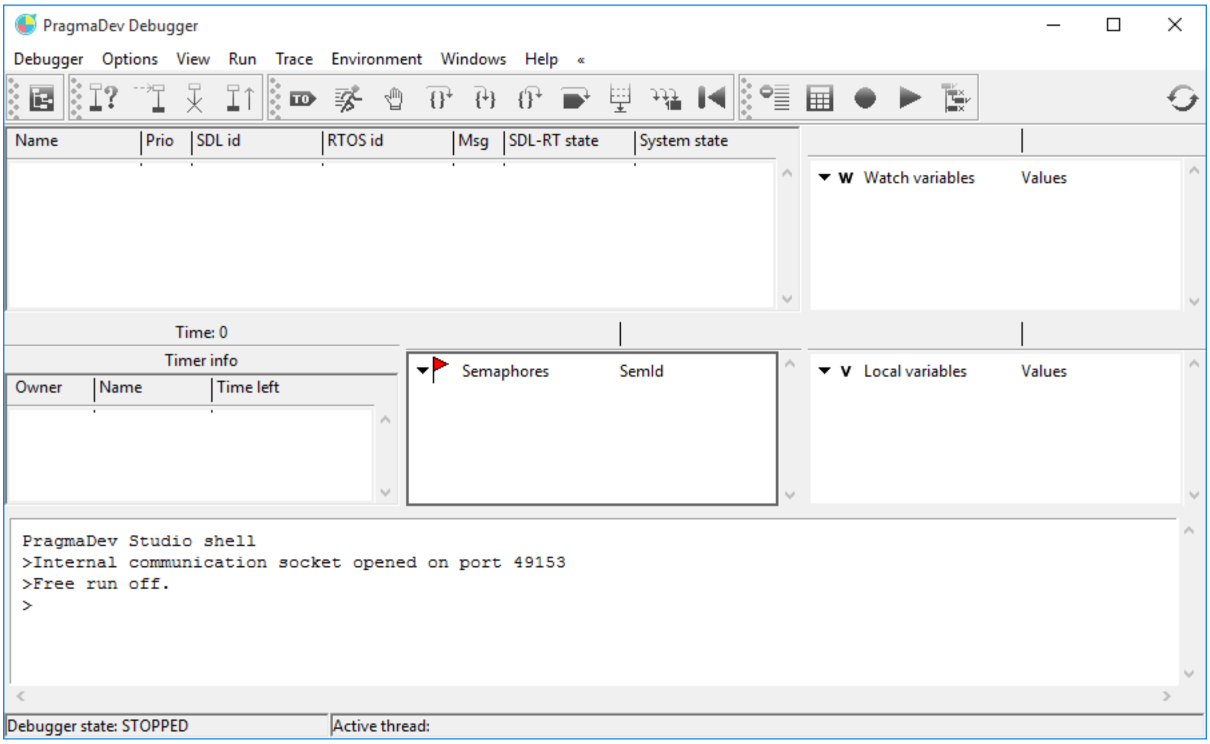
Linux
Windows

□ Set as default options

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 Model Debugger window is started automatically and you are ready to debug your system!



```
PragmaDev Studio shell
>Internal communication socket opened on port 49153
>Free run off.
>
```

Debugger state: STOPPED

Active thread:

The Model Debugger window

The Model 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.

4.6.3 Stepping levels

Since your source code is a composite of SDL and C and considering some code has also be 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 Model 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

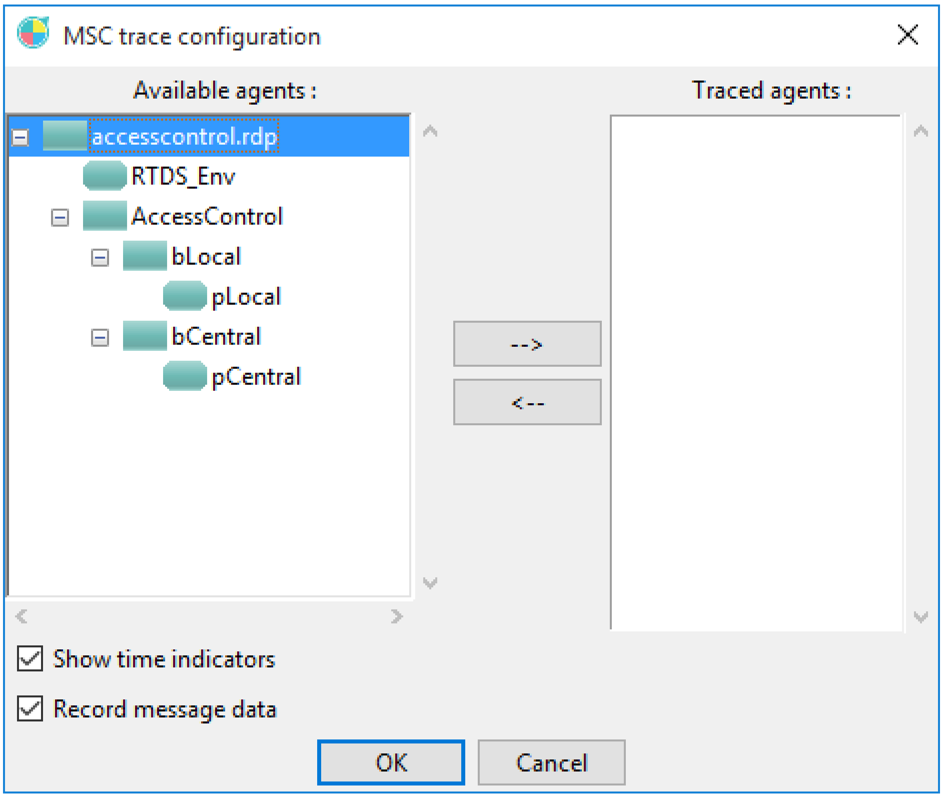
  This will run the system until one of the external message queue is empty. When using the scheduler, its internal queue is read until it is empty before the external queue is read. If the whole system is scheduled this feature can be used as a run until timer.

4.6.4 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:



☑ Show time indicators

☑ Record message data

  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.

The tracer window itself is described in “Tracer window” on page 383.

4.6.5 Displayed information

The Model 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 321.

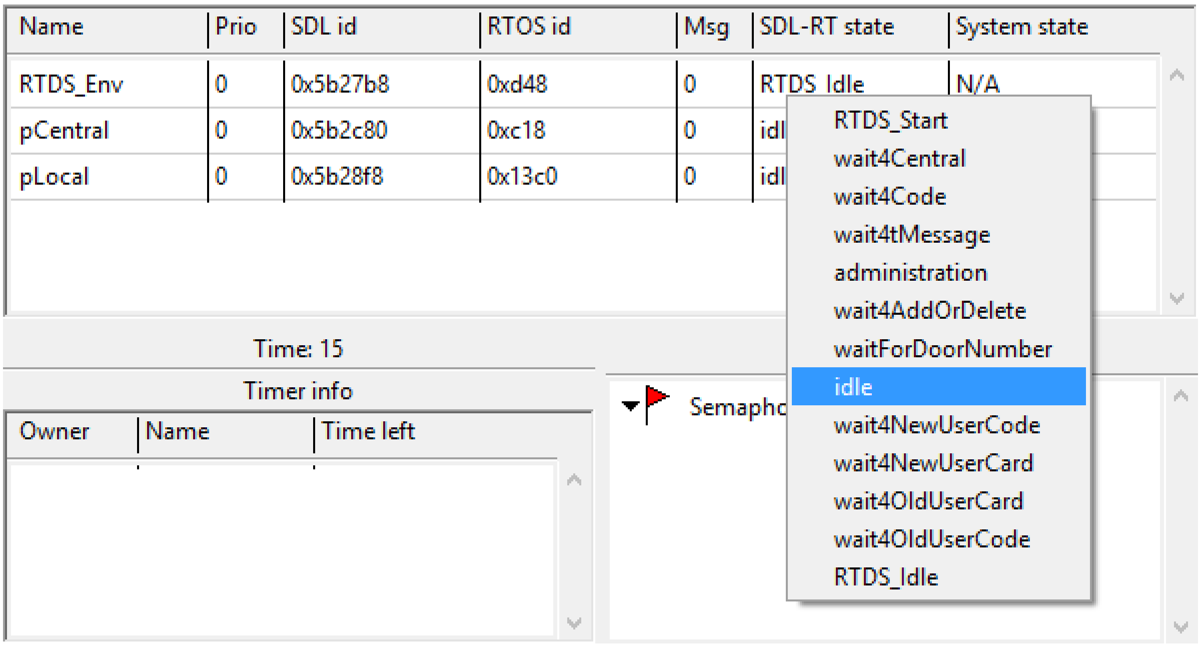
4.6.5.1 Processes

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.



Name	Prio	SDL id	RTOS id	Msg	SDL-RT state	System state
RTDS_Env	0	0x5b27b8	0xd48	0	RTDS_Idle	N/A
pCentral	0	0x5b2c80	0xc18	0	idl	
pLocal	0	0x5b28f8	0x13c0	0	idl	

Time: 15

Timer info

Owner	Name	Time left

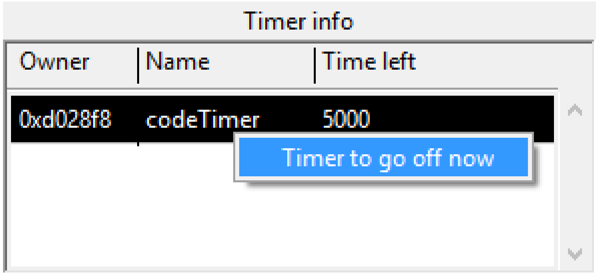
SDL-RT state modification example

4.6.5.2 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.



Timer info

Owner	Name	Time left
0xd028f8	codeTimer	5000

Example: have codeTimer to go off

4.6.5.3 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 Model 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 Model 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.

4.6.5.4 Watch

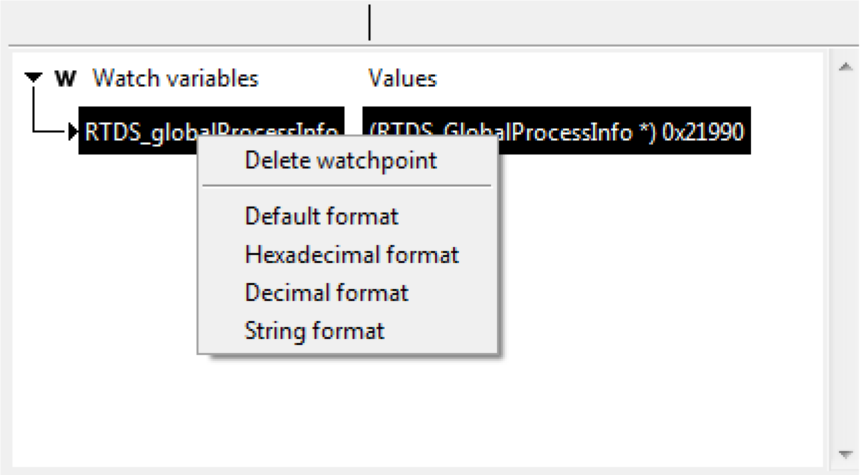
There are several ways to add a variable in the *Watch window*:

- From the shell
  Type the following command in the shell:
  `watch add <variable name>`
- From the text editor when the Model 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 *Model Debugger Watch window* with right mouse button as shown below:

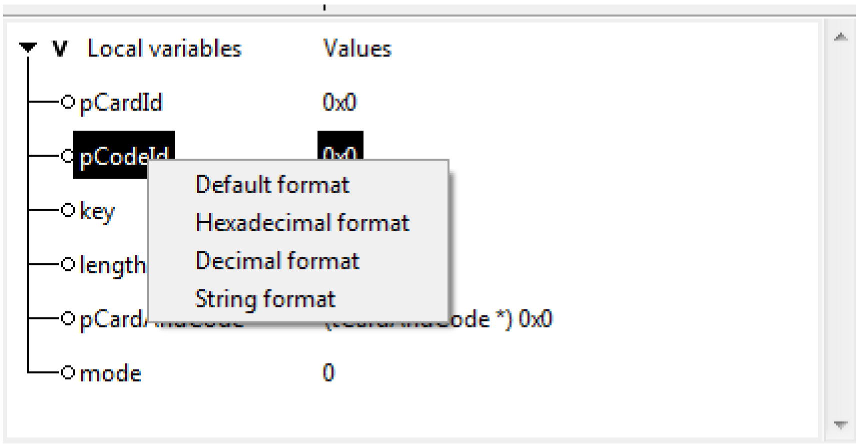


The *Watch window* also allows to modify the value of variables. To do so double click on the value of the variable to be modified. Press <Return> and the value is updated.

4.6.5.5 Local variables

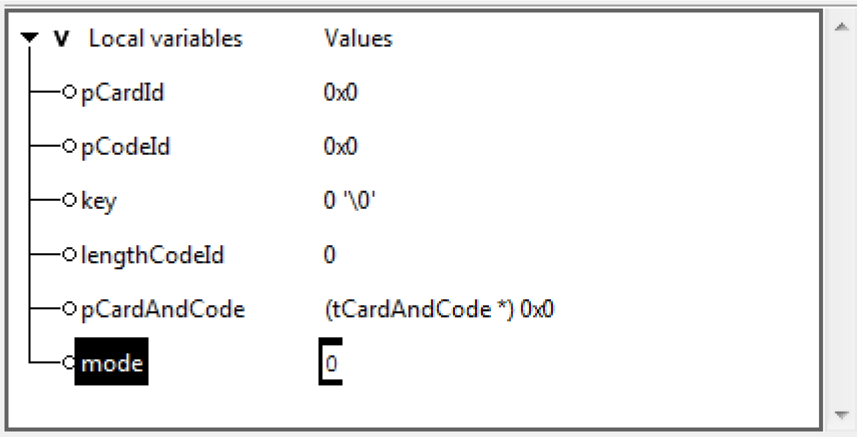
When stepping through the code the Model Debugger automatically displays the local variables of the current stack frame. That gathers all local variables of the current C function including the arguments of the function. Nothing has to be done to update the Model Debugger *Local variable window*.

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 Model 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.



Local variables	Values
pCardId	0x0
pCodeId	0x0
key	0 '\0'
lengthCodeId	0
pCardAndCode	(tCardAndCode *) 0x0
mode	0

Setting a local variable value example

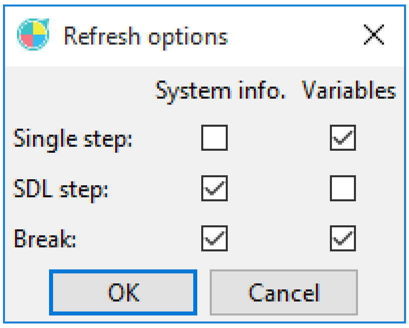
4.6.5.6 Refresh options

The information displayed in the Model 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.



	System info.	Variables
Single step:	□	☑
SDL step:	☑	□
Break:	☑	☑

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 Model Debugger but no refresh action is done.

4.6.6 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.

4.6.6.1 shell commands

To list all the available commands in this category, type:
`help shell`

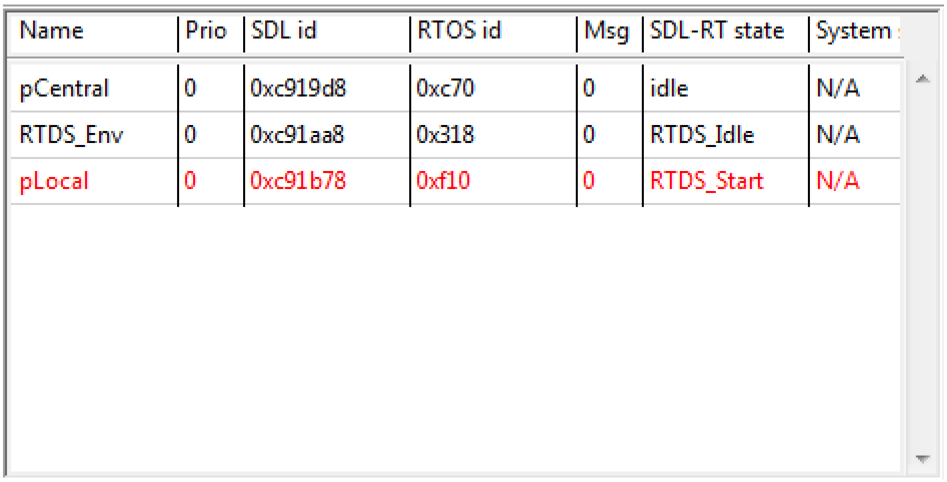
It will list the following commands:

```
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`

4.6.6.2 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 Model 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.

4.6.6.3 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 `systemQueueSetNextReceiverName` 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 model 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 model coverage results stored in it.

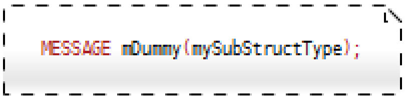
4.6.6.4 variables commands

To list all the available commands in this category, type:
`help variables`

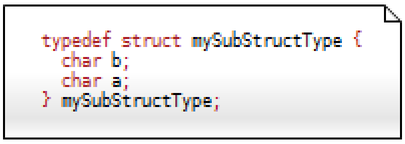
It will list the following commands:

```
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:



```
MESSAGE mDummy(mySubStructType);
```



```
typedef struct mySubStructType {
  char b;
  char a;
} mySubStructType;
```

Shell commands to define a variable based on the type:

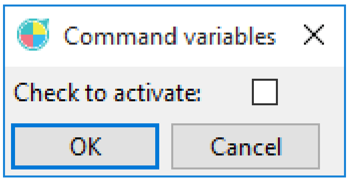
```
>varFromType myVar mySubStructType
>print myVar
|{b|= |,a|=0|}
>varFieldSet myVar.b=z
>varFieldSet myVar.a=666
>print myVar
|{b|=z|,a|=666|}
```

```
>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:


Command variables ×

Check to activate: □

OK Cancel

prints `1` if checked or `0` if unchecked.

4.6.6.5 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.

4.6.6.6 customization commands

To list all the available commands in this category, type:
`help customization`

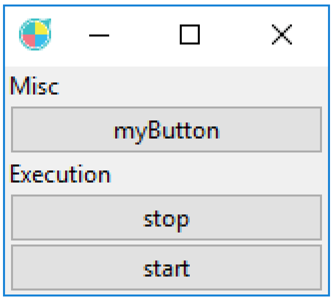
It will list the following commands:

```
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
```

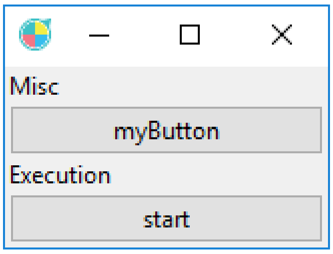
- Button windows
  It is possible to create user-defined buttons and to associate shell commands.
  Here is an example of a button window:

```
>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.

4.6.7 Status bar

The status bar is divided in two parts:

- The Model Debugger internal state
  The Model 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 Model Debugger is stuck. Restart the Model Debugger.

Table 5: Model Debugger internal states

- The active thread
  The active thread is displayed in the right part of the status bar when known.

4.6.8 Breakpoints

4.6.8.1 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.

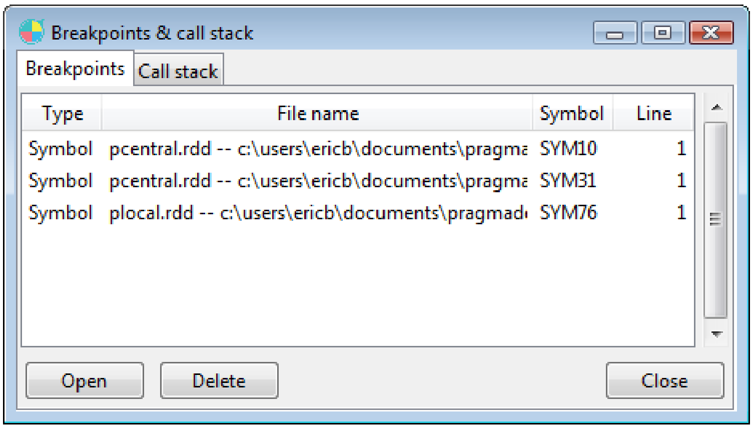
Note that setting a breakpoint interactively will record the “`break`” command in the shell history. This can be especially useful for breakpoints set on symbol, where the symbol identifier is not directly visible in the diagram.

- 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 STOP 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 STOP button in the debug toolbar.

4.6.8.2 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:



Type	File name	Symbol	Line
Symbol	pcentral.rdd -- c:\users\ericb\documents\pragma	SYM10	1
Symbol	pcentral.rdd -- c:\users\ericb\documents\pragma	SYM31	1
Symbol	plocal.rdd -- c:\users\ericb\documents\pragmad	SYM76	1

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.

4.6.8.3 Deleting breakpoints

Breakpoints can be deleted from:

- The shell with the delete command:

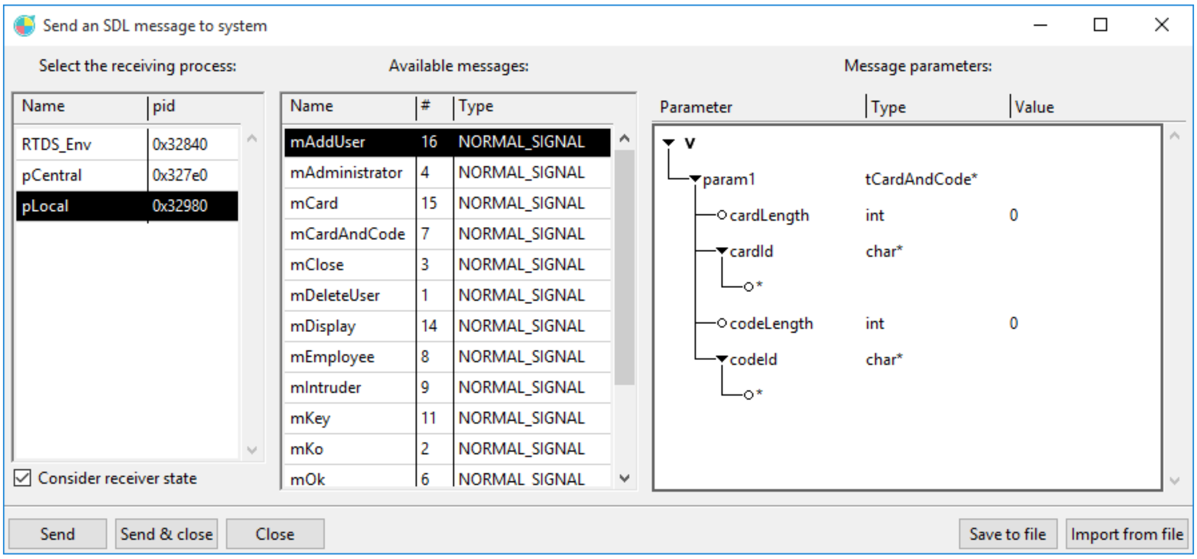
`delete <breakpoint number>`
where the `breakpoint number` is the number listed from the `list` command.

- The breakpoint list window, as explained in “Listing breakpoints” on page 333.
- 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.

4.6.9 Sending SDL messages to the running system

4.6.9.1 Send SDL message window

The Model 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:



Send an SDL message to system

Select the receiving process:

Name	pid
RTDS_Env	0x32840
pCentral	0x327e0
pLocal	0x32980

☑ Consider receiver state

Available messages:

Name	#	Type
mAddUser	16	NORMAL_SIGNAL
mAdministrator	4	NORMAL_SIGNAL
mCard	15	NORMAL_SIGNAL
mCardAndCode	7	NORMAL_SIGNAL
mClose	3	NORMAL_SIGNAL
mDeleteUser	1	NORMAL_SIGNAL
mDisplay	14	NORMAL_SIGNAL
mEmployee	8	NORMAL_SIGNAL
mIntruder	9	NORMAL_SIGNAL
mKey	11	NORMAL_SIGNAL
mKo	2	NORMAL_SIGNAL
mOk	6	NORMAL_SIGNAL

Message parameters:

Parameter	Type	Value
v		
param1	tCardAndCode*	
cardLength	int	0
cardId	char*	
*		
codeLength	int	0
codeId	char*	
*		

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:

- 0 for null pointers,
- ’ ’ for pointers to be allocated on the target.

To specify real pointer values please us the shell command described below.

4.6.9.2 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.

4.6.9.3 Prototyping GUI

This is the easiest way to interact with the system. The interface editor is described in "Prototyping GUI" on page 151. The interface is started with the ▦ quick button.

4.6.9.4 Button windows

Interaction commands can easily be assigned to graphical buttons as described in "interaction commands" on page 326.

4.6.10 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 Model Debugger is a good way to trigger specific test scenarios.

4.6.11 Model coverage

The debugger's *Get model coverage* button gets the model coverage analysis results for the running system so far. This feature is available only if the *Gen. model coverage info.* is checked in the generation options (see "Profiles" on page 248).

For more details on model coverage results, see "Code coverage results" on page 161.

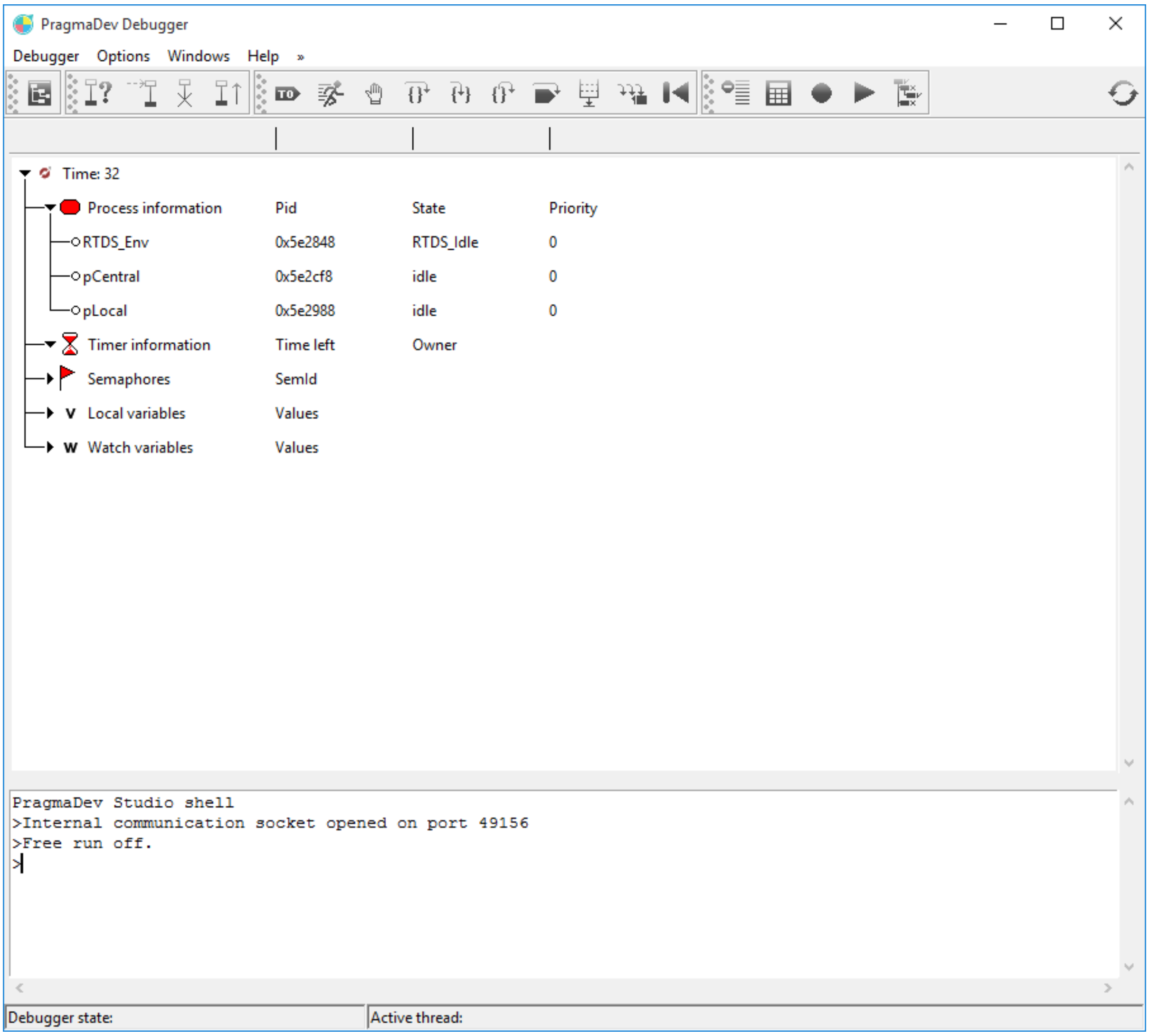
4.6.12 Connecting an external tool

An external tool can be connected to the debugger exactly the same way as for the SDL simulator, as described in "Connecting an external tool" on page 220. Both modes are available: the normal mode will allow to send debugger shell commands and receive the answers via a socket connection, and the XML mode can be used for a more structured dialog.

4.6.13 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.



Time: 32

	Pid	State	Priority
Process information	Pid	State	Priority
RTDS_Env	0x5e2848	RTDS_Idle	0
pCentral	0x5e2cf8	idle	0
pLocal	0x5e2988	idle	0
Timer information	Time left	Owner	
Semaphores	SemId		
Local variables	Values		
Watch variables	Values		

```
PragmaDev Studio shell
>Internal communication socket opened on port 49156
>Free run off.
>
```

Debugger state: Active thread:

It is the 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 Model Debugger.

4.6.14 Command line debug

The Model 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.

5 - PragmaDev Tester

PragmaDev Tester helps testers to write validation and integration tests with an abstract dedicated language. A substantial number of test cases with this technology are published by international standardization bodies to ensure conformance to their specifications.

PragmaDev Tester is based on TTCN-3 technology. For information on the language itself, please refer to the language reference documents.

5.1 - 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.
- Inter-module browsing capabilities are provided: for each element in a TTCN module - type, constant, component, template, function, and so on -, its definition can be displayed, and it is possible to jump to its definition in its defining module.
- TTCN-3 test suites can be simulated in the PragmaDev Simulator along with the system they test.

5.2 - PragmaDev extensions

To provide support for the same concepts within TTCN and SDL, an extension has been introduced to make TTCN modules aware of the notion of package, that is not present in the language. This extension is typically usd when importing a set of definitions from an ASN.1 module in TTCN: in SDL, these definitions are often put in a separate package; but TTCN has no notion of package and no syntax to import a module in a given package.

PragmaDev Studio uses the standard TTCN extension mechanism to provide a syntax to do so. For example, the following clause can be written in a TTCN module:

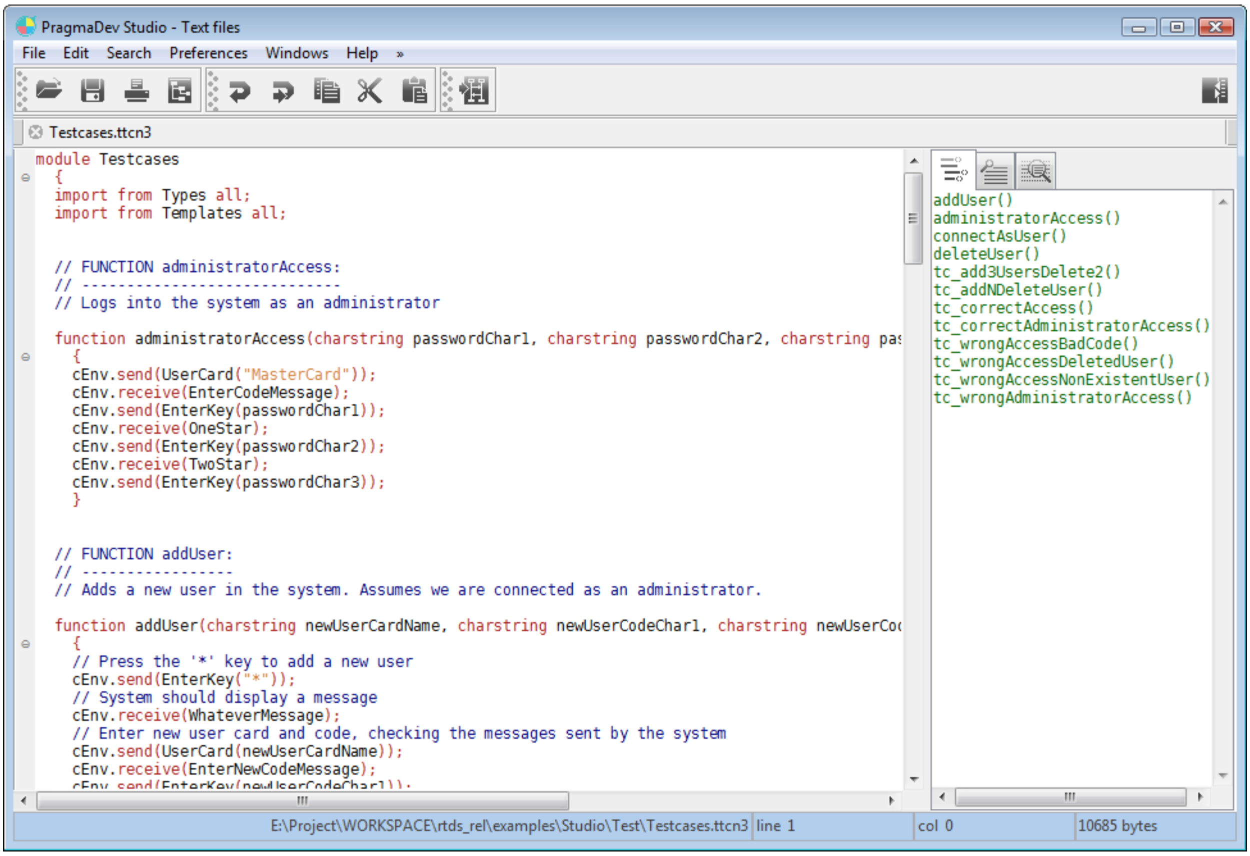
```
import  from Types language "ASN.1:2002" all
        with { extension "PragmaDev:parent_package=MyPackage" };
```

imports the ASN.1 module named `Types` from the package `MyPackage`.

This extension is supported in edition (code completion and navigation consider it when looking for elements), simulation and C++ code generation.

5.3 - 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:


PragmaDev Studio - Text files
File Edit Search Preferences Windows Help »
Testcases.ttcn3

```
module Testcases
  {
  import from Types all;
  import from Templates all;


  // FUNCTION administratorAccess:
  // -----------------------------
  // Logs into the system as an administrator

  function administratorAccess(charstring passwordChar1, charstring passwordChar2, charstring pas
    {
    cEnv.send(UserCard("MasterCard"));
    cEnv.receive(EnterCodeMessage);
    cEnv.send(EnterKey(passwordChar1));
    cEnv.receive(OneStar);
    cEnv.send(EnterKey(passwordChar2));
    cEnv.receive(TwoStar);
    cEnv.send(EnterKey(passwordChar3));
    }


  // FUNCTION addUser:
  // -----------------
  // Adds a new user in the system. Assumes we are connected as an administrator.

  function addUser(charstring newUserCardName, charstring newUserCodeChar1, charstring newUserCod
    {
    // Press the '*' key to add a new user
    cEnv.send(EnterKey("*"));
    // System should display a message
    cEnv.receive(WhateverMessage);
    // Enter new user card and code, checking the messages sent by the system
    cEnv.send(UserCard(newUserCardName));
    cEnv.receive(EnterNewCodeMessage);
    cEnv.send(EnterKey(newUserCodeChar1));
```

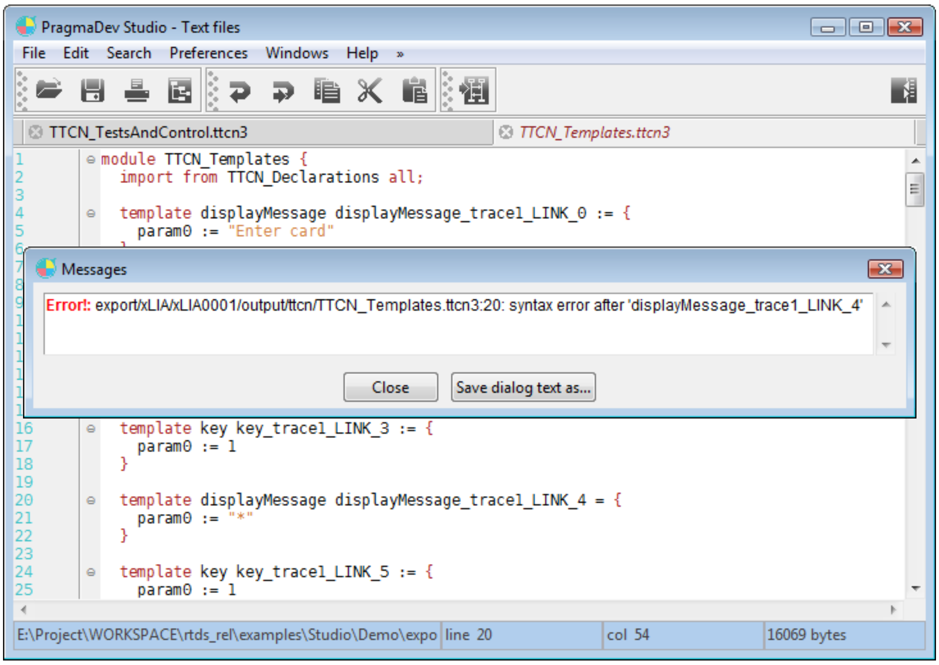
addUser()
administratorAccess()
connectAsUser()
deleteUser()
tc_add3UsersDelete2()
tc_addNDeleteUser()
tc_correctAccess()
tc_correctAdministratorAccess()
tc_wrongAccessBadCode()
tc_wrongAccessDeletedUser()
tc_wrongAccessNonExistentUser()
tc_wrongAdministratorAccess()

E:\Project\WORKSPACE\rtds_rel\examples\Studio\Test\Testcases.ttcn3 | line 1 | col 0 | 10685 bytes

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.

PragmaDev Studio supports full syntax check of the TTCN-3 files via the menu “File / Check syntax semantics...” in the source file editor:


PragmaDev Studio - Text files
File Edit Search Preferences Windows Help »
TTCN_TestsAndControl.ttcn3
TTCN_Templates.ttcn3
Messages
Close
Save dialog text as...

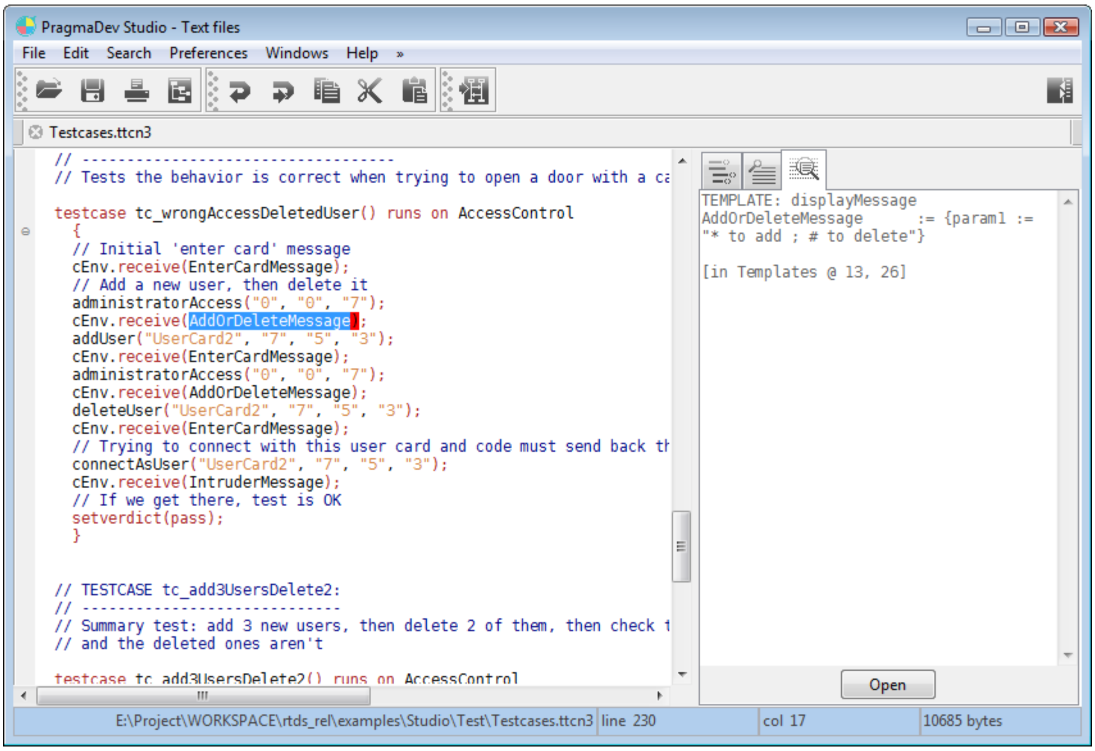
```
module TTCN_Templates {
  import from TTCN_Declarations all;

  template displayMessage displayMessage_trace1_LINK_0 := {
    param0 := "Enter card"
```

```
Error!: export/xLIA/xLIA0001/output/ttcn/TTCN_Templates.ttcn3:20: syntax error after 'displayMessage_trace1_LINK_4'
```

```
  template key key_trace1_LINK_3 := {
    param0 := 1
  }

  template displayMessage displayMessage_trace1_LINK_4 = {
    param0 := "*"
  }

  template key key_trace1_LINK_5 := {
    param0 := 1
```

E:\Project\WORKSPACE\rtds_rel\examples\Studio\Demo\expo | line 20 | col 54 | 16069 bytes

PragmaDev Studio also allows to display the definition of any element in a TTCN module in the side panel: type, constant, variable, component, template, altstep, function, and so on. To do so, just select the name of the element and select the “*Show selected item definition*” entry in the “*Search*” menu, or press F6. The definition is displayed in the zone on the right side of the text:


PragmaDev Studio - Text files
File Edit Search Preferences Windows Help »
Testcases.ttcn3
Open

```
// ------------------------------------
// Tests the behavior is correct when trying to open a door with a ca

testcase tc_wrongAccessDeletedUser() runs on AccessControl
  {
  // Initial 'enter card' message
  cEnv.receive(EnterCardMessage);
  // Add a new user, then delete it
  administratorAccess("0", "0", "7");
  cEnv.receive(AddOrDeleteMessage);
  addUser("UserCard2", "7", "5", "3");
  cEnv.receive(EnterCardMessage);
  administratorAccess("0", "0", "7");
  cEnv.receive(AddOrDeleteMessage);
  deleteUser("UserCard2", "7", "5", "3");
  cEnv.receive(EnterCardMessage);
  // Trying to connect with this user card and code must send back th
  connectAsUser("UserCard2", "7", "5", "3");
  cEnv.receive(IntruderMessage);
  // If we get there, test is OK
  setverdict(pass);
  }


// TESTCASE tc_add3UsersDelete2:
// ------------------------------
// Summary test: add 3 new users, then delete 2 of them, then check t
// and the deleted ones aren't

testcase tc_add3UsersDelete2() runs on AccessControl
```

```
TEMPLATE: displayMessage
AddOrDeleteMessage      := {param1 :=
"* to add ; # to delete"}

[in Templates @ 13, 26]
```

E:\Project\WORKSPACE\rtds_rel\examples\Studio\Test\Testcases.ttcn3 | line 230 | col 17 | 10685 bytes

The "*Open*" button under the displayed definition opens the defining module for the element at the line where it is defined. It is also possible to do so by selecting the "*Go to item definition*" entry in the "*Search*" menu, or by pressing Shift + F6.

