

User Guide for the TITAN Designer for the Eclipse IDE

Jenő Balaskó, Ádám Knapp

Version 11.2.1, 2026-08-05

Table of Contents

1. Introduction . . . 2
   1.1. Overview . . . 2
   1.2. Target Groups . . . 2
   1.3. Typographical Conventions . . . 2
   1.4. Installation . . . 2
   1.5. Reporting Errors . . . 2
2. Getting started . . . 4
   2.1. The TITAN Editing Perspective . . . 4
   2.2. Enabling TITAN Actions on the Toolbar . . . 6
   2.3. Enabling TITAN Shortcuts . . . 7
   2.4. Enabling TITAN Decorations . . . 8
   2.5. Excluding resources . . . 9
3. Setting Workbench Preferences . . . 11
   3.1. TITAN Preferences . . . 12
   3.2. Bracket matching preferences . . . 15
   3.3. Content Assist Preferences . . . 16
   3.4. Debug . . . 18
   3.5. Excluded Resources . . . 20
   3.6. Export . . . 21
   3.7. Folding Preferences . . . 22
   3.8. Indentation Preferences . . . 24
   3.9. Mark Occurrences . . . 25
   3.10. On-the-fly Checker Preferences . . . 25
      3.10.1. Pitfalls . . . 27
   3.11. Errors/Warnings Preferences . . . 28
      3.11.1. Pitfalls . . . 30
   3.12. Naming Conventions . . . 30
   3.13. Syntax Coloring Preferences . . . 31
   3.14. TITAN Actions . . . 35
   3.15. Typing Preferences . . . 36
4. Managing Projects . . . 38
   4.1. Creating a New TITAN C++ Project . . . 38
   4.2. Creating a New TITAN Java Project . . . 41
   4.3. Adding Directories to the Project . . . 41
   4.4. Adding Files to the Project . . . 43
      4.4.1. Using Wizards to Add Files to the Project . . . 43
      4.4.2. Manually Adding Files to the Project . . . 44
   4.5. Setting Project Properties . . . 46

4.5.1. Build Configurations . . . 46
4.5.2. Setting the Local Build Properties of a TITAN Project . . . 47
The Makefile Creation Attributes tab . . . 48
The Internal Makefile Creation Attributes Tab . . . 49
The Make Attributes Tab . . . 58
4.5.3. Setting the Local Build Properties of a TITAN Java Project . . . 60
The Internal Build Attributes Tab for TITAN Java Projects . . . 60
4.5.4. Setting Project and Folder Level Naming Convention Settings . . . 62
4.5.5. Setting Requirements on the Configuration of Referenced Projects . . . 64
4.5.6. Setting the Remote Build Properties of a Project . . . 65
Pitfalls . . . 68
4.6. Excluding Files and Folders from the Build Process . . . 68
4.6.1. Excluding a File from the Build Process . . . 68
4.6.2. Excluding a Folder from the Build Process . . . 69
4.7. Converting a Folder into a Central Storage . . . 70
4.8. Opening and Closing Projects . . . 70
4.9. Saving and Loading Project Properties . . . 70
4.10. Importing and Exporting Projects . . . 71
4.10.1. Exporting Projects in Native Format . . . 71
4.10.2. Importing Projects from Native Format . . . 73
4.10.3. Importing an Existing mctr_gui Project . . . 75
4.10.4. Importing Files as Linked Resources . . . 77
4.10.5. Exporting Projects into the TITAN Project Descriptor (tpd) Format . . . 81
Exporting Project manually into the TITAN Project Descriptor (tpd) Format . . . 81
Exporting Projects automatically into the TITAN Project Descriptor (tpd) Format . . . 85
4.10.6. Importing Projects from TITAN Project Descriptor Format . . . 85
4.10.7. Importing Projects from the Command Line . . . 88
4.10.8. Useful Tips for Exporting and Importing . . . 88
Pitfalls . . . 88
Native Export and Import . . . 89
Exporting and Importing Project Information and Projects via TPD Files in Case of Complex Projects . . . 89
Exporting Project Content from Command Line Using TPDs . . . 90
4.11. Formatting Log Files . . . 91
4.12. Merging Log Files . . . 91
4.13. Using Project References . . . 91
4.14. Mapping Elements of the Old Format . . . 93
4.15. Common Threats . . . 93
4.15.1. Disabling, Removing or Corrupting the Builder of the Project . . . 93
4.15.2. Removing or Corrupting the Nature of the Project . . . 94
4.15.3. Adding or Removing Resources from the Project . . . 94

4.16. Make Archive . . . 94
5. Converting Existing Projects . . . 96
5.1. The Construction Principles of Projects . . . 96
5.1.1. Makefile . . . 96
5.1.2. Mctr_gui . . . 96
5.1.3. Eclipse . . . 97
5.2. Manually Converting an Existing Project to Eclipse Format . . . 98
5.2.1. Small Project . . . 98
5.2.2. Large Project Sets Consisting of Several Included Projects or Logically Separate Parts . . . 99
5.2.3. Large Projects Using Central Storage Folders . . . 100
5.2.4. Project Referring to Specific Files Outside its Own Jurisdiction . . . 100
5.3. Convert an Existing mctr_gui Project Using an Import Wizard . . . 101
6. Building the Project . . . 102
6.1. Building the TITAN C++ Project . . . 102
6.1.1. Step by Step . . . 102
Creating Symbolic Links . . . 102
Creating or Regenerating the Makefile . . . 103
Editing the Makefile Skeleton . . . 103
Module Compilation . . . 103
Creating Dependencies . . . 104
Building . . . 105
6.1.2. Remote Build . . . 106
Remarks and Tips . . . 107
6.1.3. Building from the Command Line . . . 108
Building Directly . . . 108
Building with an External Script . . . 108
6.1.4. Cleaning the TITAN Project . . . 110
6.1.5. Pitfalls . . . 111
6.2. Building the TITAN Java Project . . . 111
6.2.1. Step by Step . . . 111
Module Compilation . . . 111
Building . . . 112
6.2.2. Cleaning the TITAN Java Project . . . 112
7. Editing with TITAN Designer Plugin . . . 113
7.1. File Types . . . 113
7.2. Syntax Highlighting . . . 113
7.3. Matching Brackets . . . 113
7.4. Folding . . . 114
7.5. On-the-fly Parsing . . . 114
7.5.1. Preprocessing of ttcnpp and ttcnin Files . . . 115
7.5.2. Limitations . . . 118

7.6. On-the-fly Semantic Checking . . . 118
7.6.1. Limitations . . . 118
7.7. Content Assistance . . . 118
7.7.1. Assistance with Keywords . . . 119
7.7.2. Assistance with Code Skeletons . . . 119
Using the Inserted Skeleton . . . 119
7.7.3. Assistance with Dynamic Elements . . . 120
7.7.4. Content Assistance Limitations . . . 121
7.8. Documentation comments . . . 121
7.8.1. Generate documentation comment . . . 122
7.8.2. Documentation comments limitations . . . 123
7.9. Find Declaration . . . 123
7.10. Find References . . . 124
7.11. Mark Occurrences . . . 125
7.11.1. Limitations . . . 125
7.12. Peek declaration . . . 125
7.13. Refactoring . . . 126
7.13.1. Rename Refactoring . . . 126
7.13.2. Limitations . . . 127
7.14. Editing Configuration Files . . . 127
7.14.1. Module Parameters Section . . . 128
7.14.2. Test Port Parameters Section . . . 128
7.14.3. Components, Groups and Main Controller Section . . . 129
Main Controller Options . . . 130
Components . . . 130
Group Section . . . 131
7.14.4. Execute and External Commands Sections . . . 131
External Commands . . . 132
Elements to be Executed . . . 132
7.14.5. Include and Define Sections . . . 133
Included Configurations . . . 133
Definitions . . . 134
7.14.6. Logging Section . . . 134
Components and Plug-ins . . . 135
Logging Options for the Selected Component/Plug-in . . . 135
7.14.7. Limitations on the Graphical Pages . . . 136
8. Contents of the Problems View . . . 137
8.1. Types of Markers . . . 137
8.2. Eclipse Provided Features . . . 137
8.3. Grouping of Problems . . . 138
8.3.1. Group by Severity . . . 138

8.3.2. Group by Type . . . 138
8.3.3. Group by TITAN Problems . . . 139
9. Contents of the Tasks View . . . 140
9.1. Types of Markers . . . 140
10. Contents of the Outline View . . . 141
10.1. The Tree . . . 141
10.2. The Toolbar . . . 141
10.2.1. Sorting Elements . . . 141
10.2.2. Categorizing Elements . . . 142
10.2.3. Grouping . . . 143
10.2.4. Filtering Elements . . . 143
10.3. Outline View Icons . . . 144
11. The Call Hierarchy View . . . 146
11.1. The Tree . . . 146
11.2. The Call List . . . 147
11.3. The Toolbar . . . 147
11.3.1. The refresh button . . . 147
11.3.2. The auto jump to definition switch . . . 148
11.3.3. The call list switch . . . 148
11.3.4. The close all button . . . 148
11.3.5. The search history . . . 149
12. Extensions to the Project Explorer . . . 150
12.1. Filtering Resources from the View . . . 150
13. References . . . 153
14. Abbreviations . . . 154

Abstract

This document describes detailed information of using the TITAN Designer for the Eclipse IDE plug-in.

Copyright

Copyright (c) 2000-2026 Ericsson Telecom AB.
All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v2.0 which accompanies this distribution, and is available at

https://www.eclipse.org/org/documents/epl-2.0/EPL-2.0.html.

Disclaimer

The contents of this document are subject to revision without notice due to continued progress in methodology, design and manufacturing. Ericsson shall have no liability for any error or damage of any kind resulting from the use of this document.

Chapter 1. Introduction

1.1. Overview

This document describes the general workflow and use of the TITAN Designer for the Eclipse IDE plug-in.

The TITAN Designer plug-in provides support for:

- creating and managing projects;
- creating and working with source files;
- building executable code;
- automatic analysis of the build results;
- remote build.

1.2. Target Groups

This document is intended for system administrators and users who intend to use the TITAN Designer plug-in for the Eclipse IDE.

1.3. Typographical Conventions

This document uses the following typographical conventions:

- **Bold** is used to represent graphical user interface (GUI) components such as buttons, menus, menu items, dialog box options, fields and keywords, as well as menu commands. Bold is also used with '+' to represent key combinations. For example, **Ctrl+Click**
- The "/" character is used to denote a menu and sub-menu sequence. For example, **File / Open**.
- `Monospaced` font is used represent system elements such as command and parameter names, program names, path names, URLs, directory names and code examples.
- **`Bold monospaced font`** is used for commands that must be entered at the Command Line Interface (CLI), For example, **`mctr_gui`**

1.4. Installation

For details on installing the TITAN Designer for the Eclipse IDE plug-in, see the Installation Guide for TITAN Designer and TITAN Executor for the Eclipse IDE.

1.5. Reporting Errors

The following information should be included into trouble reports:

- Short description of the problem.

- What seems to have caused it, or how it can be reproduced.
- If the problem is graphical in some way (displaying something wrong), screenshots should also be included.
- If the problem generates some output to:
  - TITAN Console
  - TITAN Debug Console
- If the Error view contains some related information, that should be copied too.

Before reporting a trouble, try to identify if the trouble really belongs to the TITAN Designer for the Eclipse IDE plug-in. It might be caused by other third party plug-ins, or by Eclipse itself.

Reporting the contents of the Consoles and the Error log is important as TITAN consoles display the commands executed and their results and the Error log may contain stack traces for some errors. To identify relevant log entries the easiest way is to search for classes whose name starts with "org.eclipse.titan". The location on which the Error Log view can be opened can change with Eclipse versions, but it is usually found at **Window** / **Show View** / **Other...** / **PDE Runtime** / **Error Log** or **Window** / **Show View** / **Other...** / **General** / **Error Log**.

Chapter 2. Getting started

This section explains how to setup Eclipse to access every feature provided by TITAN Designer.

2.1. The TITAN Editing Perspective

TITAN Designer provides its own perspective to Eclipse. This is a layout of visual elements that provides a good environment for working with TITAN. This layout is a starting point, since users can create their own layout in Eclipse, to set the best working environment for themselves.

Open the TITAN Designer perspective by opening **Window** / **Open Perspective** / **Other...**.

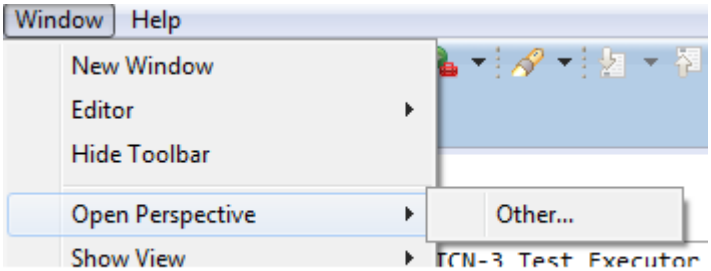


Figure 1. Opening a perspective

In the pop-up window select **TITAN Editing**.

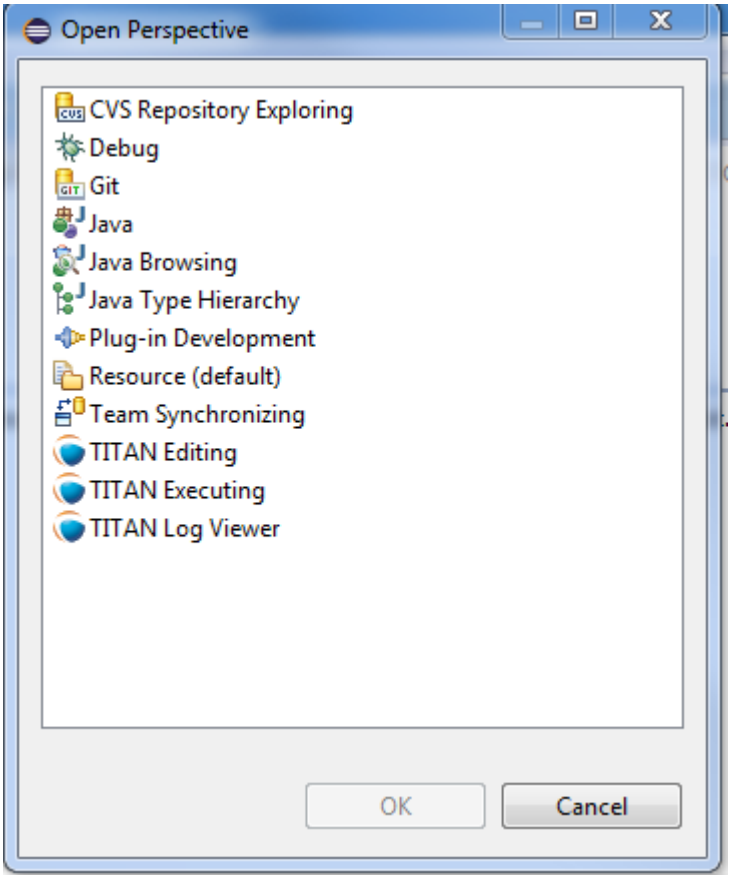


Figure 2. Selecting the TITAN Editing perspective

The perspective is divided in three fields. Figure TITAN Editing Perspective shows the default layout.

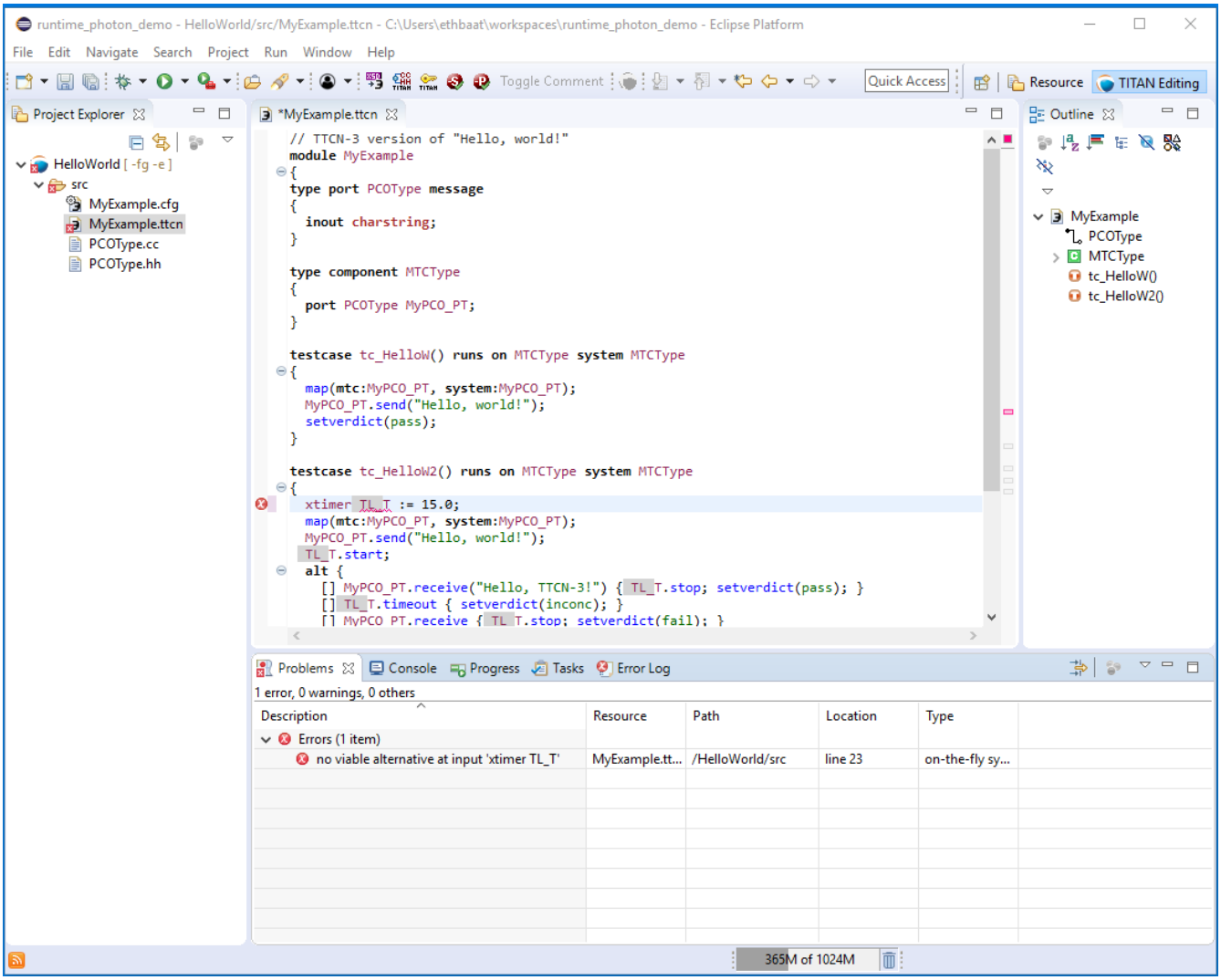


Figure 3. TITAN Editing Perspective

The tab on the left side is the Project Explorer view. This is a navigator where projects can be managed; for example, opened, renamed, or closed. Files can be added or removed from a project and so on.

The biggest pane of the perspective is the editing area (upper right). Here the code can be edited using the provided source code editors (or built-in text editors), once a file had been opened.

The four tabs at the bottom of the picture open the following views:

- The Problems view (see here) displays information about problems found in the project. The problems reported can be ordered using several criteria (see here).
- The Console view contains the commands executed and their output; Consoles only appear if there is something to display. The Console view has two sub views (TITAN console and TITAN Debug console, respectively); by default only one of both is displayed in the pane. The hidden sub view can be displayed by clicking on the display icon on the right of the pane.
- The TITAN console displays the commands executed by the parts of TITAN Designer and their results.
- The TITAN Debug console holds special debug related information for the plug-in developers. If something strange happens this might hold additional information that the user can include in his trouble report.

NOTE	The contents of this view have no effect on the work of the user.

- The Progress view contains information about the progresses of Eclipse related operations. Lengthy operations (for example building, remote building or the first on-the-fly build pass) always provide information to the user about their progress. Operations in general can be canceled in this view, provided that cancellation is allowed.
- The Tasks view contains information extracted from the projects in a sorted manner. The contents of this view differ from the contents of the Problems view in that they are usually not errors, but TODO or FIXME like notations. This view is described in detail here.

2.2. Enabling TITAN Actions on the Toolbar

TITAN Actions or Change Set Operations are commands (apart from those used in the build process) that can be executed on TTCN-3 files.

The TITAN Actions are enabled by checking the **Change Set Operations** submenu on the tab **Tool Bar Visibility** after selecting **Window / Perspective / Customize Perspective** (see the next figure).

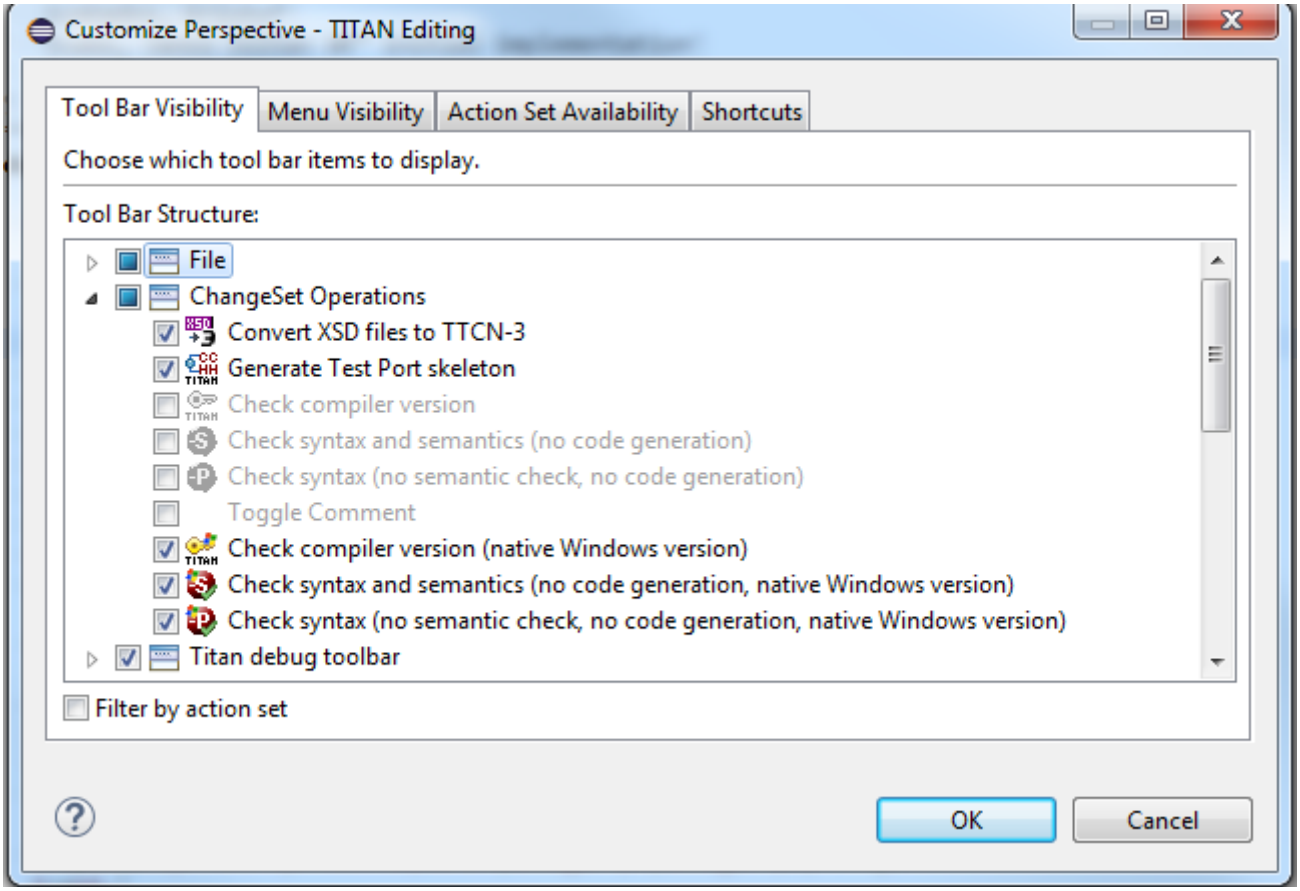


Figure 4. Enabling Titan Actions or ChangeSet Operations on the Toolbar

Enabling TITAN Actions will add a new toolbar with the TITAN Actions commands described below to the available ones:



Figure 5. TITAN Actions commands

The command `Check syntax` checks the selected files for syntactical errors; no other operation is performed. When a folder is selected, the check is performed for all the files in the folder. The command is only available if at least one file is selected.

The command `Check semantics` checks the selected files both syntactically and semantically; no other operation is performed. When a folder is selected, the check is performed for all the files in the folder. The command is only available if at least one file is selected.

The command `Check compiler version` displays the version of the compiler.

The command `Generate Test Port skeleton` generates a test port skeleton from the selected TTCN-3 file. The command is only available if there is exactly one selected file in the project.

The command Convert XSD files to TTCN-3 takes as input the files selected by the user, and converts them into TTCN-3 files. As for the output the user is asked to select a folder, where the newly created files will be written to.

The output of the commands is written to the TITAN Console. Commands are executed regardless of the file properties; for example, the selected file will be syntactically or semantically checked even if it is excluded from the build process.

2.3. Enabling TITAN Shortcuts

TITAN Shortcuts appear in the **File/New** menu and are used to open a new ASN.1 Module, a Configuration file, a TITAN Project (C++ or Java) or a TTCN-3 Module, respectively.

The TITAN Shortcuts are enabled by checking the appropriate box on the right pane of the **Shortcuts** tab after selecting **Window / Perspective / Customize Perspective...** (see the next figure).The boxes are checked by default.

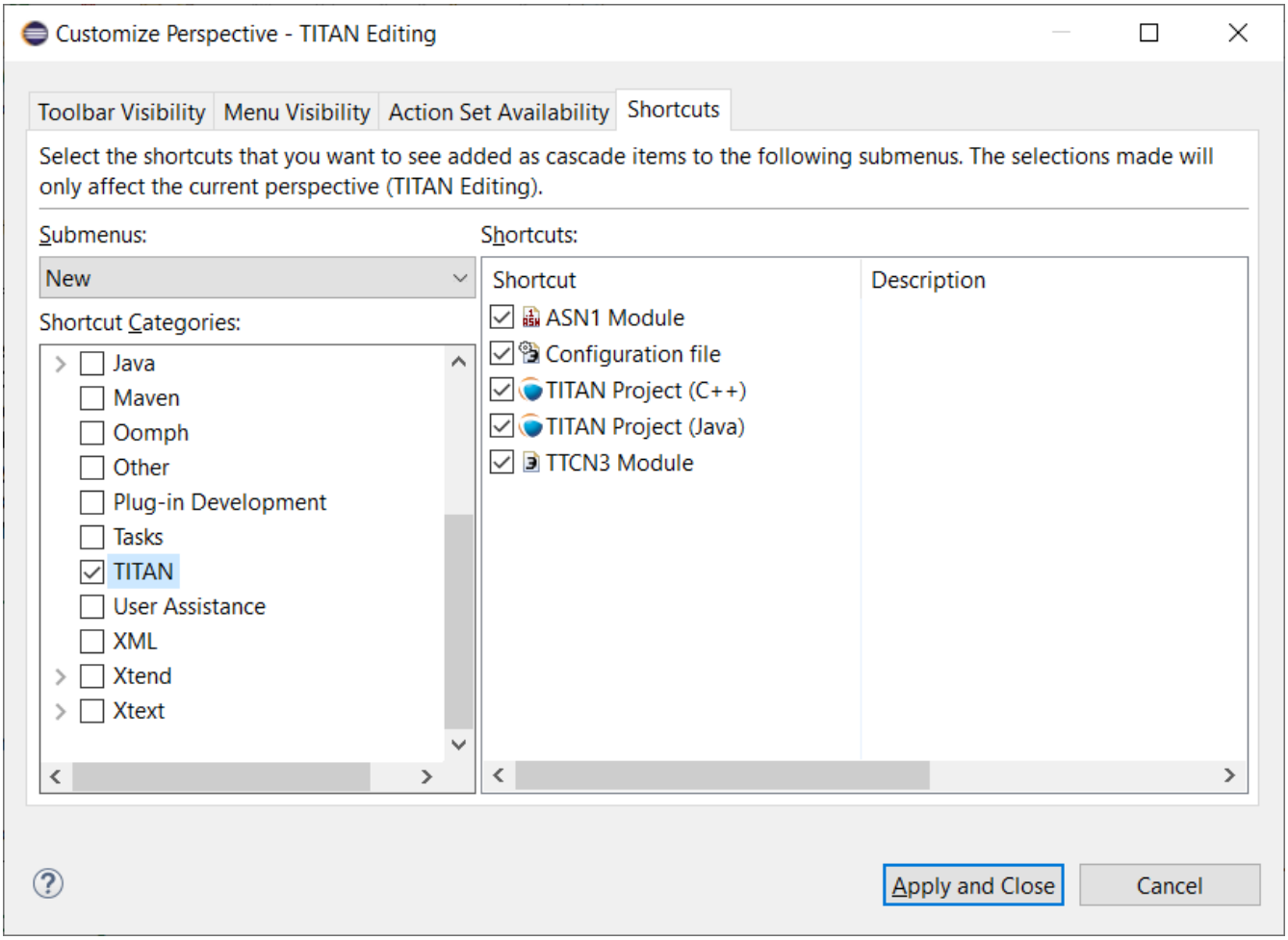


Figure 6. Enabling the TITAN Shortcuts

2.4. Enabling TITAN Decorations

Decoration here means a string added to a project, folder or file name or a picture overlapping the icon of the resource to provide the reader with additional information.

The ✔ mark on the top right corner of a project's icon means the project has been built and the binaries are up to date. If the plug-in detects the modification of a non-excluded file or folder inside the project, the check mark will disappear.

Decoration after a project name shows whether the `Makefile` has been automatically generated. If it has, the corresponding command line switches of the command `makefilegen` are displayed between brackets; for example, `[ -s ]` for single mode.

Some of these:

- a - use absolute pathnames in the generated Makefile
- c - use the pre-compiled files from central directories (central storage)
- f - force overwriting of the output Makefile
- g - generate Makefile for use with GNU make
- R - use function test runtime (TITAN_RUNTIME_2)
- s - generate Makefile for single mode

- l - use dynamic linking

No additional text is displayed if the `Makefile` has been manually generated (not even the brackets).

Decoration after a folder name indicates that the folder is used as a central storage (`[ centralstorage ]`) or the folder is excluded from build (`[ excluded by X]`). If both are true, `[ excluded by X centralstorage ]` is displayed.

Decoration after a file name denotes exclusion from build. Files excluded from build are marked `[ excluded ]`.

Decoration is enabled by checking the **TITAN Decorator** box after selecting **Window** / **Preferences** / **General** / **Appearance** / **Label decorations**; see the figure below.

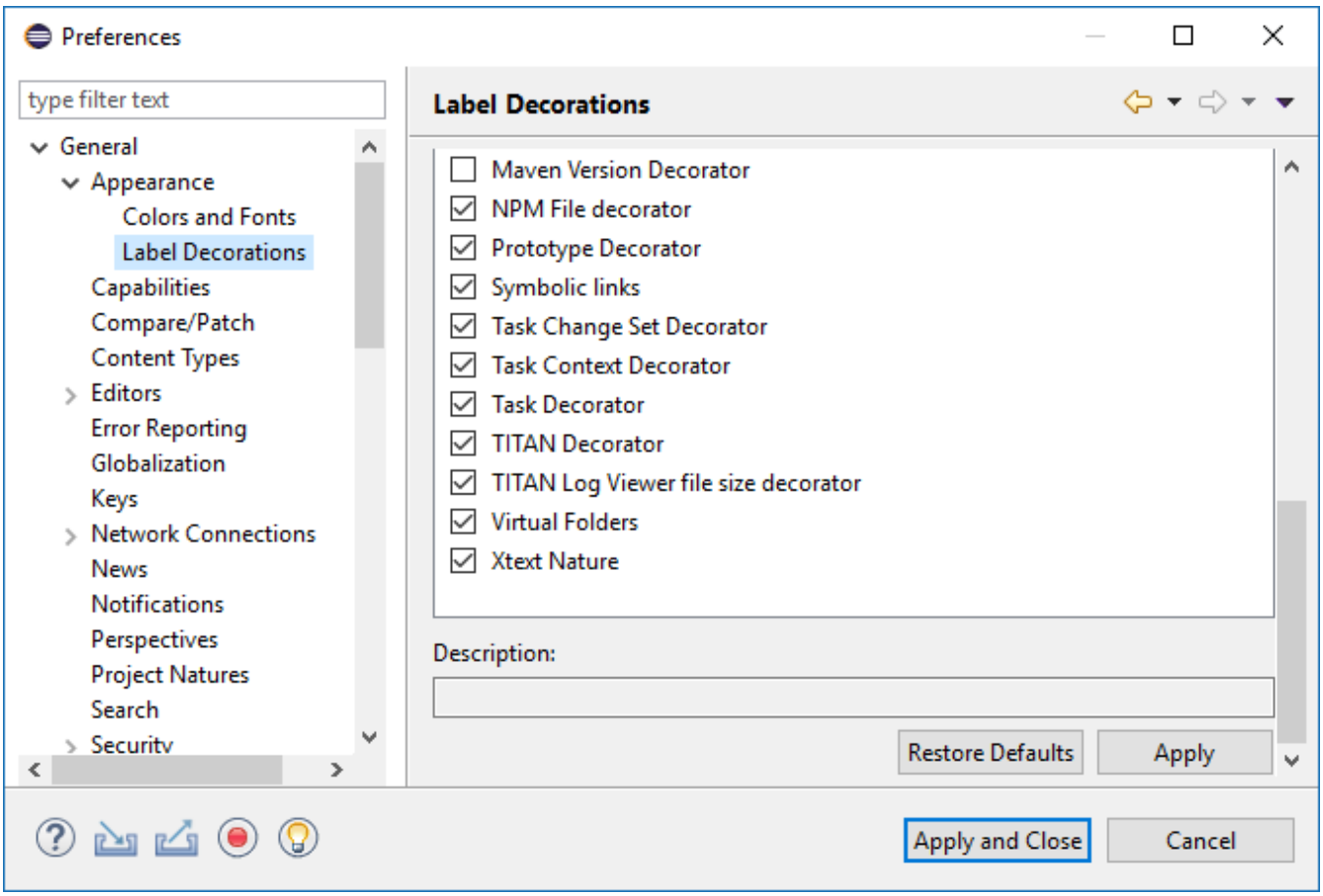


Figure 7. Enabling TITAN Decoration

NOTE	Decorations are extending the information displayed for elements. As there can be several decorations extending an element, the texts shown above might not be the only ones displayed.

2.5. Excluding resources

The possible reasons for a resource being excluded from build are as follows:

- Excluded by user:

  These resources were explicitly excluded from the build by the user. (For more information

refer here)

- Excluded as working directory:

  The working directory by definition is excluded from the build process, in order to make sure, that source files and generated file do not mix.

- Excluded by regexp:

  The names of these resources was matching one or more exclusion regular expressions provided on the **Excluded resources** preference page (for more information refer here.)

- Excluded by convention:

  On the Eclipse platform if the name of a resource (either a file or a folder) starts with a dot, it indicates that the resource is some special resource used by one of the plug-ins exclusively. All other plug-ins should exclude these files from their operation; they should not be regarded as part of the project by any plug-in other than its creator.

NOTE	When either the excluded resources or the working directory filter is active, it is indicated by the projects being decorated with the "[filtered]" decoration too. For more information on these filters please refer here.

Chapter 3. Setting Workbench Preferences

This section gives an overview about the various settings related to the workbench provided by the TITAN Designer plug-in.

In Eclipse, workbench preferences are used to set user specific general rules, which apply to every project; for example, preferred font styles, access to version handling systems and so on.

Workbench preferences are accessible selecting **Window** / **Preferences**. Clicking on the menu item will bring up the preferences page. The opening window contains a preference tree on the left pane to ease navigation (see figure below).

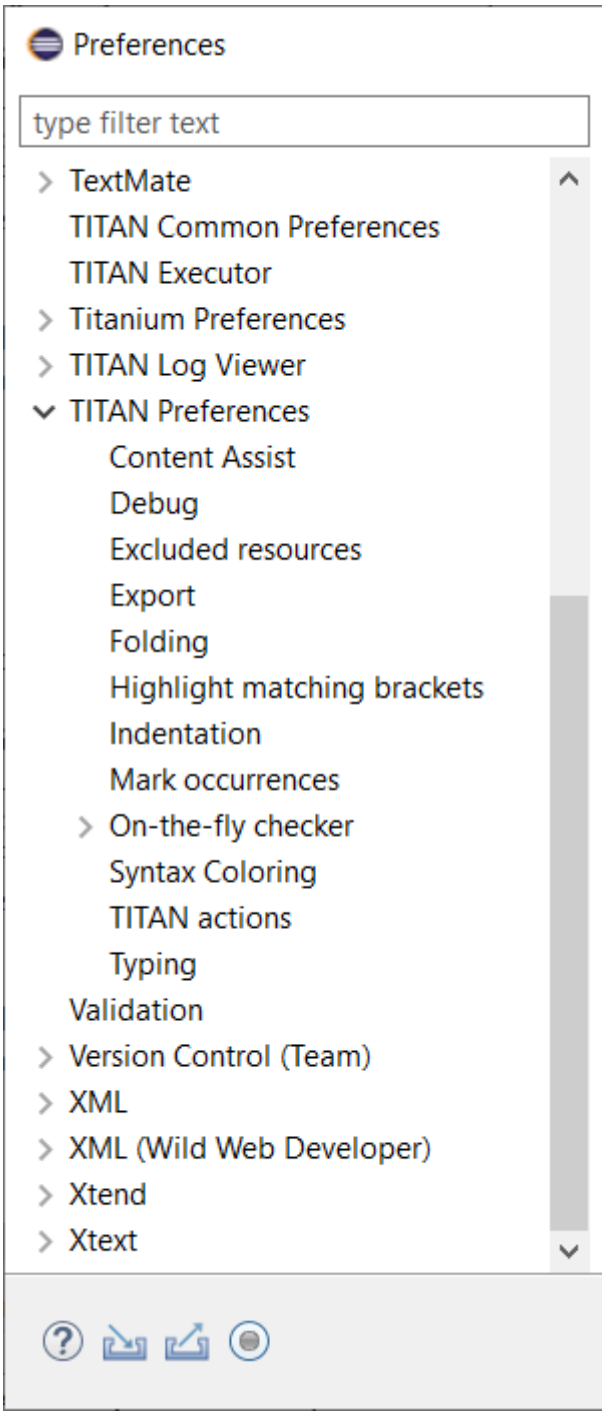


Figure 8. TITAN preferences sub-tree

This section only concerns the preferences that are available under the TITAN preferences node of this preference tree.

3.1. TITAN Preferences

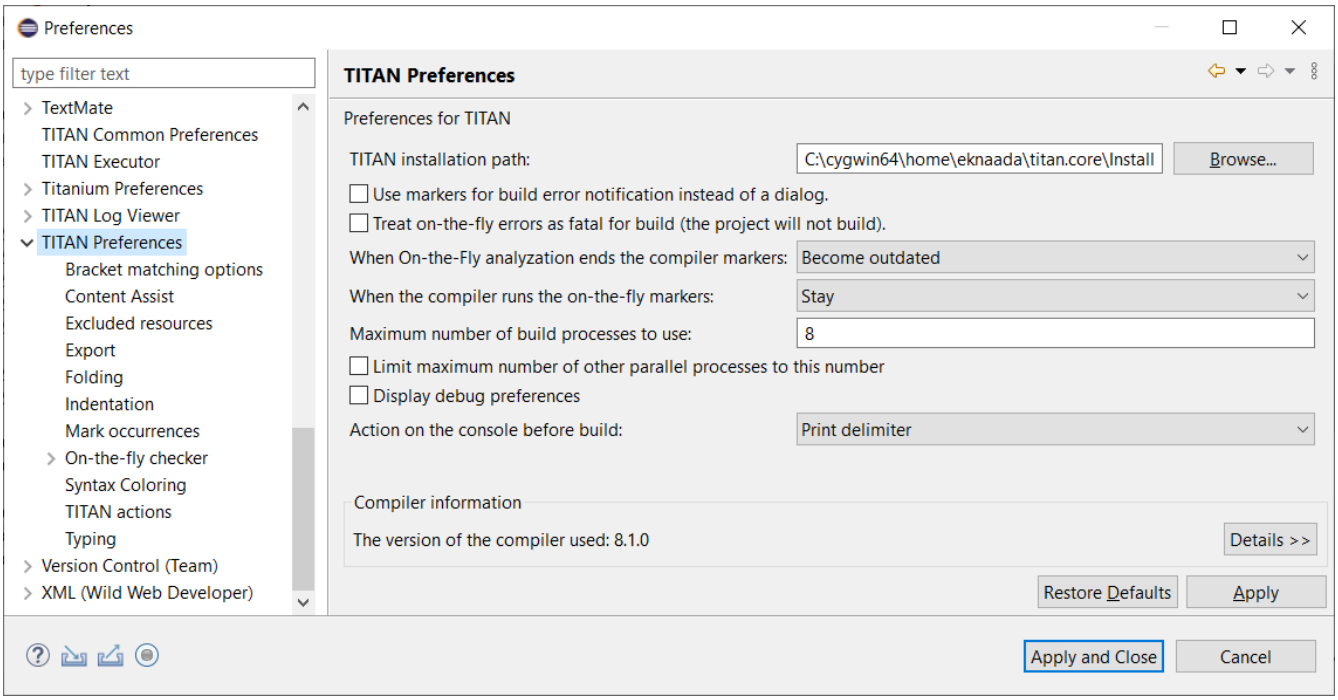


Figure 9. TITAN preferences

The following options can be set on the TITAN preferences page (see the figure above):

- **TITAN installation path.**

  The path to the TITAN installation directory. The TITAN version used to build the projects can be changed by modifying the contents of this field. The **Browse** button can be used to browse the directories.

- **License file.**The path must point to a valid TITAN license file. The **Browse** button can be used to browse the files.

  This option is not available in all versions.

- **Use markers for build error notification instead of dialog.**

  By default, an error during the build process is reported in a dialog window. However, this is sometimes the unwanted behavior; for example, when a job is running in the background. If this option is checked, no dialog window will pop-up; instead, an error marker will be placed on the project resource, seamlessly integrated into the general error processing behavior of the tool. The error message is assigned to the marker in this case.
  The option is UNCHECKED by default.

- **Treat on-the-fly errors as fatal for build.**

  By default if the on-the-fly analyzer recognizes a syntactic or semantic error, that has no effect on the build process of the project. However, most of the time this is not optimal behavior,

because if the semantic analyzer finds something erroneous, the build process will also find it erroneous and as such the build process will not be able to fully complete (plus in such cases the time spent by the build process to detect and report the problem is actually wasted as the problem is already known).
The option is NOT CHECKED by default.

- **When on-the-fly analysis ends the compiler markers.**

  When the on-the-fly analyzer starts it can trigger the following behaviors for error markers generated by the compiler previously: **“Stay unchanged”**, **“Become outdated”**, and **“Are removed”**.
  The default setting is: **“Become outdated”**

- **When the compiler runs the on-the-fly markers.**

  When the compiler starts it can trigger the following behaviors for error markers generated by the on-the-fly analyzer previously: **“Are removed”**, **“Stay”**. Setting this option to **“Stay”** can enhance the speed of the on-the-fly analyzer, because if the markers need to be refreshed, so does all syntactic and semantic information needs to be refreshed too.
  The default setting is: **“Are removed”.**

- **Maximum number of build processes to use.**

  By default, the build process is only executing in one process which is not efficient on modern multi-core hardware. Using this option the users can set how many parallel processes shall be used by the build process at the same time to compile modules.
  The option is set equal the number of processors/cores available in the system by default.

- **Limit maximum number of other parallel processes to this number.**

  During the initial processing of projects several threads are created to utilize the parallelism of modern CPUs and improve performance. However, in some cases the number of created threads exceeds the OS or user thread/process limit that results in Out Of Memory exception. This option limits the number of parallel processes to the previous number.

- **Display debug preferences**

  By default, the Designer plug-in isn’t logging debug information to the Debug Console to help solving problems. However as errors are reported to the Error Log of Eclipse this information is rarely used. Most of the time these printouts hold no value for the users. Debugging and load balancing features can be set by this option see here.

  The option is NOT CHECKED by default because most of the time these features hold no value for the users.

  If you want to set any of these options, set the options "Display debug preferences" then press button "Apply". An entry "Debug" appears under "**TITAN Preferences**" on the left pane (see the figure below).

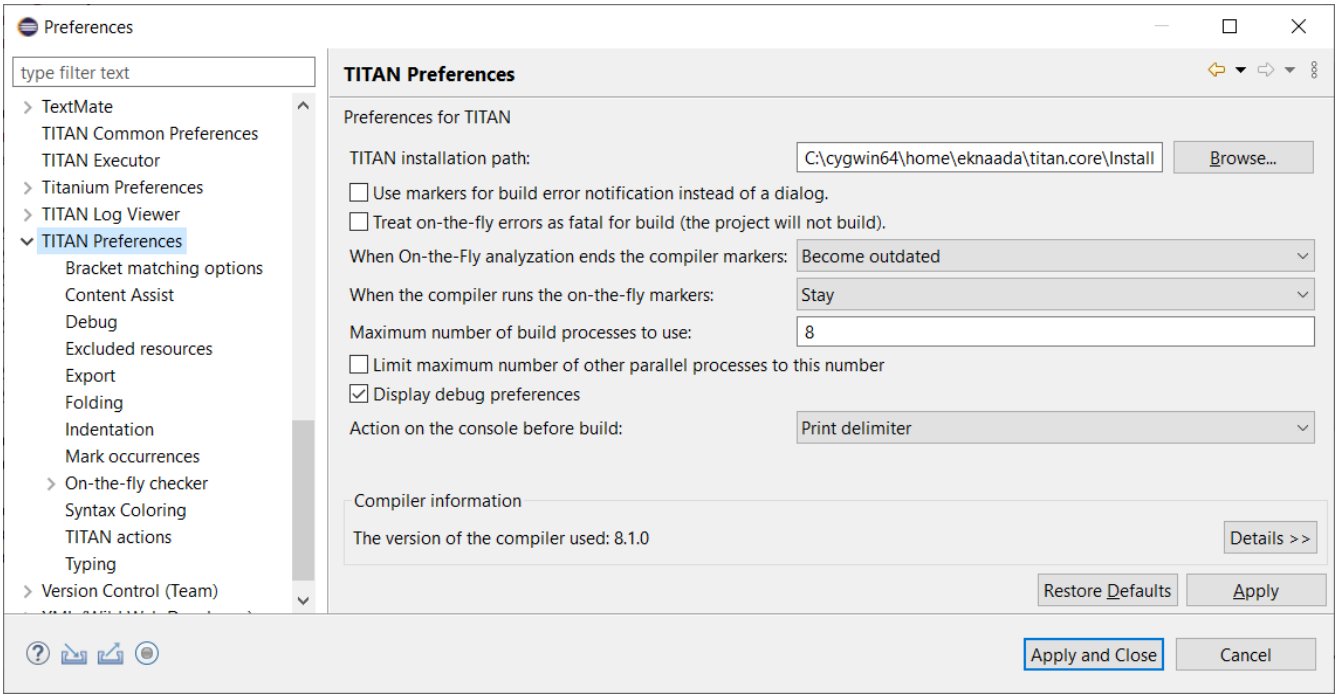


Figure 10. Display Debug preferences

Below the last option, the compiler information section is present containing the version of the currently set compiler and information about the license of the user is displayed. The "Details" button shows more information about the configured compiler that can be particularly useful when there are some problems with the configured compiler or with the user license (see the figure below).

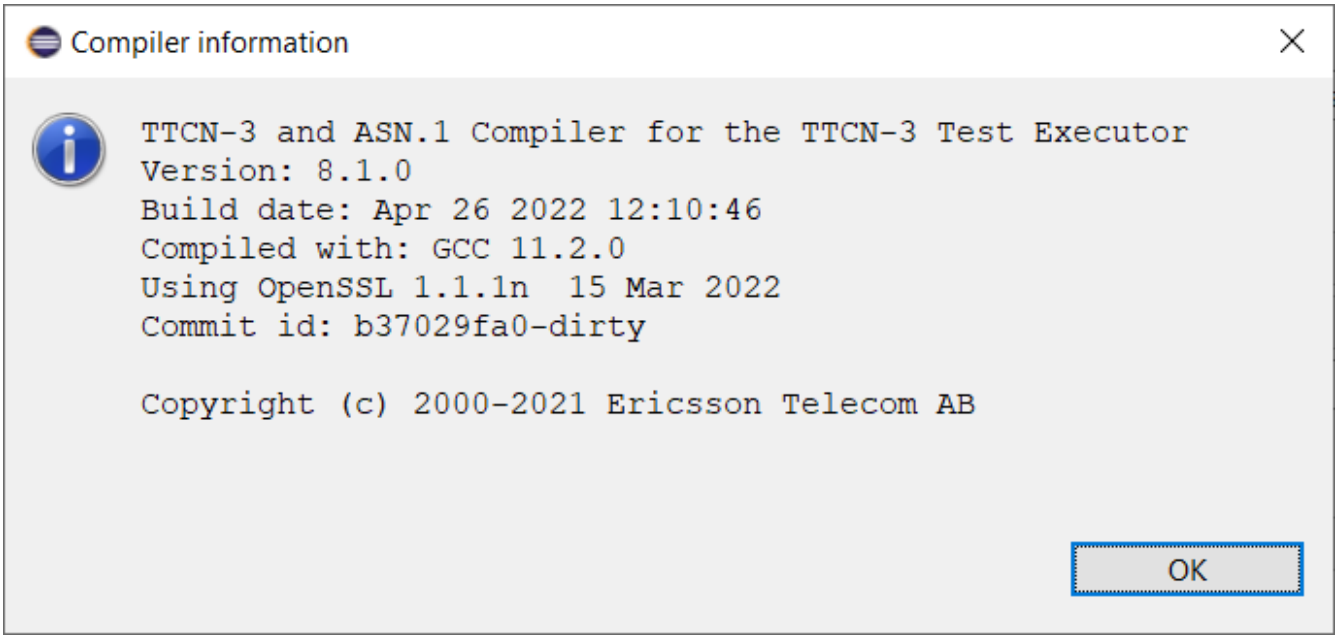


Figure 11. Compiler information

NOTE In case the license file is not provided, is not valid or has expired an additional link will appear on this page. Clicking on this link a browser will open directing the user to a web page where he can order a new license or can ask for a renewal of his existing one.

3.2. Bracket matching preferences

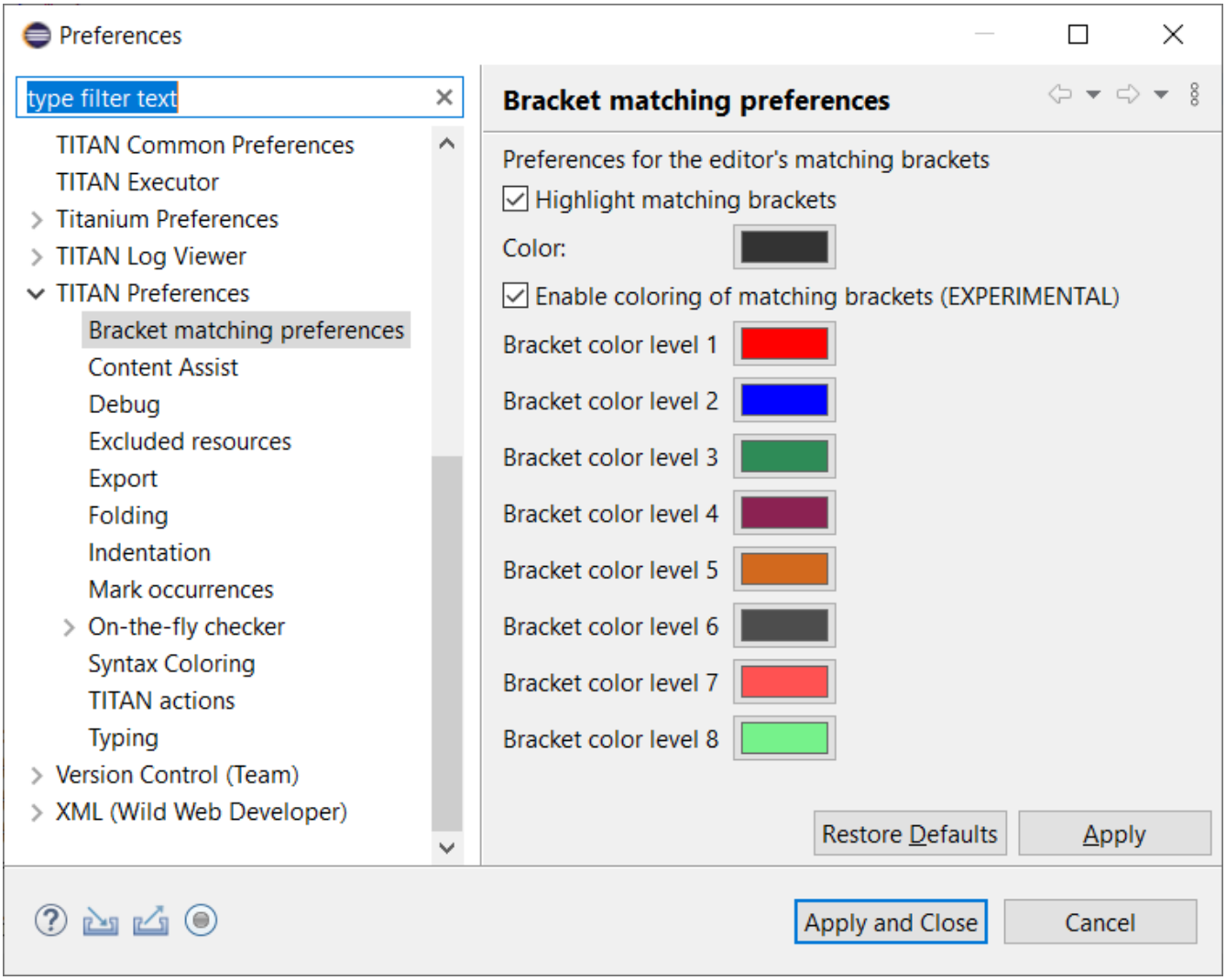


Figure 12. Bracket matching preferences

The following options can be set on the Bracket matching preferences page (see the figure above):

- **Highlight matching brackets**

  Checking this option enables highlighting of matching round, square and curly bracket pairs.

- **Color**

  The highlighting color is selected with this option.

- **Enable coloring of matching brackets**

  Checking this option enables coloring of matching round, square and curly bracket pairs according to their level of depth.

NOTE	This preference depends on the semantic highlighting option, i.e. that must be enabled to have effect of this preference. It might be necessary to reopen the current file in the editor to enable re-parsing of the file and applying the change.

- **Bracket color level x**

The highlighting color related to the specified level of depth is selected with this option. If the bracketing is more than eight level deep, the coloring starts over from level 1 (see <<,the figure below>>).

```
module coloring {

function f() {
  var integer i := f2(((((({a[1]})))));
}

}
```

Figure 13. Coloring of matching brackets

3.3. Content Assist Preferences

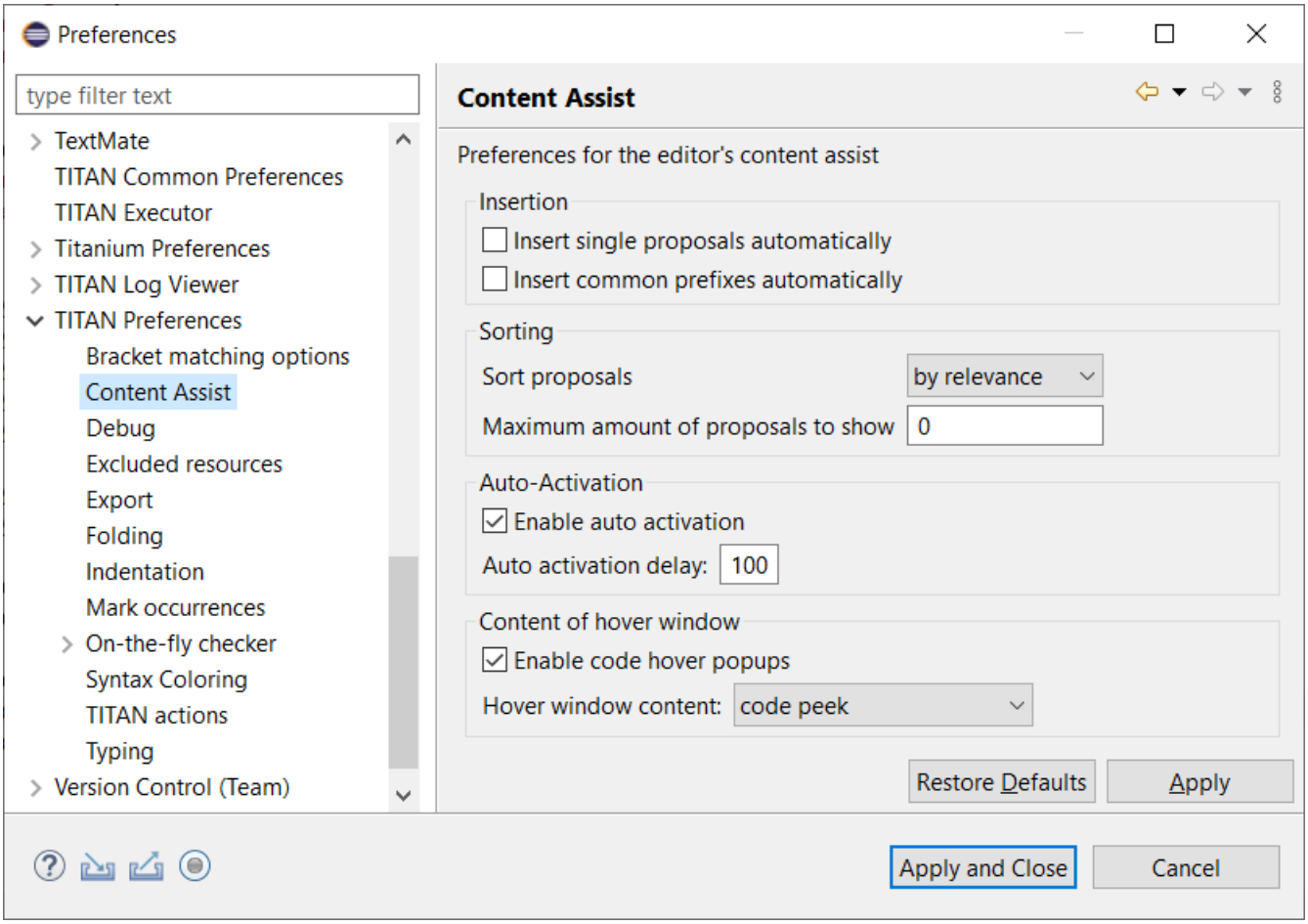


Figure 14. Content Assist

The following options can be set on the Content Assist page:

- **Insert single proposals automatically**

  When the analysis finds only one possible proposal to show to the user, it can be set whether it should be inserted automatically, or displayed anyway.
  This option is NOT CHECKED by default.

- **Insert common prefixes automatically**

  Very often all of the listed proposals start with a common prefix, that is longer than the text being extended (for example naming conventions usually have such prefixes).
  In such cases if this option is checked, the common prefix will be inserted automatically. This way the user only has to enter those characters that actually differentiate between two options, allowing finishing with the actual code completion much faster.
  This option is NOT CHECKED by default.

- **Sort proposals**

  The sorting of the proposals can set to be done either "**by relevance**", or "**alphabetically**".
  If ordered by relevance definitions that were declared closer in the scope hierarchy will be closer to the top of the proposal list. When the aim of the code compilation is usually a local variable, using this sorting method it can be found much faster.
  If ordered alphabetically all of the items will be in alphabetical order, although not as fast in completing local definitions, it might be easier to search for most people.
  The default setting is: "**by relevance**".

- **Enable auto activation**

  The code completion cannot only be activated by the user by pressing CTRL + SPACE, but it can also be set to be automatically activated every time the `.' character is entered.
  This option is CHECKED by default.

