TESSY®

TEST SYSTEM

automated

unit/integration tests

user manual

theory: basic knowledge
tutorial: general handling/practical exercises
reference book: working with TESSY

release 03/23 | revision 42.040 | TESSY v4.2

RAZORCAT

Imprint

Razorcat Development GmbH
Witzlebenplatz 4
Germany, 14057 Berlin
tel: +49 (030) 53 63 57 0
fax: +49 (030) 53 63 57 60
e-mail: support@razorcat.com
internet: http://www.razorcat.com

Windows is a registered trademark of Microsoft. TESSY and CTE are registered trademarks of Razorcat Development GmbH.
All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same is claimed.

Liability exclusion

Razorcat Development GmbH assumes no liability for damage that is caused by improper installation or improper use of the software or the non-observance of the handling instructions described in this manual.

Thanks

Various contents are based on application notes and publications on TESSY written by Frank Büchner, Hitex Development Tools GmbH. We would like to thank Frank for his valuable contribution and commitment in supporting TESSY and spotlighting functionalities and features.

Contents

Preface **xiii**

About TESSY . . . xv
How to use this manual . . . xvi
Subject matter . . . xvi
Helpers . . . xviii
Various Boxes - Important or extra information and warnings . . . xx
Safety Manual . . . xxi
Core workflow and registration for safety information . . . xxi
Verification and certification of TESSY . . . xxii
Instrumentation for coverage measurement . . . xxiii
Adaptation to target environment . . . xxiii
Command line interface (CLI) . . . xxiv
Operating limits . . . xxiv
New features in TESSY 4.0 . . . xxvi
C++ . . . xxvi
Software variant management . . . xxvi
Optimizations in command line mode . . . xxvii
Support for test driven development . . . xxvii
UUID for all TESSY objects . . . xxvii
Auto-reuse in command line execution . . . xxviii
Excluding individual tests . . . xxviii
New features in TESSY 4.1 . . . xxx
Static analysis and test case quality metrics . . . xxx
Fault injection . . . xxxi
Script perspective . . . xxxii
New Classification Tree editor (CTE) . . . xxxiii
Review of test changes . . . xxxiii
Arithmetic expressions as test data . . . xxxiv
Stimulating and measuring hardware signals . . . xxxiv
New features in TESSY 4.2 . . . xxxv
CTE extensions . . . xxxv

Component tests for C++ modules . . . xxxvi
Call pair coverage . . . xxxvi
Testing effort estimation and tracking . . . xxxvi
Automated requirement reviews . . . xxxvii
Improved error handling . . . xxxviii

1 Installation and registration 1
1.1 Technical requirements . . . 2
1.2 Setup . . . 2
1.3 Installation . . . 3
1.4 Registration . . . 6
1.4.1 Requesting a license key . . . 6
1.4.2 Registering the license . . . 9
1.4.3 The License Manager (FLM) . . . 15
1.5 Uninstallation . . . 16
1.6 Using a license without connection to the license server (FLS) . . . 17
1.6.1 Checking-out the license for use on your local computer . . . 20
1.6.2 Using a license on a computer with no connection to the license server 21

2 Migrating from TESSY 3.x to 4.x 22
2.1 Importing previous projects . . . 22

3 Theory: Basic knowledge 24
3.1 Unit testing of embedded software . . . 25
3.1.1 Standards that require testing . . . 25
3.1.2 About unit testing . . . 25
3.1.3 Considerations for unit testing . . . 27
3.1.4 Methods for unit testing . . . 29
3.1.5 Conclusion . . . 31
3.2 The Classification Tree Method (CTM) . . . 32
3.2.1 General . . . 32
3.2.2 Steps to take . . . 33
3.2.3 Example is_value_in_range . . . 39

4 Tutorial: General handling 52
4.1 Creating databases and working with the file system . . . 53
4.1.1 Creating a project database . . . 54
4.1.2 Creating, importing, cloning, editing, deleting a project . . . 60
4.1.3 Creating a template project . . . 61
4.1.4 Moving the project directory . . . 62

4.1.5 Handling with equally named projects . . . 62
4.1.6 Using a specific environment setting . . . 64
4.1.7 Updating the database . . . 64
4.2 Understanding the graphical user interface . . . 66
4.2.1 Menu bar . . . 67
4.2.2 Tool bar . . . 67
4.2.3 Perspectives and perspective (tool) bar . . . 67
4.2.4 Views . . . 68
4.2.5 Status bar . . . 72
4.3 Using the context menu and shortcuts . . . 73
4.3.1 Context menu . . . 73
4.3.2 Shortcuts . . . 73

5 Tutorial: Practical exercises 76
5.1 Quickstart 1: Unit test exercise is_value_in_range . . . 78
5.1.1 Creating a test project . . . 79
5.1.2 Specifying the target environment . . . 81
5.1.3 Adding the test object and analyzing the C-source file . . . 83
5.1.4 Editing the test object interface . . . 85
5.1.5 Designing test cases . . . 86
5.1.6 Adding test cases and test steps . . . 86
5.1.7 Entering test data . . . 88
5.1.8 Executing the test . . . 91
5.1.9 Repeating the test run with coverage instrumentation . . . 93
5.1.10 Analyzing the coverage . . . 94
5.1.11 Creating a Test Details Report . . . 98
5.1.12 Repeating the test run with requirements . . . 102
5.1.13 Reusing a test object with a changed interface . . . 114
5.2 Quickstart 2: The Classification Tree Editor (CTE) . . . 123
5.2.1 The CTE tree elements . . . 124
5.2.2 Working with the CTE . . . 126
5.2.3 Entering test data . . . 126
5.2.4 Creating test cases . . . 128
5.3 Quickstart 3: Component test exercise interior_light . . . 136
5.3.1 Creating the test project . . . 138
5.3.2 The heartbeat function . . . 140
5.3.3 Preparing the test interface . . . 143
5.3.4 Adding test cases . . . 145
5.3.5 Editing data . . . 146

5.3.6 Configuring the work tasks . . . 148
5.3.7 Designing scenarios . . . 148
5.3.8 Executing the scenarios . . . 154
5.3.9 Evaluating the scenarios . . . 155
5.4 Quickstart 4: Exercise C++ . . . 156
5.5 Quickstart 5: Test driven development (TDD) . . . 160

6 Reference book: Working with TESSY 166
6.1 Menu Bar Entries: Setting up the basics . . . 171
6.1.1 File menu . . . 171
6.1.2 Window menu . . . 172
6.1.3 Static Analysis Settings . . . 177
6.1.4 Coverage Settings . . . 178
6.1.5 Metrics Settings . . . 179
6.1.6 Interface dictionary . . . 180
6.1.7 Help menu . . . 182
6.2 Overview perspective: Organizing the test . . . 183
6.2.1 Structure of the Overview perspective . . . 183
6.2.2 Test Project view . . . 184
6.2.3 Properties view . . . 228
6.2.4 Requirements Coverage view . . . 238
6.2.5 Test Items view . . . 238
6.2.6 Test Results view . . . 248
6.2.7 Evaluation Macros view . . . 249
6.2.8 Console view . . . 249
6.2.9 Suspicious Elements view . . . 251
6.2.10 Problems view . . . 251
6.2.11 Variants view . . . 252
6.3 C/C++: Editing the C-source . . . 256
6.3.1 Opening the C/C++ perspective . . . 256
6.3.2 Structure of the C/C++ perspective . . . 257
6.3.3 Editor view . . . 258
6.3.4 Project Explorer view . . . 260
6.3.5 Outline view . . . 260
6.3.6 Properties view . . . 261
6.3.7 Console view . . . 262
6.4 Requirement management . . . 263
6.4.1 Structure of the Requirement Management perspective . . . 264
6.4.2 RQMT Explorer view . . . 265

6.4.3 Requirements List view . . . 274
6.4.4 Requirement Editor view . . . 275
6.4.5 Validation Matrix view / VxV Matrix view . . . 278
6.4.6 Test Means view . . . 279
6.4.7 Link Matrix view . . . 280
6.4.8 Suspicious Elements view . . . 285
6.4.9 Attached Files view . . . 290
6.4.10 Attributes view . . . 291
6.4.11 History view . . . 294
6.4.12 Differences view / Reviewing changes . . . 295
6.4.13 Related Elements view . . . 296
6.4.14 Problems view . . . 298
6.4.15 Document Preview . . . 298
6.4.16 Requirements Coverage view . . . 301
6.5 TEE: Configuring the test environment . . . 306
6.5.1 Starting the TEE . . . 307
6.5.2 Structure of the TEE . . . 308
6.5.3 Icons of the tool bar . . . 310
6.5.4 Status indicators . . . 310
6.5.5 Viewing all available configurations . . . 312
6.5.6 Configuration files . . . 315
6.5.7 Customizing the configurations . . . 315
6.6 THAI: TESSY Hardware Adapter Interface . . . 323
6.6.1 The THAI Configuration file . . . 324
6.6.2 Environment Editor (TEE) Settings for THAI functionality . . . 325
6.6.3 Signals within the interface . . . 327
6.6.4 Entering test data for signals . . . 328
6.7 TIE: Preparing the test interface . . . 329
6.7.1 Structure of the TIE perspective . . . 330
6.7.2 Test Project view . . . 330
6.7.3 Properties view . . . 330
6.7.4 Interface view . . . 331
6.7.5 Plot Definitions view . . . 351
6.8 CTE: Designing the test cases . . . 356
6.8.1 The basic idea . . . 356
6.8.2 Structure of the CTE perspective . . . 357
6.8.3 Test Project view . . . 357
6.8.4 Properties view . . . 357
6.8.5 Outline view . . . 358

6.8.6 Classification Tree editor . . . 358
6.8.7 Test Data view . . . 375
6.8.8 Dependencies in CTE . . . 381
6.9 TDE: Entering test data . . . 387
6.9.1 Structure of the TDE perspective . . . 387
6.9.2 Test Project view . . . 389
6.9.3 Test Results view . . . 389
6.9.4 Evaluation Macros view . . . 389
6.9.5 Test Items view . . . 389
6.9.6 Properties view . . . 390
6.9.7 Test Data view . . . 391
6.9.8 Test Definition view . . . 415
6.9.9 Call Trace view . . . 415
6.9.10 Declarations/Definitions view . . . 417
6.9.11 Prolog/Epilog view . . . 418
6.9.12 Stub Functions view . . . 427
6.9.13 Usercode Outline view . . . 432
6.9.14 Plots view . . . 434
6.9.15 Plot Definitions view . . . 435
6.10 Script Editor: Textual editing of test cases . . . 436
6.10.1 Editing test objects, test cases and test steps . . . 436
6.10.2 Structure of the Script Editor perspective . . . 437
6.10.3 Script Editor related Icons of the main tool bar . . . 438
6.10.4 Script states . . . 438
6.10.5 The Script Editor Outline view . . . 439
6.10.6 Synchronization with the internal model . . . 439
6.10.7 Merging script contents . . . 440
6.10.8 Importing and exporting script contents . . . 440
6.11 CV: Analyzing the coverage . . . 441
6.11.1 Structure of the CV perspective . . . 442
6.11.2 Instrumentation for coverage measurements . . . 443
6.11.3 Test Project view . . . 445
6.11.4 Called Functions view/Code view . . . 446
6.11.5 Flow Chart view . . . 447
6.11.6 Fault injection . . . 456
6.11.7 Statement (C0) Coverage view . . . 456
6.11.8 Branch (C1) Coverage view . . . 458
6.11.9 Decision Coverage view . . . 459
6.11.10 MC/DC Coverage view . . . 459

6.11.11 MCC Coverage view . . . 461
6.11.12 Call Pair Coverage view . . . 461
6.11.13 Coverage Report views . . . 462
6.12 IDA: Assigning interface data . . . 463
6.12.1 Structure of the IDA perspective . . . 464
6.12.2 Status indicators . . . 464
6.12.3 Test Project view . . . 465
6.12.4 Properties view . . . 465
6.12.5 Compare view . . . 465
6.13 SCE: Component testing . . . 469
6.13.1 Creating component tests . . . 470
6.13.2 Preparing the test interface . . . 472
6.13.3 Configuring the work tasks . . . 473
6.13.4 Designing the test cases . . . 475
6.13.5 Editing scenarios . . . 476
6.13.6 Executing the scenarios . . . 482
6.14 Fault injection . . . 483
6.14.1 Managing fault injections in the Coverage Viewer . . . 483
6.14.2 Creating fault injection test cases . . . 484
6.14.3 Creating and editing fault injections in the Coverage Viewer . . . 486
6.14.4 Fault injections within the report . . . 488
6.15 Backup, restore, version control . . . 489
6.15.1 Backup . . . 489
6.15.2 Restore . . . 491
6.15.3 Version control . . . 493
6.16 Command line interface . . . 495
6.16.1 Starting TESSY in headless mode . . . 495
6.16.2 Invoking “tessycmd.exe” . . . 496
6.16.3 Usage of “tessycmd.exe” . . . 497
6.16.4 Commands . . . 498
6.16.5 Execution and result evaluation . . . 498
6.16.6 Headless operation . . . 499
6.16.7 Example: DOS script . . . 500

7 Troubleshooting 502
7.1 Contacting the TESSY support . . . 503
7.2 Enhanced error handling . . . 507
7.2.1 Problems Log dialog . . . 507
7.2.2 Problems view . . . 509

7.2.3 Opening external problem logs using the Help menu 510
7.3 Solutions for common problems 512
7.3.1 TESSY does not start or gives errors when starting 512
7.3.2 TESSY gives errors when quitted 513
7.3.3 License server does not start or gives errors 514
7.3.4 Working with constant variables 515
7.3.5 Dealing with too long project paths 519

Appendix **521**

A Abbreviations **522**

B Glossary **524**

C List of Figures **529**

D List of Tables **542**

Index **546**

Preface

About TESSY **xv**

How to use this manual **xvi**

Subject matter . . . xvi

Helpers . . . xviii

Various Boxes - Important or extra information and warnings . . . xx

Safety Manual **xxi**

Core workflow and registration for safety information . . . xxi

Verification and certification of TESSY . . . xxii

Instrumentation for coverage measurement . . . xxiii

Adaptation to target environment . . . xxiii

Command line interface (CLI) . . . xxiv

Operating limits . . . xxiv

New features in TESSY 4.0 **xxvi**

C++ . . . xxvi

Software variant management . . . xxvi

Optimizations in command line mode . . . xxvii

Support for test driven development . . . xxvii

UUID for all TESSY objects . . . xxvii

Auto-reuse in command line execution . . . xxviii

Excluding individual tests . . . xxviii

New features in TESSY 4.1 **xxx**

Static analysis and test case quality metrics . . . xxx

Fault injection . . . xxxi

Script perspective . . . xxxii

New Classification Tree editor (CTE) . . . xxxiii

Review of test changes . . . xxxiii

Arithmetic expressions as test data . . . xxxiv

Stimulating and measuring hardware signals xxxiv
New features in TESSY 4.2 **xxxv**
CTE extensions xxxv
Component tests for C++ modules xxxvi
Call pair coverage xxxvi
Testing effort estimation and tracking xxxvi
Automated requirement reviews xxxvii
Improved error handling xxxviii

About TESSY

The test system TESSY was developed by the Research and Technology Group of Daimler. The former developers of the method and tool at Daimler were:
Klaus Grimm
Matthias Grochtmann
Roman Pitschinetz
Joachim Wegener

TESSY has been well-tried in practice at Daimler and is since applied successfully. TESSY is commercially available since spring 2000 and is further developed by Razorcat Development GmbH.