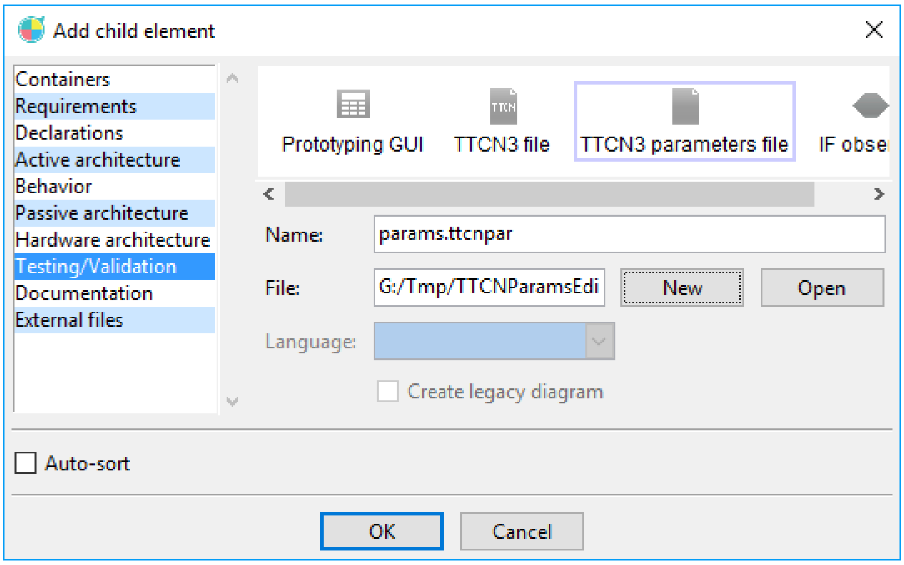
Note that for altsteps, functions and testcases, only the header is displayed, not the full definition. For all other elements, the full definition is displayed.

5.4 - TTCN-3 parameters editor

TTCN-3 allows to define values that have to be provided externally, for example to control the execution of the testcases in the test suite. These values are called module parameters and are declared using the keyword `modulepar` in the TTCN code.

Since the generated executable for TTCN (see "C++ code generation" on page 351) includes a user interface that allows to run testcases individually, there are also cases where the required information to run a testcase has to be provided from the outside: if a testcase has a parameter, since there is no code calling that testcase which can provided the value for the parameter, it has to be provided externally.

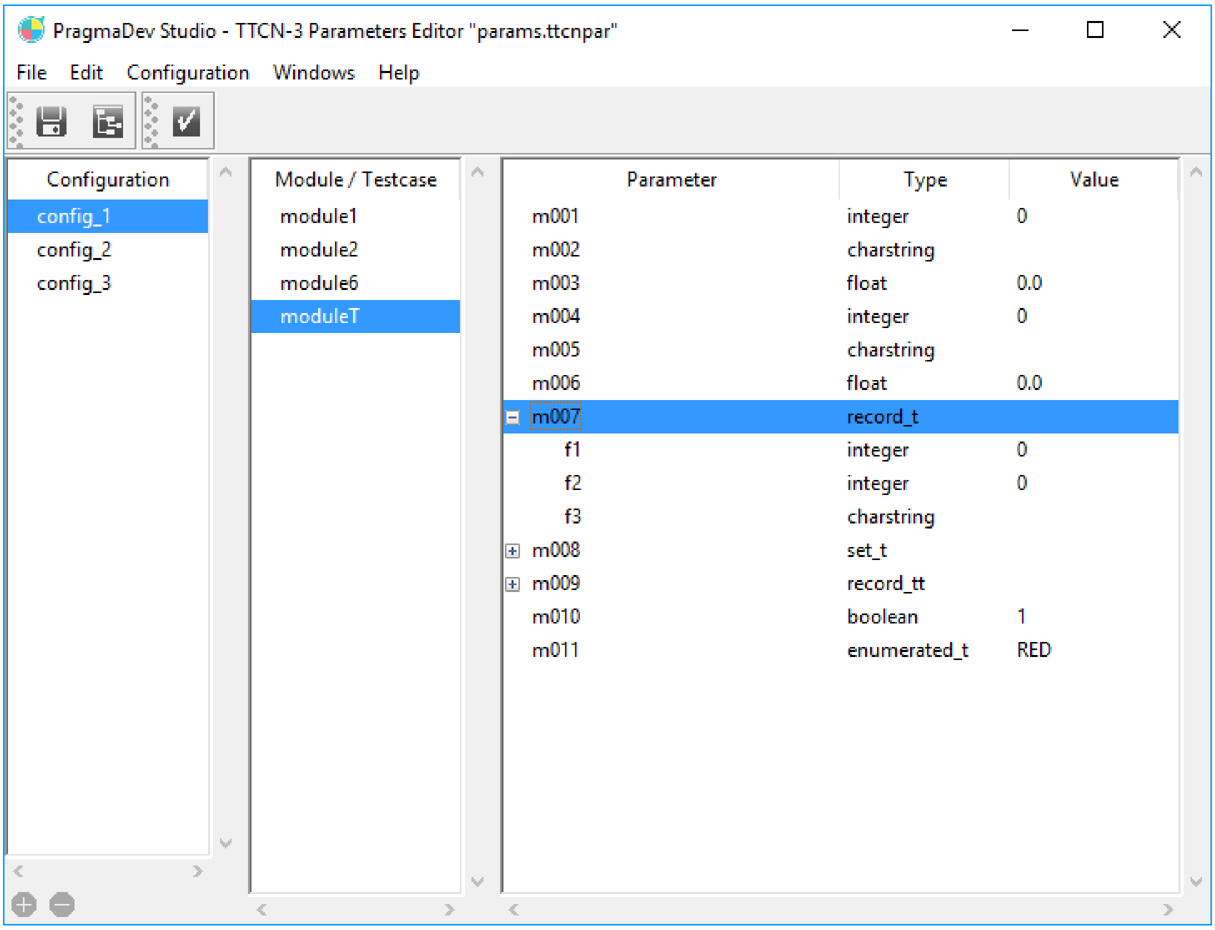
To define these values, PragmaDev Studio includes an editor for all TTCN-3 parameters, for modules and testcases. These values are stored in a TTCN-3 parameters file, that can be added anywhere in the project either via the "Element" / "Add child..." menu or the "Add child element..." contextual menu. Parameter files are found in the "Testing/Validation" category:



Name: params.ttcnpar

File: G:/Tmp/TTCNParamsEdi

Double-clicking a TTCN-3 parameters file in the project will open the editor:


PragmaDev Studio - TTCN-3 Parameters Editor "params.ttcnpar"
File
Edit
Configuration
Windows
Help

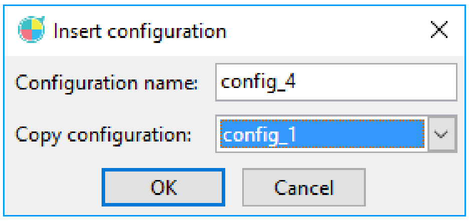
Configuration
config_1
config_2
config_3

Module / Testcase
module1
module2
module6
moduleT

Parameter	Type	Value
m001	integer	0
m002	charstring	
m003	float	0.0
m004	integer	0
m005	charstring	
m006	float	0.0
⊟ m007	record_t	
f1	integer	0
f2	integer	0
f3	charstring	
⊞ m008	set_t	
⊞ m009	record_tt	
m010	boolean	1
m011	enumerated_t	RED

The editor consists of three parts listing the Configurations, Modules/Testcases, and Parameters. The configurations allow to assign several values to the parameters while storing them into a single file. Each of the values will be identified by the configuration name. Selecting a configuration will update the list of modules & testcases shown in the middle part of the editor, and selecting a module or testcase will trigger the display of its parameters. Only modules and test cases with parameters will be displayed.

To insert a new configuration use either the ⊕ button or the “Configuration / Insert...” menu:



Configuration name: config_4

Copy configuration: config_1

A new configuration can be either created as a copy of an existing one or from the project. To create a new configuration from project leave the “Copy configuration” choice to empty. This will create the new configuration after checking the syntax/semantics of all TTCN-3 files found in the project. This step is needed to ensure correctnes of all parameters. Checking an existing configuration is possible via the ☑ button or the “Configuration / Check...” menu. To remove a configuration use the ⊖ button or the “Configuration / Remove” menu.

The value of a parameter can be edited by double-clicking it. Parameters of type boolean, integer, float, and charstring are directly editable. For enumerated types a list will be shown with possible values to choose from. Structured type parameters `record` and `set` are supported if their fields are of editable type (`boolean`, `integer`, float, `charstring`, `enumerated`) or structured type (`record` or `set`).

Once defined, these parameters can be used in the standalone executable generated from the TTCN test suite. See “C++ code generation” on page 351 for details.

5.5 - 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 Model Simulator” on page 192. 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.

5.5.1 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).

5.5.2 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 := ?};
  ```

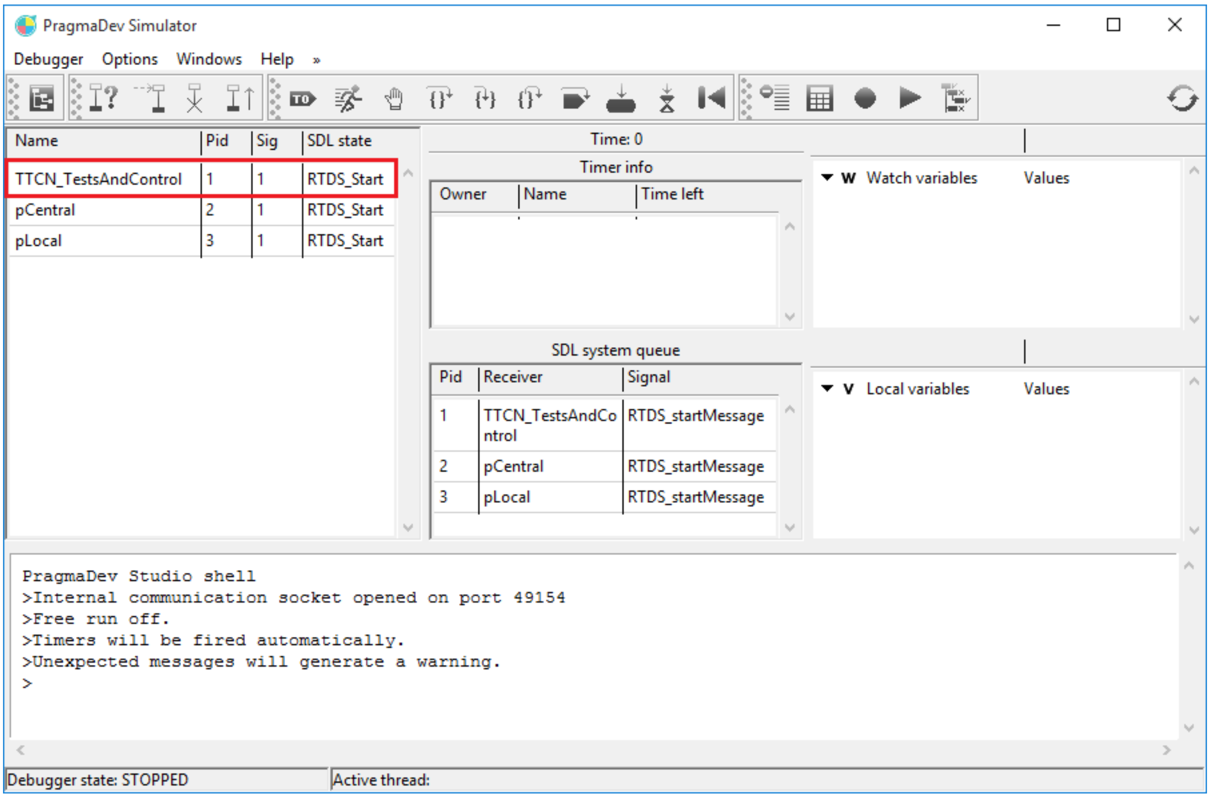
  will generate an error, as constraint on `p1` in `origin_segment` is specified via the complex value `origin`.
  Simple constraints on strings specified with `pattern` are not supported, but the `regexp` predefined function is not.

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.

5.5.3 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:



Name	Pid	Sig	SDL state
TTCN_TestsAndControl	1	1	RTDS_Start
pCentral	2	1	RTDS_Start
pLocal	3	1	RTDS_Start

Owner	Name	Time left

SDL system queue

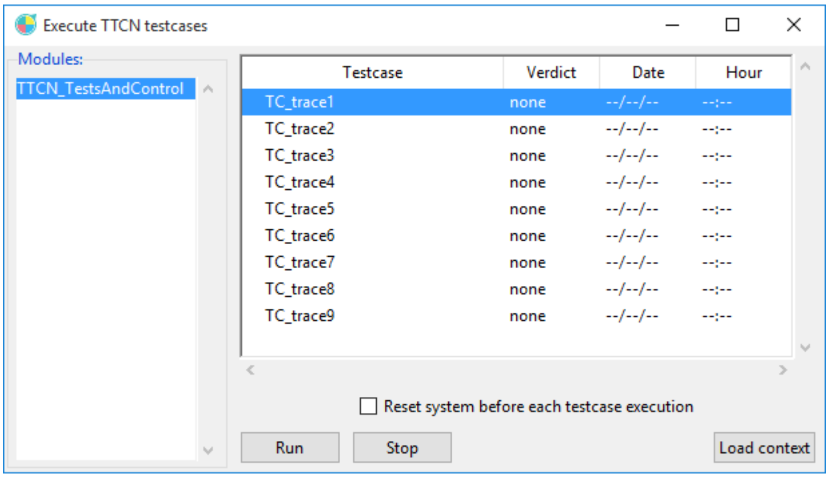
Pid	Receiver	Signal
1	TTCN_TestsAndControl	RTDS_startMessage
2	pCentral	RTDS_startMessage
3	pLocal	RTDS_startMessage

```
PragmaDev Studio shell
>Internal communication socket opened on port 49154
>Free run off.
>Timers will be fired automatically.
>Unexpected messages will generate a warning.
>
```

Debugger state: STOPPED
Active thread:

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.



Testcase	Verdict	Date	Hour
TC_trace1	none	--/--/--	--:--
TC_trace2	none	--/--/--	--:--
TC_trace3	none	--/--/--	--:--
TC_trace4	none	--/--/--	--:--
TC_trace5	none	--/--/--	--:--
TC_trace6	none	--/--/--	--:--
TC_trace7	none	--/--/--	--:--
TC_trace8	none	--/--/--	--:--
TC_trace9	none	--/--/--	--:--

☐ Reset system before each testcase execution

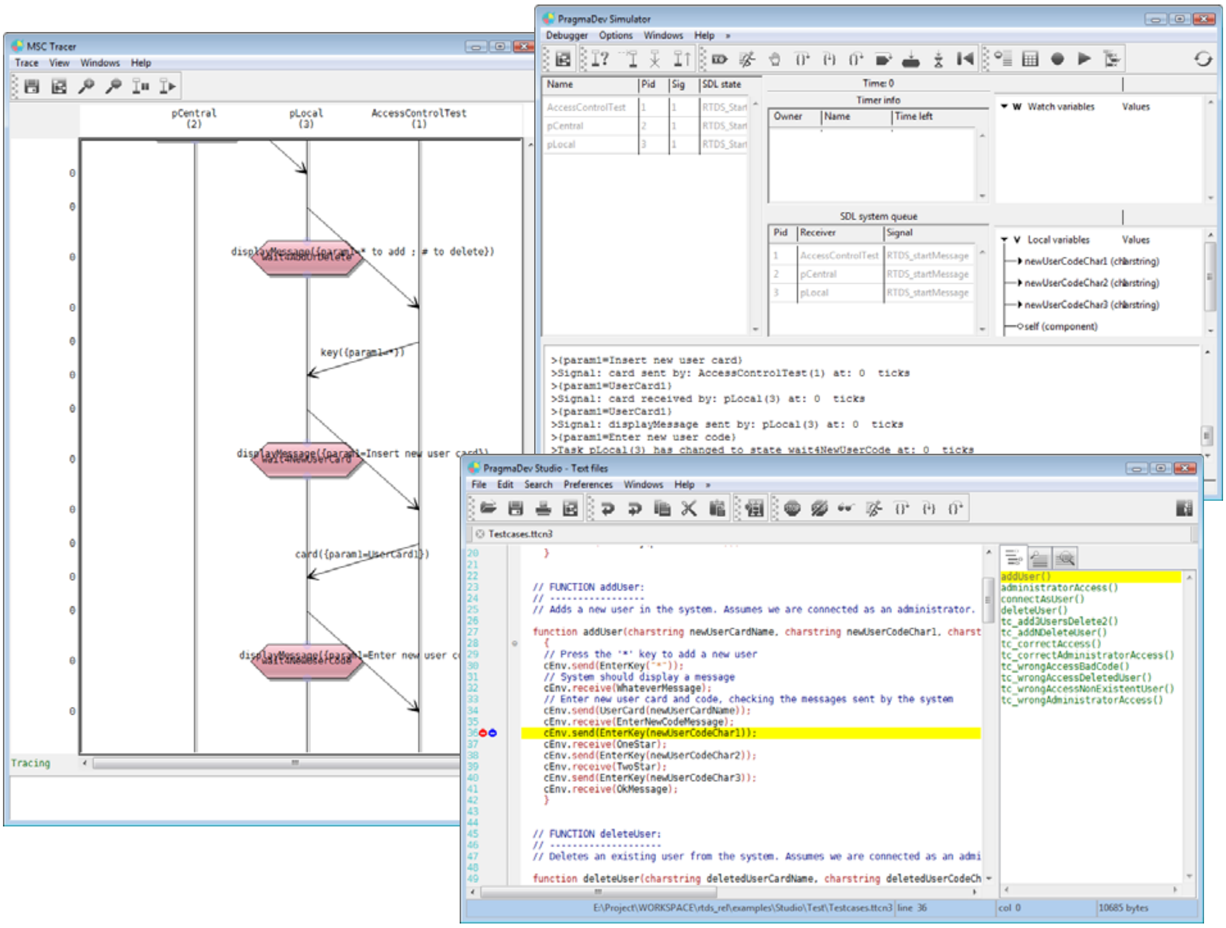
This window will show all available testcases in the module selected for the simulation and all its imported modules. Also if the selected module has a control part it will be shown.

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:



Name	Pid	Sig	SDL state
AccessControlTest	1	1	RTDS_Start
pCentral	2	1	RTDS_Start
pLocal	3	1	RTDS_Start

Owner	Name	Time left

Watch variables	Values

Pid	Receiver	Signal
1	AccessControlTest	RTDS_startMessage
2	pCentral	RTDS_startMessage
3	pLocal	RTDS_startMessage

Local variables	Values
newUserCodeChar1 (charstring)	
newUserCodeChar2 (charstring)	
newUserCodeChar3 (charstring)	
self (component)	

```
>{param1=Insert new user card}
>Signal: card sent by: AccessControlTest(1) at: 0  ticks
>{param1=UserCard1}
>Signal: card received by: pLocal(3) at: 0  ticks
>{param1=UserCard1}
>Signal: displayMessage sent by: pLocal(3) at: 0  ticks
>{param1=Enter new user code}
>Task pLocal(3) has changed to state wait4NewUserCode at: 0  ticks
```

```
  }


// FUNCTION addUser:
// -----------------
// Adds a new user in the system. Assumes we are connected as an administrator.

function addUser(charstring newUserCardName, charstring newUserCodeChar1, charst
  {
  // Press the '*' key to add a new user
  cEnv.send(EnterKey("*"));
  // System should display a message
  cEnv.receive(WhateverMessage);
  // Enter new user card and code, checking the messages sent by the system
  cEnv.send(UserCard(newUserCardName));
  cEnv.receive(EnterNewCodeMessage);
  cEnv.send(EnterKey(newUserCodeChar1));
  cEnv.receive(OneStar);
  cEnv.send(EnterKey(newUserCodeChar2));
  cEnv.receive(TwoStar);
  cEnv.send(EnterKey(newUserCodeChar3));
  cEnv.receive(OkMessage);
  }


// FUNCTION deleteUser:
// --------------------
// Deletes an existing user from the system. Assumes we are connected as an admi

function deleteUser(charstring deletedUserCardName, charstring deletedUserCodeCh
```

```
addUser()
administratorAccess()
connectAsUser()
deleteUser()
tc_add3UsersDelete2()
tc_addNDeleteUser()
tc_correctAccess()
tc_correctAdministratorAccess()
tc_wrongAccessBadCode()
tc_wrongAccessDeletedUser()
tc_wrongAccessNonExistentUser()
tc_wrongAdministratorAccess()
```

E:\Project\WORKSPACE\rtds_ref\examples\Studio\Test\Testcases.ttcn3 | line 36 | col 0 | 10685 bytes

As TTCN test cases do not have states, state change symbols in MSC traces are used to display test case verdicts.

5.5.4 Provided external functions

A number of built-in functions are available in PragmaDev Simulator. To have access to these functions the PragmaLib ttcn module must be imported:

```
import from PragmaLib all;
```

5.5.4.1 Formatted output

The following functions are available for formatted output:

- `PragmaDev_b4sprintf(<arg>) : <sprintf arg>`
  - <arg>: boolean.
  - <sprintf arg>: PragmaDev_arg4sprintf. Built-in wrapper type for <arg>.

  This function is declared as:
  ```
  external function PragmaDev_b4sprintf(boolean boolean_arg)
  return PragmaDev_arg4sprintf;
  ```
- `PragmaDev_i4sprintf(<arg>) : <sprintf arg>`
  - <arg>: integer.
  - <sprintf arg>: PragmaDev_arg4sprintf. Built-in wrapper type for <arg>.

  This function is declared as:
  ```
  external function PragmaDev_i4sprintf(integer integer_arg)
  return PragmaDev_arg4sprintf;
  ```
- `PragmaDev_f4sprintf(<arg>) : <sprintf arg>`
  - <arg>: real.
  - <sprintf arg>: PragmaDev_arg4sprintf. Built-in wrapper type for <arg>.

  This function is declared as:
  ```
  external function PragmaDev_f4sprintf(float float_arg) return
  PragmaDev_arg4sprintf
  ```
- `PragmaDev_s4sprintf(<arg>) : <sprintf arg>`
  - <arg>: charstring.
  - <sprintf arg>: PragmaDev_arg4sprintf. Built-in wrapper type for <arg>.

  This function is declared as:
  ```
  external function PragmaDev_s4sprintf(charstring charstring_arg)
  return PragmaDev_arg4sprintf;
  ```
- `PragmaDev_sprintf(<format>, <args>) : <formatted string>`
  - <format>: charstring. Based on C sprintf format specifiers (%d, %f, %s, ...).
  - <args>: PragmaDev_arg4sprintf. Built-in wrapper type for boolean, integer, real, and charstring. This should be a concatenation of the functions described above (PragmaDev_<specifier>4sprintf where <specifier> is either b, i, f, or s).
  - <formatted string>: charstring.

  This function is declared as:
  ```
  external function PragmaDev_sprintf(charstring format,
  PragmaDev_arg4sprintf args) return charstring;
  ```
  Usage example:
  ```
  charstring output := PragmaDev_sprintf("This is %s %d",
  PragmaDev_s4sprintf("number") & PragmaDev_i4sprintf(42));
  ```

5.5.4.2 File manipulation

The following functions are available for file manipulation:

- `PragmaDev_FileOpen (<file name>, <open mode>) : <file id.>`
  - <file id.>: integer.
  - <file name>: charstring. Path is relative to the project.

  - <open mode>: charstring. Based on C fopen file manipulation modes (‘w’, ‘r’, ‘a’, ‘r+’,’a+’...)

  This function is declared as:
  `external function PragmaDev_FileOpen(charstring file_name, charstring file_mode) return integer;`
- `PragmaDev_FileClose(<file id.>) : <success>`
  - <success>: boolean.
  - <file id.>: integer. Value given by the PragmaDev_FileOpen.

  This function is declared as:
  `external function PragmaDev_FileClose(integer file_id) return boolean;`
- `PragmaDev_FileReadLine(<file id.>) : <read line>`
  - <read line>: charstring. The line read in the file. Always includes at least the ending newline character. If line is empty, it means the end of the file has been reached.
  - <file id.>: integer. Value given by the PragmaDev_FileOpen.

  This function is declared as:
  `external function PragmaDev_FileReadLine(integer file_id) return charstring;`
- `PragmaDev_FileWriteLine(<line to write>, <file id.>) : <success>`
  - <success>: boolean.
  - <line to write>: charstring.
  - <file id.>: integer. Value given by the PragmaDev_FileOpen.

  This function is declared as:
  `external function PragmaDev_FileWriteLine(charstring string_to_write, integer file_id) return boolean;`

5.5.4.3 Radar graph

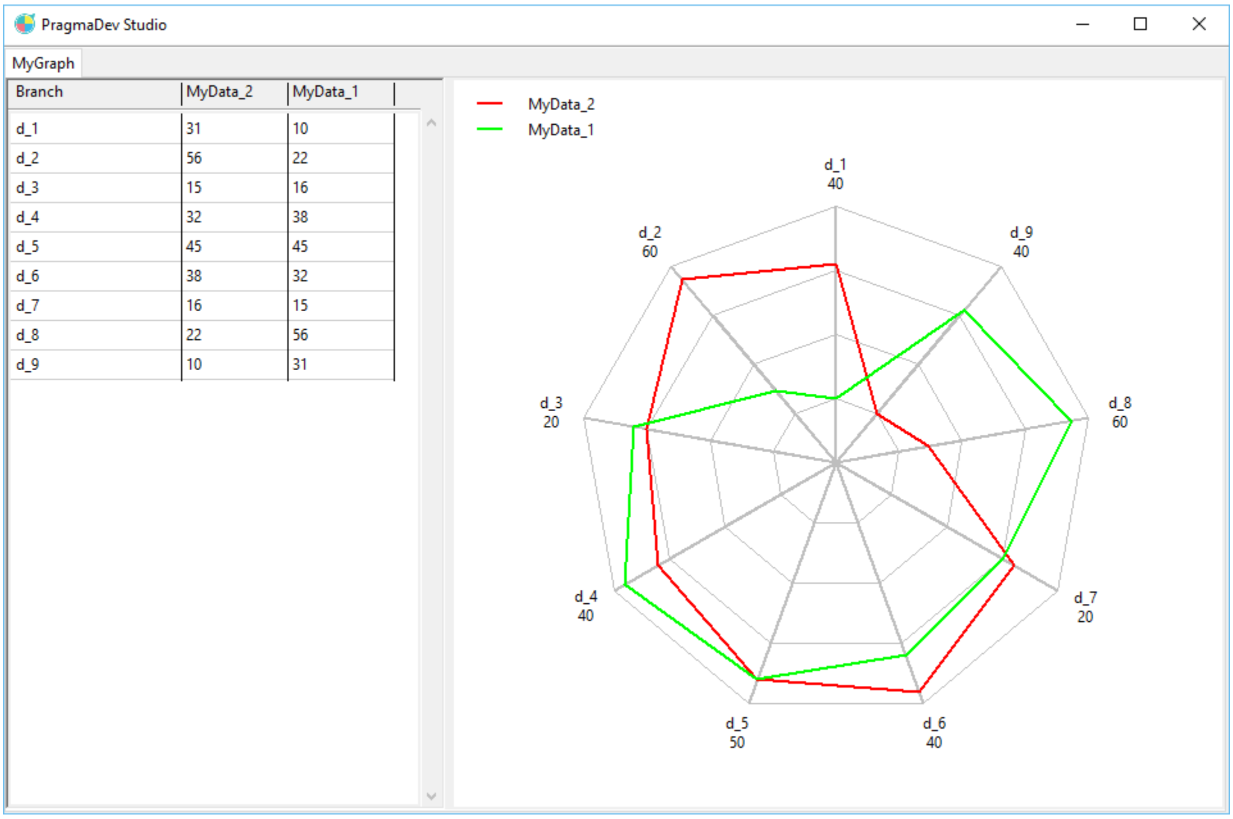
The following functions are available to generate radar graphs. Several graphs can be generated at the same time. The resulting window will organize them in tabs. The scale on the branches is automatically adjusted.

- `PragmaDev_RadarGraphCreate(<graph name>, <branch labels>) : <graph id.>`
  - <graph id.>: integer.
  - <graph name>: charsting. The name will be displayed on the tab in the window.
  - <branch labels>: charstring. Semi-column separated list of branch names.

  This function is declared as:
  `external function PragmaDev_RadarGraphCreate(charstring branch_labels) return integer;`
- `PragmaDev_RadarGraphAddLine(<graph id.>, <line label>, <line values>) : <status>`
  - <status>: boolean.
  - <graph id.>: integer. Value given by the PragmaDev_RadarGraphCreate.
  - <line label>: charstring. Label of the line in the graph.
  - <line values>: charstring. Semin-column separated list of values for each branch.

This function is declared as:

```
external function PragmaDev_RadarGraphAddLine(integer graph_id,
charstring line_label, charstring line_branch_values) return
boolean;
```



Branch	MyData_2	MyData_1
d_1	31	10
d_2	56	22
d_3	15	16
d_4	32	38
d_5	45	45
d_6	38	32
d_7	16	15
d_8	22	56
d_9	10	31

Radar graph example

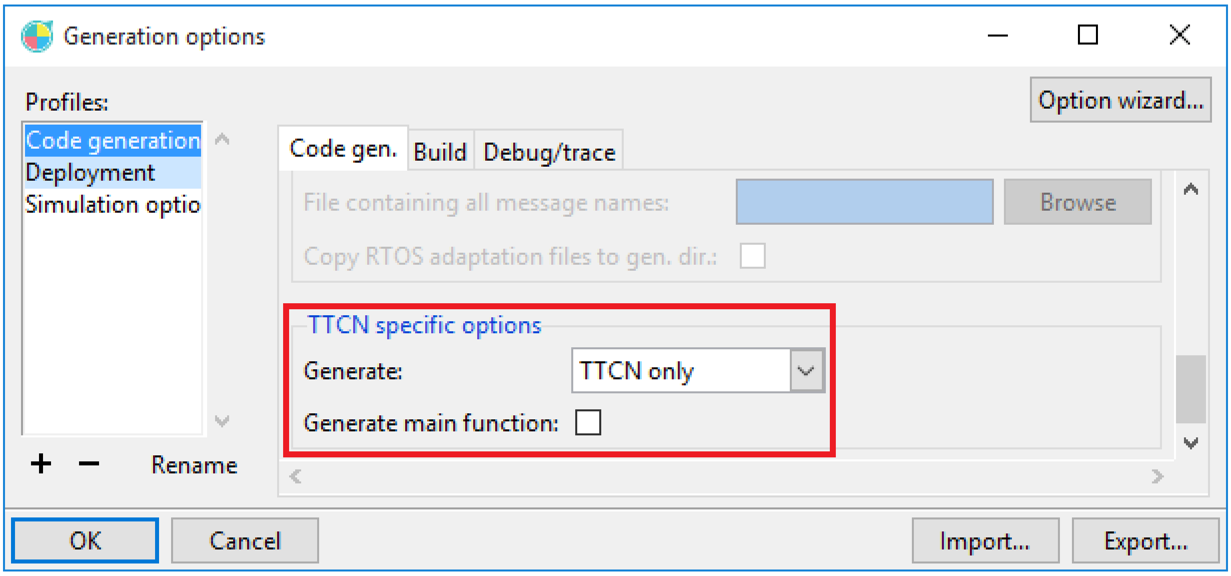
5.6 - C++ code generation

TTCN-3 modules can be generated as C++ code. It is possible to generate only the C++ code for TTCN-3 modules, or to generate the modules along 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.

5.6.1 Stand alone

To generate TTCN-3 only, in the generation options select *TTCN only* in the Generate of the *TTCN specifics options*:



File containing all message names:

Copy RTOS adaptation files to gen. dir.: □

TTCN specific options

Generate: TTCN only

Generate main function: □

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.

5.6.2 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.

5.6.3 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:

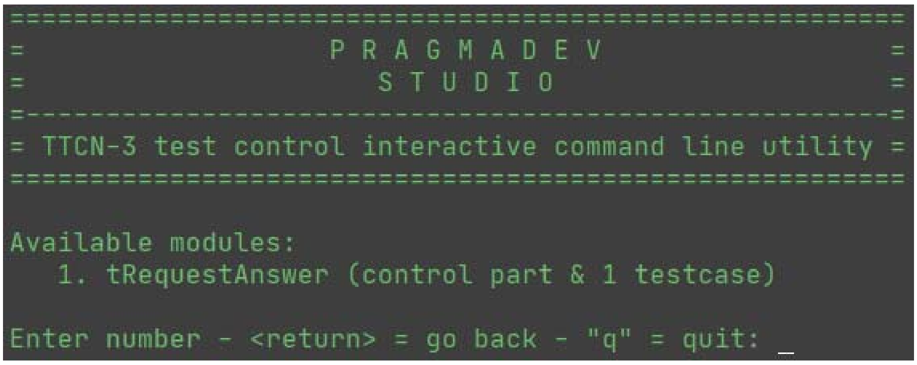
```
import from ASN1FileName language “ASN.1:2002” all;
```

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.

5.6.4 Generate the main function

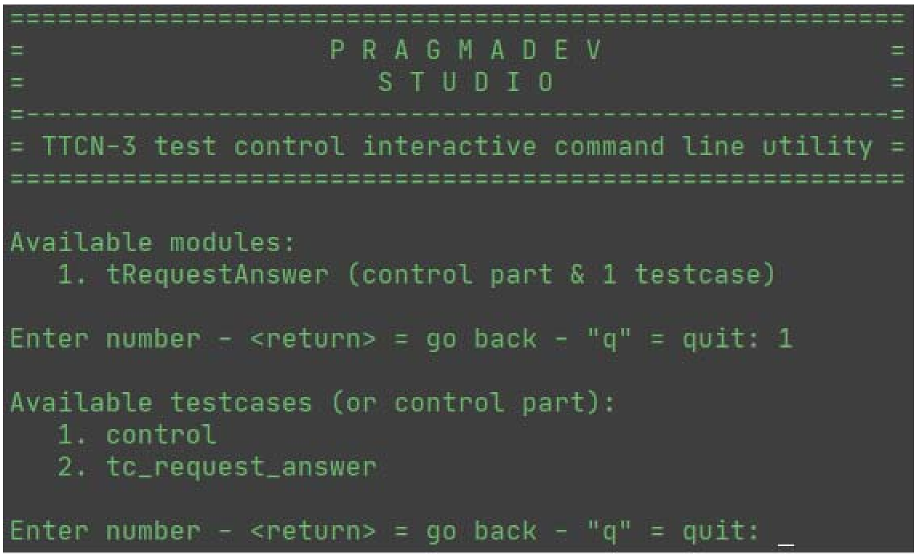
PragmaDev Studio allows to automatically generate a main function in RTDS_TTCN_main.c. To do so, the option *Generate main function* must be checked in the generation options.

The generated main function allows to run the test suite interactively, either the standard way by running the control part, or testcase by testcase. If the generated executable is launched from a console or terminal, the following choices will appear:



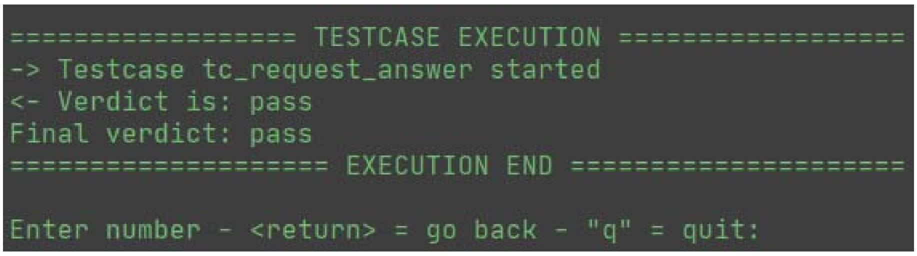
```
========================================================
=                   P R A G M A D E V                  =
=                      S T U D I O                     =
=------------------------------------------------------=
= TTCN-3 test control interactive command line utility =
========================================================

Available modules:
   1. tRequestAnswer (control part & 1 testcase)

Enter number - <return> = go back - "q" = quit: _
```

This first level of menu allows to select the parent module for the elements to execute. Only modules containing testcases or a control part are shown. Once the module is selected, a second level of choice appears:



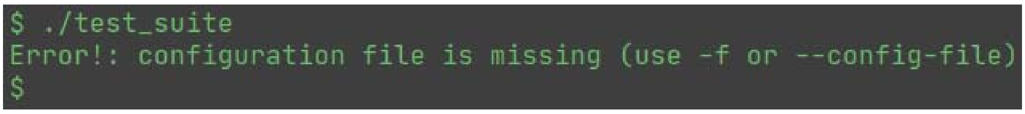
```
========================================================
=                   P R A G M A D E V                  =
=                      S T U D I O                     =
=------------------------------------------------------=
= TTCN-3 test control interactive command line utility =
========================================================

Available modules:
   1. tRequestAnswer (control part & 1 testcase)

Enter number - <return> = go back - "q" = quit: 1

Available testcases (or control part):
   1. control
   2. tc_request_answer

Enter number - <return> = go back - "q" = quit: _
```

Here, the executable elements are listed. If for example the testcase is run by typing 2, then return, it will be executed and its verdict displayed:



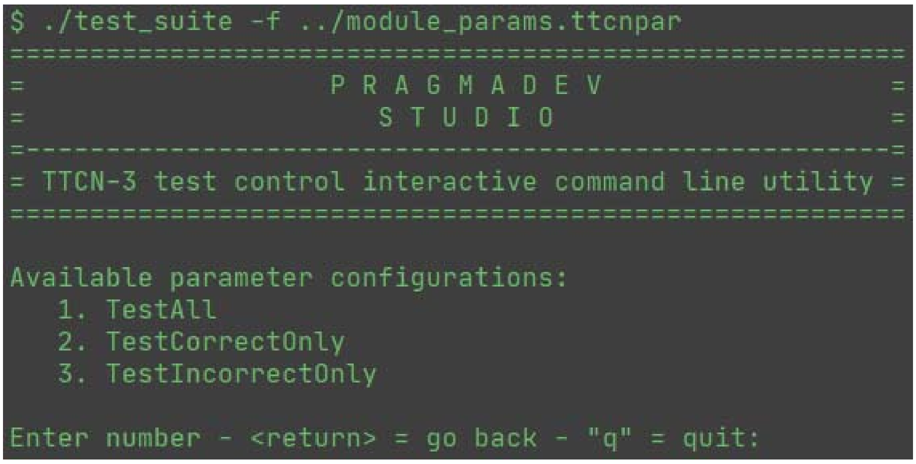
```
================== TESTCASE EXECUTION ==================
-> Testcase tc_request_answer started
<- Verdict is: pass
Final verdict: pass
===================== EXECUTION END =====================

Enter number - <return> = go back - "q" = quit:
```

Note that if a TTCN test suite requires a set of module parameters or testcases parameters to work, the generated executable won't run directly:



```
$ ./test_suite
Error!: configuration file is missing (use -f or --config-file)
$
```

The required configuration file is the one containing the values for module & testcase parameters, as described in “TTCN-3 parameters editor” on page 341. If such a file is specified via the `-f` option, the executable runs and displays a first level of choices allowing to select one of the parameter configurations stored in the file:



```
$ ./test_suite -f ../module_params.ttcnpar
========================================================
=                   P R A G M A D E V                  =
=                      S T U D I O                     =
=------------------------------------------------------=
= TTCN-3 test control interactive command line utility =
========================================================

Available parameter configurations:
   1. TestAll
   2. TestCorrectOnly
   3. TestIncorrectOnly

Enter number - <return> = go back - "q" = quit:
```

Once the configuration has been selected, the values for module & testcase parameters are defined and the execution proceeds as described above.

For details on how the main function is actually generated, see the PragmaDev Studio Reference Manual.

5.6.5 RTOS integration

Only generation for windows *win32* and *posix* are supported.

5.6.6 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.

5.6.7 Debug information

To easily debug testcases execution, it is possible to ask testcase to write relevant information in a log file. To do so, the RTDS_TTCN_DEBUG_FILE macro must be defined in the compilation options using the -D option, with the name of the file in which all debug information will be write. For example :

```
-DRTDS_TTCN_DEBUG_FILE='"myDebugFile"'
```

will make the testcases write debug information in the file myDebugFile.

For more details on printed information in the file, please refer to section 15.8 of the PragmaDev Studio Reference Manual.

5.6.8 External functions

The following external functions (described in “Provided external functions” on page 348) are already available for code generation:

- PragmaDev_b4sprintf
- PragmaDev_i4sprintf
- PragmaDev_f4sprintf
- PragmaDev_s4sprintf
- PragmaDev_sprintf
- PragmaDev_FileOpen
- PragmaDev_FileClose
- PragmaDev_FileReadLine
- PragmaDev_FileWriteLine

The implementation of these functions can be found in:

```
$(RTDS_HOME)/share/lib/ExternalProcedures.h
$(RTDS_HOME)/lib/ExternalProcedures.c
```

If one of these functions is used in the model, the header file is automatically included in the generated headers, and the source file is automatically included in the generated makefile.

5.7 - TTCN-3 automatic generation

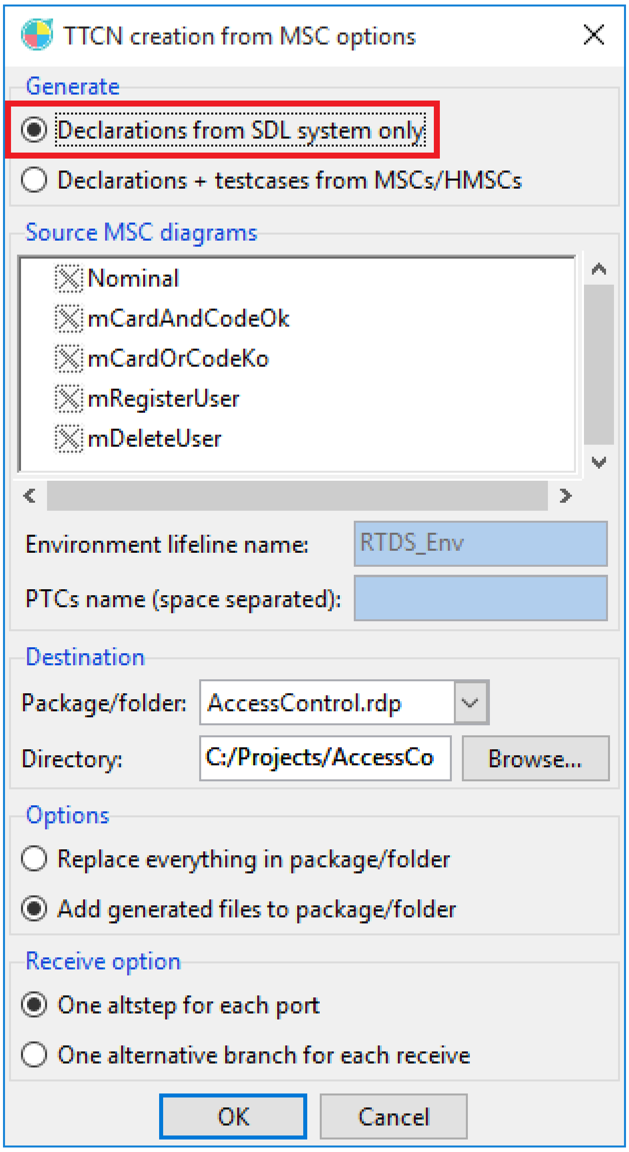
PragmaDev Studio allows to automatically generate TTCN-3 from an SDL/SDL-RT system or from MSCs and/or HMSCs.

5.7.1 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*:



TTCN creation from MSC options

Generate

● Declarations from SDL system only

○ Declarations + testcases from MSCs/HMSCs

Source MSC diagrams

☒ Nominal
☒ mCardAndCodeOk
☒ mCardOrCodeKo
☒ mRegisterUser
☒ mDeleteUser

Environment lifeline name: RTDS_Env

PTCs name (space separated):

Destination

Package/folder: AccessControl.rdp

Directory: C:/Projects/AccessCo

Options

○ Replace everything in package/folder

● Add generated files to package/folder

Receive option

● One altstep for each port

○ One alternative branch for each receive

One file will be generated:

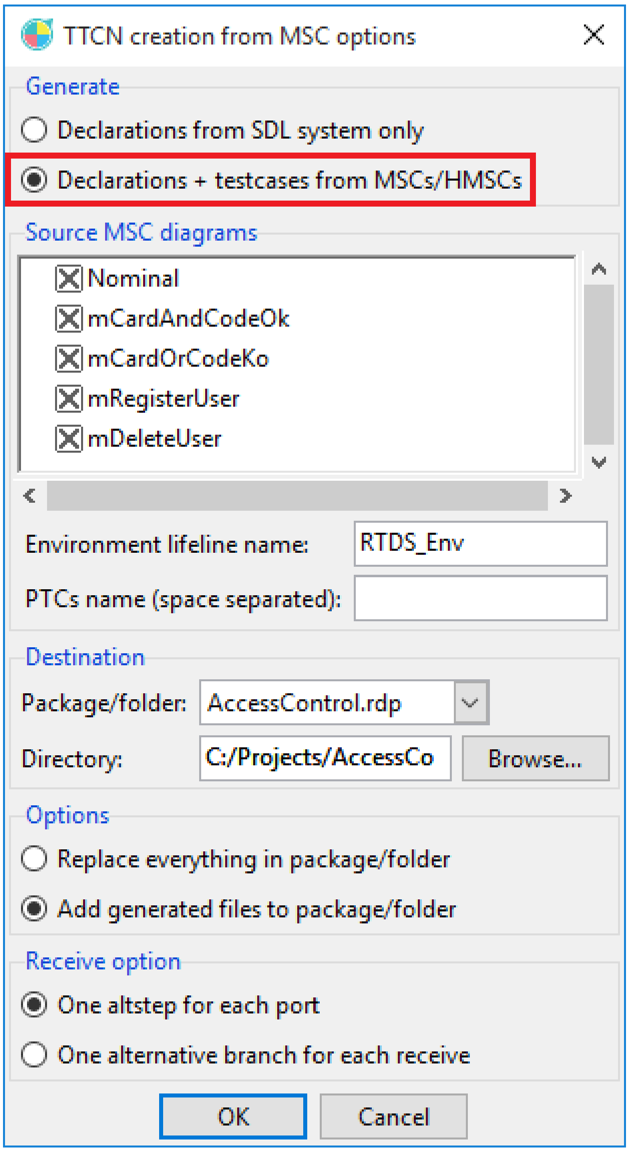
- `TTCN_Declarations.ttcn`
  - declaration of the necessary data types for message parameters,
  - declaration of the record types for all SDL messages,
  - declaration of ports (one for each SDL/SDL-RT channel connected to the environment),
  - declaration of TSI component type.

This file is overwritten for each TTCN-3 generation.

5.7.2 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*:



TTCN creation from MSC options

Generate

○ Declarations from SDL system only

● Declarations + testcases from MSCs/HMSCs

Source MSC diagrams

☒ Nominal
☒ mCardAndCodeOk
☒ mCardOrCodeKo
☒ mRegisterUser
☒ mDeleteUser

Environment lifeline name: RTDS_Env

PTCs name (space separated):

Destination

Package/folder: AccessControl.rdp

Directory: C:/Projects/AccessCo

Options

○ Replace everything in package/folder

● Add generated files to package/folder

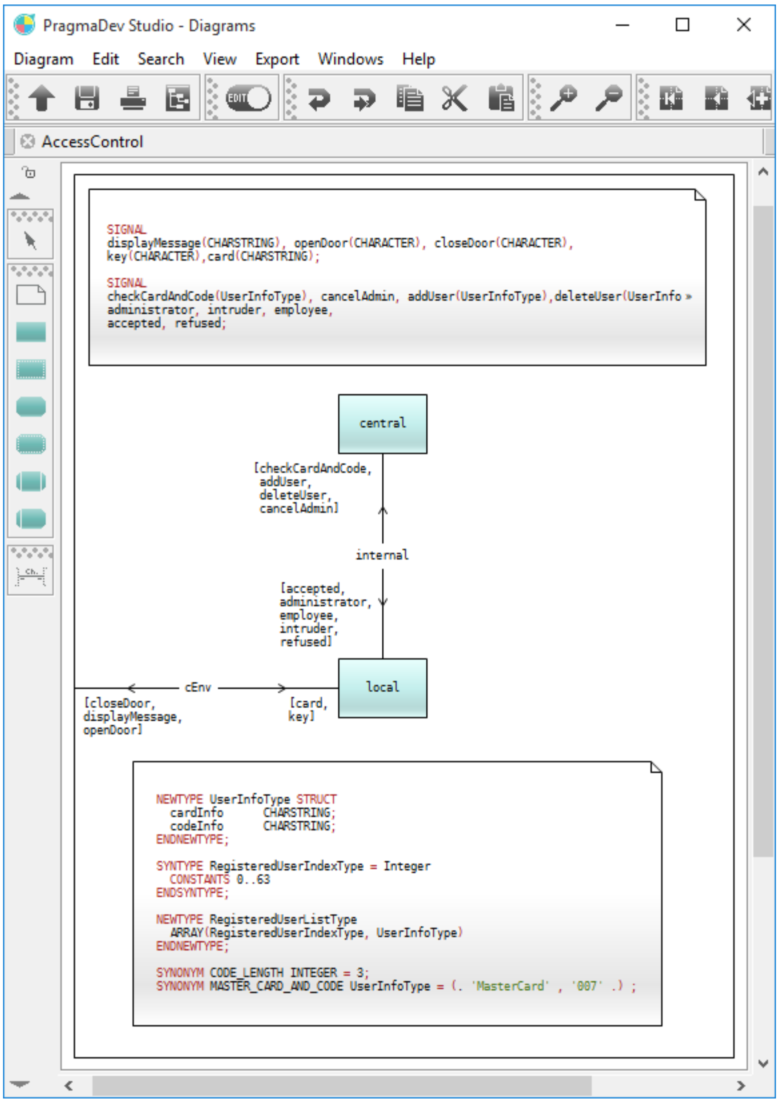
Receive option

● One altstep for each port

○ One alternative branch for each receive

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.

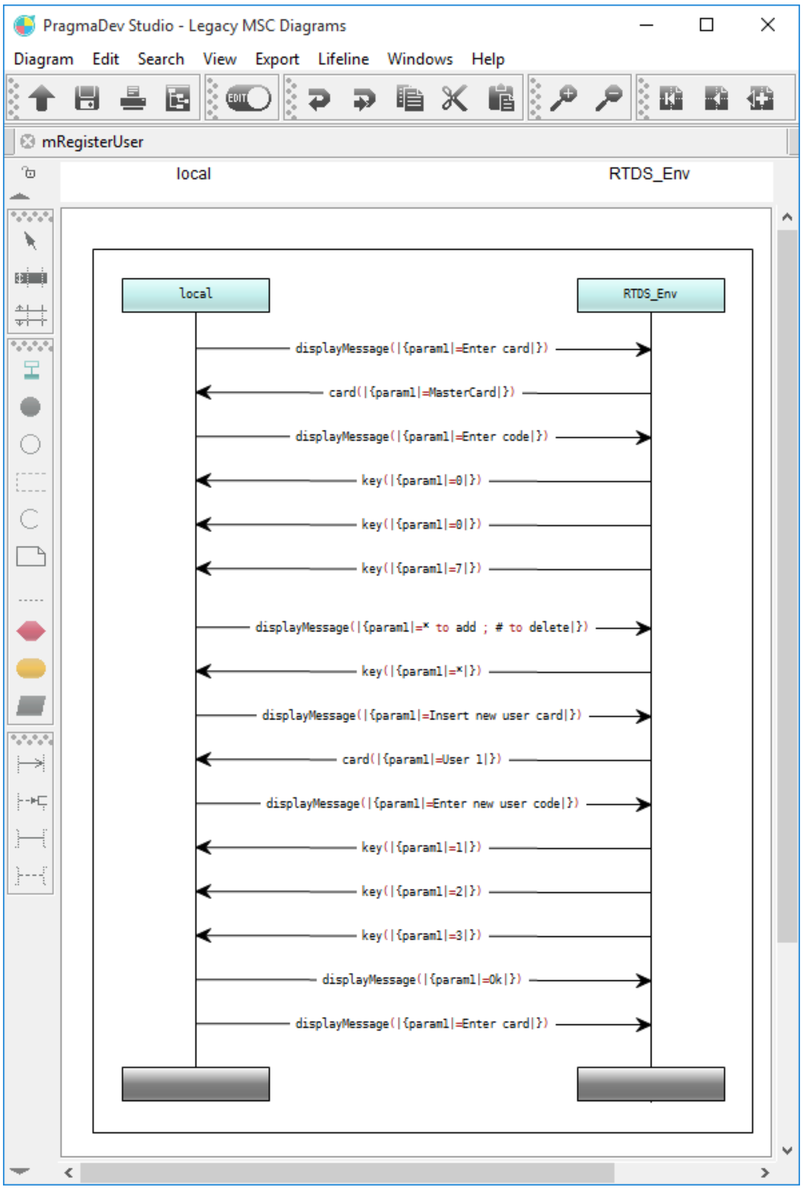
For example, for the current system:



```
SIGNAL
displayMessage(CHARSTRING), openDoor(CHARACTER), closeDoor(CHARACTER),
key(CHARACTER),card(CHARSTRING);

SIGNAL
checkCardAndCode(UserInfoType), cancelAdmin, addUser(UserInfoType),deleteUser(UserInfo »
administrator, intruder, employee,
accepted, refused;
```

```
NEWTYPE UserInfoType STRUCT
  cardInfo      CHARSTRING;
  codeInfo      CHARSTRING;
ENDNEWTYPE;

SYNTYPE RegisteredUserIndexType = Integer
  CONSTANTS 0..63
ENDSYNTYPE;

NEWTYPE RegisteredUserListType
  ARRAY(RegisteredUserIndexType, UserInfoType)
ENDNEWTYPE;

SYNONYM CODE_LENGTH INTEGER = 3;
SYNONYM MASTER_CARD_AND_CODE UserInfoType = (. 'MasterCard' , '007' .) ;
```

with the current MSC:


PragmaDev Studio - Legacy MSC Diagrams
Diagram Edit Search View Export Lifeline Windows Help
mRegisterUser
local
RTDS_Env
displayMessage(|{param1|=Enter card|})
card(|{param1|=MasterCard|})
displayMessage(|{param1|=Enter code|})
key(|{param1|=0|})
key(|{param1|=0|})
key(|{param1|=7|})
displayMessage(|{param1|=* to add ; # to delete|})
key(|{param1|=*|})
displayMessage(|{param1|=Insert new user card|})
card(|{param1|=User 1|})
displayMessage(|{param1|=Enter new user code|})
key(|{param1|=1|})
key(|{param1|=2|})
key(|{param1|=3|})
displayMessage(|{param1|=Ok|})
displayMessage(|{param1|=Enter card|})

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.

5.7.3 From a complete SDL system via model checking technology

It is possible to automatically generate TTCN out of an SDL system. For that purpose PragmaDev Studio relies on third party model checking technologies. The basic process is to:

- Export the SDL system to a pivot format (IF, Fiacre, xLIA);
- Define test objectives (coverage, property, observers);
- Call the third party tool;
- Analyze results to build up scenarios.

For more information please refer to “Checking a system against MSC scenarios” on page 435.

6 - PragmaDev Tracer

PragmaDev Tracer is one of *PragmaDev Studio* modules. *PragmaDev Tracer* is free of charge. Because it is a free tool, it is the only one that has no size restriction on its diagram in the free version of *PragmaDev Studio*.

6.1 - Overview

PragmaDev Tracer allows to generate execution traces from actual applications running on targets. Traces can be created either online, from the actual behavior being executed, or offline, using a intermediate format obtained from the running application and analyzed afterwards by the tracer. The traces are displayed using the standard ITU-T MSC representation, or the SDL-RT MSC representation (see the SDL-RT website http://www.sdl-rt.org).

Online traces are created directly by the program running on the target which sends textual commands to the Tracer via a standard socket. Two modes are available:

- The graphical mode, allowing to display all the traces as they are created;
- The command-line mode, allowing to generate trace diagrams without visual user feedback. This is the ideal mode to generate traces in batch operations.

Other features of PragmaDev Studio may be used in conjunction with the tracer itself to get a full-featured tracing utility with conformance checking against specification or property verification:

- Trace diagrams can be directly created for documentation purposes;
- Trace diagrams can be compared in a visual way;
- Specification MSC diagrams can be created, then can be compared to execution traces for conformance checking;
- Property diagrams can be created using the *Property Sequence Chart* (*PSC*) format, which can be matched against execution traces;
- All diagrams can be easily documented, for example by exporting them fully or partially to common image formats, allowing to insert them in a document.

The general graphical form of the diagrams supported by *PragmaDev Tracer* is the same as the one supported in PragmaDev Studio’s MSC Editor, and is described in section “MSC & PSC reference guide” on page 363.

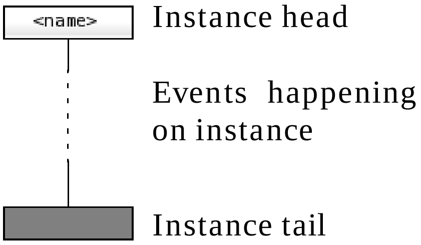
The general usage of the Tracer is detailed in “Usage” on page 381, including how to launch it, the options it accepts, a description of the tracer windows and the relevant preferences. The available commands that can be sent through the socket are described in section “Command reference” on page 387.

The tracing feature is described in “Tracer window” on page 383. The MSC and PSC editor is described in “MSC editor” on page 83, and the documentation features in “Documentation generation” on page 125. The available checks that can be performed - trace against trace, specification against trace or property matches - are described in “Conformance checking: diagram diff & property match” on page 97.

6.2 - MSC & PSC reference guide

6.2.1 General diagram format

A MSC or PSC diagram represents the interaction going on between entities called *instances* over time. Instances will typically be tasks, processes or objects. Instances are represented by symbols called *lifelines*, that look like follows:



The lifeline always starts with a head that specifies the instance name.

All events happening on the instance are then displayed on a vertical line under the lifeline head. These events are described below in “Lifeline components” on page 366.

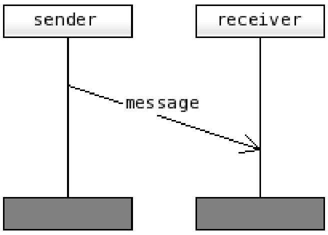
The lifeline terminates by a lifeline tail, that can take several forms depending on the status of the instance at the end of the diagram.

Lifelines are distributed along the horizontal axis, and the vertical axis represents the time, flowing from top to bottom. Events happening between lifelines are mostly represented by *links*, described in “Links” on page 363. Other symbols allow to further describe the diagram or add semantics to specification MSCs or PSCs; they are described in “Main symbols” on page 374.

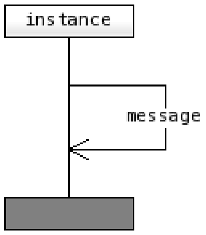
6.2.2 Links

6.2.2.1 Message links

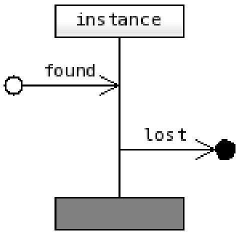
An asynchronous message sent by an instance and received by another is represented by a line with an outlined arrow at its end:



An instance can also send a message to itself:



A message can also be received from an unknown source, or sent to an unknown target. In this case, they are called a found message and a lost message respectively:



The syntax for the message link text is free, but *PragmaDev Tracer* expects a specific format for some features:

- Conformance checking by diagram comparison or property matches has an option to consider message parameters or not, as explained in “Basic MSC diff: trace vs. trace, spec. vs. spec., ...” on page 363. In this case, the message link text should have the format:
  `<message name> ( <message parameters> )`
  The `<message name>` will be everything up to the first ‘`(`’ in the text, and the parameters everything between this ‘`(`’ and the final ‘`)`’. If anything else than a space appears after the last ‘`)`’, the syntax won’t be recognized and ignoring message parameters will have no effect.
- The structured message parameters display also expects a specific format for the whole message link text. See “Message parameter format” on page 391 for more details.

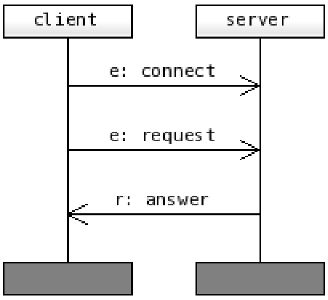
6.2.2.2 PSC-specific normal, required and failed message syntax

In PSC diagrams, texts for message links are supposed to be prefixed with one of the following:

- ‘`e:`’ indicates the message is a regular one. This means that the message is part of the precondition for the property: all messages prefixed with ‘`e:`’ must appear to trigger a property match. If any of these messages do not appear in the checked diagram, the preconditions for the property aren’t satisfied, and no matching is attempted. Regular messages should appear first in the PSC diagram.
- ‘`r:`’ indicates a required message. This means that if all the regular messages preceding this message are present in the checked diagram, this message must be present for the property to match. If it doesn’t, the property is violated.

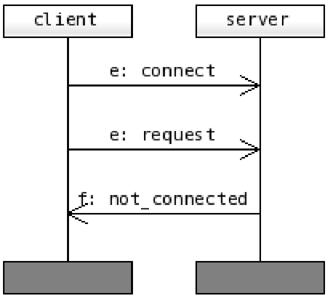
- 'f:' indicates a fail message. This means that if all the regular messages preceding this message are present in the checked diagram, this message must not appear for the property to match. If it does appear, the property is violated.

Here is an example of a required message in a property:



This means that if the client has sent a connect message to the server, then sends a request message, the server *must* send back an answer message, or the property is violated.

Here is an example with a fail message:

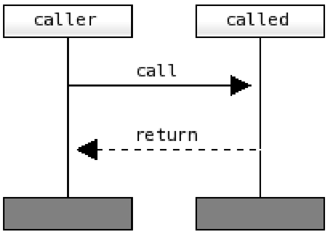


This means that if the client has sent a connect message to the server, then sends a request message, the server *must not* send back a not_connected message, or the property is violated.

Note that in PragmaDev Studio, PSC wanted or unwanted constraints will also appear in the message link text. For more details, see "PSC constraints: wanted and unwanted messages & chains" on page 371.

6.2.2.3 Operation call and return links

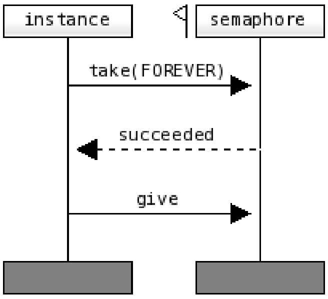
A synchronous call from an instance to another one is represented by a solid horizontal line with a block arrow at its end. The return of the call is represented by a dashed horizontal line, also with a block arrow at its end:



The syntax for these links is free, but PragmaDev Tracer will expect a specific syntax for the operation call link for some features. These features and the expected syntax are the same as for messages, as described in “Message links” on page 363.

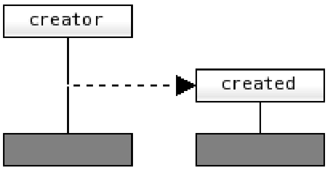
6.2.2.4 Semaphore take, take results and give

A specific kind of lifeline can be used to represent a semaphore in *PragmaDev Tracer* (see “Semaphore lifeline” on page 374). For these lifelines, the standard `take` and `give` operations are available, and are displayed like normal operation calls. The results of the `take` (success, time-out) is displayed as an operation return link:



6.2.2.5 Dynamic instance creation links

All lifelines are not necessarily present at the start of a MSC diagram, some of these can be dynamically created later. A dynamic creation is always done by another instance, and a dashed line with a block arrow to the head of the created instance is used to represent this creation:



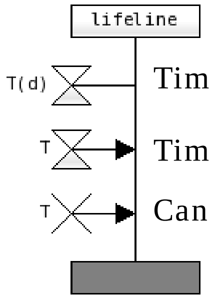
Note: in *PragmaDev Tracer*, this is the only way to have an instance starting elsewhere than at the top of the diagram.

6.2.3 Lifeline components

Lifeline components are events impacting a single lifeline. They appear as symbols attached to the lifeline.

6.2.3.1 Timer events

An instance can start timers, that will time-out in a given amount of time. A timer can also be cancelled by the instance that created it. The symbols for timers are the following ones:



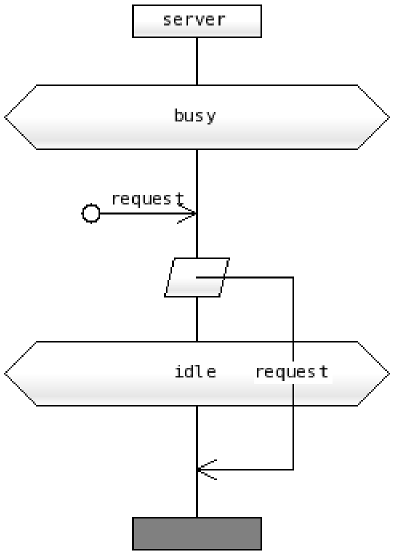
Timer start - T is the timer name, d its duration

Time-out for timer named T

Cancelling of timer named T

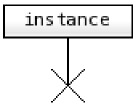
6.2.3.2 Message save

A message received by an instance can be saved to be treated later, for example after a state change. The message will be displayed as resent from the instance lifeline to itself when it is actually treated. For example:



6.2.3.3 Stop symbol

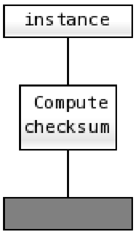
A stop symbol indicates that an instance has killed itself. Note that in standard MSCs, an instance can only kill itself, there is no notion of one instance killing another one. The stop symbol is displayed as follows:



and is always the last symbol appearing on a lifeline.

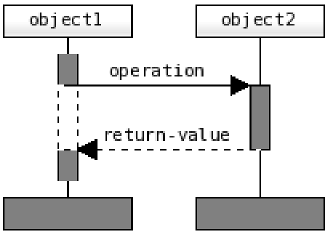
6.2.3.4 Action symbol

Action symbols describe actions performed by the lifeline. In the current version of *PragmaDev Tracer*, this description is informal. For example:



6.2.3.5 Method and suspended segments

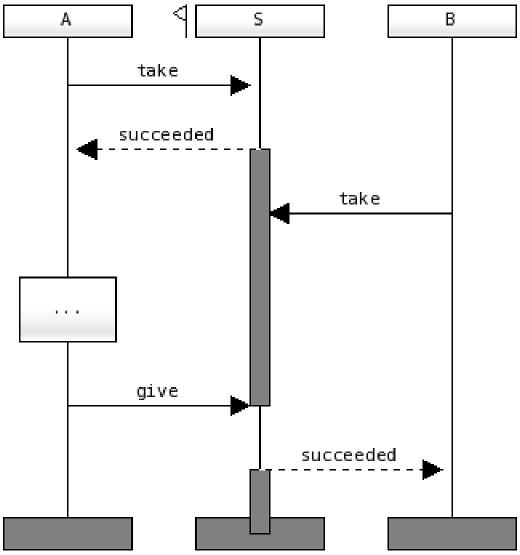
Segments are mostly used for instances describing objects that do not have parallel flows of execution. In this case, an object calling an operation on another one will be inactive (suspended) during the call while the other object actually executes the operation, or method. The control will be given back to the caller object when the operation returns. This can be shown in the MSC using method and suspended segments:



The instance `object1` is executing something (method segment) when it calls `operation` on `object2`. Then `object1` becomes inactive (suspended segment) while `object2` executes the operation (method segment), and `object1` becomes active again (method segment) when the operation returns.

6.2.3.6 Semaphore unavailability

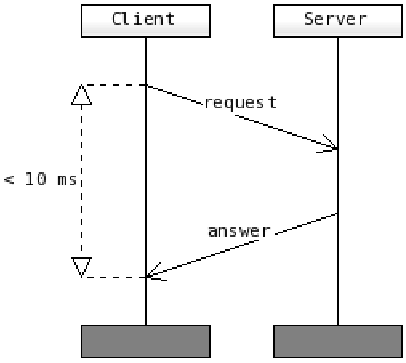
When a semaphore is taken by an instance and becomes unavailable, the same graphical representation is used on the semaphore lifeline as the method segment on an object lifeline. For example:



When the take from A to S succeeds, the semaphore becomes unavailable. When B attempts a take, it is put on hold until A gives the semaphore back. Then the take for B succeeds and the semaphore becomes unavailable again.

6.2.3.7 Relative time constraints

A relative time constraint appearing in a specification MSC diagram or a PSC diagram indicates that the sequence of events it encloses must happen within a given time. For example:

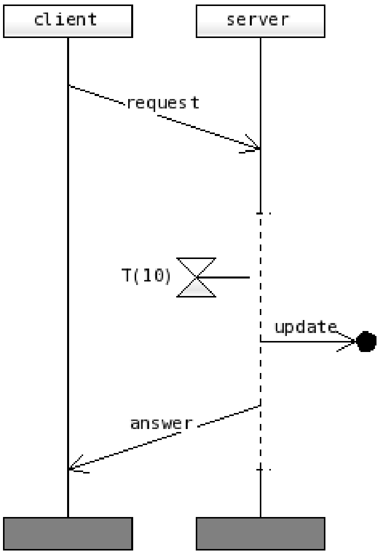


This specifies there must be less than 10 ms between the time when Client sends the request message and the time when it receives the answer message.

During conformance checking, relative time constraints are compared to absolute times in the compared diagram (see “Conformance checking: diagram diff & property match” on page 363 and “Absolute times” on page 380). Please note that units are not yet supported: relative time constraints can only contain a valid comparison operator (<, >, <=, >=, ...) followed by a real number.

6.2.3.8 Co-regions

A co-region on a lifeline specifies that all events happening on this lifeline can happen in any order, and not only the order specified graphically. For example:

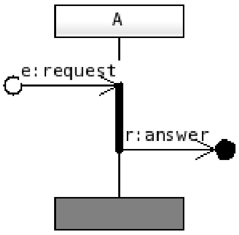


The coregion, indicated by the dotted line on the `server` lifeline, indicates that the timer and the outputs of messages `update` and `answer` can happen in any order.

Note that co-regions are not supported in the conformance checking feature of *PragmaDev Tracer* (“Conformance checking: diagram diff & property match” on page 363). The same semantics can usually be specified by using inline expressions; see “Inline expressions” on page 376 for details.

6.2.3.9 PSC strict operator

Events specified on lifeline in PSC diagrams are supposed to be loosely ordered by default. This means that if anything happens between two of these events, the property is matched anyway. It is however possible to specify a strict ordering for a set of events, meaning that these events must happen in this order without anything in between. This is done with the *strict operator*, that looks like follows:



This means that a request message received by A must be immediately followed by the output of an answer message, without anything in between (see “PSC-specific normal, required and failed message syntax” on page 364 for the PSC-specific link text syntax).

6.2.3.10 PSC constraints: wanted and unwanted messages & chains

PSC diagrams allow to specify on a message a set of messages that must or must not appear before or after it for the property to match. Unlike other messages, the messages in these constraint appear in what PSC calls the *intra-message* format, i.e as a text formatted like: `<sender instance name>.<message name>.<receiver instance name>`.

in the PSC specification, the constraint itself is represented by a symbol appearing under one end of the message link:

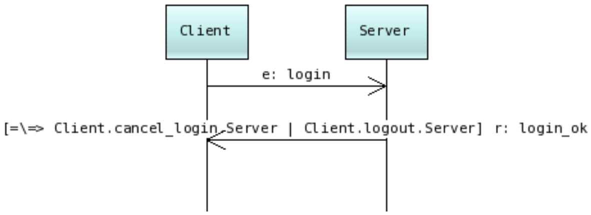
- If it appears under the link start (message output on sender), it is a *past constraint*, meaning it must be satisfied before the message is sent for the property to match;
- If it appears under the link end (message input on receiver), it is a *future constraint*, meaning it must be satisfied after the message has been received for the property to match.

In PragmaDev Studio, the constraint is actually specified directly in the text of the link. So a past constraint will appear in square brackets before the text for the message itself:
`[constraint] message_name(parameters…)`

and a future constraint will appear after the text for the message:
`message_name(parameters…) [constraint]`

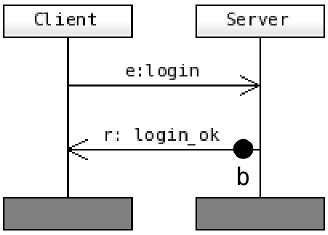
Constraints can have several forms:

- An *unwanted message constraint* specifies a set of messages that should not appear. If any of the specified messages appear, the constraint is not satisfied and the property does not match. In PragmaDev Studio, this kind of constraint is represented as follows:


Client
Server
e: login
[=\=> Client.cancel_login.Server | Client.logout.Server] r: login_ok

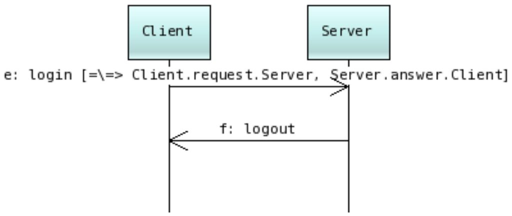
  The brackets isolate the constraint from the messgae itself, the “`=\=>`” is the standard prefix for an unwanted constraint in PragmaDev Studio, and the messages that should not appear before the `login_ok` message are separated by a “`|`”, meaning that if `Client` sends a `login` message to `Server`, `Server` must answer by sending back a `login_ok` message, unless either the message `cancel_login` has been sent from `Client` to `Server` before, or the message `logout` has been sent by `Client` to `Server` before.

Note that is standard PSC, the representation would be something like:

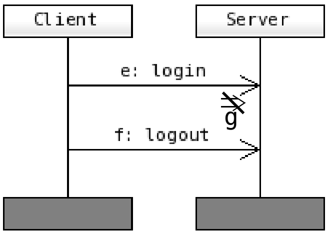


b = {Client.cancel_login.Server, Client.logout.Server}

- An *unwanted chain constraint* specifies a sequence of events that should not appear. If all messages in the constraint appear in the order specified in the constraint, then the property does not match. This kind of constraint is represented in PragmaDev Studio as follows:

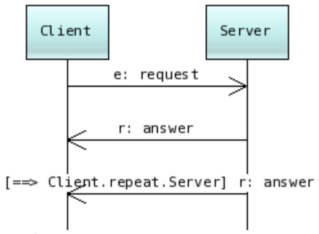


The brackets and prefix are the same as in the unwanted message constraint above, but the separator between the messages in the constarint is now a “,”, denoting a sequence. The constraint also appears after the message text, so this is a future constraint. So this means that if `Client` sends a `login` message to `Server`, it is a property failure if it sends a `logout` message after it, unless it has sent the `request` message and `Server` has sent back the `answer` message in-between.
Note that in standard PSC, the representation would be something like:

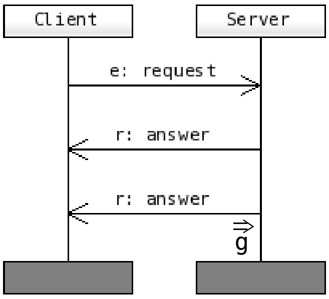


g = (Client.request.Server, Server.answer.Client)

- A *wanted chain constraint* specifies a sequence of events that must appear. If any of the messages in the constraint does not appear, or the messages appear in a different order than the one specified in the constraint, then the property does not match. This kind of constraint is represented in PragmaDev Studio as follows:



The constraint appears before the message name, so it's a past constraint. The prefix "==>" is the standard one for all wanted constraints in PragmaDev Studio. So this specifies that if `Client` sends a `request` message to `Server`, `Server` must send back an `answer` message, and then another one if `Client` sends the `repeat` message after the first `answer`.
Note that the standard PSC representation would be something like:



g = (Client.repeat.Server)

Note: PragmaDev Studio actually supports more general types of constraints called wanted and unwanted alternative chain constraint. These merge the message and chain constraints described above. The general syntax for these constraints is:

[<*constraint type prefix*> $I_1.m_1.J_1,I_2.m_2.J_2,\ldots \mid I_n.m_n.J_n,\ldots \mid I_m.m_m.J_m,\ldots$]

where <*constraint type prefix*> can be either ==> for a wanted constraint, or =\=> for an unwanted constraint.

- If the constraint is unwanted, this specifies that neither the sequence $I_1.m_1.J_1$, $I_2.m_2.J_2$, ..., nor the sequence $I_n.m_n.J_n$, ..., nor the sequence $I_m.m_m.J_m$, ... should appear for the property to match.
- If the constraint is wanted, this specifies that either the sequence $I_1.m_1.J_1$, $I_2.m_2.J_2$, ..., or the sequence $I_n.m_n.J_n$, ..., or the sequence $I_m.m_m.J_m$, ... must appear for the property to match.

This allows to represent all the PSC constraint kinds:

- An unwanted message constraint $\{I_1.m_1.J_1,\ I_2.m_2.J_2\}$ will be represented as: [=\=> $I_1.m_1.J_1 \mid I_2.m_2.J_2$]
- An unwanted chain constraint $(I_1.m_1.J_1,\ I_2.m_2.J_2)$ will be represented as: [=\=> $I_1.m_1.J_1,\ I_2.m_2.J_2$]

- A wanted chain constraint (`I`$_1$`.m`$_1$`.J`$_1$`, I`$_2$`.m`$_2$`.J`$_2$) will be represented as: `[==> I`$_1$`.m`$_1$`.J`$_1$`, I`$_2$`.m`$_2$`.J`$_2$`]`

6.2.4 Main symbols

6.2.4.1 Lifeline

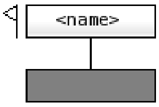
A lifeline represents an interacting entity in a MSC or PSC diagram, as explained in "General diagram format" on page 363. PragmaDev Tracer allows the instance name appearing in the lifeline head to have the following format:

```
<instance name>[:<class name>][(<instance identifier>)]
```

Lifelines can appear in all kinds of diagrams: SDL and SDL-RT MSC trace or specification diagrams, as well as PSC diagrams.

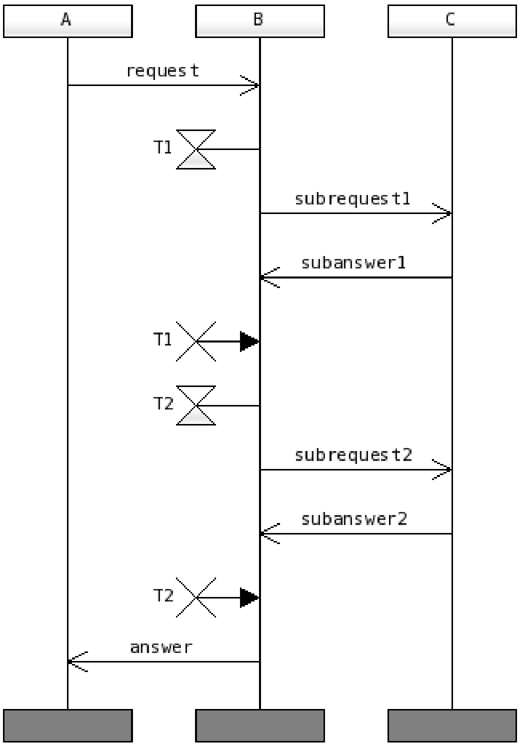
6.2.4.2 Semaphore lifeline

Semaphore lifelines are a SDL-RT extension to the standardized MSC format allowing to represent semaphores in the running system. They are displayed like a regular lifeline with an added flag near the lifeline head:

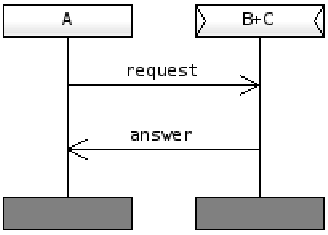


6.2.4.3 Collapsed lifelines

Collapsed lifelines are a *PragmaDev Tracer* extension and result from a 'collapse' operation. This allows to represent a set of lifelines as a single lifeline, events happening between the lifelines in the set being hidden. For example, after collapsing the instance B and C in the following diagram:



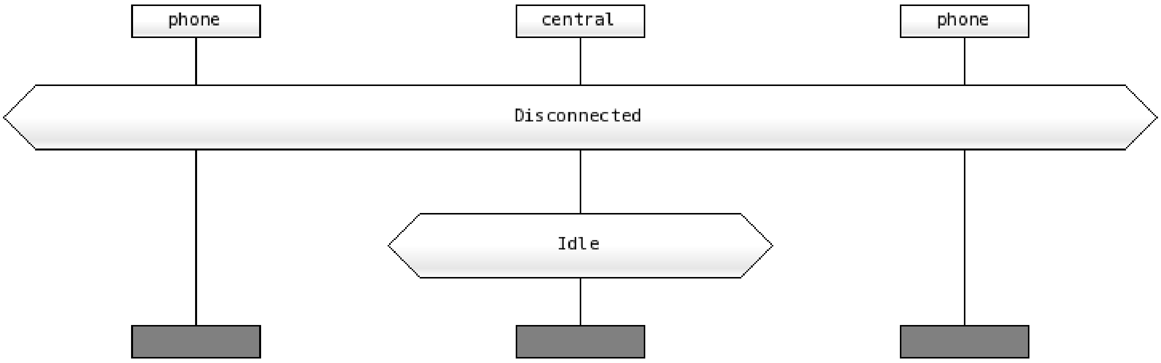
the diagram appears as follows:



6.2.4.4 Condition or instance state symbols

A condition symbol represents a condition for all the lifeline it spans. It is typically used to represent a state for a set of lifelines, the whole system, or for one specific lifeline. *PragmaDev Tracer* uses this symbol to represent state changes for an instance received via the ‘`taskChangedState`’ or ‘`ps`’ commands.

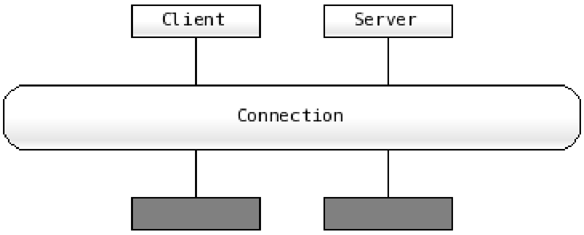
Here is an example of 2 condition symbols, the first one for the whole set of lifelines, and the second one for only one lifeline:



6.2.4.5 MSC references

A MSC diagram can reference another one by using a MSC reference symbol. This can be used to split a big MSC into smaller parts, or to reference the same sequence of events several times in a MSC diagram. This kind of symbol is normally only found in specification or PSC diagrams.

Here is an example of a MSC diagram referencing another one, called ‘Connection’:

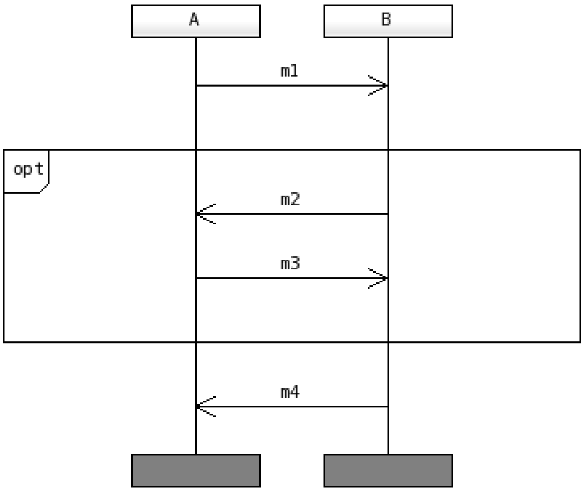


Note that in the current version of *PragmaDev Tracer*, there is no way to actually attach a MSC diagram to a MSC reference symbol. So this kind of symbol is mainly supported for documentation purposes.

6.2.4.6 Inline expressions

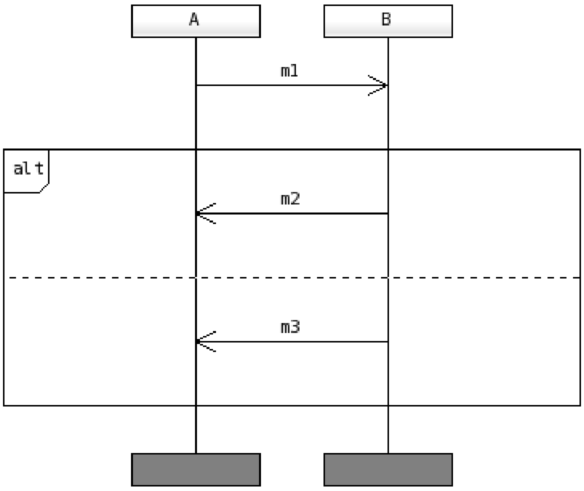
An inline expression in a specification or PSC diagram is a way to specify specific semantics for a group of events. The semantics depend on the kind of inline expression:

- An ‘`opt`’ inline expression specifies an optional set of events. For example:



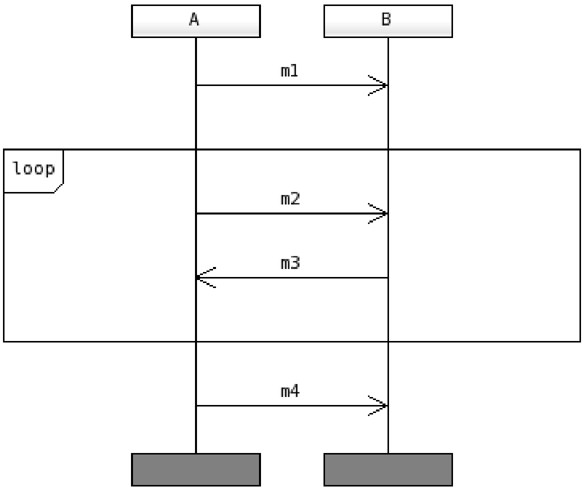
specifies that the message m1 is sent from A to B, then B may send m2 to A, which answers m3, then B sends m4 to A. So the sequences m1-m2-m3-m4, and m1-m4 are both valid.

- An ‘`alt`’ inline expression specifies a set of alternative behaviors. For example:



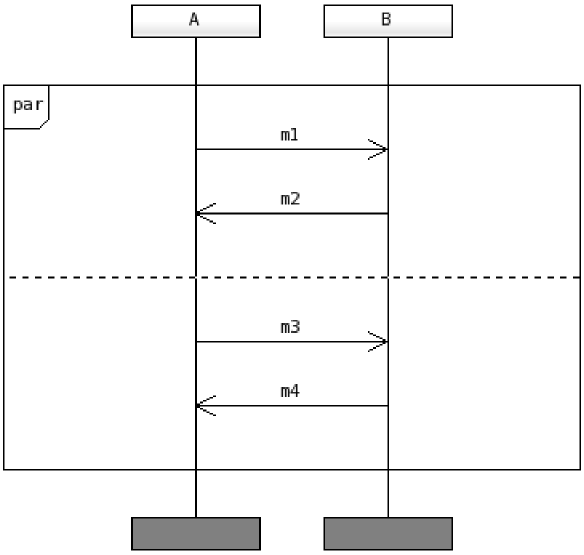
specifies that when A sends m1 to B, B may answer by sending back m2, or m3. So the sequences m1-m2 and m1-m3 are both valid.
An ‘`alt`’ inline expression must have at least two compartments in it, and can have as many as needed.

- A ‘loop’ inline expression specifies a set of events that might be repeated several times. For example:



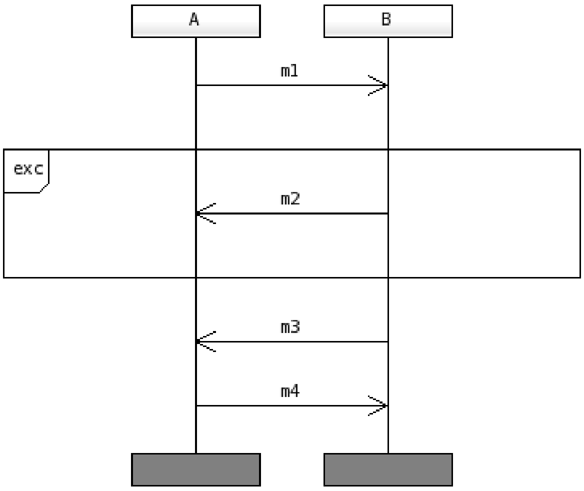
specifies that after A has sent the message m1 to B, it may send any number of messages m2, to which B will answer by the message m3, until A sends the message m4 to B. So the sequences m1-m4, m1-m2-m3-m4, m1-m2-m3-m2-m3-m4, and so on, are all valid.
Note that the MSC standard allows to indicate minimum and maximum number of repeats in loop inline expressions. This feature is not yet available in *PragmaDev Tracer*.

- A ‘par’ inline expression specifies a set of event sequences that must all happen, but in any order. For example:



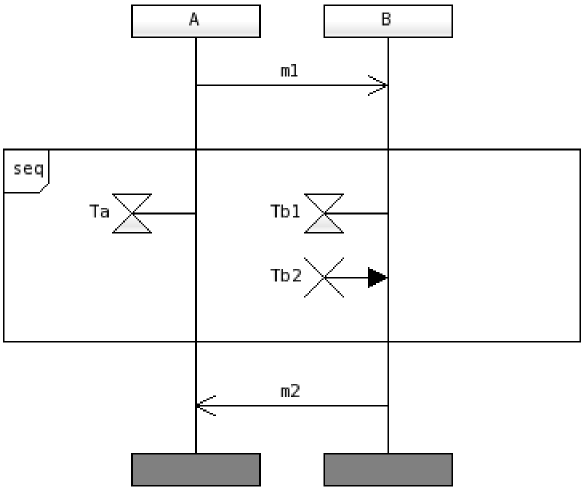
specifies that the two sequences A sending m1 to B and B answering m2, and A sending m3 to B and B answering m4 must both happen, but that the order is not significant between the sequences. So the global sequences m1-m2-m3-m4 and m3-m4-m1-m2 are both valid.
A ‘par’ inline expression must have at least 2 compartments, and can have as many as needed.

- An ‘exc’ inline expression represents an exception. This means the sequence of events in the inline expression is an error case and terminates the scenario. For example:

  specifies that when A sends m1 to B and B answers m2, there is an error and the scenario should stop. So the sequence m1-m2 is valid, but is an error case, and the sequence m1-m3-m4 is valid and is a normal execution.
  Note that the MSC standard represents an ‘exc’ inline expression with a dotted bottom line. *PragmaDev Tracer* uses a solid line in the current version.

- A ‘seq’ inline expression represents a weak sequence. This means that within such an inline expression, the events on a specific lifeline must happen in the given order, but the general ordering can be anything. For example:

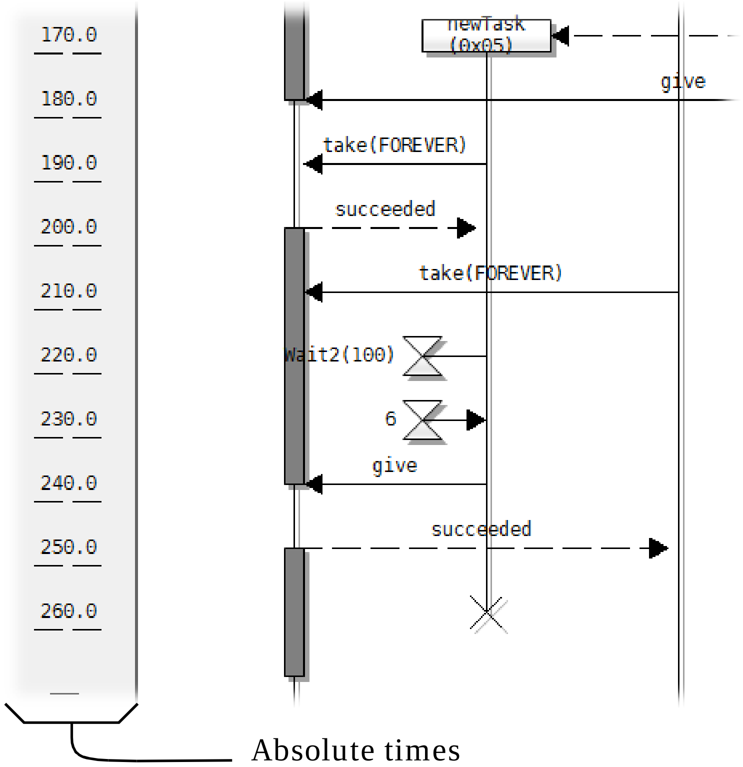
  

  This means that on lifeline B, the starting of Tb1 has to happen before the cancelling of Tb2, but that the starting of Ta by A can happen at anytime: before the starting of Tb1, after the cancelling of Tb2 or between the two.
  Note that this kind of inline expression is not supported in conformance checking (“Conformance checking: diagram diff & property match” on page 363).

6.2.4.7 Absolute times

In the MSC standard, absolute times can be associated to any event in the diagram by using a symbol consisting only in a dashed underline under the text for the time.

PragmaDev Tracer supports absolute times, but only associated to complete ‘event rows’: the times are displayed in the left margin of the diagram and are associated to all events with the same y coordinate, instead of any event. To keep the same representation as in the MSC standard, each absolute time is displayed with a dashed underline:



Absolute times

These absolute times are the reference when verifying relative time constraints during conformance checking (see “Relative time constraints” on page 369 and “Conformance checking: diagram diff & property match” on page 363). Please note that units are not yet supported: absolute time constraints must be written as a real number only.

6.2.4.8 Comments

A comment symbol just contains a documentation text for the item it is attached to. Comment symbols are not yet supported in *PragmaDev Tracer*.

6.2.4.9 Texts

A text symbol contains informal text usually describing global items in the diagram. Text symbols are not supported yet in *PragmaDev Tracer*.

6.3 - Usage

6.3.1 Launching PragmaDev Tracer

There are 3 ways to launch PragmaDev Tracer:

- It is fully included in the free version of PragmaDev Studio. So Launching PragmaDev Studio will provide a full featured tracer.
- It also has a command-line interface, wich is available via 2 commands:
    - The executable `pragmatracer` (`pragmatracer.exe` on Windows) with the `--nw` option allows to record traces without a GUI.
    - The executable `pragmatracercommand` (`pragmatracercommand.exe` on Windows) allows to access other features such as diagram comparison, without a GUI. It is described in “Command line interface” on page 386.
- It also has a dedicated graphical user interface that can be launched via the command `pragmatracer` (`pragmatracer.exe` on Windows) without the `--nw` option. Note that this way of launching PragmaDev Tracer is deprecated and will eventually stop working.

The usage for the `pragmatracer` command is:

```
pragmatracer [-p <portNumber>] [-f <fileName>] [-d <directory>] [--nw]
```

- `-p <portNumber>`: sets the port number to use when starting socket connection. If not set, the tracer will use the port number specified in the preferences.
- `-f <fileName>`: the filename where to save the trace. If provided the trace will be saved with this name; otherwise a name will be generated or asked to the user.
- `-d <directory>`: the default directory in which the trace will be saved. Used only if no file name is provided or if the file name does not indicate the entire path of the file.
- `--nw`: to launch the PragmaDev Tracer in command line mode.

6.3.2 Connection

The connection is made by socket where *PragmaDev Tracer* runs as a server: the socket is initialized on a port number and then waits for a connection from a client. The port number can be set in the tracer preferences (see “Tracer preferences” on page 31), or via an option for the command-line mode. If no port number has been provided when starting the connection, *PragmaDev Tracer* uses the default value 50000. This port number must be the same for the client application which will connect to the tracer. If *PragmaDev Tracer* is in command line mode, the socket is opened as soon as the tracer is started.

Please note several clients can connect to the same *PragmaDev Tracer* and contribute to the same trace. As the socket can accept several connections, it might get tricky to synchronize several clients. For example, if 2 client applications are executed in a row in a shell script, the beginning of the trace of the second client application might get mixed with the end of the trace of the first one. To avoid this behavior, it is possible to ask for an acknowledgment to the tracer (see “Acknowledgment” on page 398). Waiting for an acknowledgment after the last command in the client program guarantees that the next client program will start when all commands of the previous program have been treated.

6.4 - Graphical user interface

6.4.1 Integration with PragmaDev Studio

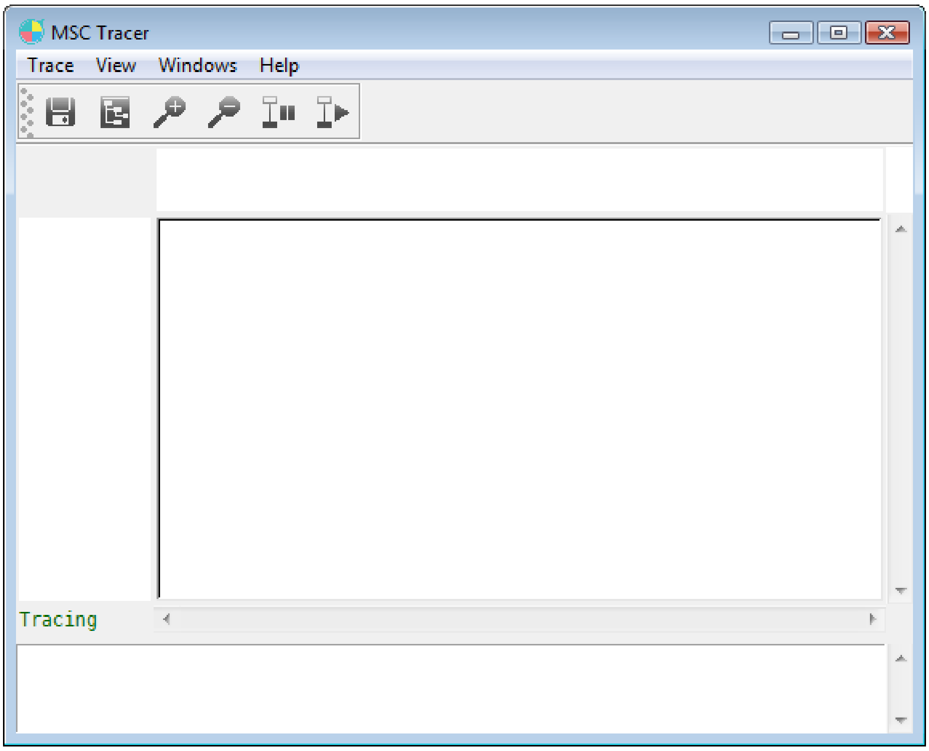
The graphical user interface for PragmaDev Tracer is completely integrated in the free version of PragmaDev Studio. The features that are specific to PragmaDev Tracer are:

- The creation of a trace by reading tracing commands from a socket. This feature is launched via the "*New MSC trace from socket...*" entry in the project manager's "*Project*" menu, or by simply clicking in the project manager's tool bar. This feature does not require a project to be opened: if there is no current project, a new empty one will be created automatically.
- The creation of a trace by reading a set of tracing command from a file. This feature is launched via the entry "*Import MSC tracer command file...*" in the project manager's "*Project*" menu. This feature only works when there is an opened project, and the resulting MSC diagram is inserted in this project.

Once created, traces are saved as regular MSC diagrams that can be edited with the standard MSC editor, as described in "MSC editor" on page 83. Specific options for the tracer are found in the "*Tracer*" tab of PragmaDev Studio's preferences (see "Tracer preferences" on page 31).

6.4.2 Tracer window

Whenever a trace is started, either from a socket connection or PragmaDev Studio's simulator or debugger, a trace window pops up:



Empty tracer window

The trace can be paused or resumed by using the buttons ⏸ and ▶ respectively.

By default the trace is running. When the trace is paused, all received commands are ignored. Note that, by pausing the trace, the socket is not released so the client application can continue to send commands without generating an error.

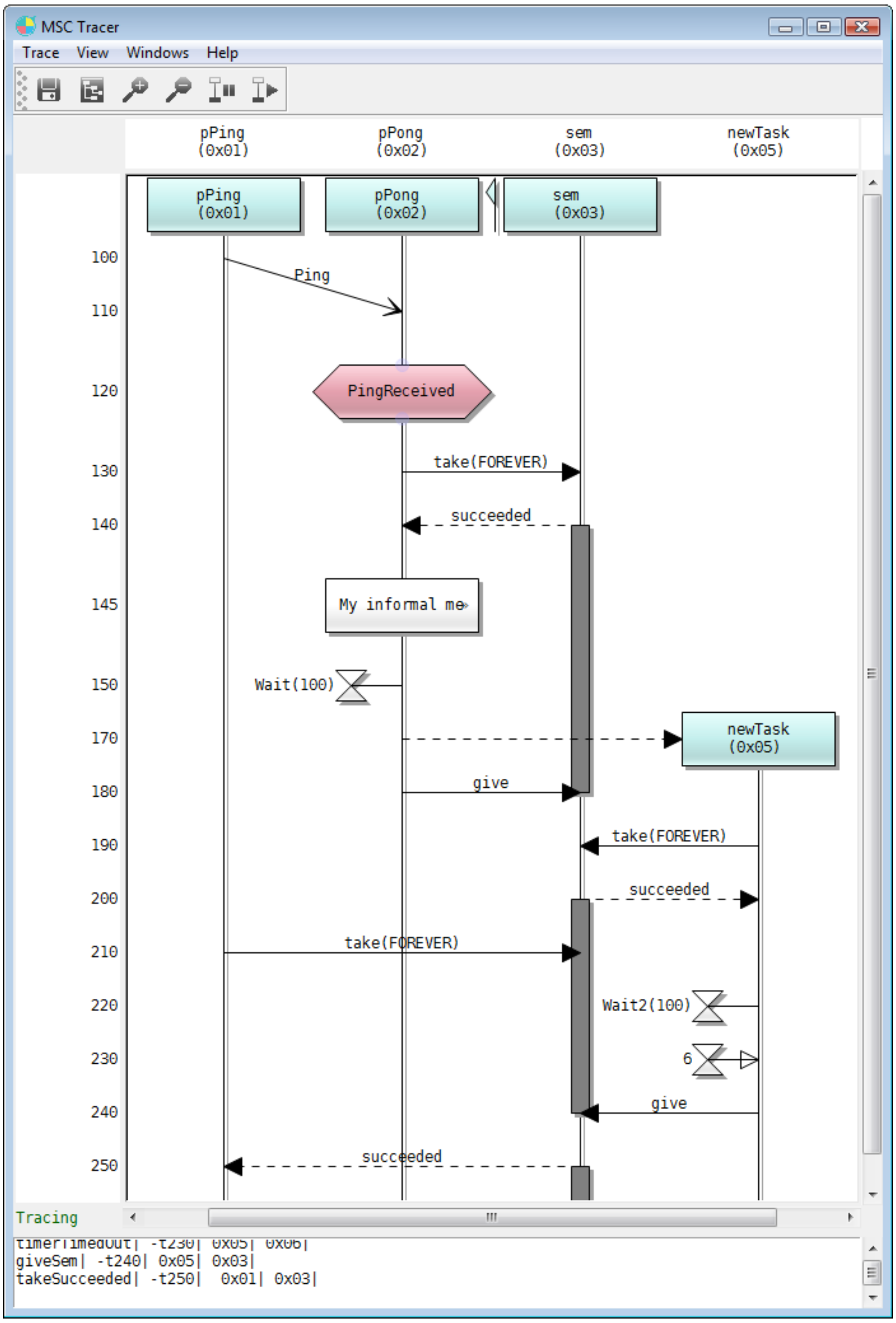
The MSC trace can be zoomed in or zoomed out with 🔍+ and 🔍− .

Note that if a limit on the number of events is set in the preferences, only the last events will appear in the trace (see “Tracer preferences” on page 31). This means that when the limit is reached, the events at the beginning of the trace will start to disappear from the tracer window. This doesn’t have any impact on the saved MSC diagram, which will still contain all recorded events from the beginning.

The trace can be saved at any moment with the button 💾 .

A name for the file to save the diagram to will be asked. When closing the tracer window, the trace stops and the socket connection is closed.

Here is an example of a trace displayed in the tracer window:



```
timerTimedOut| -t230| 0x05| 0x06|
giveSem| -t240| 0x05| 0x03|
takeSucceeded| -t250|  0x01| 0x03|
```

Trace in tracer window

6.5 - Command line interface

Specific features of *PragmaDev Tracer* are available via a command line interface via the standard executable for PragmaDev Studio's command-line interface. This executable is named `pragmastudiocommand.exe` on Windows and `pragmastudiocommand` on Unix. This executable accepts a subcommand called `tracer` wihich will run PragmaDev tracer without a GUI. The general syntax for the command as a whole is:

```
pragmastudiocommand tracer <options>
```

The available options are:

- `-f <file name>`: specifies the name of the file where the trace will be saved;
- `-d <directory>`: specifies the default directory for all files;
- `-p <port number>`: specifies the port number on which client connections must be made (default 5000);
- `-n <tracer name>`: name for the tracer window (ignored in command-line mode);
- `--nmw`: If present, only the tracer window is shown, not the main window (ignored in command-line mode).

6.6 - Tracer commands

6.6.1 Command reference

PragmaDev Tracer accepts several commands to generate the MSC trace. They represent a specific real-time event illustrated by a symbol in the MSC diagram.

The field separator in a command is:

'| ' (pipe followed by a space)

The command is ended by:

'|\n' (pipe followed by the line feed character i.e. 0x13)

To insert a printable pipe in the command, another pipe must be put up front:

'||' => printable '|'

For example, the command for a task creation named pPing with pid 1 will be:

```
taskCreated| -npPing| 0x01|\n
```

The generic syntax of a command is:

```
<command_name or alias>[| <options>]| <parameters>|
<optional_arguments>|\n
```

If a command name or alias is not recognized, the whole command is ignored.

If one of the parameters is not provided, the whole command is ignored.

If a provided option does not exist, the whole command is ignored.

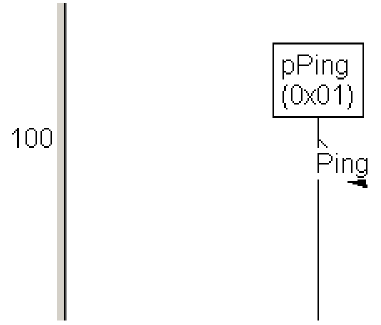
If the number of optional arguments provided is greater than the expected number, the command interpreter will ignore the additional arguments.

Each command has an alias; e.g.: `ms` has the same meaning as `messageSent`.

6.6.1.1 Common options and arguments

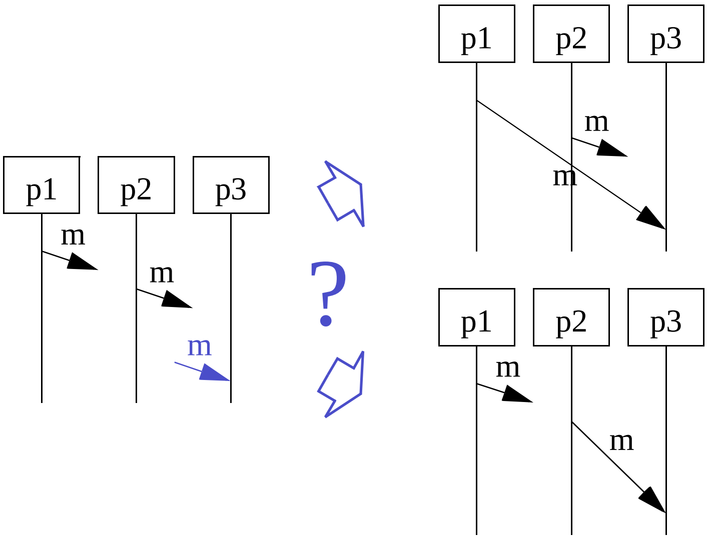
The following options and arguments may appear in several commands:

- `-t<time>`
  Represents the time when an event occur; if provided, a symbol containing the value of time of the associated event will appear at the left of the MSC trace at the same height that the event symbol (task created, message sent...) involved.

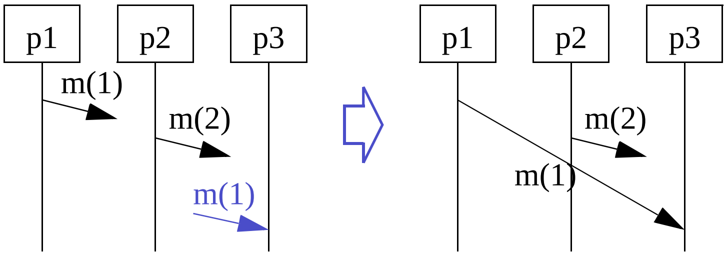


Here the sending of message ping from process pPing occurs at time 100 after the beginning of system execution.

- -i<mId>
Represents a unique identifier for a given message. It comes with the commands messageSend, messageReceive, and messageSave. This option provides a unique identifier for the sending of a message, allowing to match a message receive to the corresponding message send. If this option is not used, a message receive will always suppose that the received message is the last message sent with the same name. This may result in errors in the trace if several messages with the same name exist at the same moment.
For example:



Without message unique ids



With message unique ids (after message name)

- -M<elt_id>
Specifies the identifier for the model element for this event. The identifier is a string that is only meaningful for the modelling tool used to create the element. The identifier is not displayed, but is simply stored in the trace diagram once this one is saved.
- <sigNum>
Numerical identifier for the name of a message. Note both the name and this identifier may have to be provided.
- <pId> or <semId>

Identifier of a process or semaphore; it is an unique identifier. pId and semId represents the id of the lifeline, so it is not possible to have a semaphore and a process with the same id.

- <tId>
  unique timer identifier; it must be provided, and may be used as the name of the timer if no option -T is provided.

6.6.1.2 Task creation

To trace a task creation, the command syntax is:

```
taskCreated[| -t<time>][| -M<elt_id>]
    [| -c<creatorId>][| -n<pName>][| -N<creatorName>]| <pId>|\n
```

or

```
pc[| -t<time>][| -M<elt_id>][| -c<creatorId>][| -n<pName>]
    [| -N<creatorName>]| <pId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -c: identifier of creator process,
- -N: name of creator process,
- -n: name of created process; if this option is not present, the process name on the MSC trace will be its id.

parameters:

- <pId>: unique process identifier.

6.6.1.3 Task deletion

To trace a task deletion, the command syntax is:

```
taskDeleted[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>|\n
```

or

```
pd[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -n: process name,

parameters:

- <pId>: unique process identifier.

6.6.1.4 Messages

6.6.1.5 Message sending

To trace a message sending from a process, the command syntax is:

```
messageSent[| -t<time>][| -M<elt_id>][| -d<msgData>][| -n<pName>]
     [| -i<mId>]| <pId>| <sigNum>| <msgName>|\n
```

or

```
ms[| -t<time>][| -M<elt_id>][| -d<msgData>][| -n<pName>][| -i<mId>]
     | <pId>| <sigNum>| <msgName>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -d: data of the message. Spaces are allowed and the data format is free except for the ‘|’ character that should be doubled:’||’. Detailed format is described in “Message parameter format” on page 391.
- -n: name of the process sending the message.
- -i: unique identifier for message, See “Common options and arguments” on page 387.

parameters:

- <pId>: unique process identifier.
- <sigNum>: signal number of the message, See “Common options and arguments” on page 387.
- <msgName>: message name.

6.6.1.6 Message reception

To trace a message reception, the command syntax is:

```
messageReceived[| -t<time>][| -M<elt_id>][| -d<msgData>]
     [| -n<pName>][| -i<mId>]| <pId>| <sigNum>| <msgName>|\n
```

or

```
mr[| -t<time>][| -M<elt_id>][| -d<msgData>][| -n<pName>][| -i<mId>]
     | <pId>| <sigNum>| <msgName>|\n
```

options:

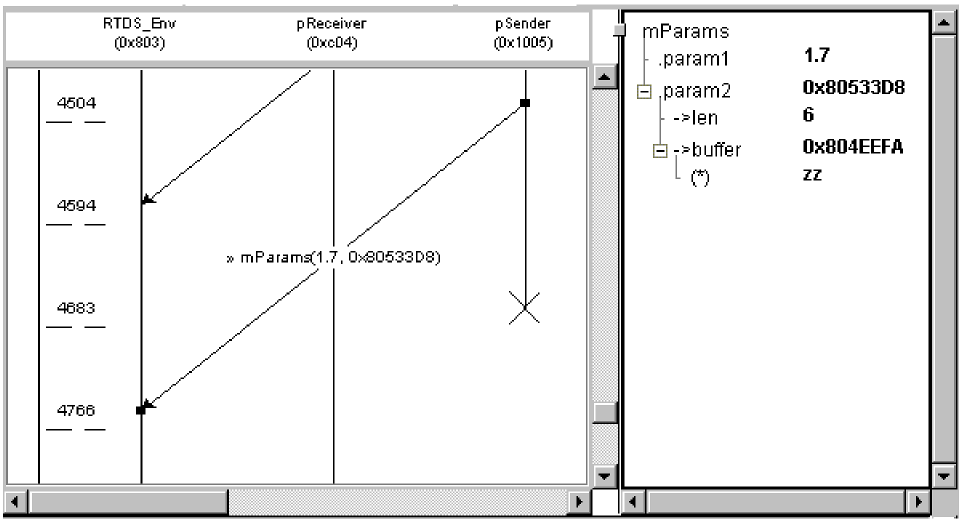
- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -d: data of the message. Spaces are allowed and the data format is free except for the ‘|’ character that should be doubled:’||’. Detailed format is described in “Message parameter format” on page 391.
- -n: name of the process receiving the message.
- -i: unique message identifier, See “Common options and arguments” on page 387.

parameters:

- `<pId>`: unique process identifier.
- `<sigNum>`: signal number of the message, See “Common options and arguments” on page 387.
- `<msgName>`: message name.

6.6.1.7 Message parameter format

The format described below applies to structured parameters in messages sent or received. Sending structured messages with the following format allows to display parameters in a tree in the MSC editor as shown below.



mParams
.param1 1.7
.param2 0x80533D8
->len 6
->buffer 0x804EEFA
(*) zz

The format for this text or argument depends on whether the message is structured or not. Structured parameters are fully described in RTDS 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. For example, for an `int*`:
    - `804A51FE|:67` will define the pointer to be `0x804A51FE` and `67` will be the pointed value;
    - `|:123` will only describe the pointed value to be `123`;

    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, the field `i` will be set to `4` and the field `s` will point to the `"abcd"`.

6.6.1.8 Message saving

To trace a saved message, the command syntax is:

```
messageSaved[| -t<time>][| -n<pName>][| -i<mId>]| <pId>| <sigNum>
     | <msgName>|\n
```

or

```
mv[| -t<time>][| -n<pName>][| -i<mId>]| <pId>| <sigNum>| <msgName>|\n
```

options:

- `-t`, `-M`: time and model element identifier for the event; See “Common options and arguments” on page 387.
- `-n`: name of process that saves the message,
- `-i`: unique message identifier, See “Common options and arguments” on page 387.

parameters:

- `<pId>`: unique process identifier,
- `<sigNum>`: signal number for the message, See “Common options and arguments” on page 387.
- `<msgName>`: saved message name

6.6.1.9 Semaphore creation

To trace a semaphore creation, the command syntax is:

```
semaphoreCreated[| -t<time>][| -M<elt_id>][| -s<semName>]
     [| -c<creatorId>][| -N<creatorName>][| -a<stillAvailable>]
     | <pId>|\n
```

or

```
sc[| -t<time>][| -M<elt_id>][| -s<semName>][| -c<creatorId>]
     [| -N<creatorName>][| -a<stillAvailable>]| <pId>|\n
```

options:

- `-t`, `-M`: time and model element identifier for the event; See “Common options and arguments” on page 387.
- `-s`: name of semaphore; if not provided the name of the created semaphore in the MSC trace will be its internal identifier,
- `-c`: identifier of the creator process,
- `-N`: name of the creator process,
- `-a`: boolean indicating if semaphore is empty (value 0) or full (value 1),

parameters:

- `<pId>`: unique identifier of the created semaphore.

6.6.1.10 Semaphore deletion

To trace a semaphore deletion, the command syntax is:

```
semaphoreDeleted[| -t<time>][| -M<elt_id>][| -s<semName>]
    [| -c<destructorId>][| -N<destructorName>]| <pId>|\n
```

or

```
sd[| -t<time>][| -M<elt_id>][| -s<semName>][| -c<destructorId>]
    [| -N<destructorName>]| <pId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -s: name of deleted semaphore,
- -c: identifier of the destructor process,
- -N: name of the destructor process,

parameters:

- <pId>: unique identifier of the deleted semaphore.

6.6.1.11 Semaphore take attempt

To trace an attempt to take semaphore, the command syntax is:

```
takeAttempt[| -t<time>][| -M<elt_id>]
    [| -n<pName>][| -s<semName>][| -T<timeout>]| <pId>| <semId>|\n
```

or

```
sa[| -t<time>][| -M<elt_id>][| -n<pName>][| -s<semName>][| -T<timeout>]
    | <pId>| <semId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -n: name of process taking the semaphore,
- -s: name of the taken semaphore,
- -T: timeout for the take; a value of -1 indicate that the process will wait until the take have succeeded,

parameters:

- <pId>: identifier of taker process,
- <semId>: identifier for taken semaphore.

6.6.1.12 Semaphore take succeeded

To trace a semaphore has been successfully taken, the command syntax is:

```
takeSucceeded[| -t<time>][| -M<elt_id>]
    [| -n<pName>][| -s<semName>][| -a<stillAvailable>]| <pId>|
<semId>|\n
```

or

```
ss[| -t<time>][| -M<elt_id>][| -n<pName>][| -s<semName>]
     [| -a<stillAvailable>]| <pId>| <semId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -n: name of the process taking the semaphore,
- -s: name of taken semaphore,
- -a: boolean indicating if semaphore is empty (value 0) or full (value 1),

parameters:

- <pId>: identifier for taker process,
- <semId>: identifier for taken semaphore.

6.6.1.13 Semaphore take timed out

To trace a semaphore take attempt timed out, the command syntax is:

```
takeTimedOut[| -t<time>][| -M<elt_id>][| -n<pName>][| -s<semName>]
     | <pId>| <semId>|\n
```

or

```
st[| -t<time>][| -M<elt_id>][| -n<pName>][| -s<semName>]
     | <pId>| <semId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -n: name of taker process,
- -s: name of taken semaphore,

parameters:

- <pId>: identifier of taker process,
- <semId>: identifier of taken semaphore.

6.6.1.14 Semaphore give

To trace a semaphore give, the command syntax is:

```
giveSem[| -t<time>][| -M<elt_id>][| -n<pName>][| -s<semName>]
     | <pId>| <semId>|\n
```

or

```
sg[| -t<time>][| -M<elt_id>][| -n<pName>][| -s<semName>]
     | <pId>| <semId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -n: name of the process giving the semaphore,

- `-s`: name of given semaphore,

parameters:

- `<pId>`: identifier of giver process,
- `<semId>`: identifier of given semaphore.

6.6.1.15 Timer start

To trace the start of a timer, the command syntax is:

```
timerStarted[| -t<time>][| -M<elt_id>]
     [| -n<pName>][| -T<timerName>]| <pId>| <tId>[| <timeLeft>]|\n
```

or

```
ts[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>]
     | <pId>| <tId>[| <timeLeft>]|\n
```

options:

- `-t`, `-M`: time and model element identifier for the event; See “Common options and arguments” on page 387.
- `-n`: name of process starting the timer,
- `-T`: name of the timer started,

parameters:

- `<pId>`: identifier of starter process,
- `<tId>`: identifier of timer, See “Common options and arguments” on page 387.

optional argument:

- `<timeLeft>`: time before timer times out.

6.6.1.16 Timer cancellation

To trace the cancellation of a timer, the command syntax is:

```
timerCancelled[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>]
     | <pId>| <tId>|\n
```

or

```
tc[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>]| <pId>|
<tId>|\n
```

options:

- `-t`, `-M`: time and model element identifier for the event; See “Common options and arguments” on page 387.
- `-n`: name of process having starting the timer,
- `-T`: name of the stopped timer,

parameters:

- `<pId>`: process identifier,
- `<tId>`: identifier of timer stopped, See “Common options and arguments” on page 387.

6.6.1.17 Timer timed out

To trace a timed out timer, the command syntax is:

```
timerTimedOut[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>]
     | <pId>| <tId>|\n
```

or

```
tt[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>]
     | <pId>| <tId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -n: name of process having started the timer,
- -T: name of the timer that times out ,

parameters:

- <pId>: process identifier,
- <tId>: identifier of the timer that times out, See “Common options and arguments” on page 387.

6.6.1.18 Task state changed

To trace a task state has changed, the command syntax is:

```
taskChangedState[| -t<time>][| -M<elt_id>][| -n<pName>]
     | <pId>| <stateName>|\n
```

or

```
ps[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>| <stateName>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.
- -n: name of process involved,

parameters:

- <pId>: process identifer,
- <stateName>: name of the new state for process.

6.6.1.19 Action symbol

To trace an action in a lifeline, the command syntax is:

```
information[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>| <message>|\n
```

or

```
in[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>| <message>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 387.

- `-n`: name of process involved,

parameters:

- `<pId>`: process identifer,
- `<message>`: information to be displayed in the action symbol.

6.6.1.20 Start a new MSC trace

To start a new MSC trace, the command syntax is:

```
newTrace[| -f<fileName>]|\n
```

or

```
n[| -f<fileName>]|\n
```

options:

- `-f<fileName>`: file name is the file name of the new trace.

In graphical mode, this command closes the current trace if it exists, saves it if needed and launches a new viewer window. In no window mode, the previous trace is stopped and saved and a new trace starts, ready to accept commands.

6.6.1.21 Pause MSC trace

To pause the current MSC trace, the command syntax is:

```
pause|\n
```

or

```
p|\n
```

The trace is resumed with the *resume* command.

6.6.1.22 Resume MSC trace

To resume the current MSC trace, the command syntax is:

```
resume|\n
```

or

```
r|\n
```

This command is used only after the trace has been paused via the graphical interface or with the *pause* command.

6.6.1.23 Close MSC trace

To stop the MSC trace, save and close the file, the command syntax is:

```
close|\n
```

or

```
c|\n
```

Use to indicate that the current trace will be definitively stopped and then saved. If a file name has been provided when the current trace started, it will be used to save the diagram; otherwise *PragmaDev Tracer* will generate a file name like "mscTracer.rdd" in no window mode or open a dialog asking to choose a file name.

Please note this command will have no effect in graphical mode.

6.6.1.24 Exit PragmaDev Tracer

This command makes the *PragmaDev Tracer* quit. If a trace was running, it will be stopped and saved as if a command `close` was sent.

```
exit|\n
```

or

```
e|\n
```

Please note this command will have no effect in graphical mode.

6.6.1.25 Set directory

To set the default directory, the command syntax is:

```
setDirectory| <dir>|\n
```

or

```
dr| <dir>|\n
```

parameters:

- `<dir>`: default directory.

Used to save traces when no file name is provided or when a file name without an entire path is given.

6.6.1.26 Acknowledgment

In order to be sure all commands sent have been received, it is possible to ask for an acknowledgment from *PragmaDev Tracer*. The command syntax is:

```
waitingForAck|\n
```

or

```
wa|\n
```

The acknowledgment sent back after receiving the `waitingForAck` command is the string:

```
ack
```

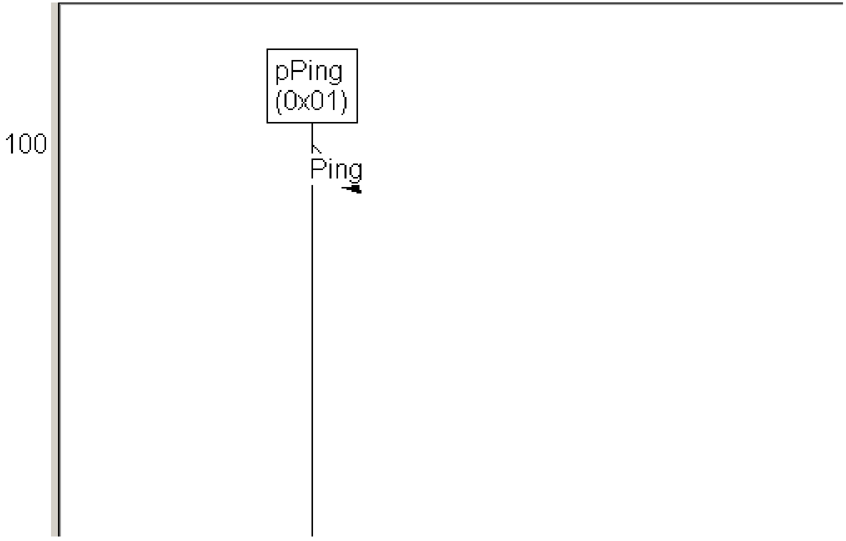
6.6.2 Tracing example

This example will illustrate the use of the tracing feature of *PragmaDev Tracer*.