- **Auto activation delay**

  The delay between the auto activation of the content assistant, and its actual starting can be set here in milliseconds.
  The default setting is: 500 milliseconds

- **Enable code hover popups**

  If enabled, a popup window is shown when hovering over different elements of the source code. The content is context-dependent and presents relevant information related to the selected part of the source code. See also the next preference.

- **Hover window content**

  The content of the popup window can be specified by this preference. This option can be directly changed using the shortcut button of the hover window (see the figure below). Two types are implemented:

  - code comment and info: relevant information is shown based on the documentation comments (if available) and the semantic information related to the selected part of the source code.
  - code peek: the definition of the selected code part is shown in the popup window.

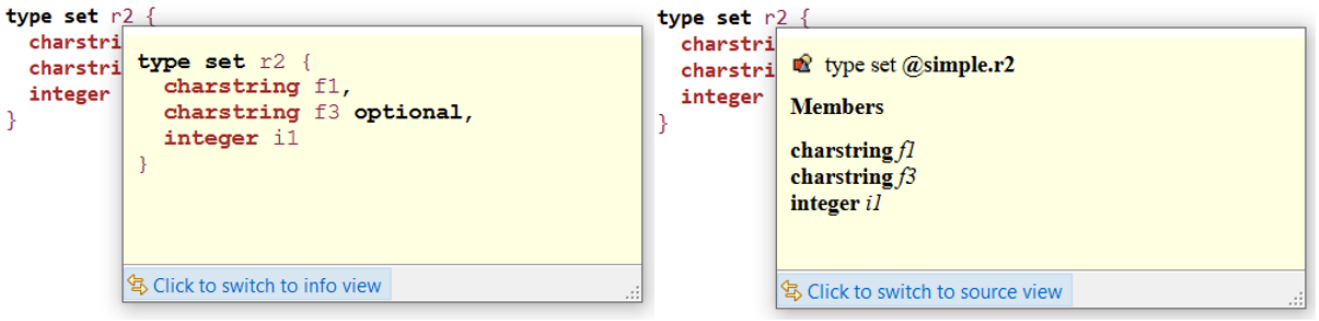


Figure 15. Hover window content

3.4. Debug

Please note that this option is only available if you enable "Display debug preferences" under "TITAN Preferences" (see here).

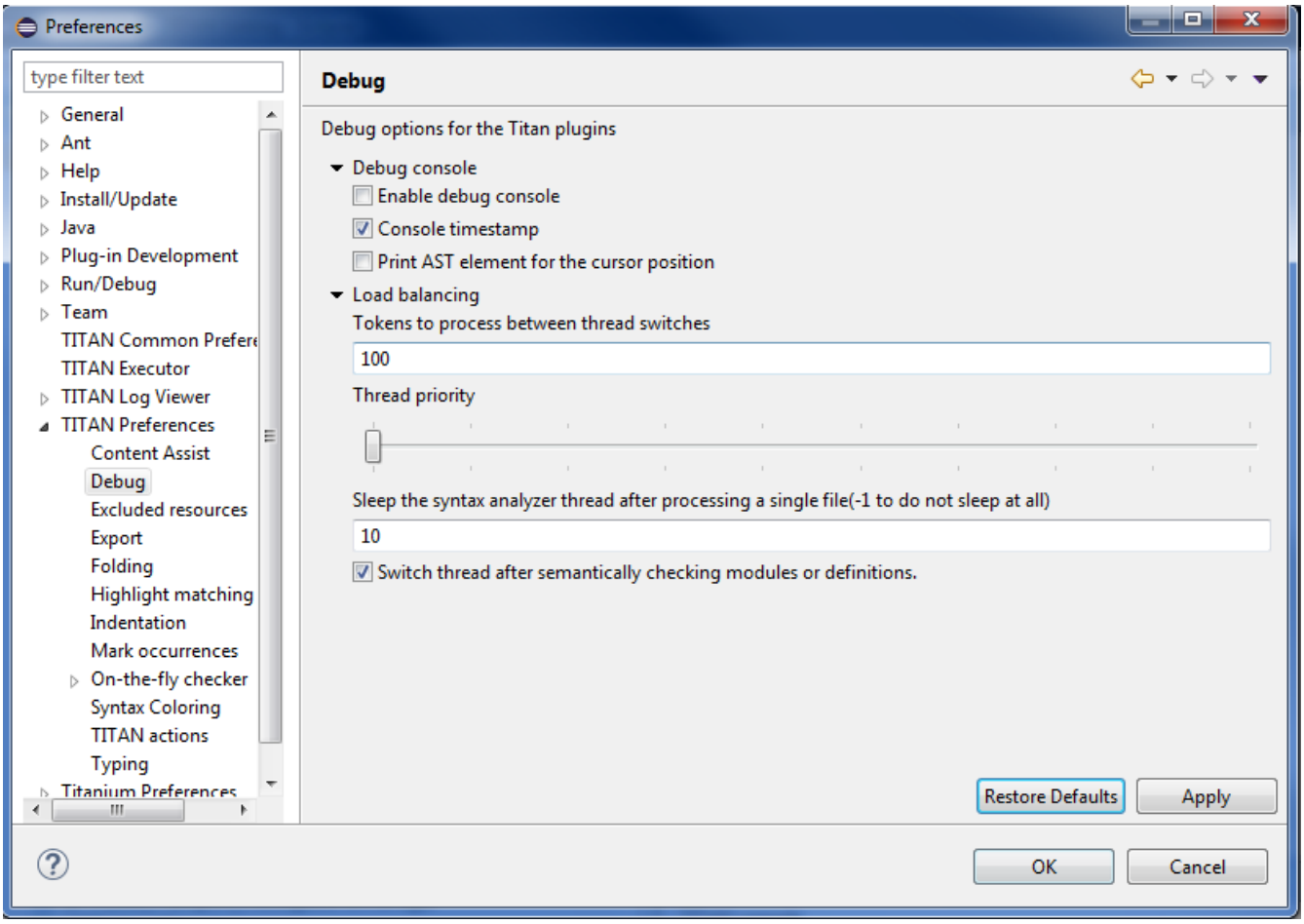


Figure 16. Debug options and Load Balancing

The following option groups can be set on the Debug page (see figure above):

- **Debug options for the Titan plugins**

  The elements of this group are rarely used but they are very useful in error reporting to the Eclipse Titan plugin developers. These settings affect the output on the **Debug Console** view.

- **Load balancing**

These options can be useful for advanced users to speed up the semantic analyzer in case of huge projects.

The Debug options are as follows:

- **Enable debug console**

  Enables the output on debug console

- **Console timestamp**

  Timestamps are inserted before each debug line

- **Print AST element for the cursor position**

  This debug information can be sent to the Eclipse Titan plugin developer as useful information to localize a bug.

  Example: The following figure shows a Debug console log with timestamp and AST element info in the first three lines.

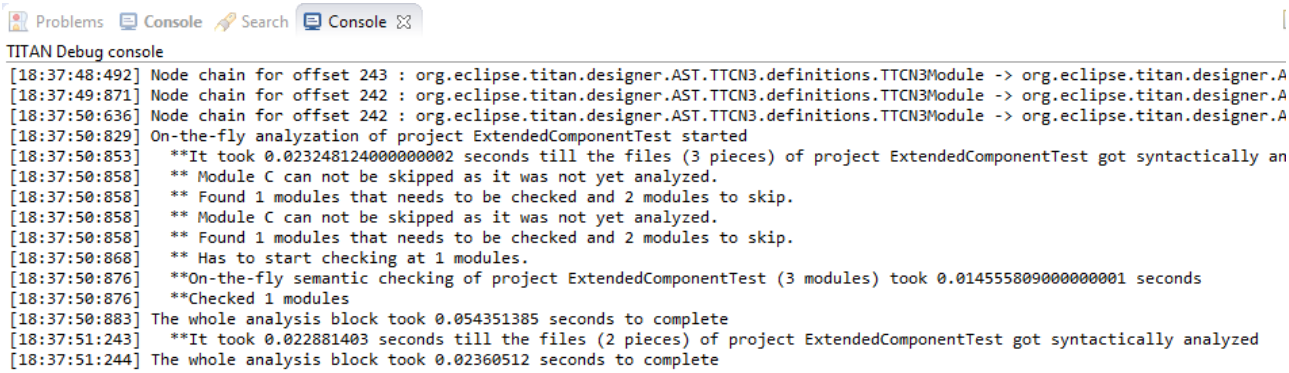


Figure 17. Debug Console log example

The Load balancing options are as follows:

- **Tokens to process between thread switches**

  Sets how many tokens shall process between switching threads. It can modify the speed of the analysis. Higher values are equivalent to faster file processing, lower values to lesser system load.

  Its default value is 100.

- **Thread priority**

  Sets the Java priority of the lexical analyzer related to other applications, leftmost being lowest, rightmost highest priority.

  Its default value is lowest.

- **Sleep the syntax analyzer text after processing a single file (-1 to do not sleep at all)**

Sets the length of sleep call after the lexical analysis of each file; Longer value means longer analysis but other activities are more possible.

Its default value is 10 ms.

- **Switch thread after semantically checking modules or definitions**

  Gives the chance to other threads (activities) to work.

3.5. Excluded Resources

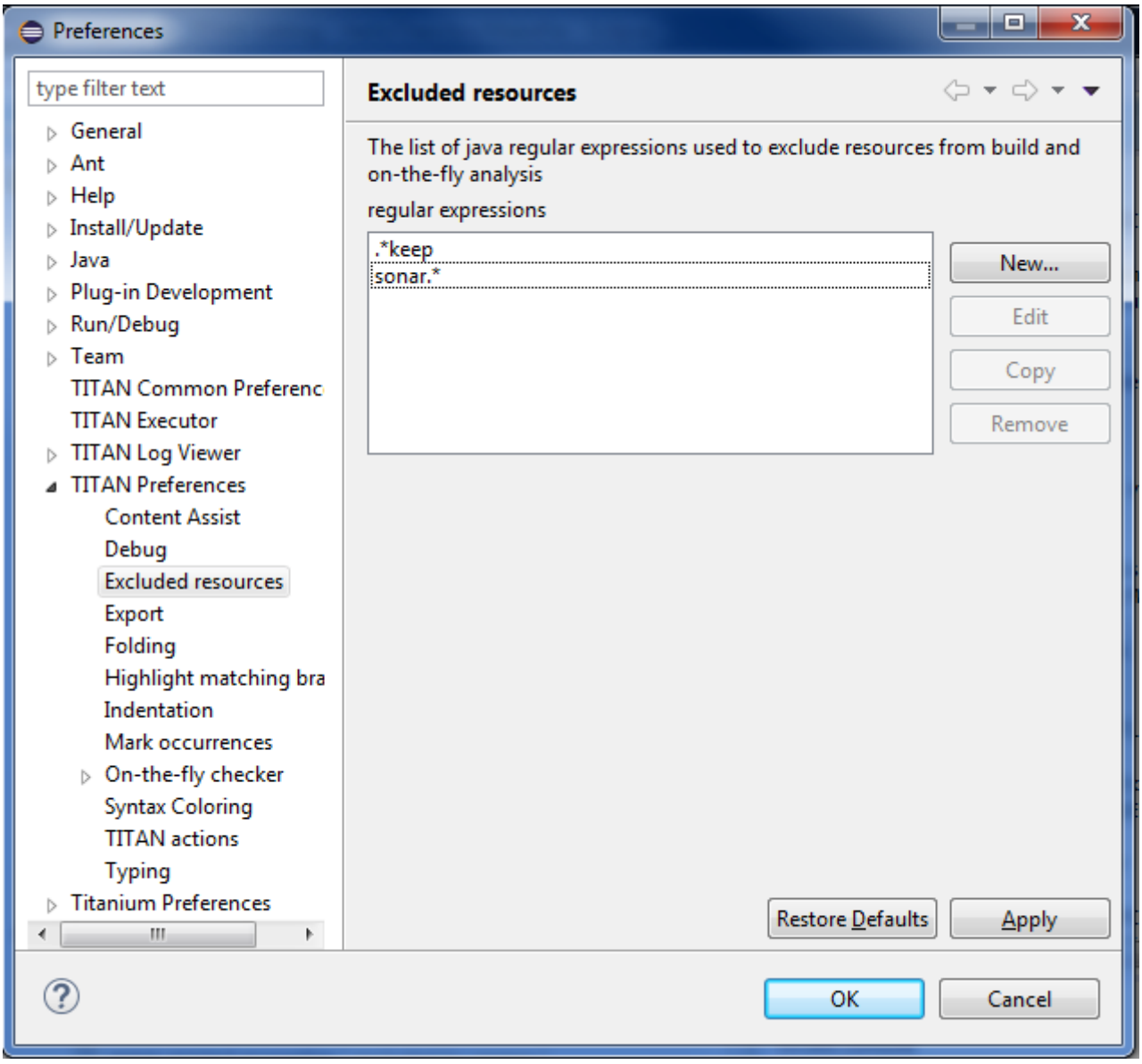


Figure 18. Excluded resources

On the excluded resources page, it is possible to provide a list of regular expressions, which should be used to exclude resources from build in the workspace. If even just one of the regular expressions matches on a name of a resource it will be excluded from build.

NOTE	The regular expressions are to be provided in the Java regular expression format.

3.6. Export

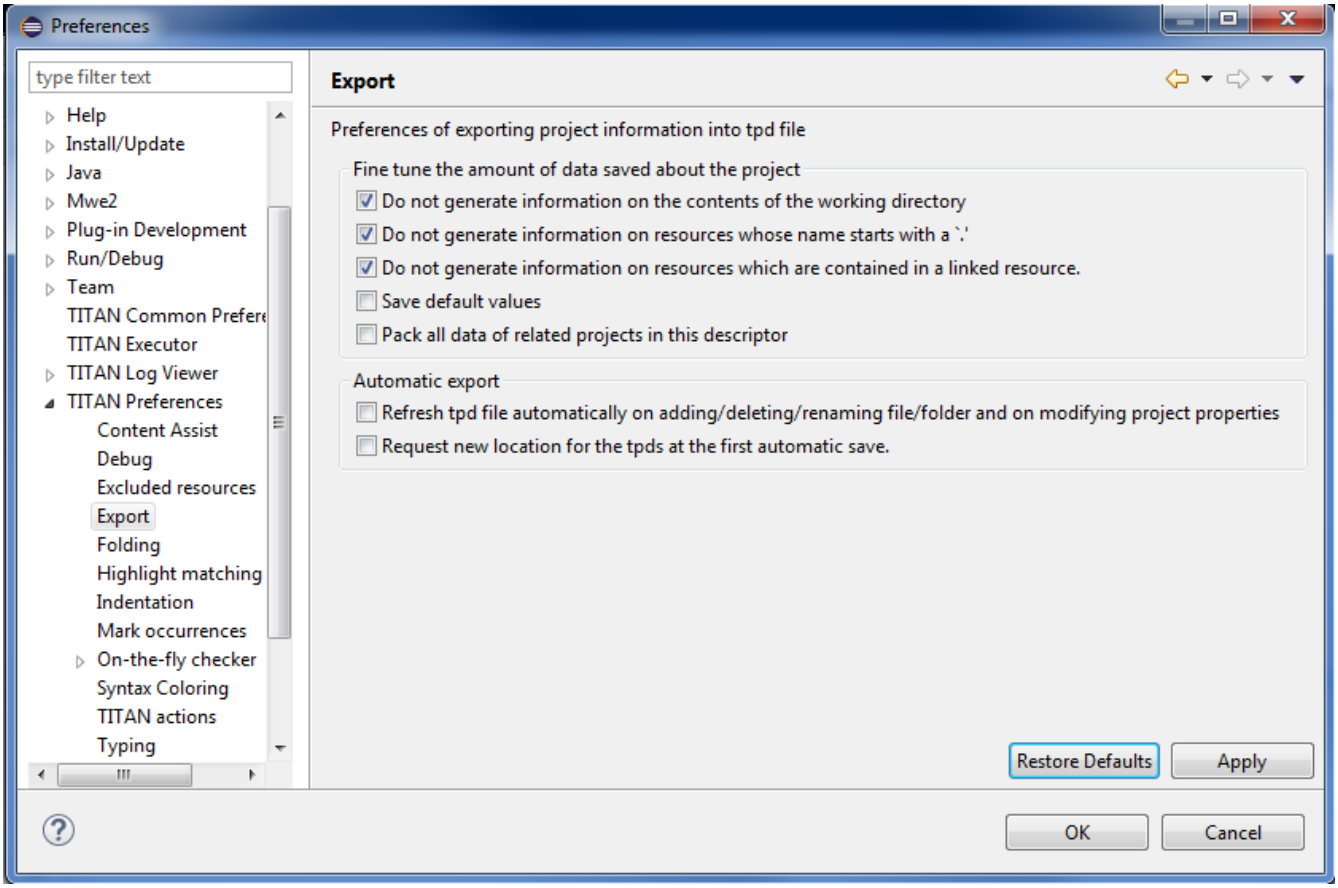


Figure 19. Export options

The export options contain 2 groups of settings.

The first group contains the export fine tuning options on workspace level. Their values are used in manual project export as default values and in automatic export as values. (Their names are the same as the option names of manual export dialog (see here).

The options in the first group are as follows:

- **Do not generate information on the contents of the working directory:**

  If the working directory is visible inside Eclipse, inside the project, its contents are by default also mentioned in the project description. As the working directory usually contains only generated files, that can be reproduced later, this behavior is not always desired. Its default value is on.

- **Do not generate information about resources whose name starts with a ".":**

  In Eclipse this naming convention is used to signal that a resource stores some tool specific options about the project. As such, from the point of view of TITAN, they are not needed. Its default value is on.

- **Do not generate information on resources contained within linked resources:**

  In many cases such links are intentionally used to connect to an existing folder whose content

might change externally. For example, version handling of files can also be done like that.

NOTE	It is recommended to use this feature with care: as there is not much connection between the Eclipse internal resource system, and the file system, the activation of this option can cause unexpected side effects. Its default value is on.

- **Save default values:**

  By default we do not include any information on any option/setting in the descriptor file, which has its default value as the actual one. This makes for a very compact description, but in cases where all information needs to be saved, this might not be ideal. Its default value is off. ** If it is switched on, the size of the tpd file is unnecessarily big. This is not a problem but perhaps it is not so easy to analyze by the user.

- **Pack all data of related projects:**

  Project references in Eclipse are a great way to structure one's work into manageable pieces. However, if one of those projects is not available, building the whole set is not possible. For this reason, it is possible to save all information from all required projects into one project descriptor. Its default value is off.

The second group contains the settings for automatic export.

The options are as follows:

- **Refresh tpd file automatically on adding/deleting/renaming file/folder and on modifying project properties**

  Choose this option if you want to have up-to-date tpd files in your workspace. This is useful if you want to store information of your project in tpd files and the content of your projects changes frequently.

- **Request new location for the tpds at the first automatic save**

  This option works if the previous option is set. Choose this option if you want to change the location of the tpd file while it is being imported or if you want to specify the location of new tpd files at the first automatic save. The automatic save shall not work if it is not set and the project does not have a tpd file yet. This way the automatic save can work only on a subset of the projects.

3.7. Folding Preferences

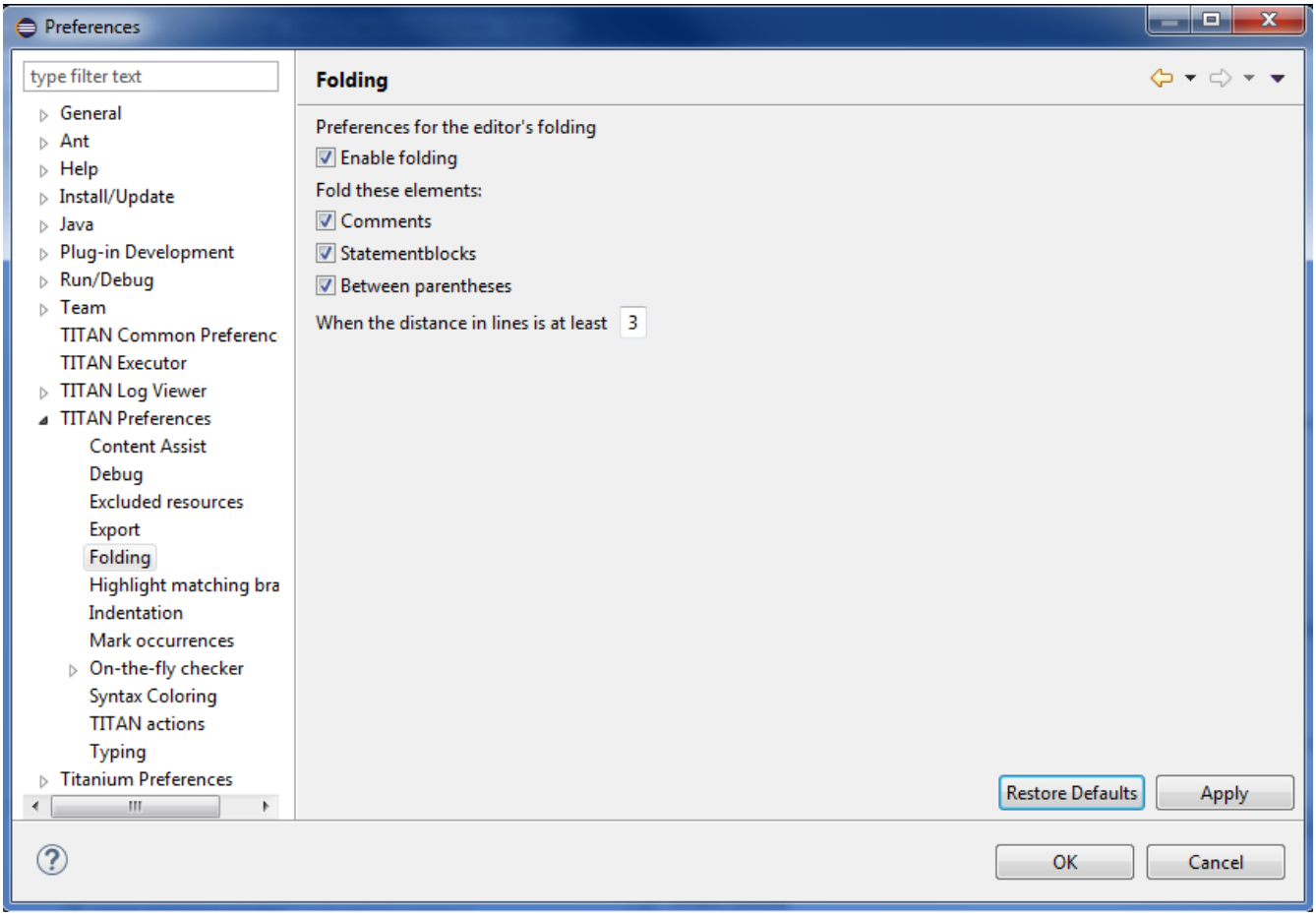


Figure 20. Folding preferences

The following options can be set on the Folding page (see the figure above):

- **Enable folding**

  Line folding can generally be enabled or disabled with this option.

NOTE	folding is called upon when parsing a modified file; thus, disabling this feature may somewhat speed up file processing.

- **Comments**

  Comments, that is, text between /* and */ will be folded if both this option and **Enable folding** are checked.

- **Statement blocks**

  Statements blocks, that is, {text between curly brackets} will be folded if both this option and **Enable folding** are checked.

- **Between parentheses**

  Parameters, that is, (text between parentheses) will be folded if both this option and **Enable folding** are checked.

- **When distance is at least**

This option disables the folding of a region unless there are at least that many lines between the ending and starting lines of the region.

3.8. Indentation Preferences

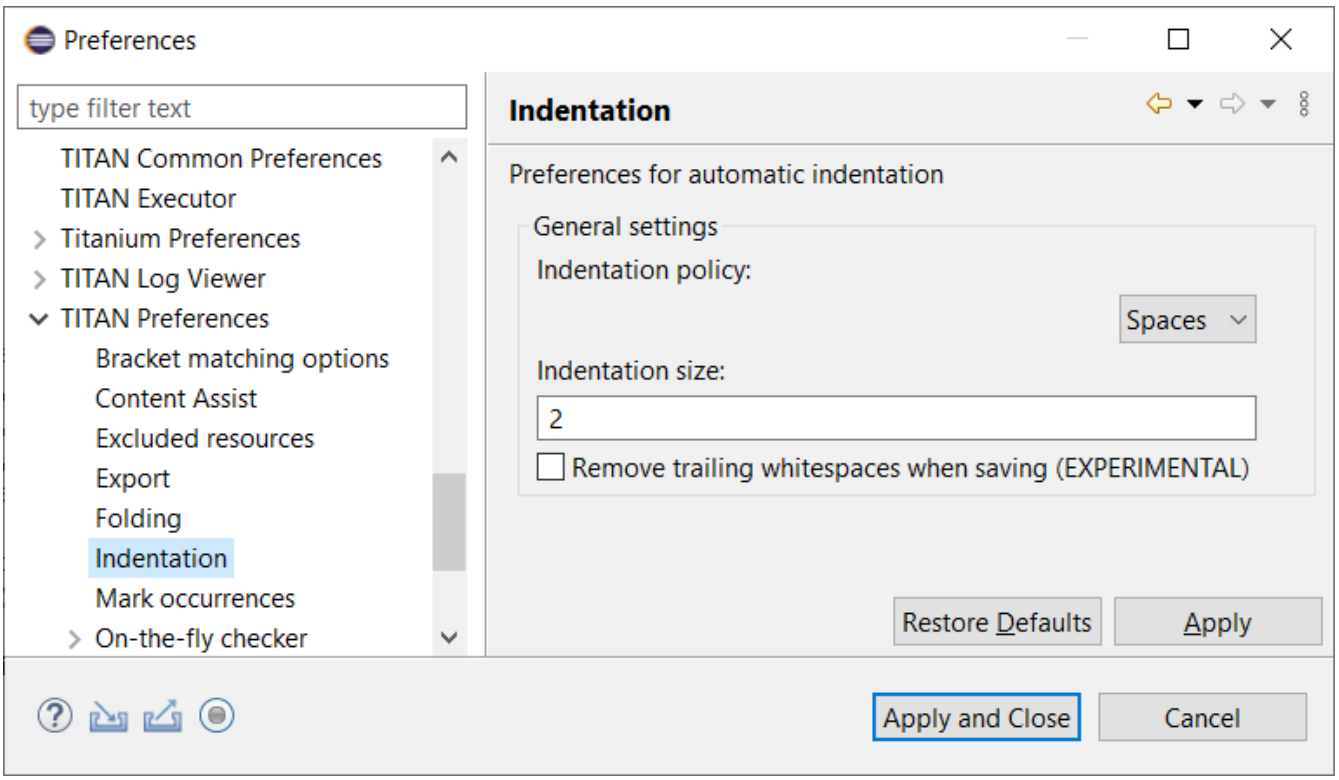


Figure 21. Indentation preferences

Indentation rules (valid for each editor provided by the TITAN Designer plug-in) are set on the Indentation page (see the figure above):

- **Indentation policy**

  The drop-down list contains two options: **Spaces** and **Tab**. When **Spaces** is selected, indentation is done by inserting a number of spaces before the text; the number of space characters is determined in the field **Indentation size**. When **Tab** is selected, indentation is performed by inserting a single tabulator character before the text.

- **Indentation size**

  This field determines the number of spaces used for indentation. It is only enabled when the indentation policy is set to **Spaces**.

In the default indentation policy, a single indentation level corresponds to inserting two spaces.

- **Remove trailing whitespaces when saving**

  Enabling this feature allows removing the trailing whitespaces during saving.

3.9. Mark Occurrences

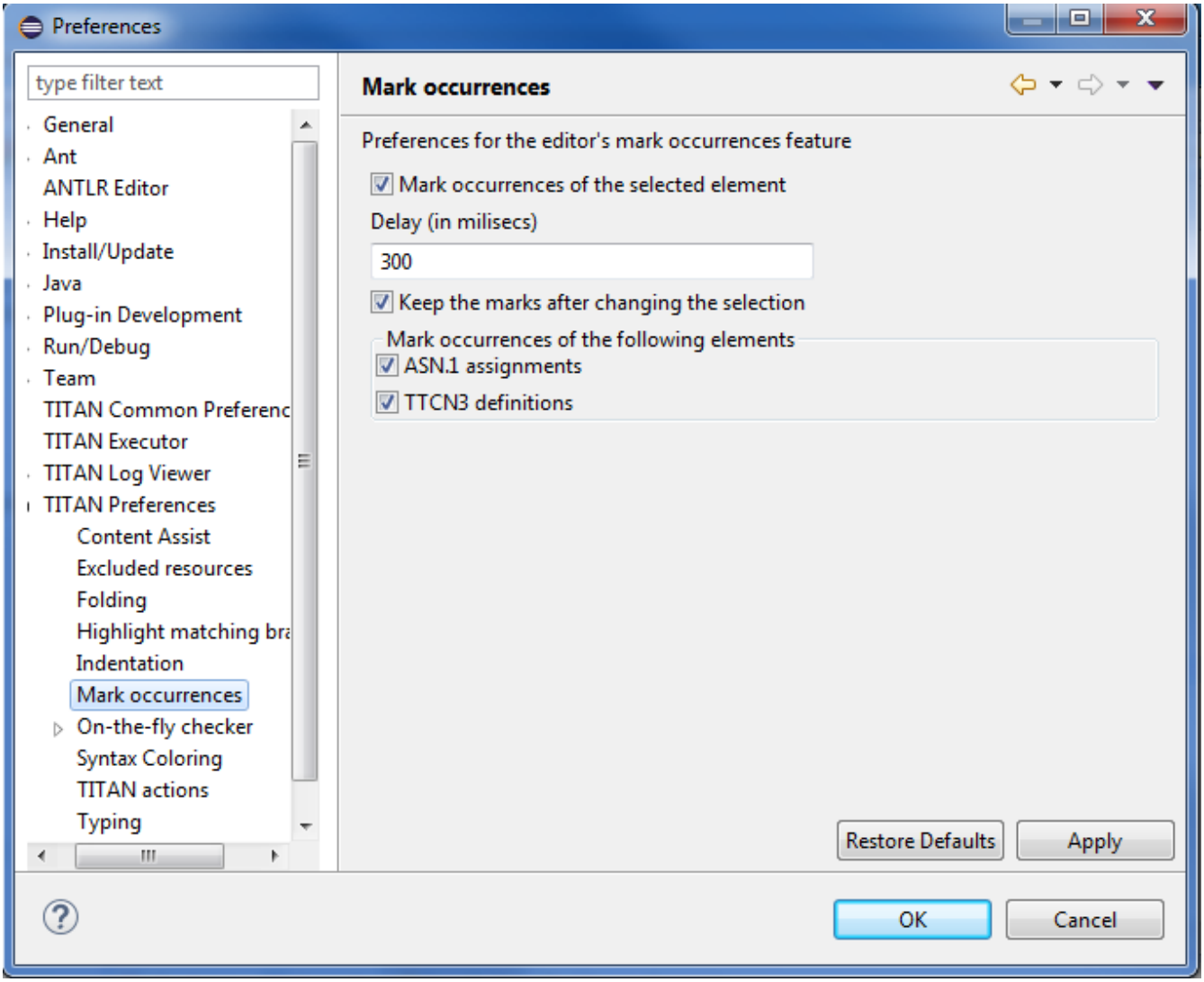


Figure 22. Indentation preferences

- **Mark occurrences of the selected element**

  the mark occurrences feature can be turned on/off by checking this checkbox. If this box is not checked, the other options are not available.

- **Keep the marks after changing the selection**

  if the selection or the position of the cursor changes and the occurrences of the newly selected element cannot be marked, the marks of the previous selection will stay visible.

- **Mark occurrences of the following elements**

  the occurrences of the selected elements will be displayed.

3.10. On-the-fly Checker Preferences

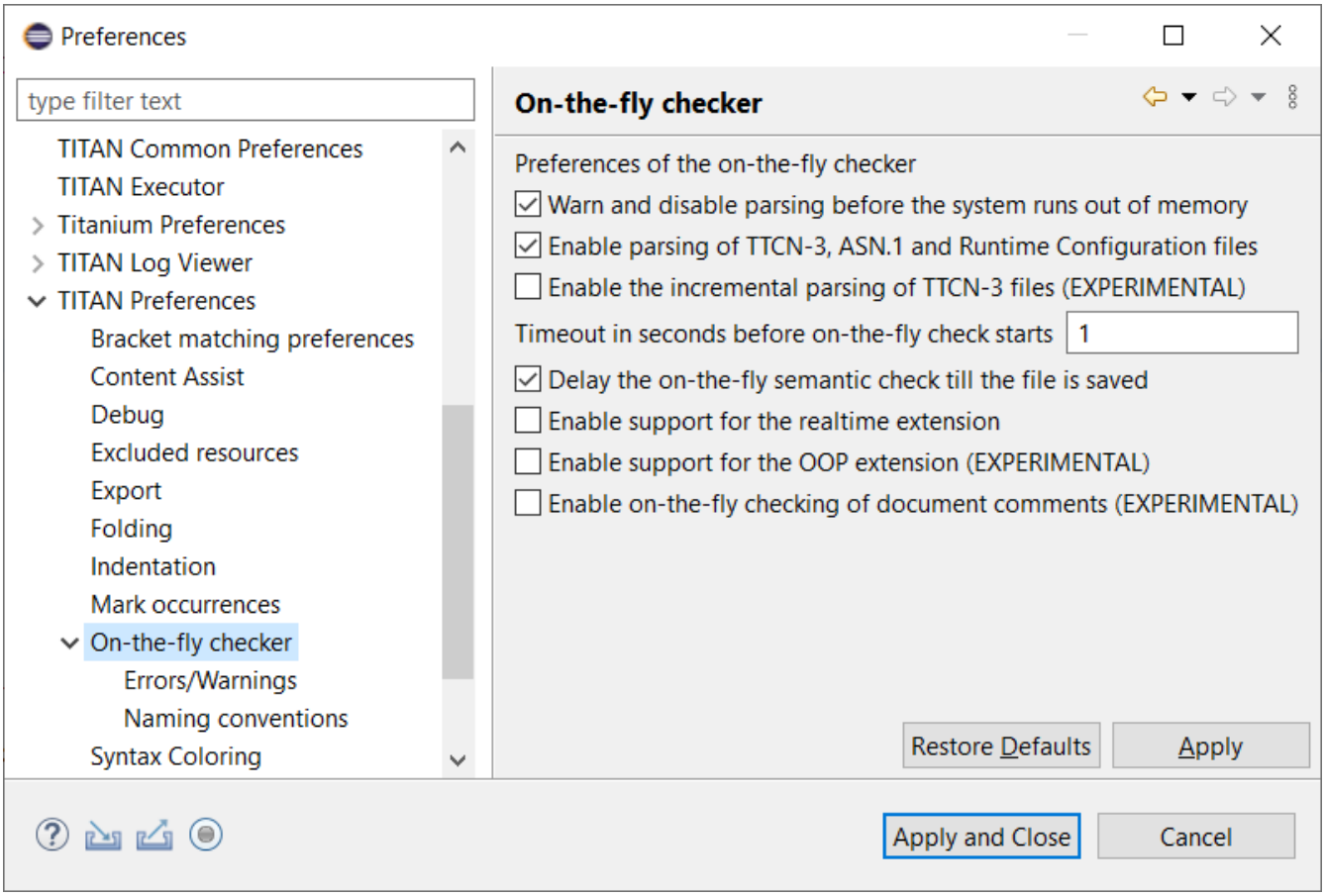


Figure 23. On-the-fly checker preferences

The following options can be set on the TITAN preferences page (see the figure above):

- **Warn and disable parsing before the system runs out of memory**

  If this option is enabled, the user is notified about the Eclipse IDE runs out of memory soon, and it is offered to disable the parsing to free up some memory. This option is CHECKED by default.

- **Enable parsing of TTCN-3, ASN.1 and Runtime Configuration files**

  Right now the on-the-fly parser might take a long time to run depending on the size or the amount of source files. For this reason, the parsing process can be disabled with this option, but disabling it will also disable most of the advanced features. This option is CHECKED by default.

- **Enable the incremental parsing of TTCN-3 files (EXPERIMENTAL)**

  By default when source code is modified the whole file needs to be syntactically re-analyzed, which can take up to a few seconds for large files. Incremental parsing tries to utilize the already existing syntactic and semantic information to speed up this process, by only re-analyzing a minimal part of the code whose semantic value might have changed because of the modification. When used correctly the length of the syntactic re-analyzing can be reduced to the `10^(-2)` second range, even for file of ten thousands of lines. It is still in experimental phase. This option is UNCHECKED by default.

NOTE	`ttcnpp` files are not analyzed incrementally even if incremental analysis is switched on.

- **Timeout in seconds before on-the-fly check starts**

  If the tool would start an on-the-fly check every time a character is entered or deleted, it would overload the machine, not letting the user to enter text. For this reason, the on-the-fly analyzer only starts up a few seconds after the last continuous editing has ended (the user stopped typing for a few seconds). In this option the length of this waiting period can be set. This option is set to 1 second by default.

- **Delay on-the-fly semantic checking till the file is saved**

  when this feature is enabled, the on-the-fly analysis done when the user edits something in a file will only involve syntactic checking, the semantic checking of the project is delayed until the file is saved. Usually there would be no need for this feature, however in huge projects the semantic checking can take a few seconds. In those cases, now the programmers will be able to edit their code with less overhead. There is however a bad side to this feature too: If there is no semantic checking the on-the-fly database is also not updated. This means that for example newly created local variables will only appear in code completion offerings after the file is saved. This option is CHECKED by default.

  The parsing process is detailed here.

NOTE	The delayed semantic checking separates the syntactic analysis from the semantic analysis, while the timeout before on-the-fly check starts feature shifts them together. As such these two features are orthogonal to each other.

- **Enable support for the realtime extension**

  When this feature enabled support for the realtime extension of the TTCN-3 standard will be activated. This also means that the now, realtime and timestamp words become keywords in TTCN-3 files.

- **Enable support for the OOP extension**

  When this feature enabled support for the OOP extension of the TTCN-3 standard will be activated. This also means that the now, e.g. class, this, super etc. words become keywords in TTCN-3 files.

- **Enable on-the-fly checking of document comments**

  When this feature enabled the document comments are parsed and analyzed on the fly as well according to the related TTCN-3 standard extension. The document comments are used for providing information upon hovering over a source code element, as well as upon showing the content assist (code completion). Furthermore, this allows for certain consistency checks between the code comments and the related semantic information, e.g. if the formal parameter list of a function differs from its commented one, the user can be notified.

3.10.1. Pitfalls

- In the worst case incremental parsing can actually take somewhat longer than a full parsing of

the file. As it is using among others the opening and closing brackets to localize the semantic effect of a change, if these are not used in a consistent way, which reduce the performance drastically. For example, if only the '{' sign is entered, but the pairing '}' is not, that might structurally damage the whole file, as all statement blocks might become syntactically invalid.

NOTE	Using the automatic typing features provided, and programming in a consistent way, can practically eliminate the chances of such performance degradations.

- It is very important to have the timeout before the on-the-fly check as low as possible. It can lead to strange phenomenon, if the text is modified too much between two checks. For example, code completion might believe that according to its outdated data the cursor has left a statement block, while in reality new statements were added to it, extending its size.

3.11. Errors/Warnings Preferences

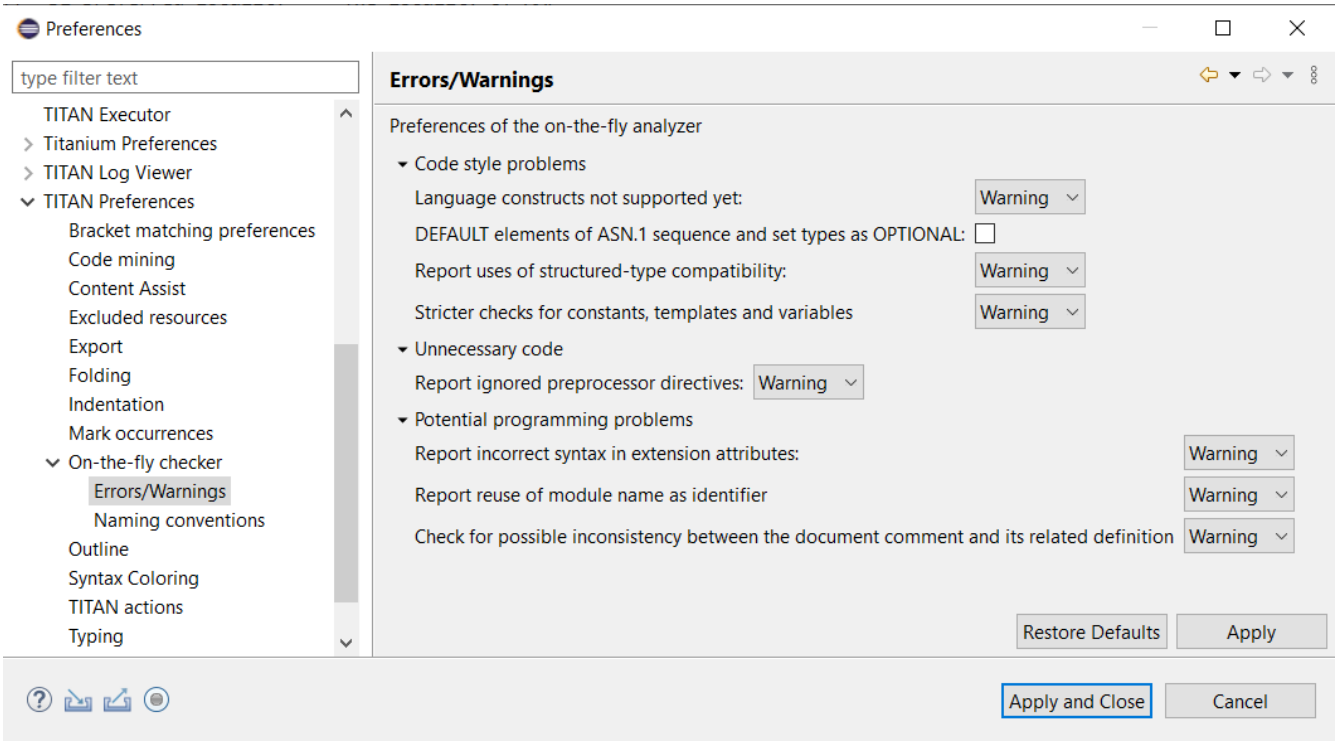


Figure 24. Errors / Warnings preferences

There are some situations which are not semantically erroneous in general, but in most of the cases they indicate bad coding practices or inefficient code. These checks in several cases are above the level of semantic checks. On-the-fly checker options determine TITAN behavior in such circumstances. These options are categorized in 3 groups based on the kind of problem they detect: code style problems, unnecessary code and potential programming problems.

NOTE	By default only the first group is in opened state.

These options are set on the On-the-fly checker page (see the figure above):

Code style problems:

- **Language constructs not supported yet**

The on-the-fly checker, suspecting unsupported language constructs, triggers one of the following behaviors: **Ignore**, **Warning**, **Error**. The default setting is: **Warning**.

- **DEFAULT elements of ASN.1 sequence and set types as OPTIONAL.**

  If this option is set the on-the-fly checker will handle elements of sequence and set types in ASN.1 modules with default values as if they were optional. The option is UNCHECKED by default.

- **Report uses of structured-type compatibility.**

  The on-the-fly checker, when a type compatibility check is detected, triggers one of the following behaviors: **Ignore**, **Warning**, **Error**. The default setting is: **Warning**.

- **Use stricter checks for constants, templates and variables.**

  Since version 4.2.1 of the TTCN-3 standard it is not required to completely initialize constant values and optional fields of records and sets, to allow more general operations. However, this also might introduce some hard to trace bugs. When such case is detected, it triggers one of the following behaviors: **Ignore**, **Warning**, **Error**. The default setting is: **Warning**.

Unnecessary code:

- **Report ignored preprocessor directives.**

  The on-the-fly checker, suspecting that preprocessor directive is ignored, triggers one of the following behaviors: **Ignore**, **Warning**, **Error**. The default setting is: **Warning**.

Potential programming problems:

- **Report incorrect syntax in extension attributes.**

  According to the standard syntax errors in the extension attribute should not be reported, but should be assumed as correct for some other tool. The on-the-fly checker, when a syntax error is detected in an extension attribute, triggers one of the following behaviors: **Ignore**, **Warning**, **Error**. The default setting is: **Error**.

- **Report reuse of module name as identifier**

  According to the standard the reuse of the module is not allowed, however technically it does not effect the code generation. The severity of this issue can be: **Ignore**, **Warning**, **Error**. The default setting is: **Error**.

- **Check for possible inconsistencies between the document comment and its related definition**

  During parsing the document comments, consistency checks are performed between the code comments and the related semantic information, e.g. if the formal parameter list of a function differs from its commented one, the user can be notified. The severity of such issues can be: **Ignore**, **Warning**, **Error**. The default setting is: **Warning**.

The on-the-fly checker is described in detail here.

NOTE	Changing these preferences will trigger a full re-checking of the projects already checked (when the changes are applied).

3.11.1. Pitfalls

The detection of unused module importations and definitions is based on the semantic analyzes done on-the-fly. As that is not yet a full semantic analyzes, these feature can also produce only heuristic behavior.

For example, every importation / definition will be reported unused, if it is not used by the semantic analyzer. This sadly does not mean that they are actually not used, but on the contrary it means that every importation / definition not marked is sure to be used. However this also means that if there are any unused importations / definitions in the project they will be contained in this list, thus considerably reducing every effort needed to find them.

3.12. Naming Conventions

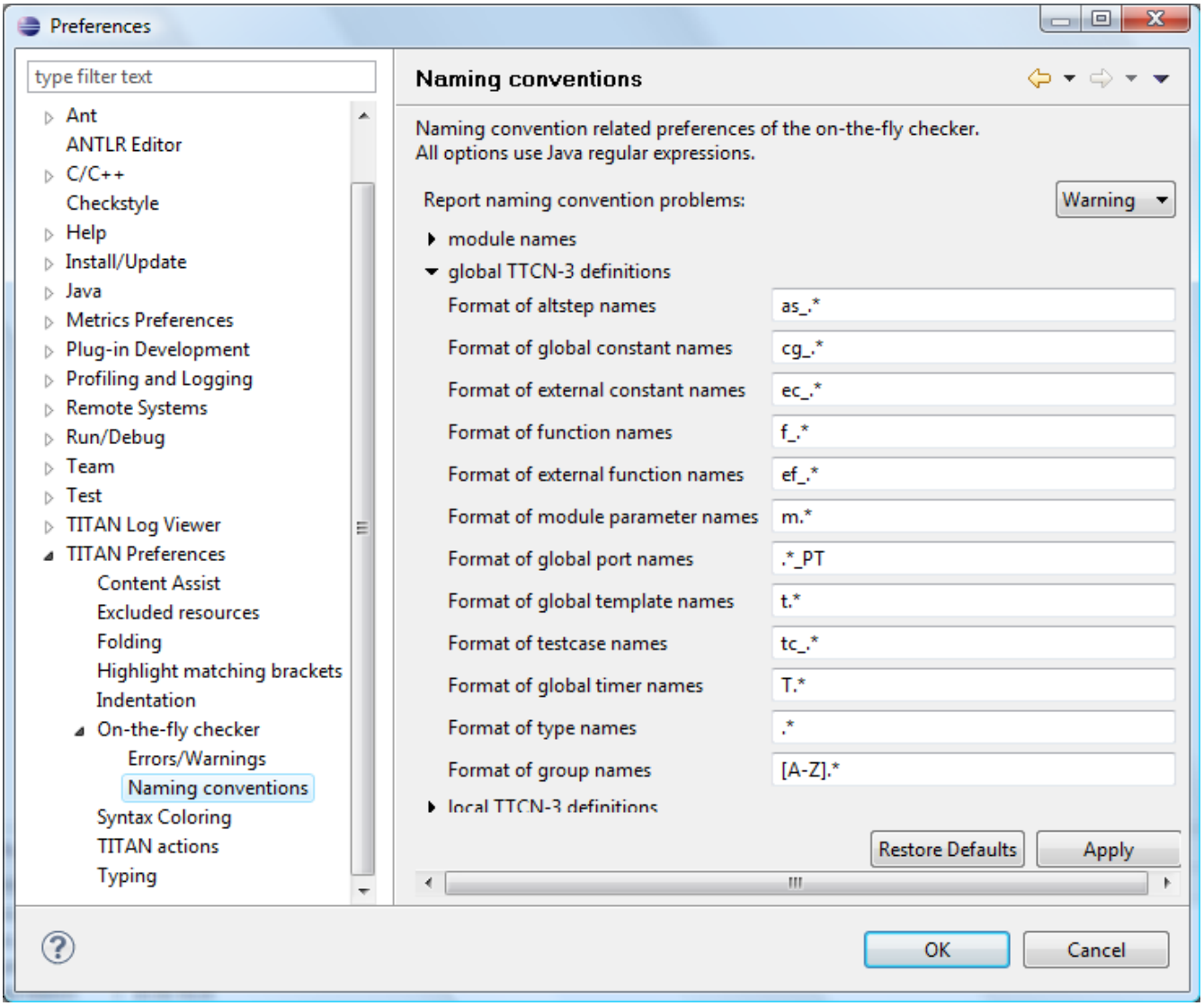


Figure 25. Workspace level naming convention settings

Usually it is preferred to follow a given naming convention in a project/environment as it decreases the maintenance cost of source code, by making it easier to understand for every developer working on it. These naming conventions can be configured on this page for the on-the-fly checker to use.

NOTE	These options can be overridden on project and folder level.
NOTE	It is suggested to switch off checking the naming convention because it significantly decreases the speed of the analysis. It should be switched on only at code cleaning.

The naming conventions are grouped into sections.

The last section, the "other naming rules", is not self-explanatory therefore it is explained below.

Section "Other naming rules":

- **Report if the name of the module is mentioned in the name of the definition**

  Definitions can be referenced in the `modulename.identifier` format, in order to avoid a name collision. Adding the module name to the definition is unnecessary, this only makes it longer. The on-the-fly checker, when it detects that a definition contains its module name, triggers one of the following behaviors: **Ignore, Warning, Error**. The default setting is: **Ignore**.

- **Report visibility settings mentioned in the name of definitions**

  Visibility attributes should not be mentioned in the names of the definitions. They should be explicitly set as visibility attributes of the definition. The on-the-fly checker, when it detects that a definition contains a visibility attribute (private, public, friend), triggers one of the following behaviors: **Ignore, Warning, Error**. The default setting is: **Ignore**.

3.13. Syntax Coloring Preferences

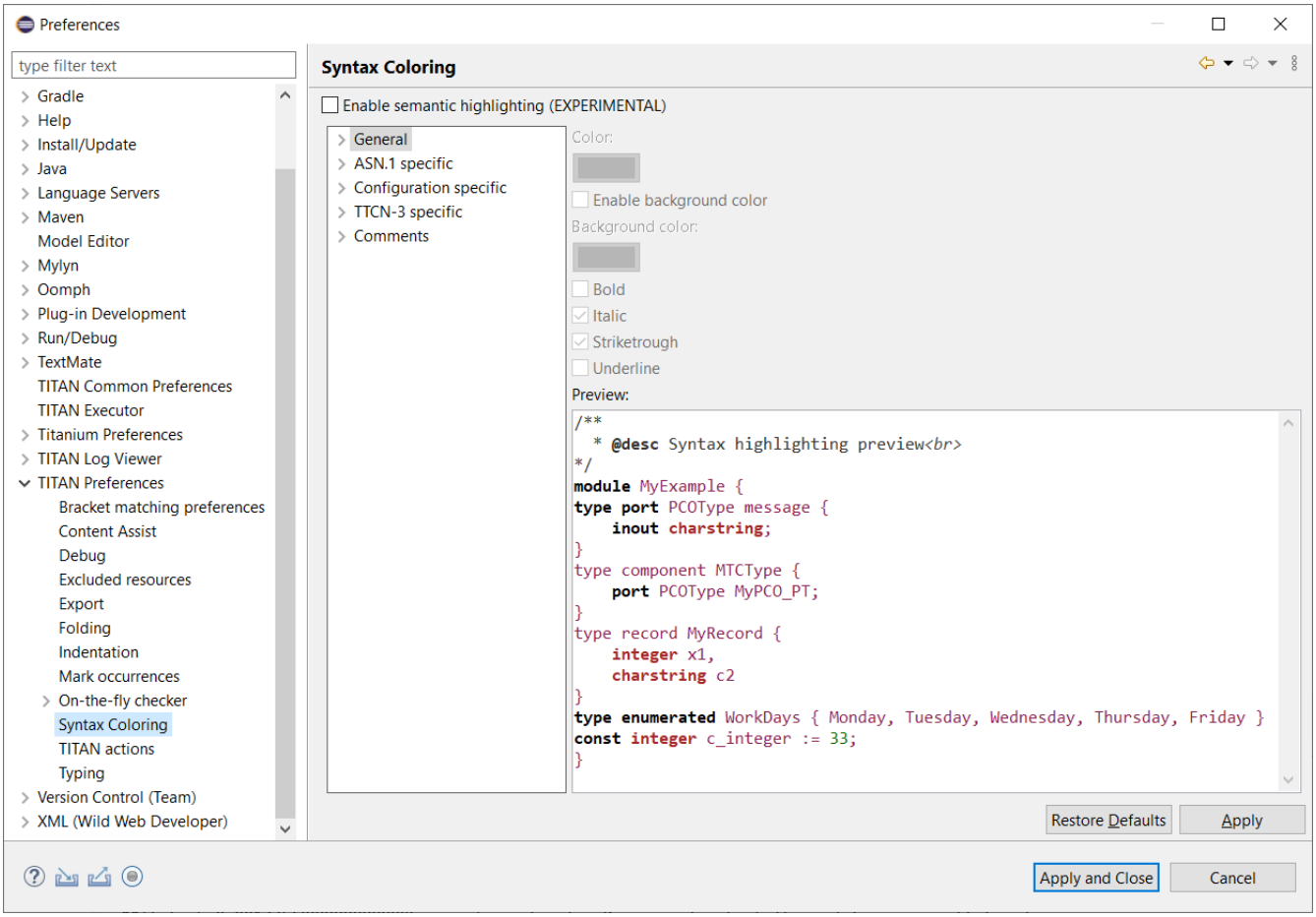


Figure 26. Syntax coloring preferences 1

On the Syntax coloring page, the syntax coloring preferences of the editors can be set. To change the color scheme and style of an element, first the element must be selected in the middle pane. To find the right element, click the > sign next to the appropriate group. The following groups have been defined:

- **General**

  The settings of these elements are applied in every editor. The style of text, comments and strings can be set here.

- **ASN.1 specific**

  The settings of these elements are applied in the ASN.1 editor. Styles of ASN.1 specific elements are determined here.

- **Configuration specific**

  The settings applied to these elements are valid for the Configuration editor. Styles of configuration specific elements are set here.

- **TTCN-3 specific**

  The settings of these elements are applied in the TTCN-3, TTCNPP and TTCNIN editors. Styles of TTCN-3 specific elements and preprocessor tokens are chosen here.

- **TTCN-3 semantic specific**

These settings are only available if the semantic highlighting is enabled on the top of the preference page. It contains such TTCN-3 specific elements that also carries semantic information, e.g. constants, type definitions etc. For certain elements, e.g. deprecated, the documentation comments have to parsed (to enable this feature see here). These settings allow the users to create more sophisticated coloring and style schemes.

The elements are only enabled if there is a node selected in the tree displayed on the middle pane. The elements are disabled if a branch is selected.

The actual attributes assigned to the selected elements are always shown (and can be modified) on the upper half of the right pane as follows:

- **Color**

  This option sets the color used for displaying characters.

- **Background color**

  This option sets the character background color.

- **Enable background color**

  This option enables background color. If this is disabled, the background color of the general text editor will be used instead of the selected one.

- **Bold**

  This option sets the style of the text to be bold.

- **Italic**

  This option sets the style of the text to be italic.

- **Strikethrough**

  This option strikethrough/strikeout the specific text.

- **Underline**

  This option underlines the specific text.

The lower half of the right pane shows an example text, where the user can follow the changes he made.

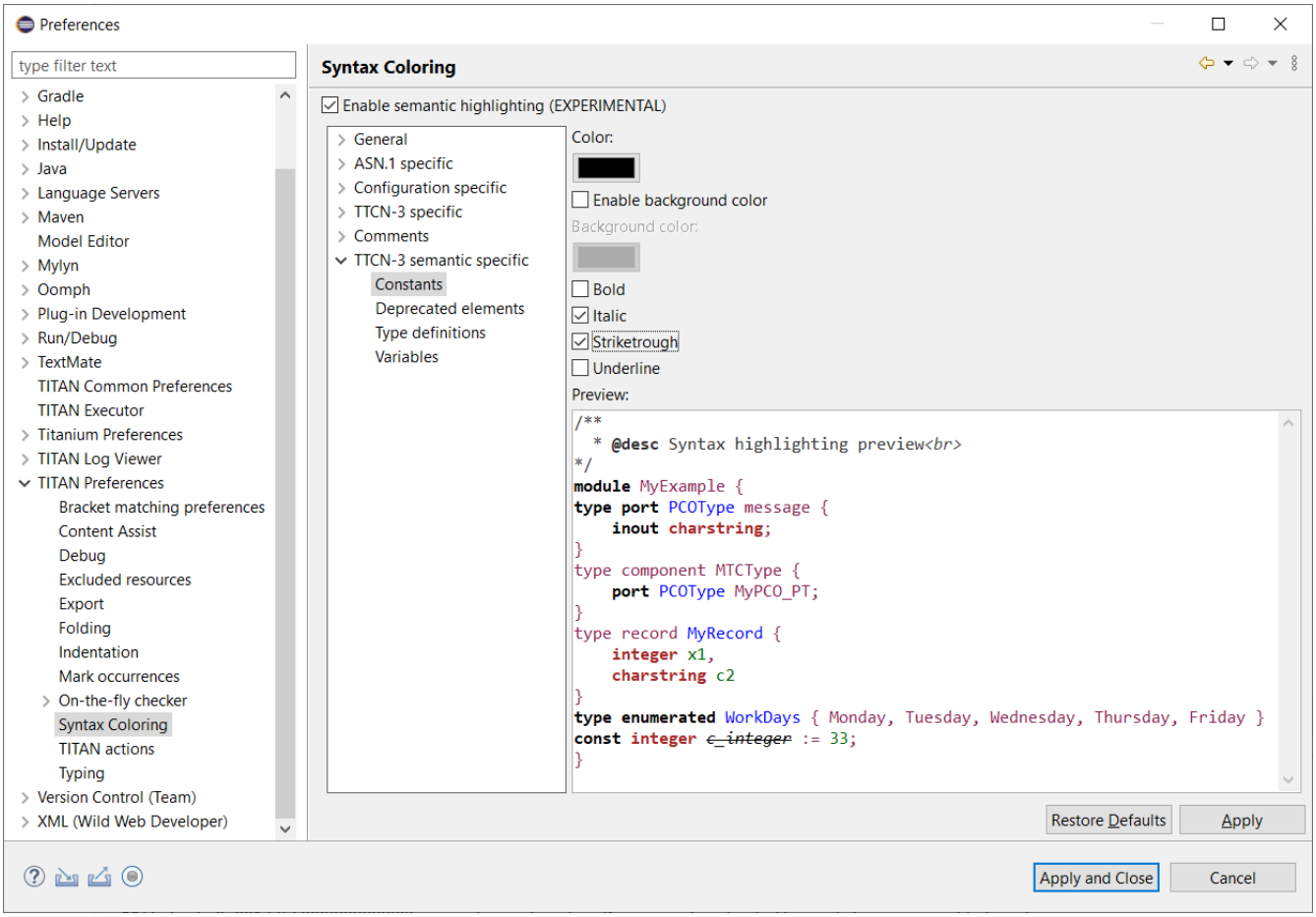


Figure 27. Syntax coloring preferences 2

To apply the new syntax color scheme, press the **Apply** or the **OK** button. Active editors are instantly adapting the changes in the color scheme.

The **Restore Defaults** button restores every setting to its default value.

TITAN Designer plug-in supports theming. The supported Eclipse built-in themes are light (default) and dark. To change the theme go to **Window** / **Preferences** and select **General**/**Appearance**. Check **Enable theming** and select the preferred theme from the drop-down list. The different themes have different syntax coloring preferences.