TESSY offers an integrated graphic user interface conducting you comfortably through the unit test. There are special tools for every testing activity as well as for all organizational and management tasks.

Dynamic testing is indispensable when testing a software system. Today, up to 80% of the development time and costs go into unit and integration testing. It is therefore of urgent necessity to automate testing processes in order to minimize required time and costs for developing high-quality products. The test system TESSY automates the whole test cycle; unit testing for programs in C/C++ is supported in all test phases. The system also takes care of the complete test organization as well as test management, including requirements coverage measurement and traceability.

How to use this manual

The TESSY User Manual provides detailed information about the Installation and registration of TESSY, Theory: Basic knowledge about testing, Tutorial: General handling and Reference book: Working with TESSY. (Please study the list under Subject matter for more details about the different chapters in this manual.)

There is also a chapter Tutorial: Practical exercises containing five basic examples of possible ways to operate with TESSY. We strongly recommend to work through these practical exercises as they are also a perfect quickstart to TESSY!

Apply for our e-mail list if you want to be informed of a new version of TESSY manual by sending an e-mail to support@razorcat.com.

Refer as well to our detailed application notes regarding compiler/target settings and other specific themes that are available in the Help menu of TESSY (“Help” > “Documentation”).

Find some videos about TESSY features as well as support videos on our website https://www.razorcat.com/en/tessy-videos.html.

Subject matter

The structure of the manual guides you through working with TESSY from the start to the specific activities possible. In order:

	Section	Matter
	Preface	Describes New features in TESSY 4.0 and New features in TESSY 4.1, also contains the Safety Manual.
1	Installation and registration	Lists all technical requirements to work with TESSY and describes how to install the software.

continue next page

Section	Matter
2 Migrating from TESSY 3.x to 4.x	Lists the changed and new functions and handling within the new version. This will help you when switching from TESSY version 3.x to the new TESSY 4.x.
3 Theory: Basic knowledge	Contains a brief introduction about unit testing with TESSY and the classification tree method (CTM).
4 Tutorial: General handling	Explains the workflow of Creating databases and working with the file system. Check this section carefully to know how to handle your project data! The TESSY interface and basic handling is explained in the following sections Understanding the graphical user interface and Using the context menu and shortcuts.
5 Tutorial: Practical exercises	**In this chapter you will get to know TESSY with the help of exercises that are prepared to follow easily though most of the TESSY functions:** Quickstart 1: Unit test exercise is_value_in_range is a very basic example to give you a fast introduction. Quickstart 2: The Classification Tree Editor (CTE) gives a short and easy introduction of handling with the Classification Tree Editor (CTE). It continues the Quickstart 1: Unit test exercise is_value_in_range. Quickstart 3: Component test exercise interior_light gives you a complete overview about working with TESSY, including all main functions of unit testing, e.g. requirement management. Quickstart 4: Exercise C++ gives gives a short and easy introduction of the handling of TESSY with a C++ source file. Quickstart 5: Test driven development (TDD) gives a short introduction into test driven development with TESSY. You should be familiar with the overall handling of TESSY before doing this exercise. **We strongly recommend to work though the practical exercises as a quickstart to TESSY!** Learning by doing is much easier than learning by just reading. So exercise first and then check the detailed information in the corresponding section of chapter "Reference book: Working with TESSY".

continue next page

Section	Matter
6 Reference book: Working with TESSY	**This chapter explains in detail the unit test activities possible with TESSY.** You will notice that the headlines of the sections follow the actions taken during a test. TESSY provides different editors and windows (“perspectives” and “views”) for different configurations and steps taken during and after a test. You will find the name of the perspective or view as well as the description of the step within the headline, e.g. 6.8 CTE: Designing the test cases. Therefore, if you need help at some point, ask either “How do I handle …?” or “Where am I?” and follow the headlines. **Important:** Read the Tutorial: General handling first, because basic functions of each editor are explained there.
7 Troubleshooting	Contains information of Solutions for common problems and how to get in touch with the TESSY support if needed.
Appendix	Contains a List of Figures, a List of Tables, a list of used Abbreviations as well as definitions of used terms in the Glossary. Please check the glossary when you need some explanations of terms! The Index in the very end of this manual provides the positions of your theme of interest with the help of alphabetically listed keywords.

Table 0.1: Where to find - matters of the several parts of the TESSY manual

Helpers

The sidearrow shows where to find information and references.

- The **Index** in the very end of this manual helps you finding topics with the help of keyword.
- Various information is clearly represented within tables, e.g. icons and indicators (symbols of the interface) and their meanings. For a fast access to all tables consult the **List of Tables** in the appendix of this manual.
- Figures are used to demonstrate described information. You may as well check the **List of Figures** in the appendix to find those figures.

- **Cross references** as well as the content directory are active links (blue colored), which makes it easy to switch to the referenced chapter or section.

Font characters and signs

To help you to work with this manual, different font characters and signs are used to mark specific information:

Font character / Sign	**Used for**	**Example**
→	instructions you are supposed to follow immediately	→ Open the TESSY interface.
>	navigation through a menu	→ Select “File” > “Open…”
[…]	variable	→ Switch to “`[project root]\tessy`”
bold	accentuation	**Important:**
`typewriter italic`	input (you need to type this information) or output (message from system)	Enter *`Test Example`*.
“`typewriter in quotes`”	path of data	“`C:\Program Files\Razorcat`”
“quotes”	indicate keys, buttons, etc.	→ Select “File” > “Open…”
Ctrl+C	Keyboard characters are not marked on the assumption that they are commonly known.	Ctrl+C for pressing control and c

Table 0.2: Font characters

Various Boxes - Important or extra information and warnings

General information:

Gray bordered information boxes
provide further information and explanations for the respective issues and operations to be executed. The information will give you an overview, while the practical part is explained in the following section.

Information about the handling of TESSY:

Important: You urgently need to know this for operating correctly!

Warning: There might be some damages to your data if you do not operate correctly! Please follow instructions carefully.

A light bulb provides hints, references and additional information on handling with TESSY for better usability.

Safety Manual

Core workflow and registration for safety information

Important: If you work with TESSY in a safety-relevant environment, please read this chapter carefully and register for our safety customer e-mail-list to be informed about known problems as described below!

TESSY can be used for testing of safety-relevant software. Therefore, the core workflow of TESSY as well as the release and test process of the TESSY product has been certified according to ISO 26262-08:2018 and IEC 61508-3:2010. (The second edition of ISO 26262 was included in the certification during the re-certification of TESSY 4.2 by TÜV SÜD Rail GmbH.) In the course of the re-certification of TESSY 4.1 by TÜV SÜD Rail GmbH the certification was extended to also cover EN 50128 and IEC 62304. Our quality management system ensures proper handling of all development processes for the TESSY product and constantly improves all procedures concerning quality and safety.

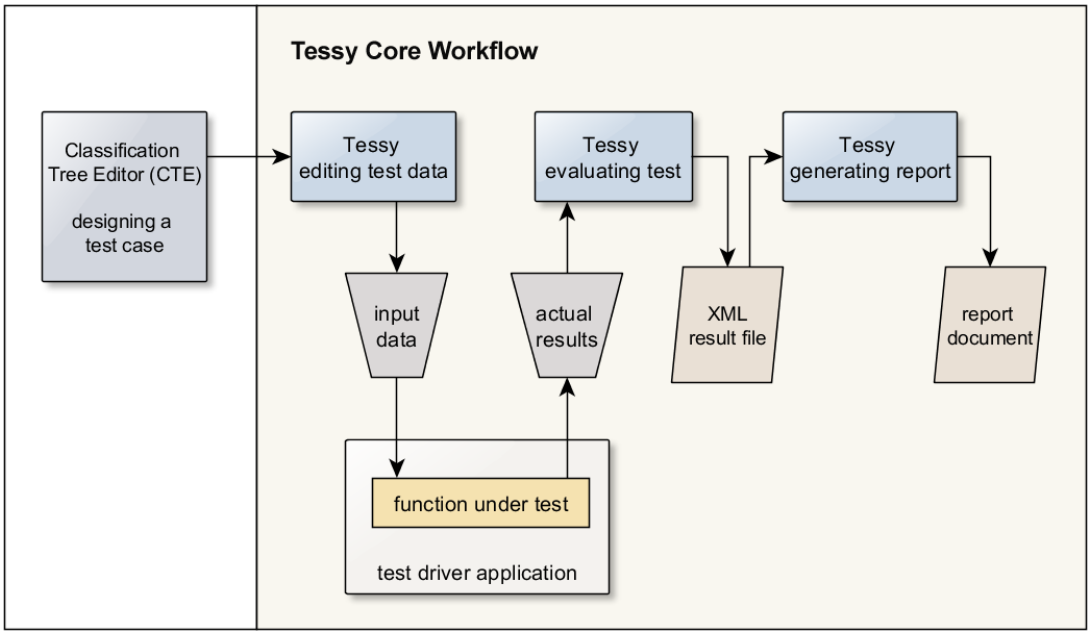


Figure 0.1: Core workflow of TESSY

The figure above shows the core workflow of TESSY that is fully automated and subject to tool qualification. All other tool capabilities like editing or environment and interface settings are additional features out of scope of the tool qualification. The core workflow of TESSY

has been certified according to ISO 26262:2011 and IEC 61508:2010 as well as EN 50128 and IEC 62304. Starting from editing of test data, the core workflow covers test execution, evaluation of test results and report generation. Additionally, the coverage measurements have been verified according to our certified safety plan. Please note, that the Classification Tree Editor (CTE) which covers test preparation activities is not part of the certified core workflow of TESSY.

Safety-relevant problems arising in released TESSY versions will be reported (once they are detected) and regarded closely to have them fixed as fast as possible. If you work with TESSY in a safety-related environment, **please register for our safety customer e-mail-list**:

- → Send an e-mail to support@razorcat.com
- → Topic: “Known problems requested”
- → Content: your contract data

You will be informed about current and newly arising “known problems” as well as work-arounds.

Verification and certification of TESSY

The “Tool Qualification Pack” (TQP) is an additional purchase of documents and tests for TESSY, provided as baseline for the certification process in order to qualify TESSY as a software verification tool according to DO-178B/C.
Please contact via support@razorcat.com.

Additionally, TESSY has been qualified by the German certification authority TÜV SÜD Rail GmbH as a testing tool for usage in safety-related software development according to ISO 26262 and IEC 61508. TESSY was also evaluated against IEC 62304 (medical technology) and EN 50128 (railway technology). EN 50128:2011 is an application standard derived from IEC 61508. TESSY was classified as a T2 offline tool in accordance with EN 50128:2011. The TÜV certificate and a certification report is available on http://www.razorcat.com.

The TQPack contains tests for ANSI-C compliant source code using the GNU GCC compiler that is part of the TESSY installation. Using an embedded compiler/debugger for a specific microcontroller requires adaptation of the TQPack for this specific target environment. This can be provided as an engineering service by Razorcat.

Instrumentation for coverage measurement

When executing tests using coverage measurements, it is recommended that all tests are executed once with and once without coverage instrumentation. TESSY uses a copy of the original source file when creating the test application. This copy of the source file will be instrumented for coverage measurements. Usually both test runs yield the same result, indicating that the instrumentation did not change the functional behavior of the test objects.

Please note, that the source code will be instrumented even if no coverage measurement has been selected in the following cases:

- When using the call trace feature
- When using static local variables

Some extra code will be added at the end of the copied source file in the following cases:

- When testing static functions
- When using static global variables

Please keep this behavior in mind when preparing and executing tests with TESSY.

Adaptation to target environment

When running tests on a specific target platform, adaptations of compiler options and target debugger settings may be needed within the respective target environment. The verification of the TESSY core workflow covers tests conducted on a Windows host system using the GNU GCC compiler. In order to verify the transmission of test data and expected results to and from the target device, there are tests available that may be executed using the adapted target environment. These tests check the communication layers of the test driver application.

For details on how to run these tests refer to the application note “048 Using Test Driver Communication Tests.pdf” within the TESSY installation directory.

It is recommended to run these tests with your specific compiler/target environment after initial project setup or after any changes of the environment settings.

Command line interface (CLI)

The command line execution mode of TESSY is designed for usage on continuous integration platforms like e.g. Jenkins. Therefore it is desired that TESSY does an auto-reuse of existing tests on interface changes and tries to execute as many tests as possible with newer versions of the source code being tested when running in CLI mode.

As a result, the tests executed in CLI mode may be run with test data that do not match with the source code being tested (e.g. with uninitialized new variables) which could hide existing or newly introduced errors within that source code. It is recommended to regularly check that the existing tests still match with the interface of the software being tested.

Operating limits

TESSY is constructed for usage as a unit testing tool in order to verify the functional correctness of the function under test. The following restrictions and prerequisites for TESSY apply:

- The source code to be tested shall be compilable without errors and warnings by the compiler of the respective microcontroller target. TESSY may fail analyzing the interface of the module to be tested, if there are syntactical errors within the source code.
- TESSY does not check any runtime behavior or timing constraints of the function under test.
- The test execution on the target system highly depends on the correct configuration of the target device itself, the correct compiler/linker settings within the TESSY environment and other target device related settings within TESSY (if applicable). Any predefined setup of the TESSY tool for the supported devices requires manual review by the user to ensure proper operation of the unit testing execution.
- The usage of compiler specific keywords and compiler command line settings may require additional tests for tool qualification. Correct operation of the TESSY toolset with respect to the Qualification Test Suite (QTS) test results is only provided for ANSI compliant C code.

Since TESSY 4.x the test driver code will be generated and attached at the end of (a copy of) each source file. The following restrictions apply:

- All types used within usercode must be available within the source file of the respective test object.

- When using usercode definitions/declarations on module level, all used types must be available within all source files of the module.

For backward compatibility, you can disable the usage of the new CLANG parser and test driver generation by setting the attribute “Enable CLANG” to “false” within the project configuration using the environment editor TEE.

New features in TESSY 4.0

Note the chapter Migrating from TESSY 3.x to 4.x and check as well the Tutorial: Practical exercises to learn about TESSY´s new features!

C++

TESSY now integrates the CLANG parser for analysis of C++ source files and provides a new test driver generator to support testing of all C++ features including templates and derived classes. The interface and constructors of each C++ class are available for testing and the TDE provides convenient editing of test data. Powerful reuse of test data by assigning the new to the old C++ interface elements allows easy regression testing.

Warning: If you open an existing module created with TESSY 3.2 or earlier, you will be warned that this module will be converted to the new interface database format. Please refer to Analyzing modules for more information about the available options.

Please note that Component testing of C++ modules is not yet supported.

Software variant management

Testing different but related configurations of software is now supported by the variant management of modules and test cases: Starting from a base module with test cases for all variants you can derive sub modules for each individual variant configuration. Such variant modules can overwrite or delete the base test cases or provide additional test cases specific for the given variant. Changes to the base module and tests will be propagated to all sub modules automatically.

Refer to Creating variant modules and Test cases and steps inherited from a variant module for details about creating and editing test cases within variant modules.

Optimizations in command line mode

The test execution (building the test driver and execution on the target) can now run in parallel depending on the number of available cores of the host PC. Also the restore of TMB files now runs in parallel. These optimizations greatly reduce the time required for automated test runs, especially on continuous integration servers (e.g. Jenkins).

Support for test driven development

Support for test driven development has been added: You can now add test objects to modules without having a source file in place. For such test objects you can also add variables for values being used for calculations by the test object according to the specification. In this way, you can prepare your tests before any code is available. Later when the first code is implemented, all tests can be assigned to the real implementation modules using the normal IDA assign operation.

Refer to Quickstart 5: Test driven development (TDD) for an example on how to use this feature.