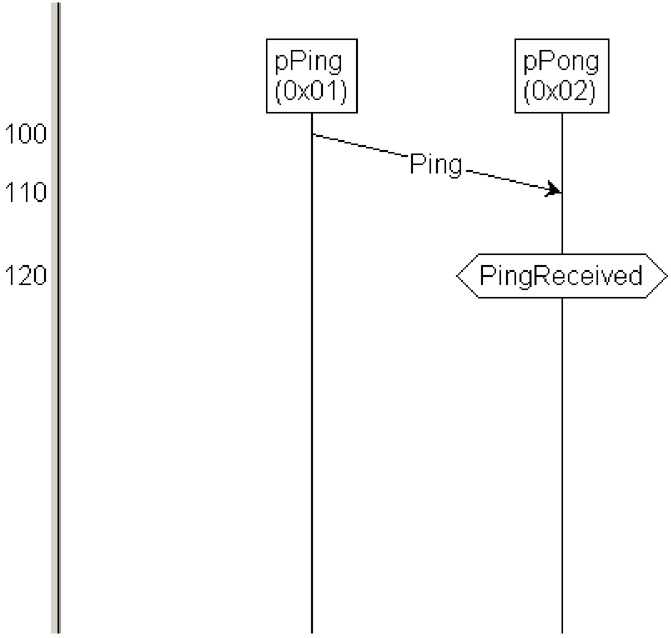
Commands sent:

- `taskCreated| -n| pPing| 0x01|\n`

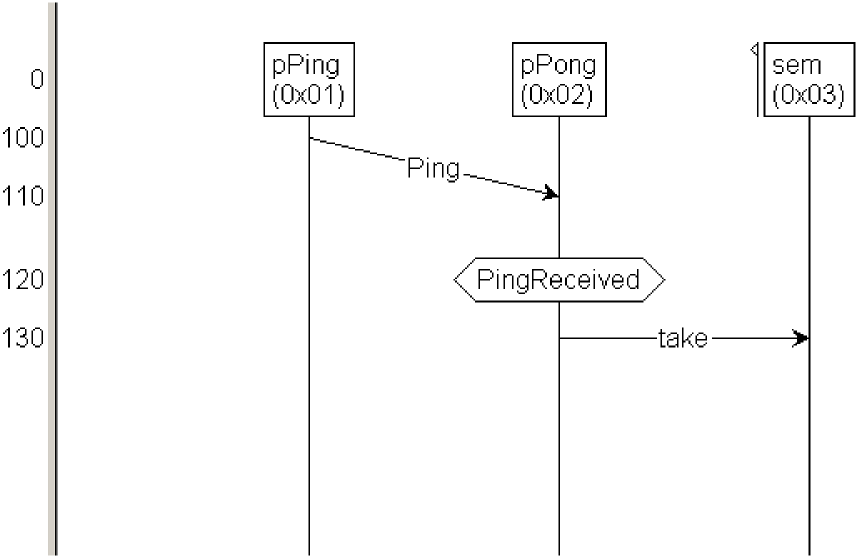
- messageSent| -t| 100| -n| pPing| 0x01| 12| Ping|\n



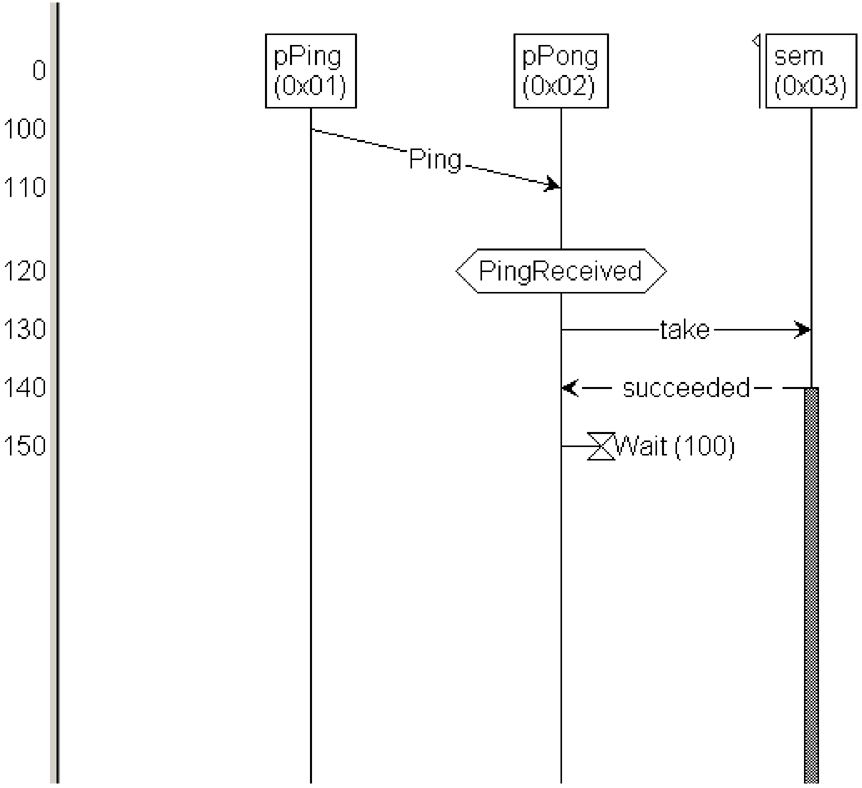
- messageReceived| -t| 110| -n| pPong| 0x02| 12| Ping|\n
- taskChangedState| -t| 120| -n| pPong| 0x02| PingReceived|\n



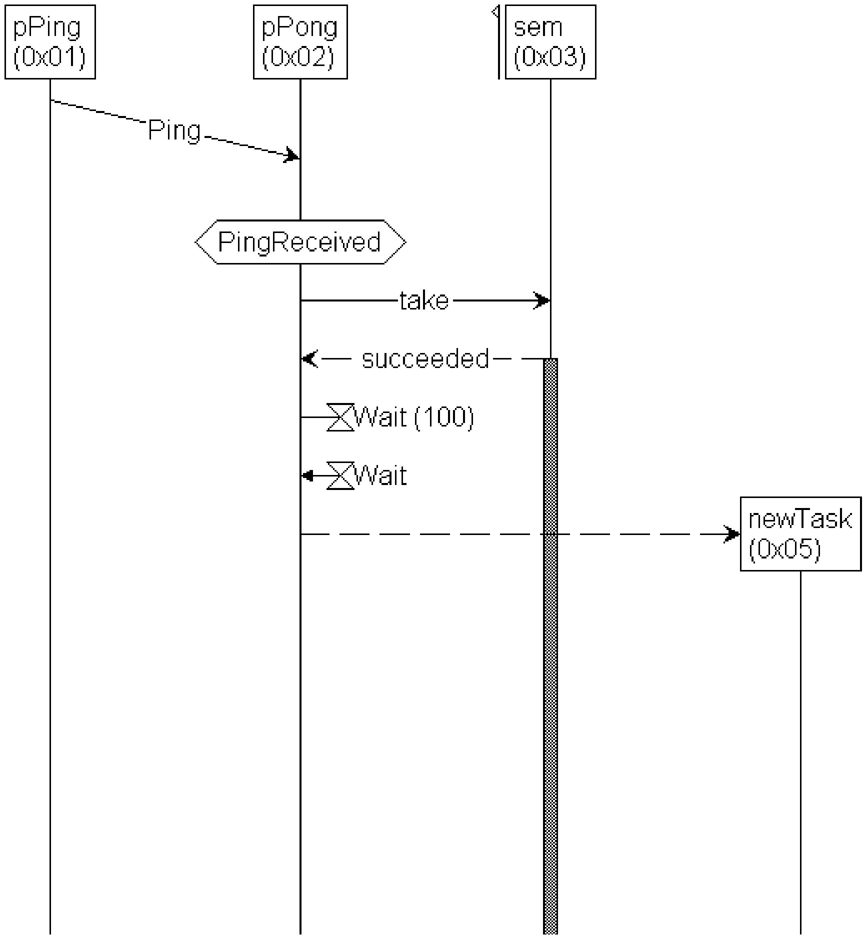
- `semaphoreCreated| -s| sem| 0x03|\n`
- `takeAttempt| -t| 130| -n| pPong| -s| sem| 0x02| 0x03|\n`



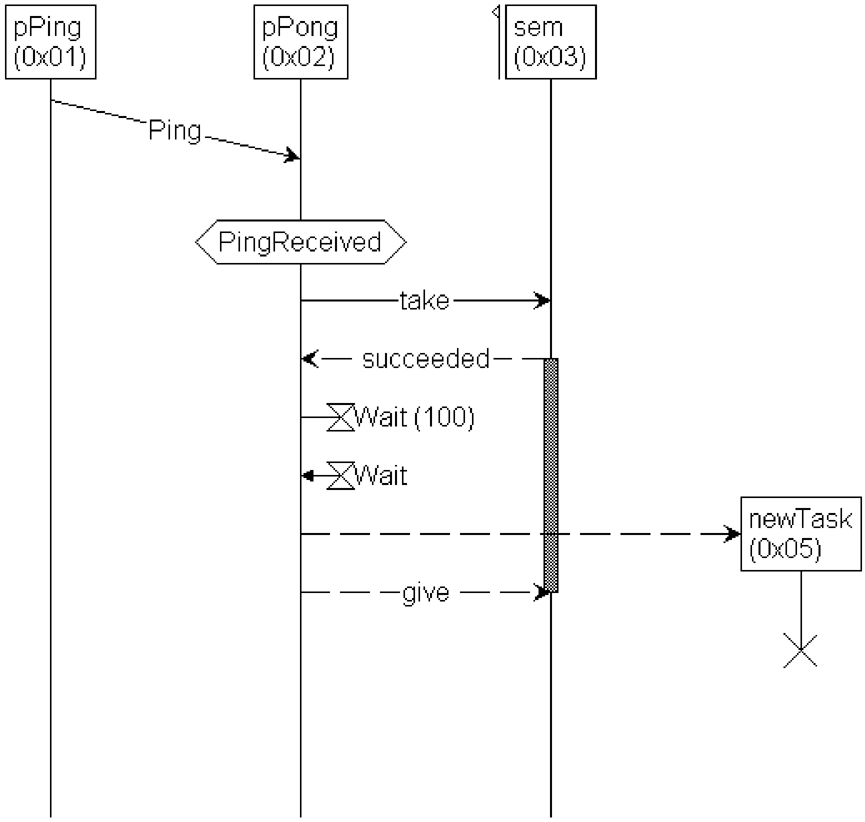
- `takeSucceeded| -t| 140| -n| pPong| -s| sem| 0x02| 0x03|\n`
- `timerStarted| -t| 150| -n| pPong| -T| Wait| 0x02| 0x04| 100|\n`



- `timerTimedOut| -t| 250| -n| pPong| -T| Wait| 0x02| 0x04|\n`
- `taskCreated| -t| 260| -c| 0x02| -n| newTask| 0x05|\n`



- `giveSem| -t| 270| -n| pPong| -s| sem| 0x02| 0x03|\n`
- `taskDeleted| -t| 280| 0x04|\n`



6.7 - Importing an 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 109. 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.

7 - PragmaDev Studio

7.1 - Scope

PragmaDev Studio includes all other modules (Specifier, Developer, Tester, and Tracer) and some extra features and links from one module to the other. This chapter covers these extra features and links exclusively. For this matter this chapter is quite advanced in the understanding of the tools capabilities.

7.2 - SDL C code generation

7.2.1 Principles

The generation of C code from SDL works mostly as generating the C code from SDL-RT systems, as described in “Code generation” on page 246. The SDL declarations and actual code are simply first translated to C. The rules for this translation are described in the reference manual.

Some code generation options described in “Code generation options” on page 249 are only meaningful for SDL code generation:

- *Data allocation*, since it only affects the C code generated for SDL declarations;
- *Operators implemented in C*, since operators can only appear in SDL type declarations;
- *Case-sensitive*, as SDL-RT systems are always case-sensitive anyway;
- *Prefix enum values names w. type name*, as it only impacts the code generated for LITERALS SDL type definitions;
- *Generate ASN.1 codecs for env. messages*, as ASN.1 encoders and decoders generation is only available for SDL systems and is not needed for SDL-RT ones.

The detailed process for ASN.1 encoders and decoders generation is described in the next section.

7.2.2 ASN.1 codecs for environment messages

If the corresponding option is the generation options is checked, the code generation will attempt to generate ASN.1 encoders and decoders for messages exchanged between the SDL system and the environment.

This requires that all types for message parameters are described in ASN.1. The only exception is SDL base types `Boolean`, `Integer`, `Real` and `CharString` that have a direct equivalent in ASN.1: `BOOLEAN`, `INTEGER`, `REAL` and `GeneralString`, respectively. These types can be used in message parameters and will be translated automatically.

When the ASN.1 codecs generation is required, the following happens after generating the code for the system:

- All messages going to the environment or coming from it are checked. If any message parameter is not defined in ASN.1, or is not a base type among those listed above, the code generation fails.
- An additional ASN.1 file is generated in the code generation directory. This file contains ASN.1 types for the messages themselves:
  - If a message m has no parameter, a type named RTDS-message-m is created which is a “syntype” for the NULL ASN.1 type;
  - If a message m has parameters, a type named RTDS-message-m is created which is a SEQUENCE with one field for each message parameter, in order. The fields are named param1, param2, and so on.

  Another type is created, always called RTDS-AllMessages, which is a CHOICE on all the messages exchanged between the system and the environment. This type is the only PDU for the ASN.1: all messages will be encoded data for this type.
- C code for the ASN.1 file is generated via an external utility, which is included in the distribution. This utility is called asn1c and is available freely at the following address: http://lionet.info/asn1c/compiler.html
  The generated code is placed in a subdirectory of the code generation directory called asn1. It includes definitions for C types representing the ASN.1 types as well as the code needed to encode and decode values of these types in the BER/DER standard ASN.1 encoding.
- A file called RTDS_asn1_codec_functions.c is generated containing the code converting PragmaDev Studio’s own internal structures for ASN.1 types to the ones used by asn1c.
- The generation of the file RTDS_messages.h is changed so that any message going to the environment uses the macro RTDS_ASN1_MESSAGE_FROM_SYSTEM, which has to be defined by the user. This macro takes as parameters the message number as defined in RTDS_gen.h as an integer, the number of bytes in the ASN.1 encoded data as an unsigned long, and a pointer on the encoded data itself as an unsigned char*.
  The file also defines another macro called RTDS_ASN1_MESSAGE_TO_SYSTEM, taking as parameter a number of bytes and a pointer on ASN.1 encoded data for the type RTDS-AllMessages described above. This macro calls a function decoding the data, buidling the descriptor for the actual message, figuring out its receiver from the message and sending it. The purpose of this macro is to be called in some external C code to send a message described as ASN.1 encoded data to the running SDL system.

The encoding used for all ASN.1 data is BER/DER. The messages sent to the environment are actually described as DER, but since DER is a subset of BER, it can be decoded with any BER decoder for ASN.1. The data describing messages sent from the environment to the system can be DER or BER.

If another ASN.1 toolkit is used to decode the data sent from the system or encode the data sent to the system, it should use all the ASN.1 files containing the definitions of the parameter types for all messages, as well as the generated file RTDS-AllMessages.asn1. All exchanged values in both ways have the type RTDS-AllMessages, described in this file.

A detailed and commented example is available in the distribution, in `Studio/ASN1Codecs`.

7.3 - Performance Analyzer

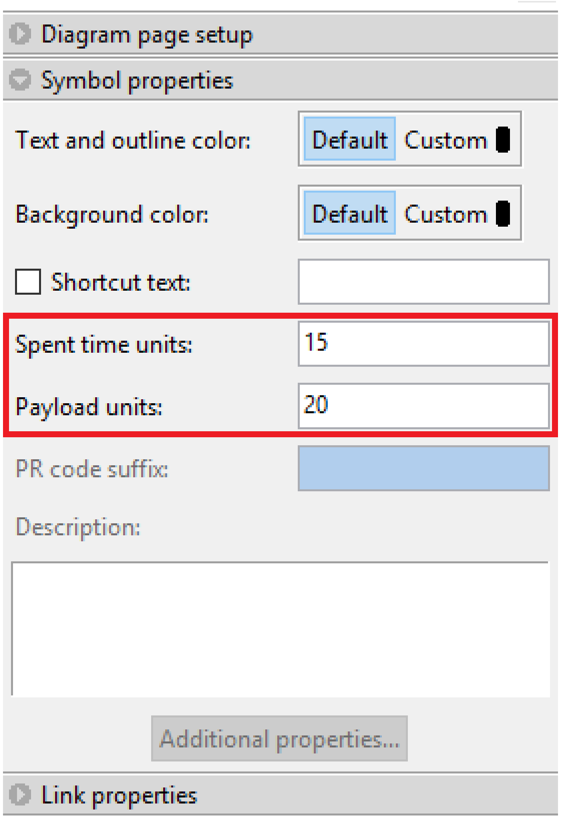
7.3.1 Objectives

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.

7.3.2 Time and payload information

7.3.2.1 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 expressions indicated in the properties of the symbol (see “Symbol and link properties” on page 62), namely “Spent time units” and “Payload units”:



Text and outline color: Default Custom

Background color: Default Custom

□ Shortcut text:

Spent time units: 15

Payload units: 20

PR code suffix:

Description:

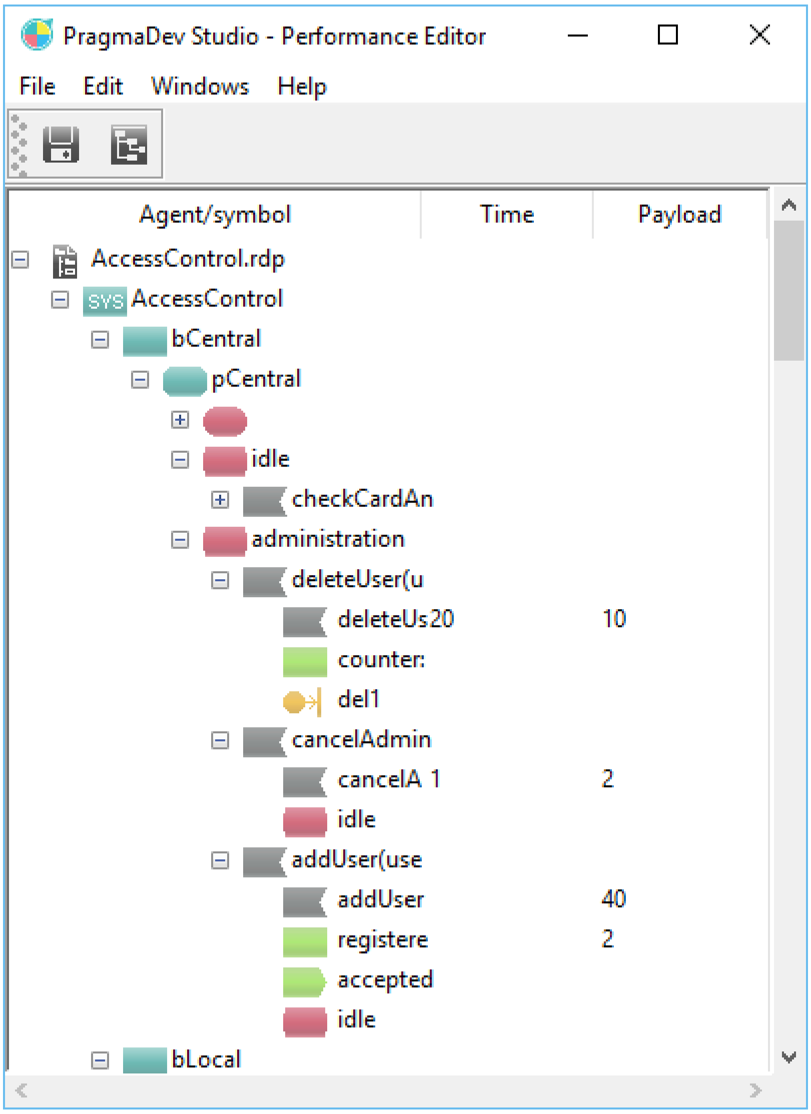
Time and payload units can be any valid SDL expression that shall however result in a positive integer value. If no value is given, then it will default to 0 (zero).

Important notes:

- The result of the SDL expression represents *units* of time and payload; the actual values are obtained by multiplying the *units* with the *factors* given in the architecture (see “The architecture” on page 408).
- The SDL expression is always evaluated *before* the symbol is executed. Exceptions to this rule are the signal input and priority input symbols (and). This is to make sure that the signal parameters (if any) are always up-to-date when used in a time and/or payload expression associated to these two symbols.

7.3.2.1.1 The Performance Editor

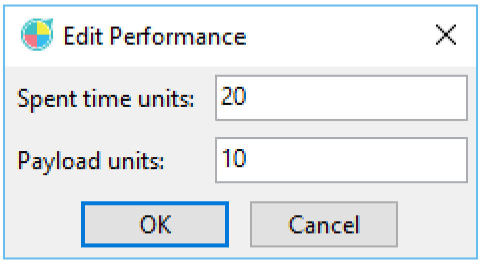
Time and payload units for symbols can be edited also via the *PragmaDev Performance Editor*. The editor can be opened from the project manager menu entry “Project / Edit performance properties...”:



Agent/symbol	Time	Payload
AccessControl.rdp		
SYS AccessControl		
bCentral		
pCentral		
idle		
checkCardAn		
administration		
deleteUser(u		
deleteUs	20	10
counter:		
del1		
cancelAdmin		
cancelA	1	2
idle		
addUser(use		
addUser		40
registere		2
accepted		
idle		
bLocal		

The editor will display (in a tree form) all SDL symbols found in diagrams with their corresponding time and payload properties. While the editor is opened, no modifications to the project and diagrams will be allowed.

The tree of symbols can be collapsed or expanded also via the “Edit” menu. Double-clicking on a symbol will trigger a dialog where time and payload units can be edited:

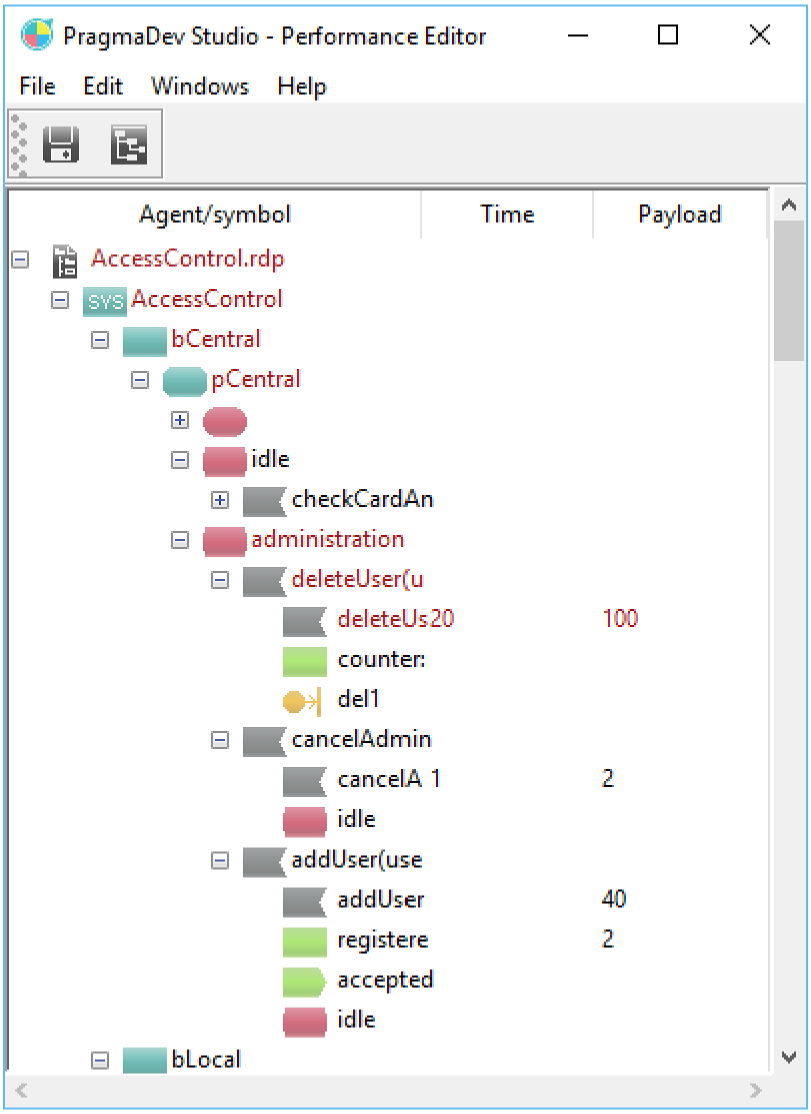


Edit Performance

Spent time units: 20

Payload units: 10

Any modifications to the properties of a symbol will be reflected in the tree by changing the color of that symbol and its ancestors:



PragmaDev Studio - Performance Editor

Agent/symbol	Time	Payload
AccessControl.rdp		
SYS AccessControl		
bCentral		
pCentral		
idle		
checkCardAn		
administration		
deleteUser(u		
deleteUs	20	100
counter:		
del1		
cancelAdmin		
cancelA	1	2
idle		
addUser(use		
addUser		40
registere		2
accepted		
idle		
bLocal		

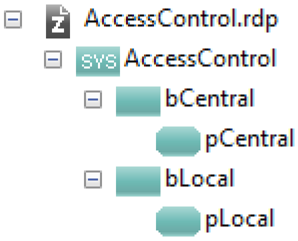
All modifications can be saved via the “File / Save” menu (or the 💾 button), or they can be discarded via the “File / Revert...” menu.

7.3.2.2 The architecture

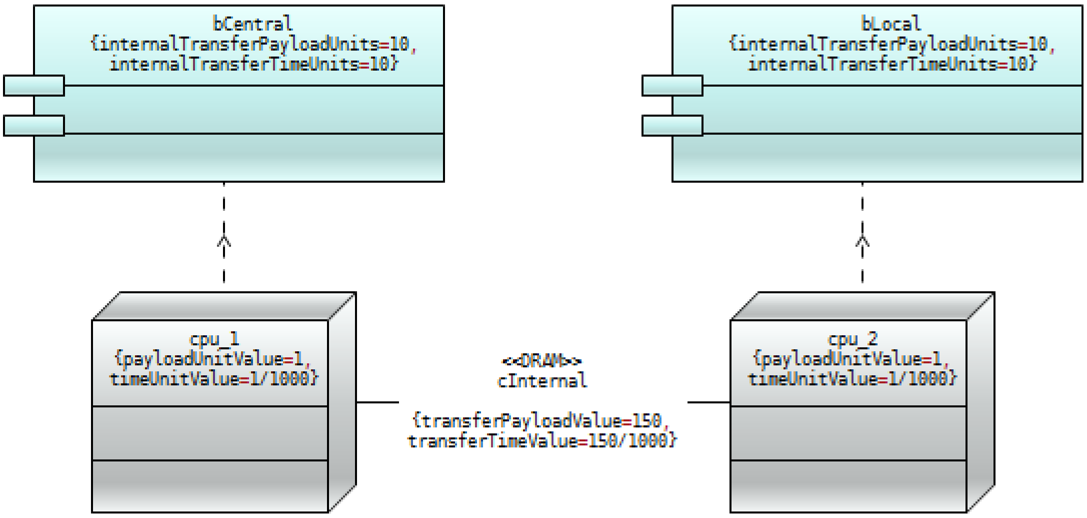
The possible allocations of the system are defined in deployment diagrams. The diagram will define:

- 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:



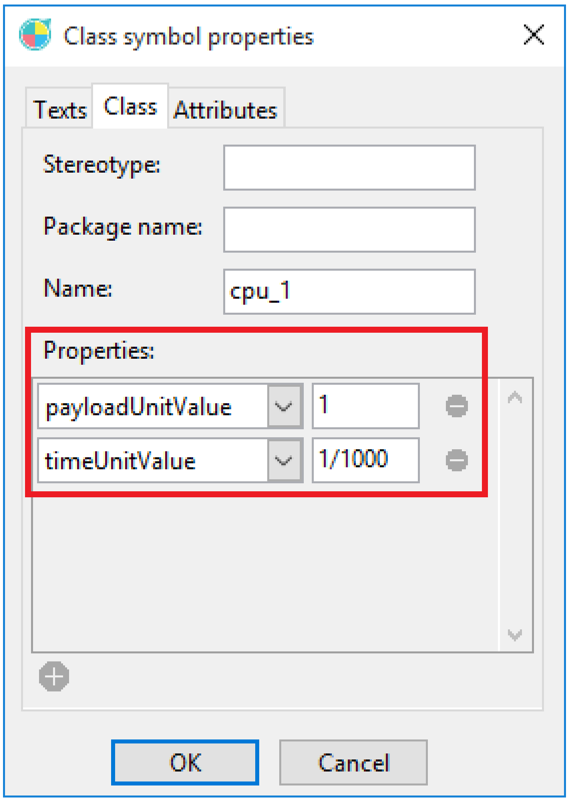
bCentral
{internalTransferPayloadUnits=10,
internalTransferTimeUnits=10}
bLocal
{internalTransferPayloadUnits=10,
internalTransferTimeUnits=10}
cpu_1
{payloadUnitValue=1,
timeUnitValue=1/1000}
<<DRAM>>
cInternal
{transferPayloadValue=150,
transferTimeValue=150/1000}
cpu_2
{payloadUnitValue=1,
timeUnitValue=1/1000}

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 expected values are positive integers; if not defined they will default to 0 (zero).

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:



Class symbol properties

Stereotype:

Package name:

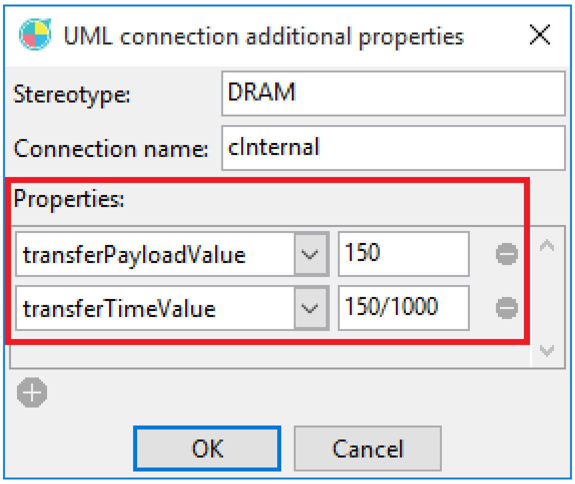
Name: cpu_1

Properties:

payloadUnitValue 1

timeUnitValue 1/1000

The same is true for the connection, where the properties can be entered in the link text or via its "Additional properties...":



UML connection additional properties

Stereotype: DRAM

Connection name: cInternal

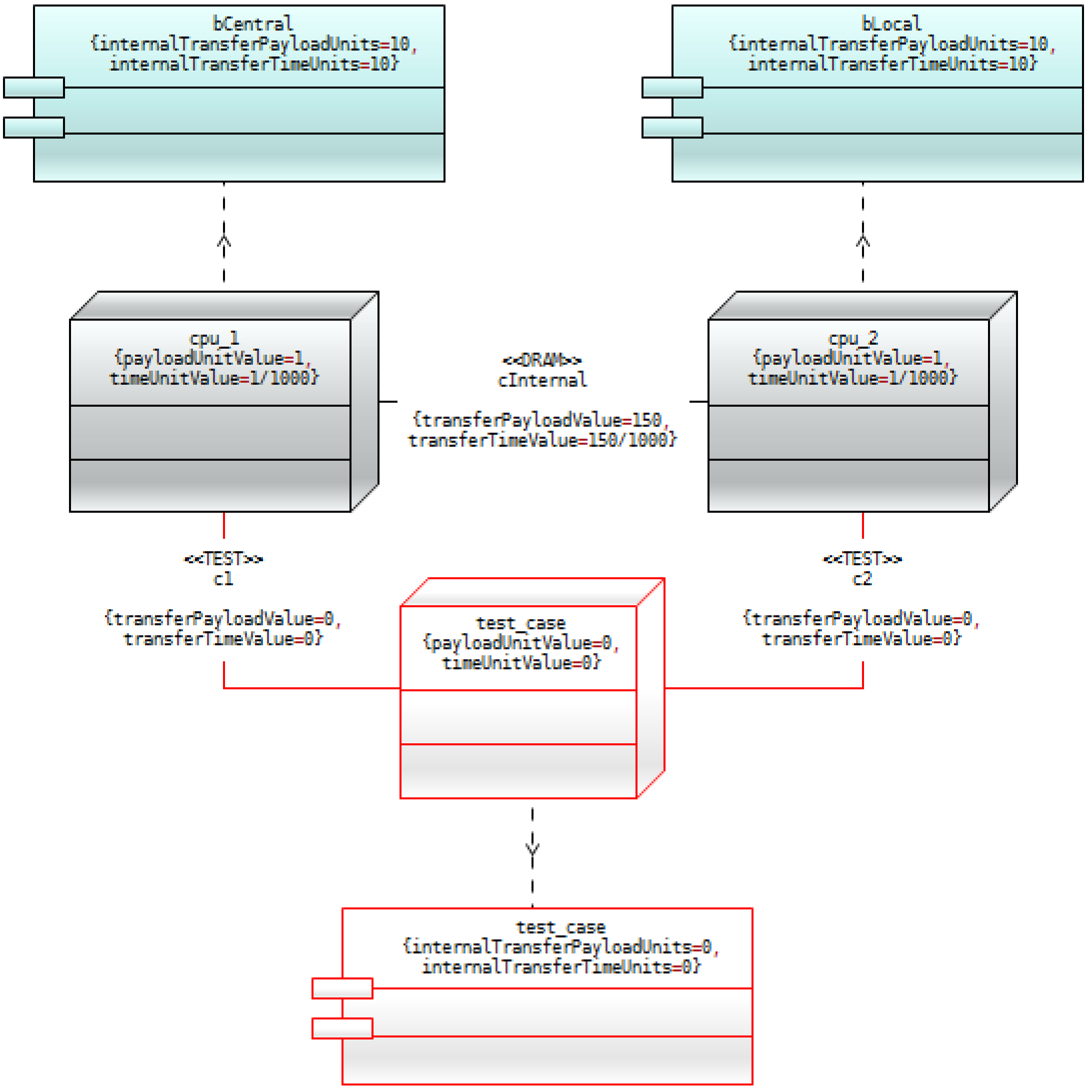
Properties:

transferPayloadValue 150

transferTimeValue 150/1000

7.3.2.3 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).



7.3.2.4 Semantics

7.3.2.4.1 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.

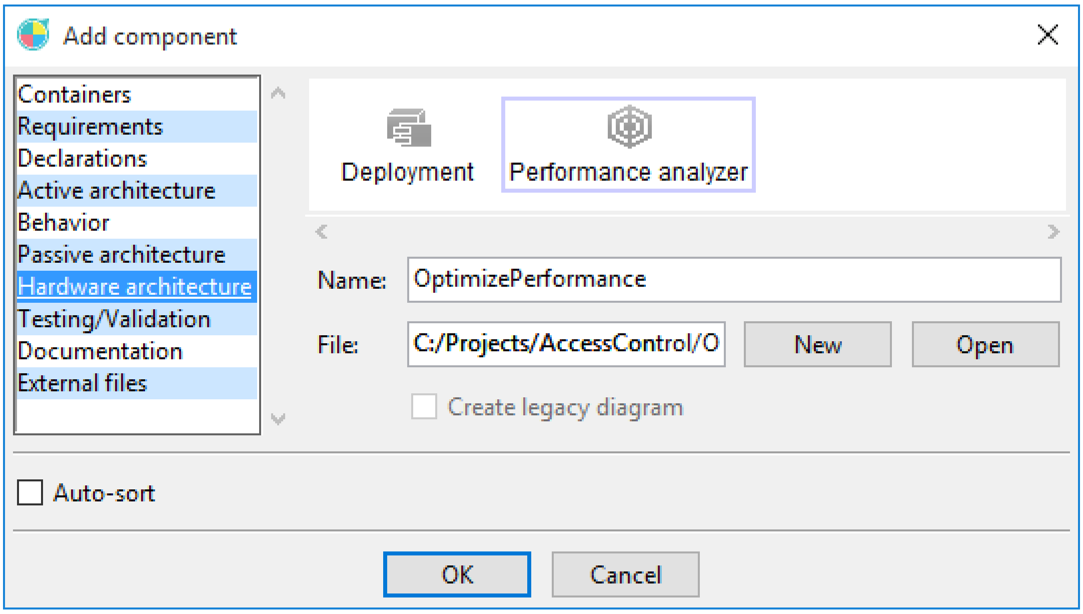
The resulting time the end of the analysis is the highest value among the clocks of all nodes, e.g., 100 in the example above.

7.3.2.4.2 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.

7.3.3 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 14):



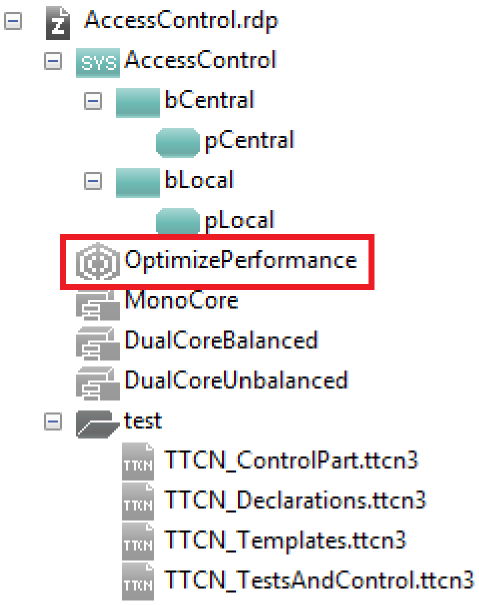
Name: OptimizePerformance

File: C:/Projects/AccessControl/O

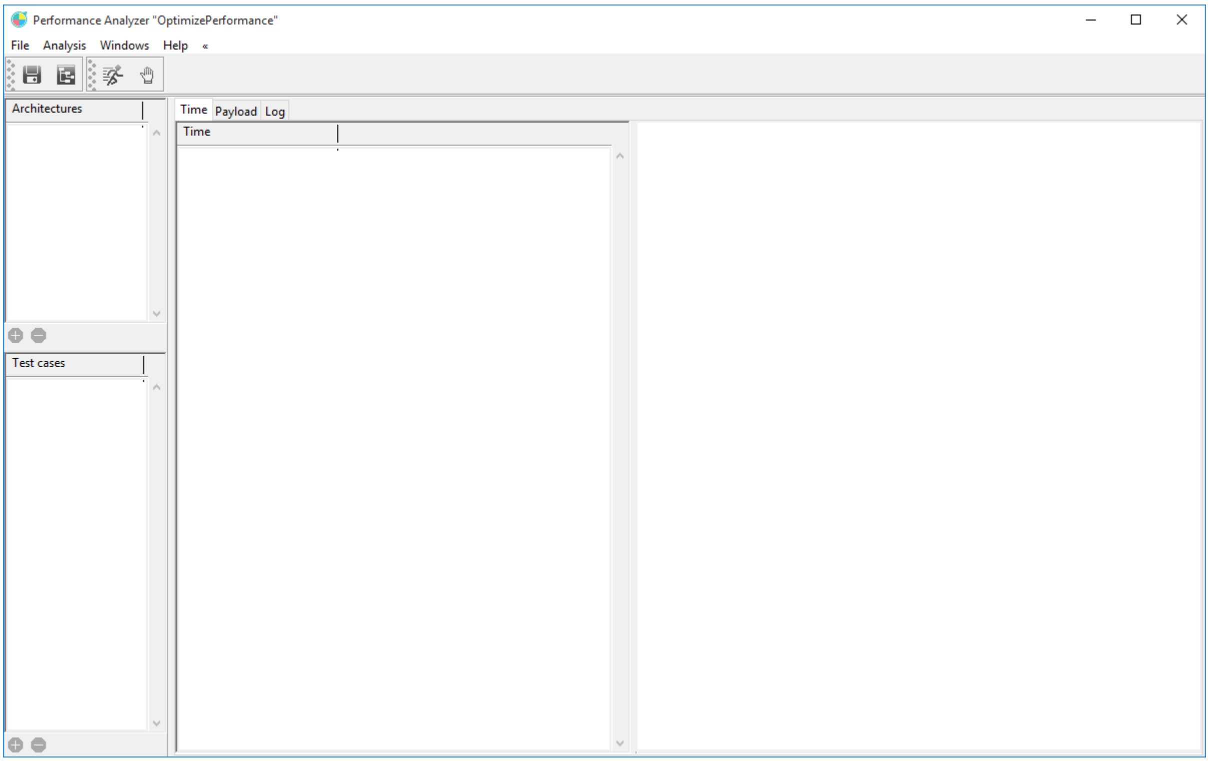
□ Create legacy diagram

□ Auto-sort

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:


Performance Analyzer "OptimizePerformance"
File Analysis Windows Help
Architectures
Test cases
Time Payload Log
Time

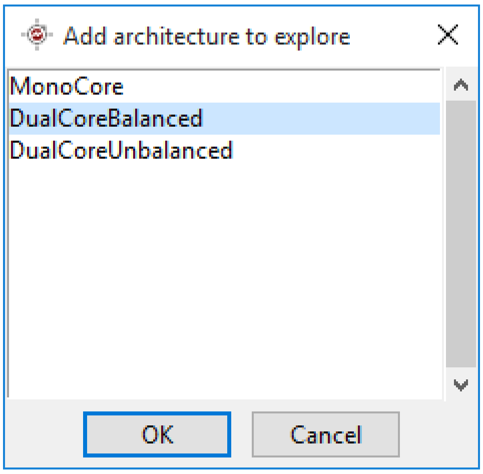
The user interface is divided in two parts:

- 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.

7.3.3.1 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:

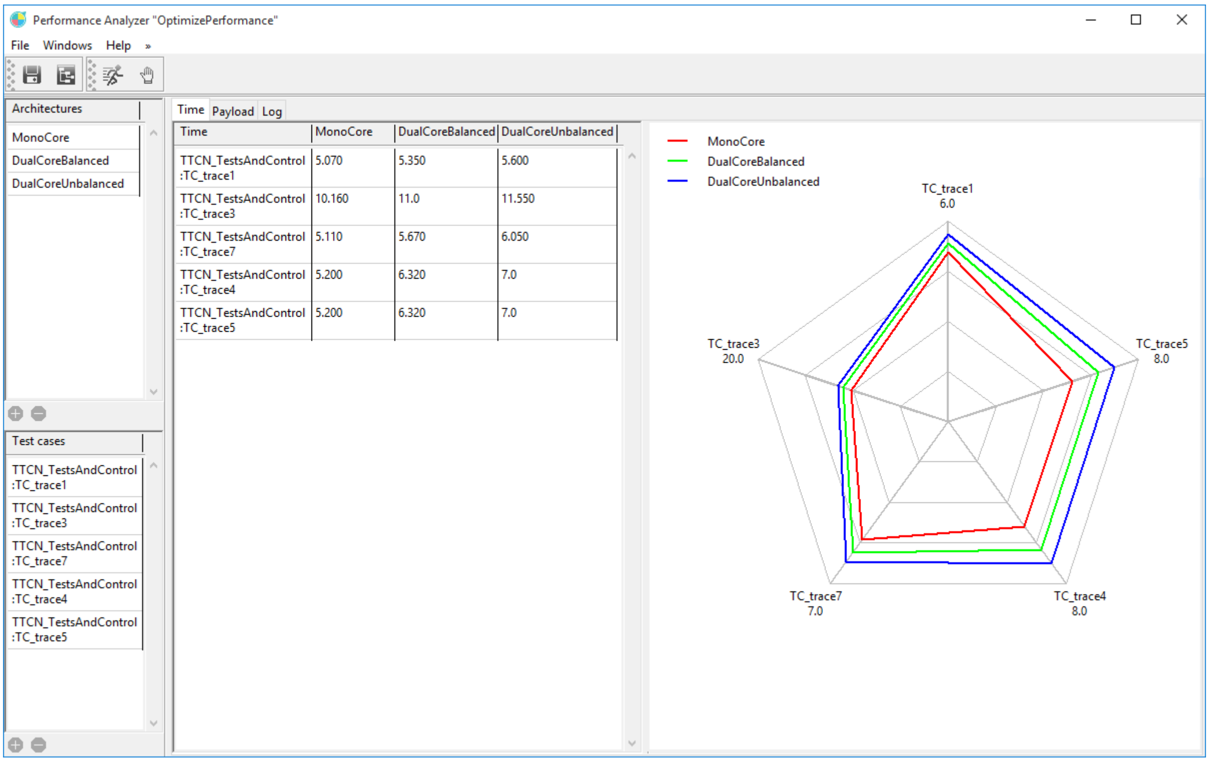

Add architecture to explore

MonoCore
DualCoreBalanced
DualCoreUnbalanced

This list is computed automatically based on the available deployment diagrams and TTCN-3 files present in the project tree.

7.3.3.2 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).



Time	MonoCore	DualCoreBalanced	DualCoreUnbalanced
TTCN_TestsAndControl:TC_trace1	5.070	5.350	5.600
TTCN_TestsAndControl:TC_trace3	10.160	11.0	11.550
TTCN_TestsAndControl:TC_trace7	5.110	5.670	6.050
TTCN_TestsAndControl:TC_trace4	5.200	6.320	7.0
TTCN_TestsAndControl:TC_trace5	5.200	6.320	7.0

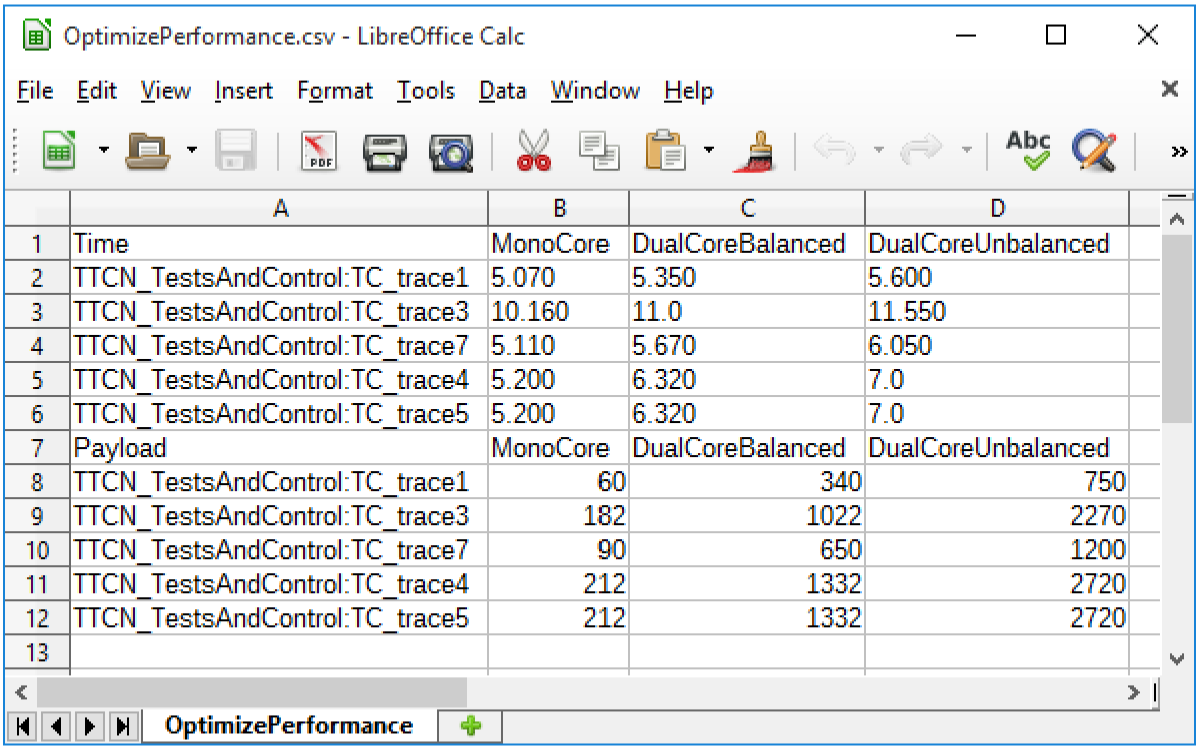
The result shown in the table and graph depends on the verdict of the test case:

- 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.