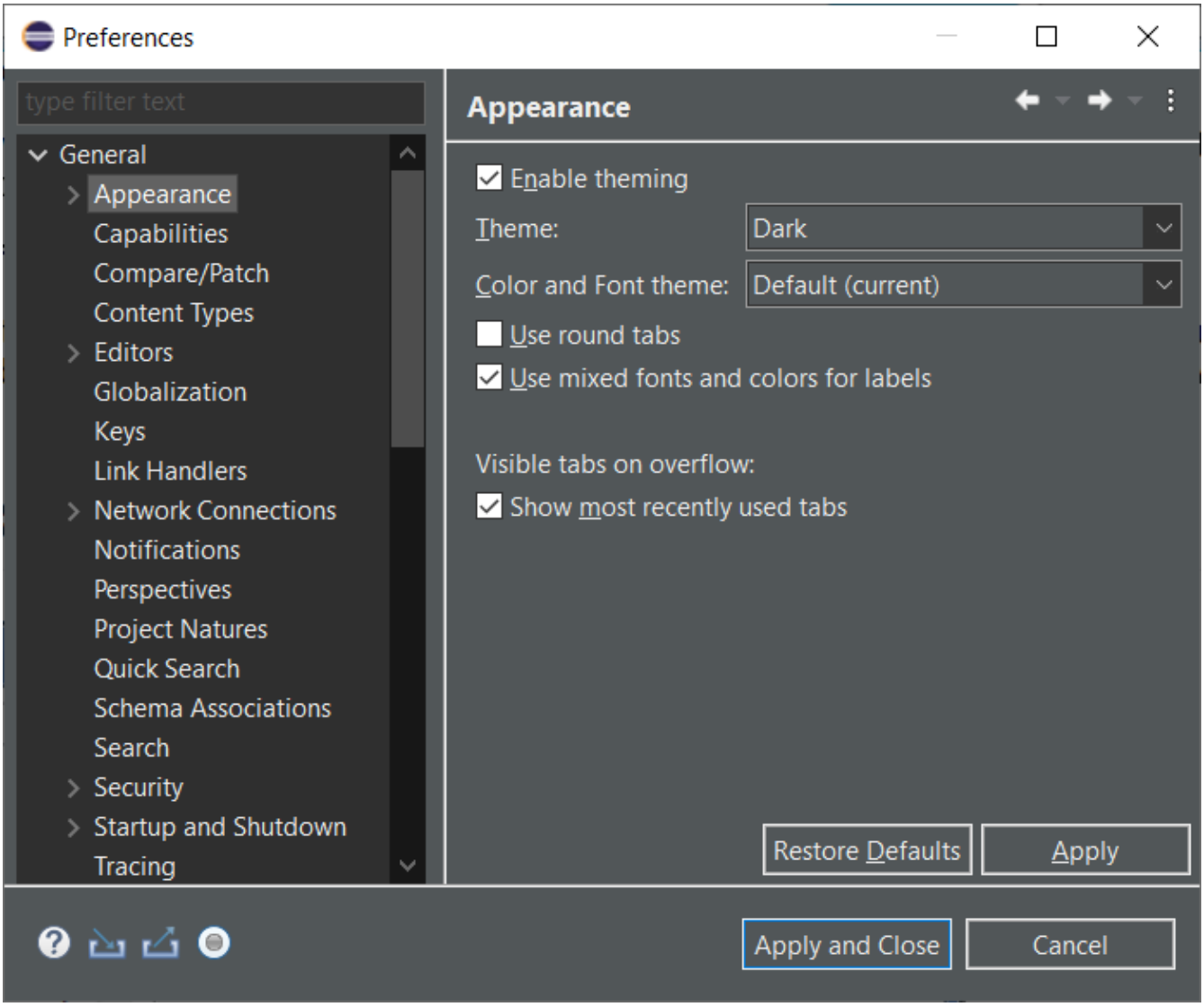


Figure 28. Setting up theming

3.14. TITAN Actions

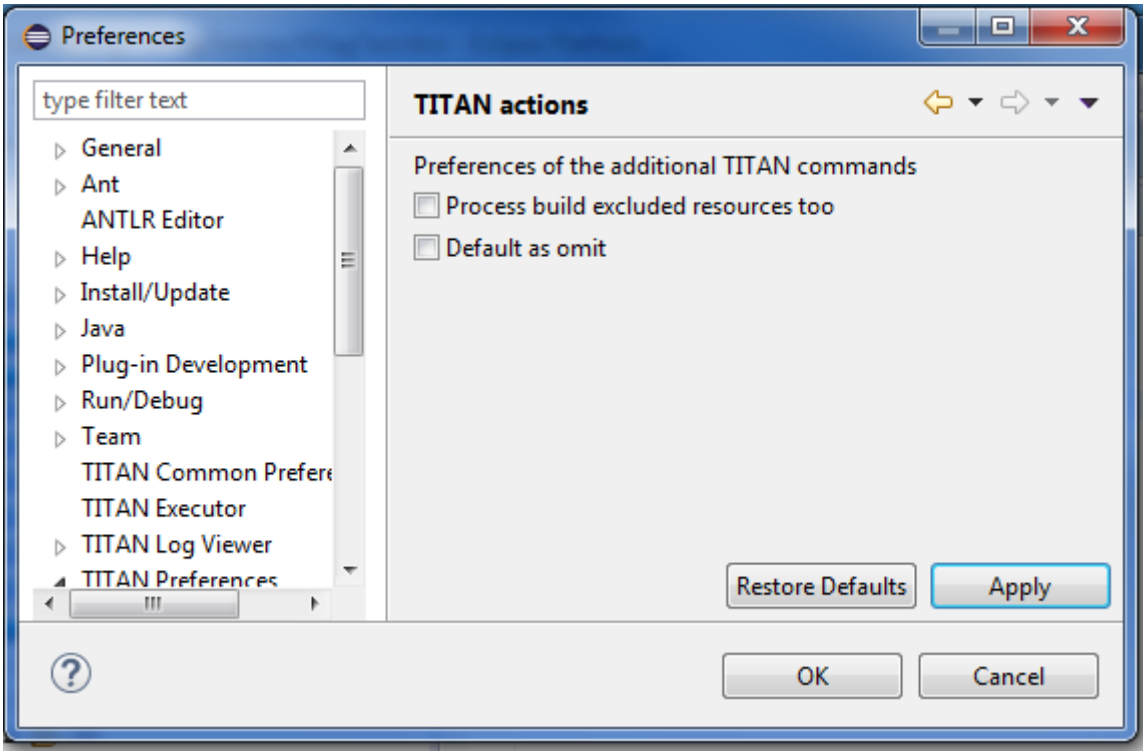


Figure 29. TITAN Actions preferences

On the TITAN Actions page the preferences of the external actions can be set. These options are available on this page (see the figure above):

- **Process build excluded resources too**

  if this option is set the external actions will also operate in resources that were excluded from the build process. The option is CHECKED by default.

- **Default as omit**

  if this options is set default values in ASN.1 structures will be handled as omitted ones.

NOTE	This is only useful in a few protocols. The option is UNCHECKED by default.

3.15. Typing Preferences

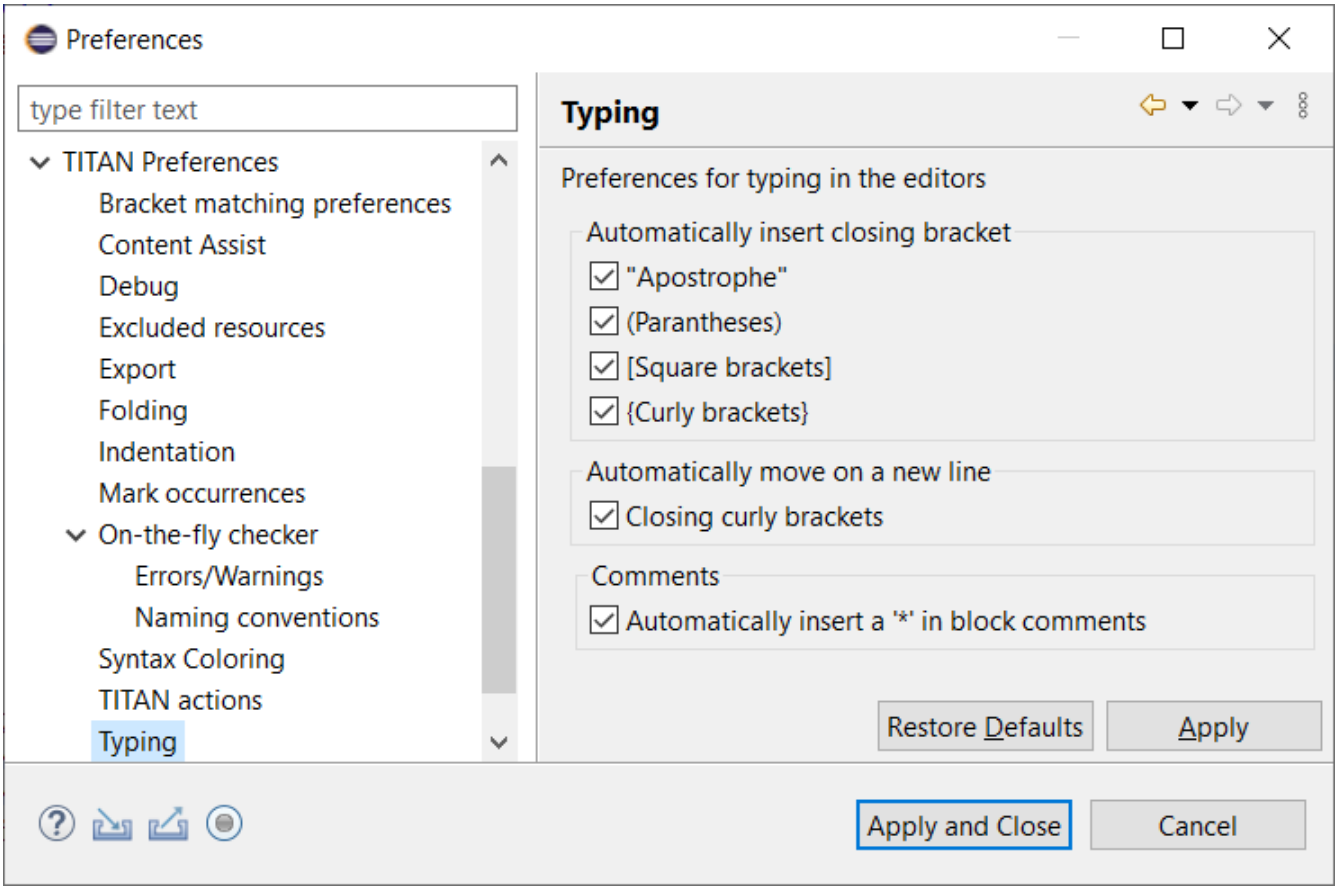


Figure 30. Typing preferences

The Typing page (above) is used to configure the automatic behavior during typing in supported editors. The variable parameters are divided into two groups.

The first group deals with automatic bracket insertion. For the last three items a checked box means that, as soon as the user types the opening bracket, the corresponding closing bracket will be automatically inserted. The cursor will be placed between the two brackets. This automatism can be invoked for three types of brackets: (parentheses), [square brackets] and {curly brackets}. For apostrophe it is somewhat different. In this case if there is some text selected when the user types an apostrophe, its pair will not be inserted right after, but rather on the other side of the selection, effectively enclosing the selected region. If there is an alphabetical character right before or after the cursor only one apostrophe is inserted. In other cases, the closing apostrophe is inserted automatically after the one typed.

The second group contains only one box for controlling new line insertion. A checked box has the following effect: if the user hits **Enter** between two curly brackets, the cursor will be moved to the next line and the closing bracket even further, to the second line. This way an empty line is formed with an opening bracket above and a closing bracket below it. The cursor will be placed on the empty line.

The third group contains only one box for controlling '*' insertion. A checked box has the following effect: if the user hits *Enter* when typing in multi-line/block comment context, the new line will automatically start with an indented '*' character. This behavior stops when the multi-line/block comment is closed (with '*/').

By default, all boxes are checked.

Chapter 4. Managing Projects

In the TITAN Designer plug-in, you work with projects. A project usually represents the complex procedure of developing a test suite and creating the executable from this test suite.

To manage these projects, it is advised to use the Project Explorer view provided by Eclipse. Other views, like the Navigator view, can also be used; however, beginners shall take special care as those views might provide completely different data. For example, by default the Project Explorer does not show the `.TITAN_properties` file, while the Navigator view does. The role of the `.TITAN_properties` will be explained later in section Saving and Loading Project Properties

For advanced users it is advised to take also a look on the other navigators, as they might be better in solving some minor problems.

Projects that are handled by the TITAN Designer plug-in will be referred to as TITAN projects although in the Eclipse terminology they should be called TITAN natured projects. More information on natures can be found in the Eclipse documentation.

From the release of TITAN version 6.6.0 the Designer supports 2 types of TITAN Projects: * TITAN C++ Project: is the original way of working, supporting a build system that translates TTCN-3 and ASN.1 modules using C as the intermediate language. * TITAN Java Project: if a way of working that supports a build system, that translates TTCN-3 and ASN.1 modules using Java as the intermediate language.

When not specified explicitly the expression "TITAN Project" might refer to both modes. As they only differ in a few settings and their way of building, most of the advanced editing features will work exactly the same way on both project types.

4.1. Creating a New TITAN C++ Project

Using the TITAN Designer, new TITAN projects can be created following these three steps:

1. Select **File** / **New** / **TITAN Project (C++)** from the main menu (see the next figure).(The corresponding TITAN shortcut must be enabled, see section Enabling TITAN Shortcuts)

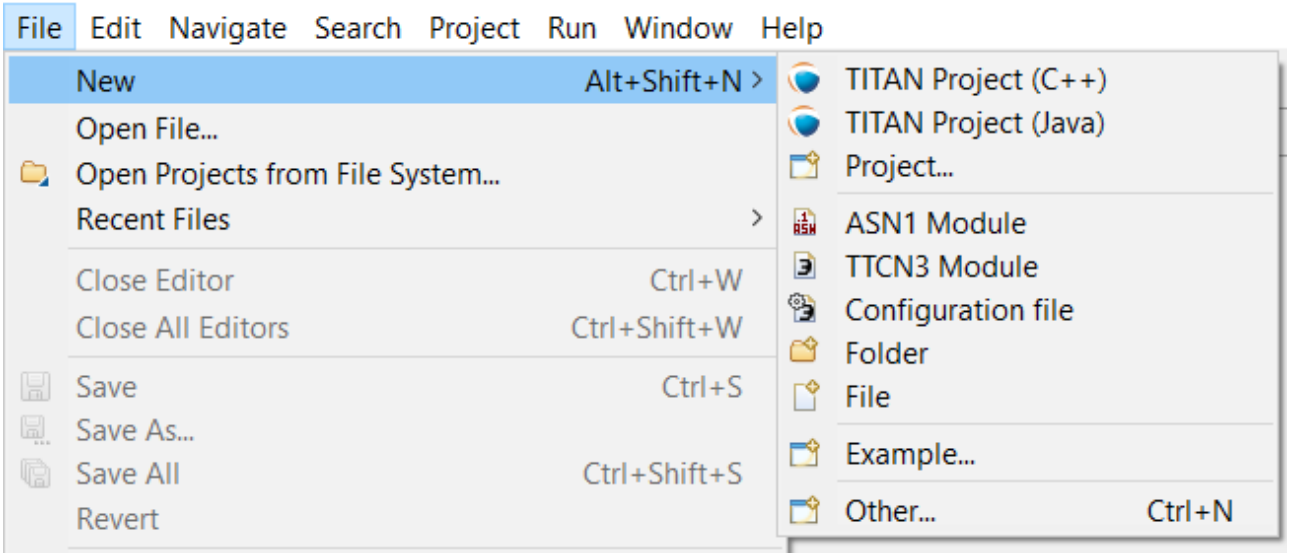


Figure 27. New resources menu

2. Enter project name and location (see the next figure). By default, the project will be created in the directory of the workbench. It is not recommended to select a path that contains special characters (like the spaces in "Documents and Settings").

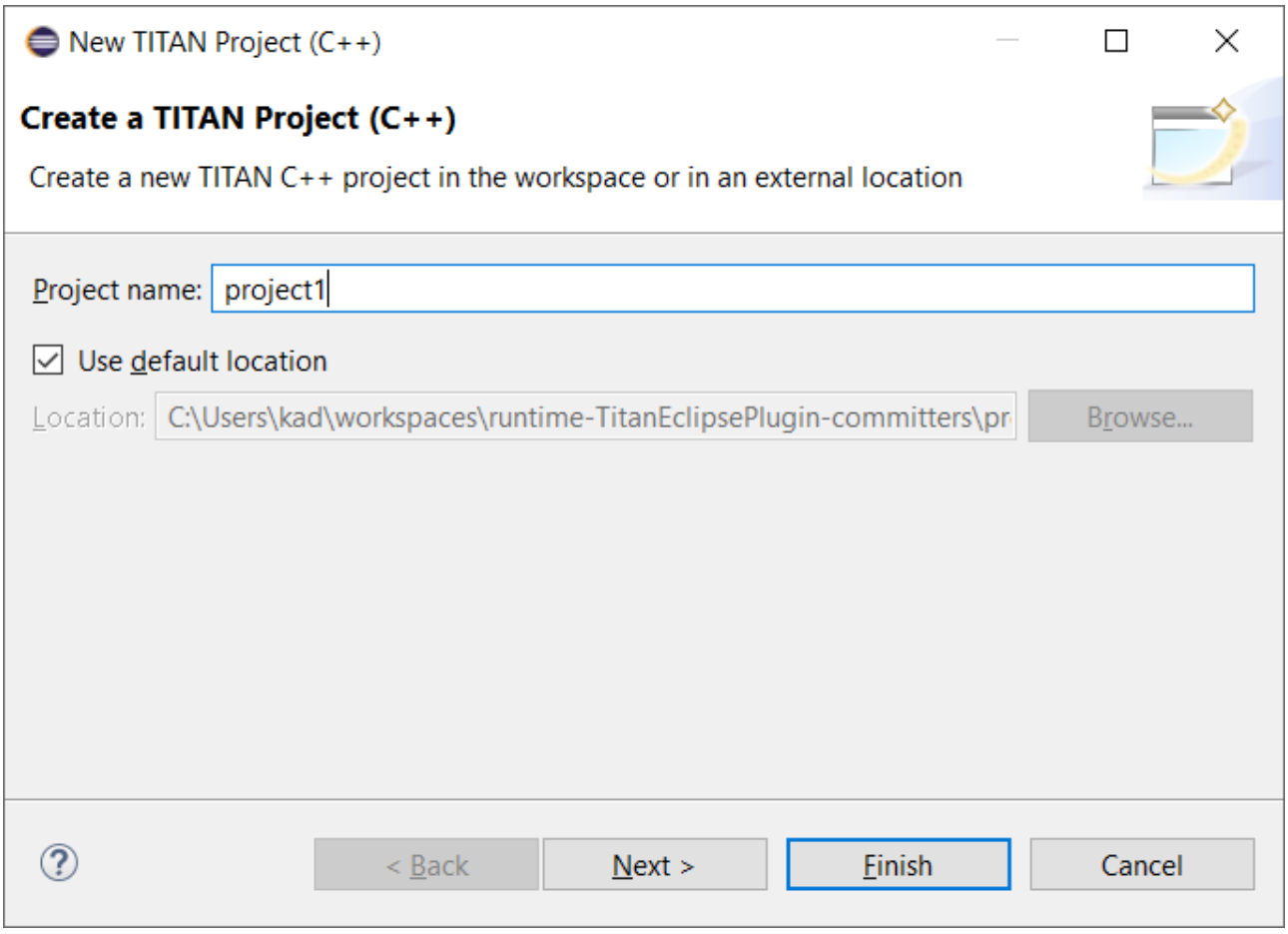


Figure 28. First page of the new TITAN Project (C++) wizard

3. At this point you can either select either **Finish** or **Next**.

   If you select **Finish,** the new TITAN project will be created immediately.

   If you select **Next,** you can customize some project properties (see the next figure): the name of

the folder containing the sources, and the name of the working directory (containing the generated binaries).In case the project to be created will need a long time to set up, before it can be used it is possible to set that the source folder should be generated as excluded from build.

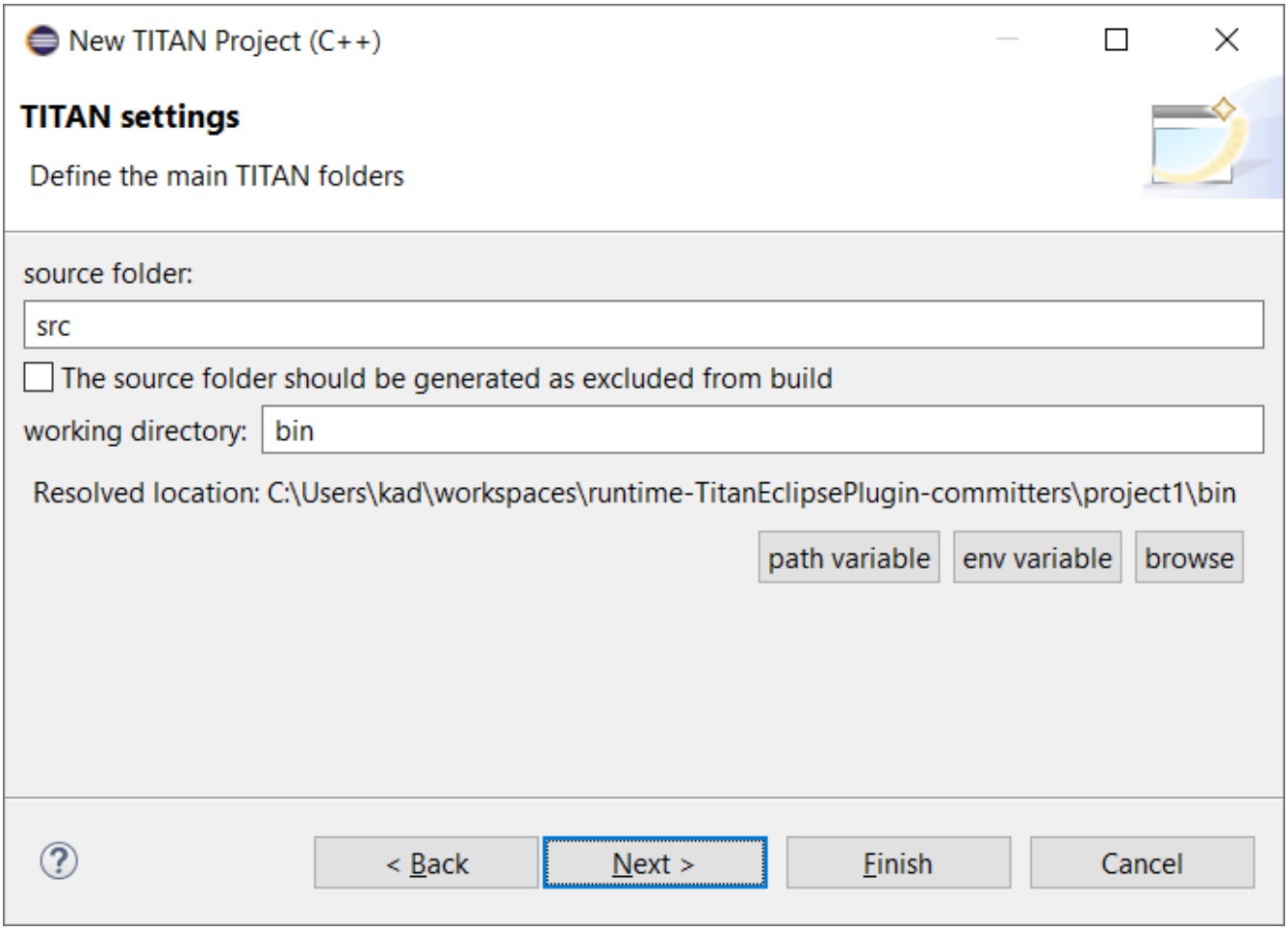


Figure 29. Second page of the new TITAN Project (C++) wizard

The final project is only created when you select **Finish**.

Now the new TITAN project (called `project1` on the figures) and the two directories are created and are listed in the Project Explorer. The TITAN logo is displayed to the left of the project name (provided that the TITAN decorator is enabled, see here). TITAN projects will generally be decorated like this.

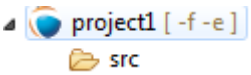


Figure 30. Example created project

The projects created with this wizard differ from other "General" projects in that the TITAN nature and the TITAN builder (responsible for building the executable) are automatically set on them.

Once the new project is created the property page of that project will be displayed, so that it can be configured immediately. For more information on project properties please refer to section Setting Project Properties

4.2. Creating a New TITAN Java Project

Using the TITAN Designer, new TITAN Java projects can be created following these three steps:

1. Select **File** / **New** / **TITAN Project (Java)** from the main menu (see the next figure).(The corresponding TITAN shortcut must be enabled, see section Enabling TITAN Shortcuts)
2. Enter project name and location (see the next figure). By default, the project will be created in the directory of the workbench. It is not recommended to select a path that contains special characters (like the spaces in "Documents and Settings").
3. At this point you can select either **Finish**.

   If you select **Finish,** the new TITAN Java project will be created immediately.

The final project is only created when you select **Finish**.

Now the new TITAN Java project (called `project1` on the figures) and the 4 directories are created and are listed in the Project Explorer. The TITAN logo is displayed to the left of the project name (provided that the TITAN decorator is enabled, see here). TITAN projects will generally be decorated like this.

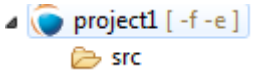


Figure 31. Example created project

The projects created with this wizard differ from other "General" projects in that the TITAN nature and the TITAN Java builder (responsible for building the executable) are automatically set on them.

Once the new project is created the property page of that project will be displayed, so that it can be configured immediately. For more information on project properties please refer to section Setting Project Properties

4.3. Adding Directories to the Project

Directories can be added to projects in the following way: **right click** the project where the directory should be added to and select **New** / **Folder** (see the next figure).

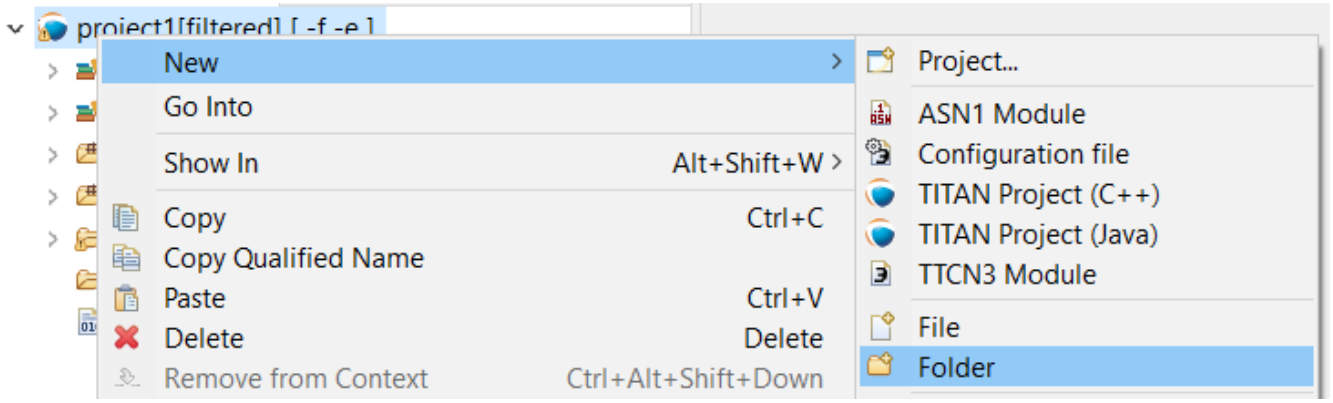


Figure 32. New menu

In the New Folder window there is a possibility to set:

- where the new folder will be placed;
- how the new folder will be called;
- whether the folder is a virtual folder ("Folder is not located in the file system (Virtual Folder)" see Eclipse general documentation)
- whether only a link to an existing folder will be established ("Link to alternate location (Linked Folder)") (This will appear in the Project Explorer just like a normal folder, but is actually a link to a folder).

NOTE	linked folders are handled entirely by Eclipse; no additional resource will be placed in the projects directory.

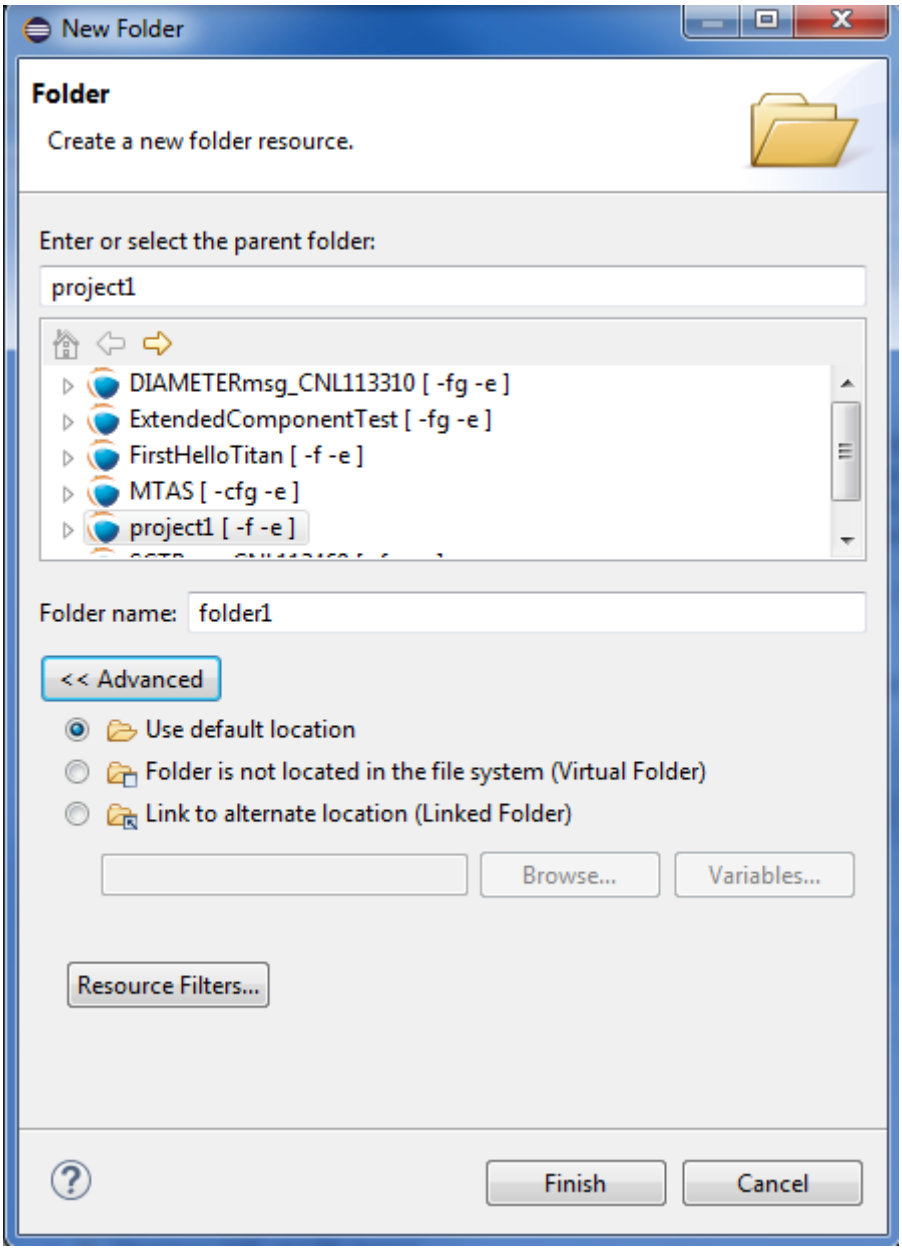


Figure 33. New folder window

Once the new folder created, you shall see something like shown on the Figure (without the filename file1.ttcn).

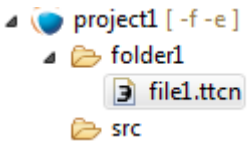


Figure 34. New file created

4.4. Adding Files to the Project

There are two ways to add files to a project. The first one, using wizards, is the recommended way to do it.

4.4.1. Using Wizards to Add Files to the Project

Wizards are available to create some of the TITAN modules [1] (TTCN-3, ASN.1 and Configuration files). This functionality is reached by selecting **File / New** (see figure New resources menu).

NOTE	In the Project Explorer view, the wizards "TTCN-3 Module", "ASN.1 Module" and "Configuration file" can be also reached by **right click**ing the content area and selecting **New / Other...**.

In the example below, the "TTCN-3 Module" wizard is shown. The wizard is launched by selecting **File / New / TTCN3 Module**.

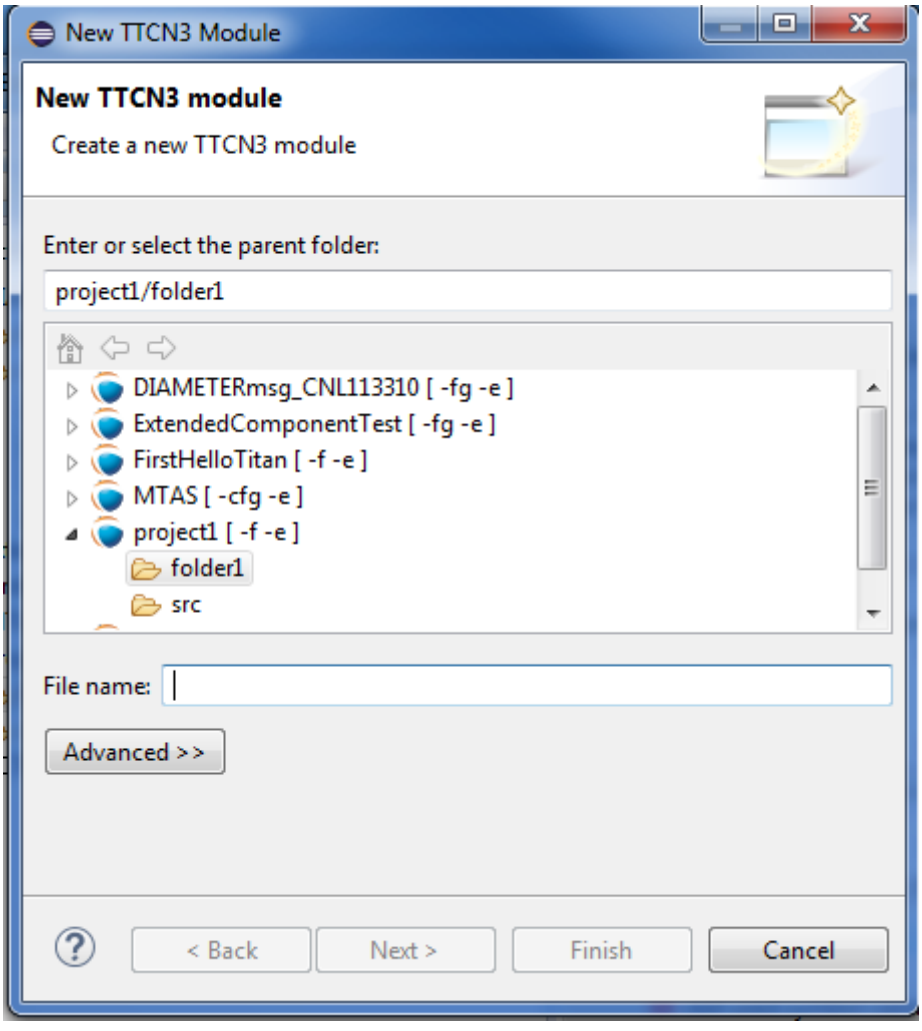


Figure 35. First page of the New TTCN3 Module wizard

On the First page of the New TTCN3 Module wizard the correctness of the new module name is verified. The file extension is checked against the type of module being created. If the extension is not set, it is automatically appended when the file is created (the defaults are: `ttcn`, `asn` and `cfg` for the respective wizards). The on-the-fly checker, if it has enough data collected, verifies that a module name is unique in the project (right now this only works for TTCN-3 modules).

On the second page of the wizard there are a checkbox and a combo box:

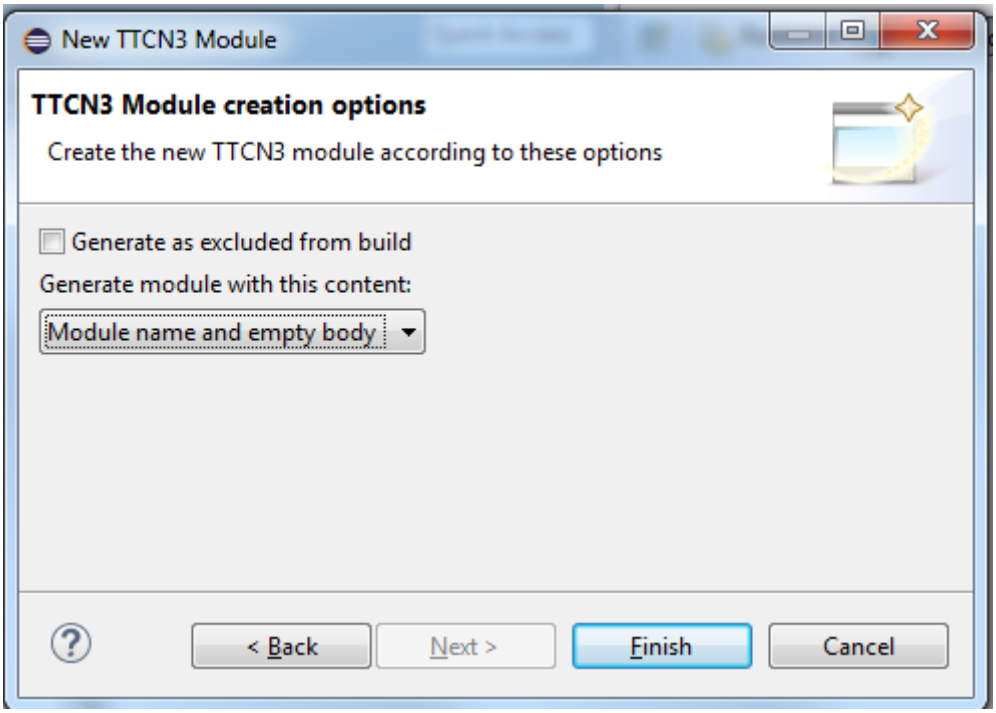


Figure 36. Second page of the New TTCN3 Module wizard

- **Generate as excluded from build**.

  If this checkbox is selected the file to be created is excluded from the build; that is, the build system will not try to build it instantly. It is advised to create new modules with this option turned on to avoid build errors until the code logic is complete.

- **Generate with module with this content**

  This Combo box contains three options: Empty module name, Module name and empty body and Module skeleton. As the names suggest, the generated file will contain empty module or module containing only module name and empty body or a module skeleton.

NOTE	Configuration files may also be created with a skeleton.

NOTE	The filename will be used as the module name in the inserted module.

4.4.2. Manually Adding Files to the Project

Manual file addition has moderate means to set file properties compared to the wizard (see here). On the other hand, some files can only be inserted into projects manually; namely the following way: **right click** on the project (or on a folder in the project) where the file should be included and

select **New / File** (see Figure New menu above).

On the New File window there is a possibility to set:

- where the new file should be placed;
- how the new file will be called;
- whether only a link to an existing file will be established (under selection menu **Advanced**>>)

  (This will appear in the Project Explorer just like a normal file, but is actually a link to a file).

NOTE	Linked files are fully handled by Eclipse; no additional resource will be placed in the projects directory.

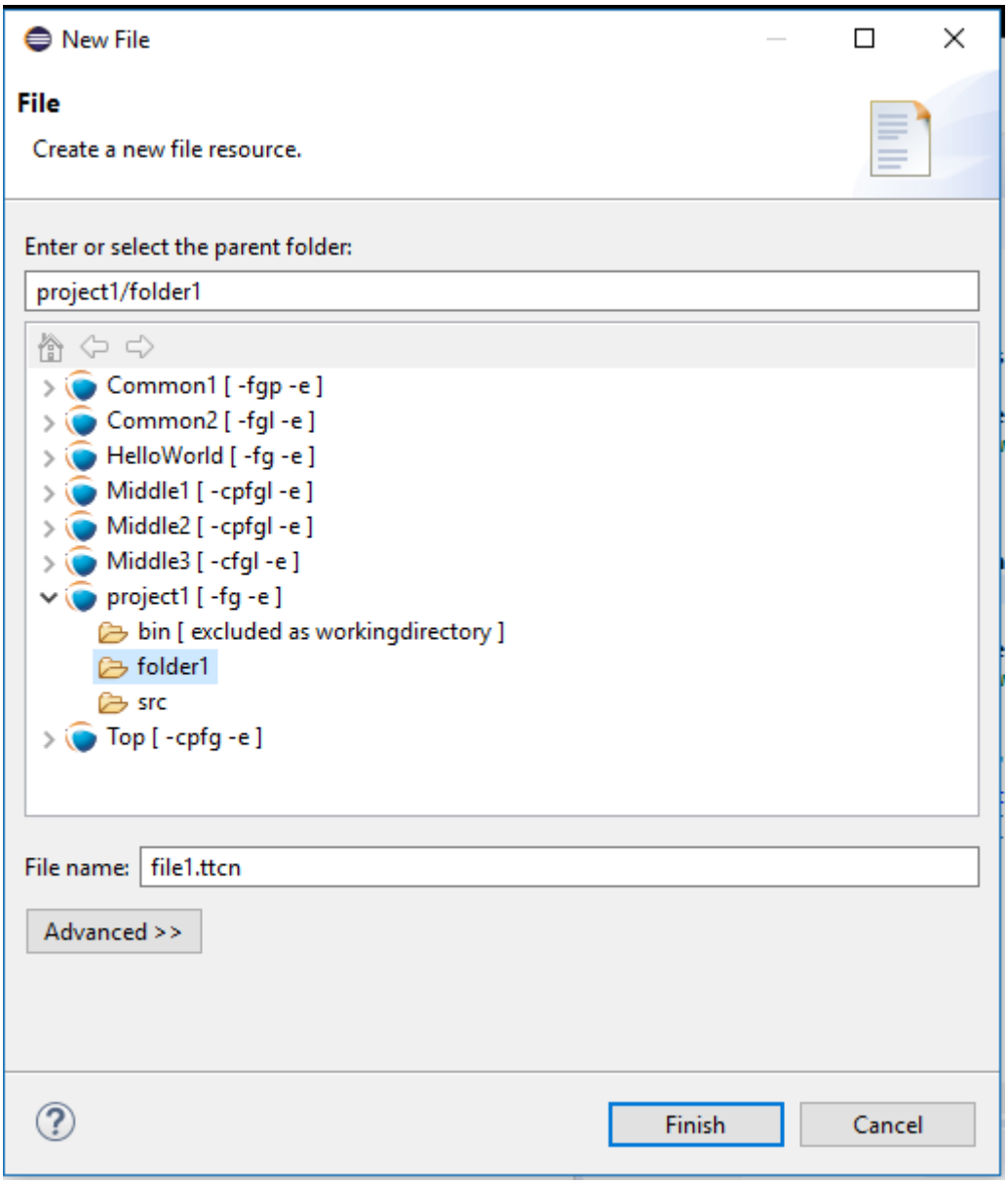


Figure 37. New file

Once the file created, you should see something like shown on New file created. You have created a project, added a folder and a file to it.

NOTE	Files handled by the TITAN Designer plug-in also have the TITAN logo to the left of their names, just like projects do. Decorators used by TITAN Designer are described here.

4.5. Setting Project Properties

Project properties for local and remote build are set in two separate windows.

4.5.1. Build Configurations

Our projects support to have several "build configurations" or "sets of build settings". This means that it is possible to create sets of build settings, which can be switched to in an easy and consistent way.

One excellent usage tip would be, to have "Development" and "Release" modes for projects. Debug could have settings tuned for very fast compilation, at the expense of generating slowly executing code: This way development could be speed up considerably while only loosing features not relevant at development time. Release mode could be fine-tuned for runtime performance, at the cost of increase in build times. This way once the development is over, and the product is ready to be tested/investigated/used, the build system could be set to use the most aggressive optimization methods available.

Changing the active build configuration is available on all project preference pages, in the upper part of the window, as seen on Figure Makefile creation attributes.

Using the drop-down control, one can select and switch to any already existing build configuration created for the actual project.

Pushing the **Manage Configurations** button a new window will pop-up.

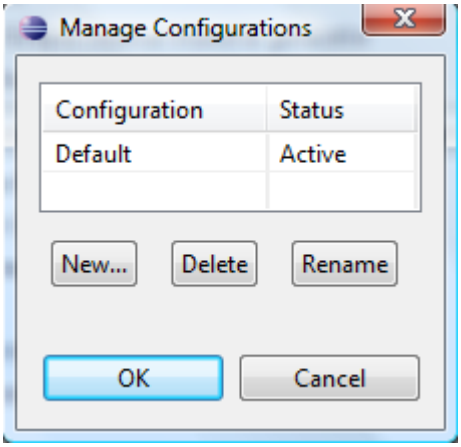


Figure 38. Manage configurations

On this window it is possible to create new configurations, delete existing ones, or simply rename one.

NOTE	Even though the settings of the Default configuration can be changed it cannot be deleted or renamed, the existence of this configuration is needed to be forward compatible with older versions of our tools. The build configuration name cannot contain whitespace character. The visible build configuration settings always refer to the active build configuration. To change a build configuration at first it shall be selected as active configuration, then some of the settings described below shall be modified then the settings shall be saved by pushing the button "Apply" or "OK".

4.5.2. Setting the Local Build Properties of a TITAN Project

To set the project properties for local build first **right click** the project and select **Properties** then select **TITAN Project Property**.

On the main window three options can be set:

- **Automatic Makefile management**

  configures the TITAN Designer to automatically manage the `Makefile`.

NOTE	disabling the automatic `Makefile` management makes it the users' responsibility to update the file when it is needed. In case it is unchecked, the buttons on the **Makefile creation attributes** tab and on the **Internal makefile creation attributes** tab will be disabled. Default: selected.

- **Generate the Makefile using Eclipse internal Makefile generator**

  figures the TITAN Designer to use its own `Makefile` generator instead of the one provided by TITAN.
  Default: selected.

- **Don't use symbolic links in the build process**

  figures the internal Makefile generator and the builder to drive the build process in a way that does not requires the creation of symbolic links.

NOTE	This option requires the internal Makefile generation option to be set. Default: selected.

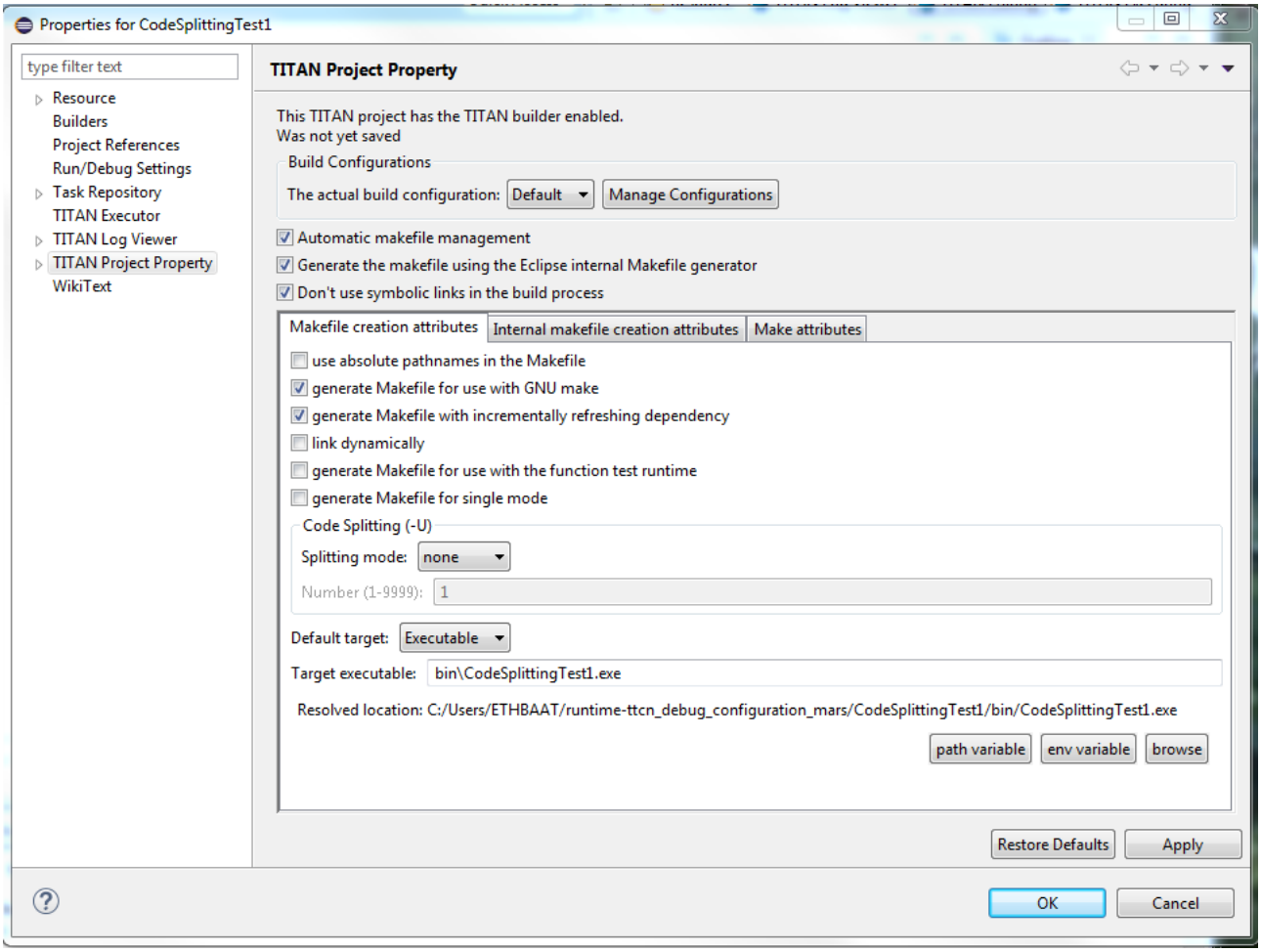


Figure 39. Makefile creation attributes

The Makefile Creation Attributes tab

Information from the **Makefile creation attributes** tab is transferred to the `Makefile` generator program. The options of the `Makefile` generator are described in the TITAN Programmer's Technical Reference [4].

The following Makefile creation attributes are set on this tab:

- **Use absolute pathnames in the Makefile**

  Specifies whether the generated `Makefile` should contain absolute or relative pathnames. Default: not selected.

- **Generate Makefile for GNU make**

  If checked, a GNU `Makefile` will be generated during the building process. The gnu make utility can handle complex `Makefile` that the Solaris make cannot. Default: selected.

- **Generate Makefile with incrementally refreshing dependency**

  If checked and GNU make style `Makefile` generation is also set, the generated `Makefile` will use GCC's dependency tracking instead of makedepend. For more information, please refer here. Default: selected.

- **Link dynamically**

  If checked, all files of the project will be compiled with `fPIC` and for each (static) object, a new shared object will be created. Then, these shared objects will be linked to the final executable instead of the (static) objects. For more information, pros and cons etc. consult the TITAN Programmer's Technical Reference [4]. Default: not selected.

- **Generate Makefile for use with the function test runtime**

  Titan has two runtime environments: one for function testing and one for load testing. The function test runtime provides more runtime checks and supports some specific features, like the negative testing feature, that is not available in the load test runtime. Therefore, for projects aiming functional testing, it is also advised to check the "generate `Makefile` for use with the function test runtime" checkbox. Default: not selected

NOTE	all dependent projects ("Project References" in Eclipse's term) shall use the same Titan runtime.

- **Generate Makefile for single mode**

  If checked, the executable will be built for single mode execution. Only one test component is allowed in single test mode. In parallel mode, on the other hand, several components can be used. Default: not selected.

- **Code splitting**

  Configures how the generated code should be organized: **none**, **type**, **number**. By default it is set to be: **none**.

- **Default target**

  Configures the default target of the generated `Makefile`:

  - **Executable:** Executable test suite
  - **Library:** Library archive

- **Name of the target executable**

  The path of the executable to be built including the name of the file. This setting will be written into the `Makefile` generated by the builder and will also be used for execution. If it is not set, the executable will be generated in the working directory having the name of the project.

The Internal Makefile Creation Attributes Tab

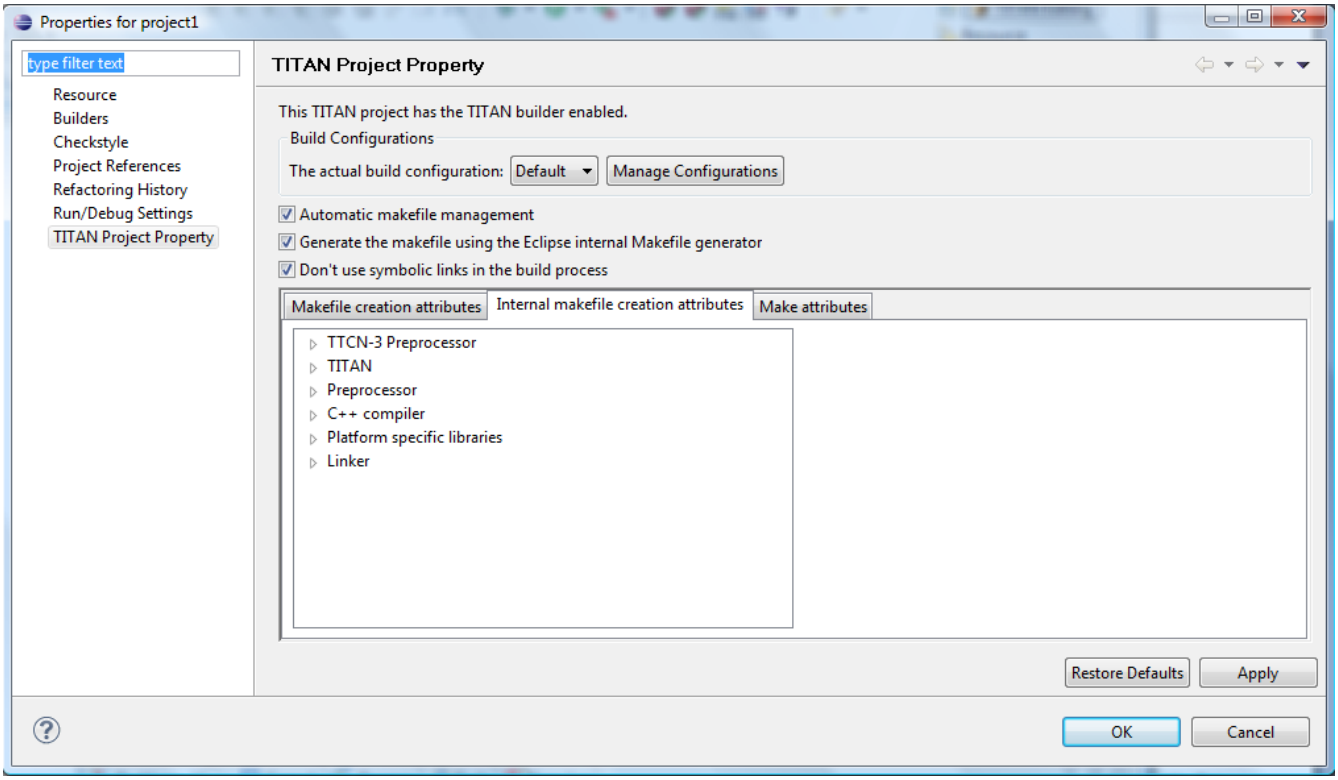


Figure 40. Internal makefile creation attributes

On the Internal makefile creation attributes tab the options to be generated into the `Makefile` can be set. To change the value of an element it must be selected. Depending on the element selected on the left side, the right hand side of the tab will contain different options.

1. TTCN-3 Preprocessor

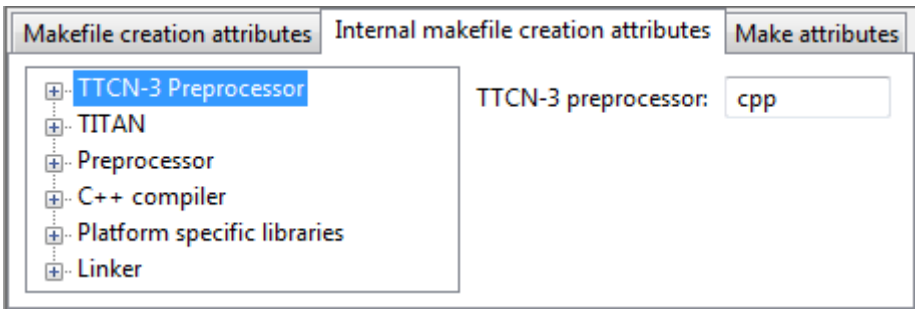


Figure 41. TTCN-3 preprocessor

On the TTCN-3 Preprocessor page it is possible to specify the preprocessor tool used to pre-process the `.ttcnpp` and `.ttcnin`.

This will be applied to the **CPP** macro. By default it is set to be: **cpp**

The pre-processing of `.ttcnpp` and `.ttcnin` files is the very first step of the build process, as the compiler is not able to analyze these file formats.

2. TTCN-3 Preprocessor Symbols

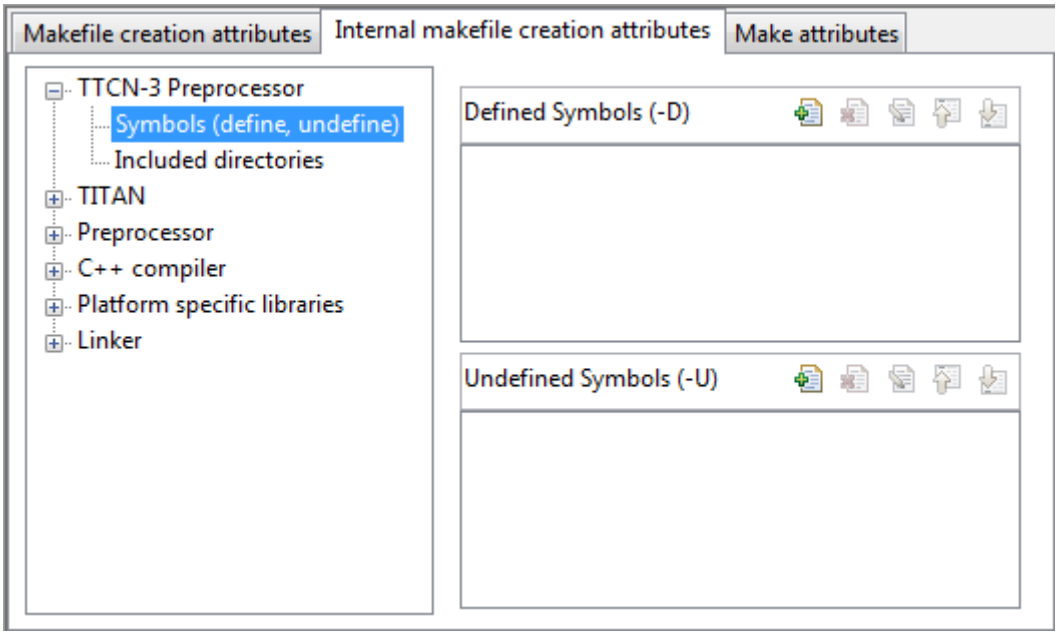


Figure 42. TTCN-3 Preprocessor symbols

On the symbols page it is possible to specify the list of symbols that should be defined and the list of symbols that should be undefined when the TTCN-3 pre-processor tool is executed.

These lists of options are applied to the **CPPFLAGS_TTCN3** macro (only present if pre-processable files are used in the project). By default both lists are empty.