UUID for all TESSY objects

All the following model elements of TESSY will get a UUID assigned to uniquely identify them even after save and restore via TMB files on different computers:

- Test collections and folders
- Modules
- Test objects
- Test cases and test steps

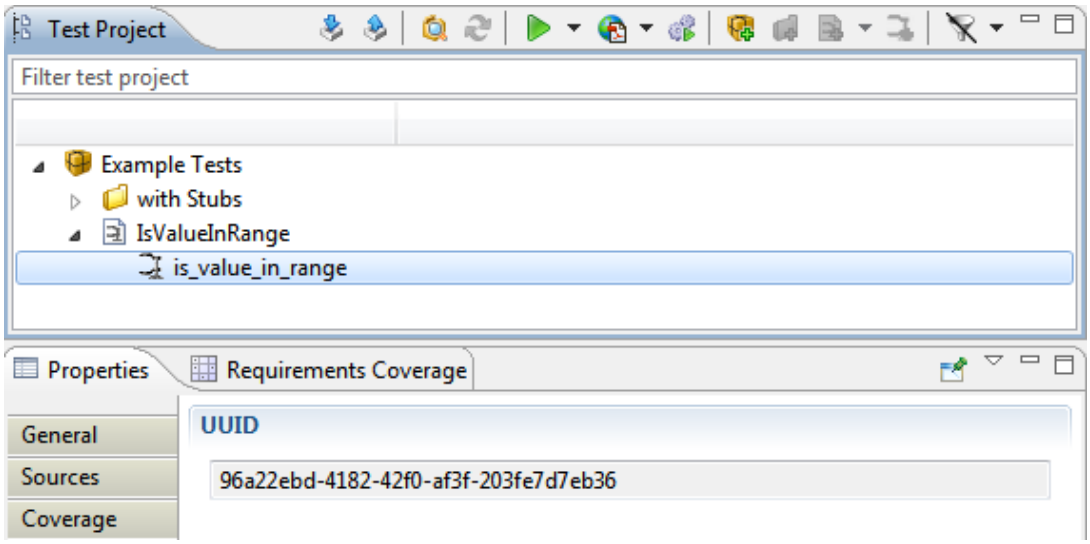


Figure 0.2: UUID property of TESSY objects

Existing projects will be updated automatically when opened using TESSY 4.0.

Auto-reuse in command line execution

Enhanced auto-reuse in command line execution mode provides more stable regression test results in case of slightly changed interfaces as follows:

- New unused struct or union members
- New unused defines
- Changed called function hierarchy
- Removed interface elements or defines

All those interface changes will be ignored when opening modules in command line mode.

Excluding individual tests

Tests that cannot be executed for some reason (e.g. which would fail due to null pointer access) can be excluded from the test execution. Such tests are displayed faded and they are automatically skipped when executing all tests of a test object.

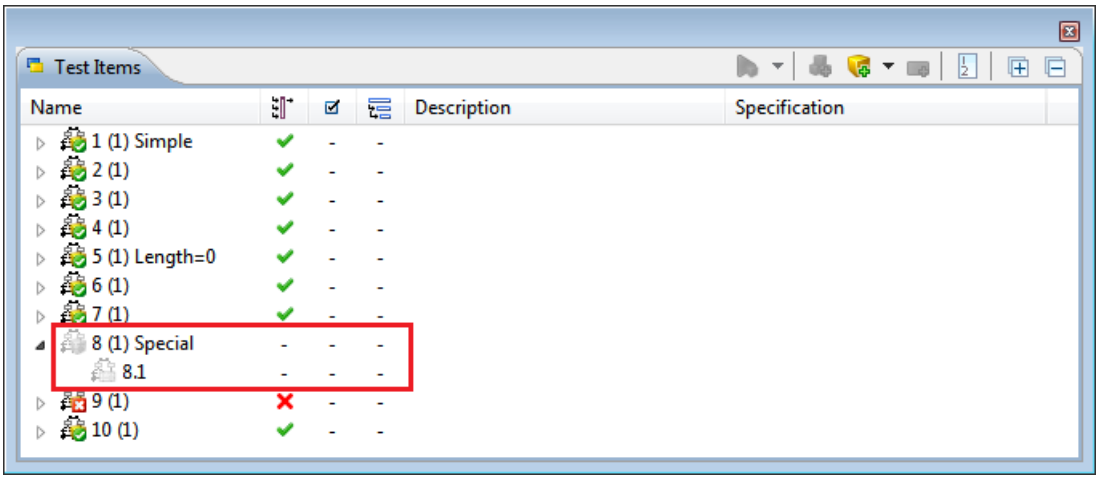


Figure 0.3: Exclude test items from the test

New features in TESSY 4.1

Static analysis and test case quality metrics

In TESSY 4.1 calculation of the McCabe metric (cyclomatic complexity, CC) has been added. (More information in subsection 6.2.2.7 Static code analysis and quality metrics).

The complexity value for each test object will be summarized on module, folder and test collection level using either the sum of all values, the average value or the maximum value.

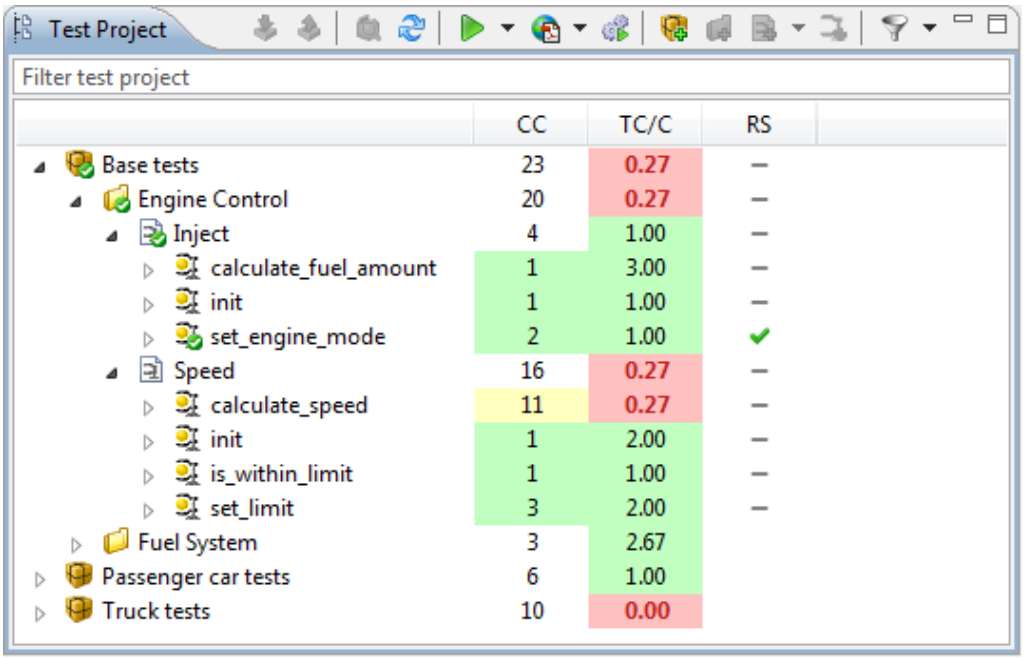


Figure 0.4: Static analysis metrics in the Test Project view

As a derived measure based on the complexity, the TC/C ratio defines the number of test cases necessary to reach 100% branch coverage. A value greater than 1 indicates that at least a minimum number of test cases has been defined.

Another new measure for the quality of test cases is the result significance (RS). It reveals weak test cases: It verifies that each test case has at least some expected results, checks the call trace or uses evaluation macros.

Fault injection

The new feature for automated fault injection on source code level overcomes problems during test implementation caused by programming paradigms that are widely used within safety critical software engineering:

- Defensive programming
- Read-after-write
- Endless loops

(For more information go to chapter 6.14 Fault injection.)

In the past the necessary code coverage of 100% could not be reached with normal test cases so that in practice additional testing code or code replacements using compiler macros were implemented.

Now TESSY provides automated fault injection without affecting the original source code. The fault injection will be placed into the flow graph of the tested function and it will be active only for one or several test cases where it is assigned to. Such fault injection test cases will be marked and documented within the test report.

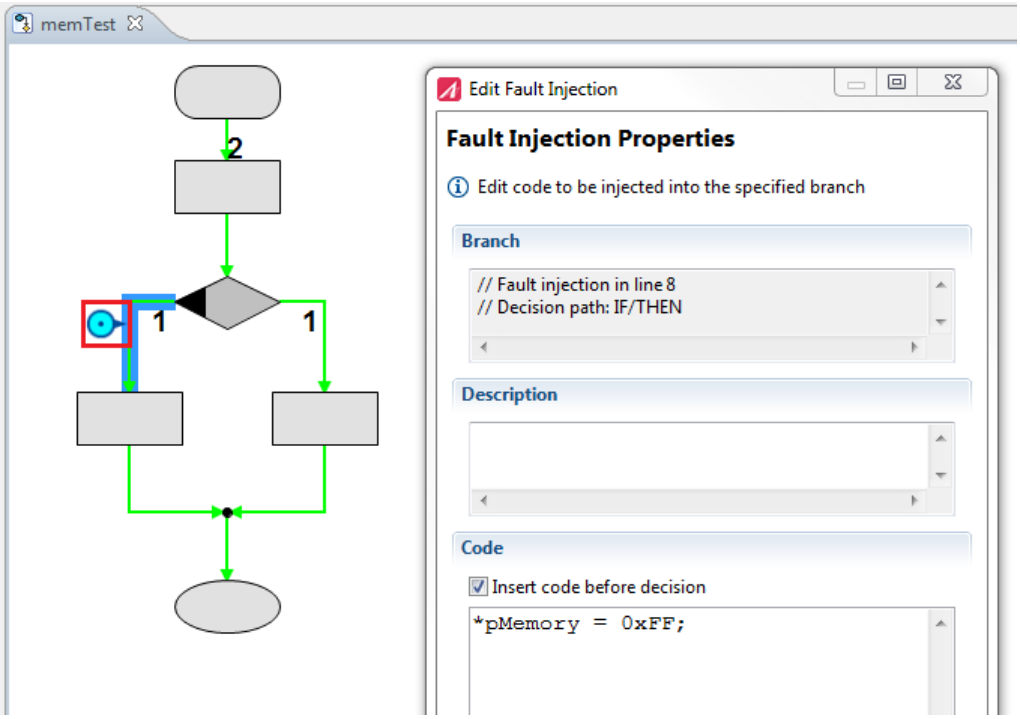


Figure 0.5: Fault injection editing using the Coverage Viewer

Fault injections are created based on unreached branches of the function flow graph. They can be applied without any change to the source code and they will be effective at the desired location even after source code changes when doing regression testing.

Script perspective

The new script perspective allows editing tests within an ASCII editor using a dedicated test scripting language. (More information in chapter 6.10 Script Editor: Textual editing of test cases.)

All test data can be converted from the script format to the TESSY internal format and vice versa. The ASCII format can also be used as complementary backup format besides the hitherto TMB file format.

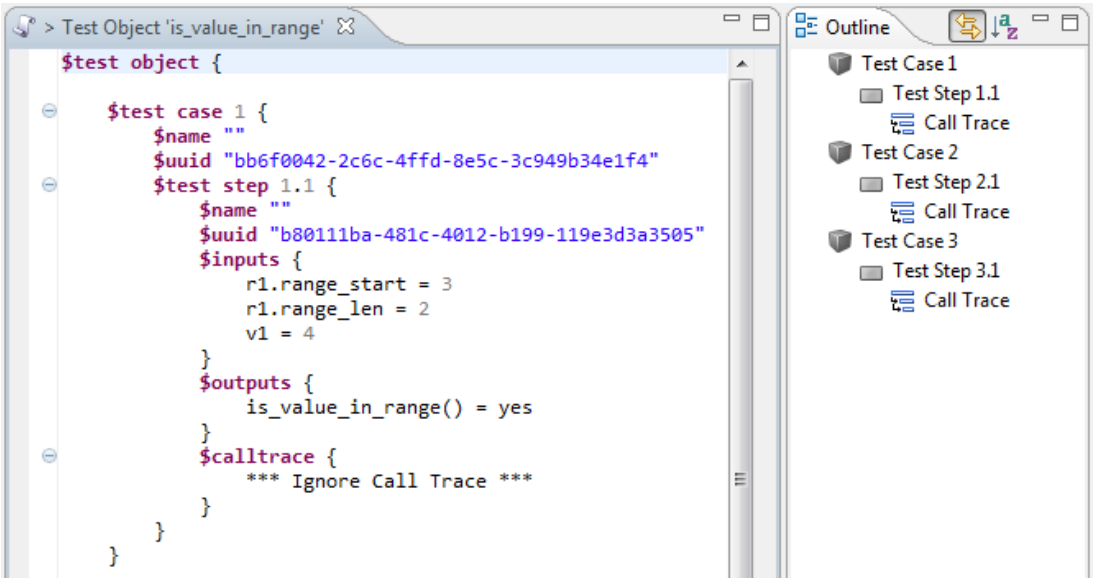


Figure 0.6: Script editor with outline view

The script editor provides syntax highlighting, auto completion, formatting, validation and an outline view.

New Classification Tree editor (CTE)

The new implementation of CTE is a full featured eclipse-based editor integrated into TESSY which enhances the design of test cases and assignment of test data to tree nodes.

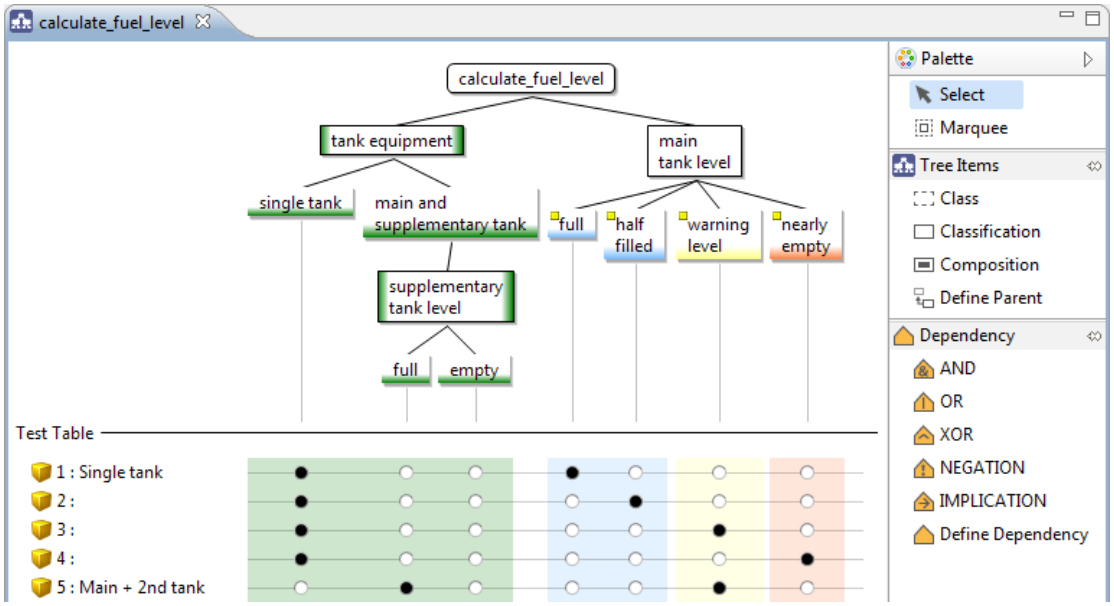


Figure 0.7: The new Classification Tree editor (CTE)

Legacy CTE trees will be updated and converted to the new CTEX file format automatically when they are opened for editing.