7.3.3.3 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...”.

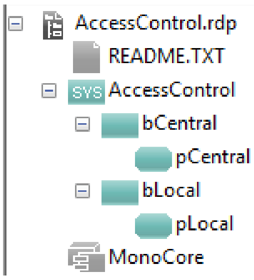
The Performance Analyzer provides also an export functionality that will save the results in CSV format. This functionality can be accessed via the menu “File / Export to CSV...”, and the resulting file can be then imported and edited with any spreadsheet application. Here is an example of the results imported in LibreOffice Calc:


OptimizePerformance.csv - LibreOffice Calc
File Edit View Insert Format Tools Data Window Help
OptimizePerformance

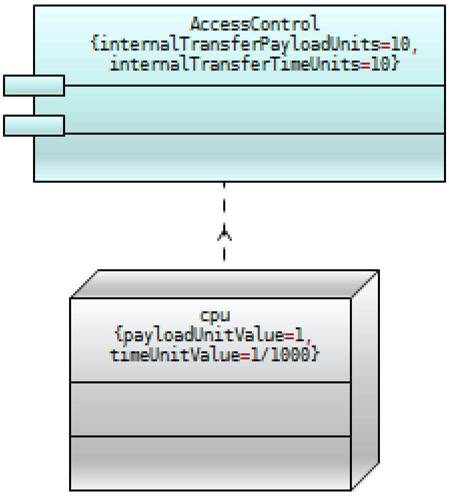
	A	B	C	D
1	Time	MonoCore	DualCoreBalanced	DualCoreUnbalanced
2	TTCN_TestsAndControl:TC_trace1	5.070	5.350	5.600
3	TTCN_TestsAndControl:TC_trace3	10.160	11.0	11.550
4	TTCN_TestsAndControl:TC_trace7	5.110	5.670	6.050
5	TTCN_TestsAndControl:TC_trace4	5.200	6.320	7.0
6	TTCN_TestsAndControl:TC_trace5	5.200	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				

7.3.4 SDL Z.100 performance simulation

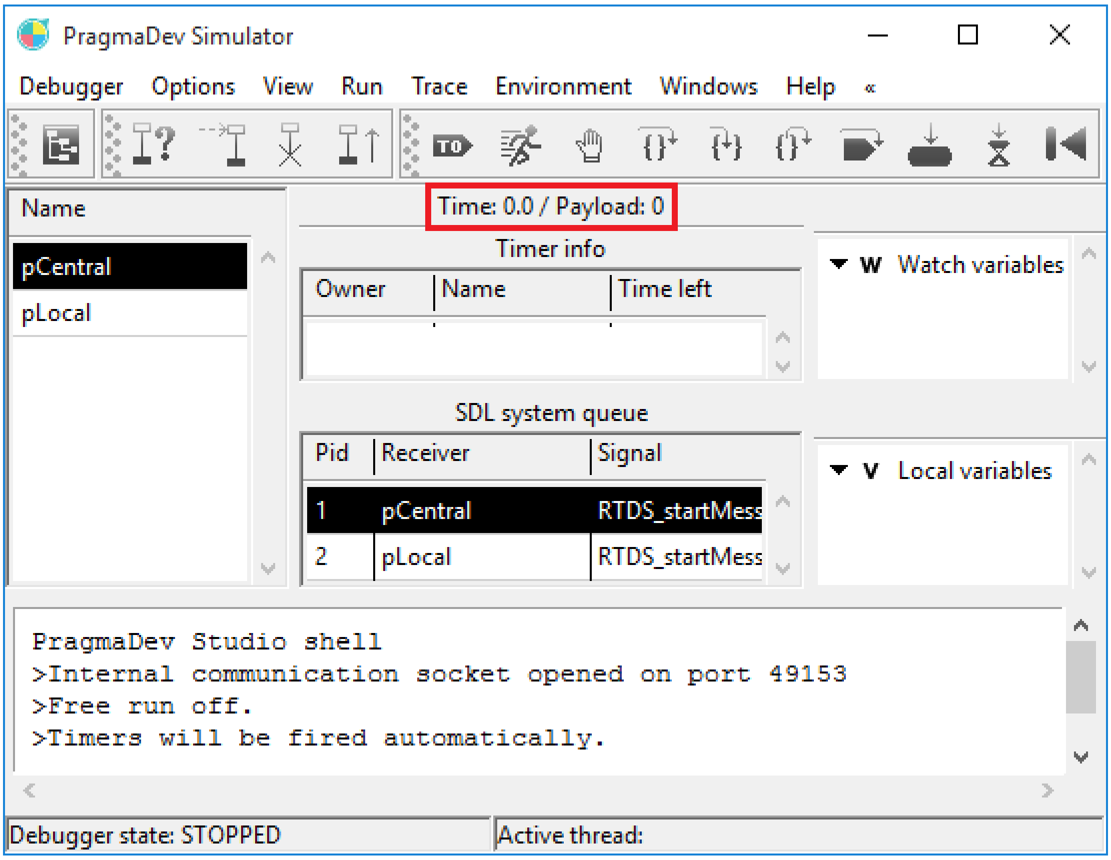
The core of *PragmaDev Performance Analyzer* is the *SDL Z.100 Simulator* with a specific *scheduler* (see “Simulator architecture” on page 186) that takes into account the time and payload properties in the model and deployment diagram. This allows the use of the *SDL Z.100 Simulator* with the *performance scheduler*. To trigger the use of such scheduler the simulator should be started on a deployment diagram:



Also, the selected deployment diagram should contain a single component (agent) executing on a single node:


AccessControl
{internalTransferPayloadUnits=10,
internalTransferTimeUnits=10}
cpu
{payloadUnitValue=1,
timeUnitValue=1/1000}

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:



Time: 0.0 / Payload: 0

Name
pCentral
pLocal

Timer info

Owner	Name	Time left

SDL system queue

Pid	Receiver	Signal
1	pCentral	RTDS_startMess
2	pLocal	RTDS_startMess

```
PragmaDev Studio shell
>Internal communication socket opened on port 49153
>Free run off.
>Timers will be fired automatically.
```

Debugger state: STOPPED

Active thread:

7.4 - 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 302.

To enable deployment simulation, in addition to the system being fully scheduled, the following are required:

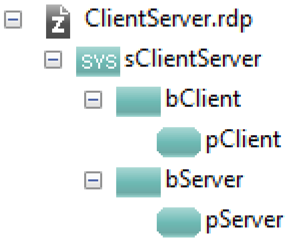
- Define how process instances will be deployed via a UML deployment diagram.
- Define a code generation profile that requires scheduling and supports deployment simulation.
- Make sure external messages coming from the environment are handled correctly.

The following paragraphs describe these points in detail.

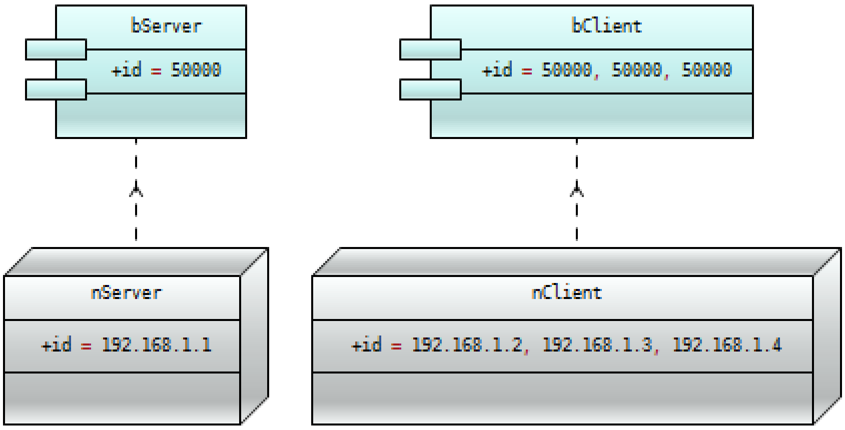
7.4.1 Deployment diagram for simulation

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:

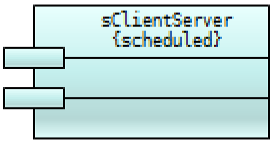


the components and nodes in the deployment diagram may be:


bServer
+id = 50000
bClient
+id = 50000, 50000, 50000
nServer
+id = 192.168.1.1
nClient
+id = 192.168.1.2, 192.168.1.3, 192.168.1.4

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 successfuly 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.

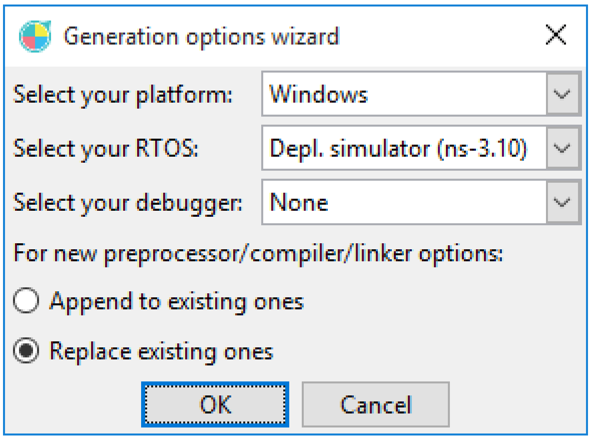
As deployment simulation requires always a fully scheduled system, by default each deployment diagram used for simulation implicitly includes a system agent component with the “scheduled” property. So, even if it is not present, the diagram above is considered to contain:



This automatically fulfills the requirement for such component as described in “Deployment diagram for scheduling policy” on page 302, 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.

7.4.2 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:



Generation options wizard

Select your platform: Windows

Select your RTOS: Depl. simulator (ns-3.10)

Select your debugger: None

For new preprocessor/compiler/linker options:

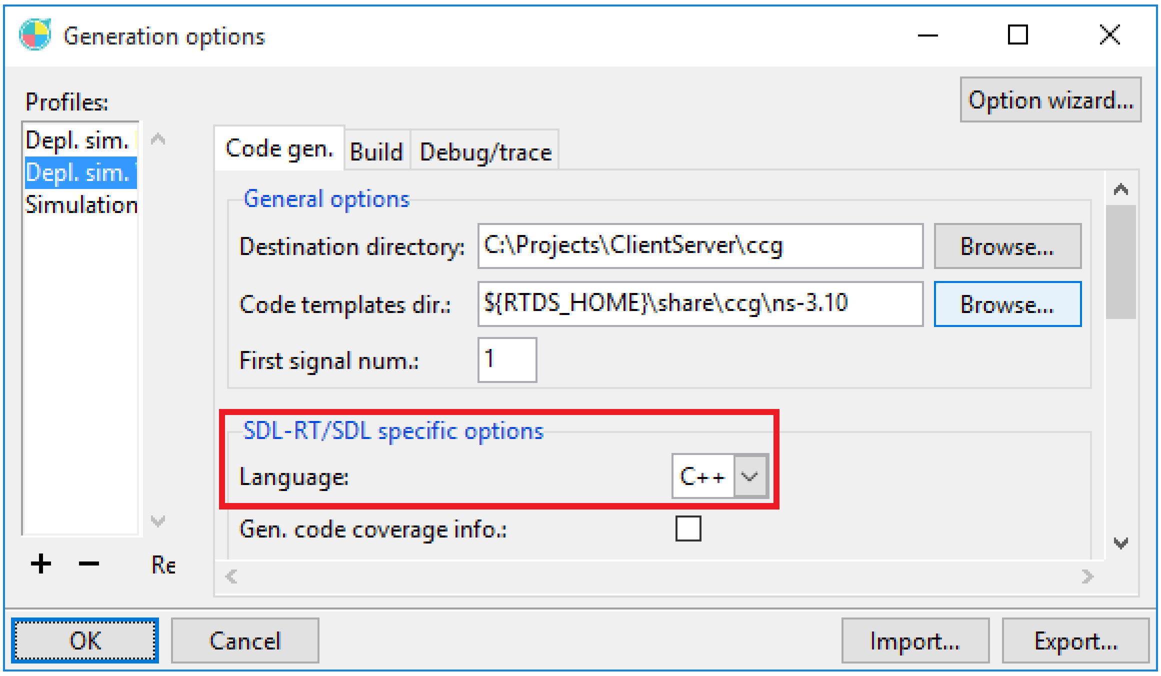
○ Append to existing ones

● Replace existing ones

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:



Generation options

Profiles:
Depl. sim.
Depl. sim.
Simulation

General options

Destination directory: C:\Projects\ClientServer\ccg

Code templates dir.: ${RTDS_HOME}\share\ccg\ns-3.10

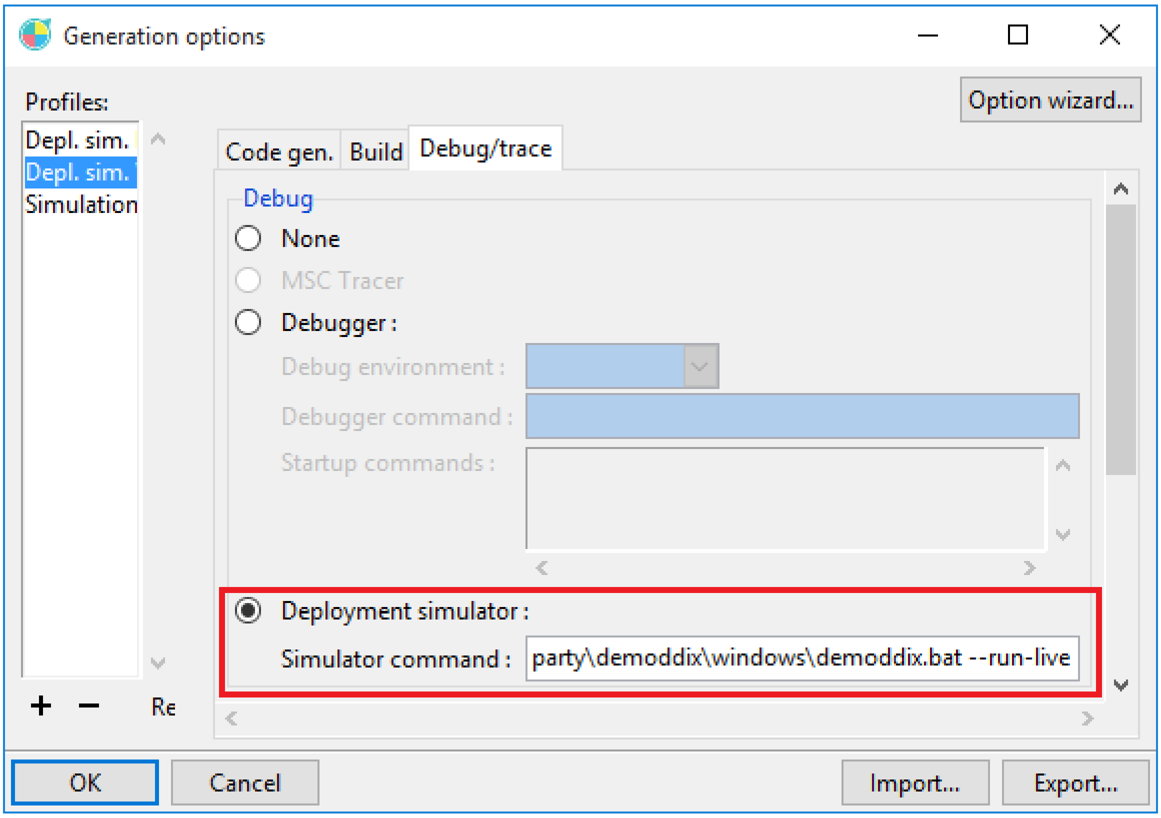
First signal num.: 1

SDL-RT/SDL specific options

Language: C++

Gen. code coverage info.: □

- The debugger must be set to “Deployment simulator” in the “Debug/trace” tab:



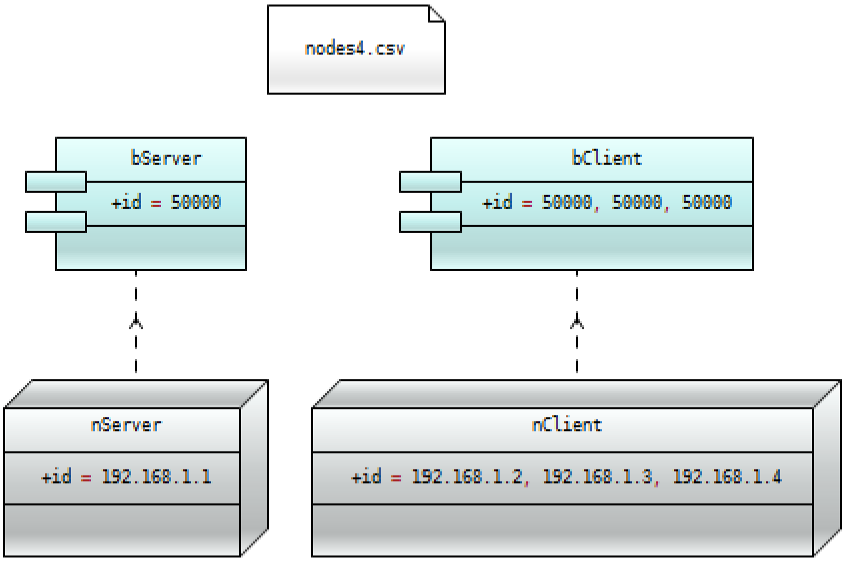
● Deployment simulator :

Simulator command : party\demoddix\windows\demoddix.bat --run-live

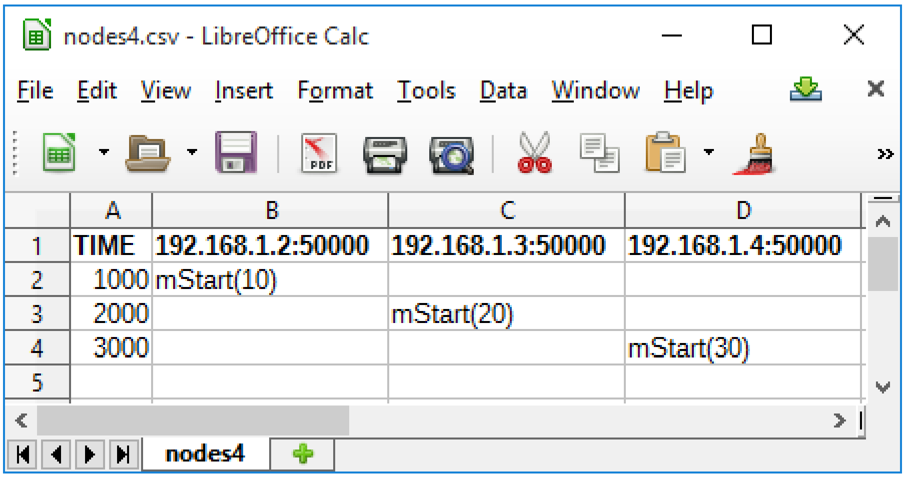
The generated code for deployment simulation will be similar to that of fully scheduled systems, however there are some important differences that ensure correct integration with the ns-3 simulator. For more details see the PragmaDev Studio Reference Manual.

7.4.3 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. Information about external messages can be included in the UML deployment diagram via a file symbol:


nodes4.csv
bServer
+id = 50000
bClient
+id = 50000, 50000, 50000
nServer
+id = 192.168.1.1
nClient
+id = 192.168.1.2, 192.168.1.3, 192.168.1.4

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:


nodes4.csv - LibreOffice Calc
File Edit View Insert Format Tools Data Window Help
nodes4

	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.

7.4.4 The deployment simulator

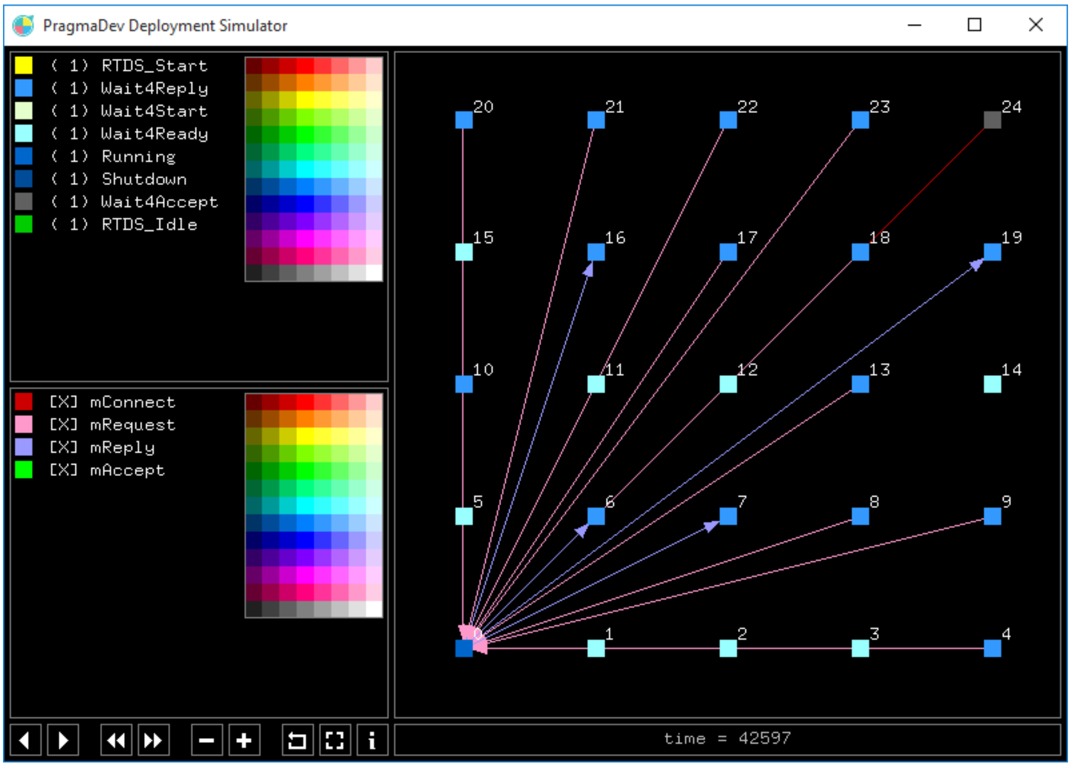
The deployment simulator is started from the “Generation / Execute” menu or the quick button. Once a profile for deployment simulation is selected, PragmaDev Studio will:

- 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.

7.4.4.1 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).

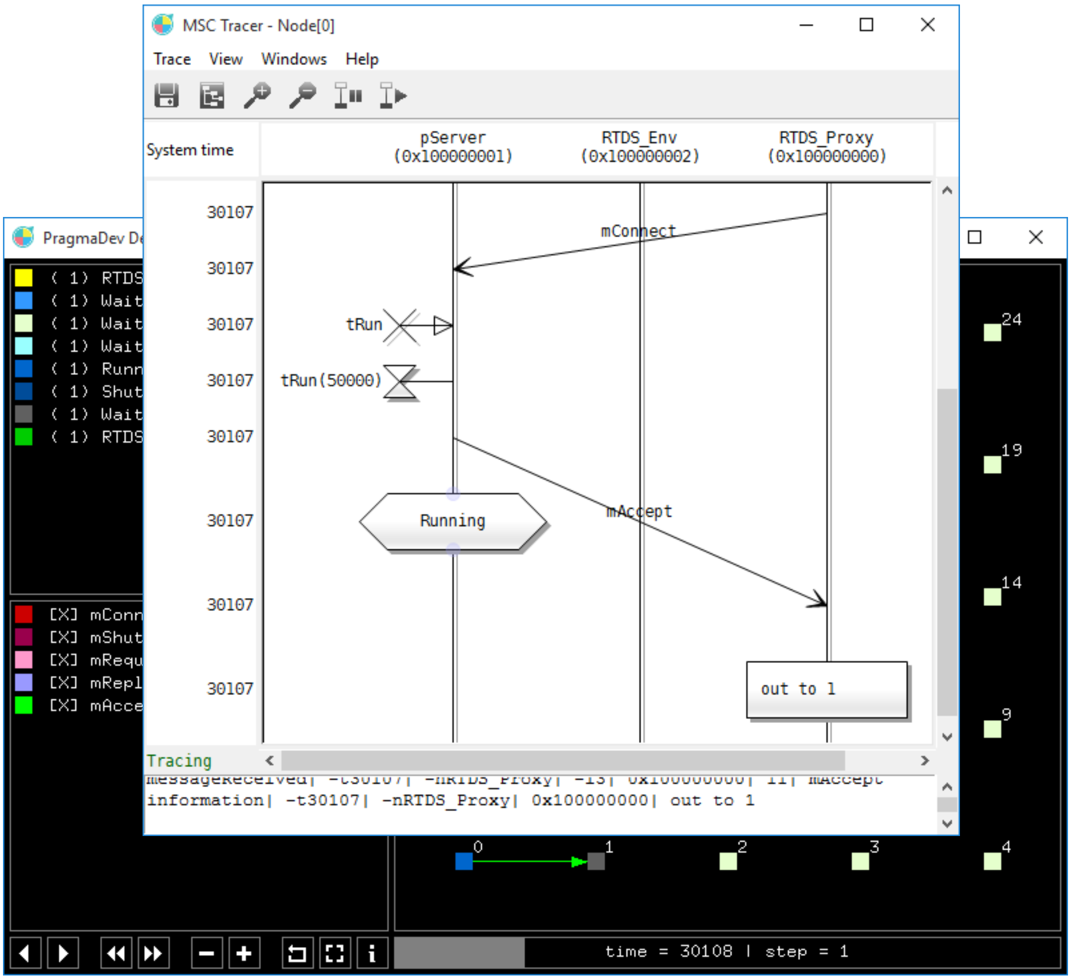
All nodes are displayed by default in a grid-like topology with an identifier assigned to them automatically. The color of each node represents the current state. This is the most recent state-change triggered by a process instance running on the node.

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 vizualization. A state-change will be displayed if the priority assigned to the new state is grater 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.

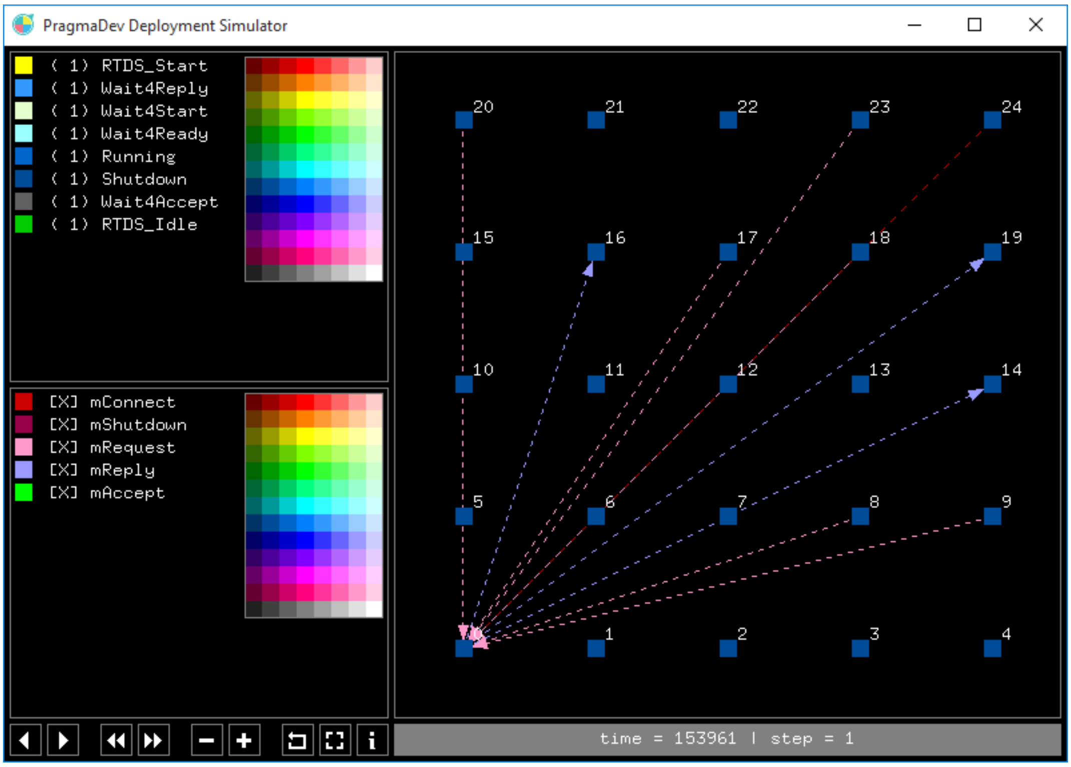
Holding down the *Crtl* key and *Left-Click*-ing on a node will launch an instance of the PragmaDev Tracer for displaying the events of all process instances running on the node.



```
messageReceived| -t30107| -nRTDS_Proxy| -13| 0x100000000| 11| mAccept
information| -t30107| -nRTDS_Proxy| 0x100000000| out to 1
```

7.4.4.2 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:



time = 153961 | step = 1

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.

7.4.4.3 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
  - i 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.

7.4.5 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.

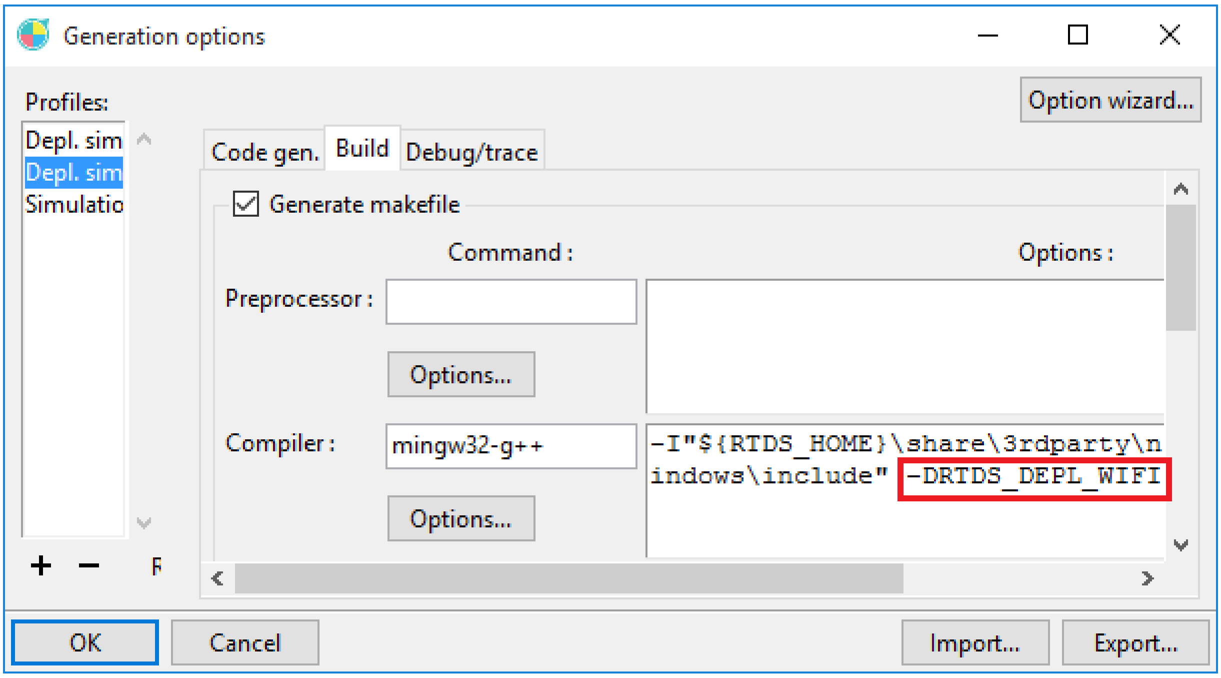
If the tool is launched from PragmaDev Studio, it should always be in live mode (*--run* or *--run-live*). This will ensure the execution and display of the generated simulation and not some other trace file that may have been produced during previous simulations.

7.4.6 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:



Generation options

Profiles:
- Depl. sim
- Depl. sim
- Simulatio

Build tab:

☑ Generate makefile

Preprocessor : (Command empty)

Compiler : mingw32-g++

Options :
```
-I"${RTDS_HOME}\share\3rdparty\n
indows\include" -DRTDS_DEPL_WIFI
```

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`.

7.5 - Model validation

7.5.1 Integration with OBP

7.5.1.1 Scope & features

PragmaDev Studio offers to validate a model by using OBP (*Observer Based Prover*), a requirement verification environment developed at ENSTA Bretagne. The specificity of this environment is that is does not rely on a transformation of the model into another language: the verification environment communicates with the modelling tool which is responsible for the support & execution of the target language. This eliminates the problems often encountered with other verification tools: the lack of expressivity of the target language, potentially causing some language features to be unsupported, and the change of semantics between common concepts, which can lead to a validation that covers too much or too little.

OBP communicates with PragmaDev Studio to actually execute the model in the SDL simulator. However, adaptations have been made to cover a range of semantics wider than the ones used in the simulator, to be able to get as close as possible to the final system behavior if it uses a code generator. See “Exploration options” on page 431.

The features available via OBP today are:

- Full coverage analysis of the model: this browses all the possible system states for the SDL model, allowing the identification of dead code. For this purpose, a code coverage analysis can be done on the results of an exploration, just as for a simulation or an execution (see “Code coverage results” on page 161).
- Error & deadlock analysis on the model: this browses the model until reaching a system state from which no transition can be executed. Simulation errors are internally described as special system states from which no transition can be triggered, allowing to identify them as deadlocks too. If an error or a deadlock is identified, a counter-example can be extracted, which can be either a MSC diagram showing the events causing the problem, or a simulator scenario that will allow to replay these events.
- Property verification: properties are described via PSC diagrams (see “MSC & PSC reference guide” on page 363), and can be checked during exploration. As in deadlock & error analysis, if a property is violated, a counter-example can be extracted either in MSC format, or as a simulator scenario.

OBP uses an exhaustive exploration technology, which can cause combinatory explosions or even infinite loops on some systems, especially when types used in system inputs are not sufficiently constrained. To workaround this problem, PragmaDev Studio allows to limit further the types of the incoming messages parameters in the system.

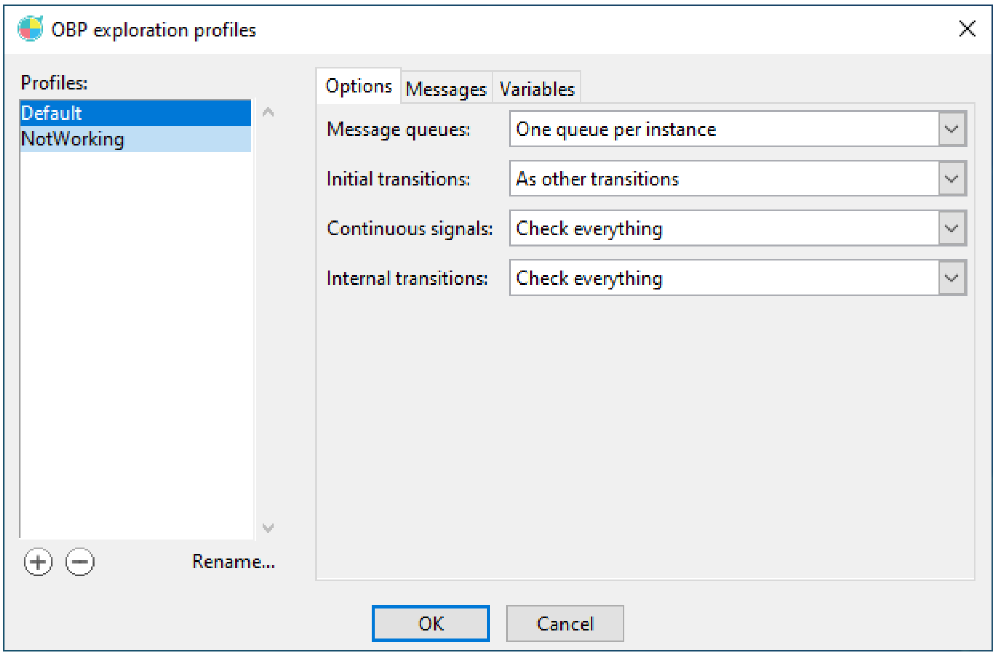
Another potential cause for combinatory explosion is the inclusion of unneeded temporary variables in the state of the system: if a variable is used only in a transition where it is set previously, it does not need to appear in the system state, and its presence can cause a lot of system states to be identified for nothing. PragmaDev Studio cannot identify automatically these variables yet, but allows to manually exclude them from the system state.

The limitation of message parameter values and the exclusion of variables are recorded in an exploration profile, that is described in detail in “OBP exploration profiles” on page 430.

OBP is included in the PragmaDev Studio distribution and does not need to be downloaded separately. Also note that the integration uses some specific OBP features, that might not be available in all OBP versions, so it is preferable to use the included version of OBP.

7.5.1.2 OBP exploration profiles

It is possible to define exploration profiles for OBP by selecting the ‘*[OBP] Exploration profiles...*’ entry in the ‘*Validation*’ menu. The following dialog opens:



Message queues: One queue per instance

Initial transitions: As other transitions

Continuous signals: Check everything

Internal transitions: Check everything

The ‘Profiles’ list on the left side lists the already created profiles. Note that there will always be one profile named ‘Default’ with all the default values for the exploration options and no message or variable limitations. The buttons under the list allow to create a new profile, optionally by copying the current one, to delete the current profile or to rename it.

The zone on the right side of the dialog gives the detail for the currently selected one:

- Exploration options;
- Message limitations;
- Variable limitations.

These are described in the following sections.

Note that the message & variable limitations depend on the agents in the project. If a change in the agents made one of the limitations obsolete, a warning message will be displayed at the bottom of the dialog.

7.5.1.2.1 Exploration options

The exploration options control how the execution will behave during exploration. The following options are available:

- '*Message queues*' controls the order in which the messages can be received by the process instances. The possible values are:
    - '*One queue per instance*': the execution simulates a message queue attached to each instance. This means that if a message `m1` is pending for `p1`, and a message `m2` is pending for `p2`, there are two possibilities for the exploration: `p1` receives `m1` first, or `p2` receives `m2` first. The order in which `m1` and `m2` have been sent is irrelevant. This simulates the behavior of the generated code on non-deterministic OS's. This is the default value.
    - '*Single queue for the whole system*': messages are always received in the order in which they were sent. So if an instance sends `m1` to `p1`, then another instance send `m2` to `p2`, there is only one possibility: `p1` receives `m1`, since `m1` has been sent first. This is the normal behavior for the SDL simulator.
- '*Initial transitions*' controls the priority of initial transitions compared to the other ones. The possible values are:
    - '*Prioritary*': this means that if an initial transition is pending, it will have priority over all other ones. If several initial transitions are pending, all combinations are tested. This is the normal behavior for the SDL simulator.
    - '*As other transitions*': this means that initial transitions are treated just like any other one. This is the default value.
- '*Continuous signals*' controls the semantics of continuous signals. The possible values are:
    - '*Pending messages have priority*': this enforces the SDL semantics, where continuous signals are not evaluated if the instance has pending messages.
    - '*Check everything*': this means whether the instance has a pending message or not is irrelevant, the continuous signal will be tested in all cases, as well as the pending message. This is introduced because the SDL semantics is often difficult to implement in RTOS's, where knowing if a task has a pending message can be impossible. This is the default value.
- '*Internal transitions*' controls the priority of internal transitions compared to transitions triggered by messages coming from the environment. The possible values are:
    - '*Prioritary over external ones*': this means that no message coming from the environment will be considered unless there are no pending messages left in the system. This replicates the behavior of the system when the option '*Treat internal messages before external ones*' is checked in the simulation options (see “Main simulator options” on page 187).
    - '*Check everything*': this means that internal and external transitions are considered the same way, so an incoming message from the environment can be accepted even if there are pending messages in the system. Note however that messages coming from the environment are considered only if an instance is in the proper state to handle them. This is the default value.

7.5.1.2.2 Message limitations

Message limitations are specified in the 'Messages' tab in the exploration profile. All incoming messages for the system in the project are displayed in the tab, with for each message:

- A field allowing to specify the maximum number of messages of this type that can be sent to the system. This allows to limit the exploration for systems that are infinite loops waiting for an incoming message.
- A field for each of the message parameters, allowing to specify constraints on the values for this parameter.

Constraints are possible values for the parameter separated with '`|`' or '`,`' characters (existing constraints always use the '`|`' character). For example, if the parameter `param1` is of type `CharString`, a constraint set to '`foo|bar`' will only consider the values '`foo`' & '`bar`' for this parameter when accepting a message from the environment. If the parameter type is based on an integer, it is also possible to specify a range of values by using the syntax '`min..max`'. For example, a constraint set to '`0..3`' is the same as the constraint '`0|1|2|3`'.

Note that the SDL constraints on the type for the message parameter are also considered, so care must be taken to make sure a constraint is compatible with the type. For example, if a message has a type defined as the following syntype:

```
syntype MyParamType = Integer
  constraints 0..3;
endsyntype;
```

and the constraint set in the exploration profile is '`4|6`', the resulting set of possible values for the parameter is empty. This will be identified, and the exploration will not be launched in such a case.

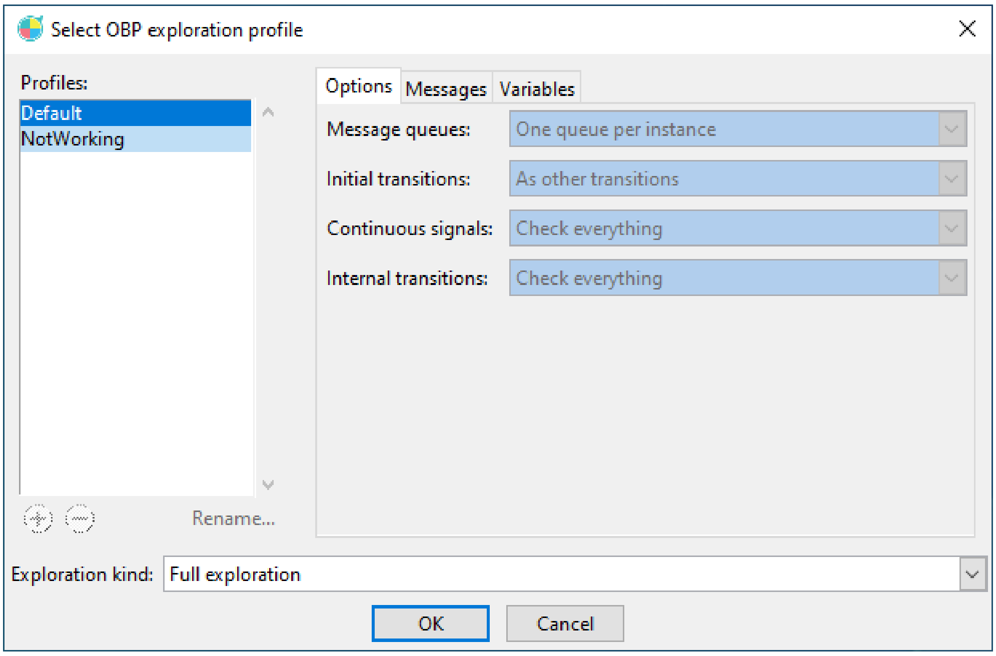
7.5.1.2.3 Variable limitations

Variable limitations are specified in the 'Variables' tab in the exploration profile. All behavioral agents in the system in the project are displayed in the tab, with for each agent a list of its variables associated to a selector with 3 choices:

- "*No*" means that the variable must not be included in the system state;
- "*Unspecified*" means that PragmaDev Studio must figure out if the variable must be included by itself. Note that this is not supported today, and means that the variable will be included in the system state.
- "*Yes*" means that the variable must be included in the system state. Note that today, this choice is the same as "Unspecified".