3. TTCN-3 Preprocessor Included Directories

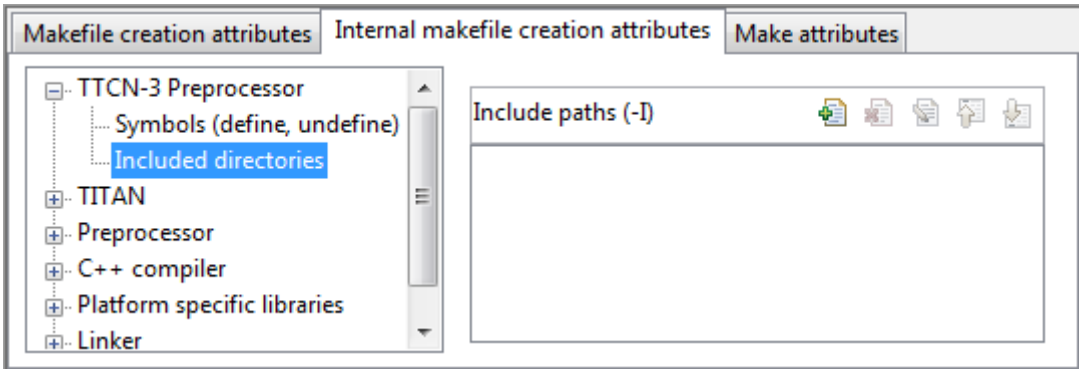


Figure 43. TTCN-3 Preprocessor include directories

On the included directories page, it is possible to specify the list of directories where the TTCN-3 pre-processor can look for included files.

The list of options is applied to the **CPPFLAGS_TTCN3** macro (only present if pre-processable files are used in the project). By default the list is empty.

4. TITAN Flags

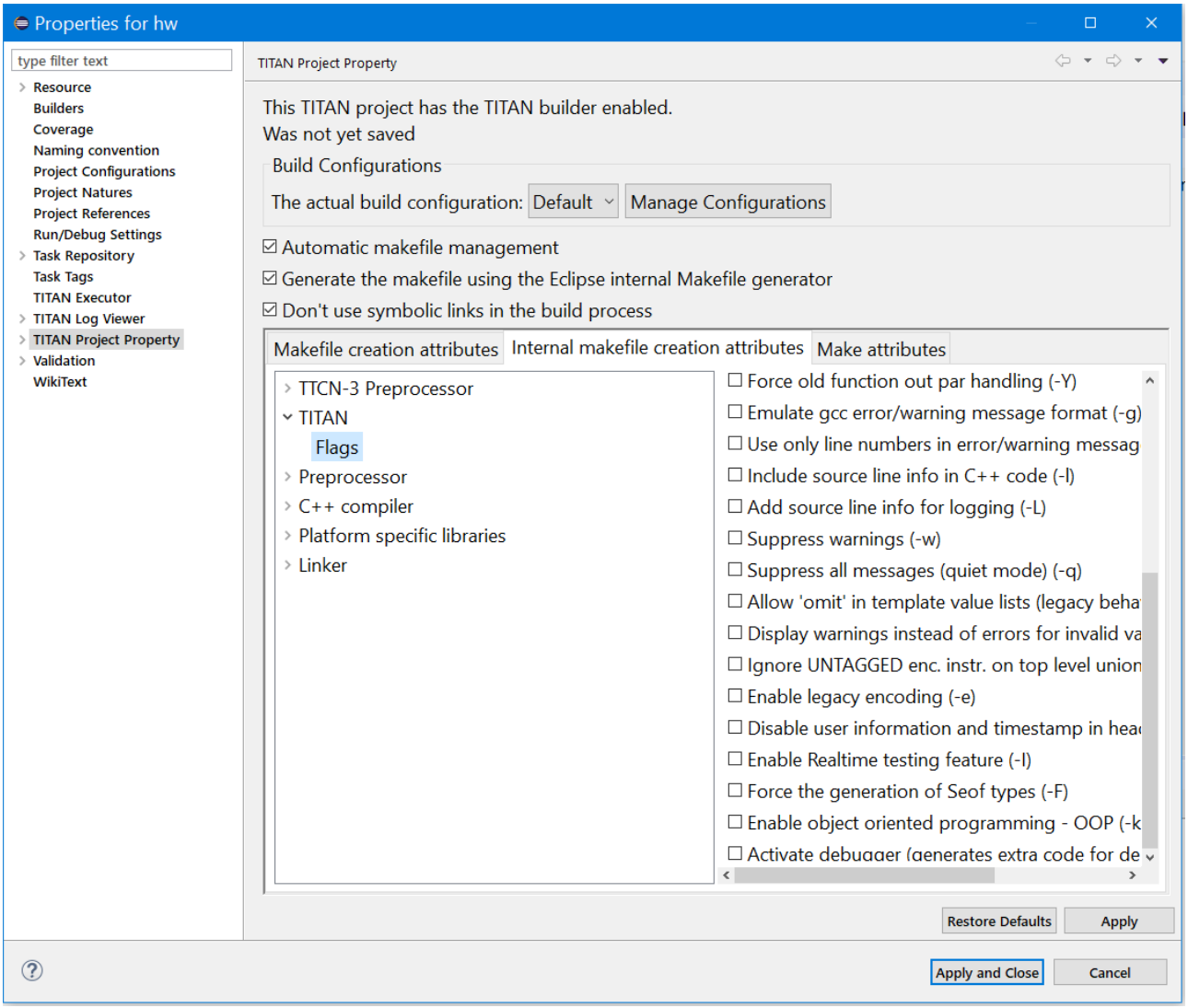


Figure 44. TITAN Flags

On the TITAN flags page, it is possible to specify the flags TITAN should be called with when compiling the TTCN-3 and ASN.1 files.

The options will be applied to the **COMPILER_FLAGS** macro. By default only the **Include source line info in C++ code** and **add source line info for logging** options are set.

NOTE	The flag responsible for function or load test runtime generation is not set here, but on the Makefile creation attributes (as that flag is handled by the Eclipse external `makefile` generator too).

NOTE	The flag `Enable object oriented programming - OOP (-k)` only controls the makefile generator and the compiler. Syntactic and semantic analyser do not support the OOP features yet.

For more information on the meanings of these options please refer to section 5.1 of the TITAN Programmer's Technical Reference guide [4].

5. Preprocessor

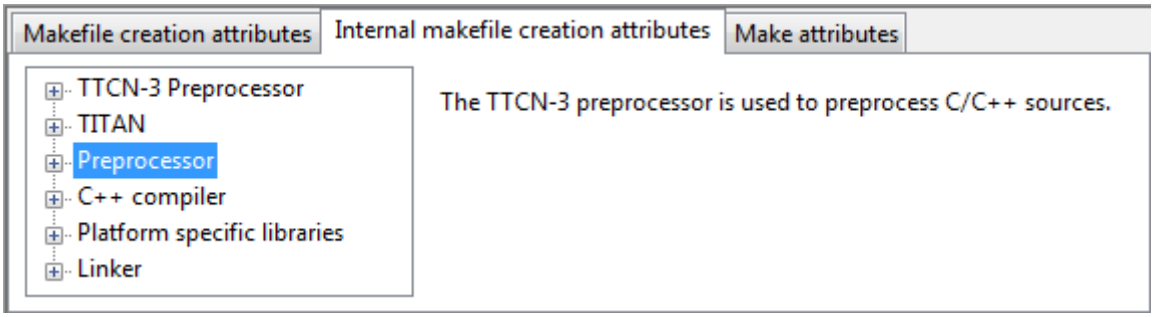


Figure 45. Preprocessor

The Preprocessor page only functions as reminder to the fact, that the generated `Makefile` uses the same tool for pre-processing the `.ttcnpp`, `.ttcnin` and C/C++ files.

6. Preprocessor Symbols

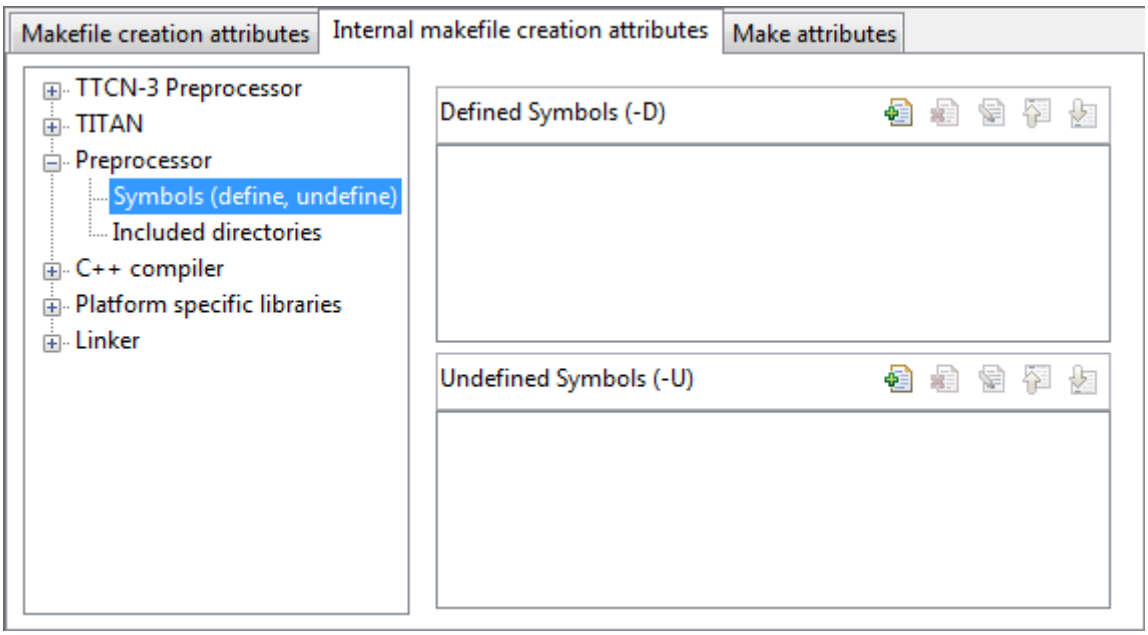


Figure 46. Preprocessor symbols

On the preprocessor symbols page, it is possible to specify the list of symbols that should be defined and the list of symbols that should be undefined when the C/C++ pre-processor tool is executed.

These lists of options are applied to the **CPPFLAGS** macro.By default both lists are empty.

NOTE	There are a few symbols that are not displayed here, but are generated into the `Makefile`. These symbols are required for proper operation.

7. Preprocessor Included Directories

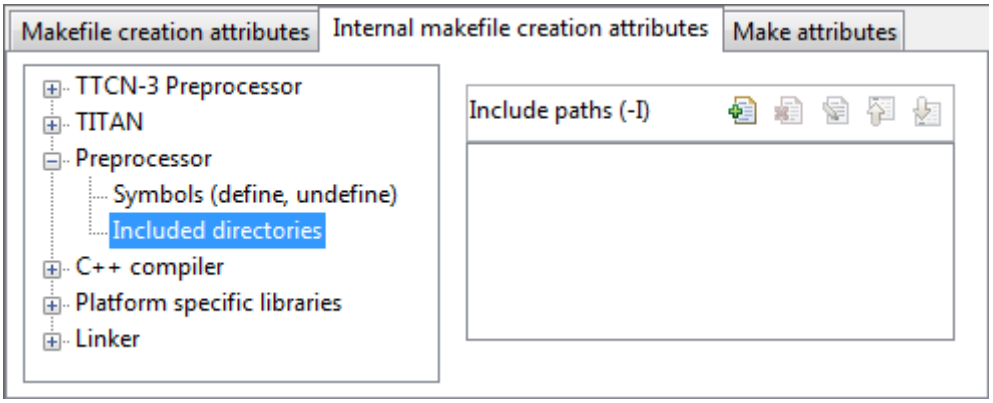


Figure 47. Preprocessor include directories

On the included directories page, it is possible to specify the list of directories where the C/C++ pre-processor can look for included files.

The list of options is applied to the **CPPFLAGS** macro. By default the list is empty.

NOTE	Some directories (like the include directory of TITAN) are not displayed here, but are generated into the `Makefile`. They are required for proper operation.

8. C/C++ Compiler

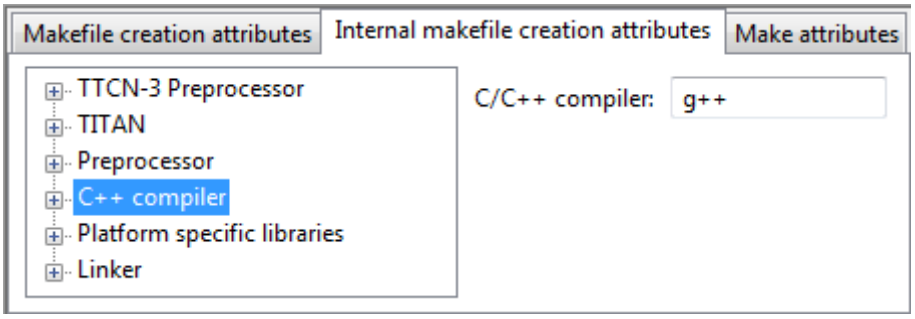


Figure 48. C/C++ compiler

A C/C\++ compiler tool used to process the generated and the user provided C/C++ files can be specified on the C/C++ compiler page.

This will be applied to the **CXX** macro. By default it is set to be: **g++**

9. C/C++ Compiler Optimization

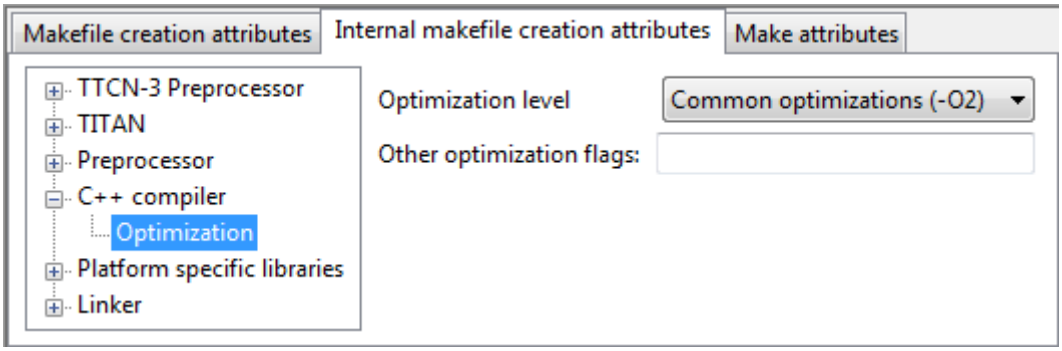


Figure 49. C/C++ compiler optimization

The C/C++ compiler optimization page allows the specification of optimization options for C/C++

compiler.

The optimization level option can be: none, minor optimizations, common optimizations, optimize for speed, optimize for size. By default it is set to: common optimizations.

The other optimization flags option allows the specification of any user defined optimization flag that is supported by the C/C++ compiler.

Both options will be applied the **CXXFLAGS** macro.

NOTE	The **–Wall** option is not displayed here, but is generated into the `Makefile`. It is required for proper operation.

For more information on the optimization flags please refer to the documentation of your C/C++ compiler. In case of the default C/C++ compiler g\\ is the manual pages of g\\ (invoked with the **man g** command line command).

10. Platform Specific Libraries

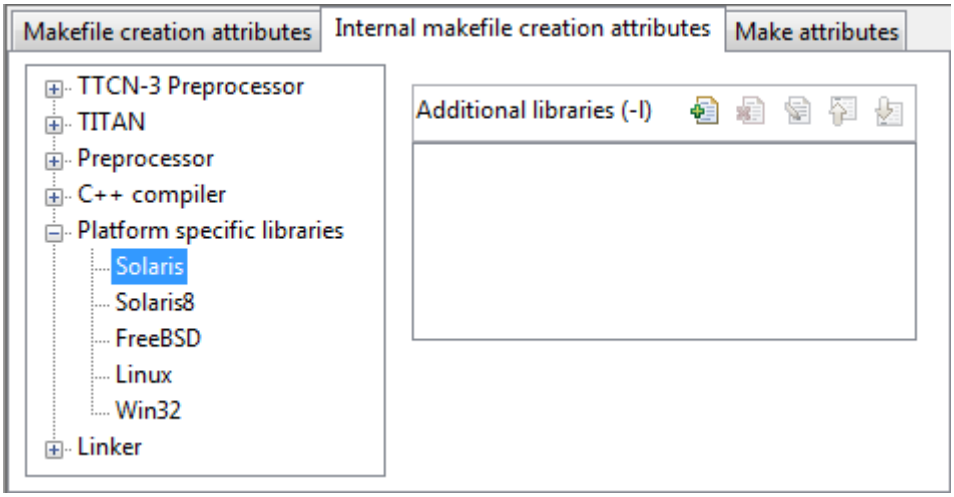


Figure 50. Platform specific libraries

On the platform specific libraries pages it is possible to specify the list of platform specific libraries that are needed to build the final executable for each supported platform.

The list of platform specific libraries is applied to the **SOLARIS_LIB**, **SOLARIS8_LIBS**, **LINUX_LIBS**, **FREEBSD_LIBS** and **WIN32_LIBS** macros respectively. By default all lists are empty.

NOTE	Some libraries are not displayed here, but are generated into the `Makefile`. These are required for proper operation on the above platforms.

11. Linker

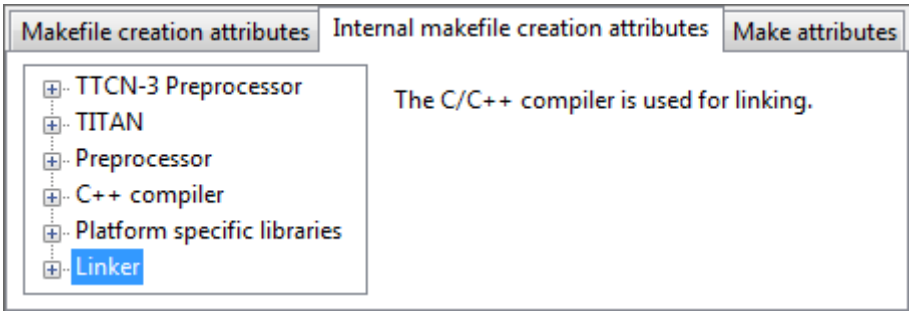


Figure 51. Linker

The Linker page only functions as reminder to the fact, that the generated `Makefile` uses the same tool for compiling C/C++ sources and linking the generated object files.

12. Linker Libraries

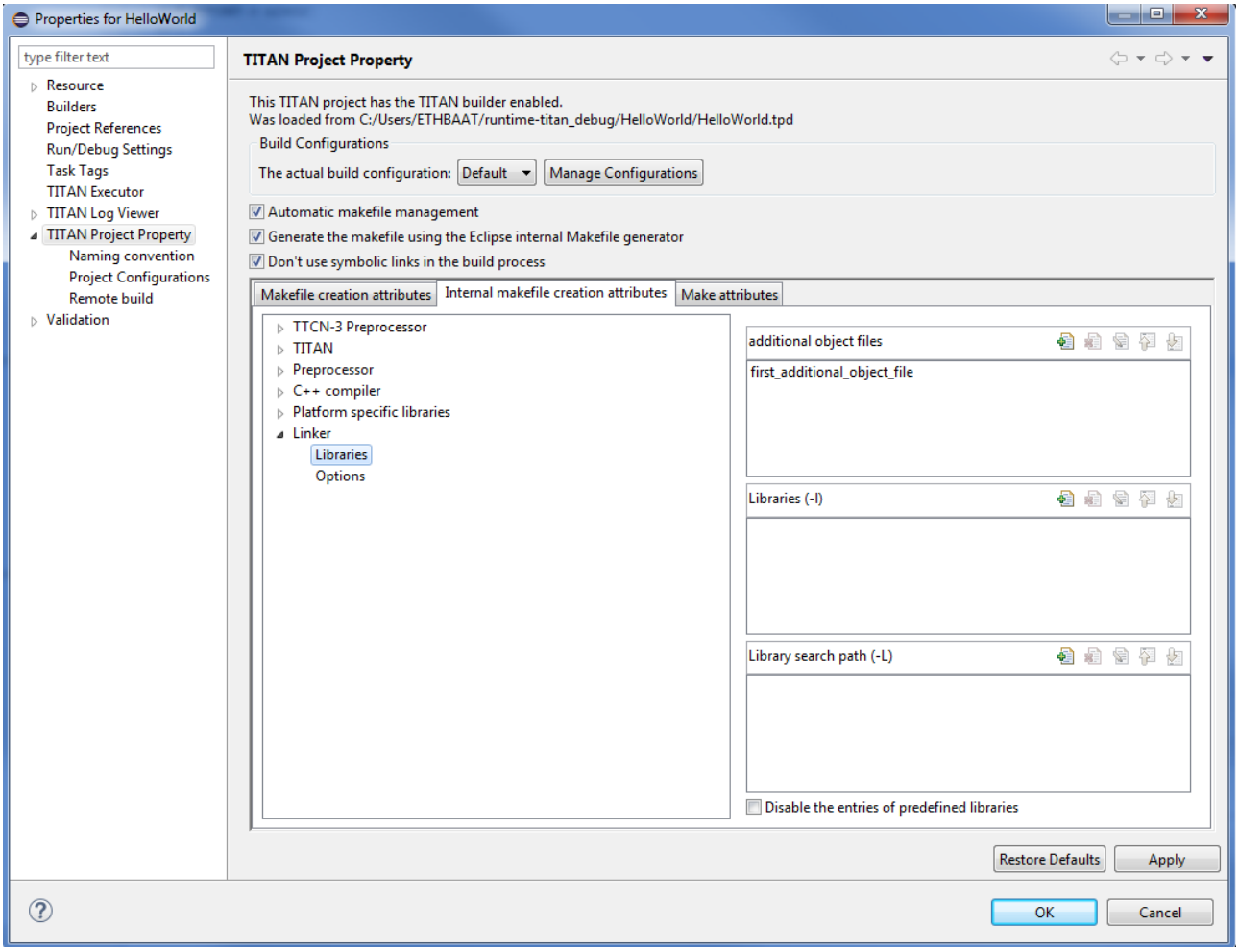


Figure 52. Linker libraries

On the linker libraries page it is possible to specify

- additional object files,
- the list of platform independent libraries (-l switch) and
- library search path (-L switch)

  that are needed by the linker to produce a valid executable.

These lists of options are generated directly into the command responsible for creating the final executable. By default the lists are empty.

NOTE	In list of the library search paths (-L), environment variables can be used. If the form `[MYVAR]` or `${MYVAR}` is used, the value of `[MYVAR]` or `${MYVAR}` will be resolved, if it is possible, while generating `Makefile`. Any other form will be regarded as a path relative to the project folder and will be prefixed with the project path.

In order for the generated `Makefile` to work and the project to compile properly there are some libraries and search locations not displayed here, but generated into the `Makefile`.

If the **Disable the entries of the predefined libraries** option is selected only the search paths related to **TTCN3_DIR** will be generated, all other libraries and search paths are left out of the generated `Makefile`. For example, in the generated Makefile, lines

```
OPENSSL_DIR = $(TTCN3_DIR)
XMLDIR = $(TTCN3_DIR)
```

will be commented out and their usage will be omitted.

By default, this option is not selected.

13. Linker Options

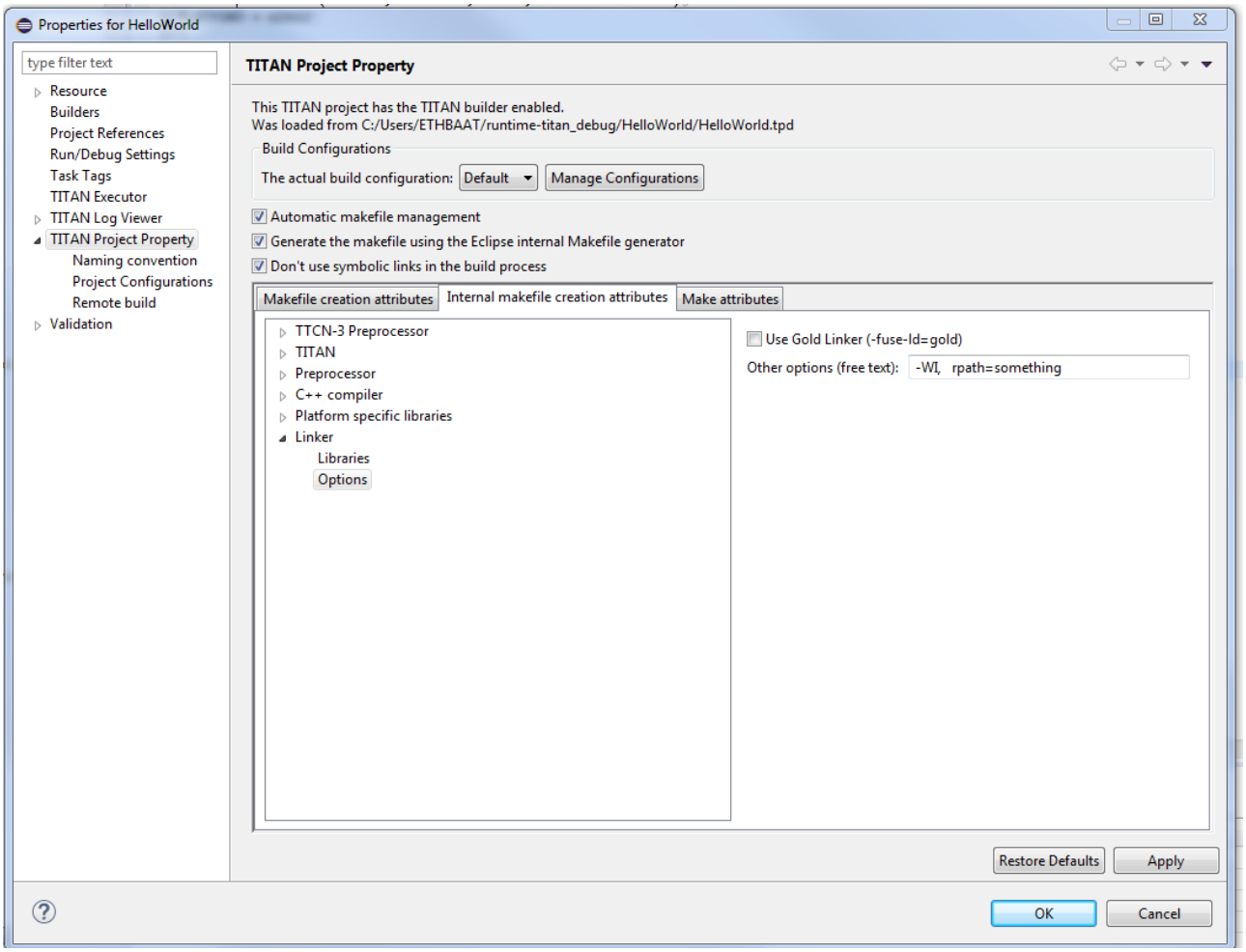


Figure 53. Linker Options

On the page "Linker Options" you can select different linker options. These will be added to the value of LDFLAGS in the Makefile.

The first option is to use the GNU "gold" linker instead of the regular one. If it is selected the text “-fuse-ld=gold” will be added to the value of LDFLAGS.

The second option is a free text. It also will be added to the value of LDFLAGS without any checking. Use it carefully!

The Make Attributes Tab

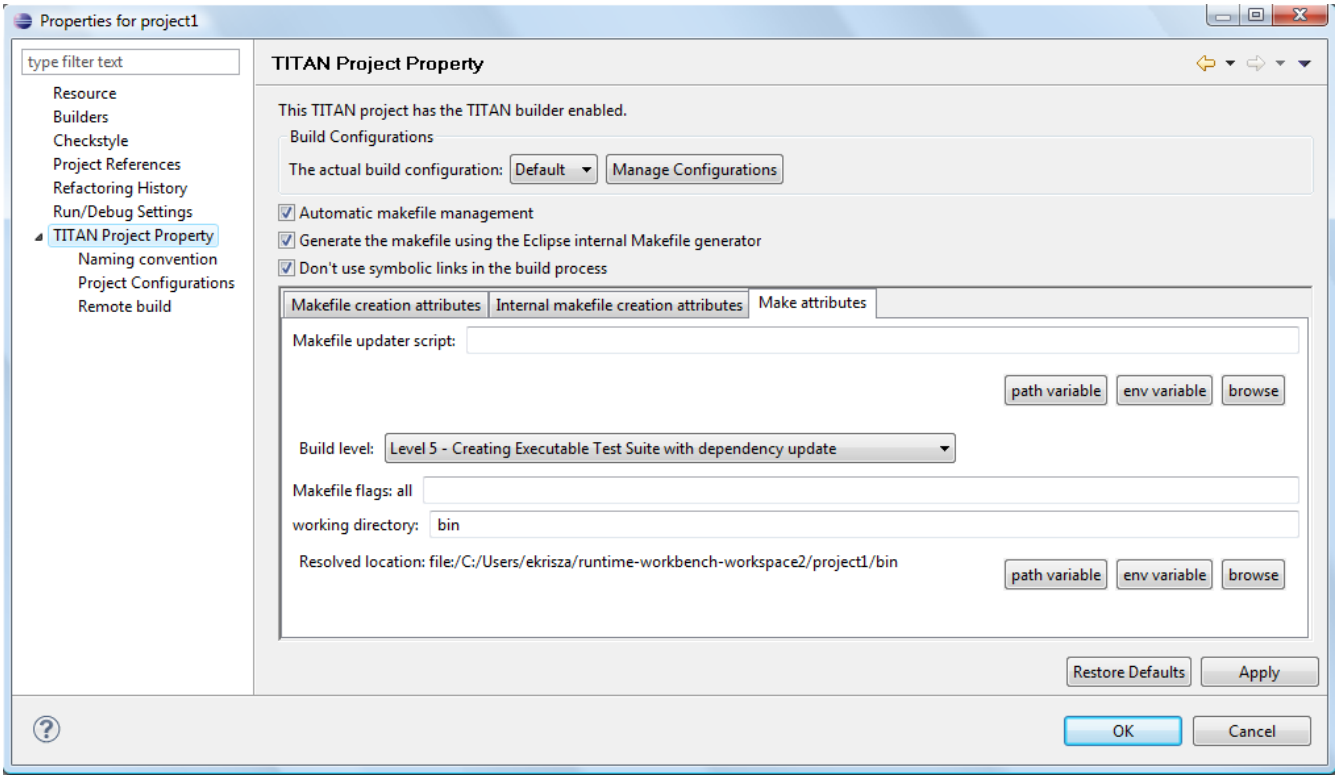


Figure 54. Make attributes tab

Figure Make attributes

On the Make attributes tab the following attributes are set:

- **The path to the Makefile updater script**

  Points out a shell script that will be run to modify to the generated Makefile. The field is checked for validity: if not empty, it must point to an existing file.

- **Build level**

  Specifies the project build level. For more information, please refer chapter Converting Existing Projects.

- **Make flags**

  Specifies the make command suffixes.

- **Working directory**

  specifies a directory used by the build operations: symbolic links and generated files will be placed in this directory. This field is checked for validity.

In the resource based project representation of TITAN projects it is impossible to tell which files are source files and which ones are generated files. For this reason, it is assumed that every file in the working directory is a generated file and every file outside the working directory is a source file (if it is not excluded from build). For this reason, the user is forced to set a working directory, or otherwise the Designer plugin does not know which files to build.

NOTE	if the provided directories are in the project, either as actual directories or linked folders, the generated files can be seen from the workbench.

4.5.3. Setting the Local Build Properties of a TITAN Java Project

For TITAN Java projects there are different configuration options available.

To set the project properties for local build first **right click** the project and select **Properties** then select **TITAN Java Project Property**.

The Internal Build Attributes Tab for TITAN Java Projects

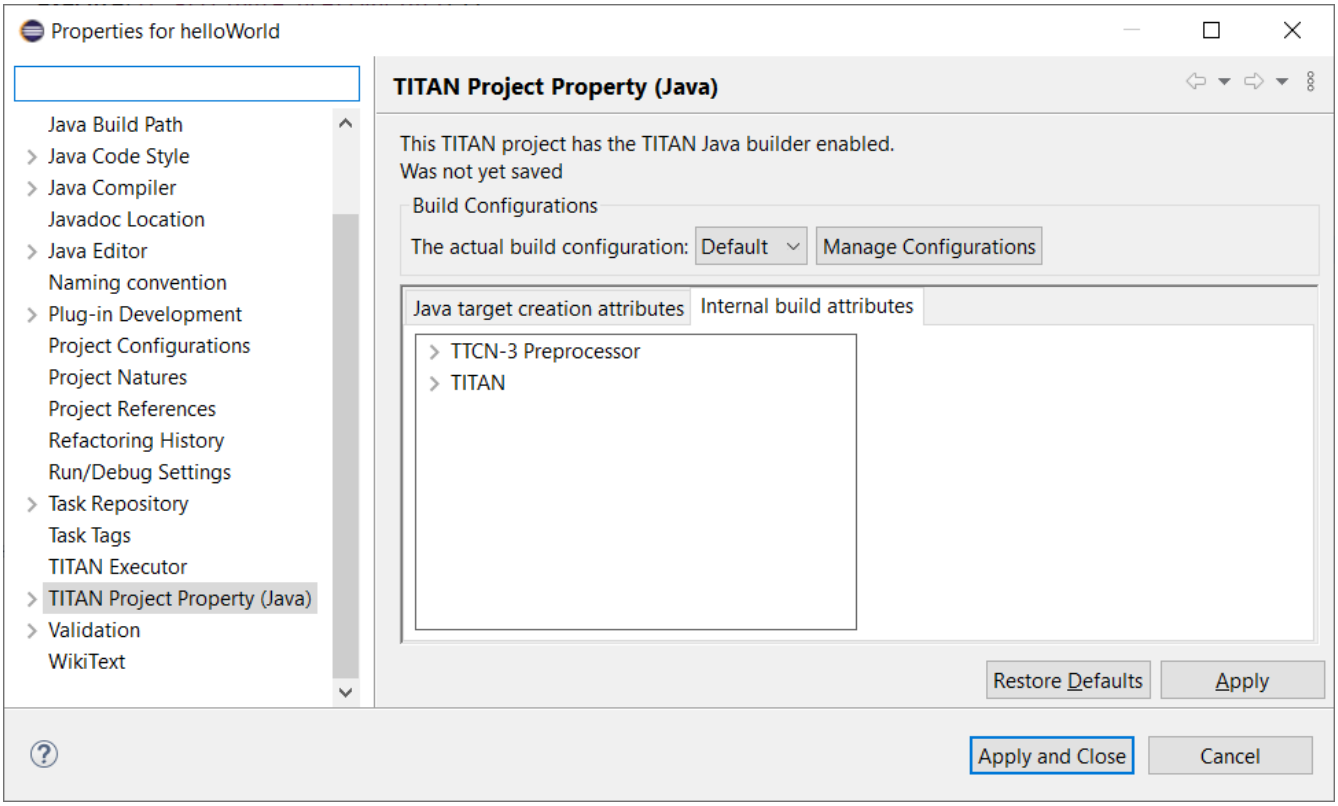


Figure 55. Internal build attributes for TITAN Java Projects

On the Internal build attributes tab the options that configure the build process can be set. To change the value of an element it must be selected. Depending on the element selected on the left side, the right hand side of the tab will contain different options.

NOTE	In TITAN Java Projects the build system is not using makefiles, but this tab resembles the Internal makefile creation attributes of TITAN Projects.

1. TTCN-3 Preprocessor
2. TTCN-3 Preprocessor Symbols

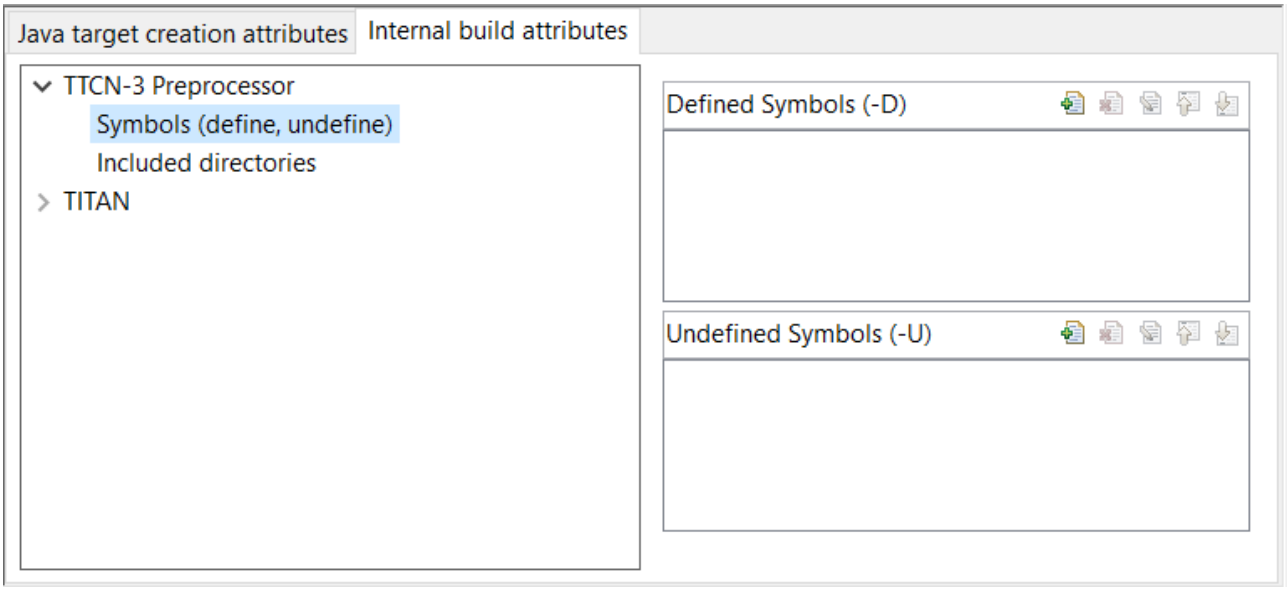


Figure 56. TTCN-3 Preprocessor symbols for TITAN Java Projects

On the symbols page it is possible to specify the list of symbols that should be defined and the list of symbols that should be undefined when the TTCN-3 pre-processor tool is executed.

These lists of options are applied during syntactic and semantic checking of the project. By default both lists are empty.

3. TTCN-3 Preprocessor Included Directories

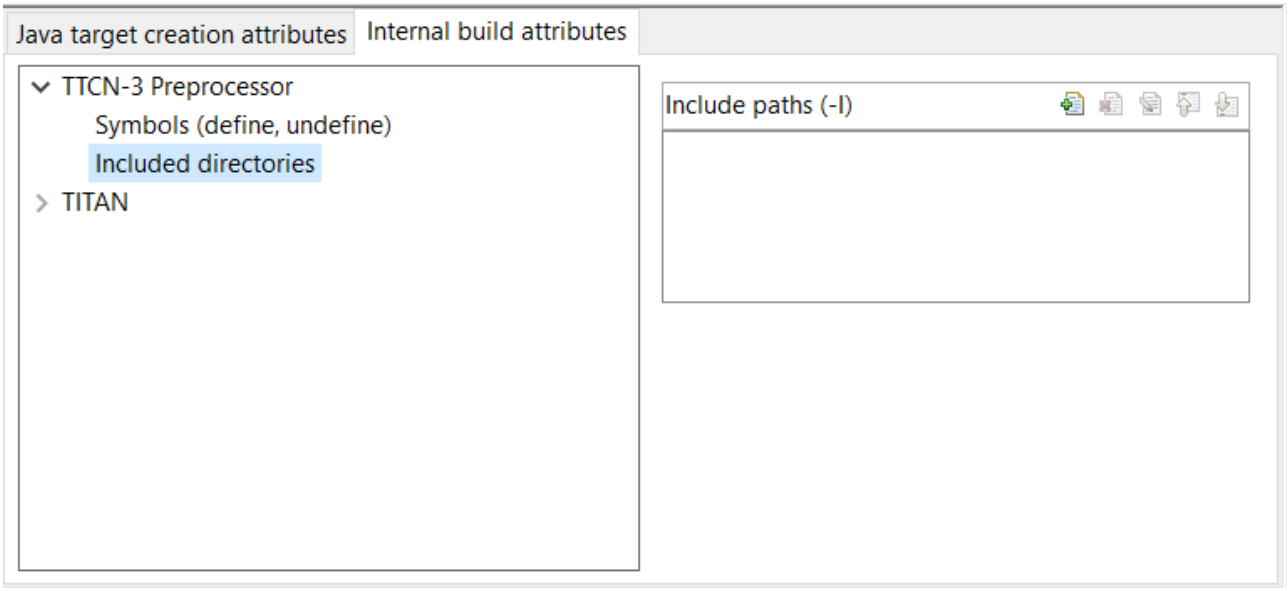


Figure 57. TTCN-3 Preprocessor include directories for TITAN Java Projects

On the included directories page, it is possible to specify the list of directories where the TTCN-3 pre-processor can look for included files.

The list of options is applied during syntactic and semantic checking of the project. By default the list is empty.

4. TITAN Flags

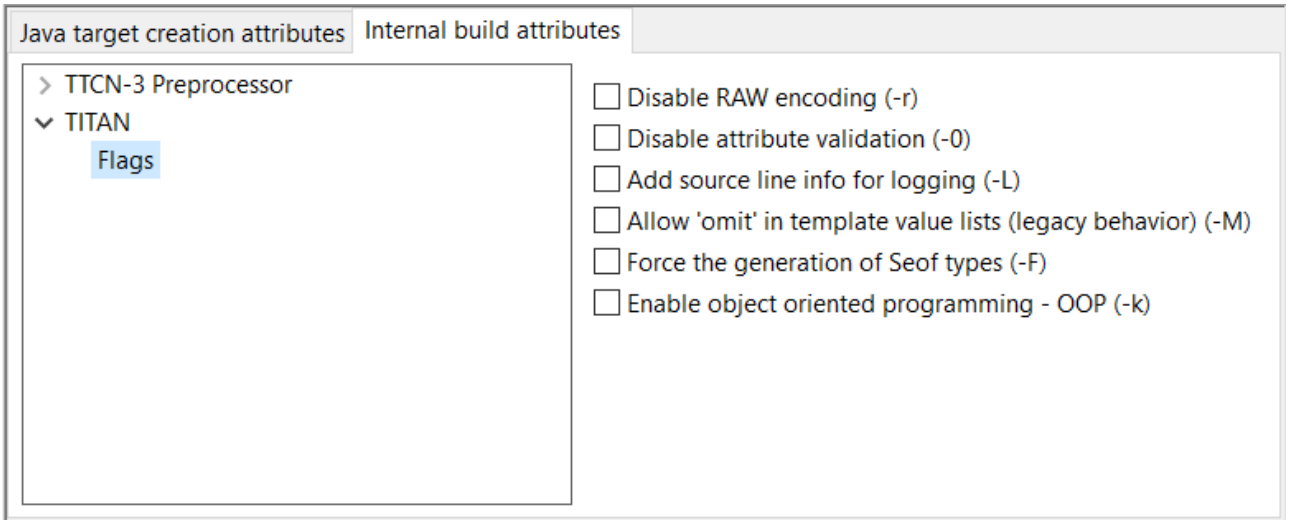


Figure 58. TITAN Flags for TITAN Java Projects

On the TITAN flags page, it is possible to specify the flags the TITAN Java code generator should use when compiling the TTCN-3 and ASN.1 files.

The options will be applied during the execution of the Java code generator.

For more information on the meanings of these options please refer to section 5.1 of the TITAN Programmer's Technical Reference guide [12].

4.5.4. Setting Project and Folder Level Naming Convention Settings

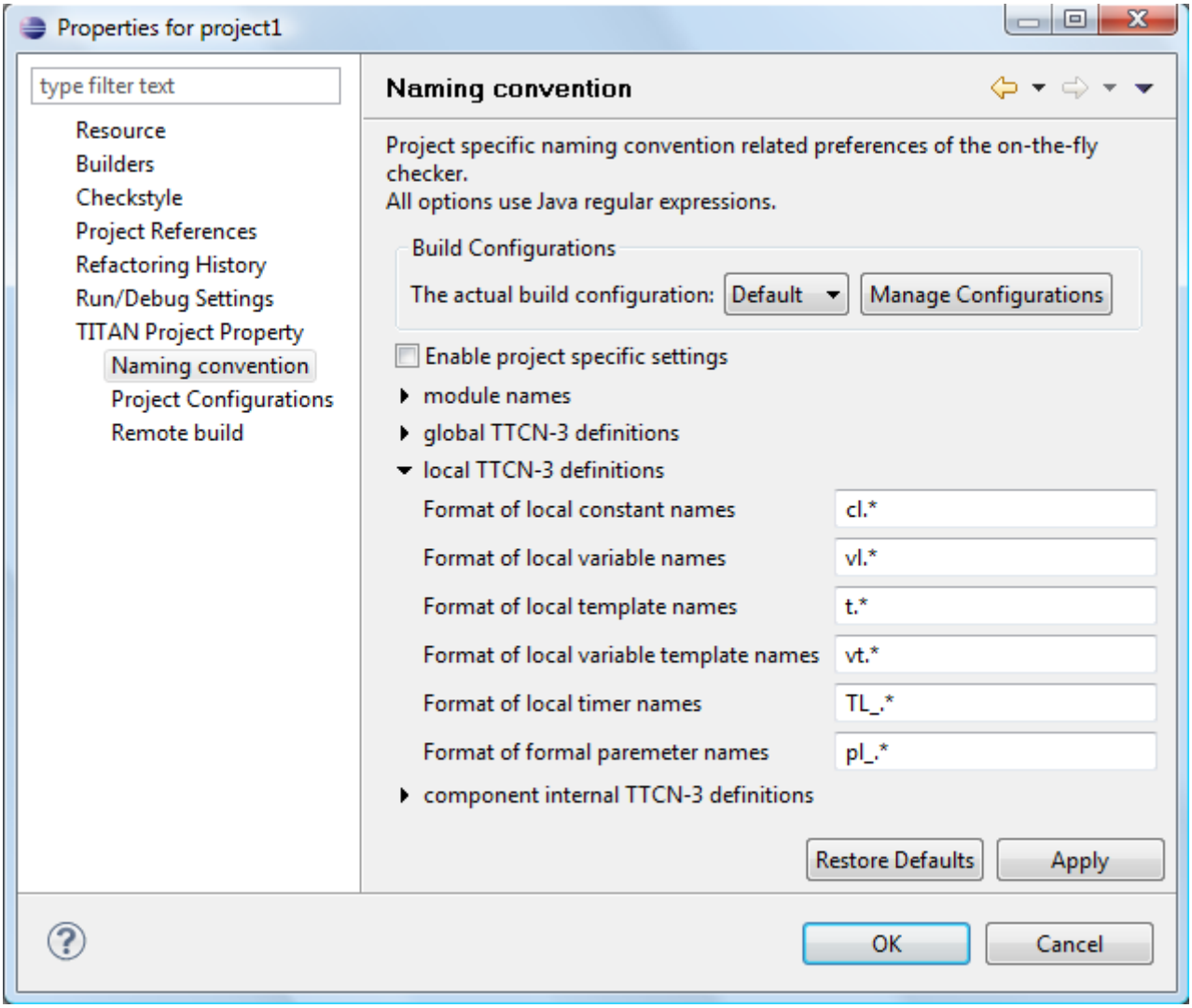


Figure 59. Project level naming convention settings

On the project and folder level it is possible to override the general workspace level naming conventions. This option can be used to further constrain the naming conventions, for example to include some project specific constants.

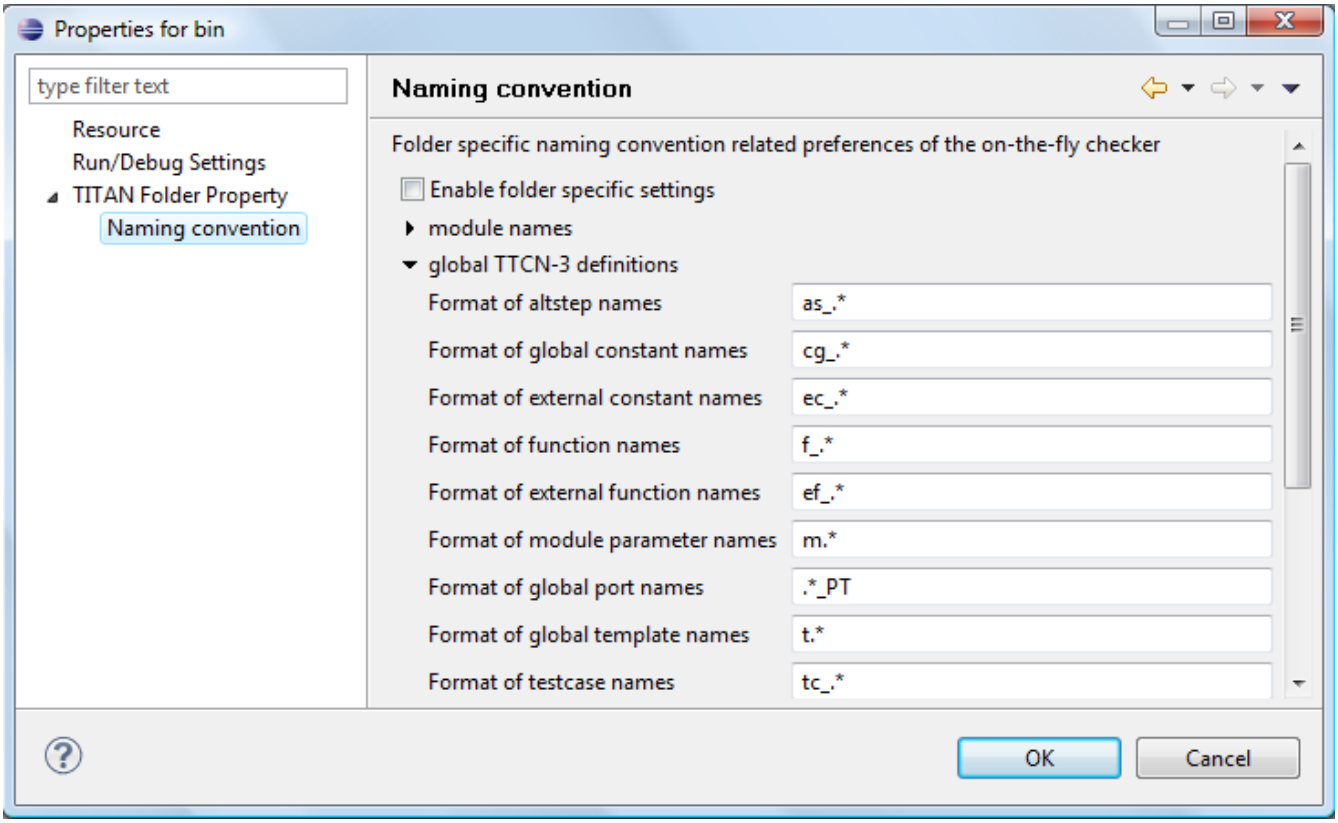


Figure 60. Folder level naming convention settings

These are same options that are available as on the workspace level.

The overriding rules are evaluated by the following algorithm:

1. It starts from the folder immediately containing the module in question.
2. It walk-searches the folder hierarchy upwards to the project either till it finds a folder that overrides the naming conventions or till it reaches the project.
3. If the folder overrides the naming conventions, it uses the settings found there.
4. If it reached the project and the project overrides the naming conventions, it uses the settings found there.
5. If it reached the project, but even the project itself is not overriding the naming conventions it will use the workspace level settings.

NOTE	It is suggested to switch off checking the naming convention because it significantly decreases the speed of the analysis. It should be switched only on at code cleaning.

4.5.5. Setting Requirements on the Configuration of Referenced Projects

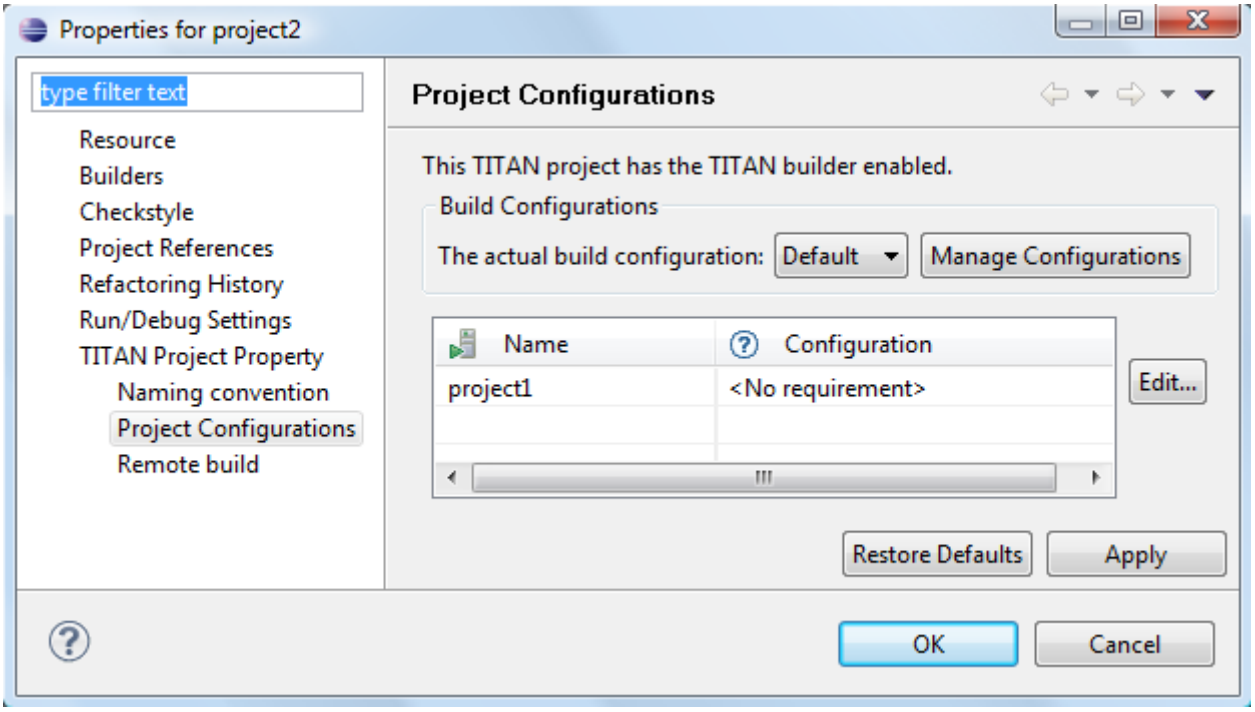


Figure 61. Requirements on the actual configuration of referenced projects

On this page it is possible to set for each project, directly referenced by the actual one, a requirement on its actual configuration. If the actual configuration on the given project is not the same as the required one it will cause a build error. This way it is possible to have fairly large project hierarchies, while still being able to consistently support build configuration for each project.

To change the requirement for a project either **select it** in the list and click on the **Edit...** button, or **double click on it** in the list.

On the window that pops up (Figure 56) it will be possible to select a configuration, from all of the configurations configured for the selected project.

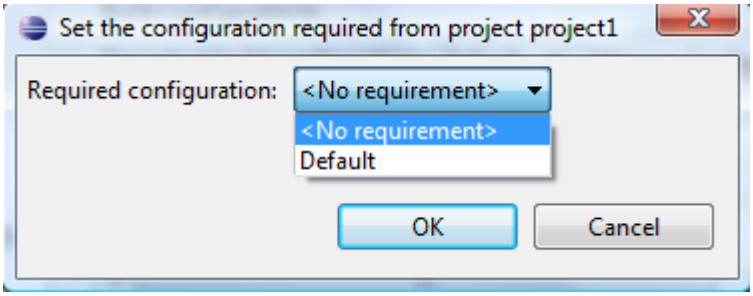


Figure 62. Configuration requirement selection window for project1

NOTE	Both in the list and on the requirement selection window the "**<No requirement>**" option is displayed if there is no requirement set for that given project at this time. If you wish to disable a previously set requirement, you have to select this option.

4.5.6. Setting the Remote Build Properties of a Project

Remote build enables building of source codes:

- on several different machines;

- on several platforms;
- in several different directories;
- with several different build settings;
- using all of the above possibilities at the same time.

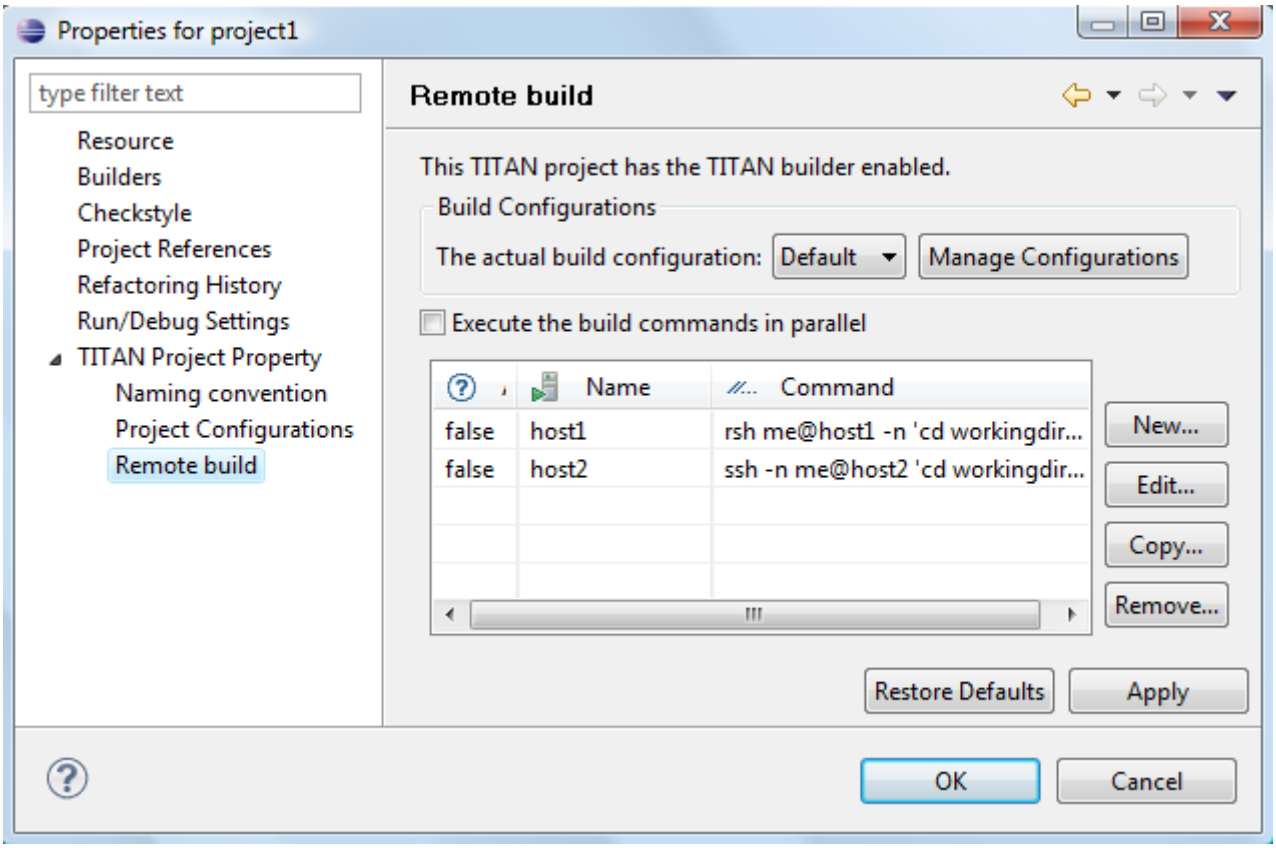


Figure 63. Remote build attributes

On this property page one or more hosts can be chosen to build the project remotely. The modalities of the remote build process on these hosts are also set.

To set the project properties for remote build, first **right click** the project and select **Properties**, then select **Remote build** on the left pane (Figure 57). (If **Remote build** is missing from the left pane, **left click** the triangle sign next to the **TITAN Project Property**; see Figure 52.)

The checkbox **Execute the build commands in parallel** controls how the provided build commands should be executed.

- If this option is NOT CHECKED (this is the default), the build commands will be executed serially, that is, one by one.
- If this option is CHECKED, the build command will be executed in a parallel fashion, meaning that each execution will start at the same time.

NOTE	The majority of the build systems requires exclusive access to the intermediate files (this is the reason why NOT SET is the default), otherwise the build process might become corrupted (this can happen for example when an intermediate file built with GCC 3.4 and another built with GCC 4.0 is linked together).

Remote build hosts have three attributes:

- **Active**

  This attribute indicates whether the host should be included in the next remote build session or not.

- **Name**

  This attribute shows the name of the host. It is only used to provide feedback to the user about the progress of the build processes. It doesn't need to be unique.

- **Command**

  This attribute contains part of the command that will be executed in the remote build process. The string inserted will be prefixed with sh –c before executing it. The default attribute content is `rsh <[user@]hostname> -n 'cd <working directory>; make dep; make'`, and the string inserted must follow this pattern.