(Please go to chapter 6.8 CTE: Designing the test cases for more information about the new CTE. Information about the Classification Tree Method in general can be found in chapter 3.2 The Classification Tree Method (CTM).)

Review of test changes

The new test scripting language provides the means to store the test contents for each test object as ASCII file beside the binary module archive (TMB file). (More information in sections 6.10.4 Script states et seq.)

This allows easy versioning of tests and comparison of arbitrary versions using a version control system. For reviews of tests, changes can easily be found using standard ASCII diff tools.

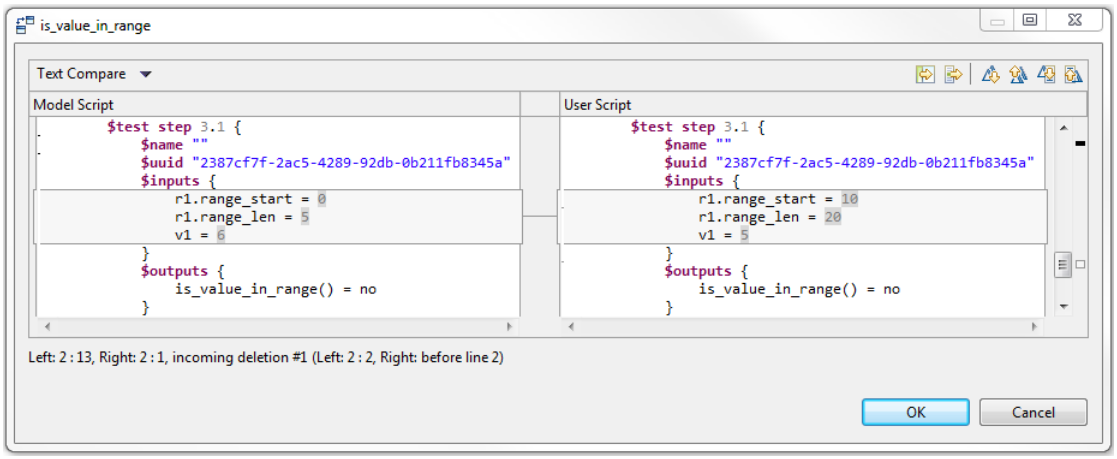


Figure 0.8: Comparison of test versions using a diff tool

Arithmetic expressions as test data

The TDE now allows entering arithmetic expressions as test data values. The resulting value of an expression will be calculated and updated on code changes. The following operators are supported: Addition, subtraction, multiplication, division, shift, binary or/and. The operands can be numbers, defines and enum constants.

Stimulating and measuring hardware signals

In order to enable hardware I/O stimulation and measurement during unit testing, TESSY provides a hardware adapter interface allowing control of external measurement hardware. This hardware device has to implement a configuration interface as well as reading and writing methods for hardware signal data. (More information in chapter 6.6 THAI: TESSY Hardware Adapter Interface.)

During module analysis, TESSY reads the configuration of the hardware device in order to determine the available interface (i.e. the available I/O signals).This list of signals (including passing directions) will appear within the interface of the TESSY module (for each test object). The signals may be edited within the Test Data Editor (TDE) as any other normal test object variable.

New features in TESSY 4.2

CTE extensions

In TESSY 4.2 a generator for classification trees has been added that generates a new tree based on the interface of a test object. A classification will be generated for each input variable with possible values as classes according to the equivalence partitioning method.

More information is provided in subsection 6.8.6.5 Automated tree generation based on function interface.

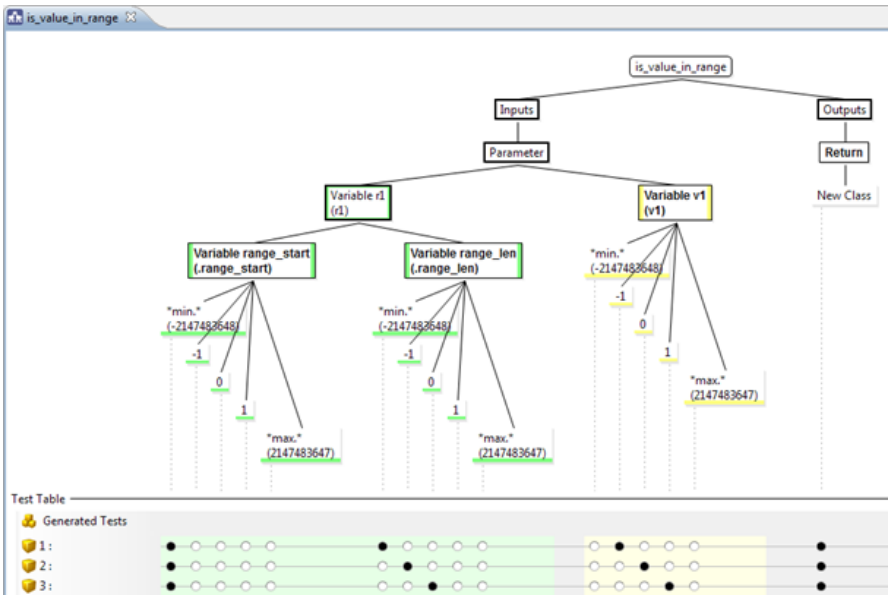


Figure 0.9: Automatically generated classification tree and test cases

The new project-wide data dictionary provides means to assign application domain related names to implementation variables that can be used throughout all CTE test specifications. Data dictionary entries will be synchronized on interface changes and related CTE classifications will be updated automatically.

Component tests for C++ modules

The component test feature has been extended and now also supports the testing of C++ source code. The necessary objects for calling the work tasks and stimulated methods will be created as synthetic variables within the test interface. The respective objects are selected within the properties view of the scenario editor (SCE) perspective.

Call pair coverage

The new call pair coverage measurement (CPC) now supports measuring whether all call locations of functions or methods within the test object have been exercised at least once. This fulfills the requirements of ISO 26262 as an alternate coverage method for integration testing instead of applying the function coverage (FC) method.

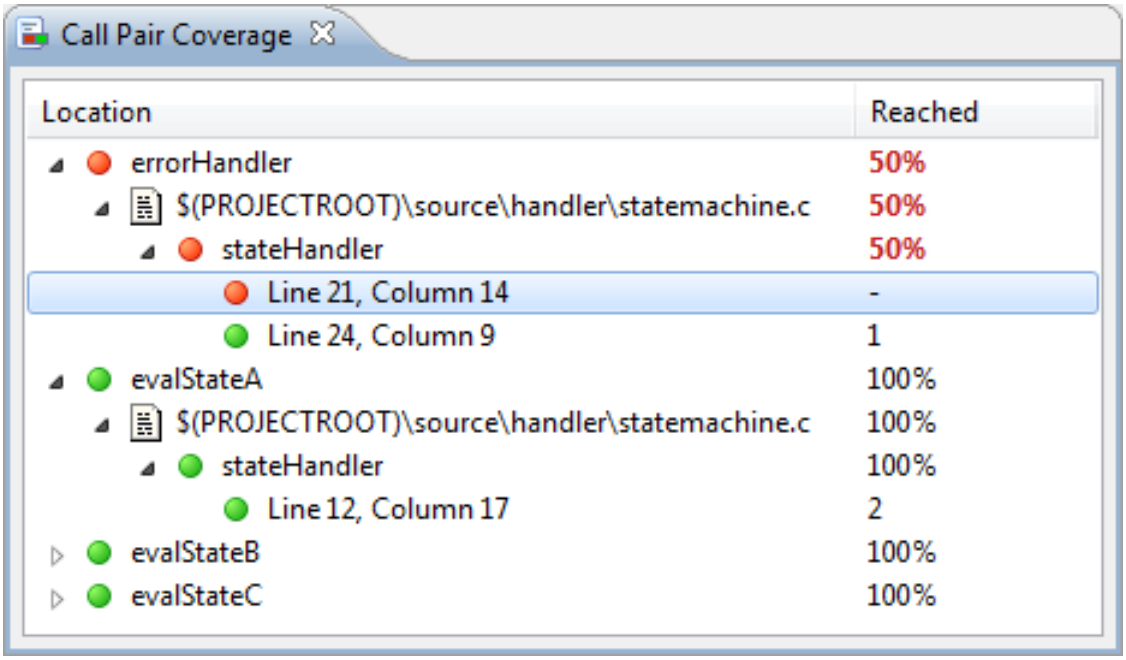


Figure 0.10: Call pair coverage view with coverage results

Testing effort estimation and tracking

The new testing effort estimation is based on a customizable formula that is based on the available metrics provided by TESSY.

For more information go to subsection 6.2.2.8 Testing effort estimation and tracking.

Additional metrics have now been added that can also be used within the time estimation formula:

- Number of statements
- Number of calls
- Maximum depth

When analyzing a module, the estimated time is updated based on the given formula. The actual time can be tracked within the test project view for each test object.

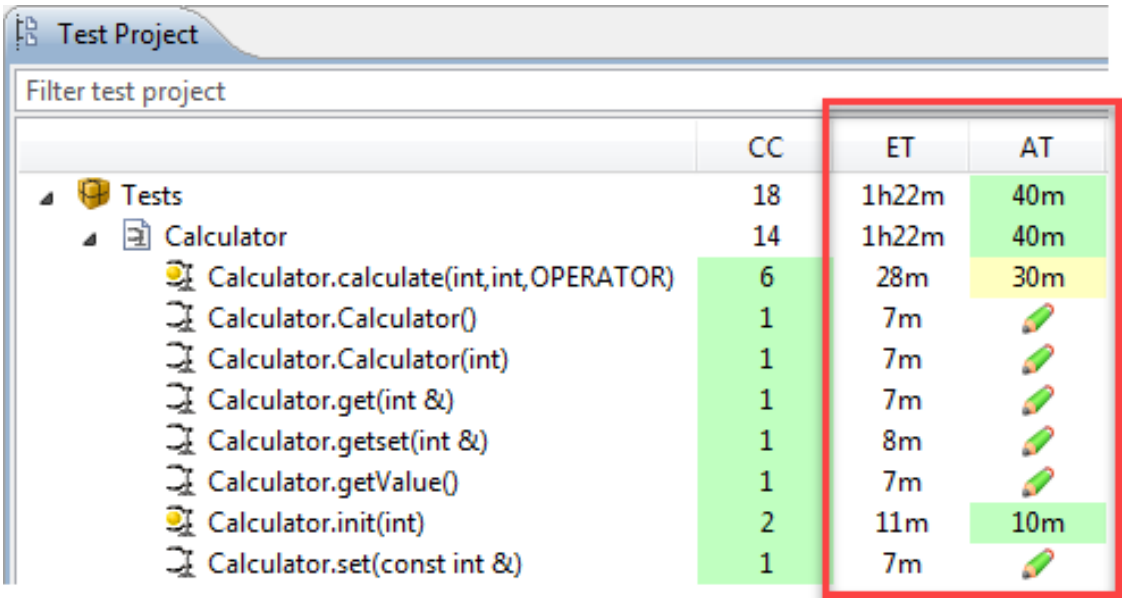


Figure 0.11: Estimated and actual time within the test project view

Warning and error level thresholds as well as the formula itself can be defined within the preferences.

Automated requirement reviews

A new task element provides means to protocol external tests and reviews and link them to requirements. This allows full verification coverage of requirements that are not testable with a normal unit or integration test.

More information can be found in subsection 6.2.2.4 Creating tests and reviews.

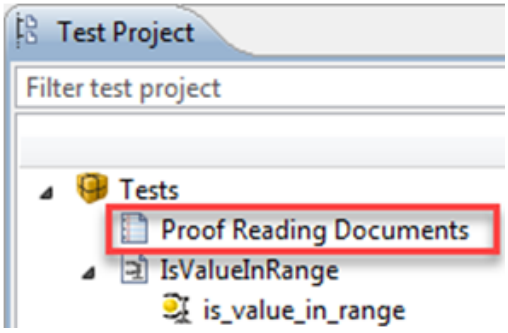


Figure 0.12: A review task within the test project view

Each task has a passed/failed test result and PDF or image files can be attached as a documentation of the review process. The result of a task counts as one test for all linked requirements which provides full coverage also by external test or review.

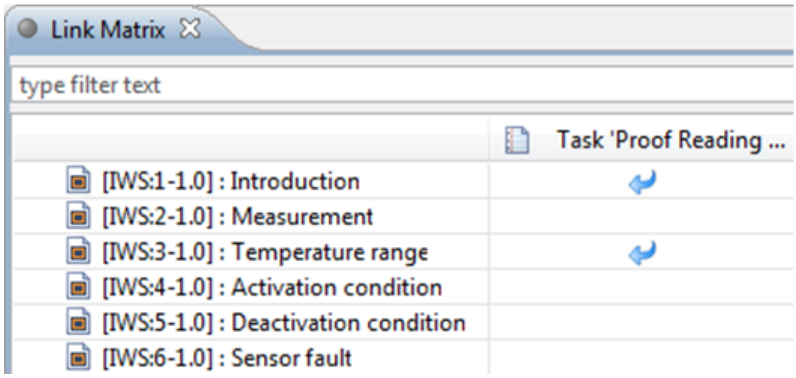


Figure 0.13: Task elements can be linked to requirements using the link matrix

Improved error handling

TESSY 4.2 offers a completely reworked error handling to provide enhanced error messages and logging capabilities for command line execution. A new error dialog shows the full exception chain, the context (e.g. the affected test object) and provides easy access to the error log file.

1 Installation and registration

1.1. Technical requirements **2**

1.2. Setup **2**

1.3. Installation **3**

1.4. Registration **6**

1.4.1. Requesting a license key 6

1.4.2. Registering the license 9

1.4.3. The License Manager (FLM) 15

1.5. Uninstallation **16**

1.6. Using a license without connection to the license server (FLS) **17**

1.6.1. Checking-out the license for use on your local computer 20

1.6.2. Using a license on a computer with no connection to the license server 21

1.1 Technical requirements

- Windows 10 (64bit), Windows 8 (64bit) or Windows 7 (64bit).
- TESSY 4.x can be installed and run in parallel to any previous major TESSY version.
- To be able to open TESSY documentation files and enable the generation of test reports in PDF format you need to install a third party PDF viewer like Adobe Reader 7.0 or higher, Sumatra PDF, Foxit etc.

If you are using Windows 10, please remember to associate pdf files with your third party PDF viewer.

- To run TESSY 4.x you need at least a 1.5 GHz CPU and 4 GB RAM for TESSY.

Important: TESSY 4.1 is 64bit only! Please make sure the computer you want to use with TESSY 4.1 is running on a 64bit version of Windows.
Older TESSY versions can be installed on Windows PCs with 32bit or 64bit.

1.2 Setup

Warning: Deactivate your firewall or virus scanner temporarily while installing TESSY! Otherwise the firewall can cause problems during the installation. Note that some firewalls and anti-virus software can limit the functionality of applications, including TESSY and might need to be modified. For further help ask the producer of the firewall or anti-virus software.

Important: You need local administrator privileges to install TESSY!

TESSY allows you to have multiple TESSY installations with different versions on the same computer. You do not have to uninstall older versions.

1.3 Installation

To install TESSY 4.x on your computer,

→ download TESSY from the ”Downloads“ section of the RAZORCAT webpage (http://www.razorcat.com).

→ After the download is complete click on the setup.exe.
The InstallAware Wizard will start. This will take a few moments.

→ When the InstallAware Wizard is ready, click “Next”.