7.5.1.3 Running an OBP exploration

Once an exploration profile has been defined, the exploration can be launches by selecting ‘*[OBP] Run exploration...*’ in the ‘*Validation*’ menu. The profile selection dialog appears:


Select OBP exploration profile

Profiles:
Default
NotWorking

Message queues: One queue per instance
Initial transitions: As other transitions
Continuous signals: Check everything
Internal transitions: Check everything

Exploration kind: Full exploration

The layout is the same as the profile definition dialog, except the options are not modifiable and profiles cannot be added, deleted or renamed. This allows to check the options defined in the profile to use before running the exploration.

The ‘*Exploration kind*’ field at the bottom allows to select the type of exploration to perform:

- ‘*Full exploration*’ will run a full exploration, with an optional code coverage extraction in the end.
- ‘*Check for deadlock & errors*’ will try to identify deadlocks and errors in the system, and stop as soon as one is encountered, with the possibility of extracting a counter-example in the end.
- ‘*Check property <diagram name>*’ will check the system to figure out if the property described as a PSC in the diagram named <diagram name> is verified or not. If the property is violated, a counter-example can be extracted at the end of the exploration.

For the property check, please note that PSC diagrams cannot be distinguished from regular MSC diagrams, so all MSC diagrams will be listed as properties. It is up to the user to check that the diagram actually contains a PSC property.

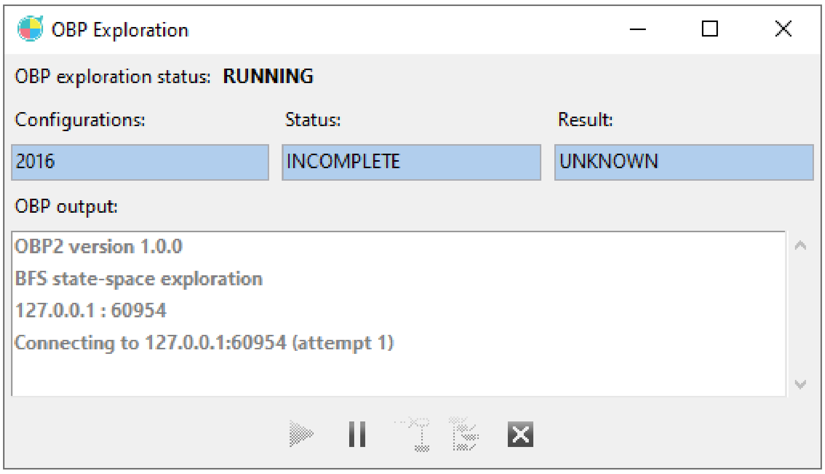
Also note that all PSC features are not supported for property checking with OBP. The supported features are:

- Regular, required and failed messages (prefixes ‘`e:`’, ‘`r:`’ & ‘`f:`’);
- Strict operators;

- Inline expressions of type ‘`alt`’;
- Past negative chain constraints, i.e sequences of messages between ‘[’ and ‘]’ at the beginning of a message text and prefixed with ‘=\=>’.

Future chain constraints (positive or negative) and past positive chain constraints are not supported, as well as any other type of inline expressions. They will trigger an error before the exploration, which will not run.

Clicking ‘OK’ in the profile selection dialog runs the exploration. The progress is displayed in the following dialog:



OBP Exploration

OBP exploration status: **RUNNING**

Configurations:	Status:	Result:
2016	INCOMPLETE	UNKNOWN

OBP output:

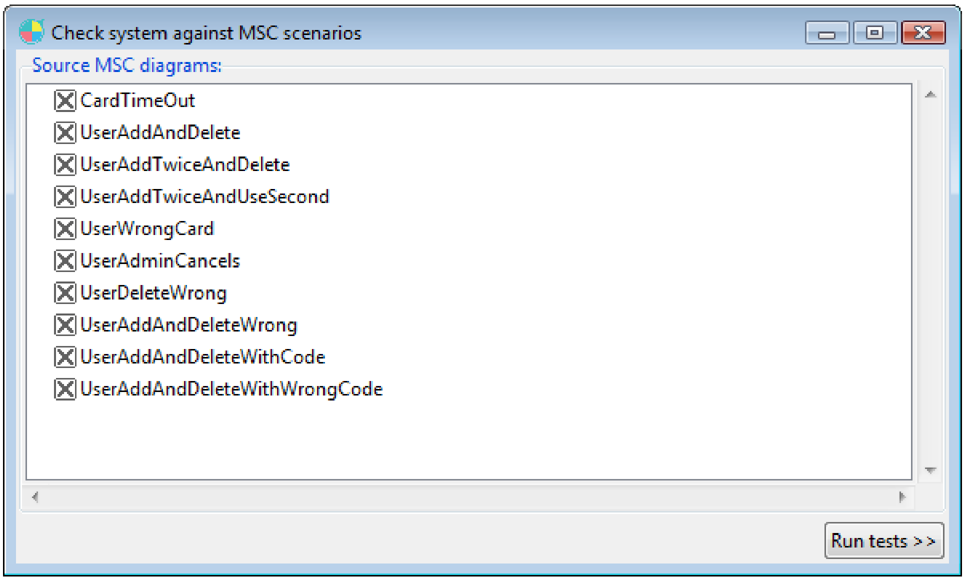
```
OBP2 version 1.0.0
BFS state-space exploration
127.0.0.1 : 60954
Connecting to 127.0.0.1:60954 (attempt 1)
```

Once the exploration is complete, the ‘*OBP exploration status*’ will be set to ‘*COMPLETE*’. Depending on the type of exploration, the buttons at the bottom of the dialog will be activated:

- extracts the code coverage information for the exploration and displays it in a regular code coverage results viewer as described in “Code coverage results” on page 161. This is only available for full explorations.
- is activated only if the ‘*Result*’ field is ‘*VIOLATED*’. It displays a menu allowing to select the format for the counter-example, then extracts it. The available formats are:
  - ‘*As MSC*’: the counter-example is extracted as a MSC, which will appear in a standard MSC tracer window (see “Tracer window” on page 383);
  - ‘As simulator scenario’: the counter-example is extracted as a scenario of commands for the SDL simulator, as described in “Shell” on page 202. The destination file for the scenario is selected via a standard file dialog. This scenario can be replayed via the ▶ button in the simulator window.

7.5.2 Checking a system against MSC scenarios

PragmaDev Studio offers a way to check the behavior of a new version of the system, provided you have a set of scenarios described as MSC diagrams that were obtained with a previous version, for example via manual testing using the tracer. It will check that the scenarios are still executing correctly and how much of the model they cover. This feature is available in the project manager's “*Validation*” menu, via the “*Check system against MSC scenarios...*” entry. When selecting the system to test and launching the analysis, the following dialog will appear:

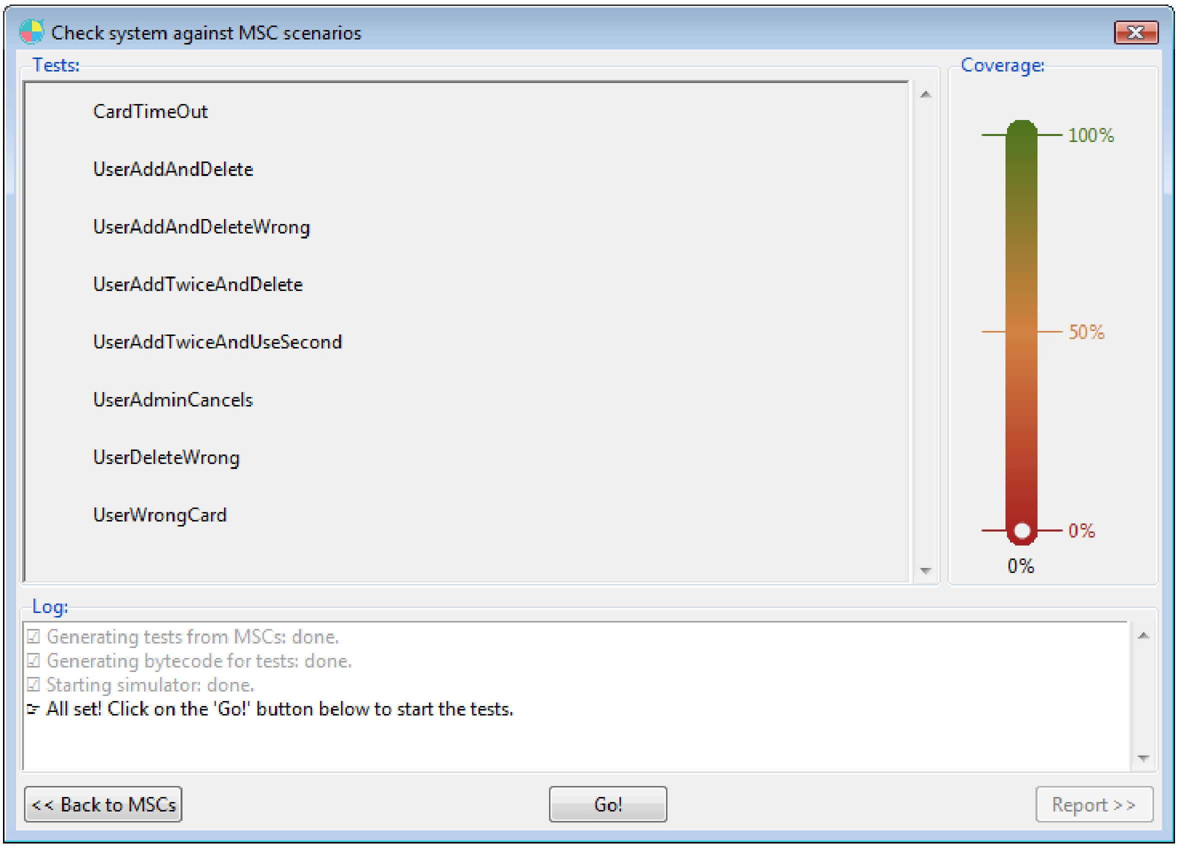


Check system against MSC scenarios

Source MSC diagrams:

☒ CardTimeOut
☒ UserAddAndDelete
☒ UserAddTwiceAndDelete
☒ UserAddTwiceAndUseSecond
☒ UserWrongCard
☒ UserAdminCancels
☒ UserDeleteWrong
☒ UserAddAndDeleteWrong
☒ UserAddAndDeleteWithCode
☒ UserAddAndDeleteWithWrongCode

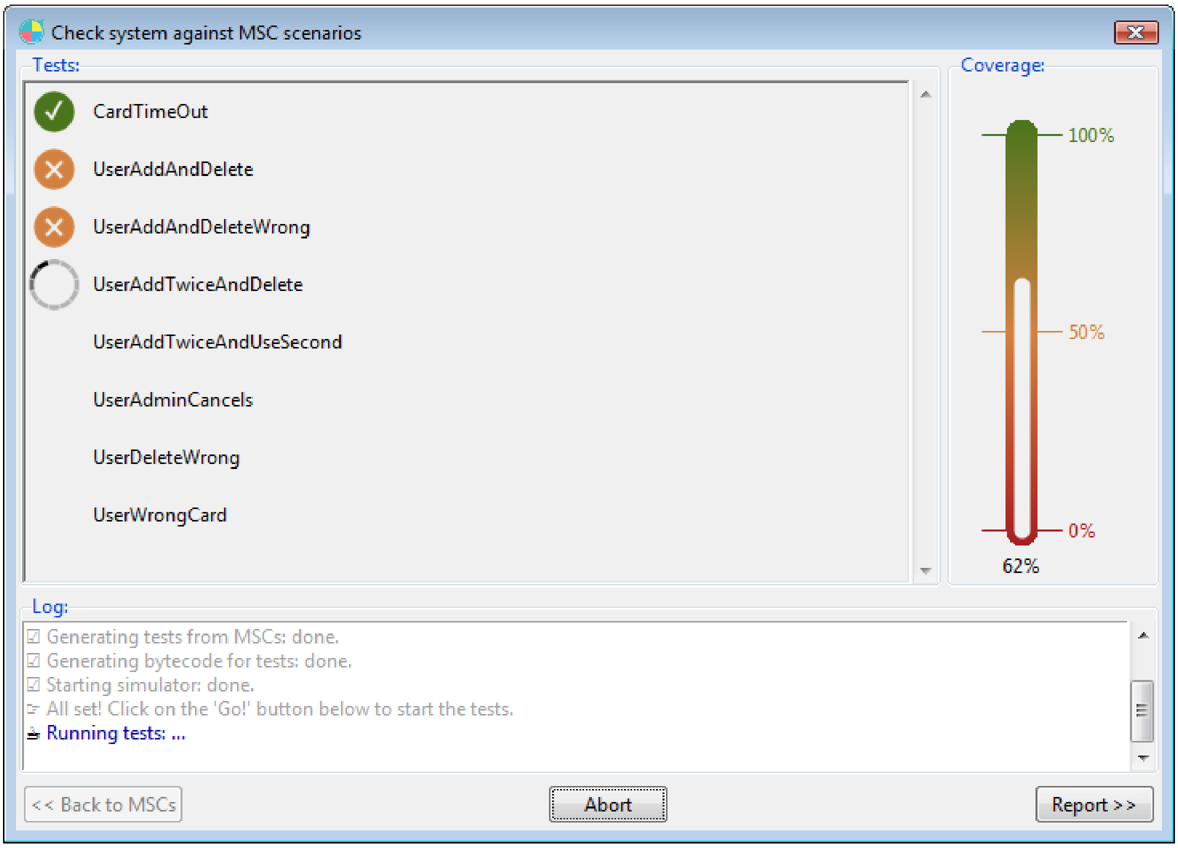
Here are listed all MSC diagrams that are present in the project. Only those that represent a valid execution scenario should be selected. Once this is done, pressing the "*Run tests >>*" button will display the next panel in the dialog:



```
☑ Generating tests from MSCs: done.
☑ Generating bytecode for tests: done.
☑ Starting simulator: done.
☞ All set! Click on the 'Go!' button below to start the tests.
```

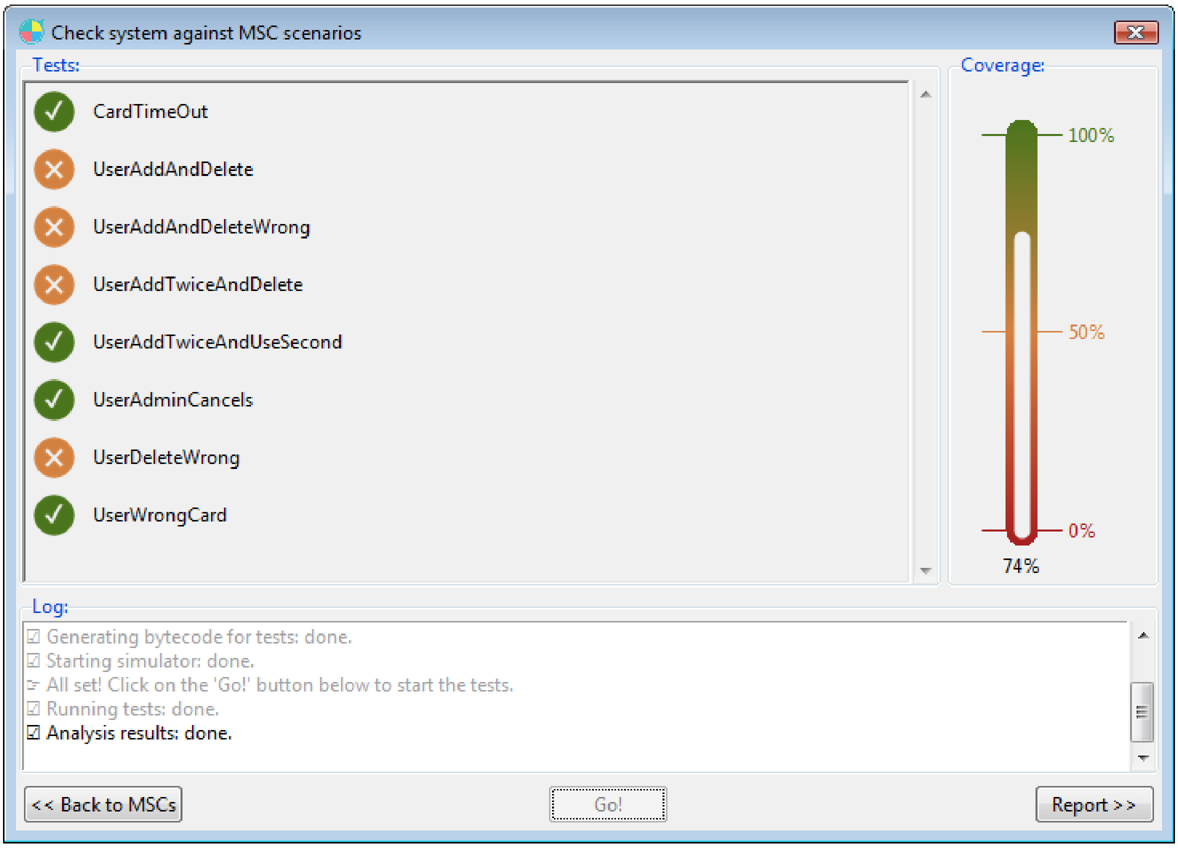
Note: the operations indicated in the panel's notification zone ("*Generating tests from MSCs*", "*Generating bytecode for tests*" & "*Starting simulator*") might take a while to be completed.

This is the scenario execution panel. Once the message "*All set! Click the 'Go!' button to start the tests*" is displayed, the "*Go!*" button can be pressed, which will execute the tests listed in the upper left zone one by one. While the execution is running, the model coverage will be computed and displayed via the slider on the upper right:



```
☑ Generating tests from MSCs: done.
☑ Generating bytecode for tests: done.
☑ Starting simulator: done.
☞ All set! Click on the 'Go!' button below to start the tests.
Running tests: ...
```

Everything is automatic until all the tests are executed. The final dialog will show all the verdicts for the tests, as well as the total model coverage reached:



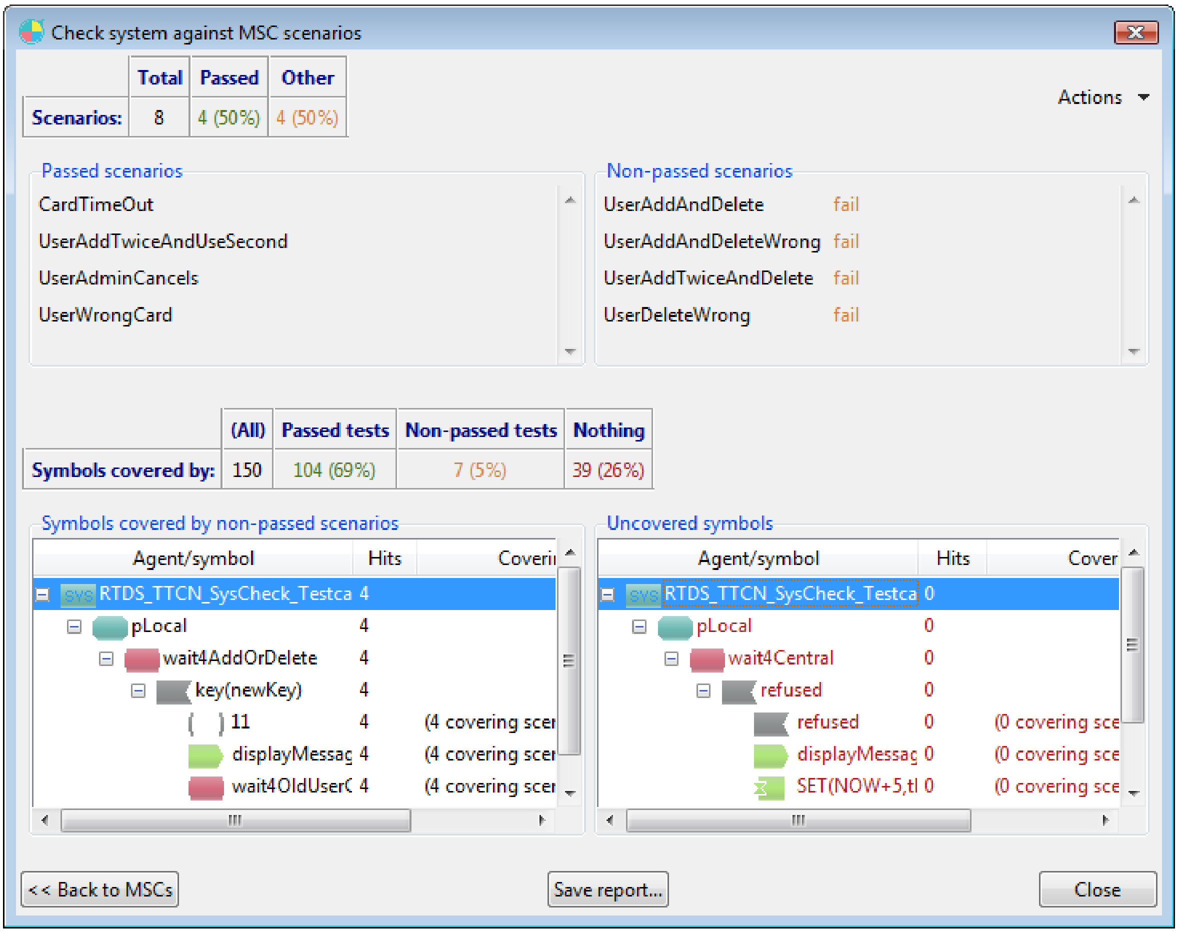
- ✓ CardTimeOut
- ✗ UserAddAndDelete
- ✗ UserAddAndDeleteWrong
- ✗ UserAddTwiceAndDelete
- ✓ UserAddTwiceAndUseSecond
- ✓ UserAdminCancels
- ✗ UserDeleteWrong
- ✓ UserWrongCard

```
☑ Generating bytecode for tests: done.
☑ Starting simulator: done.
☞ All set! Click on the 'Go!' button below to start the tests.
☑ Running tests: done.
☑ Analysis results: done.
```

A green check mark means the test has passed; an orange X means it has failed for some reason. Hovering over the X will display the actual verdict as a tooltip. The verdicts are those used by TTCN testcases.

After all the tests have been run, a report is generated that can be accessed via the "*Report >>*" button in the dialog. It is also possible to get back to the MSC selection panel to change the set of scenarios; note that it will discard the execution results for the current set of tests.

The report panel looks like follows:



Check system against MSC scenarios

	Total	Passed	Other
Scenarios:	8	4 (50%)	4 (50%)

Passed scenarios

- CardTimeOut
- UserAddTwiceAndUseSecond
- UserAdminCancels
- UserWrongCard

Non-passed scenarios

UserAddAndDelete	fail
UserAddAndDeleteWrong	fail
UserAddTwiceAndDelete	fail
UserDeleteWrong	fail

	(All)	Passed tests	Non-passed tests	Nothing
Symbols covered by:	150	104 (69%)	7 (5%)	39 (26%)

Symbols covered by non-passed scenarios

Agent/symbol	Hits	Coveri
RTDS_TTCN_SysCheck_Testca	4	
pLocal	4	
wait4AddOrDelete	4	
key(newKey)	4	
11	4	(4 covering scer
displayMessag	4	(4 covering scer
wait4OldUser(	4	(4 covering scer

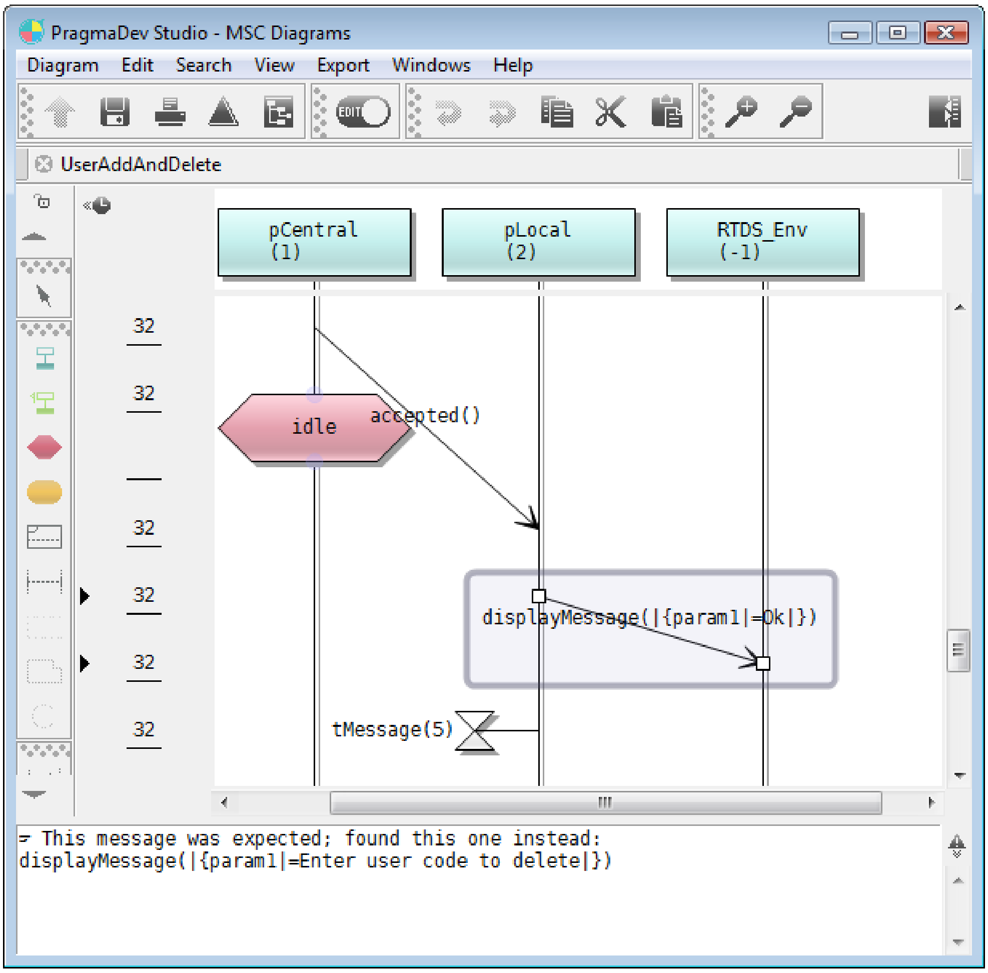
Uncovered symbols

Agent/symbol	Hits	Cover
RTDS_TTCN_SysCheck_Testca	0	
pLocal	0	
wait4Central	0	
refused	0	
refused	0	(0 covering sce
displayMessag	0	(0 covering sce
SET(NOW+5,tl	0	(0 covering sce

The information displayed in this panel is:

- On the upper left corner, the number of executed scenarios, the number of scenarios that worked as expected (“*passed*”) and the number of scenarios that didn’t, each with a percentage compared to the total number.
- Below are two tables listing the passed scenarios, and the non-passed ones, with their actual verdict (in the example above, all have a “`fail`” verdict).

Double-clicking on one of the non-passed scenarios will open the corresponding diagram and show exactly where the problem has occured:

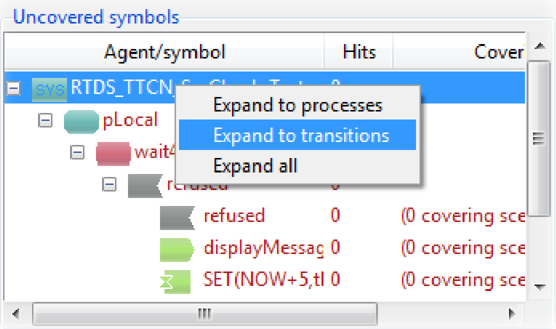

PragmaDev Studio - MSC Diagrams
Diagram Edit Search View Export Windows Help
UserAddAndDelete
pCentral (1)
pLocal (2)
RTDS_Env (-1)
32
idle
accepted()
displayMessage(|{param1|=Ok|})
tMessage(5)

```
This message was expected; found this one instead:
displayMessage(|{param1|=Enter user code to delete|})
```

The expected message is selected, and the notification zone displays the one that has happened instead. The message is usually a message sent from the system to the environment.

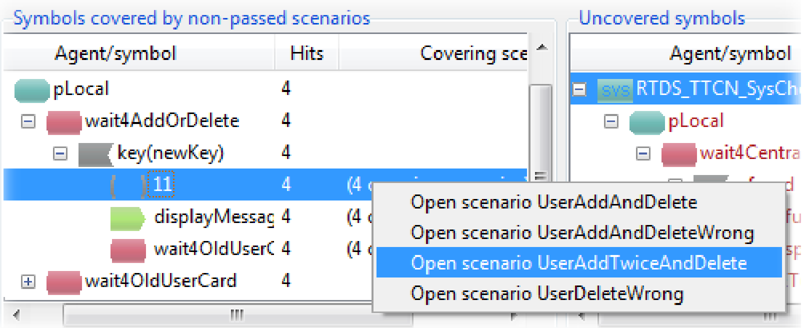
- The zone below the lists of scenarios in the report panel displays a summary of the model coverage:
  - The total number of symbols in the system;
  - The number and percentage of symbols covered by passed scenarios only;
  - The number and percentage of symbols covered only by scenarios that ended up having a non-passed verdict;
  - The number and percentage of symbols that weren’t covered by any scenario at all.
- The two trees at the bottom of the panel display the detail of symbols covered only by non-passed scenarios (on the left) and not covered at all (on the right). These trees are actually model coverage trees, as described in “Code coverage

results” on page 161. The same features are available, such as expanding the tree to a given level by right-clicking on the root node:



Agent/symbol	Hits	Cover
RTDS_TTCN		
pLocal		
wait4		
refused		
refused	0	(0 covering sce
displayMessag	0	(0 covering sce
SET(NOW+5,tI	0	(0 covering sce

or opening the covering tests for a given symbol by right-clicking on the covering tests count in the tree’s third column:

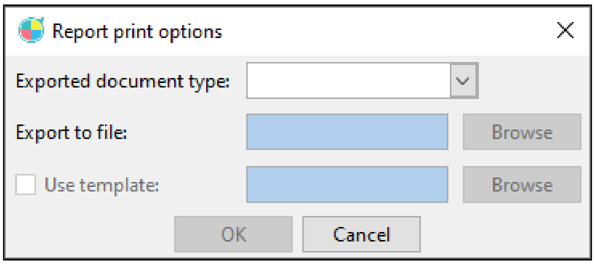

Symbols covered by non-passed scenarios
Uncovered symbols
Open scenario UserAddAndDelete
Open scenario UserAddAndDeleteWrong
Open scenario UserAddTwiceAndDelete
Open scenario UserDeleteWrong

Agent/symbol	Hits	Covering sce
pLocal	4	
wait4AddOrDelete	4	
key(newKey)	4	
11	4	(4 c
displayMessag	4	(4 c
wait4OldUserC	4	(4 c
wait4OldUserCard	4	

Agent/symbol
RTDS_TTCN_SysCh
pLocal
wait4Centra

The “*Actions*” button in the top right corner allows to perform the following actions:

- “*Export report as document*” allows to export the displayed report in a supported document format. This item opens the following dialog:



Report print options

Exported document type:

Export to file: Browse

□ Use template: Browse

OK Cancel

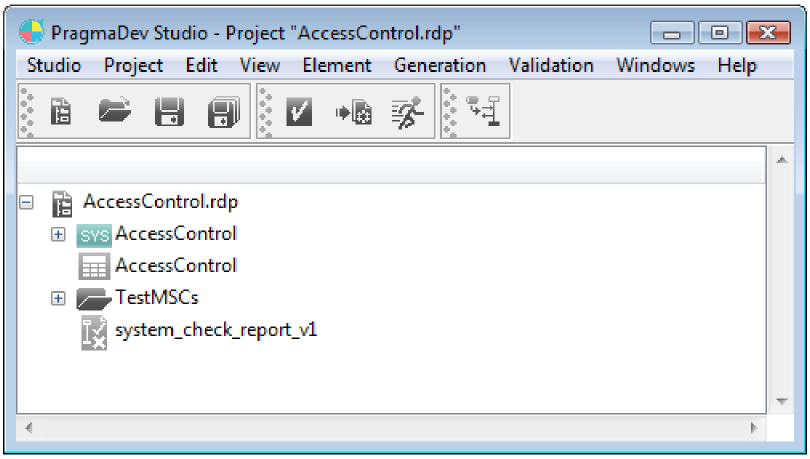
The available document types are HTML, PDF, RTF, OpenDocument and LaTeX. Once a type has been selected, the actual name for the exported file can be entered and a template can be selected for the document. This template has the same meaning as for document export, as explained in section “Exporting documents” on page 141. Note that a template has to be specified for OpenDocument format export, just as for documents.

- “*Restart with passed scenarios only...*”: this will go back to the MSC scenario selection panel, unselecting all those that ended up having a non-passed verdict in the previous execution.

- “*Generate TTCN tests for passed scenarios...*”: generates the TTCN testcases for all scenarios that ended up having a “passed” verdict. This is done as described in “From MSCs and/or HMSCs” on page 356.
- “*Generate TTCN tests for all scenarios...*”: same as above, except testcases for all scenarios are generated, not only passed ones.
- “*Delete non-passed scenarios from projects...*”: deletes all the scenarios that ended up having a non-passed verdict from the project. A confirmation will be asked via a standard node deletion dialog, also allowing to specify if the diagram files should also be deleted.

The three buttons at the bottom of the panel are:

- “*<< Back to MSCs*“ allows to get back to the MSC selection panel. Note that this will discard all the results of the previous execution.
- “*Save report...*” saves the execution report in a file. After the file name is entered via a standard file dialog, the report will appear in the project tree:



- “*Close*” closes the dialog, also discarding the results of the previous execution.

Note: this feature is actually an aggregation of various other features in PragmaDev Studio, made available in an easier to use interface for system checking. What is happening behind the scenes is:

- TTCN testcases are generated from the selected MSC diagrams, as described in “From MSCs and/or HMSCs” on page 356.
- TTCN testcases are executed against the system using the model simulator, as described in “Simulation” on page 346. The model coverage is automatically extracted during this execution, and all verdicts are recorded and associated to the initial MSC scenario.
- Simplified traces are also remembered during testcase execution, allowing to identify the message that made the testcase fail if needed.

7.5.3 Other validation experiments

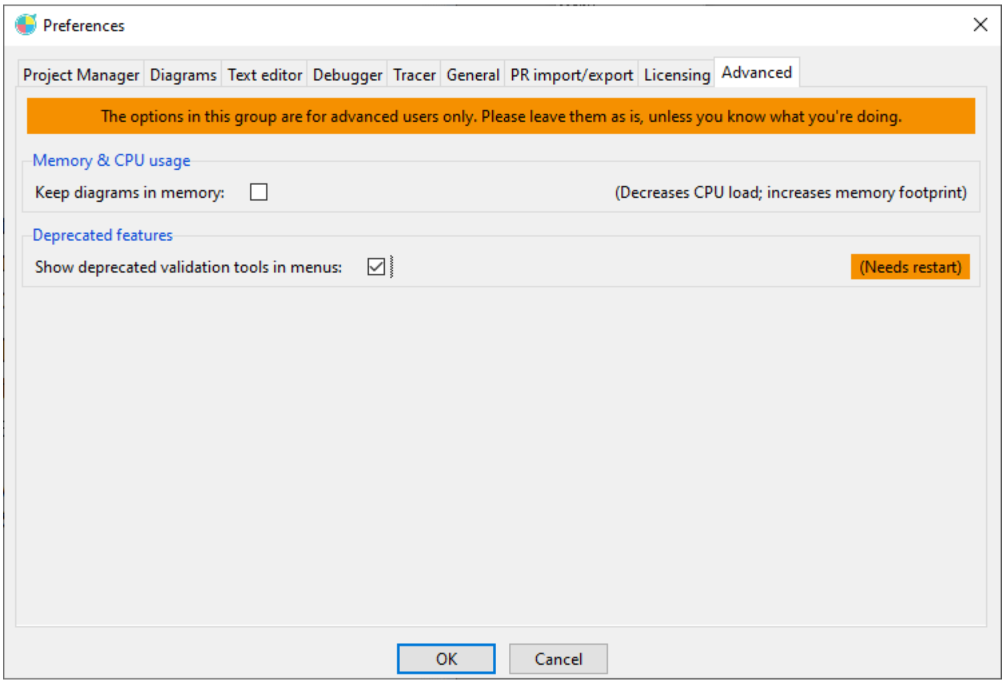
7.5.3.1 Scope and TRL

PragmaDev Studio offers a gateway to several technologies for system validation: the *IF toolbox* from Verimag, *Diversity* from the CEA List, and *Fiacre* from LAAS. Please note these tools are issued from research work and their TRL is not as high as the tools within PragmaDev Studio environment. For that reason we have estimated a TRL (Technology Readiness Level) for each technology:

- xLIA format in Diversity tool from CEA: TRL 5
- IF format in IFx tool from Verimag: TRL 5
- Fiacre format in Tina tool from LAAS: TRL 4

Since the third party tool might not have the exact same semantics or concepts support, the model can not be fully translated. Please refer to the Reference manual for possible restrictions in order to make sure it can be exported to one of these tools.

By default, because of their low TRL, these gateways are not available in the menus. In order to activate them go to *Studio / Preferences* window, and to the tab *Advanced*. Then check *Show deprecated validation tools in menus* in *Deprecated features* section.



The options in this group are for advanced users only. Please leave them as is, unless you know what you're doing.

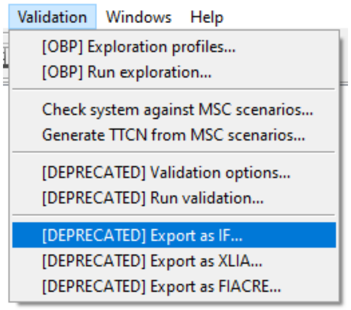
Memory & CPU usage

Keep diagrams in memory: □ (Decreases CPU load; increases memory footprint)

Deprecated features

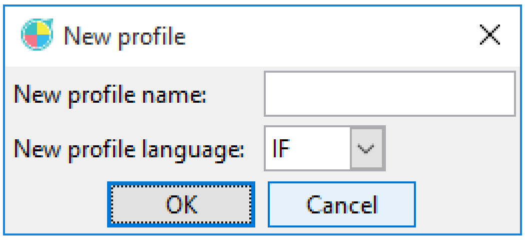
Show deprecated validation tools in menus: ☑ (Needs restart)

Preferences window


Validation
Windows
Help
[OBP] Exploration profiles...
[OBP] Run exploration...
Check system against MSC scenarios...
Generate TTCN from MSC scenarios...
[DEPRECATED] Validation options...
[DEPRECATED] Run validation...
[DEPRECATED] Export as IF...
[DEPRECATED] Export as XLIA...
[DEPRECATED] Export as FIACRE...

Added items in the Validation menu

It is then possible to create a validation profile from the “Validation / Options...” menu. When creating a new profile, a pop up window will ask the language used for the verification:



New profile name:

New profile language: IF

7.5.3.2 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.

7.5.3.2.1 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.

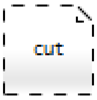
Observers, as supported in PragmaDev Studio, look like SDL processes but they are not SDL processes. The syntax of the statements is based on IF language, and it does not have any message queue by default.

7.5.3.2.1.1 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:



7.5.3.2.1.2 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 6: 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;

Table 6: IF constructs

Constructs	Declaration	Usage

enum	type MyType =
	enum red, green, blue
	endenum;
		var v MyType;
		task v := green;

record	type MyRecord=record
	FirstField integer;
	SecondField boolean;
	endrecord;
		var v MyRecord;
		task v.FirstField := 3;

range	type MyRange=range 0..4;	

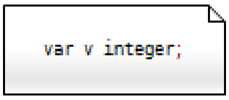
array	type MyArray=array[4] of integer;	var v MyArray;
		task v[3] := 5;

string	type MyString=string[5] of integer;
	var v MyString
		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

while	while (<condition>) do
	<statements>;
	endwhile;
		while (i<4) do
		output sig1;
		task i:=i+1;
		endwhile

if	if (<condition>) then
	<statements>;
	else
	<statements>
	endif;
		if (i<4) then
		output sig1;
		else
		output sig2;
		endif

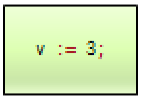
Variables are declared in the text symbol:



```
var v integer;
```

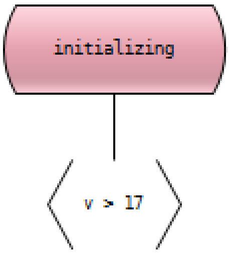
Variables are manipulated in:

- Action symbols

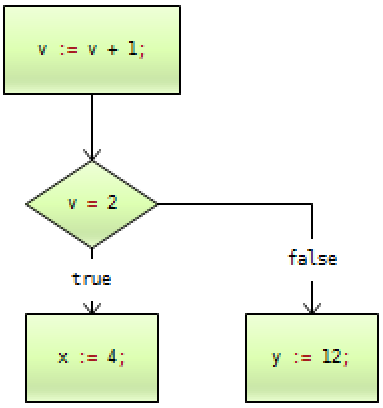


```
v := 3;
```

- Provided symbols



- Decision symbols

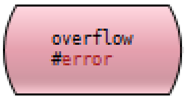


7.5.3.2.1.3 Action symbols in Observers

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:

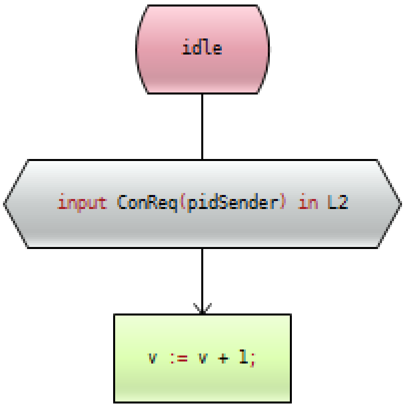


By default the state is considered ordinary.

Triggers

Three possible events can trigger the state: *match*, *provided*, *when*.

A match statement is described as a closed continuous signal SDL symbol:



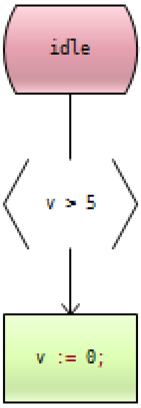
In this example, `pidSender` has previously been declared like a pid.

The possible match statements are:

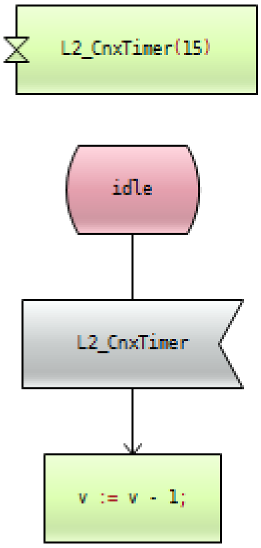
- match `input <sig(sender pid, params)> in <pid>`
- match `output <sig(sender pid, params)> from (<pid>) via (<channel>) to (<pid>)`
- match `fork(<pid>) in <pid>`
- match `kill(<pid>) in <pid>`
- match `<deliver>`
- match `informal "my text" in (<pid>)`

The match keyword is omitted in the graphical symbol.