The user can control the build hosts using the buttons to the right from the table.

The **New...** button is used to create a new remote build host. It brings up the remote build host configuration window (Figure 58), where the properties of the new build host can be set. The new build host will be added to the end of the list of build hosts. Host creation can be cancelled by pressing the **Cancel** button, while the new host data is validated by pressing the **OK** button.

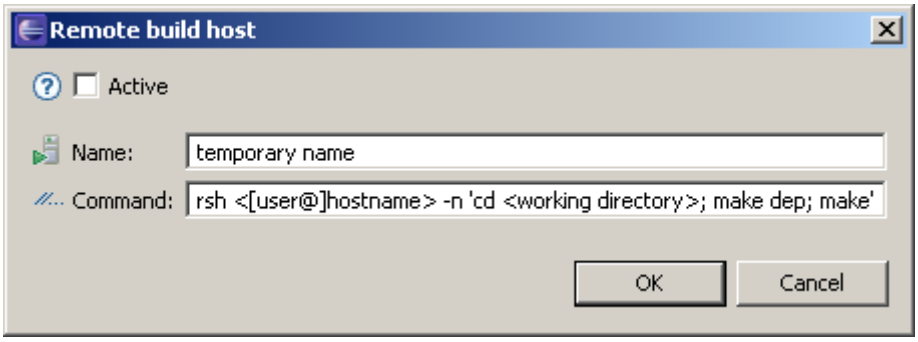


Figure 64. Remote build attributes of a host

The **Edit...** button is used to edit the attributes of an existing remote build host. Before pressing the button, the host to be edited must be selected from the table. By pressing the button, the remote build host configuration window (Figure 58) will appear, showing with the current properties of the selected host. Changes made to the host can be revoked by pressing the **Cancel** button, while modifying the host is done by pressing the **OK** button.

The **Copy...** button is used to create a copy of an already existing host. Pressing this button will create an exact copy of the currently selected host. This way of creating a new host can be beneficial for example when the build command of the new host only slightly differs from the build command of the source host. Copying is abandoned by pressing the **Cancel** button, while it is confirmed by pressing the **OK** button.

The **Remove...** button is used to remove an existing host from list of remote build hosts. The command is abandoned by pressing the **Cancel** button, while it is confirmed by pressing the **OK**

button.

NOTE	The saving of every change done on this page is validated by pressing the **Apply** or **OK** buttons at the bottom on the property page (Figure 57).

NOTE	The TITAN Java Projects do not have a remote build preference page. As Java is platform independent, there is no need to create platform specific binaries for particular machines.

Pitfalls

In case the rsh command is not present one should use the ssh command instead. In this case the default command to start from should be: `ssh -n <[user@]hostname> 'cd <working directory>; make dep; make`

As there is no way to enter a password when logging in to a remote machine, it is of crucial importance to set the login mechanism of the remote machine, to not require a password on login.

4.6. Excluding Files and Folders from the Build Process

A file or a folder excluded from the build process won't be placed into the generated `Makefile`. For this reason, once an exclusion or inclusion has taken place, the `Makefile` and the symbolic links are updated (provided that automatic `Makefile` management is enabled for the project).

Excluding a folder from the build process also means that every file and subfolder contained in that folder will be excluded, too.

If a file or folder is excluded from build, its name is decorated with the string `[excluded]`, provided that TITAN decoration is enabled (see here).

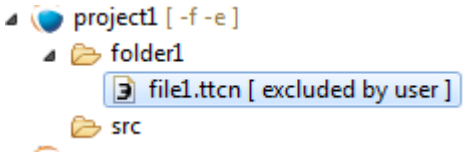


Figure 65. Excluded from build

4.6.1. Excluding a File from the Build Process

A file can be excluded from build or included in the build in two different ways described below.

NOTE	There are some special files that can never be included into the build. In Eclipse these are project related plug-in resources, which by convention never have a name, just an extension, for example `.TITAN_properties`. Such files (that don't have a name), are always excluded from build, no matter how their property is set.

To access File properties (the first alternative): **right click** the file and select **Properties**. On the **Properties for ...** window, select **TITAN File Property**. Here the exclusion state of the file can be set via ticking the **Excluded from build** box.

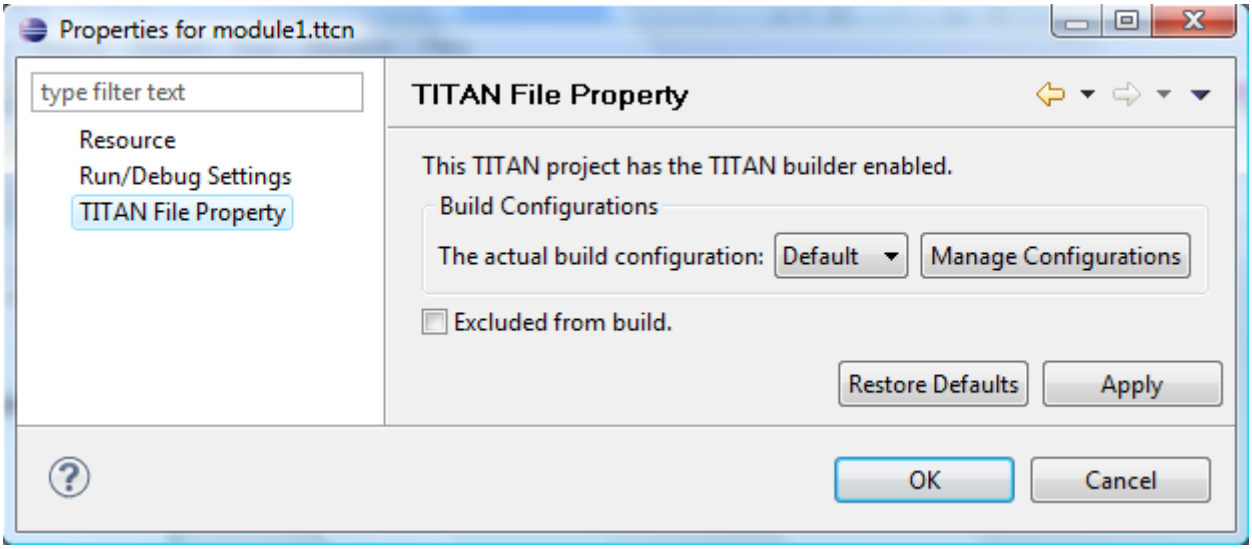


Figure 66. TITAN file property

To access the Pop-up menu (the second alternative), **right click** the file and select **TITAN / Toggle exclude from build state**. This method has the advantage that the exclusion state of several selected files can be changed all at once.

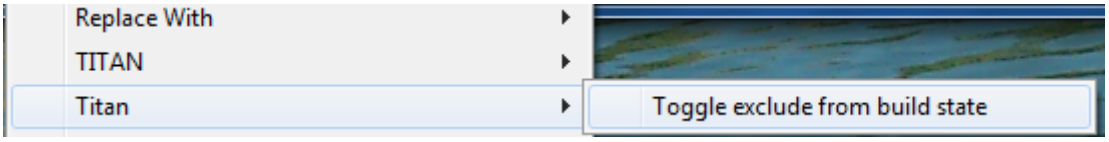


Figure 67. Toggle exclude from build menu

4.6.2. Excluding a Folder from the Build Process

A folder can be excluded from build or included in the build in two different ways described below.

NOTE	There are some special folders that can never be included into the build. In Eclipse by convention folders having a name which starts with a . (dot) are used for storing special files or folders, that one or more plug-ins might temporarily create. Such folders and for this reason their whole content is always excluded from build, no matter how their property is set.

To access Folder properties (the first alternative), **right click** the folder and select **Properties**. On the **Properties for ...** window, select **TITAN Folder Property**. Here the exclusion state of the folder can be set via ticking the **Excluded from build** box. (The other checkbox, **Folder is in central storage**, is described here.)

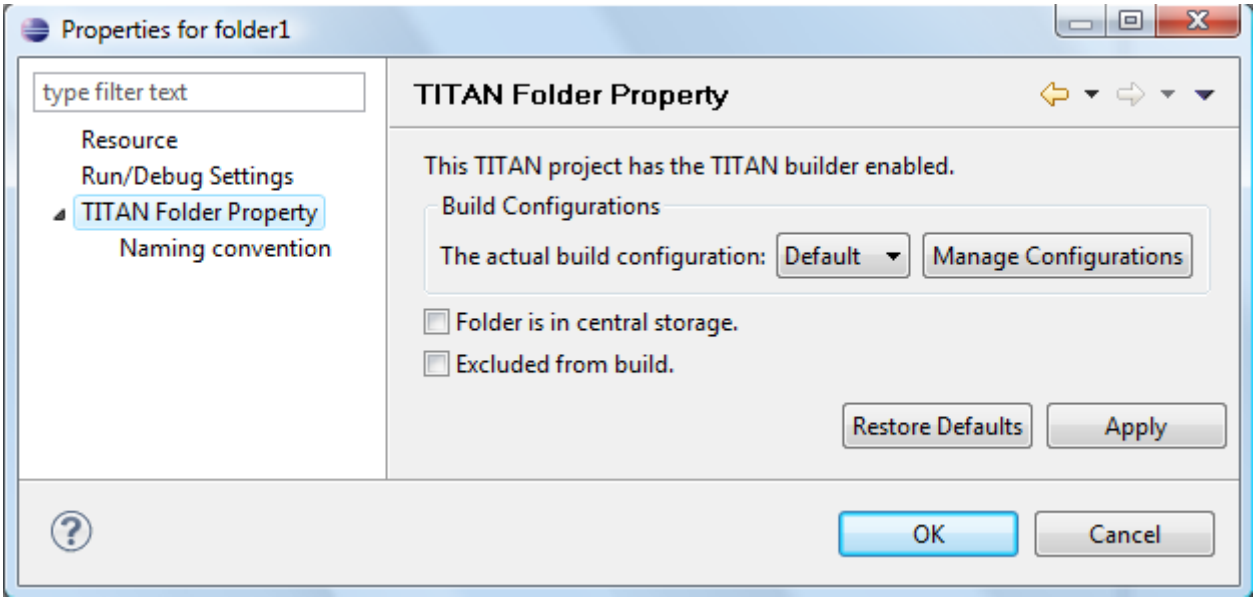


Figure 68. TITAN folder property

To access the Pop-up menu (the second alternative), **right click** the folder and select **TITAN / Toggle exclude from build state**. This method has the advantage that the exclusion state of several selected folders can be changed all at once (see Figure 61 above).

4.7. Converting a Folder into a Central Storage

A folder marked as Central Storage is assumed to have its own `Makefile`. For this reason, when this property of a directory is toggled, the `Makefile` and the symbolic links are updated (provided that automatic `Makefile` management is enabled for the project). For description of the Central Storage concept, please refer to the TITAN User Guide ([3]).

A directory's Central storage property can be toggled the following way:

Right click on the folder, select **Properties** and in the **Properties for ...** window click **TITAN Folder Property**. Here the central storage state of the folder can be toggled via ticking the **Folder is in central storage** button (Figure 62).

4.8. Opening and Closing Projects

A closed project cannot be edited; even its contents are hidden. This is useful to decrease memory occupation and computational load: a closed project does not use any resources.

In Eclipse, projects can be opened and closed by **right clicking** the project and selecting **open project** respective **close project**.

4.9. Saving and Loading Project Properties

There is no need to save or load the project properties file, as this is done automatically. When files or folders are added or removed, or their properties are changed, the TITAN Designer plug-in automatically saves the new properties into the `.TITAN_properties` file, which always resides in the root directory of the project. When the content of this file is edited and saved, or when the TITAN

Designer plugin starts up noticing that files were changed while it was not active, then it automatically loads the file's contents and modifies the resources properties accordingly.

Besides the obvious use this is useful if more people are working on the same project. Someone updates the properties of the resources and sends the file to the others; when the recipients save the file the properties of their resources will be updated automatically.

4.10. Importing and Exporting Projects

Importing and exporting projects can be done in many ways in Eclipse. Out of those 3 will be shown in detail: a native way, one using the TITAN project descriptor format, and a way to import project from the old mctr_gui format.

It is important to turn off automatic building and to refresh the project before importing and exporting. Because of the changing nature of the projects, it can be expected that there will always be files which are out of synchrony with the file system. Importing and exporting can only be done if every file in the project is in synchrony with their file system counterparts.

NOTE	Exporting and importing without archiving is almost exactly the same.

The following steps should be done before exporting a project:

1. Automatic building should be turned off, so that further operations will not invoke any build related functionality.
2. Optionally the project should be cleaned to reduce the size of the exported data.
3. The project should be refreshed (**right click** the project and select **Refresh**), to synchronize the files and the file system.

4.10.1. Exporting Projects in Native Format

To export a project using a native way, for example into an archive file, follow the steps described below:

1. **Right click** the project to be exported and select **Export**.

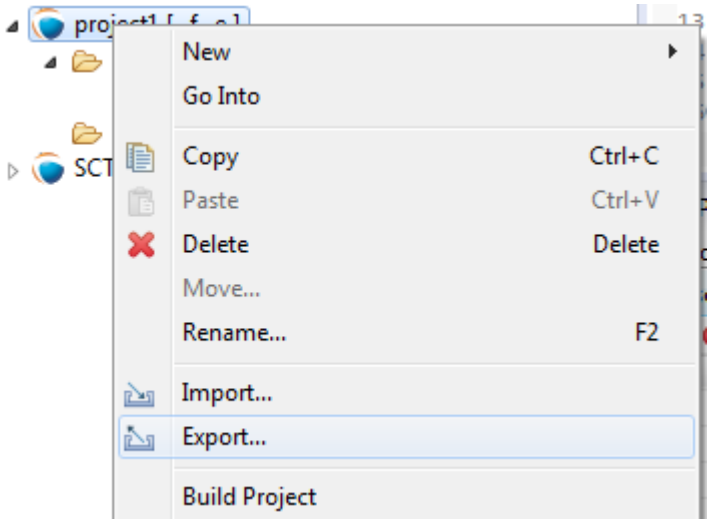


Figure 69. Export menu

2. On the **Export** window select **General** / **Archive File** and press **Next**.

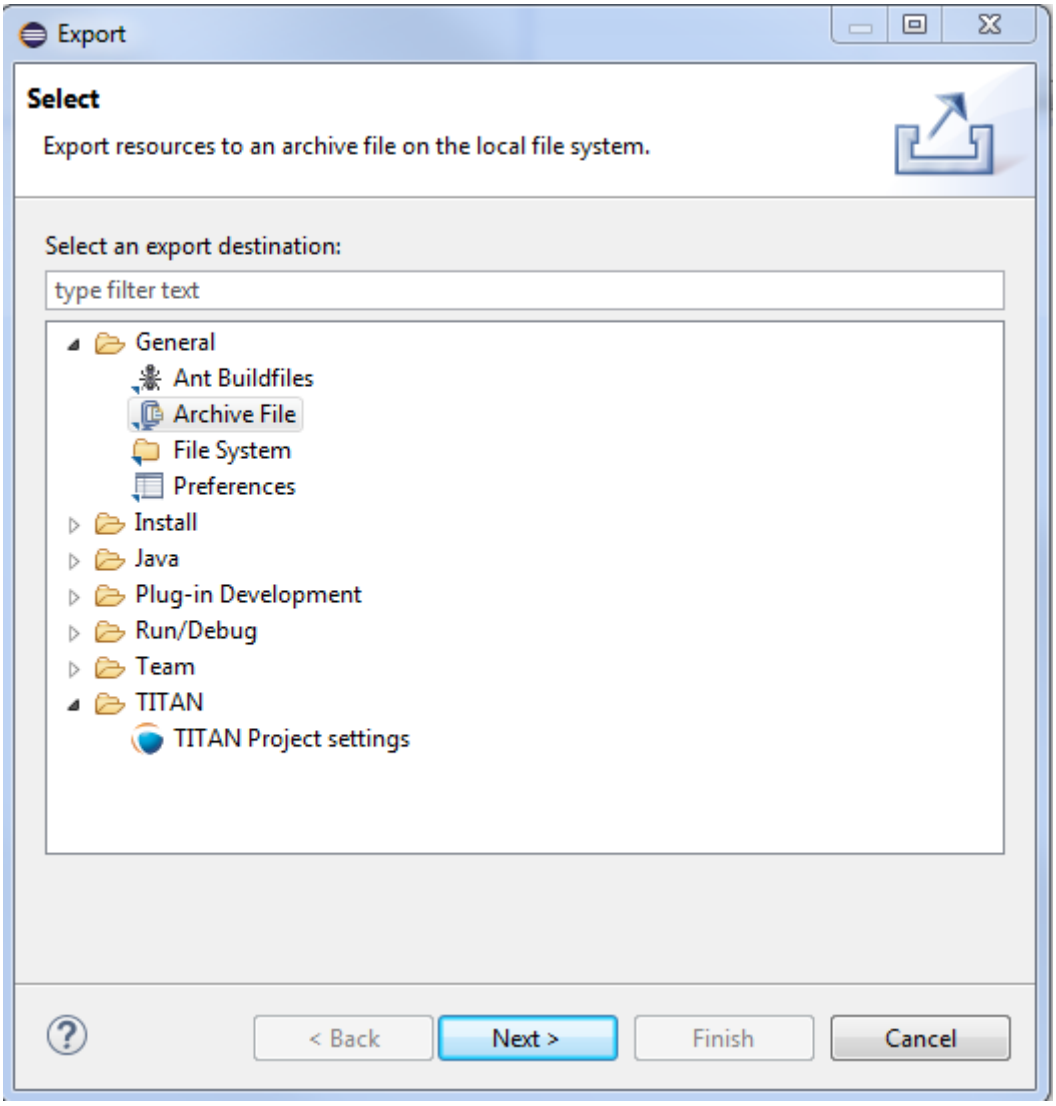


Figure 70. Export common dialog

3. Fill in the fields in the **Export Archive file** wizard.

NOTE	it is advised to export every file related to the project, and also to export only those files in the archive which belong to the project.

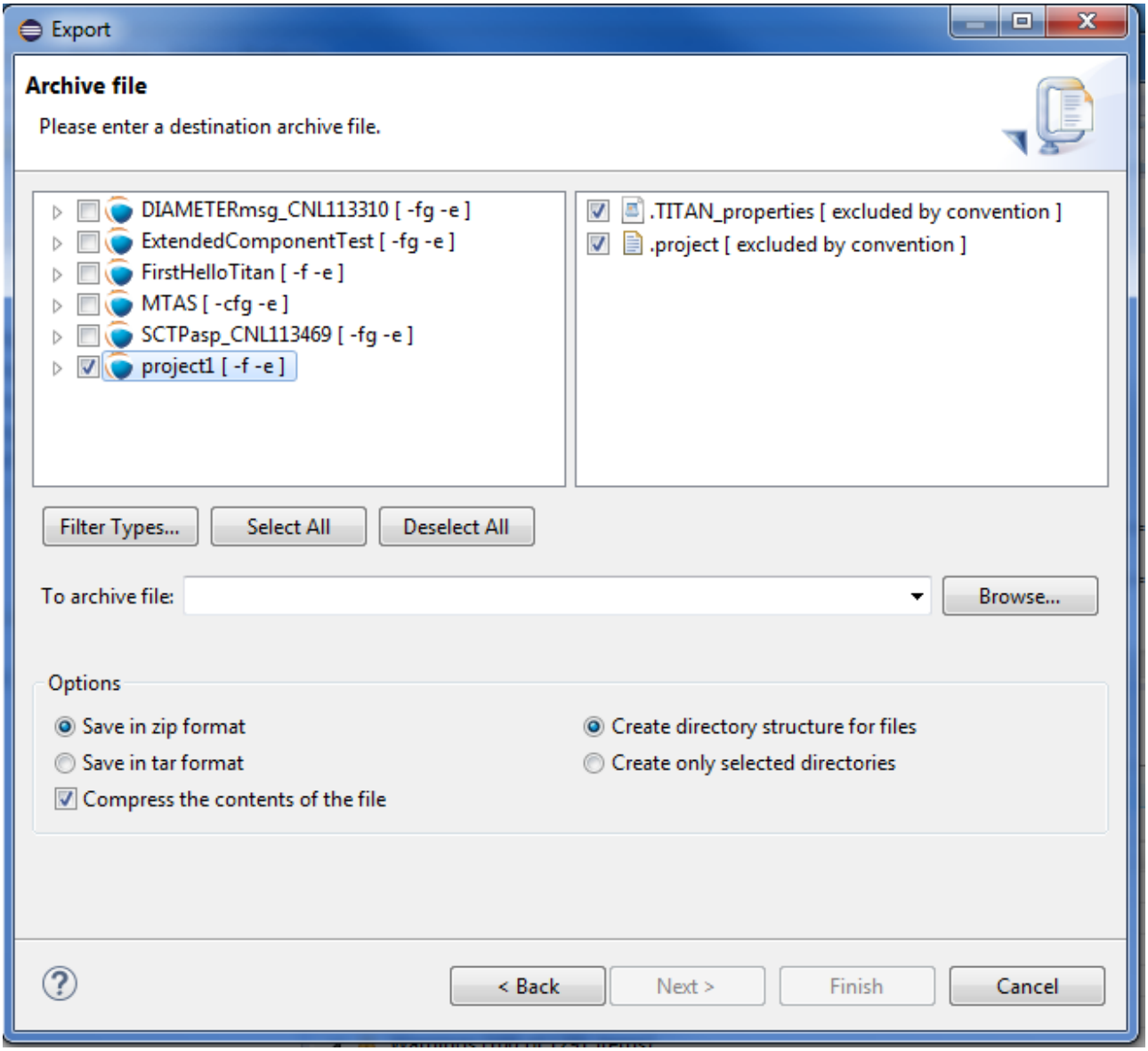


Figure 71. Export Archive file wizard

NOTE	This will export the whole project: not just the information on settings, but also the files and folders themselves.

4.10.2. Importing Projects from Native Format

To import a project from a native format, for example an archive file, follow the steps described below:

1. **Right click** somewhere in **Project Explorer** and select **Import**, as shown on Figure 63 above.
2. On the **Import** window select **General / Existing Projects into Workspace** and press **next** (below).

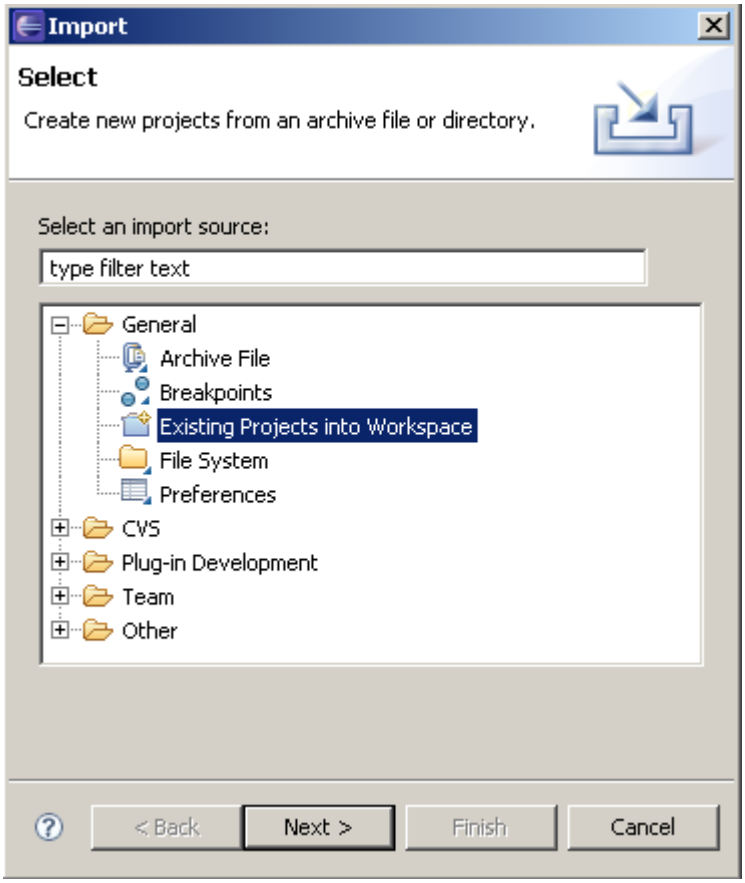


Figure 72. Import common dialog

3. In the **Import Projects** wizard select the archive to import from. Eclipse will list the projects the archive contains. Select one or more of them and press **Finish**.

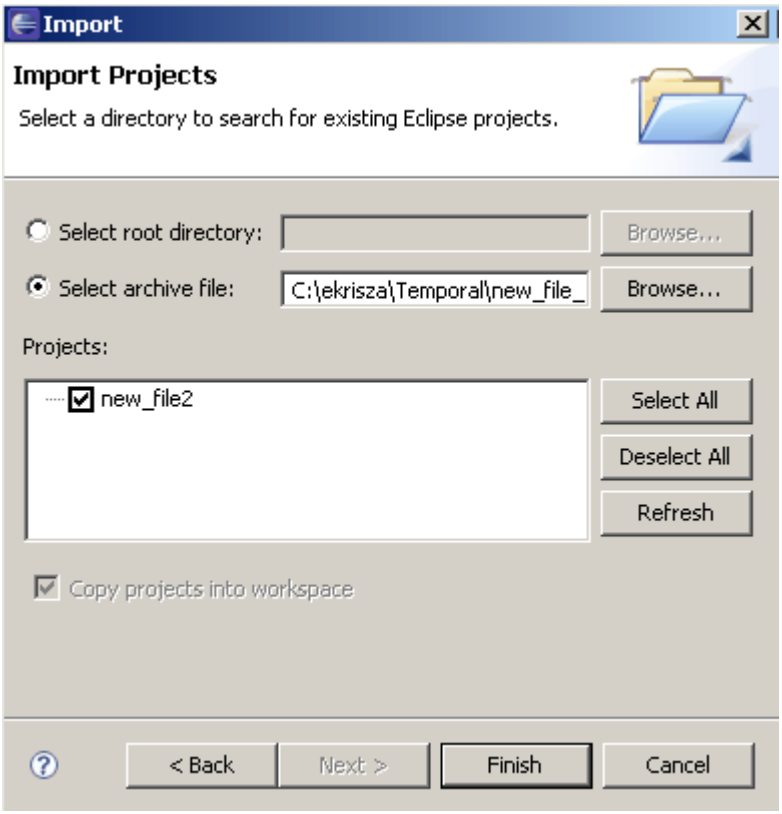


Figure 73. Import Archive file wizard

4.10.3. Importing an Existing mctr_gui Project

To import a project from an existing mctr_gui project file follow the steps described below:

1. **Right click** somewhere in **Project Explorer** and select **Import**, as shown on Figure 63. On the **Import** window select **TITAN / Project from .prj file** and press **next** (below).

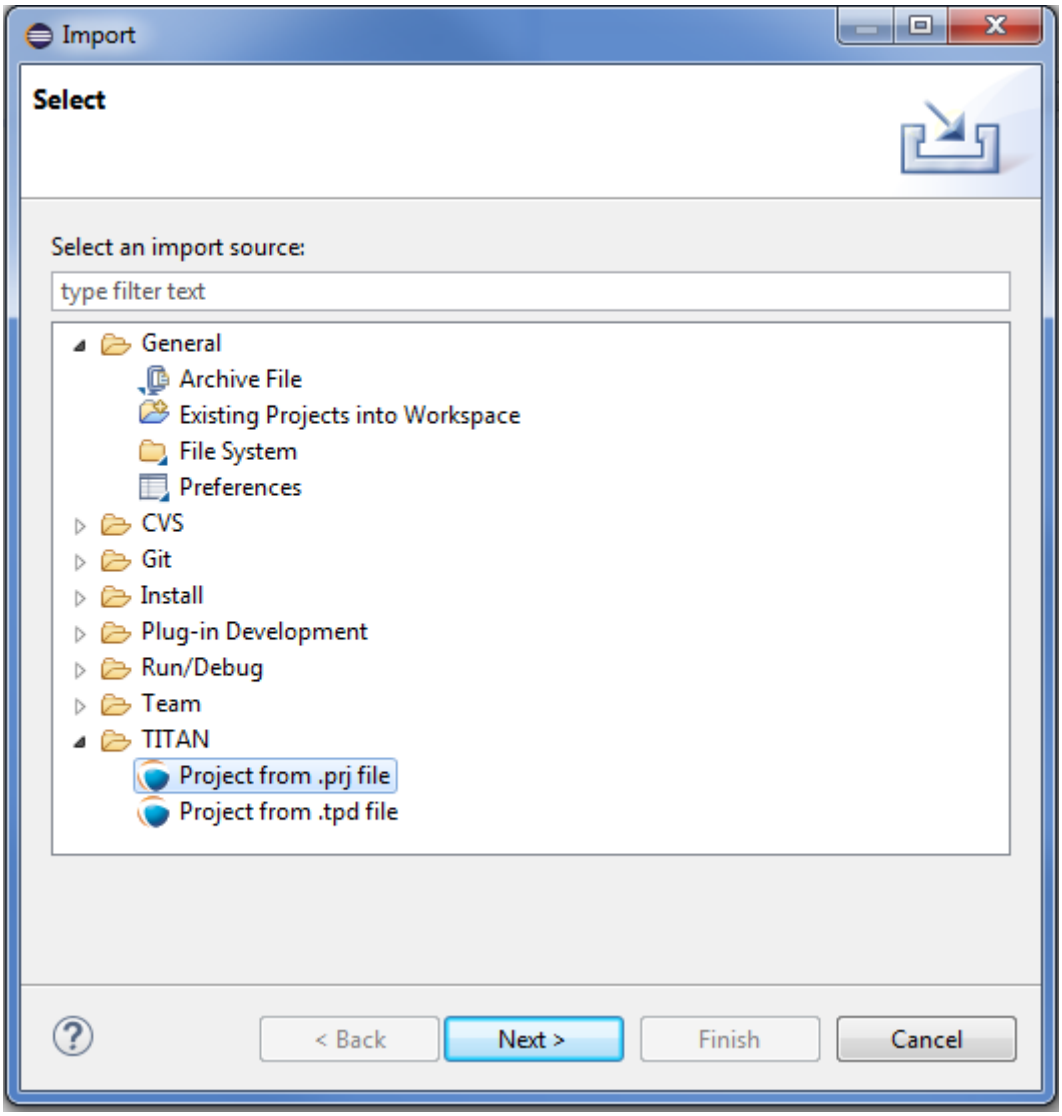


Figure 74. Import from .prj file

2. On the **Import new TITAN Project from .prj file** wizard select the original project file to import from and press **Next**.

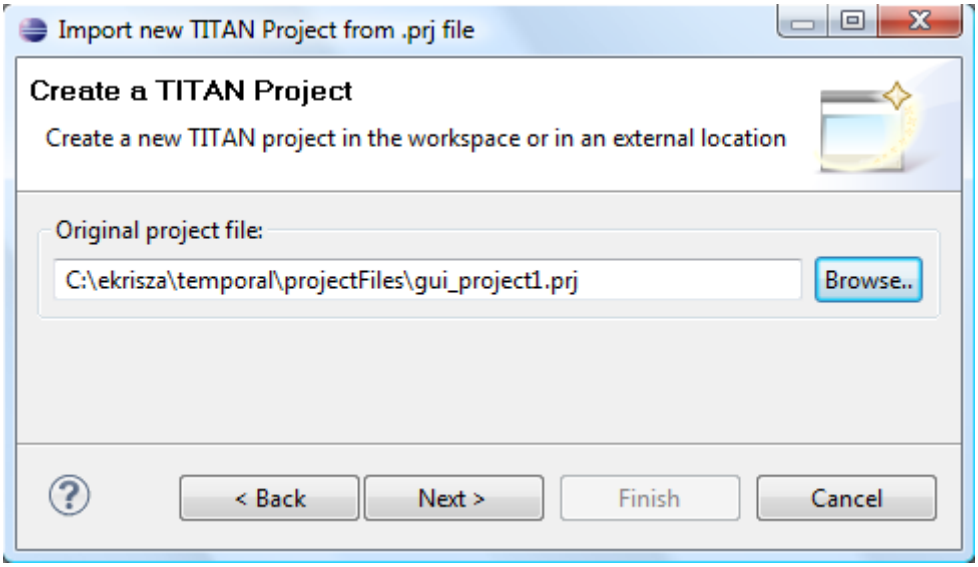


Figure 75. Import new TITAN Project from .prj file

3. Select the name and location of the new project to be created.

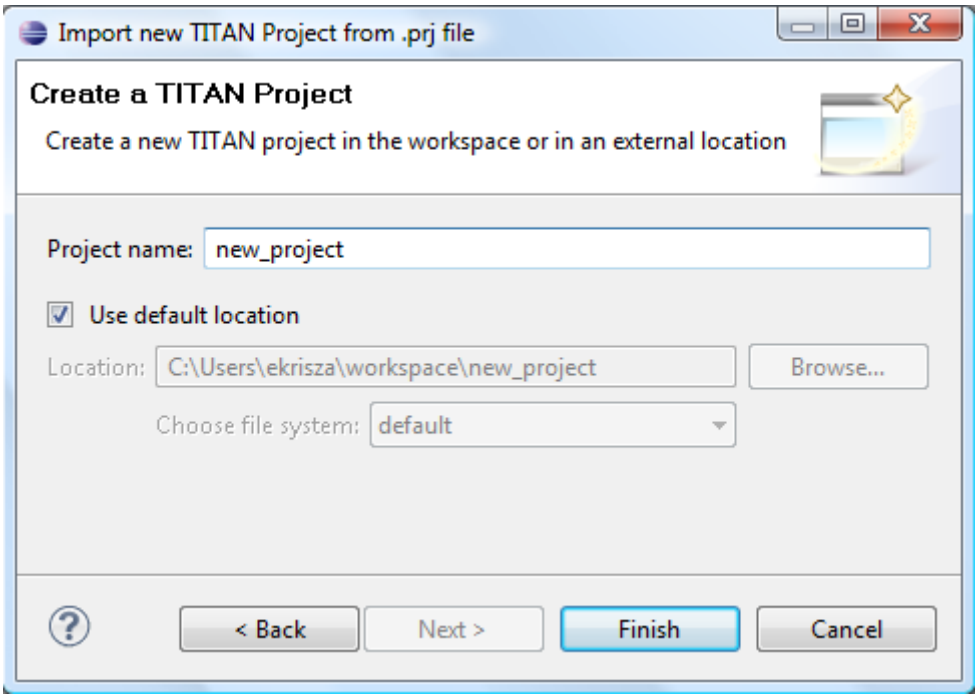


Figure 76. Name of the new project

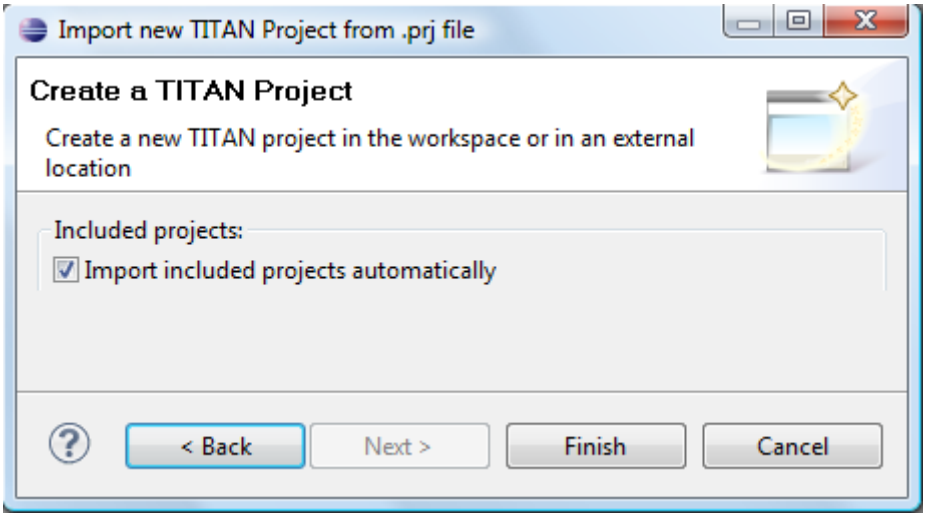


Figure 77. Create the included projects automatically

4. On the last page of the wizard it is possible to select whether included projects (if any exists) should be imported automatically or not.

The wizard will now create the new project, populate it with the files referring to the ones provided by the mctr_gui project file and set all options for the project which can be transferred.

For more information on how the project is converted to this format please refer to Section 5.

4.10.4. Importing Files as Linked Resources

Linked resources are files and folders which are not physically copied into the Eclipse workspace nor linked as soft or hard linked there (at least not into the source folder just later into the build folder under the building process). Linked resources are stored primarily internally in the Eclipse. When linked resources are modified, the original files will be modified. This is the most useful ttcn source file handling method.

To import folders and files as "linked resources" follow the steps described below.

1. Create an empty project without `src` subfolder according to Creating a New TITAN C++ Project. The project name should be the same as the name of the project to be imported.
2. Right click on the project name and select **Import**, as shown on Figure 63 above. On the **Import** window select **General** / **File System** and press **Next** as shown on below.

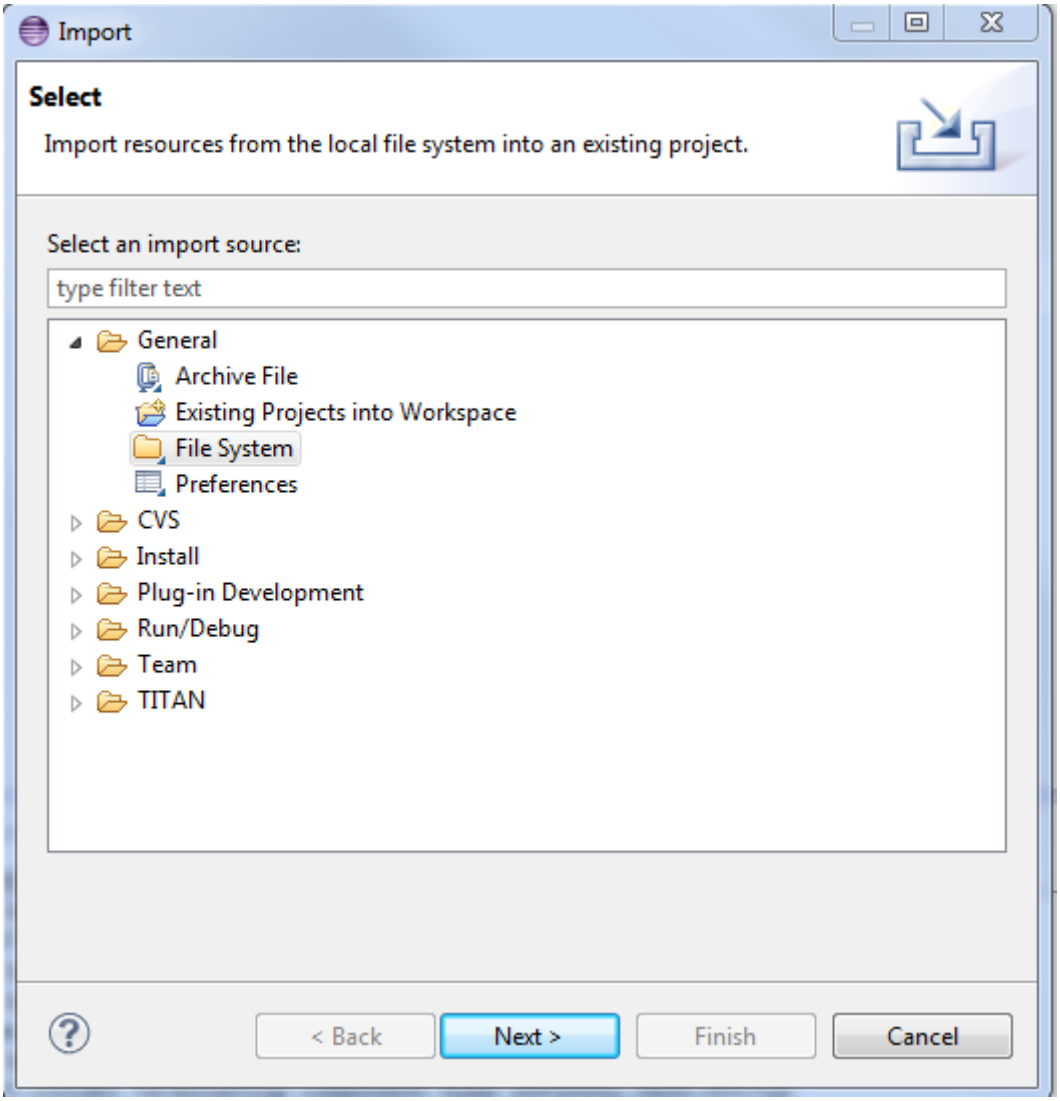


Figure 78.

3. In the Import / File system dialog select **Browse** (as seen below), then find and select the `src` folder of the project to be imported.

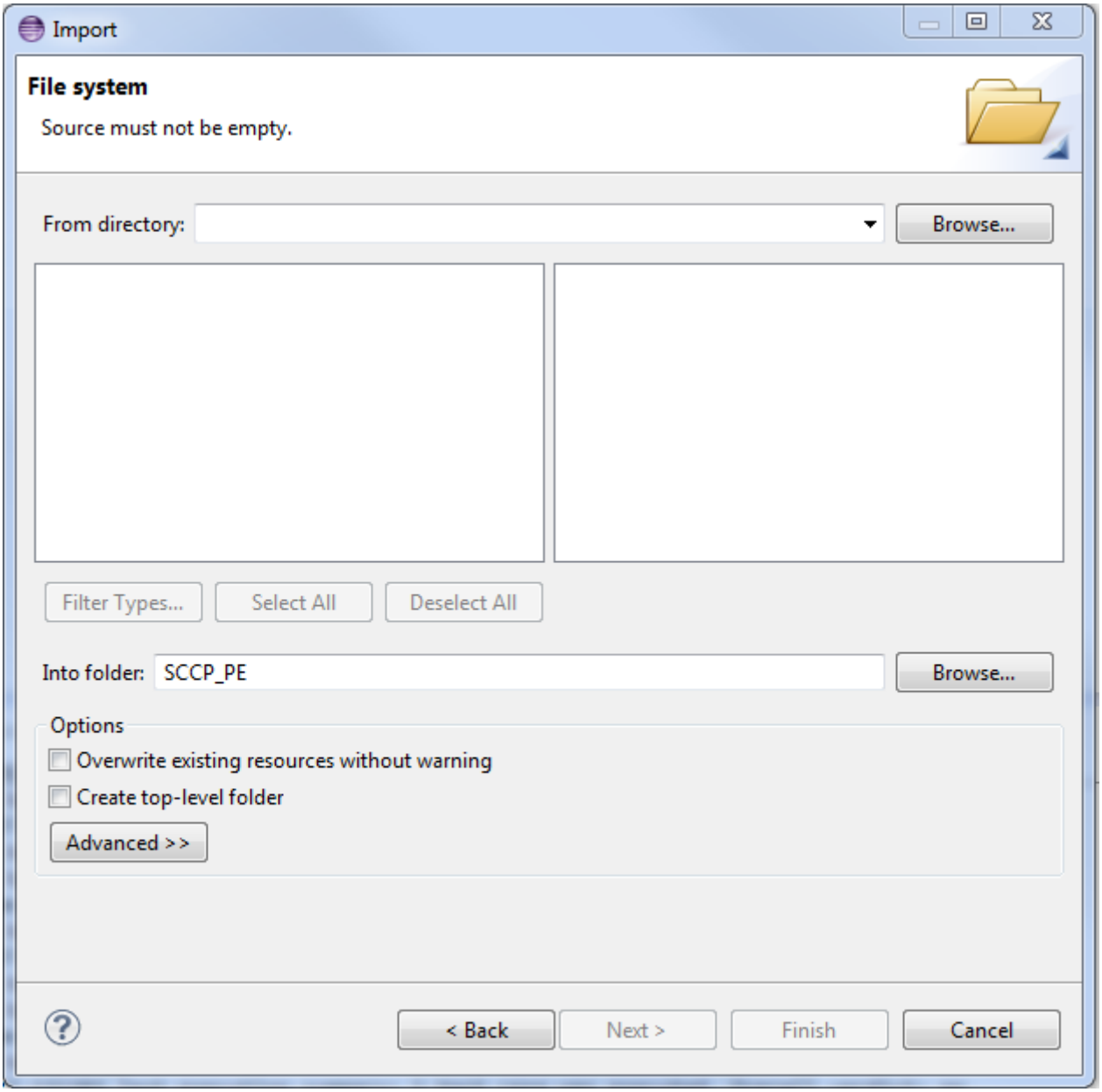


Figure 79.

4. Click on the button **Advanced**>> in the dialog, select the options **Create link in workspace** and unselect options **Create virtual folders** and **Create link locations relative to:** as shown on below.

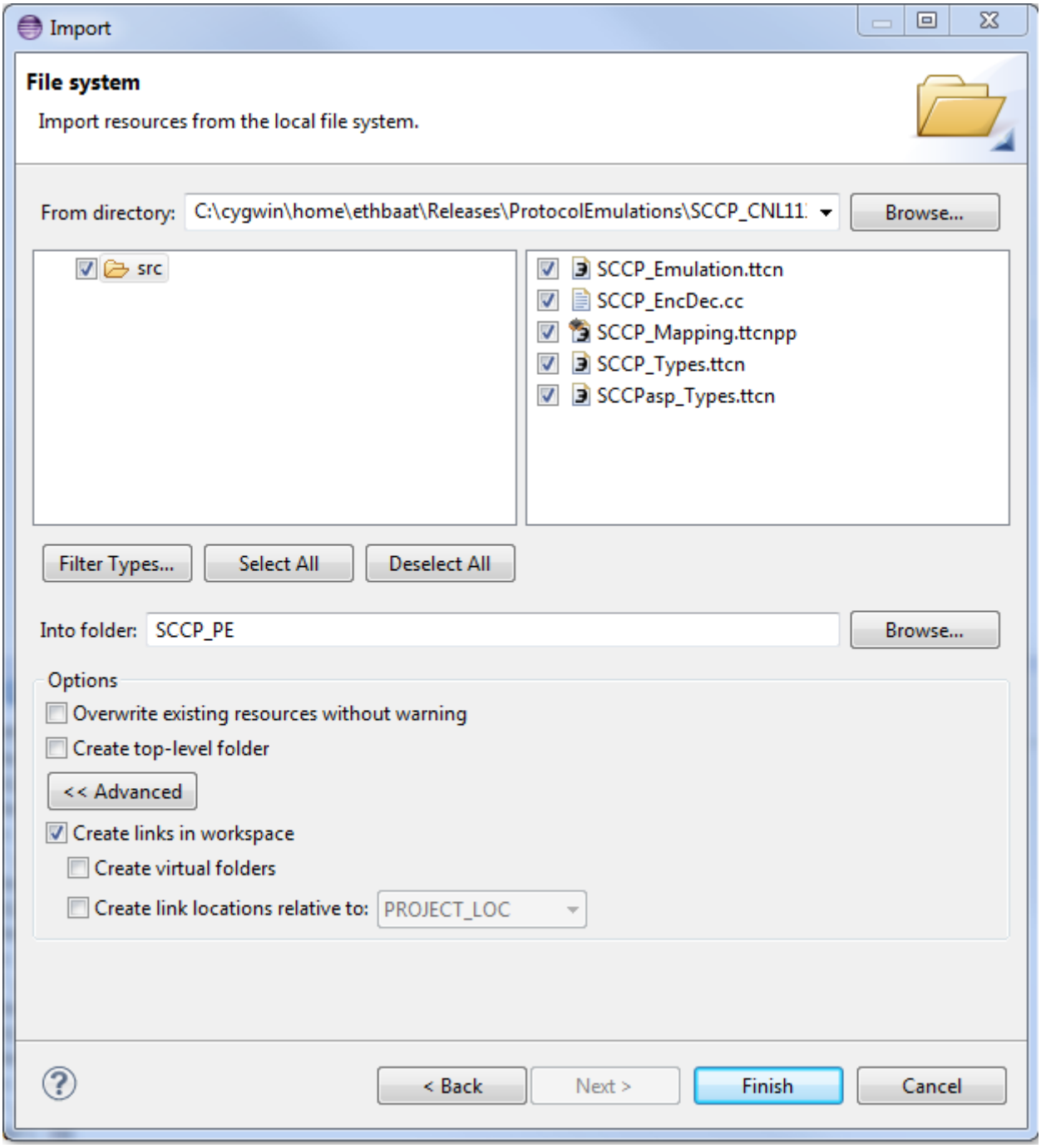


Figure 80.

5. Push Finish. The `src` folder appears under the project name in the Project Explorer as linked resource (the icon before the `src` folder contains a little link arrow) as shown below.

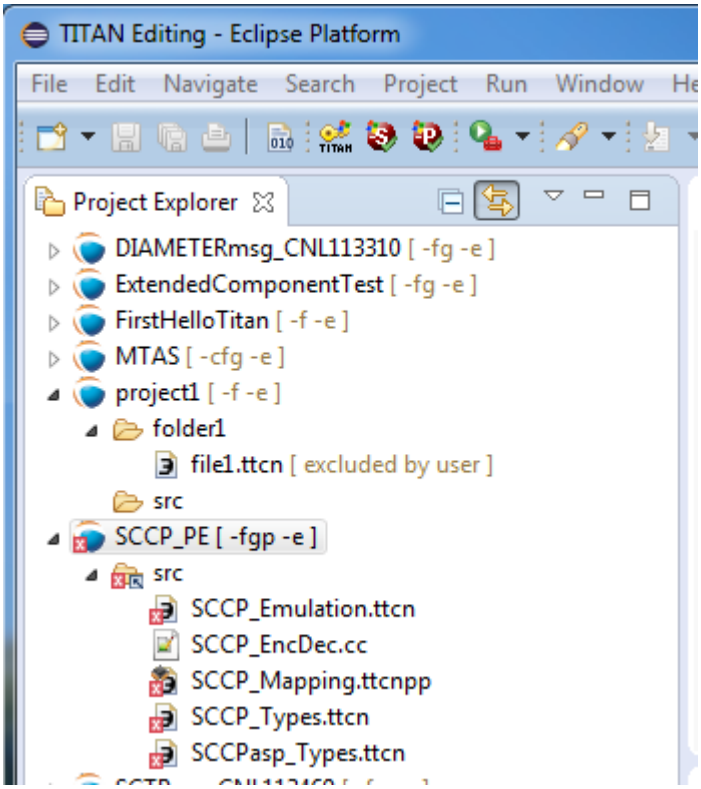


Figure 81. The result of the import

4.10.5. Exporting Projects into the TITAN Project Descriptor (tpd) Format

Exporting only project information into TITAN Project Descriptor (tpd) format can be performed manually or automatically.

NOTE	Exporting TITAN Java projects into TITAN Project Descriptor (tpd) format is not available.

Exporting Project manually into the TITAN Project Descriptor (tpd) Format

To export the project information into a `tpd` file, follow the steps described below:

1. **Right click** on the project to be exported and select **Export**.
2. On the **Export** window select **TITAN / TITAN project settings** and press **Next** (see the figure below):

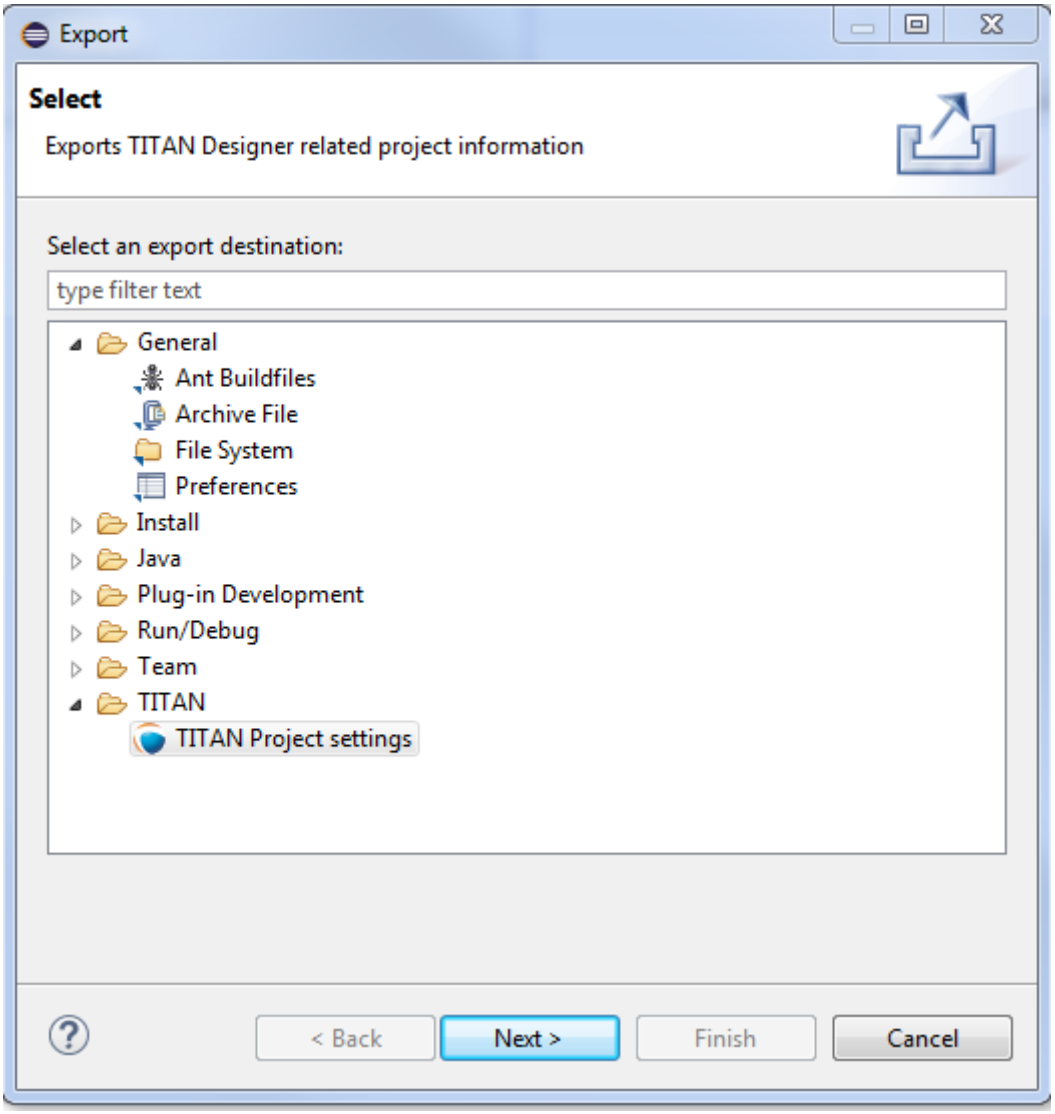


Figure 82. Export to TITAN project descriptor

3. Select the file where the information should be exported to, and press **Next** (see the next figure).

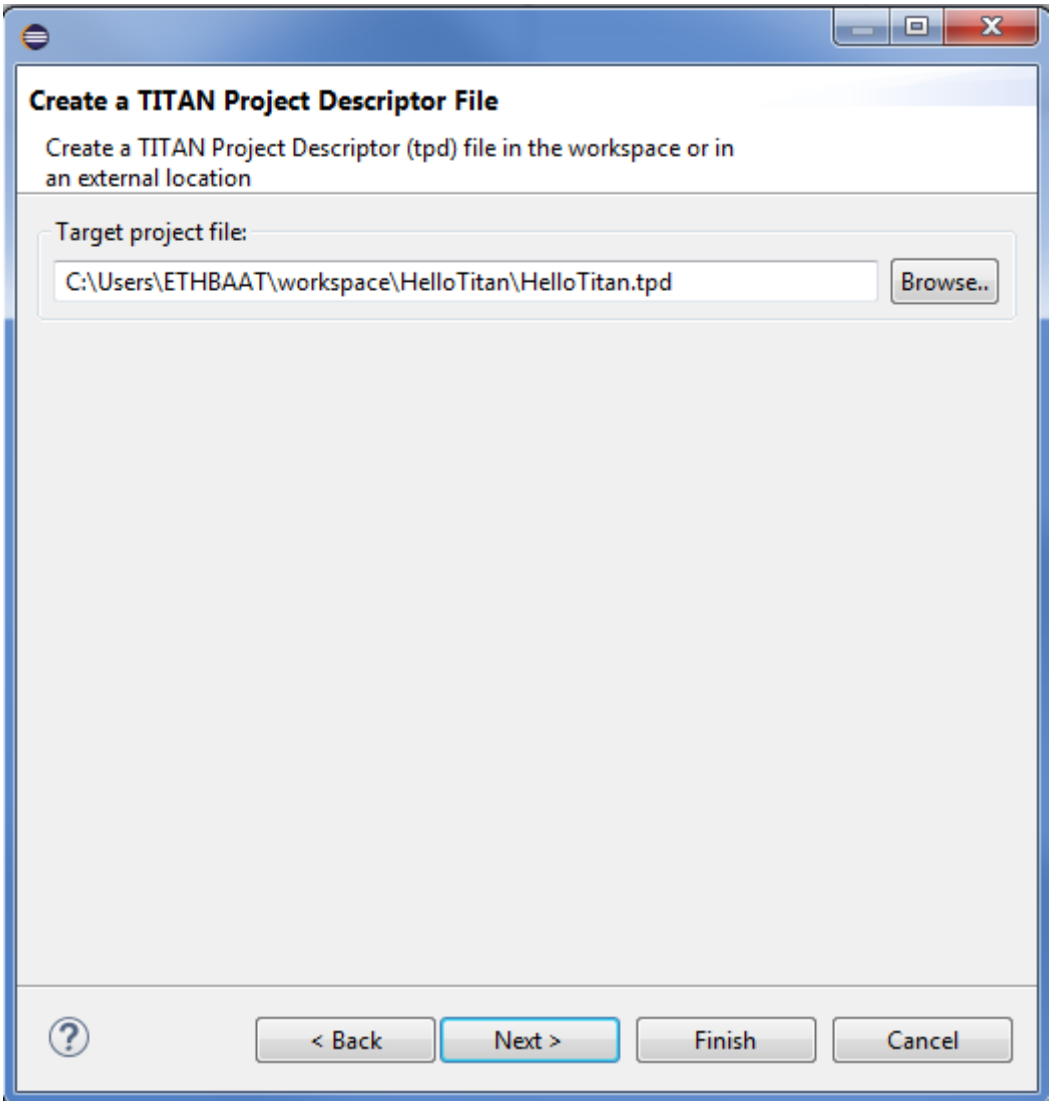


Figure 83. File selection page

4. On the options page fine tune the amount of data to be exported and press **Finish**.

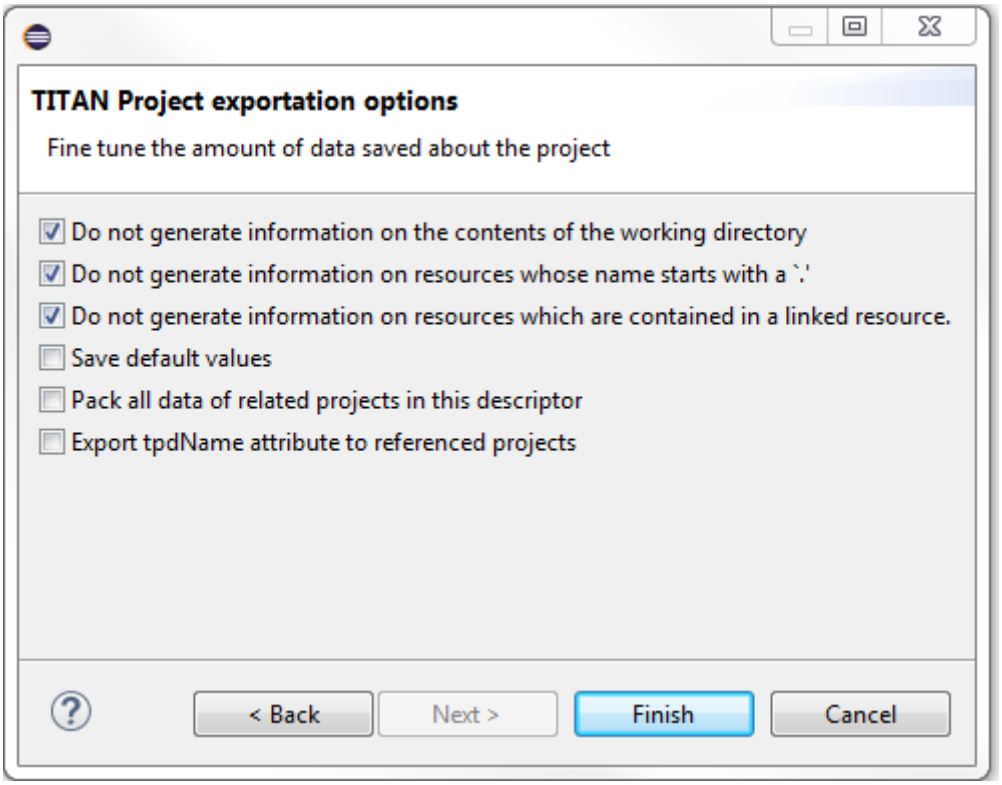


Figure 84. Export options

The available options are:

- **Do not generate information on the contents of the working directory:**

  If the working directory is visible inside Eclipse, inside the project, its contents are by default also mentioned in the project description. As the working directory usually contains only generated files, that can be reproduced later, this behavior is not always desired. Its default value is on.