Download TESSY

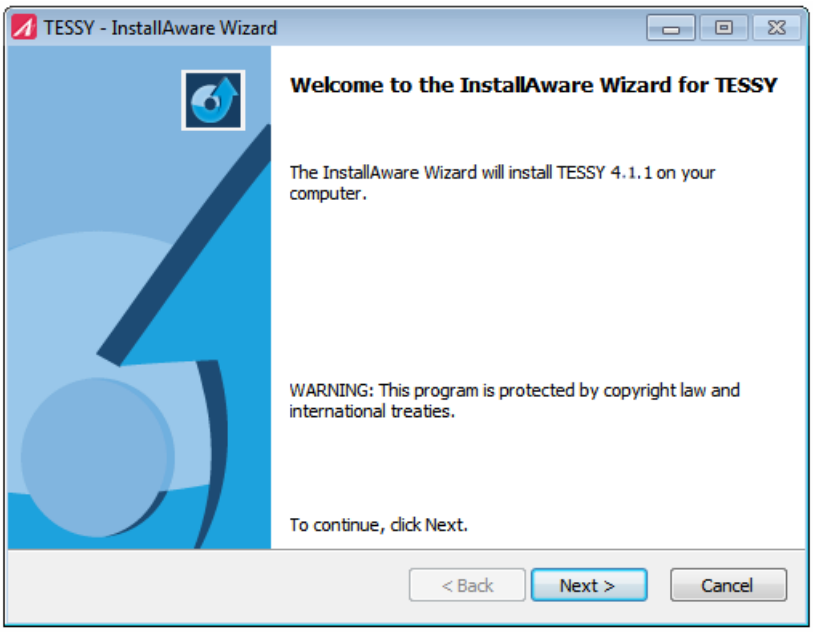


Figure 1.1: InstallAware Wizard

→ Read the license agreement carefully. Check the box to accept and click “Next”.

→ Now select the setup type: “Complete” (default) is recommended. click “Next”.

→ Select the destination folder (default “`C:\Program Files\Razorcat`”) and click “Next”.

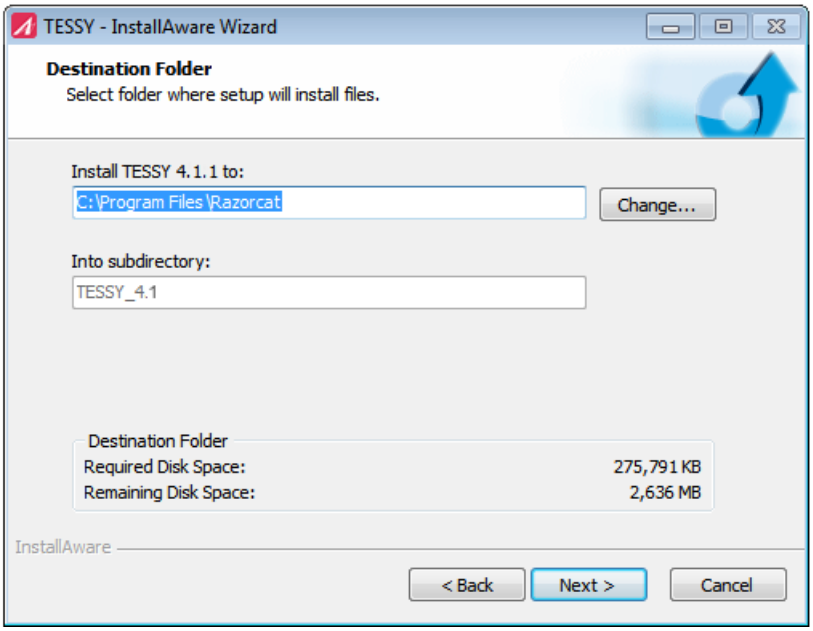


Figure 1.2: Destination Folder

→ Select the TESSY testarea folder (“Folder for temporary files:”; default “`C:\tessy`”) click “Next”.

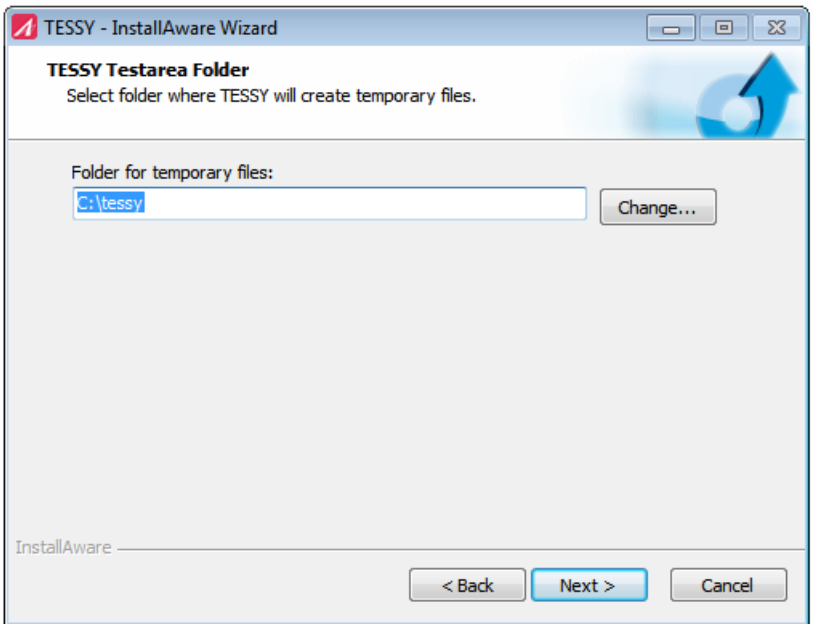


Figure 1.3: TESSY Testarea Folder

→ Select the program folder (default “TESSY 4.x”) and decide, whether the installation on the system is for “all users” or for a certain user. Click “Next”.

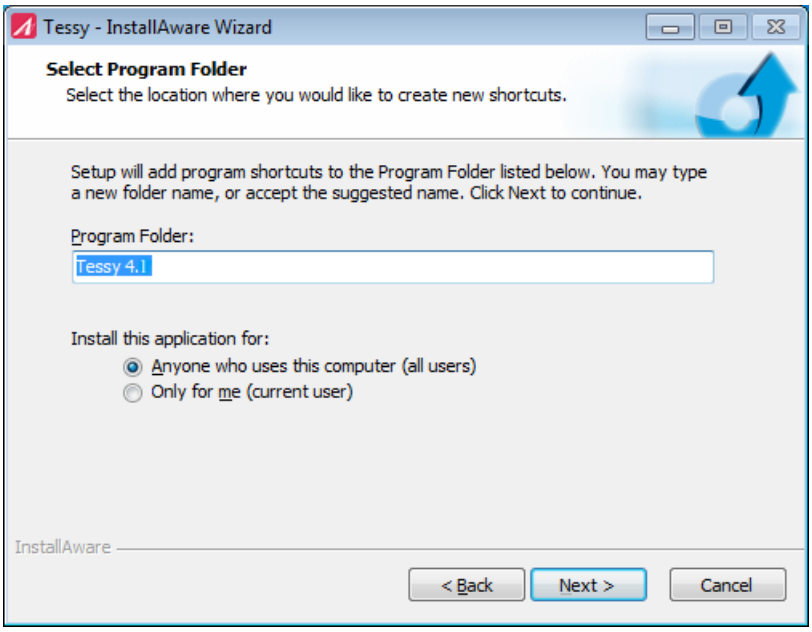


Figure 1.4: Program folder and user selection

→ Start the installation by clicking “Next”. The installation will take a few moments.

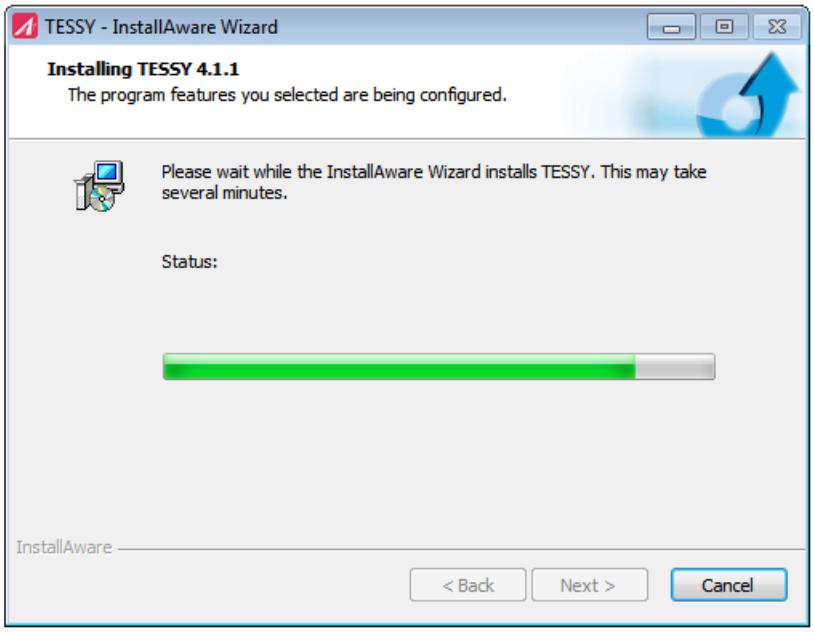


Figure 1.5: Installing process of TESSY

→ When the Install Aware Wizard is completed, click “Finish”.

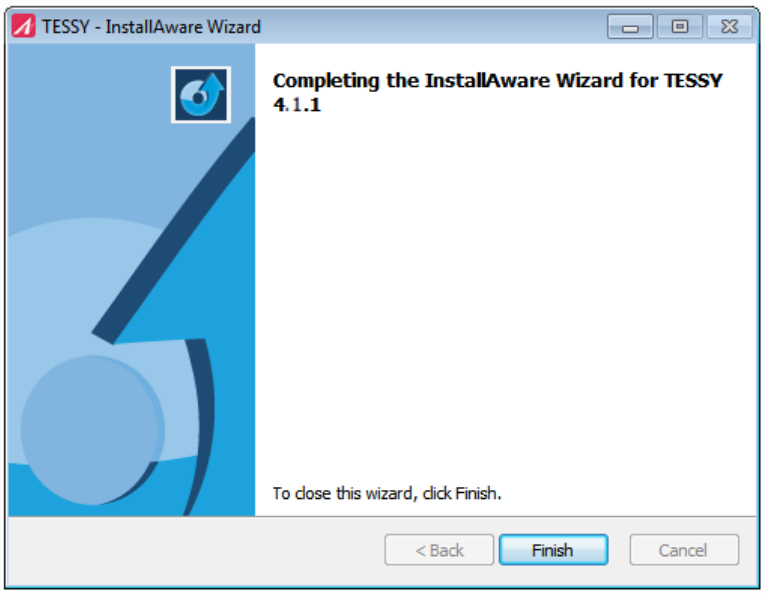


Figure 1.6: Installation is completed

Now you have completed the installation of TESSY.

1.4 Registration

1.4.1 Requesting a license key

→ Start TESSY by clicking “Start” in Windows > “TESSY 4.x” > “TESSY 4.x” (see fig. 1.7).

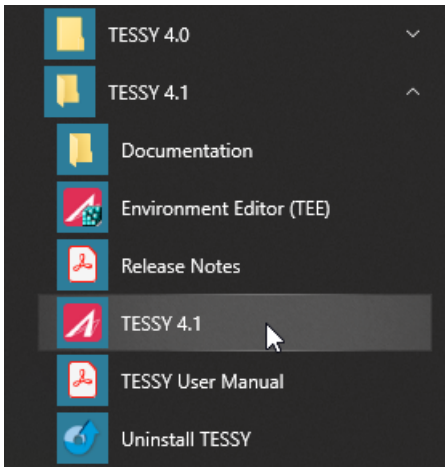


Figure 1.7: Starting TESSY 4.x

If a valid license is found, TESSY will start. If there is no valid license, the License Manager will start with a License Key Request popup window (see figure 1.9).

Important: To request a TESSY license you need access to the internet!

To start the request manually,

→ start the License Manager similarly to starting TESSY by clicking "Start" in Windows > "Razorcat Floating License Server 8.x" > "Floating License Manager".

→ Click on "License" > "Request" (see figure 1.8).

License key request

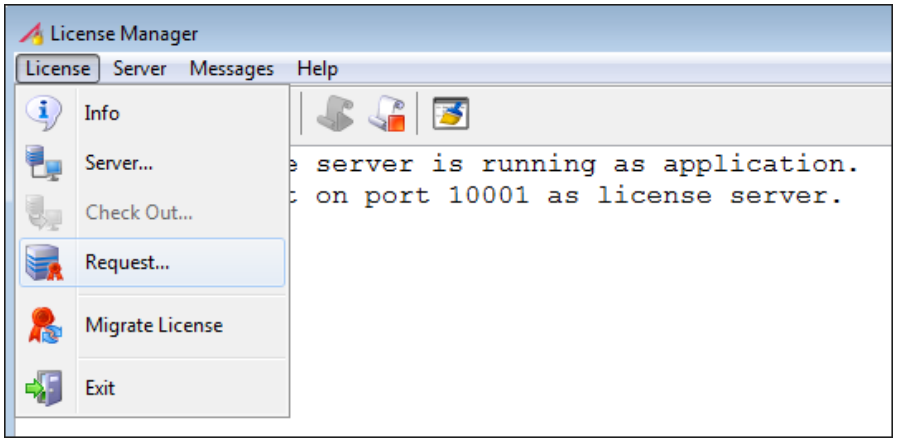


Figure 1.8: Starting the key request

The License Key Request popup window will appear (see figure 1.9).

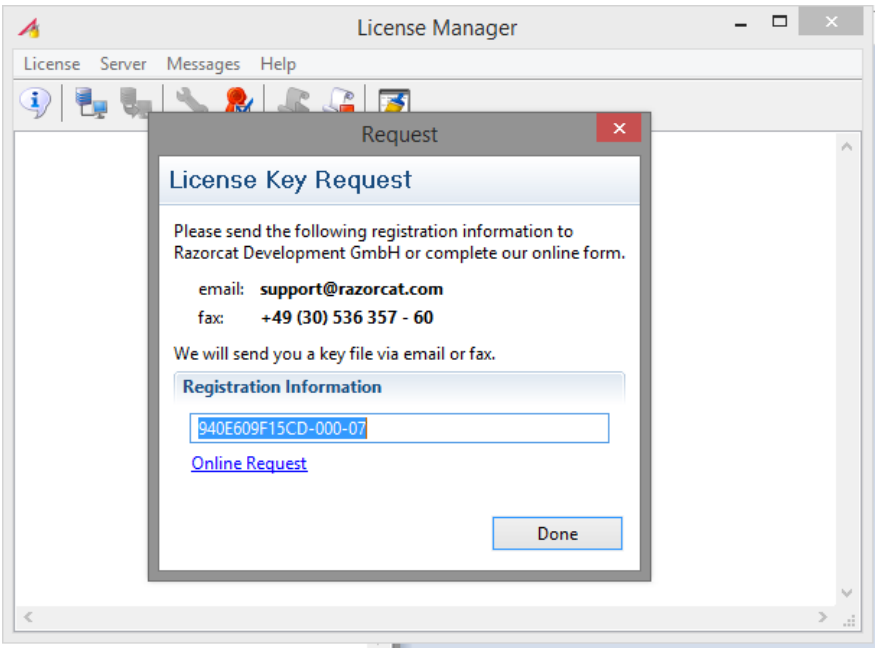


Figure 1.9: License key request popup window

→ Click on “Online Request” and fill out the form for a license key request.