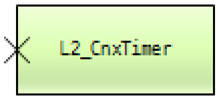
A provided statement is described with the SDL continuous signal symbol:



A when statement is described with the SDL start timer symbol to start the clock, and an input symbol:

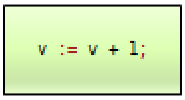


A timer can also be cancelled with the SDL cancel timer symbol:



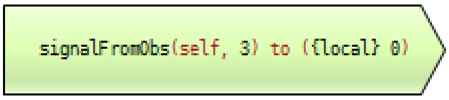
7.5.3.2.1.4 Statements

Any IF statement can be written in the SDL action symbol:



7.5.3.2.1.5 Output

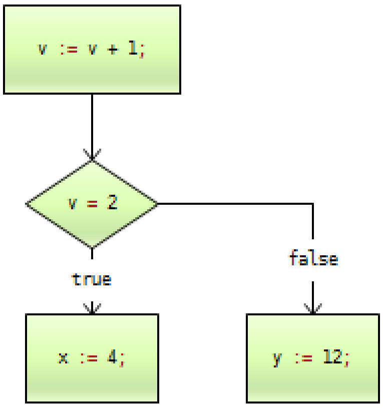
An IF intrusive observer is able to send signals to any process of the system, but can not receive any signal



self has always to be add as first parameter for an output in an observer.

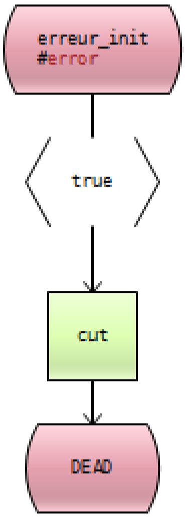
7.5.3.2.1.6 Decisions

Decisions are handled with unstable states in IF. It uses the SDL decision symbol:

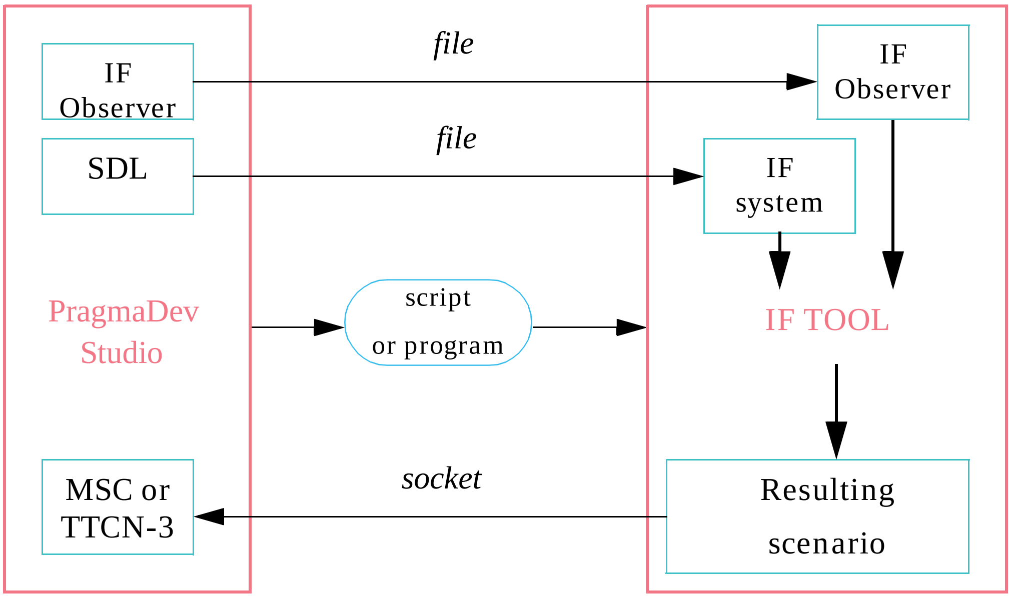


7.5.3.2.1.7 Reducing state space

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.

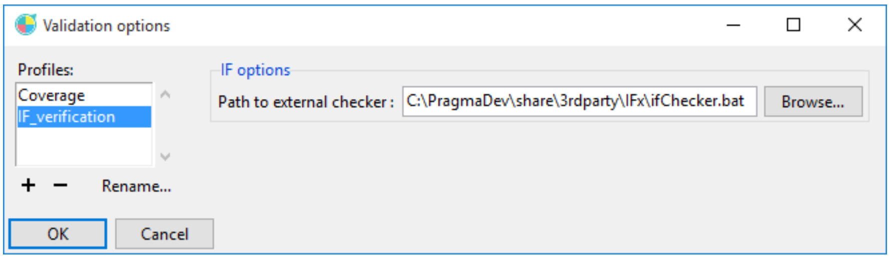


7.5.3.2.2 General architecture



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:



IF_verification

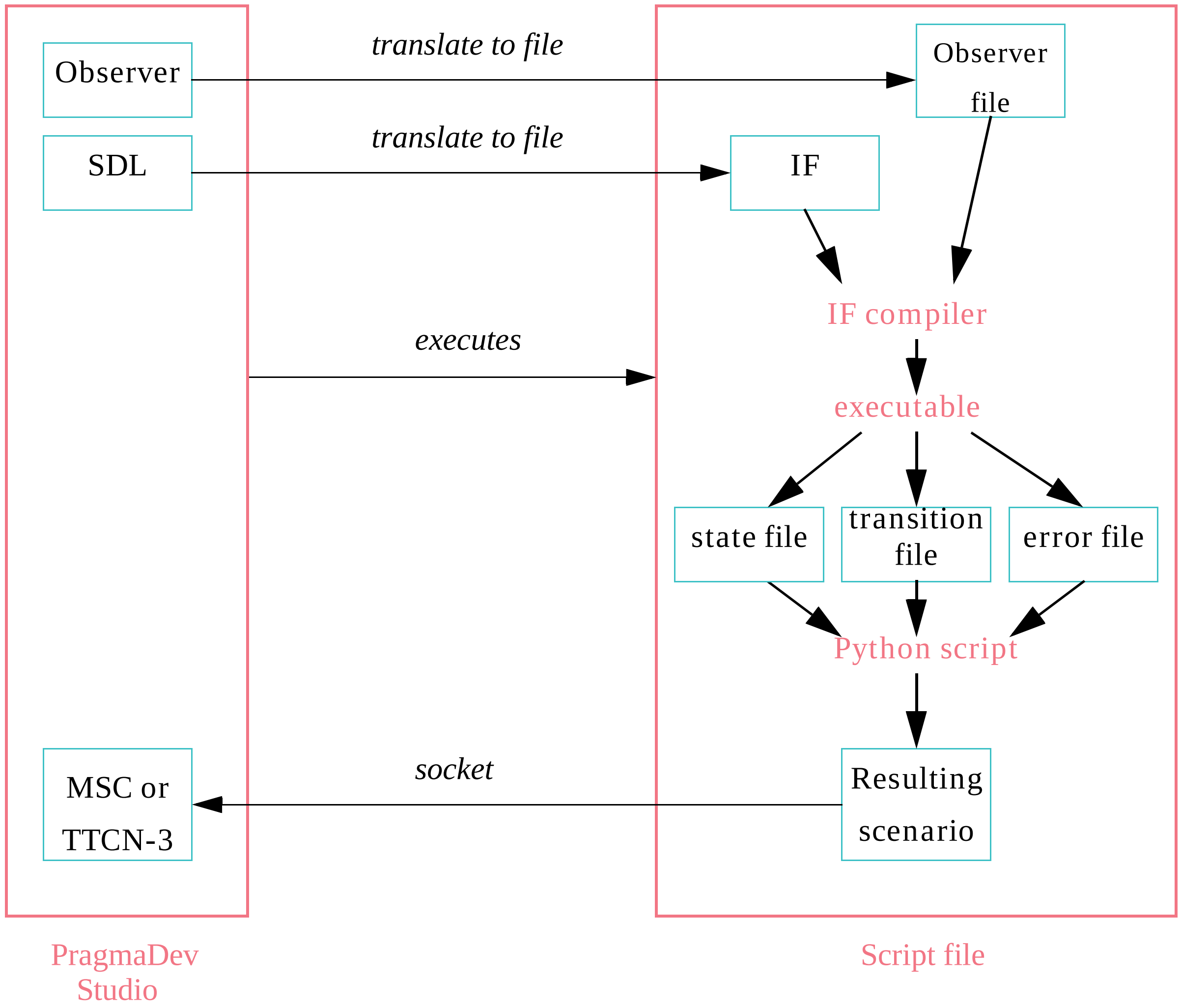
IF options

Path to external checker : C:\PragmaDev\share\3rdparty\IFx\ifChecker.bat

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.

7.5.3.2.3 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:

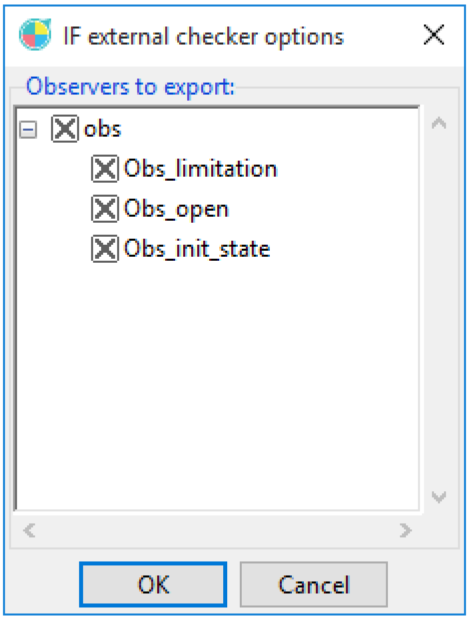


The script can be found in the sub-directory `share/3rdparty/IFx/ifChecker.sh` of the PragmaDev Studio installation directory. It requires an IF observer to specify the properties to check in the SDL system. This observer is specified via the environment variable `RTDS_IF_OBSERVER`, which must contain 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;

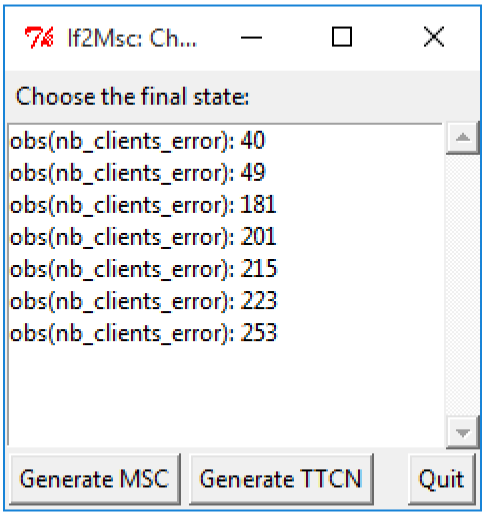


IF external checker options

Observers to export:

- ☒ obs
  - ☒ Obs_limitation
  - ☒ Obs_open
  - ☒ Obs_init_state

- 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:



If2Msc: Ch...

Choose the final state:

```
obs(nb_clients_error): 40
obs(nb_clients_error): 49
obs(nb_clients_error): 181
obs(nb_clients_error): 201
obs(nb_clients_error): 215
obs(nb_clients_error): 223
obs(nb_clients_error): 253
```

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.

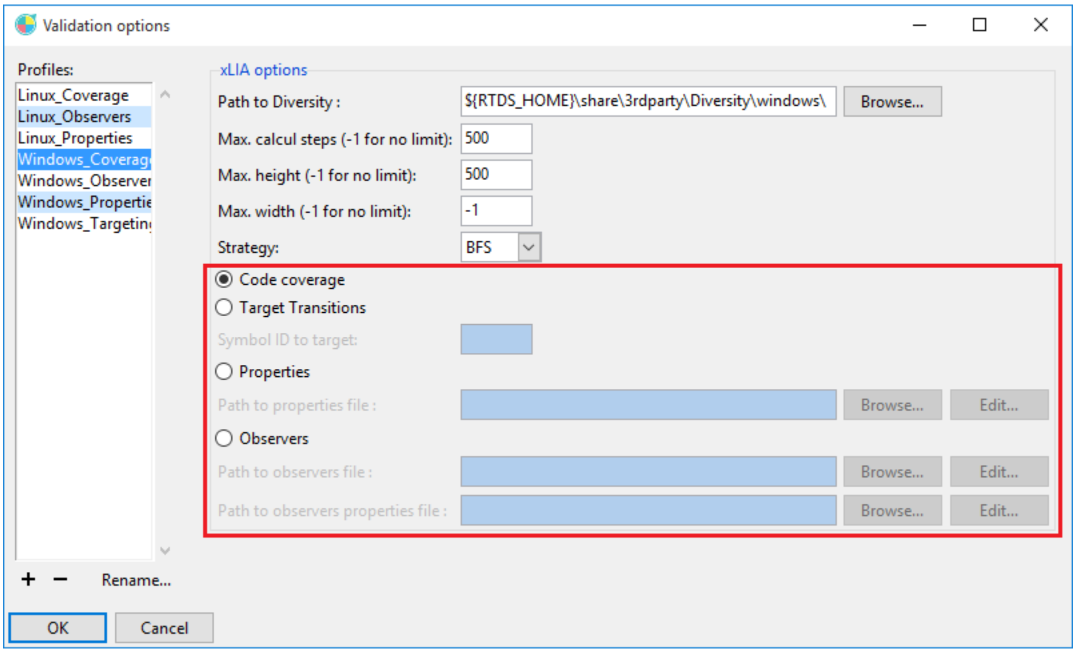
7.5.3.3 Diversity

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:



Profiles: Linux_Coverage, Linux_Observers, Linux_Properties, Windows_Coverag, Windows_Observer, Windows_Propertie, Windows_Targetin

Path to Diversity : ${RTDS_HOME}\share\3rdparty\Diversity\windows\

Max. calcul steps (-1 for no limit): 500

Max. height (-1 for no limit): 500

Max. width (-1 for no limit): -1

Strategy: BFS

● Code coverage

○ Target Transitions

Symbol ID to target:

○ Properties

Path to properties file :

○ Observers

Path to observers file :

Path to observers properties file :

7.5.3.3.1 Coverage

This is the simplest profile since the goal is to cover all transitions in the model.

7.5.3.3.2 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 };
```

7.5.3.3.2.1 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
```

7.5.3.3.2.2 Variable value

To verify the value of a variable, the syntax is:

```
EqualitySymbol spec::ProcessContext.VariableName value
```

with `EqualitySymbol` the symbol of equality verification (=,!=,<,>).

7.5.3.3.2.3 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}
```

7.5.3.3.2.4 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.

7.5.3.3.3 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.

7.5.3.3.3.1 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:

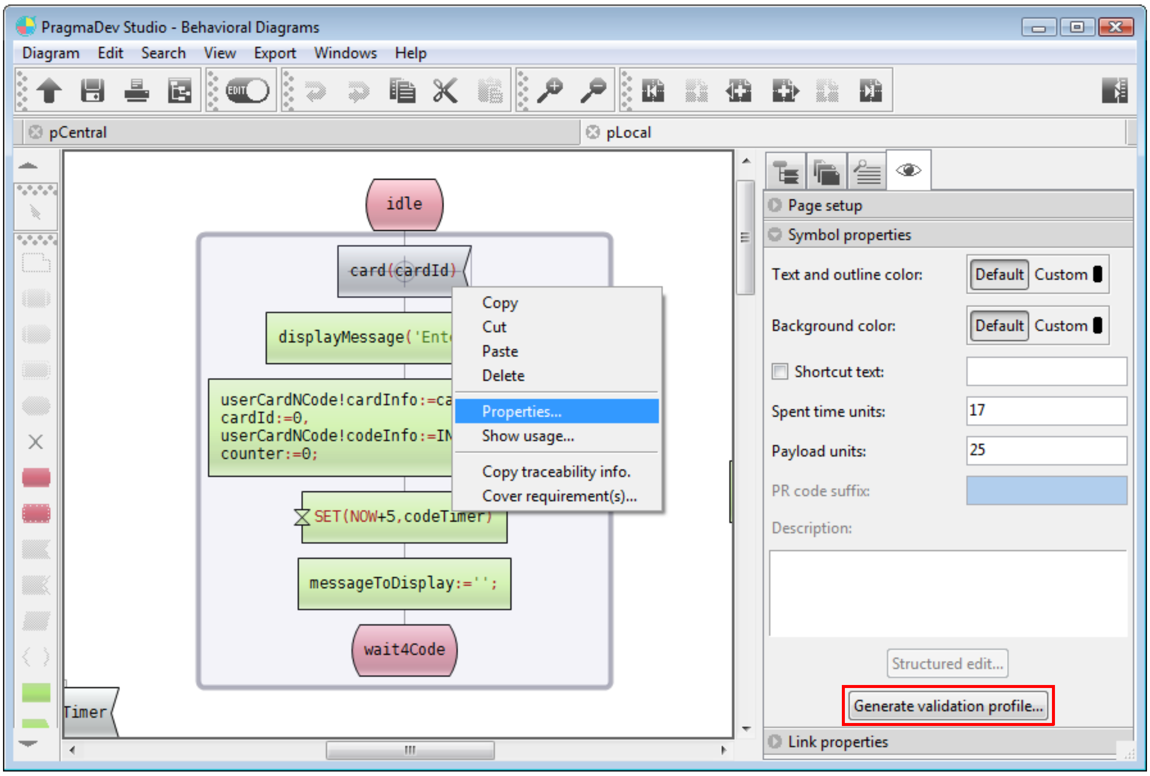
```
guard (: ProcessContext.StateName schedule#in ProcessContext);
```

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.

7.5.3.3.4 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.



Text and outline color: Default Custom

Background color: Default Custom

□ Shortcut text:

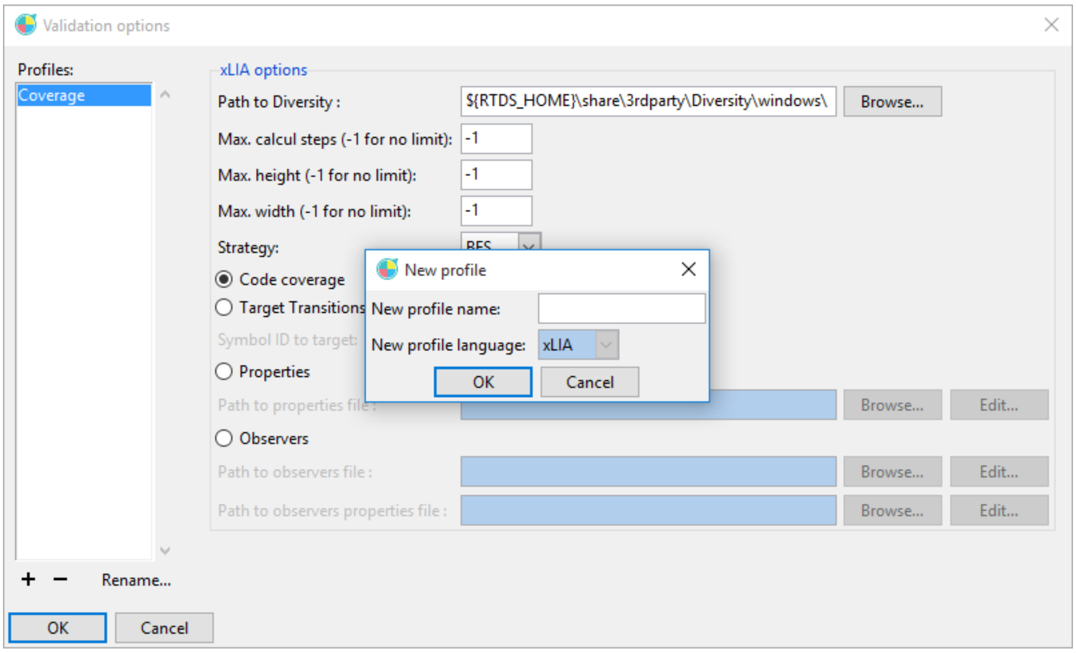
Spent time units: 17

Payload units: 25

PR code suffix:

Description:

Doing so will trigger the creation on a new validation profile:



Profiles: Coverage

Path to Diversity : ${RTDS_HOME}\share\3rdparty\Diversity\windows\

Max. calcul steps (-1 for no limit): -1

Max. height (-1 for no limit): -1

Max. width (-1 for no limit): -1

Strategy: BFS

● Code coverage

○ Target Transitions

Symbol ID to target:

○ Properties

Path to properties file :

○ Observers

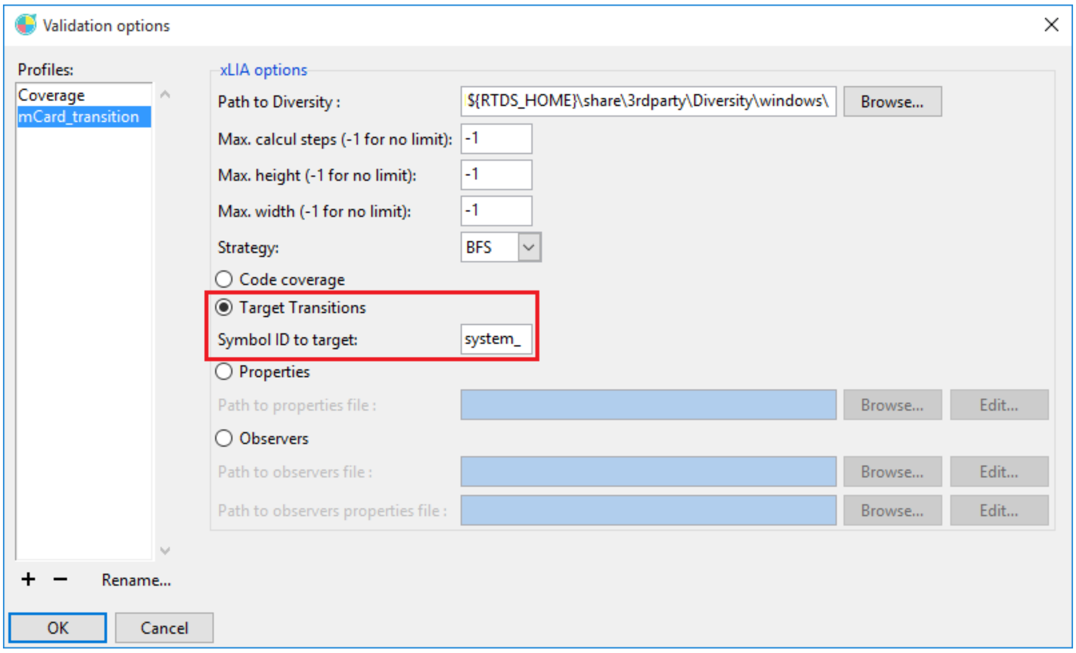
Path to observers file :

Path to observers properties file :

New profile name:

New profile language: xLIA

The Symbol ID to target will be filled automatically:



Validation options

Profiles:
- Coverage
- mCard_transition

xLIA options

Path to Diversity : ${RTDS_HOME}\share\3rdparty\Diversity\windows\

Max. calcul steps (-1 for no limit): -1

Max. height (-1 for no limit): -1

Max. width (-1 for no limit): -1

Strategy: BFS

○ Code coverage

● Target Transitions

Symbol ID to target: system_

○ Properties

Path to properties file :

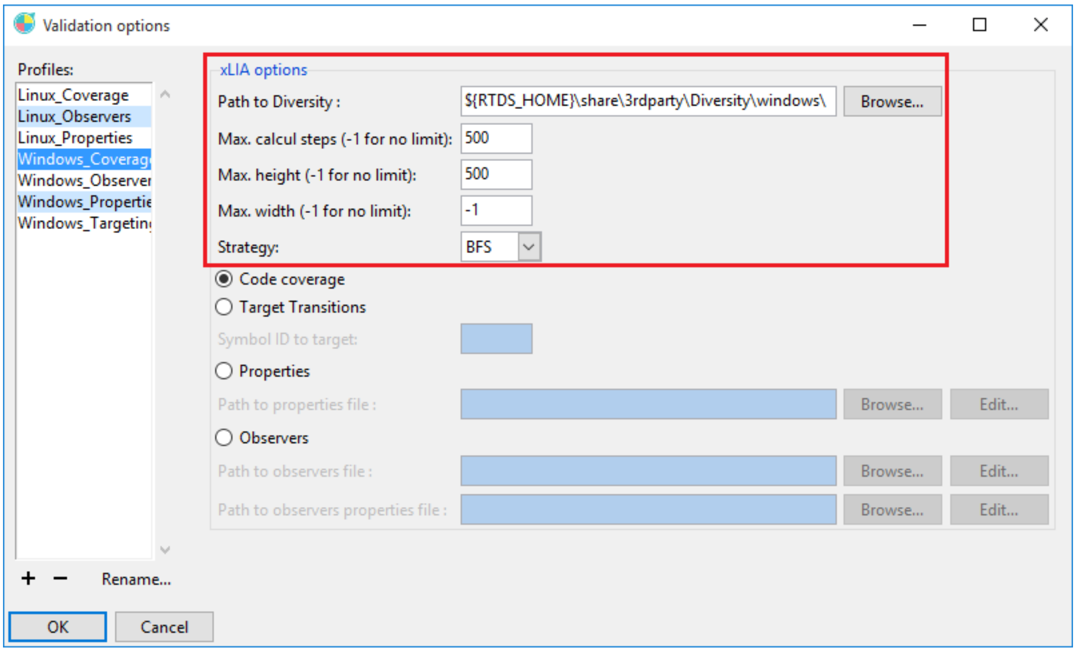
○ Observers

Path to observers file :

Path to observers properties file :

7.5.3.3.5 General validation properties

There are other options in xLIA profile:



Validation options

Profiles:
- Linux_Coverage
- Linux_Observers
- Linux_Properties
- Windows_Coverag
- Windows_Observer
- Windows_Propertie
- Windows_Targetin

xLIA options

Path to Diversity : ${RTDS_HOME}\share\3rdparty\Diversity\windows\

Max. calcul steps (-1 for no limit): 500

Max. height (-1 for no limit): 500

Max. width (-1 for no limit): -1

Strategy: BFS

● Code coverage

○ Target Transitions

Symbol ID to target:

○ Properties

Path to properties file :

○ Observers

Path to observers file :

Path to observers properties file :

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.

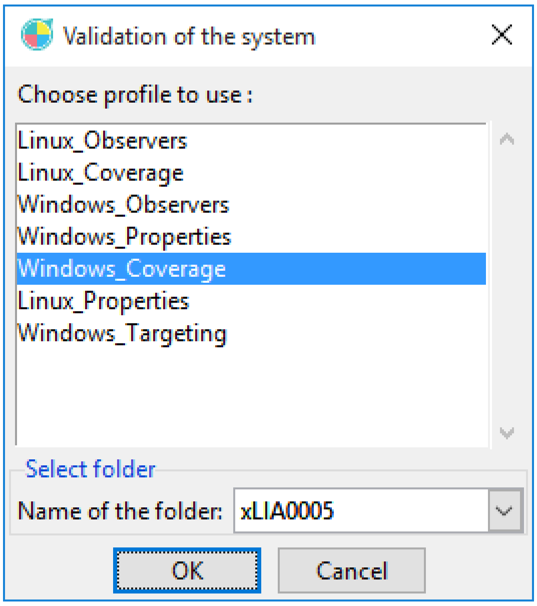
Strategy is for the exploration strategy type:

- DFS: Depth First Search
- BFS: Breadth First Search

- RFS: Random First Search

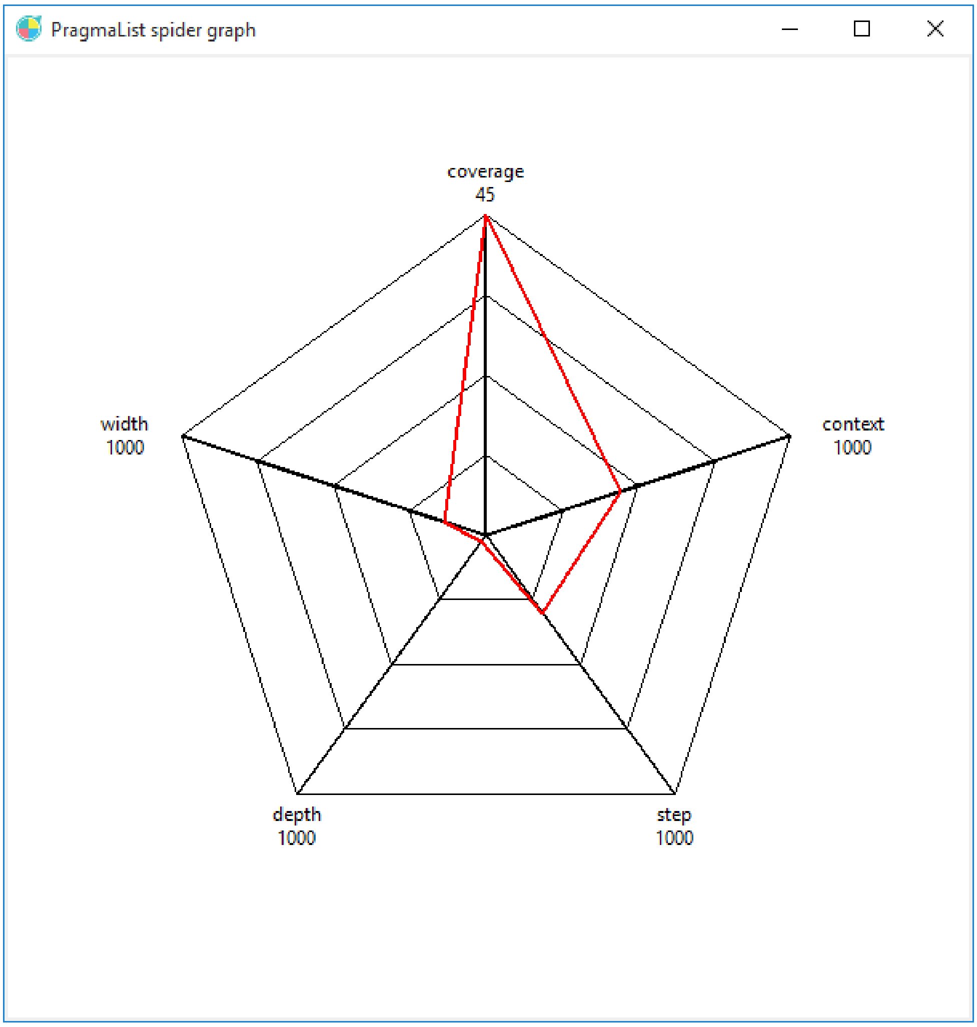
7.5.3.3.6 Results of verification

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:



Name of the folder: xLIA0005

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 symbolic 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:

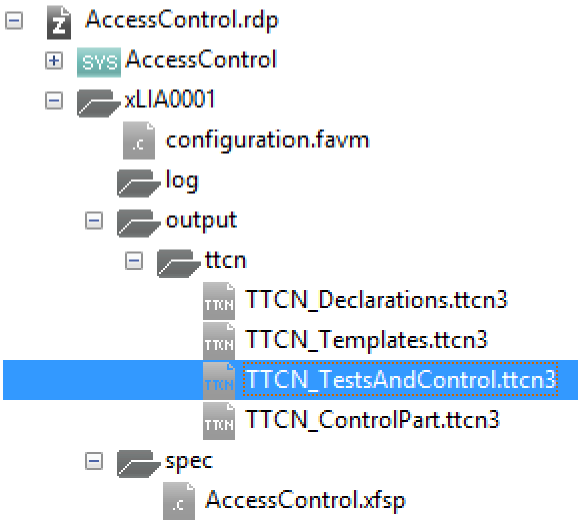
```
PROGRAM COVERAGE PROCESSOR
All the << 46 >> transitions are covered!
```

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:
```

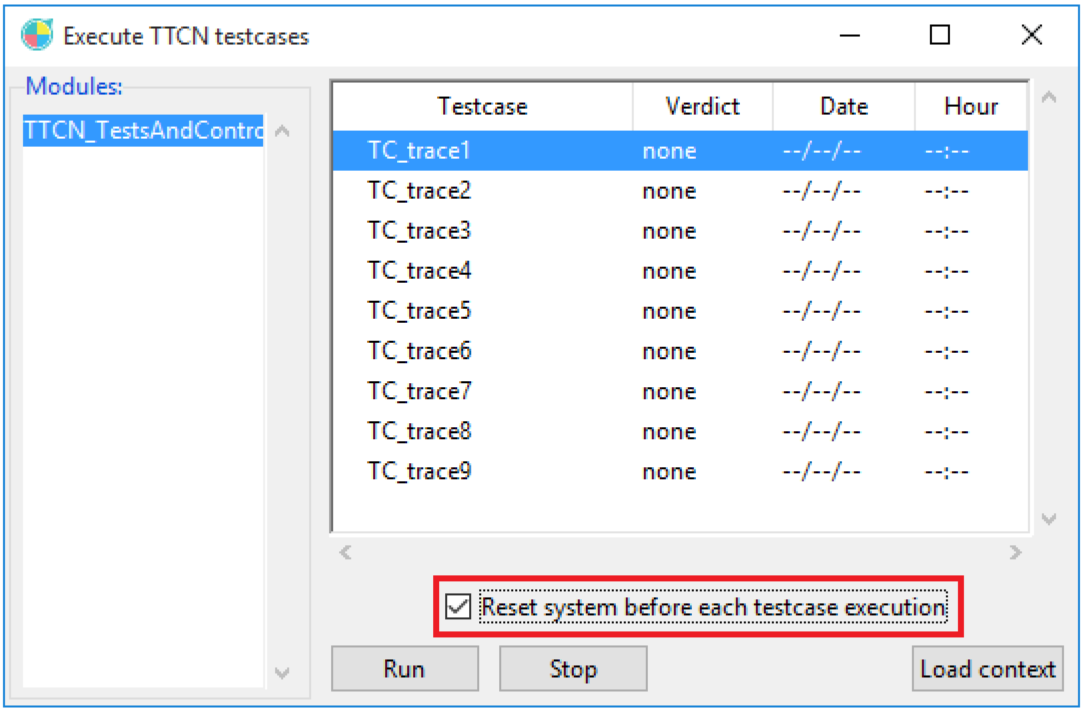
Finally, as a result, many files are generated in the specified folder:

- *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 ttcn files in *output/ttcn* sub-folders, results of Diversity exploration. The testcases are in the *TTCN_TestsAndControl.ttcn3* 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.

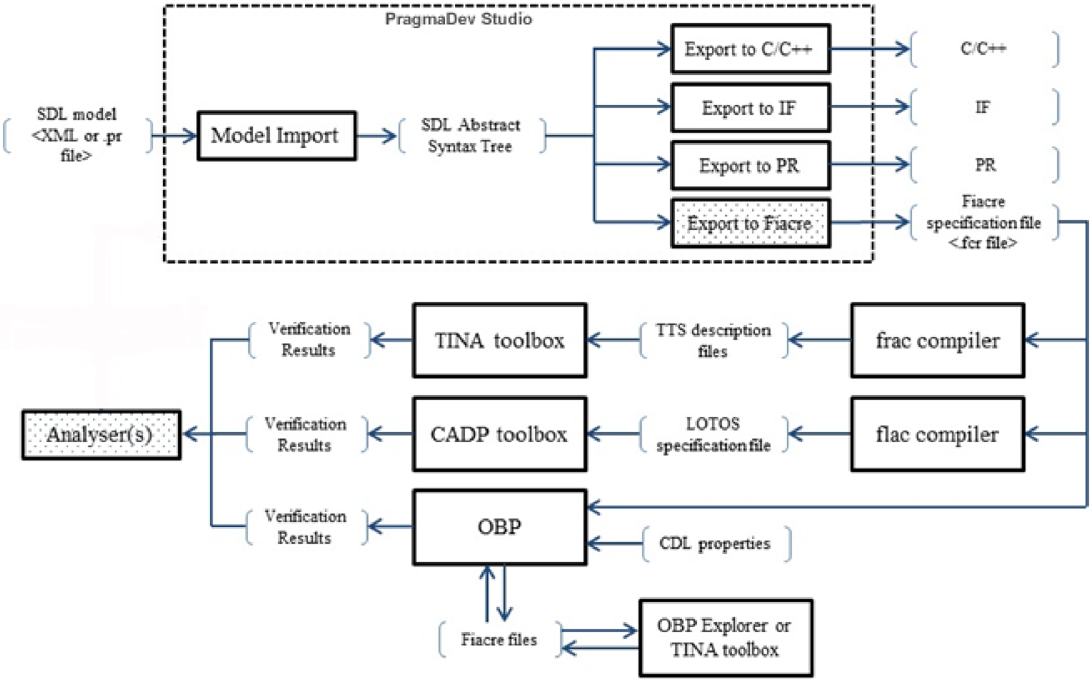


Testcase	Verdict	Date	Hour
TC_trace1	none	--/--/--	--:--
TC_trace2	none	--/--/--	--:--
TC_trace3	none	--/--/--	--:--
TC_trace4	none	--/--/--	--:--
TC_trace5	none	--/--/--	--:--
TC_trace6	none	--/--/--	--:--
TC_trace7	none	--/--/--	--:--
TC_trace8	none	--/--/--	--:--
TC_trace9	none	--/--/--	--:--

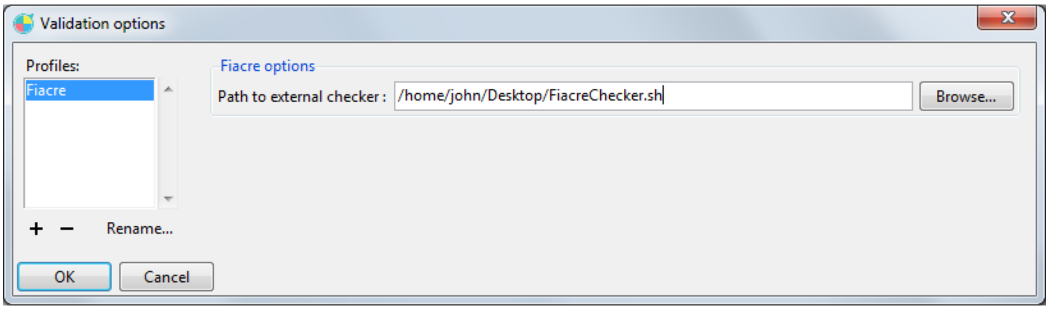
☑ Reset system before each testcase execution

7.5.3.4 Tina from LAAS

An export to Fiacre pivot language is available from PragmaDev Studio, but there is no feedback from the results to the original SDL model.



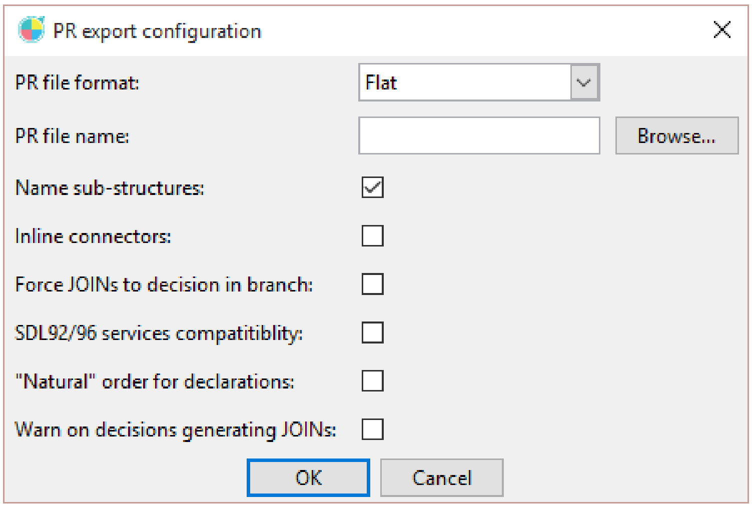
The only thing to do in this validation profile is to provide the path to the checker:



Path to external checker : /home/john/Desktop/FiacreChecker.sh

7.6 - Exporting the project as an SDL/PR file

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



PR file format: Flat

PR file name:

Name sub-structures: ☑

Inline connectors: □

Force JOINs to decision in branch: □

SDL92/96 services compatitiblity: □

"Natural" order for declarations: □

Warn on decisions generating JOINs: □

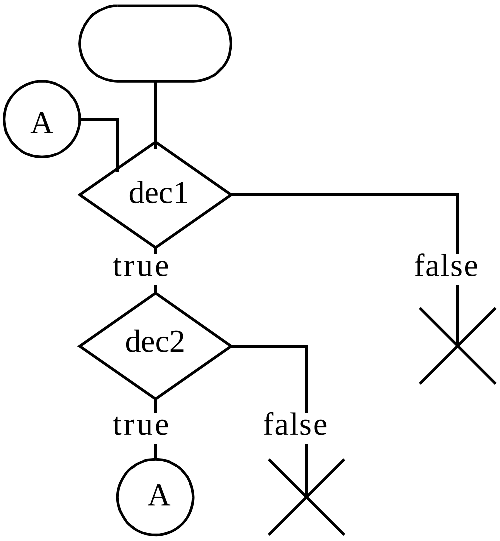
The PR file format indicates how agent definitions will be inserted in the exported file:

- 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;
```

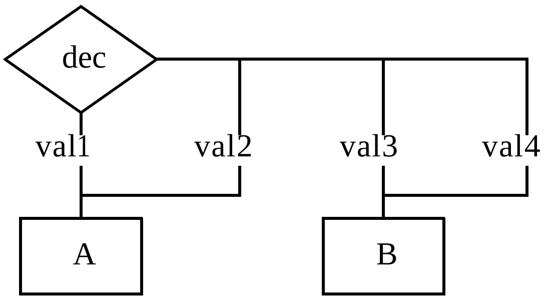
This is sometimes not desirable, since the `JOIN` to the label before the first `DECISION` is generated in a branch of the second `DECISION`. This makes it difficult for external code generators to figure out that the `JOIN` may actually be converted to a while-style loop.

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 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 additionnal 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 additionnal 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.

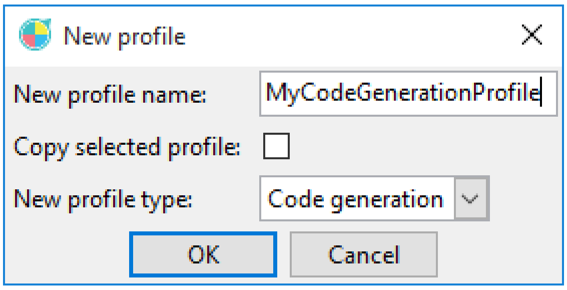
Note: To import PR files please refer to “Importing a PR/CIF file” on page 224.

7.7 - 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:



New profile

New profile name: MyCodeGenerationProfile

Copy selected profile: □

New profile type: Code generation

Select “Code generation” and fill in the options as explained in the “Profiles” on page 248.

The graphical debugger architecture and features are described in “Model Debugger” on page 311.

The following external procedures (described in “Provided external procedures” on page 214) are already available for code generation:

- `PragmaDev_b4sprintf`
- `PragmaDev_i4sprintf`
- `PragmaDev_f4sprintf`
- `PragmaDev_s4sprintf`
- `PragmaDev_sprintf`
- `PragmaDev_FileOpen`
- `PragmaDev_FileClose`
- `PragmaDev_FileReadLine`
- `PragmaDev_FileWriteLine`

The implementation of these procedures can be found in:

```
$(RTDS_HOME)/share/lib/ExternalProcedures.h
$(RTDS_HOME)/share/lib/ExternalProcedures.c
```

If one of these procedures is used in the model, the header file is automatically included in the generated headers, and the source file is automatically included in the generated makefile.

8 - Index