- **Do not generate information about resources whose name starts with a ".":**

  In Eclipse this naming convention is used to signal that a resource stores some tool specific options about the project. As such, from the point of view of TITAN, they are not needed. Its default value is on.

- **Do not generate information on resources contained within linked resources:**

  In many cases such links are intentionally used to connect to an existing folder whose content might change externally. For example, version handling of files can also be done like that.

NOTE	It is recommended to use this feature with care as there is not much connection between the Eclipse internal resource system, and the file system, the activation of this option can cause unexpected side effects. Its default value is on.

- **Save default values:**

  By default it is not include any information on any option/setting in the descriptor file, which has its default value as the actual one. This makes for a very compact description, but in cases

where all information needs to be saved, this might not be ideal. Its default value is off. If it is switched on, the size of the `tpd` file is unnecessarily big. This is not a problem but perhaps it is not so easy to analyze by the user.

- **Pack all data of related projects:**

  Project references in Eclipse are a great way to structure one's work into manageable pieces. However, if one of those projects is not available, building the whole set is not possible. For this reason, it is possible to save all information from all required projects into one project descriptor. Its default value is off.

- **Export tpdName attribute to referenced projects:**

  If this option is on, then the referenced projects will have a `tpdName` attribute. The value of the `tpdName` attribute by default is the project's name and the `.tpd` suffix. If the referenced project had a `tpdName` attribute during the import, then that value will be stored. By default this option is on, if the project was imported from a `tpd` file using `I` switches.

The default settings can be changed under **Window** / **Preferences** / **TITAN Preferences** / **Export** (see Setting Workbench Preferences).

For more information, related to this file format, please refer to Section 8 of the TITAN Programmer's Technical Reference [4].

Exporting Projects automatically into the TITAN Project Descriptor (tpd) Format

The automatic export of projects can be set on workspace level. The fine tuning of the information can be set. It can be set to ask/request the location of the `tpd` file when the first automatic save happens.

To export your projects automatically, follow the steps below:

1. Select **Window** / **Preferences** / **TITAN Preferences** / **Export**. An option dialog appears (see Figure Export options).
2. Switch on the option "Refresh tpd file automatically".
3. Switch on the option "Request new location for the tpds at the first automatic save" if your projects to be automatically saved have not been saved yet or if you want to change the location of your tpds when importing them.
4. Optionally change the options in the group "Fine tune the amount of data saved about the project" if it is necessary. (It is not suggested.)
5. Press **Apply** or **OK** to save the settings.

4.10.6. Importing Projects from TITAN Project Descriptor Format

To import a project using an existing TITAN project descriptor file follow the steps described below:

1. **Right click** somewhere in **Project Explorer** and select **Import**, as shown on Figure 63.
2. On the **Import** window select **TITAN** / **Project from new project file** and press **Next** (below).

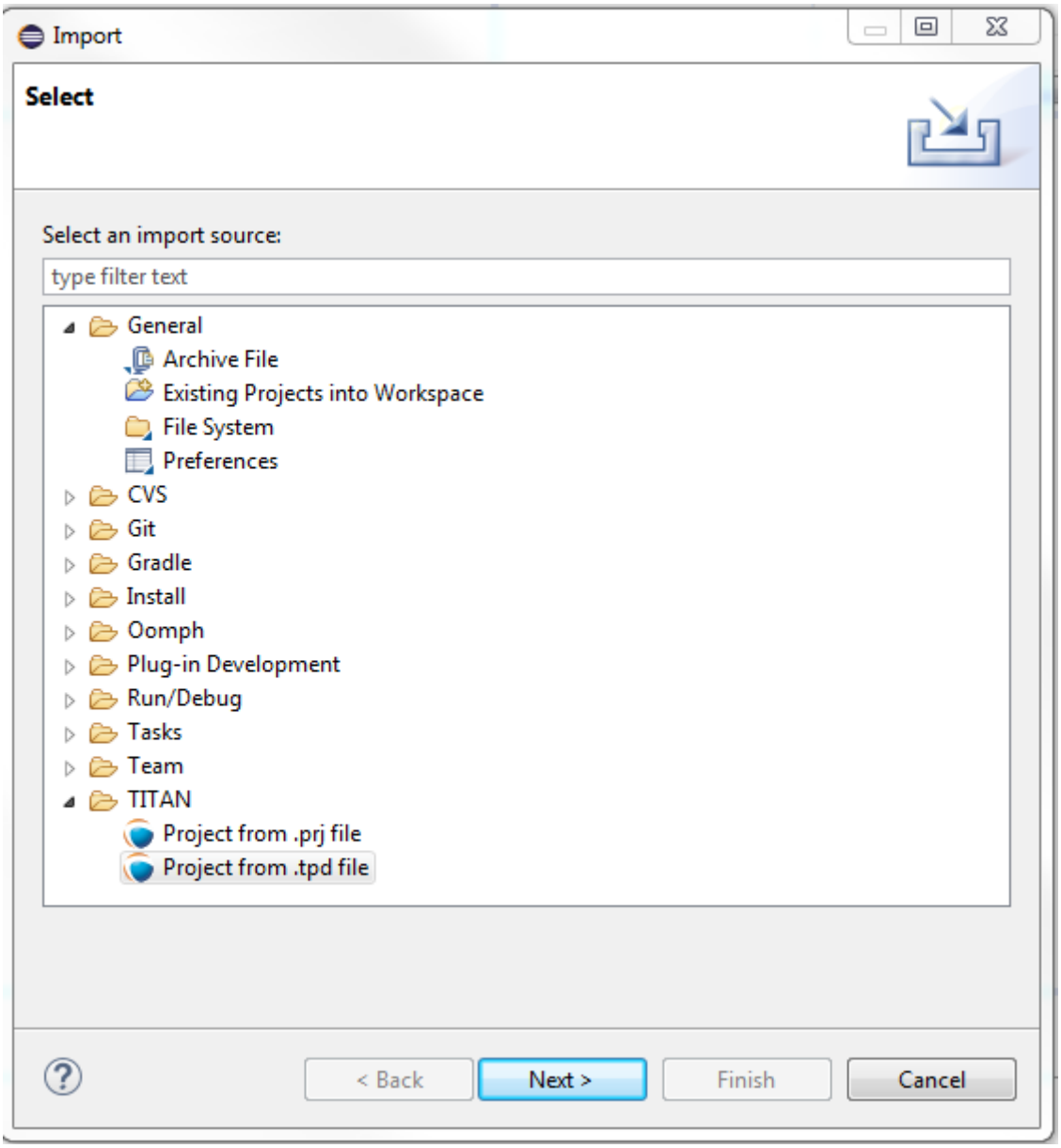


Figure 85. Import from project descriptor

3. On the **Import new TITAN Project from .tpd file** page select the original project file to import from. There is an optional field where search paths can be entered in the format of `-Ipath` where path must be an absolute path. The mechanism of the `-I` flag is described in the Referred project usage with `-I` switch in the TITAN Programmer's Technical Reference [4].
4. Press **Next**.

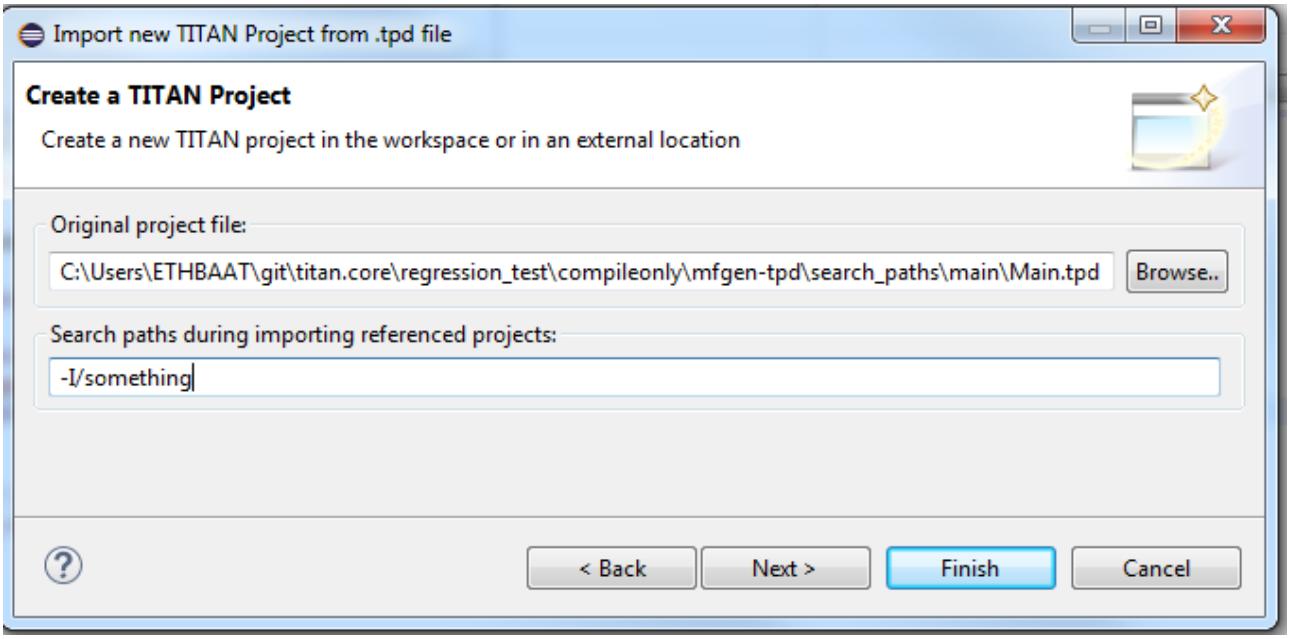


Figure 86. Press Next

5. On the options page select how the importer should behave in certain situations.

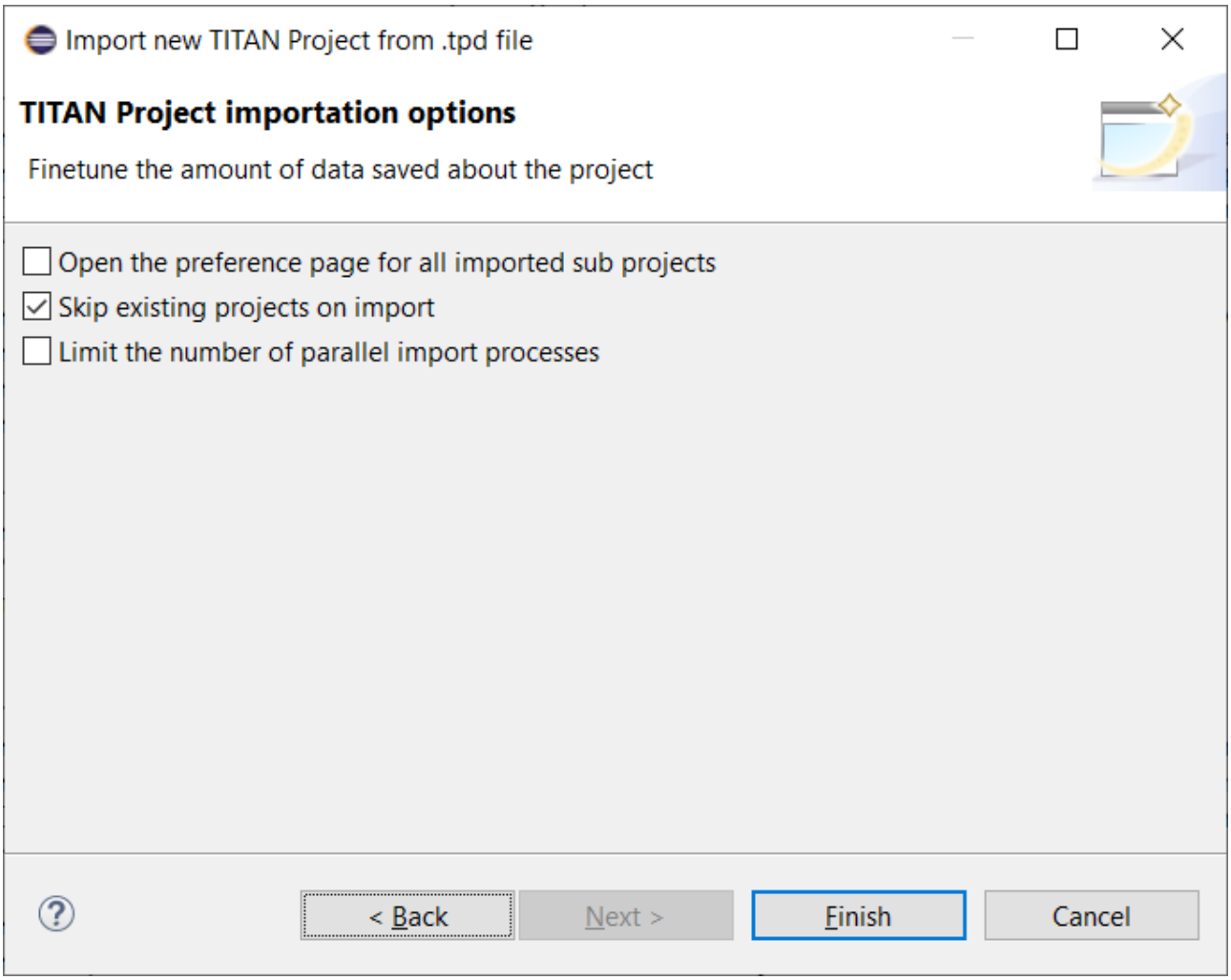


Figure 87. Import options

Available options:

- **Open the preference page for all imported sub projects:** By default the page where the

project preferences can be configured is only displayed for the top level project, referenced projects don't trigger this mechanism. However, if several projects are imported it can be useful to open this page for each of them.

- **Skip existing projects on import:** This is important when a project with a name, which is about to be loaded as a referenced project, already exists in the workbench. By default, there will be no warning, and the importation of that project will not take place.
- **Limit the number of parallel import processes:** During the import procedure several threads are created to utilize the parallelism of modern CPUs and improve performance. However, in some cases the number of created threads exceeds the OS or the user thread/process limit that results in Out Of Memory exception. This option limits the number of parallel processes to the number of native threads provided by the CPU.

4.10.7. Importing Projects from the Command Line

It is possible to invoke the import process of Eclipse from the command line, without Eclipse showing even the splash screen. The project to import must be an Eclipse project, i.e. the project folder has to contain `.project` and `.TITAN_properties` project descriptor files. The main use case of this import method is to bring the project into the specified workspace without modifying the file system. If the project already exists in the selected workspace, i.e. there is a project in the workspace with the same name, the import will be not executed. This way both TITAN C/C++ and TITAN Java projects can be imported into the specified workspace.

An example invocation:

```
eclipse.exe -noSplash -consoleLog -data location_of_workspace -application
org.eclipse.titan.designer.application.ImportProject location_of_project_folder
```

4.10.8. Useful Tips for Exporting and Importing

Pitfalls

During the importation there might be several behaviors which might look strange at first.

When importing a project description containing Eclipse path variables, it is asked for permission from the user to add new variables, or in case the variable exists with a different value, override variables in his system.

However, if the project description does not store, or the user does not add the necessary Eclipse path variable to his own system, this will not be treated as an error by our tool. Instead either the platform, or any other tool trying to access a resource being unavailable, will report this error.

If a project with the same name to be loaded already exists:

- If it is the top level project the user will be asked to change the name.
- If it is not the top level project the default is to silently ignore the import request, as the project is already imported.
- If it is not the top level project and the user asked not to skip existing projects, the name

changing dialog will be displayed. Upon name change all references to the new project will use the new name.

It is worth to mention, that in order to re-import a project from a project descriptor file, it is required to first delete the actual project. It is not supported to overwrite the current contents automatically.

As an example, in the `mctr_gui` the process of closing the user interface and re-opening it while loading the same project, will load the newest version of the project description (and if it is not saved it will also lose all intermediate changes). However, as the closing of Eclipse does not change any state of the imported projects, after re-opening it, the original project with the original settings will be present. In order to load the new settings, the old project has to be explicitly removed from the working environment.

For more information, related to this file format, please refer to Section 8 of the TITAN Programmer's Technical Reference [4].

Native Export and Import

If your projects contain absolute pathnames, the project can be natively exported and then imported only if the places defined with their absolute paths are visible from the new workspace. This is a strong requirement/restriction but it can be satisfied within the same group or working environment. But in that case why should the project be compressed, relocated and uncompressed?

Exporting and Importing Project Information and Projects via TPD Files in Case of Complex Projects

All project information can be stored in TPD files as it is described in the previous subchapters but not all way of working achieves portability. The next method is applicable for projects of any complexity.

Terminology:

Source root folder or root folder is the folder which contains all source files of all projects. For example, for ClearCase titan users it can be /vobs/ttcn/TCC_Releases.

Workspace is the Eclipse workspace. It is a folder containing Eclipse related project information (and generally it can contain even source files).

Source project is a project of our complex project. It is stored in a subfolder of the source root folder. The name of the source project is the name of its containing folder.

General requirements

1. The projects should be handled from bottom to top, precisely string from the projects independent from any others.
2. The Eclipse workspace and the folders containing the project and the source code shall be totally disjoint (they shall not have any common element).

Suppose that the source codes are created and hierarchically stored under the source root folder.

Follow the steps for each project of our complex project.

1. Create an empty project in the workspace with the same name as the source project (see Creating a New TITAN C++ Project).
2. Import the `src` folder of the project as linked resources according to Importing Files as Linked Resources.
3. Fill in project properties according to Setting the Local Build Properties of a TITAN Project.
4. Export project properties into tpd format according to Exporting Projects into the TITAN Project Descriptor (tpd) Format.

NOTE	The target place should be the folder of the original project where the project was imported from.

5. Import the `tpd` file from the source project into the Eclipse project.
6. Export the project into tpd format as in step 4.

NOTE	This way the new `tpd` file will contain the information about itself. It is extremely important if the whole set of project should be exported as a compressed file for example to send to a test lab as a product or to the TITAN support to report a bug.

Exporting Project Content from Command Line Using TPDs

To export the content of whole project sets if each project has a `tpd` file, follow the steps described below. Unix environment is required.

1. Go to the folder of the top level source project.

NOTE	It is located in the source root folder not in the workspace!

2. Use this command from command line:

   ```
   ttcn3_makefilegen -V -P rootdir_to_split -t top_level_tpd.tpd | xargs tar cfz
   my_target_tar.tgz
   ```

   for example:

   ```
   ttcn3_makefilegen -V -P /home/ethbaat/DiameterApplib/Diameter_Applib_2013_03_01 -t
   Libraries/EPTF_Applib_Diameter_CNL113521/EPTF_Applib_Diameter_CNL113521.tpd | xargs
   tar cfz DiamAppLibTest.tar.gz
   ```

NOTE

The compressed file will contain the files in the same structure as they have been stored in the source root directory.

See more information about the command `ttcn3_makefilegen` in Sections 6.1.2 and 6.1.3 in the TITAN Programmer's Technical Reference [4].

4.11. Formatting Log Files

To format a log file (one having `.log` as extension) **right click** the file and select **TITAN** / **Format log**.

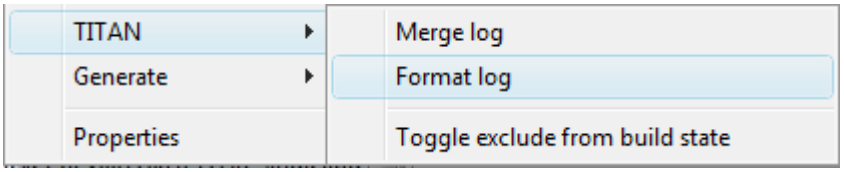


Figure 88. Format log menu

This will produce a formatted log file in the very same directory, with the same name, but having the extension `.formatted_log`.

NOTE

For the duration while the formatted log is being created progress indication is provided in the **Progress view**.

4.12. Merging Log Files

To merge several log files (ones having `.log` as extension) select them, and after **right clicking** on one select **TITAN** / **Merge log**.

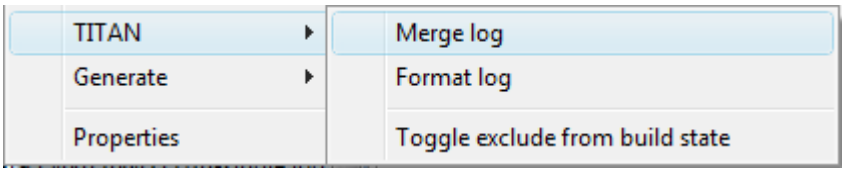


Figure 89. Merge log menu

This will first ask for the file where the results have to be saved, processing the log files will only start after a new or an existing files is selected.

NOTE

For the duration while the formatted log is being created progress indication is provided in the **Progress view**.

4.13. Using Project References

In Eclipse for the creation of a hierarchy of projects building on other projects one can use project references (Figure 81).

When a project references another project, this means for Eclipse that all of the resources of the referenced project are available for use in the referring project. For example if *Project_2* is

referencing *Project_1*:

- All modules available in *Project_1* can be used in *Project_2* too (for importation, code completion ...). For the on-the-fly toolset is will seem as if those modules were also part of Project_2.
- The order in which *Project_1* and *Project_2* are built will always be handled automatically:
- If *Project_1* changes, *Project_2* will be refreshed too.
- If *Project_2* is built *Project_1* will also be built, but only if it has also changed since the last time it was built.
- When *Project_2* is built, it will not attempt to build the modules from *Project_1* again, but rather use their already built form from the working directory of *Project_1*.

NOTE Project reference hierarchies are not limited to two projects they can contain any number of projects.

Project references for one project can be managed in the following way: **right click** the project whose references should be changed and select **Properties** / **Project References.** Adding or removing a reference to a project can be done by simply selecting or unselecting to projects the references should point to.

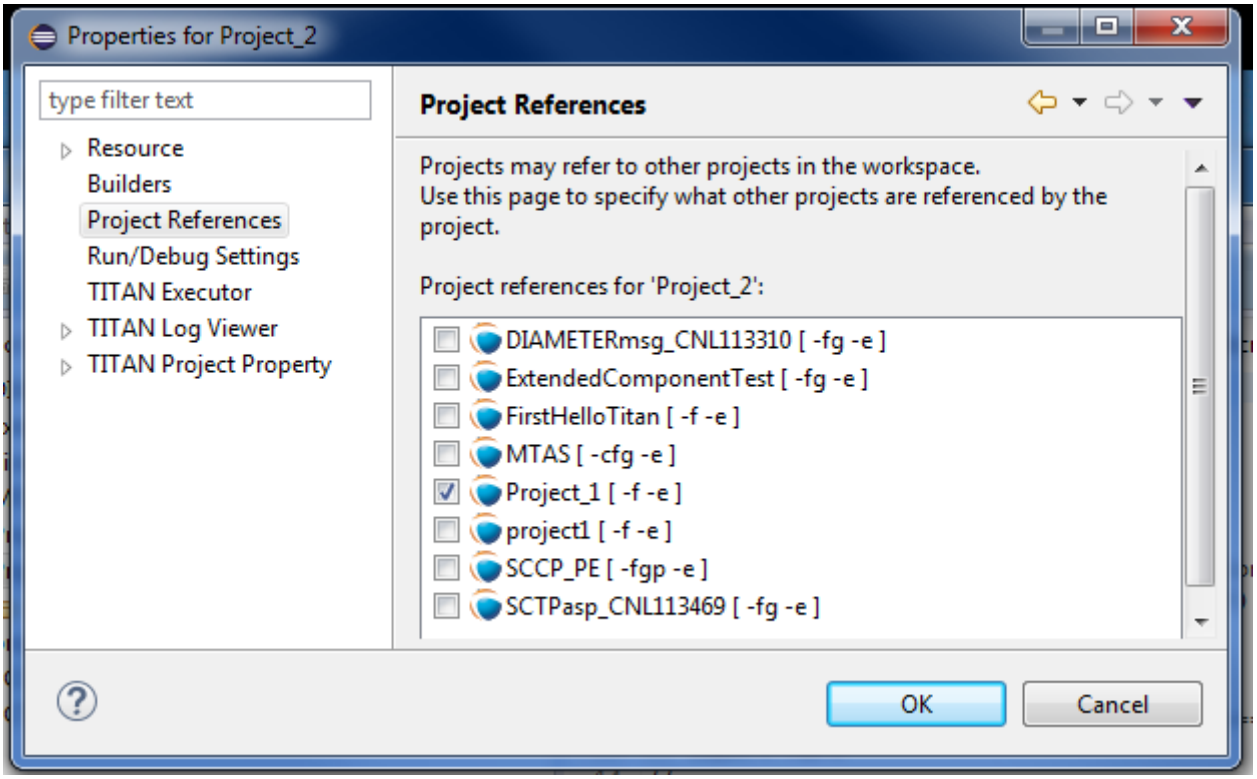


Figure 90. Project references

NOTE These references are operating system and file system independent. This means that it is possible to connect projects coming from different physical locations / version handling systems, as long as each is project is set up to work correctly within its own rules.

4.14. Mapping Elements of the Old Format

The elements of the old GUI can usually be mapped to the new GUI as folders. So, for example, a testports folder should be created in the project, and the files of testports should be placed there. This provides the users with much more configurable project hierarchy, as they can organize their sources as they wish.

Included projects can be generally mapped to simple or linked folders, provided that the central storage property of the folder is set (see Section 4.7). Included projects are fully functioning projects that can be built separately, but are included in the actual project because they provide some useful features. Generally speaking, they are folders (projects are practically stored separately), which might be linked (as they are expected to be on a different computer in the network, if they are just local folders then they can be mapped to local directories) and they have their own `makefile` (because they can be built separately).

NOTE	Linked folders with their central storage property set provide the same features.

Automatic conversion between the old and new format is not a part of the TITAN Designer plug-in for the time being.

4.15. Common Threats

There are some very dangerous operations related to project management in Eclipse.

These are "good to have" features in a general sense, and they also provide more flexibility, but if someone misuses them, then it is sometimes impossible to revert the situation to its original state.

4.15.1. Disabling, Removing or Corrupting the Builder of the Project

This may happen when editing the `.project` file, where Eclipse stores the natures and projects associated to the given project. Any modification of the `.project` file is discouraged.

Repair can be attempted using the functionality **Toggle TITAN project nature**. It can be activated by **right clicking** the project and selecting **TITAN / Toggle TITAN project nature**. As shown on Figure 80, this functionality is used to add the TITAN nature and TITAN builder to (or to remove them from) a given project. Removing is useful if only the builder was removed; the user should then first remove the nature from the project, and thereafter add it back together with the builder.

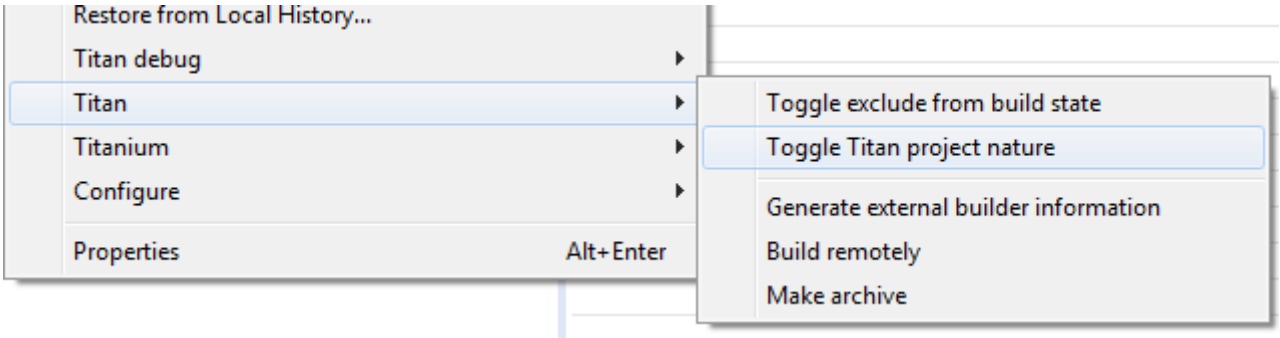


Figure 91. Toggle TITAN project nature

NOTE	the result of this problem (or its repairing) can result in losing every project specific settings. So these settings must be checked after using this functionality.

4.15.2. Removing or Corrupting the Nature of the Project

This problem is almost exactly the same as the one mentioned just above: editing the `.project` file is probably its cause. The possible remedy is also the same.

4.15.3. Adding or Removing Resources from the Project

Modifying project resources in the operating system (outside Eclipse) can temporarily create problems for the users as the project structure they see might not be the actual one.

This problem can be solved easily: **right click** the project and select **Refresh**. Eclipse also does similar operations regularly.

4.16. Make Archive

It can happen that the source code shall be sent to another team member or to the Titan support team to debug.

This can be done

- by exporting the whole project (by **right clicking** on the project, selecting the option **Export...** > **General** > **Archive File**) or
- by executing the command "Make archive" from the Eclipse IDE. It can be executed if the `Makefile` exists in the working directory and a UNIX shell can be executed. **Right click** on the name of the project and select the option **Titan>Make archive**. The command `make archive` will be executed in the working directory and a backup directory will be generated in it. This directory will contain a `tgz` file including the source files, the `Makefile` and optionally the `tpd` file.

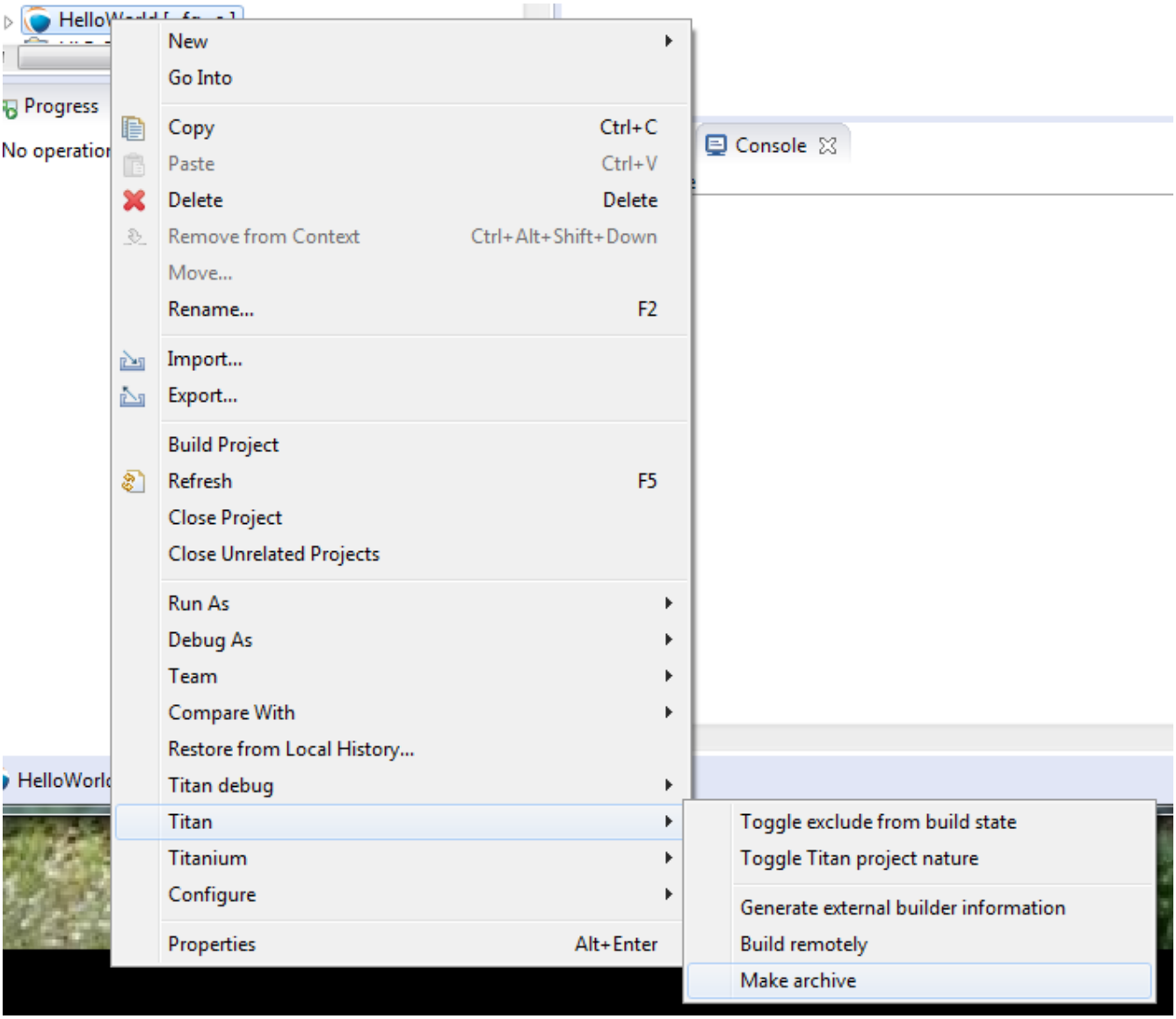


Figure 92. Create Make archive

NOTE

The TITAN Java Projects do not use makefiles to drive their build process, as such the "Make archive" option is also not available. TITAN Java projects can be exported into archive files by **right clicking** on the project, selecting the option **Export...** > **General** > **Archive File** .

[1] The terms "modules" and "files" are used interchangeably in this section.

Chapter 5. Converting Existing Projects

In the TITAN toolset we are supporting 3 different tools/project handling principles at this time: Makefiles, mctr_gui projects and Eclipse projects. Before going into detail on how to convert one of the first two into an Eclipse project, we should review the features offered by these tools to work with projects.

5.1. The Construction Principles of Projects

5.1.1. Makefile

Makefiles support the following ways of working with projects:

- Direct access:

  The files are in the same folder as the Makefile.

- Central storage:

  Some of the files are in a different folder, which also has its own Makefile. The actual Makefile will call the Makefile of this folder, if needed to build the binary files, instead of building them itself. This efficiently reduces build times even for a single user scenario, and can also be used where several users refer to the same already built folder.

- Anything else:

  The Makefile is available for the users to modify, so any kind of project structure can be created. It is also possible to add new commands, new build rules, new behaviors.

5.1.2. Mctr_gui

The mctr_gui supports the following ways of working with projects:

- Referring to files directly anywhere in the file system:

  when the project is built, symbolic links are created for all of these files in the working directory of the project. Practically this maps to the direct access feature of the Makefile mode.

- Referring to file groups:

  file groups recursively declare a list of files and file groups that they represent. When a file group is used, all files and other file groups it references are also automatically used. The files included in a group do not have to be in the same folder or be related any other way. For each file added to the project this way the build system creates symbolic links in the working directory.

- Included projects:

  it is possible to refer to a whole project, instead of referring to files or file groups one-by-one. In

this case at build time the working directories of the included projects are used as central storages for the actual project.In this mode, if something is changed in a project (build mode, additional files) all projects including that one will also see that change, at the next build, as it will go differently.

5.1.3. Eclipse

In Eclipse the fundamental difference to all previous systems is that in this case Eclipse as the platform provides all of the options for structuring the projects. Our IDE only extends the platform with TTCN-3 related features (and doesn't define the whole platform).

On one side this is a limitation, on the other side this means, that anyone can extend his projects with additional capabilities, either by developing his Eclipse extensions (for example a builder that converts some 3rd party file format into TTCN-3 files), or by using existing 3rd party tools (for example CDT for working with C/C++ and Makefiles, JDT for Java, documentation supporting tools, supporting writing command line scripts easier ... and the list goes on).

The following ways of structuring are provided by Eclipse: [2]:

- Each file and folder below the project's folder is part of the project, and by default should be used to operate the project. However, plug-ins working on the project can choose to ignore some of them on their own. For folders this is very much like file groups in mctr_gui, but in this case all files/folders in a given folder are part of that project by default.
- Linked resources can be used to refer to files/folders that are not contained within the folder of the actual project. This way the linked files/folders will also be members of the project in the resource system of Eclipse. It is important to understand that linked resources are only represented in Eclipse as the path they point to. When such a project is moved (or checked out on a different location), the contents of the linked resources are not moved together.
- Linked folders can be marked to be central storages. In this case the contents of the folder are not built with the actual project, but used as central storages.
- Linked files/folders can be set to use Path variables to refer to the target location. Using this method, it is possible to refer to files/folders that are outside the project in the local file system, in a semi-transportable way. In this case the contents of the files are not moved together with the project, but if the receiving user has the same folder structure as the sending one, and has the same path variables set, the linked resources will point to valid locations at his site too.
- Referenced projects:

  the projects in Eclipse can reference any other project inside the same workspace (see Chapter Using Project references). Similar to included projects in the mctr_gui this feature also maps the working directories of the referenced projects as central storages. However there is difference between the two features: If a setting or file is changed in the mctr_gui project, the projects including it will only notice the change when they are being loaded / built the next time. As in Eclipse most of the time all projects are available and interactively worked with, if something changes in a project, all accessible projects referring to that one will automatically (and supposedly instantly) react. For example if a function is removed from the source code in one project, all of its call sites will notice and report the error, even if they are located in different projects. Also the internal Makefile generator is able to make use of settings of the referenced

projects, to make its own job better: for example if a library is set to be used at linking time for a project, all projects referencing that one (either directly or indirectly), will also include that library in the Makefiles they generate.

NOTE	Referenced projects are represented with their name only. As long as there is a project in the workspace with the same name it will be ok to use, without regard to where it might be located, how it is version controlled, or if it really exists or is just emulated by a 3rd party plug-in.

Additional information related to Eclipse:

It is important to note, that using referenced projects is also a good way to manage complex projects, and the possibly large load of build and analysis. In such a hierarchy if something changes the command line build, and the on-the-fly analysis will only reanalyze only those projects, that might be affected by the change, usually only a small part of all of the sources.

As Eclipse defines the base folder of the project as the folder where the ".project" file resides, it comes naturally, that in a single folder we can only have one Eclipse project.

No matter where they are originating from, in the workbench of Eclipse all projects are located on the same level: directly below the root of the workbench. For this reason, creating connections between projects, by any means other than "project references" is not really recommended, as even importing, or joining such a project can create a structure different from the one seen in the native file system of the projects involved.

Referring outside, the project should be discouraged in case of files and folders, as those methods are not always transportable. In those cases, the project might not be transferable as it is not the contents of these references that will be transferred, but the reference itself. In case of referenced projects this is not that much of a problem, as in that case it is natural, that in order to transport a project, we also need to transport all projects that it builds onto. As long as each project can be transferred on its own their referencing sets will be transferable also.

5.2. Manually Converting an Existing Project to Eclipse Format

5.2.1. Small Project

If the project is so small that all of its files are located in one place (in or below one folder) it can be converted easily.

If done from Eclipse one just has to create a new project, setting the location of the project to be linked to the folder where the sources are located in. [3]. This will create the project in Eclipse, and all of the files needed to store the settings of the project (which are set to default values at this time). For more information, please refer to chapter Managing Projects/Creating a New Project.

If it is needed to perform this step from the command line, one needs to place a default ".project" and ".TITAN_properties" file in the base folder of this project.

In the "project" the name of the project has to be set. Eclipse should be able to import the project and all further configurations can be done from there.

5.2.2. Large Project Sets Consisting of Several Included Projects or Logically Separate Parts

This can be easily mapped to referenced projects inside Eclipse. For each separate project or logically separate part there should be one project created, and the proper referring relation between each one should be set. It is recommended to set this attribute in Eclipse, so that all needed modifications are done in the internal representation. For more information, please refer Using Project References.

If we have to do the changes externally the ".project" file has to be extended with the following code:

```
<projects>
<project>included_project_name</project>
</projects>
```

As Eclipse will use the name of the project as reference, this will be a transportable solution, as neither local file system paths, nor the relation between the actual and the referenced project is fixed (with symbolic links we would be forced to build the same project structure which is not possible in Eclipse, as all projects have to be on the same level).

Next figure gives an example on how it might look if 2 large projects are separated into smaller referring projects.

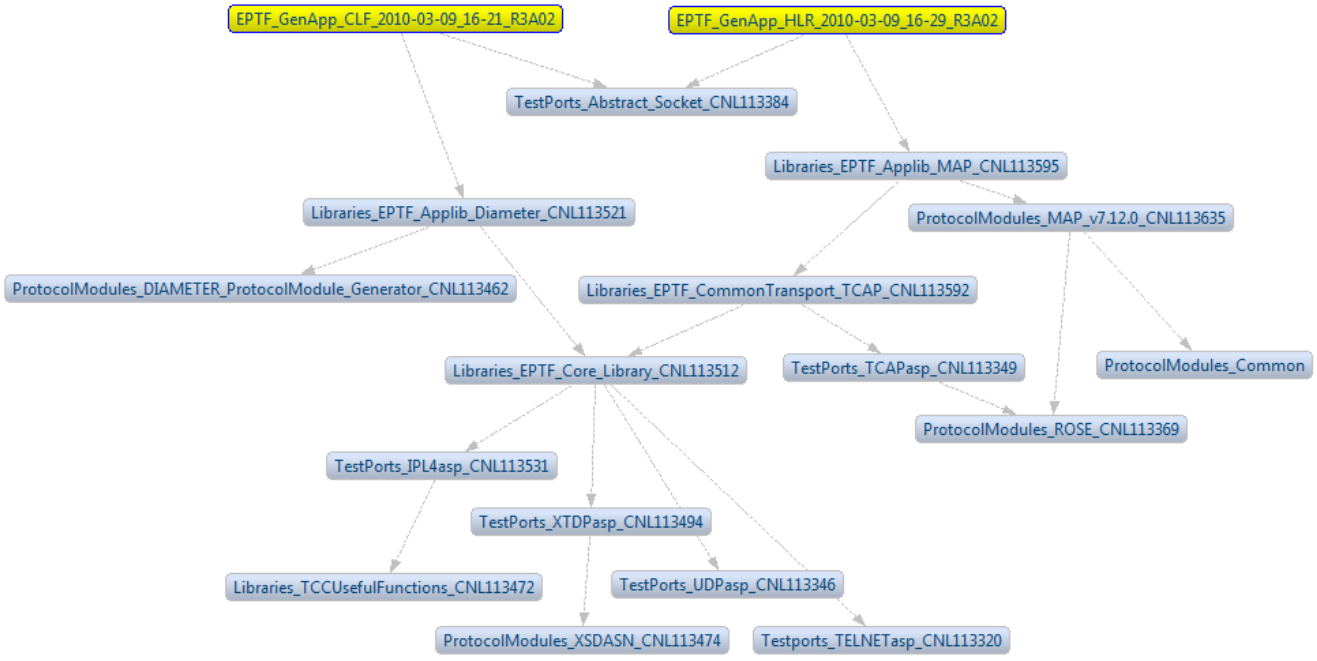


Figure 93. Two large projects

5.2.3. Large Projects Using Central Storage Folders

If the project uses central storage folders there are two good solutions possible:

- If it is possible these cases should be solved by converting the central storage relation into a referencing relation between 2 projects. As such the folder declared to be a central storage should be converted into a project on its own, and the original project should be set to reference this project. For more information, please refer to section 4.6.
- A second solution is to create a folder in the project for each such reference and set it as central storage. It is recommended to do this change from Eclipse by a single right click on the folder. If this has to be done from the command line, the ".TITAN_properties" file's "FolderProperties" section has to be extended with the following code:

```
<FolderResource>
<FolderPath>path_of_the_folder_in_the_project</FolderPath>
 <FolderProperties>
  <CentralStorage>true</CentralStorage>
 </FolderProperties>
</FolderResource>
```

When loading this project the Designer plug-in will know, that that folder is not to be handled as a normal folder, but instead as a central storage. This solution will also let the user/converter chose whether he wishes to have the central storage inside the project, or use Eclipse linked resources to refer to places outside the project no matter whether the folder is inside or outside the project.

NOTE	Even though the second solution sounds to be the better one at first, because of the similar terminology, actually it is not.

Creating referencing relations between projects reflects the logical structure of such folders better, promotes reuse of projects (and so source code) and in the longer run could be used to validate the relations between projects in a hierarchy.

5.2.4. Project Referring to Specific Files Outside its Own Jurisdiction

In some cases, it might have happened, that people did break logical relations and either created symbolic links to files in other projects, or referred to them in the mctr_gui one-by-one specifically.

If it is not possible to map this relation to referring projects or central storages the only solution left is to create a linked resource. This new resource should be placed in the actual project, but setting its location as a link to the original file.

NOTE	It is not recommended to have symbolic links in a project pointing to some other location as those projects are typically not transportable, and also this introduces a hidden dependency between projects, that cannot be validated automatically.

5.3. Convert an Existing mctr_gui Project Using an Import Wizard

The Designer feature comes with an import wizard, which is able to create an Eclipse project out of an existing mctr_gui project automatically. For information on how to find this wizard, and what its steps are please refer here.

As this wizard has no knowledge about the internal semantic structure of the project to be loaded (the mctr_gui did not helped the organization of project parts too well), the conversion is rather simple:

Projects mentioned as included project in the input project file will be converted to references to Eclipse project.

File referred to directly will be linked in the base folder of the newly created project, with Eclipse links.

Group files are read, but as such an automated wizard is not allowed to create arbitrary folder structures, the files in each group will be linked to the base folder of the project, just like directly referenced files.

In the last two cases if the location of the project directly contains any of the files to be imported, instead of creating Eclipse links, the original files will be used.

Although it might be possible to work with the project created, it is recommended to fine tune it by hand afterwards (or for large projects do the conversion by hand to start with). As the generated out is known to have serious flows: not structured, not easy to version handle and contains links to all files … even if it would be possible to create a project hierarchy using existing projects.

[2] There is one more dimension of structuring in Eclipse when several plug-ins are used on the same project /_by default all plug-ins are active on all projects _/.If there are several plug-ins active in/on a given project, this can create several ``layers'' of responsibilities. This is an important feature, as this makes it possible to mix plug-ins that each provide some separate functionality into a working environment that best supports the user's daily work routine. For example on a parallel cooperation the Designer supports editing TTCN-3, ASN.1 and configuration files, while CDT support editing C/C++ and Makefiles practically covering all aspects of working with TITAN by default. For an example of sequential cooperation we can say, that the working directory we use to output the final product of TITAN (the executable test system), can be viewed by CDT as the source of information for debugging/profiling the generated executable.

[3] In case the original project has some kind of structure like src, doc folders the new project should also be created in this base directory instead of using the src folder directly.

Chapter 6. Building the Project

In this chapter a detailed, step-by-step procedure description is provided about how to build a project according to the workflow.

Building a project from the TTCN–3 or ASN.1 source modules and perhaps test port files is a procedure consisting of several steps. In the TITAN Designer plugin, the procedure is fully automated. The commands issued by the build related functionalities and their progress messages are displayed in the TITAN console, so the successful completion of the processes can easily be verified. Also, in case of an error, the analysis of the progress messages helps to find the cause of the problem (this is also automated to some extent; please refer here). The build process also provides Eclipse with user friendly information about its progress.

The building process is automated; that is, the executable is updated in the background when project resources change (because they have been created, deleted or updated). There is no need for user interaction—provided that automatic building is enabled.

There is a way to build the project manually, by selecting **Project** / **Build project** or **Project** / **Build all.** This is useful when automatic building (**Project** / **Build Automatically**) is disabled**.**

NOTE	The problem markers of the compiler are parsed from the output of TITAN, for this reason they are updated when the compiler is run (the project is built, or the files are checked). If automatic building is not used, the projects should be built regularly, to have up-to-date problem markers (see here).

6.1. Building the TITAN C++ Project

6.1.1. Step by Step

The following sections describe the steps of the build process. These steps are carried out either automatically by the TITAN plugin or manually by the user; the sections indicate which way applies.

Creating Symbolic Links

By default, the first step of the build process is creating or updating symbolic links in the working directory of the project. The working directory contains symbolic links pointing to every file included in the project (this is not true for files contained in a central storage directory, because they are handled differently). For information please see the TITAN Programmer's Technical Reference [4].

Symbolic link creation is done automatically by the build process; no user action is required.

NOTE	The creation of symbolic links can be turned off in the Designer plug-in, for more information please refer here.

Creating or Regenerating the Makefile

The second step of the build process, if needed, is creating or updating the project `Makefile`. Automatic `Makefile` management should be enabled on the **Properties / TITAN Project Property** page of the projects.

Every time it is required, the `Makefile` generator of TITAN will be called with the parameters provided on the **Makefile creation attributes** tab (see here). It is possible to indicate a `Makefile` updater script on the **Make attributes** tab (see here) that will be run on the generated `Makefile`.

Information about the flags of the TITAN `Makefile` generator and the `Makefile` updater script can be found in the TITAN Programmer's Technical Reference [4].

It is the user's responsibility to create and update the `Makefile` when automatic `Makefile` management is disabled.

Editing the Makefile Skeleton

If the generated `Makefile` is not suitable then either the options that direct its generation should be changed or (after having disabled automatic building) the `Makefile` should be created by hand. Everyone is allowed to write his own `Makefile`; however, the `Makefile` skeleton generated by the compiler always serves as a good starting point. For an extensive description of what shall be checked in the generated `Makefile`, see the TITAN User Guide [3].

The TITAN plug-in has knowledge about the following `Makefile` commands:

- `make`
- `make all`
- `make dep`
- `make check`
- `make clean`

NOTE	the TITAN plug-in has some assumptions on what functionality the `Makefile` offers. The real `Makefile` should support these functions, and they should be conforming to what behavior TITAN would create.

This step, if needed, is carried out manually by the user.

Module Compilation

In this step C++ files are generated from TTCN-3 and ASN.1 files. When a C++ file already exists, then the timestamp of the Compile file is used to decide whether a C++ file in question is up-to-date or not. A C++ file is refreshed only if the corresponding TTCN–3 or ASN.1 module was modified later than the timestamp in the Compile file indicates, or the project was refreshed by **right clicking** the project and selecting **Refresh**; otherwise the generated C++ file is considered up-to-date.

The first compilation of the modules will result in addition of the following files in the working directory:

- C/C++ header files:

  These are the header files of the generated C++ code. One `.hh` file is generated for every TTCN–3 and ASN.1 module in the project with the same name.

- C/C++ source files:

  These are the body files of the generated C++ code. One `.cc` file is generated for every TTCN–3 and ASN.1 module in the project with the same name.

- Compile file:

  This is an empty file. The attributes of the file indicate the date and the time of the last compilation process.

- `Makefile.bak`:

  This is the backup of the `Makefile`, created when the `make dep` command has been issued.

Module compilation is done automatically by the build process; no user action is required.

Creating Dependencies

Once the symbolic links have been created and the `Makefile` of the project has been properly edited if necessary, the command make dep has to be issued to find the dependencies between the resulting C++ codes. It is extremely important that the dependencies are always uptodate. If, for example, a TTCN–3 module is removed from the project, the dependencies between the C++ files must be updated, otherwise the command `make` fails.

Dependencies appear at the end of the `Makefile` as dependency lines. They are determining the conditions of the binary object code recompilation launched by the command make.

It is discouraged to edit the appended dependency lines.

```
135 PCOType.o: PCOType.hh MyExample.hh
136 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/TTCN3.hh
137 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/version.h
138 PCOType.o: /usr/include/string.h /usr/include/iso/string_iso.h
139 PCOType.o: /usr/include/sys/feature_tests.h /usr/include/sys/isa_defs.h
140 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/Basetype.hh
141 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/Types.h
142 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/Error.hh
143 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/Textbuf.hh
144 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/Encdec.hh
145 PCOType.o: /usr/include/stdio.h /usr/include/iso/stdio_iso.h
146 PCOType.o: /usr/include/sys/va_list.h /usr/include/stdio_tag.h
147 PCOType.o: /usr/include/stdio_impl.h
148 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/RAW.hh
149 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/Template.hh
150 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/Integer.hh
151 PCOType.o: /mnt/TTCN_GUI/demo/TTCNv3-cvs/include/Optional.hh
```

Figure 94. Dependencies

The dependency update is done automatically if the build level mentioned here is set to three or five. Otherwise it must be carried out manually.

Alternatively, incremental generation of dependency information is available when using Makefiles

written for GNU `make`. Instead of modifying the `Makefile`, dependency information is written into separate files with `.d` extension (one for each `.cc` file). These files are included into the main `Makefile`. This has the advantage that the `Makefile` is not modified every time a dependency changes. Another benefit is that the dependencies are always updated during `make`; there is no need to explicitly run `make dep`. For information on how to set this option please refer here.

Building

In the final step of the project building procedure a conventional C++ compiler is used to compile Test port codes and the generated C++ source code to a binary object code. The resulting code is linked with the Base Library. The Base Library contains important supplementary function libraries used for the execution of the generated code (for example verdict handling, Host Controller code, and so on).

If automatic building is enabled, Eclipse will invoke the build process whenever project resources change (are created, deleted or updated), or you refresh your project by **right clicking** the project and selecting **Refresh**.

If automatic building (**Project** / **Build Automatically**) is disabled, then the build process is started by a click on **Project** / **Build project, Project** / **Build all** or by **right clicking** the project name and selecting **Build**.

The build process will result in the generation of the following files in the working directory:

- Object files:

  For every C++ file in the project (source code files, test ports, and so on), an object file (with the extension `.o`) will be created by the C++ compiler.

- Shared object files (if dynamic linking is enabled, see here):

  For every (static) object file (with extension `.o`) in the project a shared object file (with the extension `.so`) will be created by the C++ compiler.

- Executable:

  The executable file has the same name as the project has.

The build process can be configured to set the build level for the given project (see here). The following build levels are supported:

- Level 0 – Semantic Check

  Only syntactic and semantic checks are carried out on the TTCN-3 and ASN.1 source files. Uses the Makefile target **check**.

- Level 1 – TTCN3 → C++ compilation

  In addition to the syntactic and semantic checks, the C++ code is also generated from the TTCN-3 and ASN.1 source files if there were no errors found. Uses the `Makefile` target **compile**.

- Level 2 – Creating object files

  Executes the syntactic and semantic checks, generates the C++ code and tries to compile it into object (`.o`) and if applicable, into shared object (`.so`) files. Uses the `Makefile` target **objects** or **shared_objects**.

- Level 2.5 – Creating object files with heuristic dependency update

  Executes the syntactic and semantic checks and generates the C++ code, but before generating the object and if applicable, shared object files it also updates the dependencies of the source codes if this is needed. This means that the long lasting dependency refresh will not be executed if only such files that the on-the-fly analyzer is able to analyze were changed since the last build, and none of the changes made make a dependency refresh mandatory. Uses the `Makefile` targets **objects** or **shared_objects**; or **dep objects** or **dep shared_objects**.

- Level 3 – Creating object files with dependency update

  Executes the syntactic and semantic checks and generates the C++ code, but before generating the object and if applicable, shared object files it also always updates the dependencies of the source codes. Uses the `Makefile` targets **dep objects** or **dep shared_objects**.

- Level 4 – Creating Executable Test Suite

  Carries out a full build and creates the Executable Test Suite, but the dependencies are not updated. Uses the `Makefile` target **all**.

- Level 4.5 – Creating Executable Test Suite with heuristic dependency update

  Carries out a full build, creates the Executable Test Suite and the dependencies are also updated if that is needed. This means that the long lasting dependency refresh will not be executed if only such files that the on-the-fly analyzer is able to analyze were changed since the last build, and none of the changes made make a dependency refresh mandatory. Uses the `Makefile` target **all** or **dep all**.

- Level 5 – Creating Executable Test Suite with dependency update

  Carries out a full build, creates the Executable Test Suite and the dependencies are also always updated. Uses the `Makefile` target **dep all**.

Some hints for selecting the appropriate build level: on build levels 0-3 the executable will not be generated, only levels 4 and 5 produce an Executable Test Suite. Dependency update is only required when the import hierarchy of the source files is changed.

6.1.2. Remote Build

Projects might need to be built for several platforms, for several different GCC versions, or it might just happen that the user's computer is not powerful enough to assure short build times.

Building remotely is chosen by **right clicking** the project and selecting **Titan / Build remotely**, as shown on Figure 80 above.

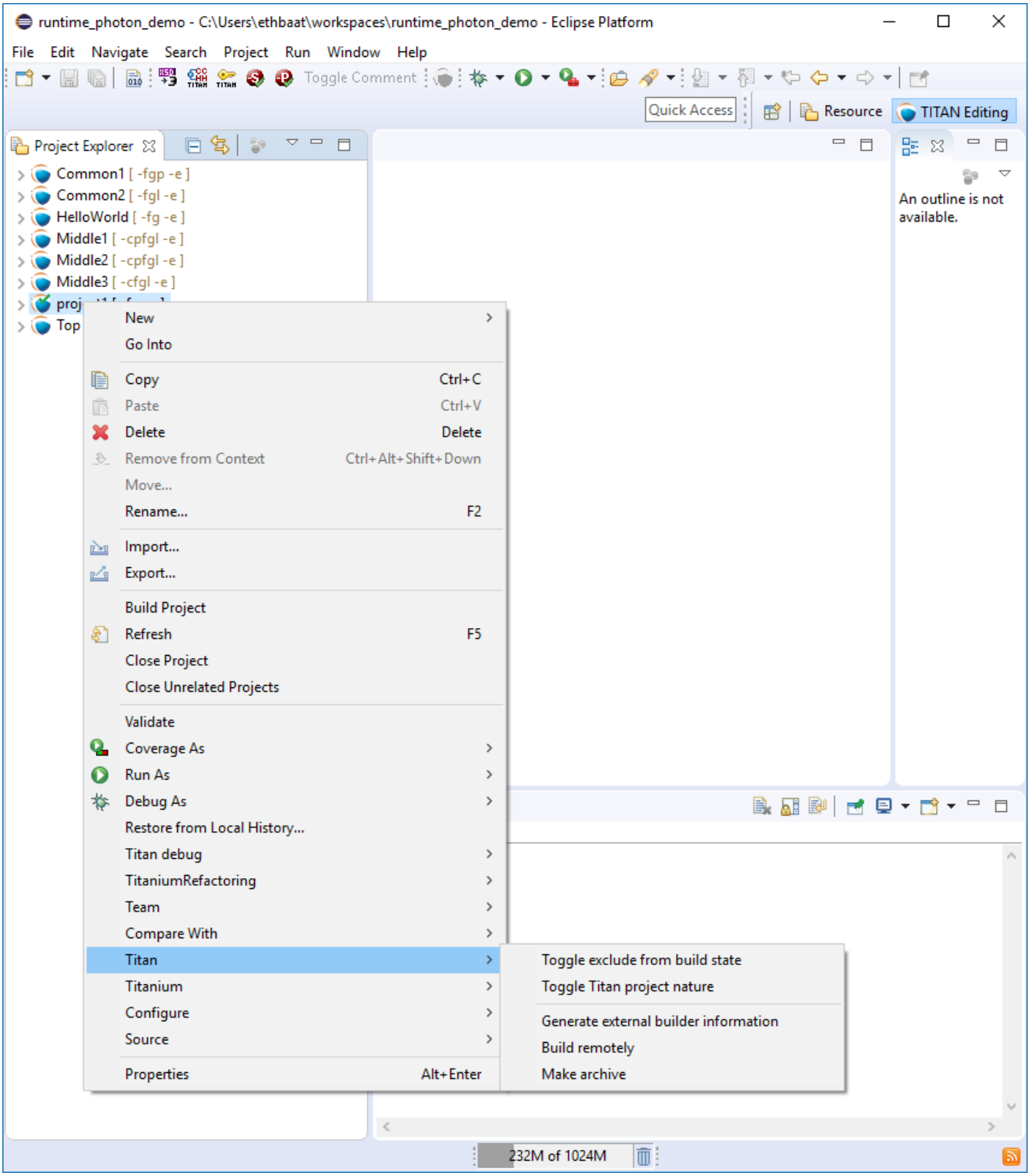


Figure 95. Build remotely

The outputs of the remote build processes are displayed in the TITAN Console view. Every piece of such an output is prefixed by the host name that provided it.