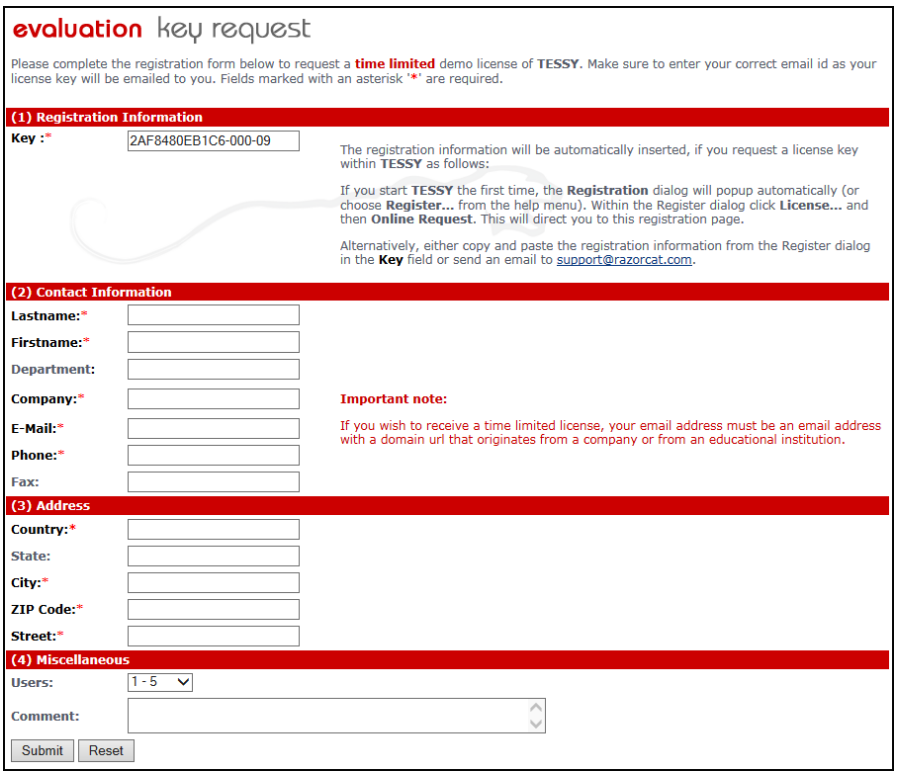


Figure 1.10: Form for the license key request

You will get a license key via e-mail within a license key file. The license key file is a plain text file with the ending .txt.

Important: The license key file is not generated and send out automatically, therefore it can take up to a workday until you receive it!

1.4.2 Registering the license

TESSY offers two types of licenses:

- Node-locked license
- Floating license

The node-locked license is a single user license issued for a given host ID. It's not possible to share the license with other users within your network. A node-locked license operates only on the particular computer for which the license is issued.

The floating license is a server license issued for a given host ID for a dedicated server within your network. It's possible to share the license with other users within your network.

The Floating License Server (FLS) is running on a central network server and manages the licenses that are in use. Thus TESSY can be used on any computer within the network. The number of users who can use the software simultaneously is determined by how many licenses you have purchased.

The License Manager (FLM) is started locally on a computer and displays the state of the FLS (e.g. how many licenses are currently checked out).

If you want to run TESSY with a time limited demo license, just follow the instructions in Registering a node-locked license on one computer. The License Manager (see figure 1.14) will keep you informed about the validity of your license key.

Warning: The registry entries for TESSY's Floating License Server (FLS) are generated during installation and must not be altered manually! Otherwise your host ID and therefore the license key might become invalid.

During the installation the FLS needs an operational Ethernet network interface. The FLS may not be used as a floating license server but as a node-locked license server only if a proper Ethernet network interface is missing.

The tool “hostid.exe”, which can be found in the FLS installation folder, outputs the current status if called with option “-f”.

If the host ID was destroyed for some reason, you can reinstall the license server. The installer will repair the corresponding registry entries and generate a new host ID.

Please keep in mind, that the last operational host ID is saved in the registry as well. Therefore you should not delete the registry manually before reinstalling the license server.

After startup the license server will generate a special key which you can sent to repair@razorcat.com in order to receive a new license file.

1.4.2.1 Registering a node-locked license on one computer

Node-locked license

When you have received the license key file (*.txt file),

→ open the License Manager by clicking “Start” in Windows > “Razorcat Floating License Server 8.x” > “Floating License Manager”.
→ In the opening popup window click on “Done”. (see figure 1.9)
The Configure window for the License Server will open (see figure 1.11).

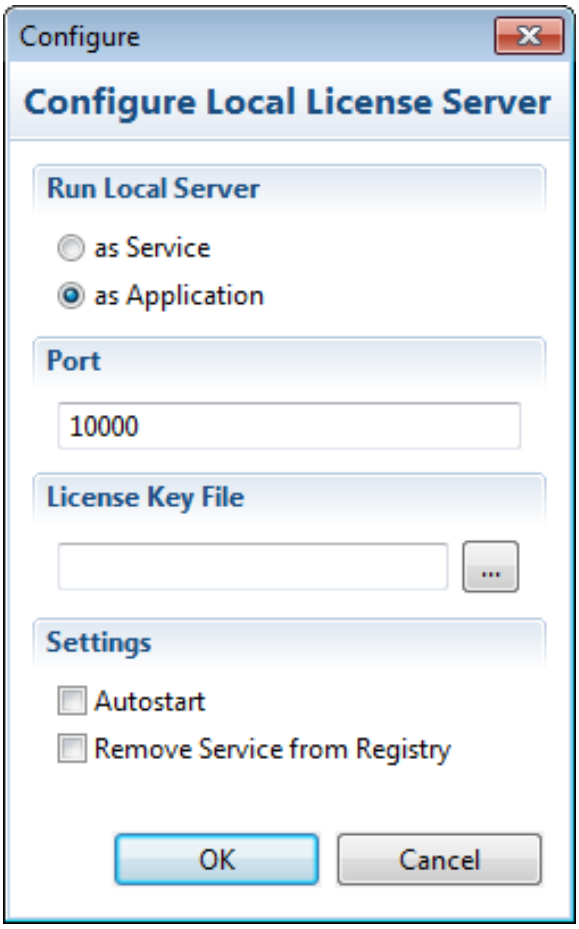


Figure 1.11: Configure menu of the license server

→ Under “Run Local Server” click “as Application” (default).

→ Under “License Key File” choose the license key file (*.txt) you have received.

→ Tick the box “Autostart” to automatically start the license server in the background whenever the computer starts. The system will try to set the autostart for the actual Windows user.

→ Click “OK”.
The license server will start automatically and the License Manager (see figure 1.14) will display information about your license, server and registration.
If the license server does not start, click on “Start Local Server” in the License Manager.

→ Close the License Manager and start TESSY by clicking “Start” in Windows > “TESSY 4.x” > “TESSY 4.x” (see fig. 1.7).

Optionally you can run the license server as service. In this case the autostart will be set for the computer and all its users.

Important: Please note: You need administrator privileges to run the license server as service.

→ Select “Service” under “Run Local Server”.

→ Then follow the further instructions as described above.

1.4.2.2 Registering a floating license for network usage

The floating license is a server license issued for a given host ID for a dedicated server within your network. It’s possible to share this kind of license with other users within the respective network.

Important: Make sure that the license you want to use for the following process really is a floating license not a node-locked license.

Floating license

To run a central license server within your network, please take the following steps:

→ Login as administrator.

→ Install the Floating License Server (FLS) on your network server.
You can download the latest standalone version of the Razorcat FLS from https://www.razorcat.com/de/downloads-tessy.html.

Important: This download is a standalone setup for the FLS and it is used for server installations without using a TESSY setup.

→ Start the License Manager by clicking “Start” in Windows > “Razorcat Floating License Server 8.x” > “Floating License Manager”.

→ Click on “Configure” in the toolbar.
The following dialog window will pop up. Please choose following options within the dialog to run the license server as service (see fig. 1.12).

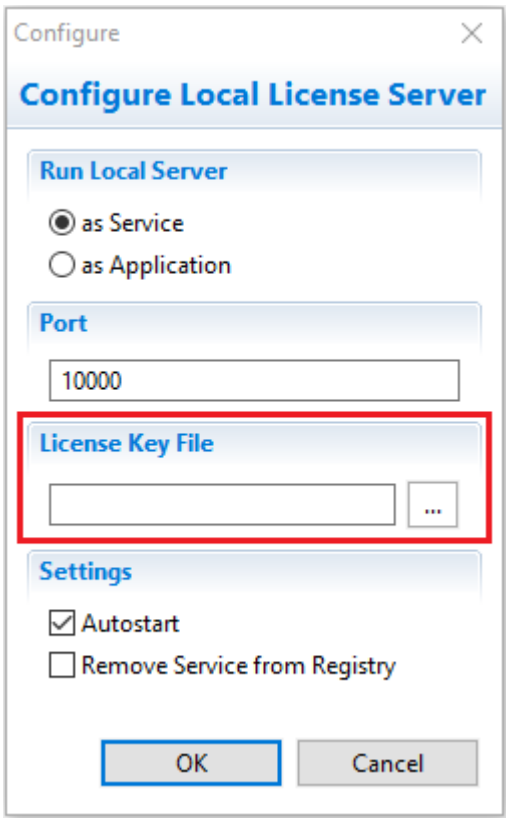


Figure 1.12: Settings for a floating license in the configure menu

→ Under “License Key File” choose the license key file (*.txt) you have received. The license key file will be copied into the bin directory of the license server installation. Please check if the license file has been copied after completing these steps.

→ Click on “OK”.
The license server will start automatically and the License Manager (see figure 1.14) will inform you about the configuration changes you have just made.

Important: Make sure that the license key file can be found in the bin directory of the license server. If not, it needs to be copied to this place.

1.4.2.3 Selecting a license server for a network computer

To be able to work with TESSY on various computers within one network you have to select the floating license server on every single computer you want to use TESSY.

Selecting a floating license on one network computer:

→ Start the License Manager by clicking “Start” in Windows > “Razorcat Floating License Server 8.x” > “Floating License Manager”.
→ Then click “License” > “Server”.
The following dialog window will pop up.

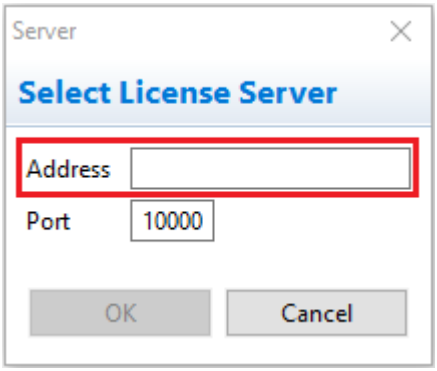


Figure 1.13: Dialog window to select license server in a network

→ Under “Address” insert the license server name or IP address within your network.
→ Click “OK”.

1.4.2.4 Updating a Floating License Server (FLS)

During the process of updating TESSY it is possible that TESSY also requires an updating of the Razorcat Floating License Server (FLS) on your network server and an updated license key file.

Download the latest FLS

You can download the latest standalone version of the Razorcat FLS (for server installations) from https://www.razorcat.com/de/downloads-tessy.html.

A newer version of the license server can be installed in parallel to any existing FLS installation. To start the updated version of the license manager the previously running license server needs to be stopped and deactivated.

Detailed descriptions about the updating procedure can be found on https://www.razorcat.com/de/tessy-faq.html.

1.4.3 The License Manager (FLM)

The Floating License Manager (FLM) is used to control and to configure the Floating License Server. In general there is no need to use the FLM, because the configuration of a local license server automatically takes place during the installation of the license file. In some cases it is helpful to use the FLM to change the default settings to suit your needs.

The FLM is in any case necessary to configure a central license server (see Registering a floating license for network usage) as well as in case of problems.

→ Start the License Manager by clicking "Start" in Windows > "Razorcat Floating License Server 8.x" > "Floating License Manager"

Functions and icons of the FLM:

Icon	Name	Function
	Info	Show information about the license.
	Server	Select license server.
	Check Out	Check out licenses from the server for local usage.
	Configure	Configure local license server.
	Start Local Server	Start the server manually.
	Stop Local Server	Stop the server manually.
	Configure	Configure the server.
	Check	Check your license key file.
	Clear Window	Clear FLM window.

Table 1.1: Functions of the FLM

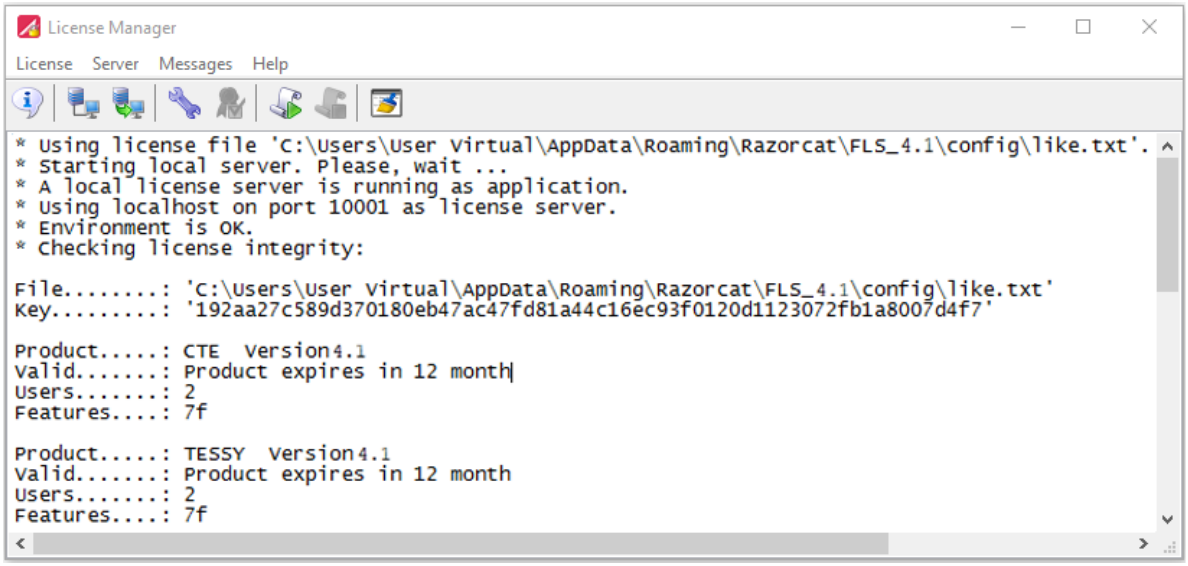


Figure 1.14: License key check successful: this license key is correct

For more information about the handling and status of licenses please read the following sections: 1.6 Using a license without connection to the license server (FLS), 1.6.1 Checking-out the license for use on your local computer, and 1.6.2 Using a license on a computer with no connection to the license server.

1.5 Uninstallation

Important: By uninstalling TESSY the project root will not be deleted, neither your project data or your configuration file will be deleted. Nevertheless: Please make sure that your data is saved.

To uninstall and remove all components of TESSY,

→ click "Start" in Windows > "TESSY 4.x" > "Uninstall TESSY".
All components of TESSY will be removed. This will take a few seconds.

The "Razorcat Floating License Server" (FLS) and the "Razorcat Shared" installation files will remain on the computer. If you want to delete those as well, you can use the windows "Apps & features" function (see figure 1.15).

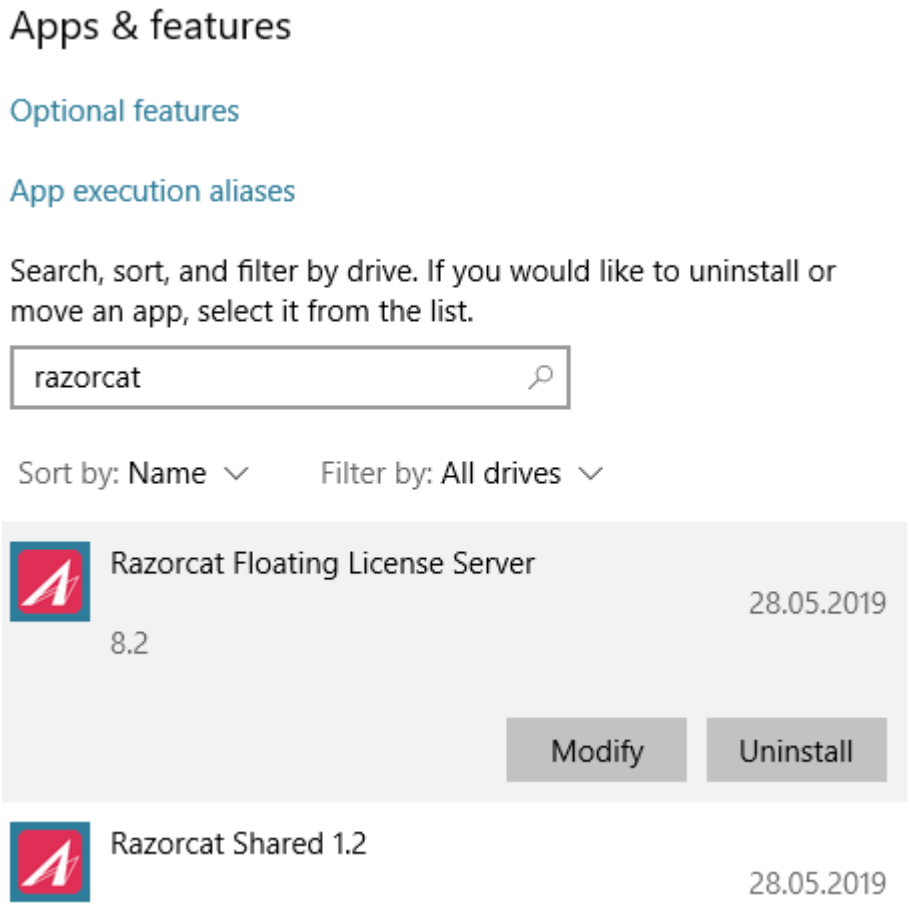


Figure 1.15: Uninstalling FLS and Shared installation files

1.6 Using a license without connection to the license server (FLS)

Important: For using your TESSY license temporarily without connection to the license server, you need a “Floating License”.

You can temporarily check-out your TESSY floating license on your computer and work independently of the connection to the Floating License Server (FLS).

This is useful when there is temporarily no connection to the license server available (e.g. when traveling, see subsection 1.6.1 Checking-out the license for use on your local computer).

It is also possible to check-out a license for a computer that will never be able to reach the license server directly (e.g. a stand-alone computer or a computer within an isolated network, see subsection 1.6.2 Using a license on a computer with no connection to the license server).

Checking-out a License

This check-out option

- is only possible with a TESSY floating license.
- is effective for 30 days altogether: You can check-out the license for one, for two or up to 30 days each time, but overall for 30 days at the most.
- means, that you can use TESSY **either** on computer 1 **or** on computer 2. When you check-out the license on computer 1 for three days, you can **not** use TESSY on computer 2 for these three days.

Important: It is **not** possible to return a checked-out license prematurely to the computer from which the licenses were checked-out!

To see how many days you may check-out a license

→ click “Start” in Windows > “Razorcat Floating License Server 8.x” > “Floating License Manager”.
→ In the License Manager click on “License” > “Info”.
→ Next to “State” the amount of days for possible check-outs will be displayed (see figure 1.16).

Important: If the state says “disabled” contact your administrator!

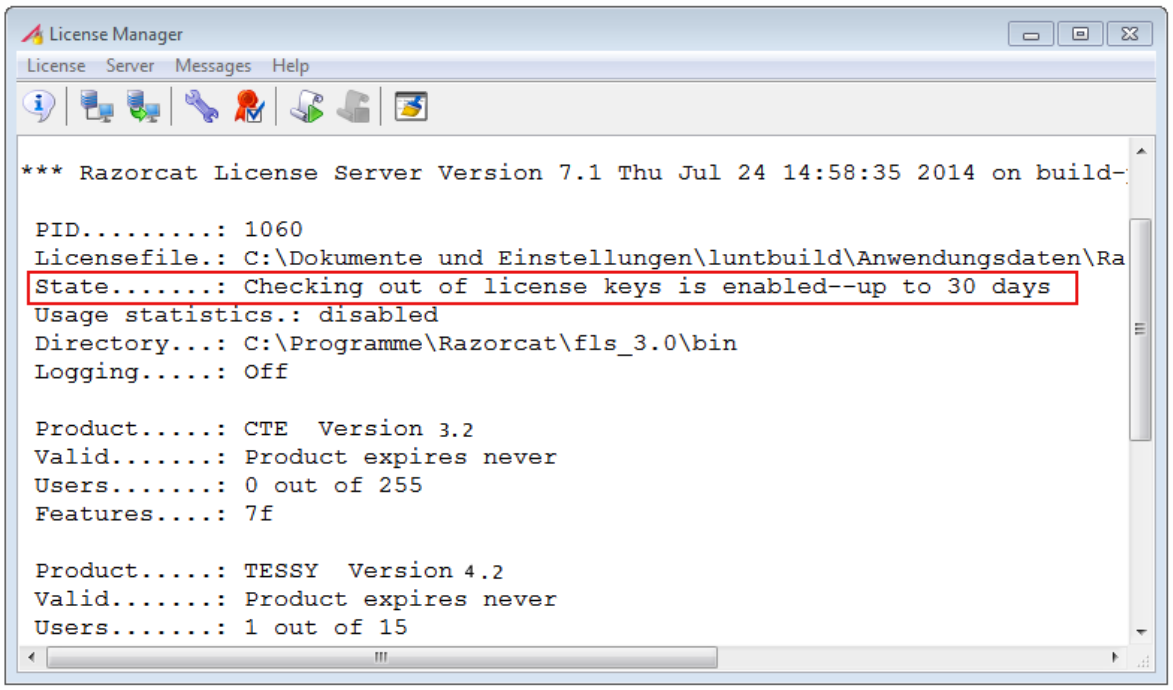


Figure 1.16: The license info shows the possible number of days for checking out the license

Licenses in use

The License Manager also provides information about the licenses that are in use on your computer or were transferred part-time to another computer (click on > “Info”).

Info text example	Meaning

`joe@company.com:0`
`checked out for 2 day(s) since`
`Wed Jun 06 17:21:07 2018`
	License checked out statically
	(for a selected number of days)

`joe@company.com:50874`
`currently checked out since`
`Wed Jun 06 17:21:39 2018`
	License checked out dynamically
	(as long as TESSY is in use)

Table 1.2: Information about licenses in use in the License Manager

1.6.1 Checking-out the license for use on your local computer

Use this license check-out option if you want to make use of the license independently of the license server connection (e.g. when traveling).

→ In the menu bar click on “License” > “Check Out…” (see figure 1.17).

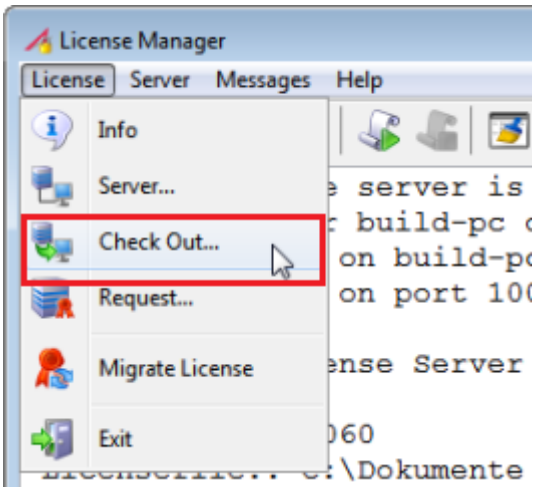


Figure 1.17: Checking out the TESSY license

→ Choose the amount of days. The Registration Information will be filled out automatically (see figure 1.18).

→ Click “ok” and save the file.
You can now use this license file on your local computer. To register the license refer to section 1.4.2 Registering the license.

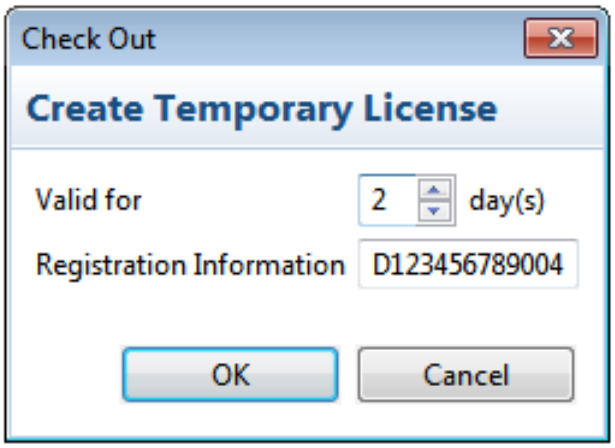


Figure 1.18: Determine the amount of days for the check-out

1.6.2 Using a license on a computer with no connection to the license server

To be able to use the TESSY floating license on another computer with no license server connection (e.g. a stand-alone computer or a computer within an isolated network) the procedure described in section 1.6 Using a license without connection to the license server (FLS) needs to be modified.

Transmitting the license file

- → Open the license manager on your computer with NO license server connection (see 1.4.3 The License Manager (FLM)).
- → In the menu bar click “License” > “Request…” .
- → Copy the first 12 characters in “Registration Information” (see figure 1.9).
- → Transmit these 12 characters to your local computer WITH license server connection.
- → Insert the transmitted 12 characters into “Registration Information” (see figure 1.18) when checking-out on your local computer WITH license server connection (see section 1.6.1 Checking-out the license for use on your local computer).
- → Choose the amount of days, click “ok”, and save the license file.
- → Make this file available on the other computer with NO license server connection.
- → You can now register this license file on the other computer with NO license server connection (see section 1.4.2 Registering the license).

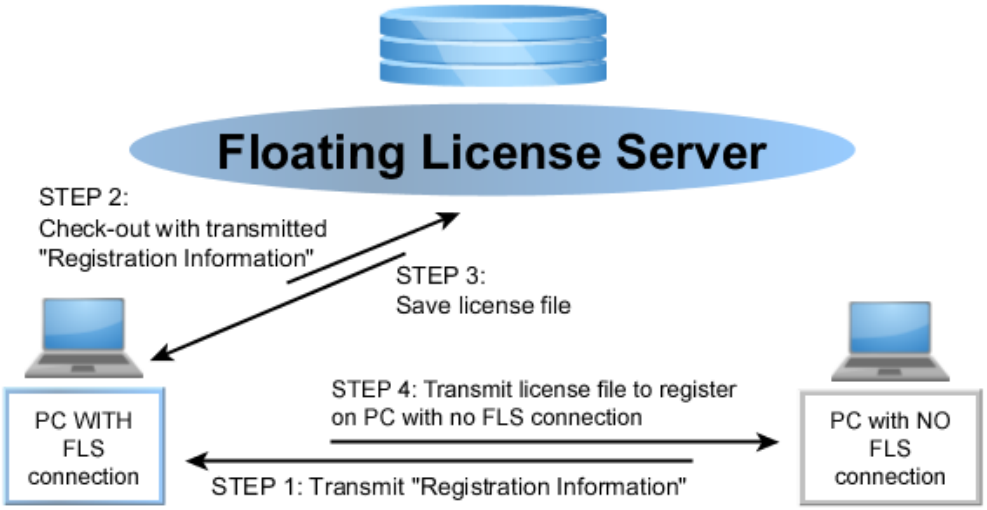


Figure 1.19: Transmitting the license file to a computer with no FLS connection

2 Migrating from TESSY 3.x to 4.x

In the new version of TESSY you will find some new functions as mentioned in section New features in TESSY 4.0 and New features in TESSY 4.1.

2.1 Importing previous projects

You will have to convert your projects to use them with the new TESSY 4.x version. When you open a project, TESSY will ask you if you want to convert your project. By clicking “Yes” TESSY will convert the project automatically.

Warning: Once you have converted your project it cannot used by TESSY 3.x anymore! If you want to use your project with TESSY 3.x, make a copy of the project.

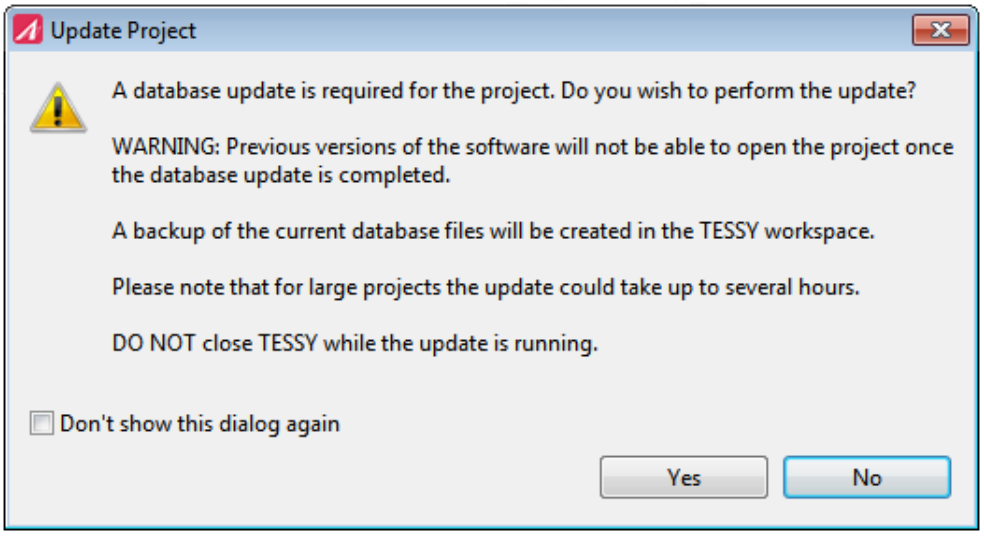


Figure 2.1: Updating a project

3 Theory: Basic knowledge

This chapter offers a brief introduction about unit testing with TESSY and the classification tree method. It provides basic knowledge for organizing and executing a unit test in general and in particular with TESSY. The chapter about the classification tree method helps you to understand the logical system and to use the CTE.

3.1. Unit testing of embedded software **25**

3.1.1. Standards that require testing 25

3.1.2. About unit testing 25

3.1.3. Considerations for unit testing 27

3.1.4. Methods for unit testing 29

3.1.5. Conclusion 31

3.2. The Classification Tree Method (CTM) **32**

3.2.1. General 32

3.2.2. Steps to take 33

3.2.3. Example is_value_in_range 39

3.1 Unit testing of embedded software

3.1.1 Standards that require testing

International standards like IEC 61508 require module tests. According to part 3 of IEC 61508, the module test shall show that the module under test performs its intended function, and does not perform unintended functions. The results of the module testing shall be documented.

IEC 61508 classifies systems according to their safety criticality. There are four safety integrity levels (SIL), where 1 is the lowest level and 4 the highest, i.e. systems at level 4 are considered to be the most critical to safety. Even for applications of low criticality (i.e. at SIL 1), a module test is already "highly recommended". The tables contained in the annexes of IEC 61508, Part 3 specify the techniques that should be used, e.g. for module testing the technique "functional and black box testing" is highly recommended at SIL 1 already. Other techniques, such as dynamic analysis and testing are recommended at SIL 1 and highly recommended at SIL 2 and higher.