Remarks and Tips

It is impossible to clearly identify which source files were some errors reported for, for this reason precise build problems reported by remote build hosts are not redirected to the graphical interface. Only those problems are reported and marked, which are the errors in the build process itself (for example: abnormal termination is reported, but as a build process is not terminated by build errors, such errors are not redirected).

As it is the user's responsibility to keep the files on the remote host uptodate, no file transfer or file synchronization is provided by the TITAN plugin. Therefore, the remote build process cannot be run automatically.

Building remotely might start up the shell of the remote host in interactive mode. If the remote build host reports missing environmental variables, it is a good idea to check how the shell of the remote build host is configured in interactive mode (this is usually user specified).

The overall length of the name and build commands of the remote hosts should be less than about 2,000 characters. However, assuming that an automated login mechanism and a build script is used on the remote hosts (creating remote build commands like `rlogin rhea; buildscript.sh`), means that the build process might still be executed in parallel on about 60 remote hosts, which should be enough for now.

6.1.3. Building from the Command Line

Building Directly

It is possible to invoke the build process of Eclipse from the command line, without Eclipse showing even the splash screen.

An example invocation:

```
eclipse.exe -noSplash -consoleLog -data location_of_workspace -application
org.eclipse.titan.designer.application.InvokeBuild project_name_to_build
```

This command instructs Eclipse to call our application with the name of the project to be built, while not displaying even the splash screen, redirecting all error log to the console too and using the workspace from the provided location. The project must be available in the specified workspace. If it is not, then it could be imported with a different command line application (see Importing Projects from the Command Line). This way both TITAN C/C++ and TITAN Java projects can be built.

The benefit of using this feature over generating the Makefile and building by hand is that this way one will build with the exact same settings he uses inside Eclipse. If for example 3rd party tools are also used as part of the build process, this method will invoke them too properly.

Furthermore, TITAN Java projects can be automatically exported as runnable JAR files at the end of the build process:

```
eclipse.exe -noSplash -consoleLog -data location_of_workspace -application
org.eclipse.titan.designer.application.InvokeBuild project_name_to_build -jar
path_to_jar
```

Building with an External Script

It is possible to create an XML file for each Eclipse project, which will store all the information needed to create the Makefile and build the project from the command line.

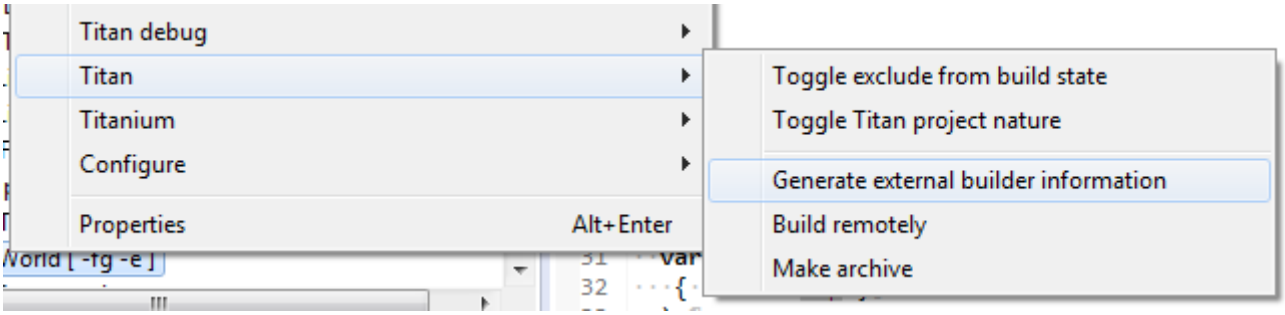


Figure 96. Generate external builder information

In order to create this file, right click on a project and select the **TITAN** / **Generate external builder information** menu entry. This will create a new file in the root of the project called **external_builder_information.xml**

The XSD schema definition of this file looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
id="TITAN_External_Builder_Information">
  <xs:element name="TITAN_External_Builder_Information">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Makefile_settings">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="useAbsolutePath" type="xs:boolean"/>
              <xs:element name="GNUMake" type="xs:boolean"/>
              <xs:element name="incrementalDependencyRefresh" type="xs:boolean"/>
              <xs:element name="dynamicLinking" type="xs:boolean"/>
              <xs:element name="singleMode" type="xs:boolean"/>
              <xs:element name="codeSplitting">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:pattern value="none|type"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
              <xs:element name="projectName" type="xs:string"/>
              <xs:element name="projectRoot" type="xs:anyURI"/>
              <xs:element name="workingDirectory" type="xs:anyURI"/>
              <xs:element name="targetExecutable" type="xs:anyURI"/>
              <xs:element name="MakefileScript" type="xs:anyURI"/>
              <xs:element name="MakefileFlags" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="ReferencedProjects">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" minOccurs="0"
```

```
name="ReferencedProject">
                <xs:complexType>
                  <xs:attribute name="location" type="xs:anyURI" use="required"/>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                  <xs:attribute name="cygwinPath" type="xs:anyURI"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Files">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" minOccurs="0" name="File">
                <xs:complexType>
                  <xs:attribute name="path" type="xs:anyURI" use="required"/>
                  <xs:attribute name="relativePath" type="xs:anyURI" use="required"/>
                  <xs:attribute name="centralStorage" type="xs:boolean"/>
                  <xs:attribute name="fromProject" type="xs:string"/>
                  <xs:attribute name="cygwinPath" type="xs:anyURI"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="version" type="xs:decimal"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

NOTE	After this information was generated it is the user's responsibility to create and use the script files that actually do the building of the project.
NOTE	This file will only hold information relevant from the point of view of TITAN. If other tools are also integrated on the project (to help its build, execution) their data will not be included.

6.1.4. Cleaning the TITAN Project

After switching to a newer version of the test executor or simply to save disk space, the project might need to be cleaned by removing the generated files from the working directory.

To remove all generated files from the project, select **Clean** in the **Project** menu option in Eclipse.

The following files will be deleted from the working directory:

- All object files (files with suffix `.o`) and if applicable, all TITAN generated shared object files (files with suffix `.so`)

- All C++ sources files translated from the original TTCN–3 and or ASN.1 modules
- The Compile file
- The executable file

6.1.5. Pitfalls

Every build related action is executed as a command line command. If the command line is not responsive, the tool will not be able to extract messages from it.

In the `Makefile` generation process the size of the longest allowed command can become a serious limitation. For example, on Windows 2000 this number is around 2048 characters by default; this is not enough for larger projects. However, as every command that we try to execute, this is also displayed in the TITAN Console, making it is possible to copy and paste it into a proper command line window (in this case into a Cygwin console).

Manually editing of the `Makefile` can kick off a vicious build cycle if automatic `Makefile` generation is enabled. Explanation: saving a file is a resource change and can start the build process. On the other hand, the build process, with automatic `Makefile` generation enabled, might re-create the `Makefile`. Next, the editor detects that the `Makefile` has been changed and tries to open it which is also a resource changing operation and triggers the build process.

6.2. Building the TITAN Java Project

6.2.1. Step by Step

The following sections describe the steps of the build process. These steps are carried out either automatically by the TITAN plugin or manually by the user; the sections indicate which way applies.

Module Compilation

In this step Java source files are generated from TTCN-3 and ASN.1 files. When a Java file already exists, its content is re-checked to decide whether the Java file in question is up-to-date or not. A Java file is refreshed in the current build only if the generated code from the corresponding TTCN–3 or ASN.1 module would be different; otherwise the generated Java file is considered up-to-date.

The first compilation of the modules will result in addition of the following files in the java_src directory:

- Java files:

  These are the Java files of the generated Java code. One `.java` file is generated for every TTCN–3 and ASN.1 module in the project with the same name.

Module compilation is done automatically by the build process; no user action is required.

NOTE	The Java files will be located in a Java package that is created using the project's name. This process will also create some subdirectories inside the java_src folder. For more information please refer to section 5.1 of the TITAN Programmer's Technical Reference guide [12].

Building

In the final step of the project's building procedure, Eclipse's built-in Java compilation feature is used to compile Test port codes, external functions and the generated Java source codes to executable Java format (`.class` files).

If automatic building is enabled, Eclipse will invoke the build process whenever project resources change (are created, deleted or updated), or you refresh your project by **right clicking** the project and selecting **Refresh**.

If automatic building (**Project** / **Build Automatically**) is disabled, then the build process is started by a click on **Project** / **Build project**, **Project** / **Build all** or by **right clicking** the project name and selecting **Build**.

The built in Java compiler infrastructure of Eclipse takes the generated code (from the java_src folder), the test ports and external functions (preferably from the user_provided folder) and compiles them into `.class` files generated into the java_bin folder.

6.2.2. Cleaning the TITAN Java Project

After switching to a newer version of the test executor or simply to save disk space, the project might need to be cleaned by removing the generated files.

To remove all generated files from the project, select **Clean** in the **Project** menu option in Eclipse.

This action will delete all files from the java_src and java_bin folders.

Chapter 7. Editing with TITAN Designer Plugin

This chapter presents the editors provided by TITAN Designer plug-in and their features.

7.1. File Types

The TITAN Designer plugin includes editors for the following file types supported by the TITAN executor (the default extensions are given in brackets):

- ASN.1 (`.asn`, `.asn1`)
- TTCN-3 (`.ttcn`, `.ttcn3`)
- TTCN-3 preprocessable (`.ttcnpp`)
- TTCN-3 includable (`.ttcnin`)
- Configuration (`.cfg`)

Additional file extensions can be associated to these editors by selecting the Eclipse menu point **Windows** / **Preferences** / **General** / **Editors** / **File Associations**, but this is discouraged for source files. Although the file will be well colored, the dynamic analysis will not work. For the same reason it is also discouraged to use the extensions listed above for other file types.

NOTE	The editors may throw an exception if a file is deleted while being edited.

7.2. Syntax Highlighting

Each of the included editors has its own syntax highlighting schema that can be customized by modifying the workbench preferences (see here)

```
template ContentDisposition ContentDisposition_s1 := {
  fieldName := CONTENT_DISPOSITION_E,
  dispositionType := "session",
  dispositionParams := omit
};
```

Figure 97. Syntax coloring of TTCN-3 files

7.3. Matching Brackets

Bracket matching in source code provides structural information to the user. The functionality is activated by placing the mouse cursor after an opening or closing bracket. The figure below shows the open bracket in the end of the first line and its closing pair in the last line.

```
type MyRecord MyRecordSub1 (
  { f1 := omit, f2 := "user", f3 := "password" },
  { f1 := 1, f2 := "User", f3 := "Password" }
)
```

Figure 98. Matching brackets highlight in TTCN-3 files

The function can be customized on the workbench preferences; see here.

7.4. Folding

Folding is another mechanism to provide structural information to the user. Folding decreases the amount of information displayed on screen, thus, users only sees that part of the code they are working with. Text regions can be folded and unfolded with a single click on the folding marker on the left-side ruler.

The figure below shows a possible folding but the text is not folded.

NOTE	Folding marker on the left-side ruler at line 47.

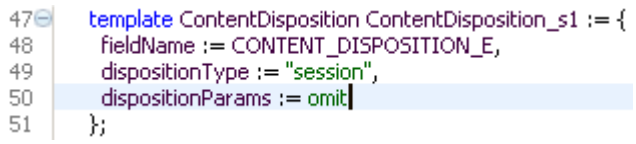


Figure 99. Template not folded

The next figure shows the folding range, the smallest folding region the selected line belongs to. The folding range is displayed when the mouse pointer is placed over the left-side ruler.

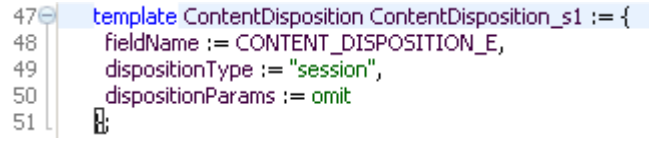


Figure 100. Folding range shown

The following figure shows the text folded. Regions of text can be folded and unfolded with a single click on the folding marker on the left-side ruler.

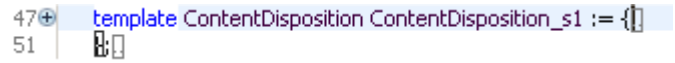


Figure 101. Template folded

The function can be customized by modifying the Folding preferences on the workbench (see here).

7.5. On-the-fly Parsing

On-the-fly parsing means that the text is automatically parsed and checked as it is changing.

The parser starts one second after the last character is typed. (This duration should be long enough for the parser to operate without disturbing the user.) The problems found by this parser are automatically and instantly indicated to the user, allowing a fast and precise feedback on errors and reducing the detection time to almost zero.

This figure shows an example error marker. The user was about to type the keyword `template`, but as soon as he has typed `tem` the on-the-fly parser noticed that the file was syntactically faulty.

Before parsing, the error markers created by the on-the-fly parser are removed. As parsing proceeds, new markers are appearing ensuring that the markers are always up-to-date (except for

the markers of the compiler as they are updated by the compiler itself, see here).

The three steps of the parsing process:

1. Every file in the given project is checked whether it needs to be parsed or not. A file needs to be parsed if at least one of the following is true:
   a. There is no information stored related to its content or the information extracted from the file could not be stored in the data storages (for example, two or more modules exist with the same name in the project).
   b. The file has changed since the stored information was extracted last time.
   c. The execution of the TITAN compiler removed syntax error markers reported by the on-the-fly parser.
2. The file is parsed.
3. The on-the-fly data storage is updated.

The parsing process (like every other long running operation in the plugin) provides progress indication. Overall parsing of a file is usually very fast; however, the duration of the on-the-fly parsing is adversely influenced by the size of the actually edited file. The size of the project does not matter except for the first parsing of a project, when every file needs to be analyzed once. However, if several files need to be parsed, our algorithm will try to do this in parallel, where the level of parallelism is only limited by the amount of hardware support available (for example a computer with 2 or 4 processor cores, will finish this task about 2, 4 times faster in the optimal case).

If too slow, the parsing can be turned off on the TITAN preferences page (see here). Disabling the parsing does not destruct the data stored in the memory; rather, the data cannot be updated while this option is set. If parsing is enabled again, the parser will try to update outdated data.

Parsing of files can be slow in the following cases:

- Files containing more than twenty thousand lines of statements.
- Files containing more than fifty thousand lines of definitions only.
- Any time if the virtual machine does the garbage collection while parsing.

The on-the-fly parser is able to parse ASN.1, TTCN-3 and runtime configuration files.

7.5.1. Preprocessing of ttcnpp and ttcnin Files

In Titan it is supported to use the C pre-processor for creating TTCN-3 files. For this 2 file extensions are defined: the files with ttcnpp extension are to be preprocessed into TTCN-3 modules, while the ones with ttcnin extension hold code snippets that can be included into ttcnpp and ttcnin files. For detailed information please see the TITAN Programmer's Guide.

The designer plug-in provides support for a subset of the features of the C preprocessor. The supported features are conditional compilation, inclusion of files and the use of some other directives. There is a limited support for macros, object macros are limited to be integer values which can be used in conditional expressions of #if directives, there is no recursive substitution of macro values, function macros are not supported. Identifiers in TTCN-3 code will not be replaced by

the values of macros, macros are used only for conditional compilation. A preprocessor directive is usually one line, except when the line continuation is used by placing a backslash at the end of the line. Line continuation of TTCN-3 code lines is not legal. Example of a multi-line macro:

```
#if 100== \
50+50
```

The above two lines are one logical line: #if 100==50+50

C preprocessor conditional expressions are integer expressions which can contain literal values (64 bit signed integers) and macro identifiers. These expressions are evaluated while parsing the preprocessor directive, in case of #define the value of the macro will be the result of the evaluation. For example:

```
#define MACRO 1+2+3#
if MACRO==(12/2)
log(MACRO); // in TTCN-3 code macros are not used!
// if there is no constant or variable named MACRO in TTCN-3 then
// there will be a semantic error here
#endif
```

the value of MACRO is 6, this value is used in the #if directive. Integer literals can be decimal, octal and hexadecimal. Conditional constructs can be nested. Inactive branches are displayed in a darker color in the editor.In conditional expressions operators used on integer values in the preprocessor of the C language can be used. For logical operations the integer value 0 is false and non-zero values are true. The special operator **defined** can be used to check if a macro value exists. Example code:

```
#if (M1 + M2 * 123) > ( M3 & 0xABCD )
const integer cint := 123;
#elif defined M4 || !defined M5
const integer cint := 234;
#else
#error Invalid macro settings!
#endif
```

File inclusion is supported, the included files should have the extension ttcnin. The content of these files is included at the position of the #include directive, multiple inclusion of the same file and recursive inclusion is supported. Example *myfile.ttcnin* file:

```
#ifndef _MYFILE_INCLUDED_
#define _MYFILE_INCLUDED_
const integer cint := 123;
#endif
```

The conditional part prevents multiple inclusion of the same source code, this is useful if

ttcnin.myfile is included in other ttcnin files which are included in the same ttcnpp file.

To define macros outside of files the Eclipse TITAN plug-in uses the settings given in the **TTCN-3 Preprocessor** part of the **Internal makefile creation attributes** tab on the **TITAN Project Property** page. Macros can be defined by adding them in the **Defined Symbols (-D)** table. The Included directories setting is not used, the name of included ttcnin file must always be relative to the ttcnpp file in which it is included). Every project in Eclipse has its own defined macros (symbols), other projects do not see these macros. This is an important difference between the command line tools and the designer plug-in, the makefile does not know about projects.

`#define MACRO_NAME <expression>`	Define a macro, it's value is the value of the integer expression
`#undef MACRO_NAME`	Delete a macro
`#ifdef MACRO_NAME`	The code in this branch is active if the macro `MACRO_NAME` was defined previously
`#ifndef MACRO_NAME`	The code in this branch is inactive if the macro `MACRO_NAME` was defined previously
`#if <expression>`	The code in this branch is active if the expression evaluates to non-zero (true)
`#if defined MACRO_NAME`	The same as `#ifdef`
`#if ! defined MACRO_NAME`	The same as `#ifndef`
`#if not defined MACRO_NAME`	The same as `#ifndef MACRO_NAME"` or `#if ! defined MACRO_NAME"`
`#elif <expression>`	The code in this branch is active if the expression evaluates to non-zero and no previous branches were active
`#else`	else branch, active if no previous branches were active
`#endif`	End the conditional construct (end last branch)
`#line, #pragma, null, linemarker`	These directives are ignored, makers: ignored/warning/error depending on the setting in preferences
`#include "filename.ttcnin"`	The file name must be provided in a string (`<filename.ttcnin>` notation is not supported). If the file does not exist or it is not found in the project then an error is displayed
`#error <free text>`	Display the free text as an error marker
`#warning <free text>`	Display the free text as a warning marker

NOTE ttcnpp files are not analyzed incrementally even if incremental analysis is switched on.

7.5.2. Limitations

The on-the-fly parser does not support the single line comment in ASN.1 files when placed right after non-comment elements. A simple workaround for this problem is to insert a SPACE character between the last non-comment character and the “—” sign.

Limitations of preprocessing:

Advanced editing features such as rename refactoring may fail or not work as intended in some cases when pre-processor macros are present in the code. According to the preprocessing logic, code in inactive branches of preprocessor conditionals must be ignored, and so exempt from advanced functionalities (like semantic checking, rename refactoring). In case of multiple inclusion of the same code the same source code may be part of different semantic constructs, for example in rename refactoring the changed source code can affect all other related semantic constructs.

In case of file inclusion, the locations of error and warning markers may be invalid, pointing to the wrong file (usually to the ttcnpp file instead of the ttcnin file where the error is located). This is a limitation of the current parsing mechanism which is optimized for the 1 module == 1 file assumption.

Character constants cannot be used in conditional expressions

7.6. On-the-fly Semantic Checking

On-the-fly semantic checking is done after the on-the-fly parser has finished parsing. The level and complexity of this check is on the same level with the command line compiler, but is done much faster.

7.6.1. Limitations

The following structures are not yet analyzed, and as such not all error cases related to them will be detected:

- Encoding and variant attributes are not analyzed, in fact not even parsed. This implies, that for example it is not able to detect if encoding/decoding functions are used with types that does not have the required encoding attributes.
- Charstring and universal charstring patterns are not analyzed. This implies that even though in some cases matching with regular expressions could be evaluated in compile time, the semantic checker will not be able to do that.
- In ASN.1 table constraints, any type values (open type notation) are not checked.

7.7. Content Assistance

Content assistance is a feature providing context-sensitive content completion upon user request for source files.

The content assistant can be activated either by a key combination (which by default is set to **CTRL + SPACE**) or by typing a . (dot) before the keyword. To insert an element from the proposed ones,

double click it or **select** it and press **Enter**. If only one element is proposed it is inserted automatically.

When an element is selected in the list of the proposed elements, a pop-up window containing a short description may appear.

7.7.1. Assistance with Keywords

Editors support a basic level of content assistance, namely the listing of the appropriate keywords (next figure).

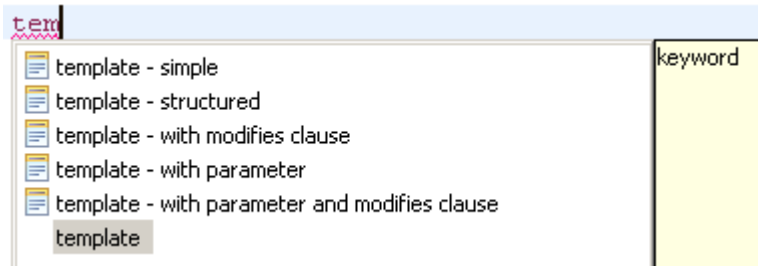


Figure 102. Content assistant

7.7.2. Assistance with Code Skeletons

The intermediate level of assistance inserts structural elements into the source code (following figure). Inserting skeletons is only supported for TTCN-3, TTCNPP, TTCNIN and ASN.1 files.

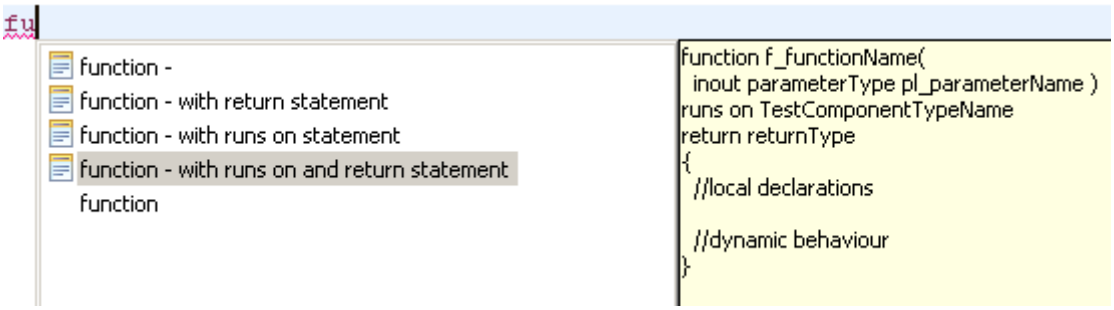


Figure 103. Skeletons in the content assistant

Static and dynamic skeletons are both marked with a unique icon.

A short description of them is provided after the name of the skeleton if a skeleton has several slightly different versions. A popup window shows the text about to be inserted.

Using the Inserted Skeleton

The insertions may contain linked editing points (next figure).

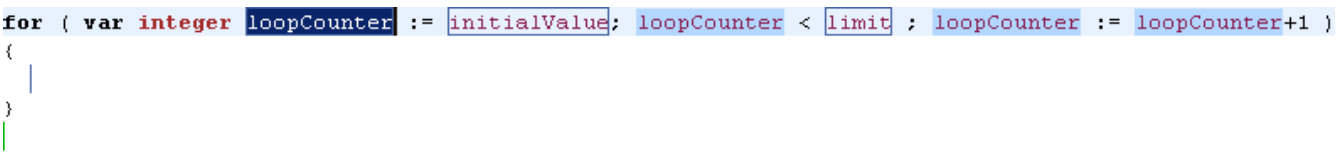


Figure 104. Example inserted skeleton

Hints for using the inserted skeleton:

- The **TAB** key can be used to move between the editing points.
- If two or more editing points are linked, they will have the same content. This means that no matter which one of them is edited, the others take up the same value.
- To leave this insertion mode and validate the insertion, press the **ESC** key.

7.7.3. Assistance with Dynamic Elements

The highest level of content assistance is available for TTCN-3 and ASN.1 files. It is providing scope and type structure information that has been parsed and collected by the on-the-fly parser. The calculation of the proposals is done this way:

1. The reference to be completed is identified strictly using character data available before the completion point.
2. Based on the position of the completion point the smallest enclosing scope is looked up.
3. From the smallest scope found the scope hierarchy is traversed in a bottom-up manner to find the possible definitions. (The definitions imported are checked after the definitions of the actual module).
4. When a relevant definition is found the search for possible proposals continues inside its structure. For example, if the definition is a variable of a structured type, the reference is used to detect the subtypes or fields that the reference could point to if it were to be completed that way.

The proposals are ordered in the following way (definitions don't hide each other in the proposal list):

1. Dynamic elements available in the given scope. The elements are ordered by the distance of the element definition from the completion point in the scope hierarchy. For example, a local variable will always precede module definitions. The definitions that are most likely to be used are placed earlier in the list. If there is more than one proposal from the same scope, they are ordered alphabetically.
2. Skeletons available in the given scope. The skeletons are ordered alphabetically.
3. Keywords available. The keywords are ordered alphabetically.

As an experimental feature, the content assistant is made context-aware in some cases meaning that it lists more relevant proposals based on the context where it was initiated. The following cases are handled (not a complete list):

- list of modules after `import from` and `friend` keywords;
- list of component types after `runs on` and `system` keywords;
- list of matching/compatible type of elements from the visible scope for the right side of an assignment;
- list of matching/compatible type of elements from the visible scope for a function parameter;
- list of the enumerated items of the matching type for the right side of an assignment;
- documentation comment tags when editing documentation comment section.

When this context-aware content assist is invoked, it also provides the documentation comments (if available) related to the selected proposal.

7.7.4. Content Assistance Limitations

Full context sensitivity is not possible yet. Only the scopes and the type structures are used to filter the list of proposals. For this reason, the content assistant might offer completion proposals, which are possible in the actual scope but not in the actual context. It is the user's task to choose the right proposal.

Only data gathered and stored by the on-the-fly parsers can be offered. If this data is outdated or not complete, the content assistance will also offer outdated or limited information. Section 3.1 explains how this can happen.

7.8. Documentation comments

Documentation comments are supported according to TTCN-3 Documentation Comment Specification and also available for extending the capabilities of some IDE features. Beside having a brief description in the code about the commented language element, the documentation comments are parsed and able to assist the users who are writing or browsing the code through integrating the documentation comments in some standard IDE features. The documentation comments are integrated into the following IDE functions:

- Code hover popups: when hovering over certain language elements, a popup window appears containing the related and parsed documentation comment. The documentation comments are completed with relevant semantical information, e.g. with the types of the formal parameters of a function. An example of this feature is presented in the figure below. The content of this popup window can be changed to peek the definition of that language element (for details see Peek declaration). To enable this feature see the Content Assist Preferences.

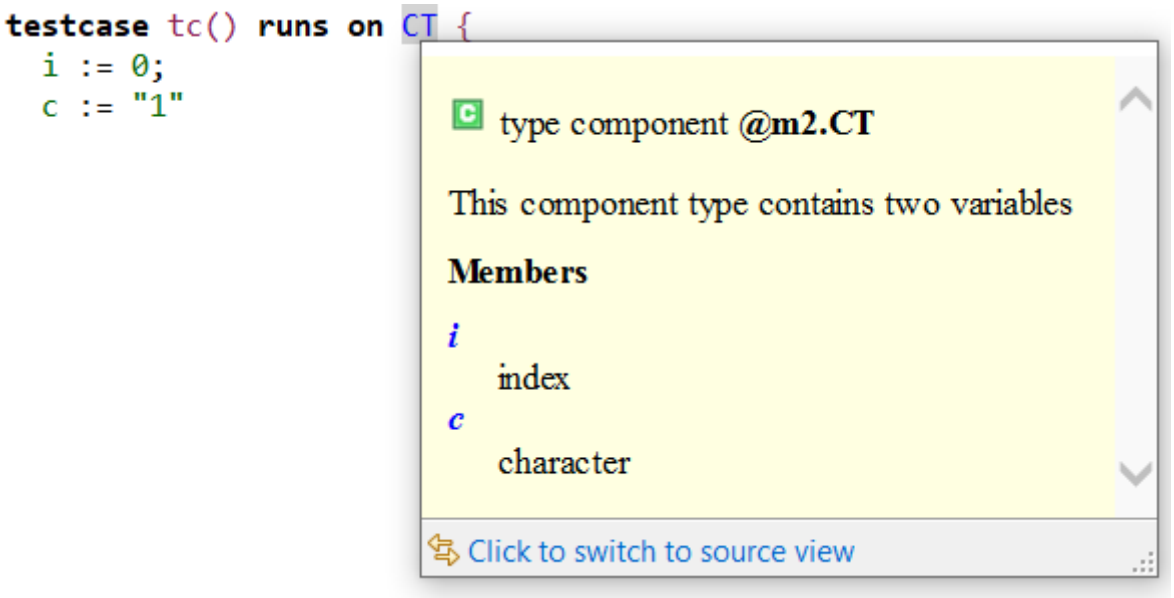


Figure 105. Documentation comments usage during hovering

- Content assistance: in certain use cases the documentation comments are also presented to the listed proposals when the user requesting for code assistance/completion. This is illustrated in

the figure below. Such use cases are listed in Assistance with Dynamic Elements. In these cases, the content of the popup window is identical to the previous one.

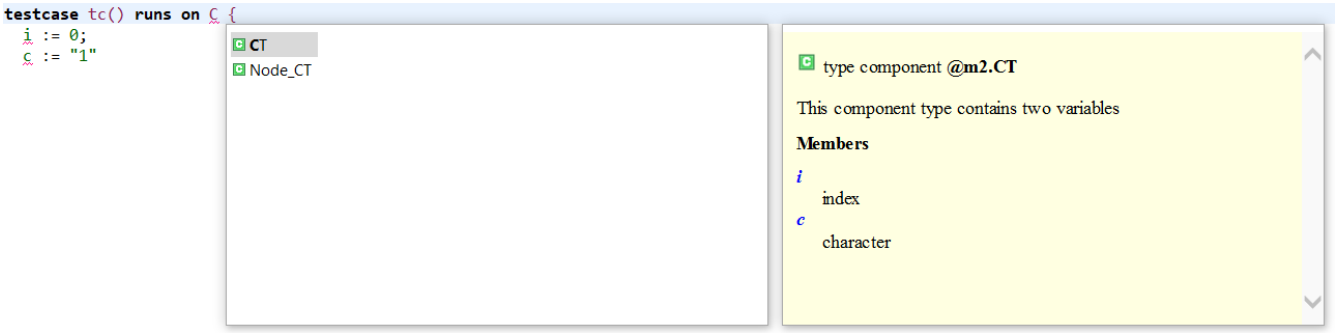


Figure 106. Documentation comments usage during code assist

- Semantic information and documentation comments consistency checks: it is intentionally left a big user freedom to write basically anything in the documentation comment section of the code, however, there are several validation mechanisms available to notify the users about possible problems with the code and/or with the documentation comment. The severity of these problems is configurable (see Errors / Warnings Preferences). The comment tags are validated against the language element to which they are related, e.g. the `@verdict` tag is marked if it used in documentation comment of a type definition. Also, the related comments of the formal parameter lists are also checked against the definition, e.g. the number of the documented parameters differ from the related function definition, etc.
- Code highlight: the specification recommends to use the `@status` tag for describing the actual status of the object. If the deprecated status (`@status deprecated`) is used, the highlight of the relevant code parts is changed according to the configured style (the semantic highlighting preference must be enabled, see Syntax Coloring Preferences).

To have the listed functions, the on-the-fly checking of document comments preference must be enabled (see On-the-fly Checker Preferences) beside the function specific configuration.

Additional features:

- the `@url` tag is automatically converted to a link.
- HTML tags are allowed in the documentation comments. Actually, the popup windows work as simple web browsers, thus complete pages or PDF documents can also be shown.

7.8.1. Generate documentation comment

To simply the work with the documentation comments, skeletons can be generated taking into account the actual language element, on which the generation was initiated. Generate documentation comment can be invoked either by a key combination (by default **Ctrl+F3**) or by selecting/clicking on the language object, then **right clicking** and selecting **Generate doc comment**. The generated comment skeleton can be easily filled with the relevant information by the user. Only the most typical comment tags specific to the given language element are generated, see the examples below.

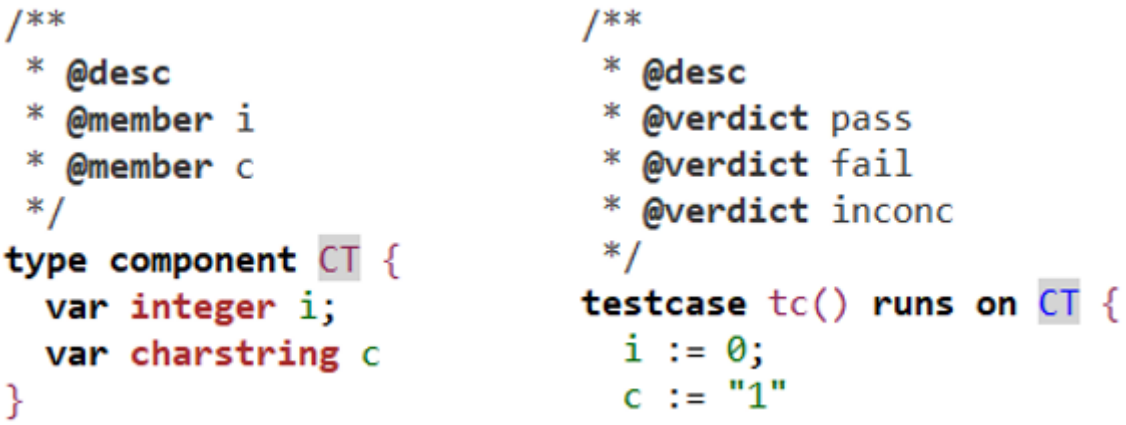


Figure 107. Examples of the generated documentation comments

7.8.2. Documentation comments limitations

Implicitly tagged documentation information is not supported.

The documentation comment has to directly precede the language element to which it is related to, i.e. new line characters are not allowed between the language object and its documentation comment. Otherwise, it is considered as there is no documentation comment to the specific object.

7.9. Find Declaration

Open Declaration provides a feature to jump to the declaration point of the selected element.

Open Declaration can be invoked either by a key combination (by default **F3**) or by **right clicking** anywhere on the screen and selecting **Open Declaration**. The element is determined by the current position of the cursor when the functionality is invoked.

The search for the declaration is done this order:

1. The reference to be searched for is identified using only character data available before the completion point and after the completion point up to the next dot, opening bracket, opening square brace (or another character that cannot be part of a reference). For example, in case of the string `module.definition.field`:
    a. If the cursor is somewhere inside, right before or right after the word `module`, the reference will be `module`.
    b. If the cursor is somewhere inside, right before or right after the word `definition`, the reference will be `module.definition`.
    c. If the cursor is somewhere inside, right before or right after the word field, the reference will be `module.definition.field`.
2. Based on the position of the completion point the smallest enclosing scope is looked up.
3. From the scope found the scope hierarchy is traversed in a bottom-up manner, to find the possible definitions. (The definitions imported are checked after the definitions of the current module).
4. When a relevant definition is found, the search for possible proposals continues inside its structure. For example, if the definition is a variable of a structured type, the reference is used

to detect the subtypes or fields that the reference could point to.

5. If no definitions could be found in the actual module or in the ones imported by it, a special search takes place. It traverses every module of the actual project to find possibly matching definitions.

Jump to the location of the declaration takes place automatically if a declaration was found in the actual module or in one of the imported modules. The target file will be opened in an editor window taking the focus (if not already done so). The location of the declaration is revealed and selected.

If no valid declarations could be found in the whole module, this will be stated in the **TITAN Debug Console** and **the status line of Eclipse**, without presenting any extra pop-up windows. This way the user can invoke the functionality again, without the need to close several error indicating dialogs.

Open Declaration works for TTCN-3 and ASN.1 modules and configuration files. For configuration files Open Declaration can be used to:

Open configuration files listed in the include section. If the selected configuration file cannot be found the error is reported in the **TITAN Debug Console** and **the status line of Eclipse**.

Find module parameter declarations. If the module parameter is given as a module specific module parameter (e.g. `module.parameter`) only the given module is searched through for the declaration. Otherwise (e.g. `.parameter` or `parameter`) all modules of the project are taken into account. Duplicate module parameter declarations and errors are reported in the same way as for macro definitions.

7.10. Find References

"Find references" provides a feature to search for all elements that refer the selected TTCN-3 or ASN.1 element. The user can select TTCN-3 definitions of types, constants, variables, templates, variable templates, functions, testcases, altsteps, components, ports, formal parameters, enumerated values, etc. ASN.1 type and value assignments can be selected in ASN.1 files. In case of structured types (record, set, union, etc.) the individual fields can be selected, in this case all references to that field will be displayed. The source files should be syntactically and semantically correct prior to starting the search, otherwise it cannot be guaranteed that all references to the given element will be found.

Find References can be invoked either by a key combination (by default **F4**) or by **right clicking** anywhere on the screen and selecting **Find References**. The element is determined by the current position of the cursor when the functionality is invoked.

The found references will be displayed in the standard Eclipse search result view, it is usually displayed at the bottom as a new tab. The found references are displayed in a tree view, grouped by module. If it cannot be determined what element we are trying to search for, an error message will be displayed and the search result view will not be opened. The error message will be displayed in **the status line of Eclipse**, without presenting any extra pop-up windows. In the search result view clicking on an occurrence will open the source file and jump to the reference location.

A more precise description of this feature is searching for identifiers that are used in a context

where they identify the language element that we are searching for. A reference can contain multiple identifiers, for example in the case of a recursive record definition:

```
type record MyRec {
    MyRec rec optional,
    charstring str
}
...
var MyRec v_myrec;
...
v_myrec.rec.rec.rec.str := "foo";
```

Searching for field rec will give 3 hits in the above line, because the reference `v_myrec.rec.rec.rec.str` contains the identifier of the rec field 3 times.

It is not always guaranteed that all references to the selected element will be found or that an element that should be selectable can be selected, because parsing and semantic analysis of all the source code is not 100% completed in the Eclipse plug-in.

7.11. Mark Occurrences

The TTCN-3 and ASN.1 editors are able to highlight the occurrences of the currently selected element in the source code. The search for the occurrences is based on semantic information (see here). As the selection or the position of the cursor changes in the editor, the marks are updated automatically. The feature can be configured on the TITAN Preference page (see here).

7.11.1. Limitations

Occurrences of the following language elements are not highlighted:

- References to modules
- Sub-references of a reference

  in the example below, if the cursor is on the `field1` sub-reference, the occurrences will not be marked.

  ```
  myRec.field1 := 1;
  ```

- Fields of types in the assignment notation. In the example below, if the cursor is on one of the fields (`field1` or `field2`) the occurrences will not be marked. `var MyRec myRecord := {field1 := 0, field2 := 1};`
- The occurrences of keywords, predefined functions, primitive data types and literals are not marked.

7.12. Peek declaration

As experimental feature, this functionality shows the definition of the selected TTCN-3 language element in a popup window to avoid navigating to the actual location of the definition (see the

figure below). If enabled and configured (for details see Content Assist preferences), the popup window appears when hovering over elements of the source code. This feature builds upon the Find References feature. It works with most of the TTCN-3 language elements. The feature can be initiated from the popup menu when selecting a TTCN-3 language element or by pressing the hotkey F9 as well.

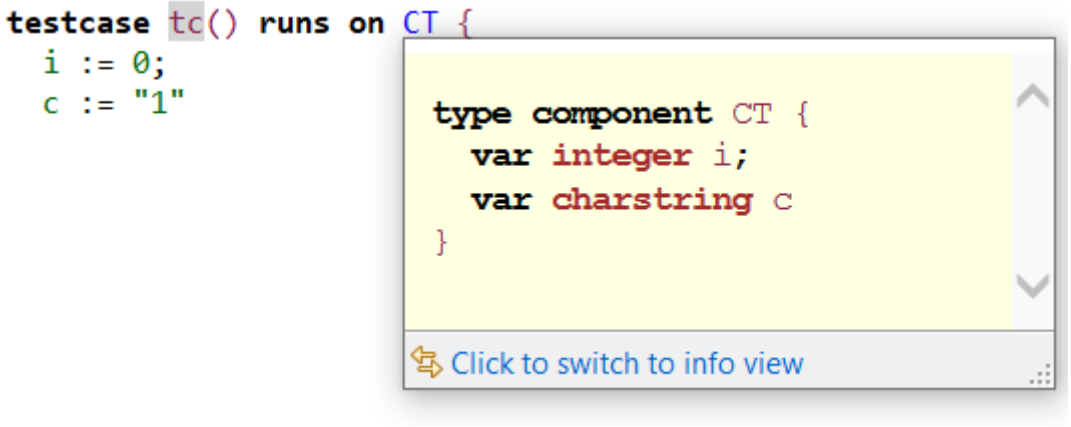


Figure 108. Peek declaration example

7.13. Refactoring

7.13.1. Rename Refactoring

This feature builds upon the Find References feature, it can be invoked the same way and it works on the same language elements. Most of the TTCN-3 and ASN.1 language elements can be renamed using this feature.

The user can select TTCN-3 definitions of types, constants, variables, templates, variable templates, functions, testcases, altsteps, components, ports, formal parameters, enumerated values, etc. ASN.1 type and value assignments can be selected in ASN.1 files. In case of structured types (record, set, union, etc.) the individual fields can be selected. The source files should be syntactically and semantically correct prior to starting the renaming. By default, projects containing errors or ttcnpp files cannot be refactored, but this behavior can be changed in the TITAN Preferences on the On-the-fly checker page. If refactoring is done on projects which contain syntax or semantic errors or ttcnpp files, then it cannot be guaranteed that all occurrences of the given definition or field will be renamed because some occurrences may reside in places that are inside erroneous source code or places that are not active after pre-processing of ttcnpp files.

Rename refactoring can be invoked either by a key combination (by default **Ctrl+F4**) or by **right clicking** anywhere on the screen and selecting **Rename Refactoring**. The element is determined by the current position of the cursor when the functionality is invoked. If it cannot be determined what element we are trying to rename, an error message will be displayed. The error message will be displayed in **the status line of Eclipse**, without presenting any extra pop-up windows.

The refactoring process starts with a dialog box where the new name should be specified, the new name must be a valid TTCN-3 or ASN.1 identifier.

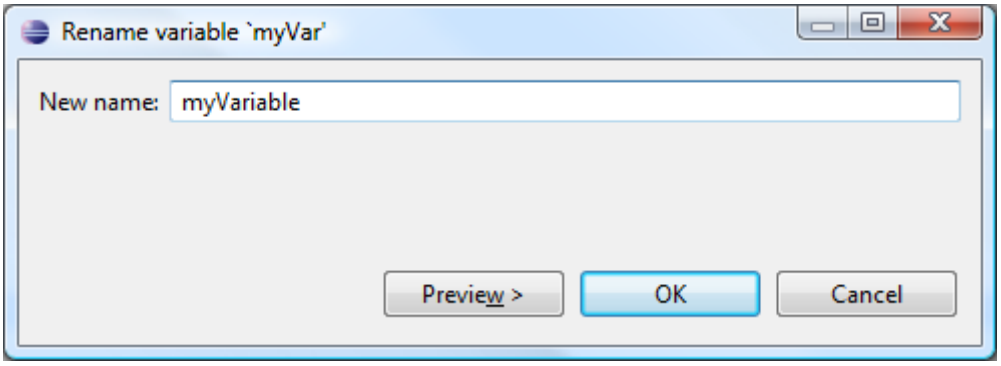


Figure 109. Rename refactoring

A preview of all modifications is available; the preview window shows the original and the refactored source code side by side. The source code will be modified only if the OK button was selected.

7.13.2. Limitations

Refactoring might not be able to operate correctly in the following case:

- If macro definitions are used in the source code, refactoring will not be able to operate on the code parts which are not active at the time of the refactoring. The reason for this is, that those parts are not visible for the semantic analyzer. In this case the user is warned for possible issues.

7.14. Editing Configuration Files

Configuration files can be edited in their own editor in a textual format just like any other file; however, the editor also provides graphical pages to ease this process. As it can be seen on Figure 93, these graphical pages can be selected by clicking on the tabs in the bottom of the editing area.

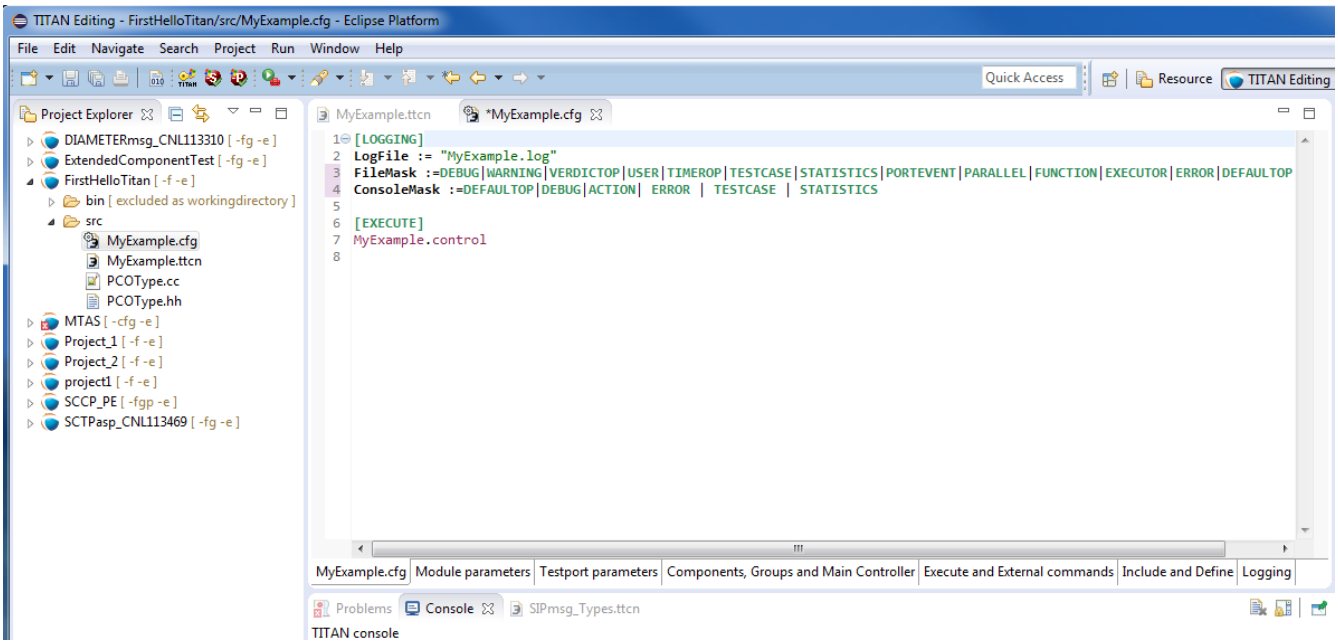


Figure 110. Editing a configuration file

Whenever the textual page is edited the on-the-fly parser is run within one second and the contents

of graphical pages get updated; however, to save the contents of the graphical pages (and to execute the on-the-fly parser on them) pressing the buttons **Ctrl+S** is required.

+ NOTE: The content of the textual page is also updated when it becomes active. The example on the figure below shows an error detected.

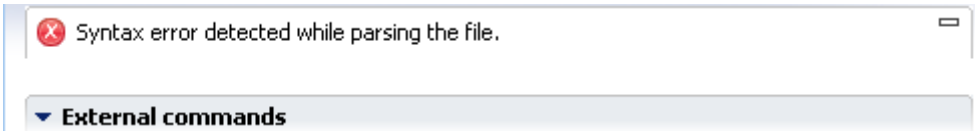


Figure 111. Syntax error detected

The graphical pages are explained in detail in the sections below.

7.14.1. Module Parameters Section

On this page (new) values can be assigned to parameters defined in the TTCN–3 modules.

A new parameter can be added by clicking the **Add...** button. The column **Module name** contains the name of the module where the parameter is used. The parameter can be used in all modules when this column is left blank or filled with an asterisk. The column **Module parameter name** is self-explanatory. The value of the parameter is determined by the string in the pane **Module parameter details** in free form as parameters may have different formats.

Highlighted existing parameters are removed by clicking the **Remove** button.

The field **Total** under the buttons shows the number of the defined module parameters.

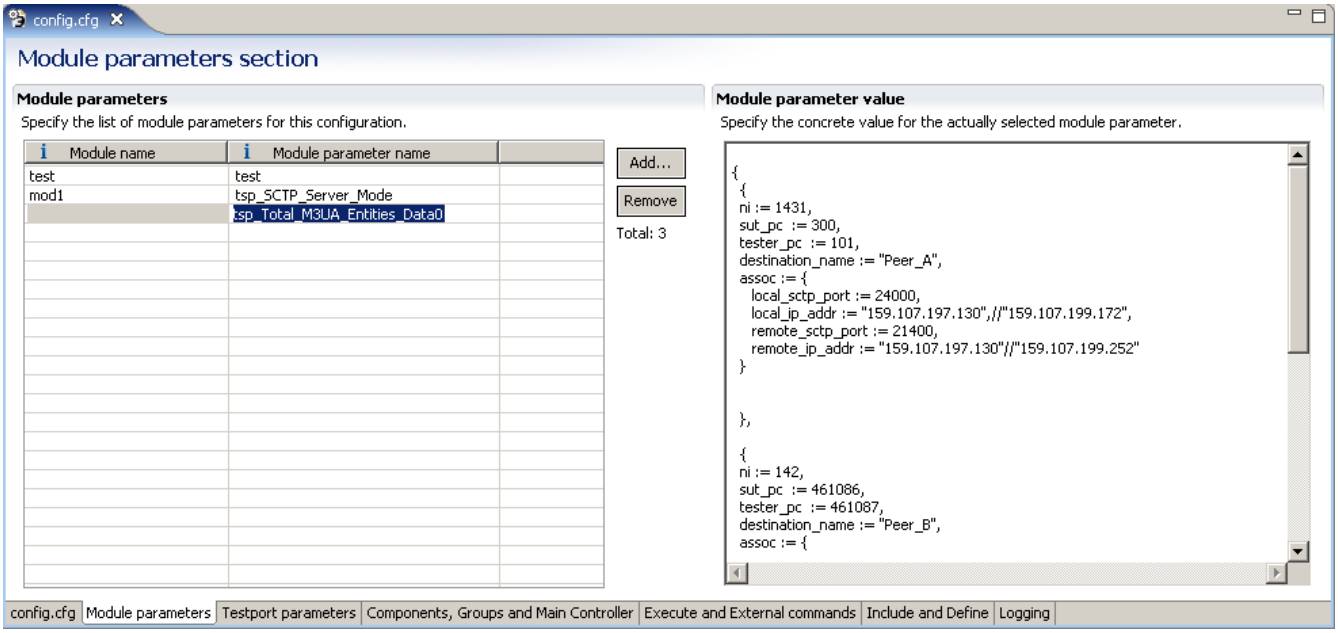


Figure 112. Module parameters

Changes made to the parameters must be saved by the shortcut key **Ctrl+S**.

7.14.2. Test Port Parameters Section

The values of all parameters on this page are passed to test ports.

A new parameter can be added by clicking the **Add...** button. The column **Component name** contains the name of the component defining the test port. An asterisk (*) denotes all ports of the Test System Interface. The column *Test port name* is the name of the test port. The column **Parameter name** is self-explanatory. The value of the parameter is determined by the string in the pane **Test port parameter details** in free form as parameters may have different formats.

Highlighted existing parameters are removed by clicking the **Remove** button.

The field **Total** under the buttons shows the number of the defined module parameters.

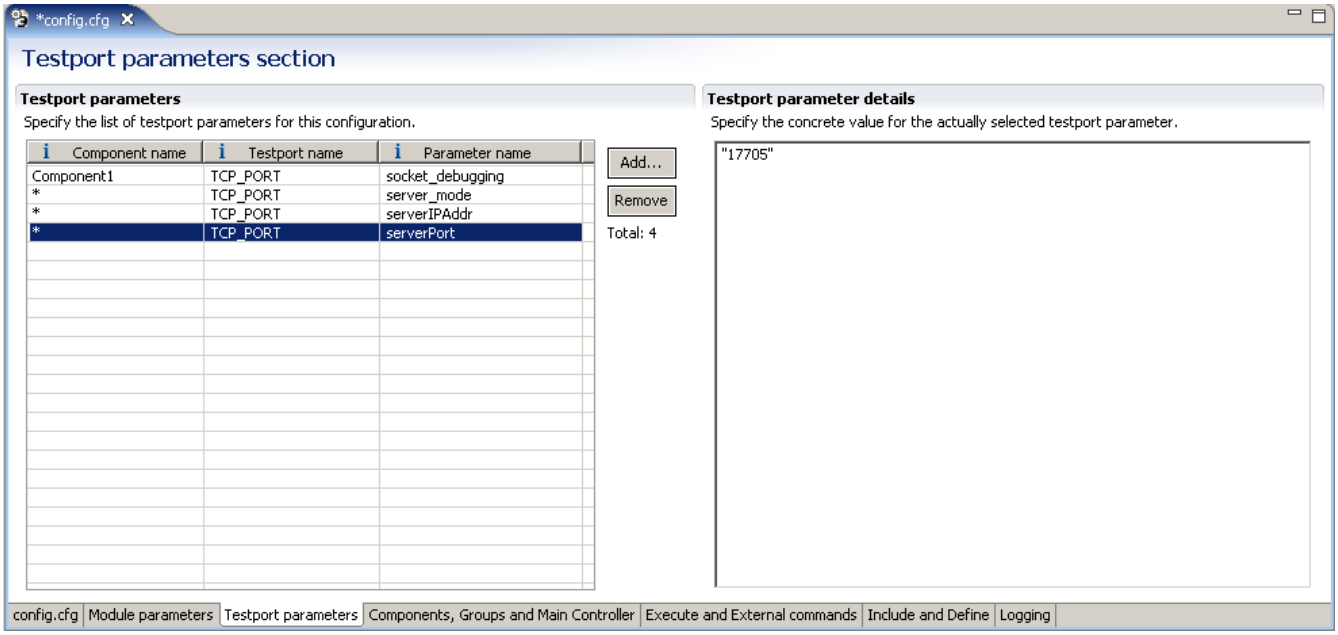


Figure 113. Test port parameters

Changes made to the parameters must be saved by the shortcut key **Ctrl** + **S**.

7.14.3. Components, Groups and Main Controller Section

This page contains parameters of three configuration file sections. The parameters make only sense in parallel mode.

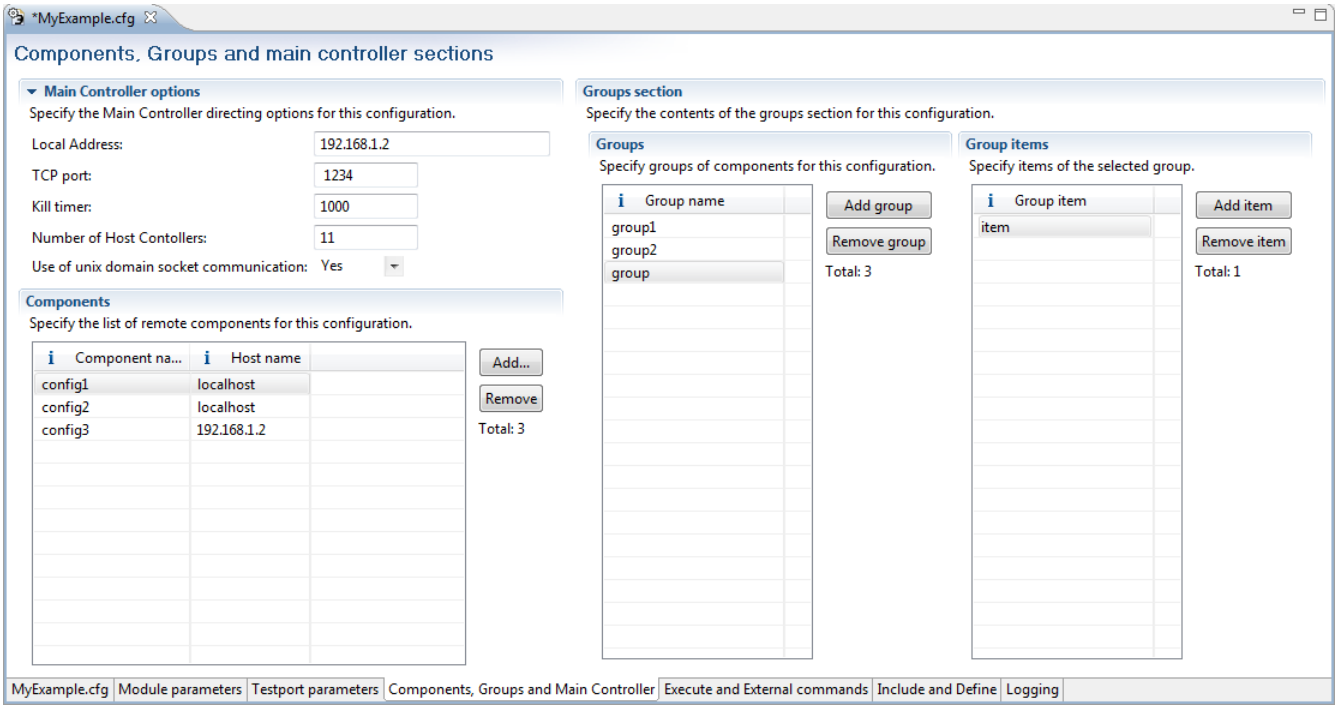


Figure 114. Components, groups and Main Controller

Changes made to the parameters must be saved by the shortcut key **Ctrl** + **S**.

Main Controller Options

The options herein control the behavior of Main Controller (MC). Clicking the triangle in the title line will collapse the section leaving more room to the tables.

The options **Local Address** and **TCP port** determine the IP address and TCP port where the MC application will listen for incoming HC connections. The value of **TCP port** is an integer number between 0 and 65535. The recommended port number is 9034. The MC will listen on an ephemeral port chosen by the kernel when this field is left empty or set to zero.

The value **Kill timer** determines how long the MC waits for a busy test component (MTC or PTC) to terminate when it was requested to stop. The value of **Kill timer** is measured in seconds and can be given in either integer or floating point notation. Setting **Kill timer** to zero disables the kill functionality, that is, busy test components will not be killed even if they do not respond within a very long time period. When omitted, the default value of **Kill timer** is 10 seconds.

Number of Host Controllers provides support for automated (batch) execution of distributed tests. When set, the MC will not give a command prompt, but wait for the specified number of HCs to connect. When all connected, the MC automatically creates MTC and executes all items defined in the page **Execute** (see section 7.12.4).

The **Use of unix domain socket communication** field can turn on or off the usage of efficient communication between the main controller and other components of the test system. By default it is turned on except on Cygwin because of performance degradation.

Components

The aim of the **Components** table is to restrict component execution to certain (group of) hosts. These constraints are useful when distributed tests are executed in a heterogeneous environment.

The participating computers may have different hardware setup, computing capacity or operating system.

A new restriction is added by clicking the **Add...** button to the right of the first table. The column **Component name** contains component to be restricted. The column **Host name** contains either a host name, a group of hosts (see here) or an IP address of a host.

Highlighted components are removed by the button **Remove**.

The field **Total** under the buttons shows the number of the restrictions in force.

Group Section

The aim of the tables **Group** and **Group item** is to specify groups of hosts. These groups are used to restrict creation of certain PTCs to a given set of hosts.

A new group can be added by clicking the **Add group** button to the right of the table in the middle. The first column contains the name of the group. are added to the table **Group items** by pressing the button **Add item**.

Highlighted group members or entire groups are removed by the button **Remove item** and **Remove group**, respectively.

The field **Total** under the buttons shows the number of the defined groups and group members.

7.14.4. Execute and External Commands Sections

This page contains parameters of two configuration file sections.

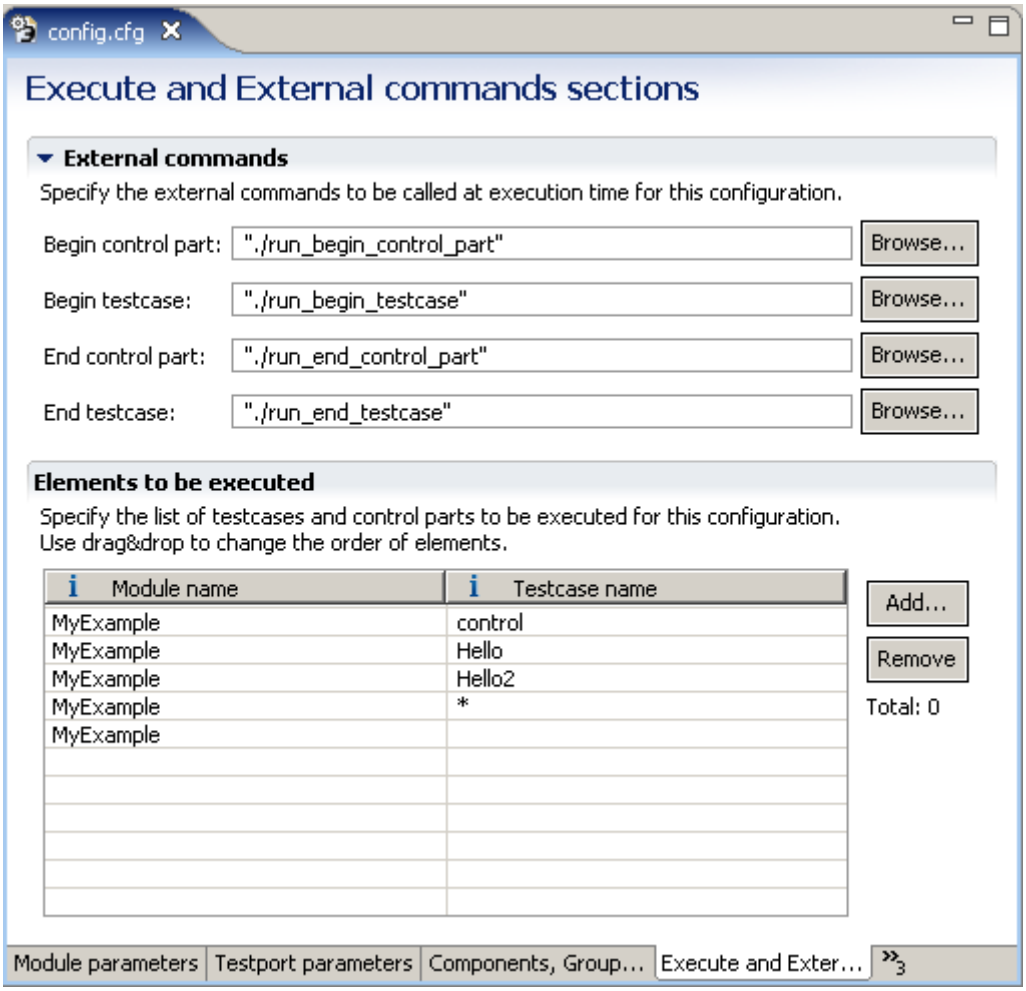


Figure 115. Execute and external commands

Changes made to the parameters must be saved by the shortcut key **Ctrl** + **S**.

External Commands

This section defines external commands (shell scripts) to be executed by the Executable Test Suite whenever a control part or test case is started or terminated. Clicking the triangle in the title line will collapse the section leaving more room to the table. The button **Browse** can be used to locate the shell script.

The field **Begin control part** contains the path to the shell script executed before control part procession.

The field **Begin testcase** contains the path to the shell script executed before testcase execution.

The field **End control part** contains the path to the shell script executed after the control part is processed.

The field **End testcase** contains the path to the shell script executed after a testcase has been executed.

Elements to be Executed

This table points out parts of the test suite to be executed. Only test cases having no parameters can be executed from this section.

A new test case is added by clicking the **Add...** button to the right of the table. The column **Module name** contains the name of the module where the test case is defined. The column **Testcase...** lists the test cases to be executed. An asterisk (*) denotes that all test cases in the given module must be executed.

Highlighted test cases are removed by the button **Remove**.

The field **Total** under the buttons shows the number of the rows in the table.

7.14.5. Include and Define Sections

This page contains parameters of two configuration file sections. Clicking the triangles in the title line will collapse the section leaving more room to the other section.

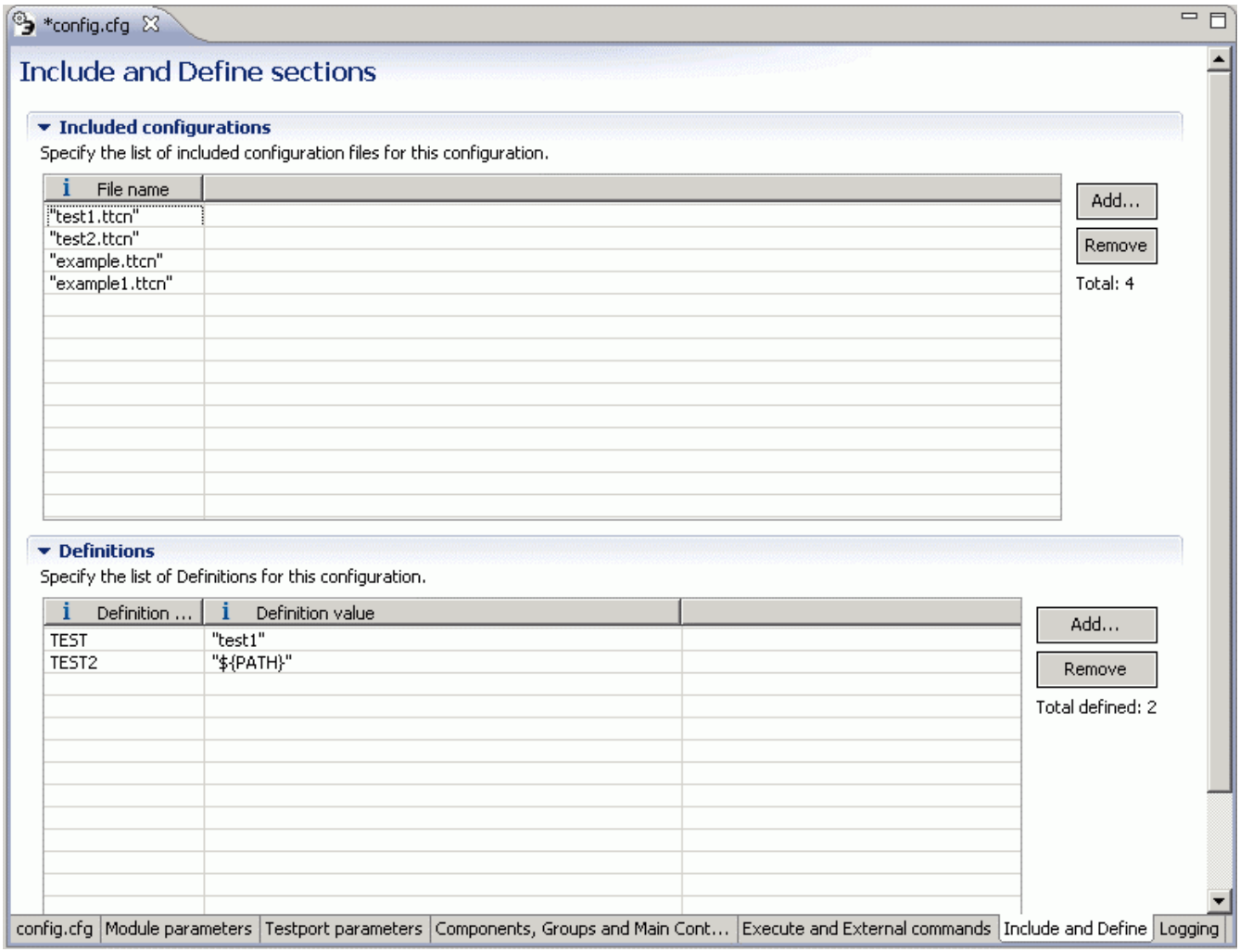


Figure 116. Include and define

Changes made to the parameters must be saved by the shortcut key **Ctrl** + **S**.

Included Configurations

This table lists the configuration files to be imported. This way there is no need to merge configuration files when parameter definitions needed are dispersed over several files.

A new configuration file is imported by clicking the **Add...** button to the right of the upper table. The column **File name** contains between quotation marks the name of the files to be imported.

Highlighted files are removed by the button **Remove**.

The field **Total** under the buttons shows the number of the imported files.

Definitions

This table contains macro definitions that can be used in other configuration file sections.

A new macro definition is added by clicking the **Add...** button to the right of the lower table. The column **Definition** contains the macro name whereas the column **Definition value** contains the macro itself between quotation marks.

Highlighted macros are removed by the button **Remove**.

The field **Total** under the buttons shows the number of the defined macros.

7.14.6. Logging Section

The executable test program produces a log file during its run. The log file contains important test execution events with time stamps. Logging may be directed to file or displayed on console (standard error). This section explains how to set the parameters connected to the log file.

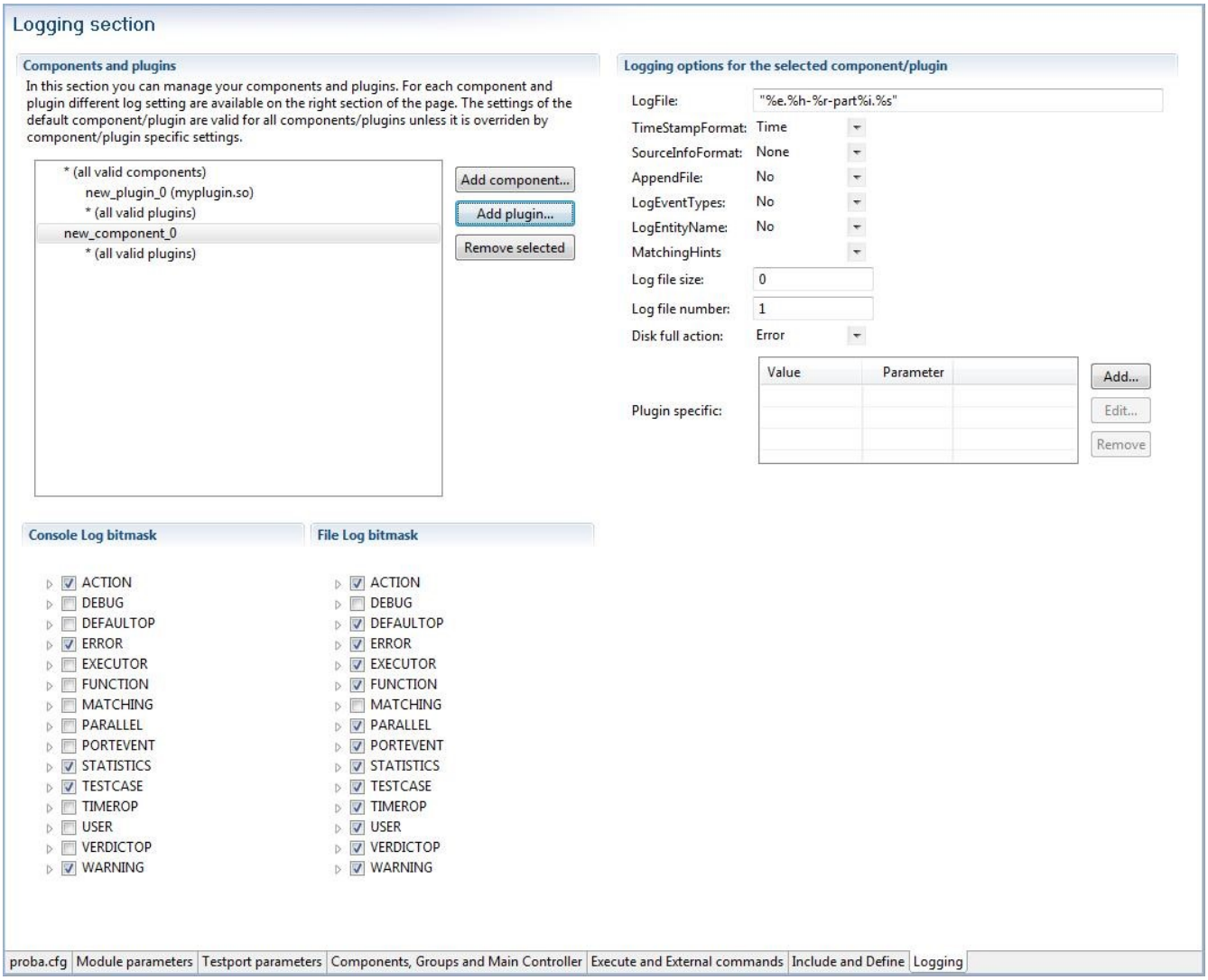


Figure 117. Logging

Components and Plug-ins

In the components and plug-ins section a tree of components and plug-ins can be created and managed.

On the first level of the tree components can be added using the *Add component...* button.Using the *Add plug-in...* button plug-ins can be added under each component on the second level of the tree.

Both component and plug-in names must be valid identifiers. The only exception is the "*" component, this can be used to specify settings which are applied to all components and plug-ins.The "*" plug-in is automatically inserted; this can be used to specify settings which are applied to all plug-ins of the selected component. To specify settings for a specific component and plug-in one of the tree elements must be selected.

Any component or plug-in can be deleted using the *Remove selected* button.

Logging Options for the Selected Component/Plug-in

LogFile: the name of the log file between quotation marks. The string value entered may contain metacharacters that are substituted dynamically during test execution. The available metacharacters are listed in the section `LogFile` of [4].

TimeStampFormat can have three possible values:

`Time` stands for the format `hh:mm:ss.microsec.`

`DateTime` results in `yyyy/Mon/dd hh:mm:ss.microsec.`

`Seconds` results relative timestamps in format `s.microsec.`

SourceInfoFormat controls the appearance of the test event location information (position in the TTCN–3 source code). The option can take one of the three possible values: `None`, `Single` and `Stack`. If set to `Single`, the location information of the TTCN–3 statement is logged that is currently being executed. When `Stack` is used, the entire TTCN–3 call stack is logged. The value `None` disables the printing of location information.

AppendFile controls whether the run-time environment shall keep the contents of existing log files when starting execution. The possible values are `Yes` or `No`. The default is No, which means that all events from the previous test execution will be overwritten.

LogEventTypes can be useful for log post-filtering scripts. The possible values are `Yes`, `No`, `Detailed` and `Subcategories`. These values are explained in the section `LogEventTypes` of [4].

LogEntityName: if set to `Yes`, the name of the TTCN–3 entity is indicated in the log file along with the file name and line number.

MatchingHints: controls the verbosity of the logger regarding to template matching. The possible values are `Compact` and `Detailed`. The default is `Compact`, which shows the matched/unmatched fields of messages in a dot-separated notation. The Detailed version is similar to the former logging format. It's more verbose and preserves the message structures.

Log file size limits log file growth: when the file reaches the limit given in kilobytes, the log file is closed and a new one is opened with a different name. The naming scheme is explained in the section `LogFileSize` of [4].

Log file number limits the number of log files stored. When this limit is reached (because new ones are being opened as described in the paragraph above), the oldest log file of the component is deleted.

Disk full action determines what to do when writing to the log file fails.

`Stop:` test case execution continues without logging.

`Retry:` TITAN attempts to restart logging activity periodically.

`Delete:` the oldest log file is deleted; logging continues to a new log file fragment.

`Error:` a runtime (dynamic) test case error is triggered.

Plug-in specific: this table lists the key-value pairs, that a given plug-in should be called with to parameterize its behavior.

Console Log Bitmask and **File Log Bitmask** determine what sort of events will be logged to the console respectively to the log file. Tables 11 to 22 of [4] explain the meaning of the different logging classes.

7.14.7. Limitations on the Graphical Pages

The entered parameter values are not verified: any character string can be entered in any field.

Chapter 8. Contents of the Problems View

This section presents how TITAN Designer plugin is integrated in the Problems view.

Whenever a problem is found in a project related resource, a marker is placed on that resource in the TITAN Designer.

In general, when any part of the TITAN Designer plugin checks a given file for problems, it first removes the markers from the resource then does the checking; and if any problems were found new markers are placed on the resource. The only exception to this is that the on-the-fly parser cannot remove markers generated by the compiler; but instead it turns them grey, this was designed so because the checks of the compiler are much more precise than the checks of the on-the-fly parser. The compiler overwrites also the markers of the on-the-fly parser, of course.

8.1. Types of Markers

There are three error marker types indicating:

- issues reported by the compiler;
- syntactic errors reported by the onthefly parser;
- semantic errors reported by the onthefly checker.

Issues are reported as warnings (minuscule issues) or errors (severe issues that must be repaired as soon as possible).

8.2. Eclipse Provided Features

Every time a marker is created the TITAN plugin tries to provide as much information about the issue as possible to fully profit from the Eclipse features.

The TITAN plugin makes use of the following features:

- Collecting of markers:

  Eclipse collects all of the markers in the **Problems view**, so that they can be handled together in a single place.

- Jumping to a given position:

  The TITAN plugin provides Eclipse sufficient information to make Eclipse jump to the exact problem location when the user **double clicks** a marker. If the file is not opened in the editor Eclipse will first open it and then jump to the location.

- User configurable presentation:

  The users can configure the presentation of the problems by selecting **Window** / **Preferences** / **General** / **Editors** / **Text Editors** / **Annotations**. Here the presentation of errors and warnings can be configured (for example, whether they should be underlined and shown on the side

rulers, what color to use).

- Grouping of markers:

  These markers can be grouped in several, semantically different ways. This will be shown here.

- Displaying the error text:

  Every editor provided by the TITAN plug-in is able to show the error texts of markers placed on a line. The mouse pointer must be placed over a marker to activate this functionality. If several errors were found in the same line, each of their texts is displayed on a new line.

8.3. Grouping of Problems

Grouping of markers can be activated by selecting **Triangle / Group By**.

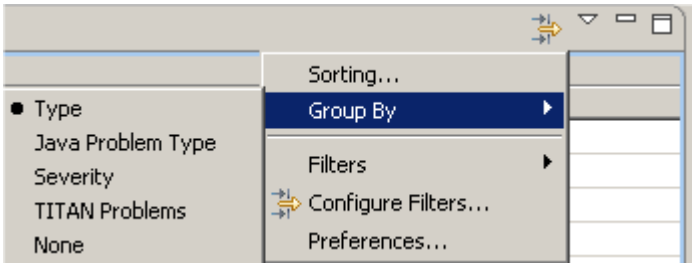


Figure 114. Grouping problems

Other elements of Eclipse can also report problems; these issues will be called other problems. General problems, for example not being able to execute a program, are reported as general Problems by both the local and the remote build procedures.

Groupings supported by TITAN plugin are described in the following sections.

8.3.1. Group by Severity

Here the markers are grouped by their severity, that is, whether they are representing errors or warnings. This grouping is preferable when treating errors first. Other problems are mixed into the problems reported by TITAN plugin.

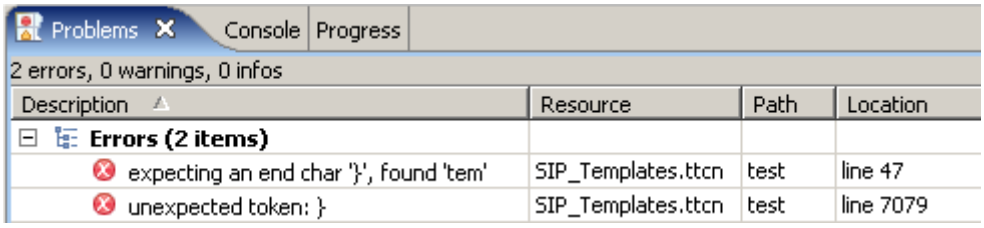


Figure 115. Grouping by severity

8.3.2. Group by Type

Here the problems are grouped by the reporting entity. The following groups are composed by the TITAN plugin:

- TITAN complier problems

- TITAN on-the-fly semantic problems
- TITAN on-the-fly syntactic problems

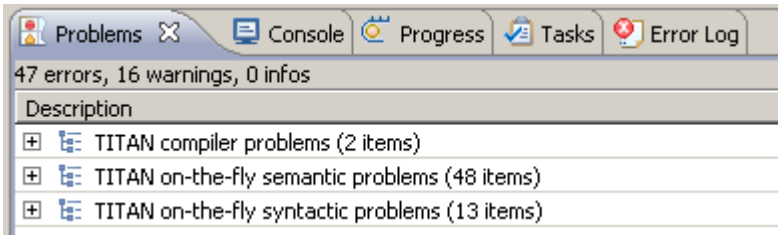


Figure 116. Grouping by type

8.3.3. Group by TITAN Problems

Here every problem reported by the TITAN plugin is placed into the same group labeled TITAN Problems. Other problems are placed into a group labeled Other Problems.

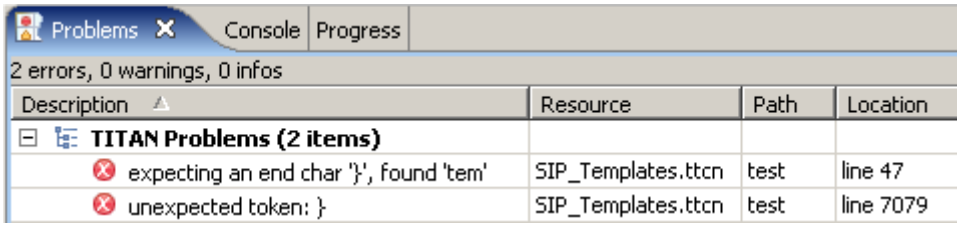


Figure 117. Grouping by TITAN problems

Chapter 9. Contents of the Tasks View

This section presents how TITAN Designer plugin is integrated in the Tasks view.

There are many cases when a developer would like to mark parts of the code; not necessarily because of errors. For example, the programmer may be working on a huge project consisting of many small parts easy to overlook. In this case it is invaluable for the programmer if he can mark parts of the code as not finished. It happens several times in real life development that the design of smaller program parts is shifted so many times and so much in time that people actually forget about it.

9.1. Types of Markers

There are two task marker types:

- TODO markers are created in the code with a single line TODO comment, for example `//TODO this function still needs to be implemented`.
- FIXME markers are created in the code with a single line FIXME comment, for example `//FIXME division by 0 might be possible here`.

It is the on-the-fly parser creating these notifications, not the TITAN compiler.

Eclipse provides all the nice features for Task markers as it did for Problem markers, with the only exception being that grouping is not supported; see here

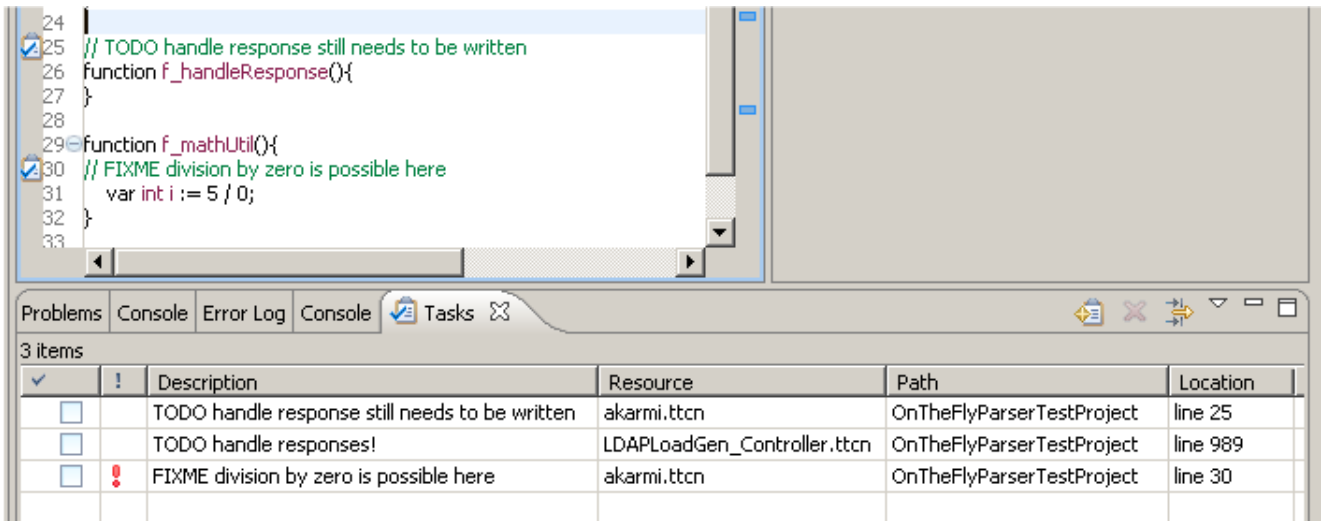


Figure 118. TODO and FIXME task markers

Chapter 10. Contents of the Outline View

This section presents how TITAN Designer plugin is integrated in the Outline view.

It is often useful to get a higher level view of the actual TTCN-3/ASN.1 module, especially if the module is thousands of lines long. The Outline view provides a solution to this problem and makes it easy to navigate inside TTCN-3/ASN.1 modules. If an element is selected in the Outline view the editor jumps to the position of the selected element in the source code.

The Outline view consists of two main parts. The toolbar and the actual tree view.

10.1. The Tree

The Outline view contains a tree, representing the structure of the current TTCN-3/ASN.1 module. Each element is represented in the Outline view by an icon that makes the type of the item easily recognizable and by a text that shows the name and the type of the element or in case of structures with formal parameters their calling convention.

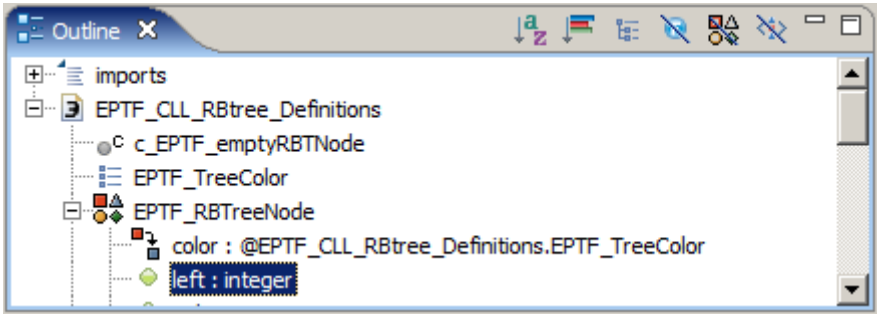


Figure 119. Outline view

The root of the tree always represents the current TTCN-3/ASN.1 module and optionally the list of module importations if there were any. The structure of the underlying levels shows data structure hierarchies, type definition groupings etc.

10.2. The Toolbar

With the functionality available through the toolbar buttons, the elements of the Outline view can be ordered, restructured or the visibility of specific elements can be changed. In the following subsections these toolbar actions will be described.

10.2.1. Sorting Elements

By default the elements in the Outline view are in the order of their position in the TTCN-3/ASN.1 module.

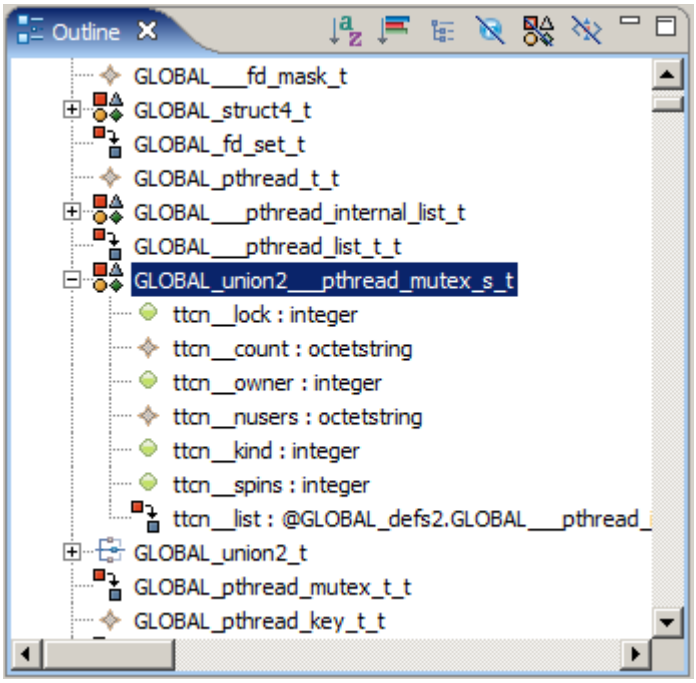


Figure 120. Sorted by position

The elements can be sorted alphabetically with toggling the icon.

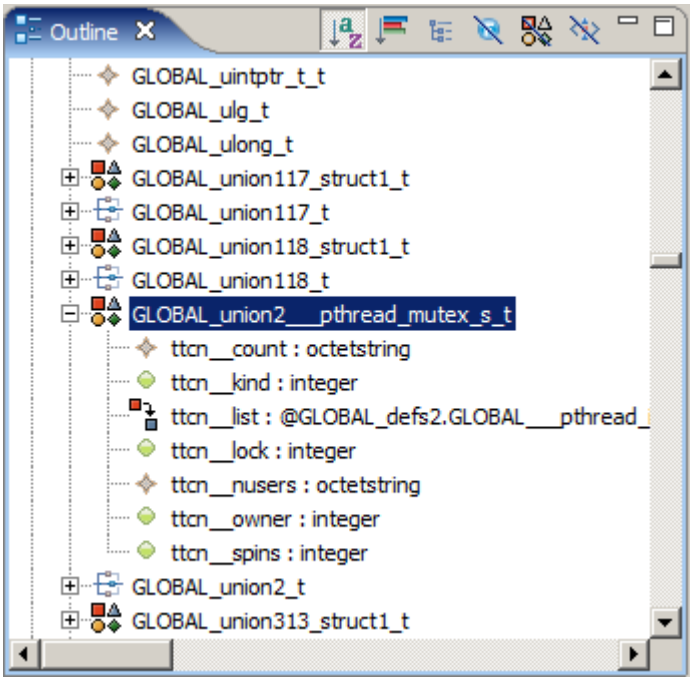


Figure 121. Sorted alphabetically

10.2.2. Categorizing Elements

It is possible to order the outline view t categorizes the elements to be displayed before sorting them. This function is useful if one is only interested records, or functions as this way functions, types, module parameters will be found together in the outline view.

Categorizing of the outline elements is possible with toggling the icon.

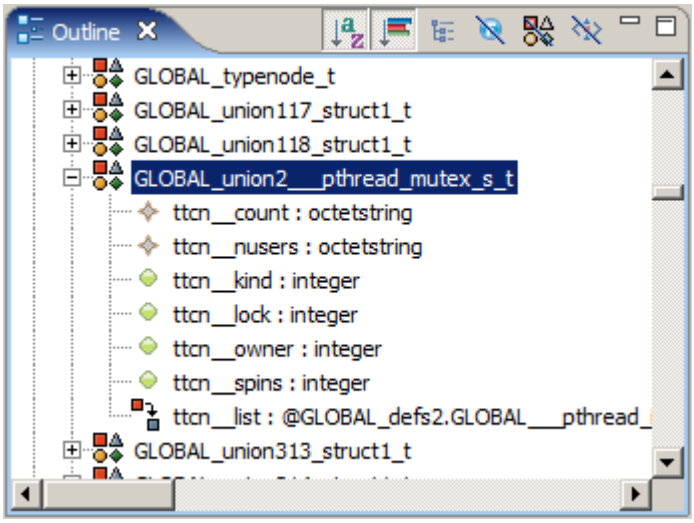


Figure 122. Categorized and sorted alphabetically

10.2.3. Grouping

By default the Outline view does not show the group hierarchies in the module, as the groups do not have any effect on scoping. However they can be used to group semantically similar functions, type definitions etc. To make group hierarchies visible in the Outline view the button can be used.

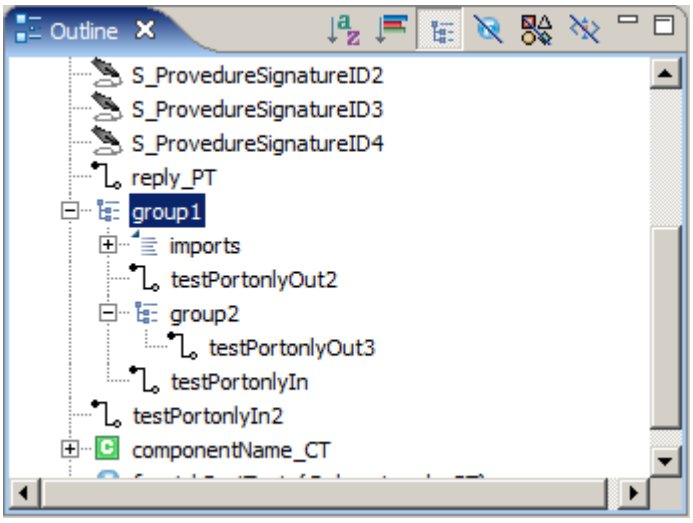


Figure 123. Grouping

10.2.4. Filtering Elements

If there are lots of elements in the Outline view it can be hard to find the appropriate one, so it is possible to filter the elements based on their types, using the filtering buttons in the toolbar. Filtering is additive, more filters can be active at the same time.

Filters for TTCN-3:

Hide functions ( )

Hide templates ( )

Hide types ( )

10.3. Outline View Icons

TTCN-3	ASN.1
import statements	any
group	bitstring
address	bmpstring
altstep	choice
array	embedded_pdv
bitstring	enumeration
boolean	external
character string	generalised time
component	generalstring
constant	graphicstring
constant external	IA5string
default	integer type
enumeration	null type
external function returning a template	numericstring
external function returning a value	objectdescriptor
external function without return statement	opentype
function returning a template	printablestring
function returning a value	relative object identifier
function without return statement	selection
hexadecimal string	sequence
integer	set
module parameter	teletexstring
object identifier	universalstring
octetstring	unrestrictedstring
port	UTCtime
real float	UTF8string
record	videoteststring
record of	visiblestring
referenced	
sequence	
sequence of	
set	
set of	
signature	
template	
template variable, dynamic template	
testcase	
timer	
type	
union	
unversal charstring	
variable	
verdict type	

Figure 124. Outline view icons for TTCN-3 and ASN.1

Pitfalls

NOTE

As long we have to re-parse the whole TTCN-3 and ASN.1 modules on a change the outline view will always have to reinitialize all of its contents. This means, that all structures actually open at such a change will be closed (in fact the old structure will be deleted and a new will be created).

Chapter 11. The Call Hierarchy View

This section presents how TITAN Designer plugin implement the Call Hierarchy View.

During the development is often useful to get an overview of the TTCN-3 function ( ), testcase ( ) or external function ( ) calls. This view help see your functions location in the call tree and you can see witch other functions call yours.

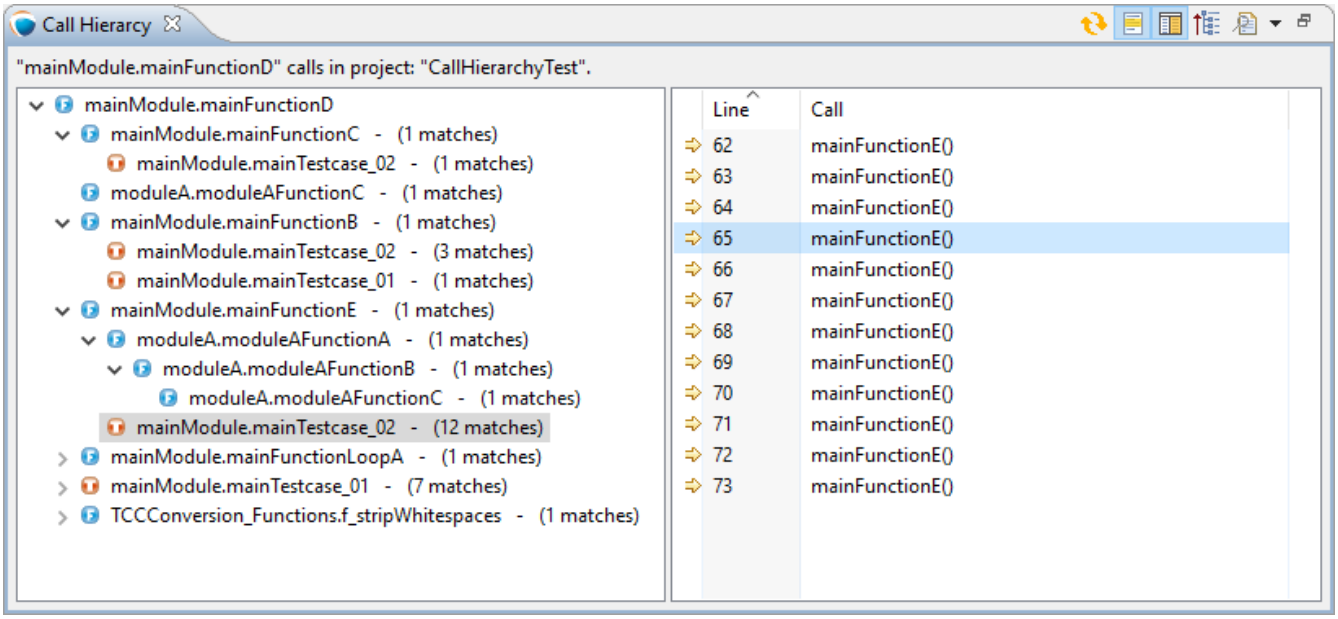


Figure 125. The Call Hierarchy view

You can call the view from the **Window/Show View** menu, from the **right click menu** or with the **CTRL+ALT+H** command. The Call Hierarchy View consists of three main parts. The toolbar, the actual tree view and the current call list.

11.1. The Tree

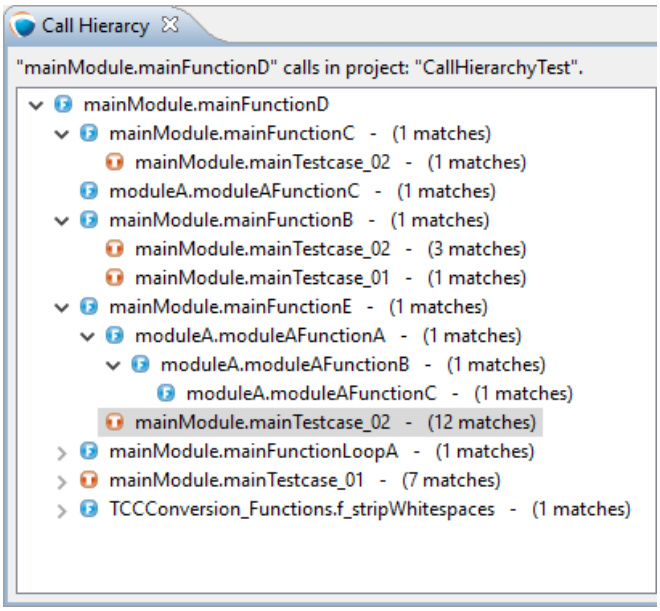


Figure 126. The Call Hierarchy Tree

The root of the tree always represents the searched TTCN-3 function ( ), testcase ( ) or external function ( ). The second level of the tree contains the functions what call the searched (root) function. Near the tree nodes you can see the number of calls.

When you click to a tree node, the editor jump to the function definition and select it automatically (if this option is enabled) and the call list on the right side show the current calls, if the call list is enabled ( ). When you click to the small arrow near a subnode (mainModule.mainFunctionLoopA - (1 matches)), you start a subsearch on the selected node. You can build recursively the part of the tree what you need.

11.2. The Call List

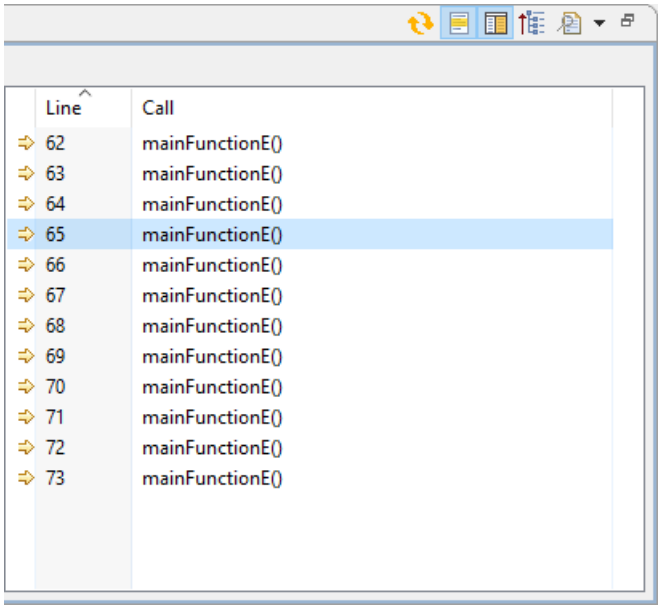


Figure 127. The Call List

Near the tree nodes you can see the number of calls (mainModule.mainFunctionLoopA - (1 matches)), when you click to a tree node the call list show the calls with the row number. When you click to a row in the list, the editor jump to the row of the call. (64 mainFunctionE()) You can switch off the call list in the toolbar. ( )

11.3. The Toolbar

On the top of the view you can see a toolbar with five buttons:

11.3.1. The refresh button



Figure 128. The refresh button

The refresh button ( ) update the current search. (Update the unsaved changes too.)

11.3.2. The auto jump to definition switch

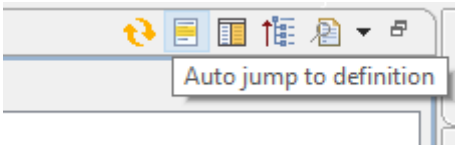


Figure 129. Auto jump to definition switch

When this option ( ) is switched on, the editor jump to the definition of the selected function automatically, when you choose a node in the tree.

11.3.3. The call list switch

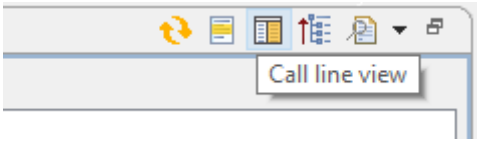


Figure 130. Call list switch.

This switch ( ) show or hide the function call list table.

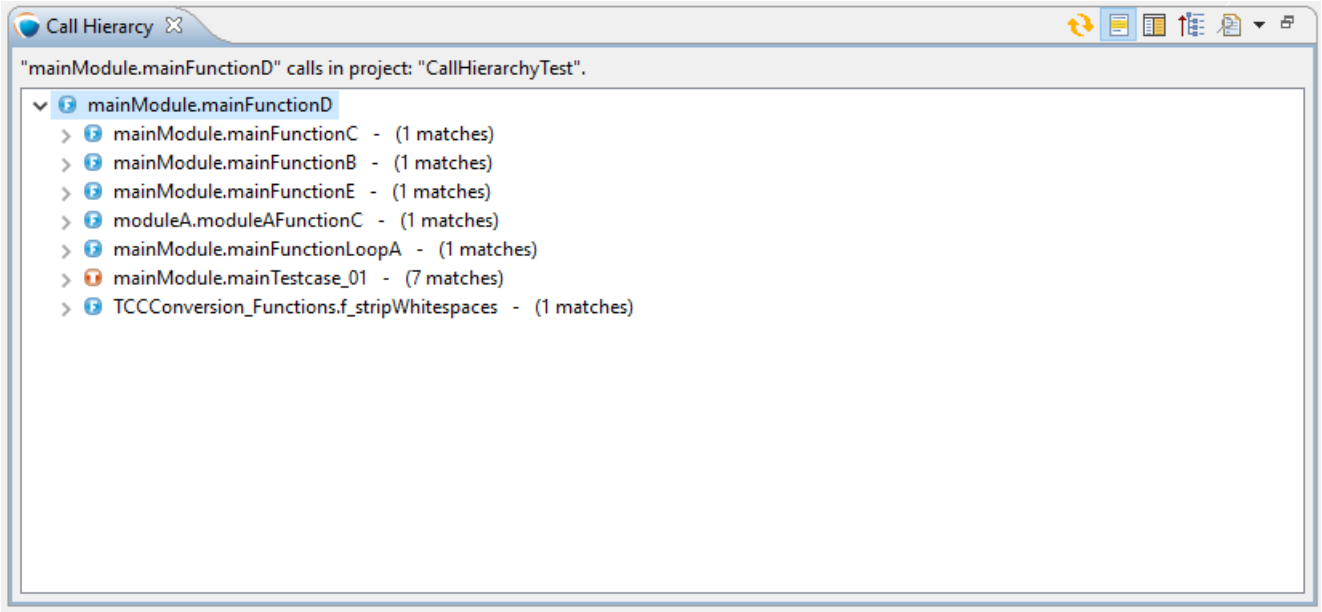


Figure 131. Closed call lines table.

11.3.4. The close all button

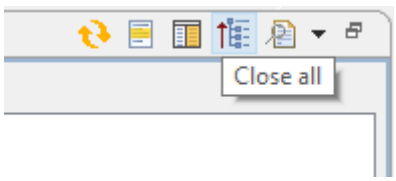


Figure 132. Close tree button.

This button ( ) collapse the call hierarchy tree.

11.3.5. The search history

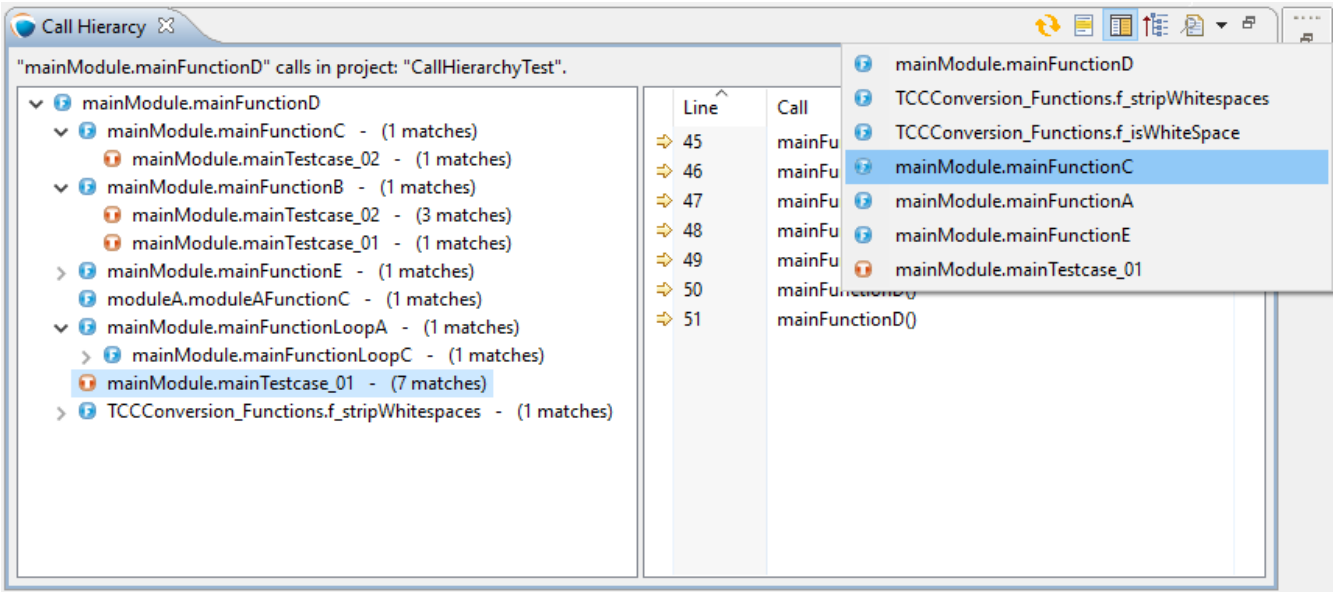


Figure 133. TThe hystory list.

The history droppdown menu ( ) list the prevouse searches and you cen recall these searches.

NOTE	The search discover your unsaved changes too under the tree bulding, updating or under the hystory recall.

Chapter 12. Extensions to the Project Explorer

12.1. Filtering Resources from the View

It is possible to hide excluded resources from the Project explorer view.

To achieve this go to **View Menu** / select **Filters and Customization...** (or **Customize View...**)

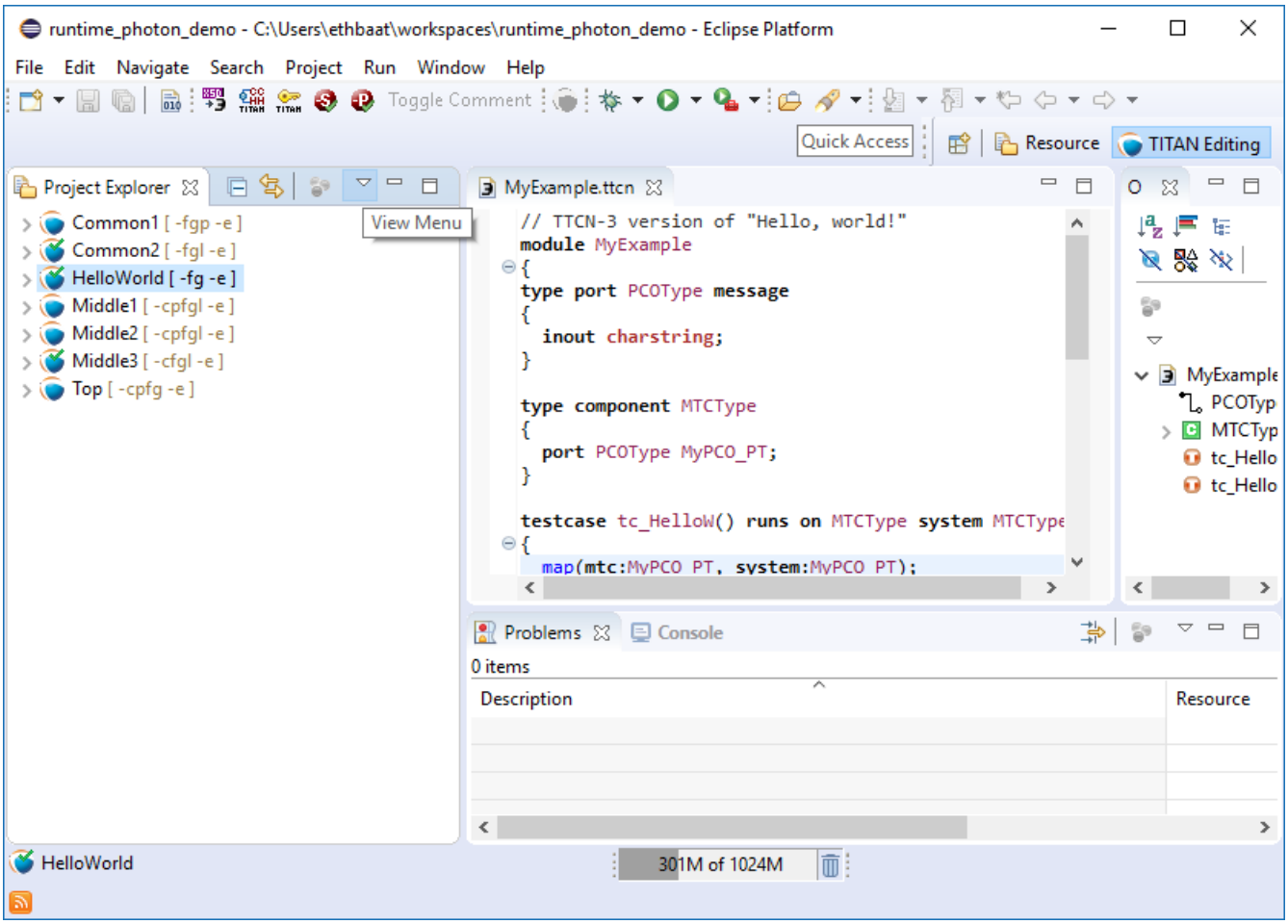


Figure 134. View Menu

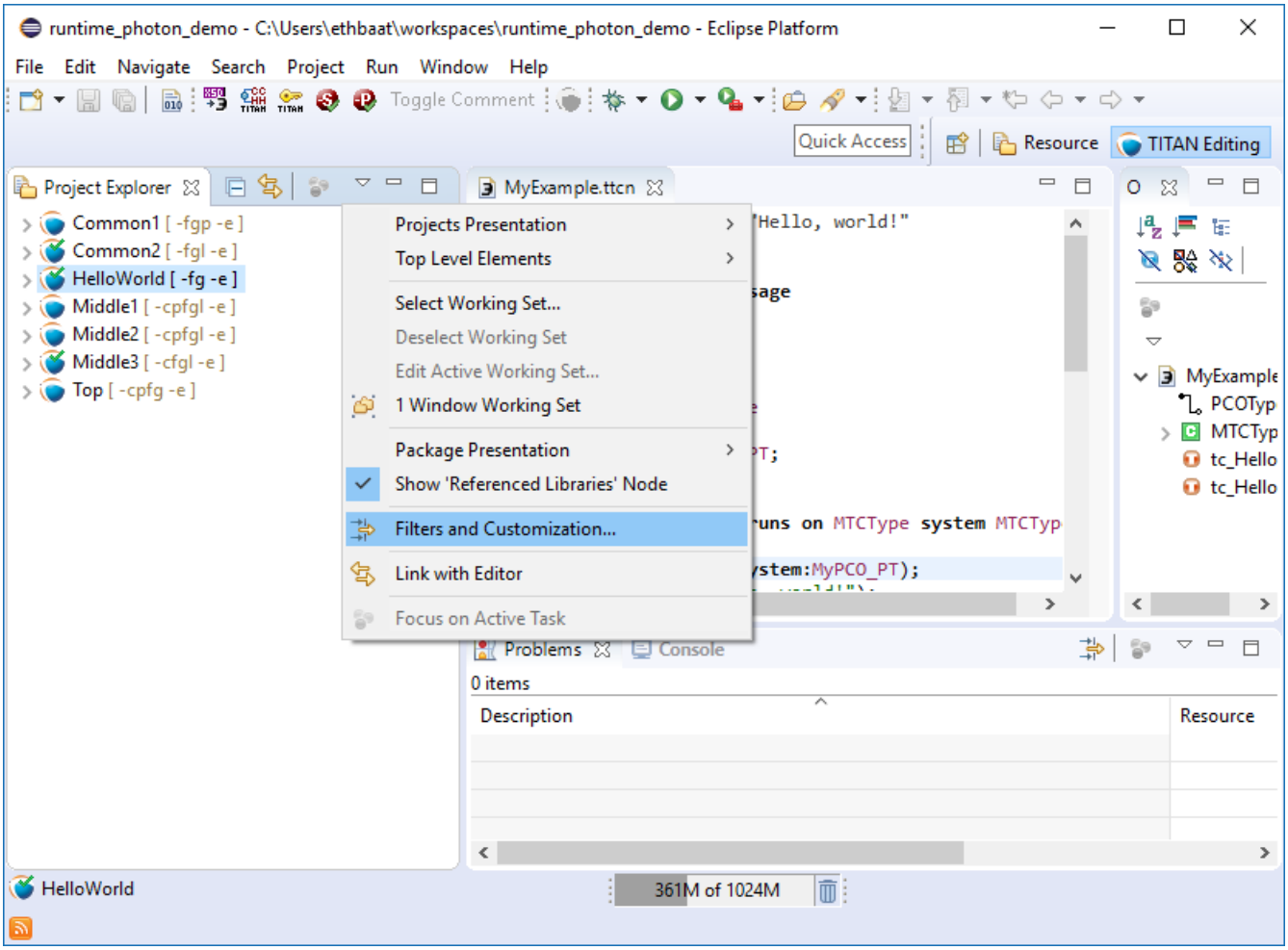


Figure 135. Filters and Customization...

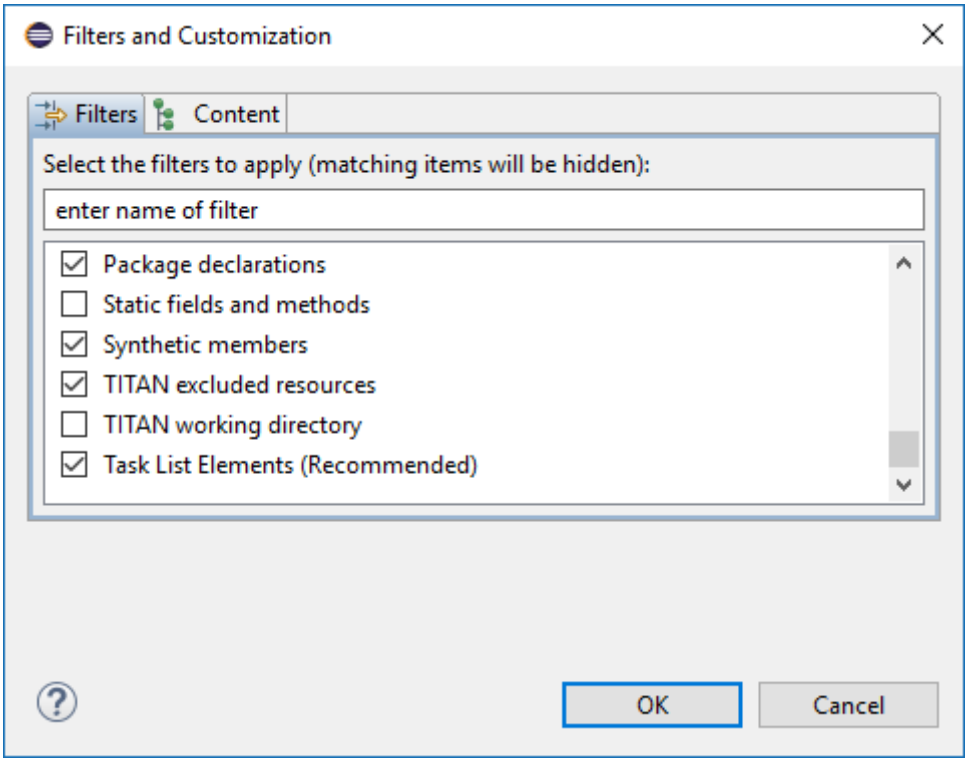


Figure 136. Filters and Customization window

On the **Filters and Customization** window (or **Available Customizations** window) there are two exclusion filters provided by the Designer plug-in:

- TITAN working directory.

  When selected the working directories of the projects will be filtered from the **Project Explorer** view.

- TITAN excluded resources.

  When selected all resources excluded from the build on some way, will be filtered from the view. For more information on how a resource can be excluded from build please refer here.

By default the "TITAN working directory" filter is selected.

Chapter 13. References

- [1] Installation guide for TITAN TTCN-3 Test Executor
- [2] Installation Guide for TITAN Designer and TITAN Executor for the Eclipse IDE
- [3] User Guide for TITAN TTCN-3 Test Executor
- [4] Programmers Technical Reference for TITAN TTCN-3 Test Executor
- [5] Release Notes for TITAN TTCN-3 Test Executor
- [6] TTCN–3 Style Guide 1/0113-FCPCA 101 35
- [7] TTCN–3 Naming Convention ETH/R-04:000010
- [8] Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3.Part 1: Core Language European Telecommunications Standards Institute. ES 201 873-1 Version 4.5.1, April 2013
- [9] Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3.Part 4: TTCN–3 Operational Semantics European Telecommunications Standards Institute. ES 201 873-4 Version 4.4.1, April 2012
- [10] Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3.Part 7: Using ASN.1 with TTCN–3 European Telecommunications Standards Institute. ES 201 873-7 Version 4.5.1, April 2013
- [11] Programmers Technical Reference for the Java side of the TITAN TTCN-3 Test Executor
- [12] Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3.Part 10: TTCN-3 Documentation Comment Specification European Telecommunications Standards Institute. ES 201 873-10 Version 4.5.1, April 2013

Chapter 14. Abbreviations

ASN.1

Abstract Syntax Notation One

GCC

GNU Compiler Collection

GUI

Graphical User Interface

HC

Host Controller

IDE

Integrated Development Environment

IP

Internet Protocol

MC

Main Controller

MTC

Main Test Component

PTC

Parallel Test Component

SUT

System Under Test

TCP

Transmission Control Protocol

TTCN–3

Tree and Tabular Combined Notation version 3 (formerly)Testing and Test Control Notation (new resolution)

TTCNPP

TTCN Preprocessable (file)

TTCNIN

TTCN Includable (file)

URL

Universal Resource Locator