Part 4 of IEC 61508 defines a (software) module as a construction that consists of procedures and/or data declarations, and that can interact with other such modules. If we consider embedded software which is written in the C programming language, we can take a C-level function as a module. To prevent a mix-up between C-level functions and C source modules, we will refer to the C-level functions as units from now on.

Also other standards like the British Def Stan 00-55, ISO 15504 or DO-178B require module testing (where the nomenclature ranges from "module" to "unit" to "component"). However, all standards have more or less the same requirements for that kind of test: the tests have to be planned in advance, test data has to be specified, the tests have to be conducted, and the results have to be evaluated and documented.

3.1.2 About unit testing

3.1.2.1 What is unit testing?

During unit testing of C programs, a single C-level function is tested rigorously, and is tested in isolation from the rest of the application. Rigorous means that the test cases are specially made for the unit in question, and they also comprise of input data that may be unexpected by the unit under test. Isolated means that the test result does not depend on the behavior

of the other units in the application. Isolation from the rest of the application can be achieved by directly calling the unit under test and replacing the calls to other unit by stub functions.

Unit testing tests at the interface of the unit, and unit testing does not consider the internal structure of the unit, and therefore unit testing is considered as black-box testing.

The interface of the unit consists of the input variables to the unit (i.e. variables read by the unit) together with the output variables (i.e. variables written by the unit). A variable can both be an input and an output (e.g. a variable that is incremented by the unit), and the return value of the unit - if present - is always an output. The structure of a test case follows from the structure of the interface.

Unit testing is conducted by executing the unit under test with certain data for the input variables. The actual results are compared to those predicted, which determines if a test case has passed or failed.

Unit testing (of C-level functions, as described) is well suited to fulfill the requirements of module testing for IEC 61508, because unit testing is

- functional, because the functionality of the unit is tested, and
- a black-box, because the internals of the unit are not taken into account, and
- dynamic, because the test object is executed during the test.

3.1.2.2 What are the benefits?

- **Finding errors early:** Unit testing can be conducted as soon as the unit to be tested compiles. Therefore, errors inside the unit can be detected very early.
- **Saving money:** It is general knowledge that errors which are detected late are more expensive to correct than errors that are detected early. Hence, unit testing can save money.
- **Reducing complexity:** Instead of trying to create test cases that test the whole set of interacting units, the test cases are specific to the unit under test. Test cases can easily comprise of input data that is unexpected by the unit under test or by even random input test data, which is rather hard to achieve if the unit under test is called by a fully-functioning unit of the application. If a test fails, the cause of the failure can be easily identified, because it must stem from the unit under test, and not from a unit further down the calling hierarchy.

- **Giving confidence:** After the unit tests, the application is made up of single, fully tested units. A test for the whole application will be more likely to pass, and if some tests fail, the reason will have probably stemmed from the interaction of the units (and not from an error inside a unit). The search for the failure can concentrate on that, and must not doubt the internals of the units.

3.1.3 Considerations for unit testing

3.1.3.1 Which units are good test candidates?

Unit testing verifies that certain input data generates the expected output data. Therefore, units that do data processing in its widest sense, e.g. generation of data, analysis of data, sorting, making complex decisions, difficult calculations are best suited for unit testing. To find such units, the application of metrics (e.g. the cyclomatic complexity according to McCabe) may be appropriate.

Other criteria for selecting units to test may be how critical the functionality is to the unit's operation, or how often a unit is used in the application.

3.1.3.2 What is not in the scope of unit testing?

The interaction of the units is not tested during the unit test. This includes the semantic of the parameters passed between units (e.g. the physical unit of the values), and the timely relationships between units (e.g. does a unit fulfill its task fast enough to let a calling unit fulfill their tasks also at the required speed?) In addition, the interrupt behavior of the application is not in the scope of unit testing. Questions like "Does my interrupt really occur every 10 ms?" or "Which interrupt prolonged my unit unacceptably?" are not addressed by unit testing, because unit testing explicitly aims at testing the functional behavior of the unit isolated from environmental effects such as interrupts.

3.1.3.3 Why is regression testing necessary?

Regression testing is the repetition of tests that have already passed after the implementation of bug fixes or improvements in the software. Regression testing proves that a change in the software did not result in any unexpected behavior. Regression testing is a key to software quality. Obviously, the practice of regression testing requires the automation of the tests, because the effort to repeat the tests manually is too high. Even for non-repetitive unit tests,

the proper tool support will save you lots of time, but tool support is indispensable for the repetition of the unit tests.

3.1.3.4 Who should conduct the tests?

The dilemma: It is commonly accepted that a software developer is badly suited to test his own software, especially if the complete implementation, or the compliance of the implementation with the specification is an issue (blindness against own faults). If the developer has forgotten to implement a certain functionality, it is likely he will also forget a test that will reveal the missing functionality. If the developer has misinterpreted the specification, it is likely that his tests will pass in spite of the wrong functionality.

On the other hand, experience has shown that a tester, who should test a code not written by him must put a lot of effort into understanding the function´s interface. The tester must find out the meaning of the variables, and which values to use to conduct certain tests. E.g., if the test specification requires the test of something "green", which variable (or variables) represents the color, and which value of the variable represents green? The prediction of the expected results poses similar problems.

If the developer does not do tests, this gives rise to additional efforts, because the failed test has to be passed to the developer, he has to reproduce the failure, correct the problem, and then normally a concluding external regression test has to take place. Furthermore, additional effort rises due to the fact that the developer will not hand out his software to the QA department without having done at least some tests. This duplicated test effort could be saved if the developer immediately starts testing by using the externally predefined test cases.

The way out: A way out of that dilemma could be that a tester, who has not written the code, specifies the test cases according to the functional specification of the unit, including the expected results. He can use abstract data for this (e.g. color = green). The set of test cases is handed over to the developer of the software. For him, it should be no problem to set the input variables to the required values (e.g. the appropriate RGB value for green). If a test fails, the developer can immediately correct the problem and re-run all tests that have passed so far (regression testing). Testing is seen as an additional step during the implementation of software, in comparison to the compiling step, where the compiler finds all syntax errors, and the developer corrects them interactively, verifying his changes by subsequent compiler runs.

However, standards require the organizational separation of development and test, due to the initial mentioned reason of blindness against own faults. Possibly, it could be sufficient to

only separate the specification of the test cases from the development, and to consider the conduction of predefined test cases not to suffer under the above mentioned blindness.

Furthermore, a developer´s time is often considered as too valuable to be wasted on testing, which is why developer testing is not found often in practice. However, this is going to be reconsidered.

3.1.3.5 What is special for testing embedded software?

For embedded software it is essential that the unchanged source code with all the non-ANSI keywords and non-ANSI peculiarities is used for testing. For instance, some cross compiler for embedded systems allow for bit fields that are smaller than the integer size, e.g. 8-bit wide bit fields in a 16-bit application. This is forbidden by the ANSI C standard, but justifiable by the perfect adaptation to the embedded system. Naturally, the unit test results are worthless, if this illegal size cannot be maintained during the tests. This requires specialized tools.

Furthermore, it is also essential that the concluding tests at least execute on the actual hardware, i.e. the embedded microcontroller. This is a challenge, but there are ways to attenuate this. Using a cross compiler for the microcontroller in question is a prerequisite, preferably the exact version that will be used also for the user application.

3.1.4 Methods for unit testing

Unit test tools can follow two technical approaches towards unit test: The test application approach uses a special application for conducting the unit tests. This is the usual approach. The original binary test uses the unchanged user application for testing.

3.1.4.1 a. Test application

The usual method for unit test tools to conduct unit tests is to generate a test driver (also called test harness) and compile the test driver together with the source code of the unit under test. Together they form the test application. The test driver includes startup code for the embedded microcontroller, the main() function entry, and a call to the unit under test. If required, the test driver contains also code for stub functions and the like. For each unit to test, an own test application is created. This test application is used to conduct the unit tests. For that, the test application is loaded into an execution environment capable of executing the test application. This execution environment is normally a debugger connected to an (instruction set) simulator, an in-circuit emulator stand-alone or connected to a target system, a JTAG or

BDM debugger or the like. After test data is transferred to the execution environment, (the test data may already be included in the test application), tests are conducted and the results are evaluated.

To execute the test application on the actual hardware, the test application must not only be compiled using a cross compiler for the microcontroller in question, but also the test application must fit into the memory present on the actual hardware. Also, the startup code of the test application must take into account peculiarities of the actual hardware, e.g. the enabling of chip selects and the like. Making the test application fit into memory can be simplified by using an in-circuit emulator, which provides emulation memory, and serves as a kind of generalized hardware platform for the microcontroller in question.

When the actual hardware has to be used and if memory on this hardware is very limited, the test application must be minimized to fit into this memory. This is especially challenging for single chip applications, where only the internal memory of the microcontroller is available. If test data is included in the test application (and memory is limited), a single test application can only include a few test cases, which in turn means several test applications for the test of one unit, which is cumbersome. An approach which avoids this, keeps the test data separated from the test application, which allows not only for a minimized test application, but also allows you to change the test data without having to regenerate the test application.

3.1.4.2 b. Original binary test

Another approach is to use the unchanged user application for unit testing. This resembles the manual test that is usually done by a developer after the application is completed. The complete application is loaded into the execution environment, and the application is executed until the unit to be tested is eventually reached. Then the input variables are set to the required values, and the test is conducted.

3.1.4.3 Pros and cons

The advantage of the Original Binary Test approach is that the unit under test is tested exactly in its final memory location. There is no extra effort (or hassle) for compiling and linking a test application, because the user application is used, which is already compiled and linked or had to be compiled and linked anyway. Because the user application must fit in the memory anyway, problems regarding the size of the application can be neglected. Even applications that already reside in the ROM of the hardware can be tested. Even if the cross compiler used to compile the user application is no longer at hand, tests are still feasible.

However, this Original Binary Test approach has some disadvantages compared to using a test application:

- There is no control over the test execution. It depends on the user application, when the unit under test is reached. It may be the case that the unit under test is never reached, or only after some special external event has happened, e.g. the push of a button of the actual hardware and an interrupt resulting from this.
- During the Original Binary Test, stub functions cannot be used. This is clear because the application is already linked using the current functions that are called by the unit under test. A unit is always tested using the other units of the application. Therefore, the unit under test is not isolated from the rest of the application, and errors of called units may show up during the test of the unit under test.
- It is not possible to use arbitrary test data for the unit test. For instance, if the unit under test gets its test data by a pointer pointing to a memory area, the amount test data must fit into this memory area, which was allocated by the user application.

Apart from its easy usage, which possibly could be the only means to do some unit testing at all, the Original Binary Test has strong disadvantages, which are essential for proper unit testing and therefore one could even insist that it is not a unit test in its strictest sense.

3.1.5 Conclusion

Besides being required by standards, unit testing reduces the complexity of testing, finds errors early, saves money, and gives confidence for the test of the whole application. If used in the right way, unit testing can reduce development/test time and therefore reduce the time-to-market. To conduct regression tests, test automation is indispensable. This requires tool support.

3.2 The Classification Tree Method (CTM)

The objective of the Classification Tree Method (CTM) is to transform a (functional) definition of a problem systematically into a set of error-sensitive, low redundancy set of test case specifications. This document gives a comprehensive overview of the CTM.

3.2.1 General

Testing is a compulsory step in the software development process. The planning of such testing often raises the same questions:

- How many tests should be run?
- What test data should be used?
- How can error-sensitive tests be created?
- How can redundant tests be avoided?
- Have any test cases been overlooked?
- When is it safe to end testing?

Anyone who has been confronted with such issues will be glad to know that the CTM offers a systematic procedure to create test case specifications based on a problem definition.

The objective of the CTM is to transform a (functional) definition of a problem systematically into a set of error-sensitive, low redundancy set of test case specifications. The systematic approach yields a high probability that the resulting set of test specifications is complete and no relevant tests are overlooked. Naturally, correct usage of the method and an appropriate integration in the development process are prerequisites. Having a complete set of tests gives evidence when it is safe to end testing.

The CTM is applied by a human being. Therefore, the outcome of the method depends on the experiences, reflections, and appraisals of the user of the CTM. Most probably two different users will come out with a different set of test case specifications for the same functional problem. Both sets could be considered to be correct, because there is no absolute correctness. It should be clear that there are set of test cases that are definitively wrong or incomplete. Because of the human user, errors cannot be avoided. One remedy is the systematic inherent in the method. This systematic guides the user and stimulates his creativity. The user shall specify test cases with a high probability to detect a fault in the test object. Such test cases are called error-sensitive test cases. On the other hand, the user shall avoid that too many test cases are specified, that are superfluous, i.e. do not increase test intensiveness

or test relevance. Such test cases are called “redundant” test cases. It is advantageous, if the user is familiar with the field of application the method is applied in.

The CTM is a general method: It can not only be applied to module/unit testing of embedded software, but to software testing in general and also to functional testing of problems, that are not software related. The prerequisite to apply the method is to have available a functional specification of the behavior of the test object. The CTM incorporates several well-known approaches for test case specification, e.g. equivalent partitioning, and boundary value analysis.

The CTM stems from the former software research laboratory of Daimler in Berlin, Germany.

3.2.2 Steps to take

a. Defining the functional problem

The first step is to describe the expected behavior of the test object, e.g. “If the button is pushed, the light will go on; if the button is released, the light will go off”. Data processing software normally solves functional problems, since input data is processed according to an algorithm (the function) to become output data (the solution).

b. Determining the test-relevant aspects

Analyze the functional specification. This means, you think about this specification with the objective to figure out the test-relevant aspects of the specification. An aspect is considered relevant if the user expects that aspect to influence the behavior of the test object during the test. In other words, an aspect is considered relevant if the user wants to use different values for this aspect during testing. To draw the tree, these aspects are worked on separately. This reduces the complexity of the original problem considerably, what is one of the advantages of the CTM.

Example for a test-relevant aspect

Consider systems that measures distances in a range of some meters, e.g. the distance to a wall in a room. Those systems usually send out signals and measure the time until they receive the reflected signal. Those systems can base on two different physical effects: One can use sonar to determine the distance, whereas the other can use radar.

The question is now: Is the temperature of the air in the room a test relevant aspect for the test of these measurement systems? The answer is yes for one system and no for the other:

The speed of sound in air (sonar) is dependent on the temperature of the air. Therefore, to get exact results, the sonar system takes this temperature into account during the calculation of the distance. To test if this is working correct, you have to do some tests at different temperatures. Therefore, the temperature is a test-relevant aspect for the sonar system.

On the other hand we all know that the speed of a radar signal, that travels at the speed of light, is independent from the temperature of the air it travels in (it did not even need air to travel). Therefore, the temperature of the air is not a test-relevant aspect for the testing of the radar system. It would be superfluous to do testing at different temperatures.

This example shows that it needs careful thinking to figure out (all) test relevant aspects. It would lead to poor testing if someone simply takes the test cases for the radar system and applies them to the sonar system without adding some temperature-related test cases. Additionally, this example illustrates that it is advantageous to have some familiarity with the problem field at hand when designing test cases.

c. Classifying the values of a test-relevant aspect

After all test relevant aspects are determined, the values that each aspect may take are considered. The values are divided into classes according to the equivalence partitioning method: Values are assigned to the same class, if the values are considered equivalent for the test. Equivalent for the test means that if one value out of a certain class causes a test case to fail and hence reveals an error, every other value out of this class will also cause the same test to fail and will reveal the same error.

In other words: It is not relevant for testing which value out of a class is used for testing, because they all are considered to be equivalent. Therefore, you may take an arbitrary value out of a class for testing, even the same value for all tests, without decreasing the value of the tests. However, the prerequisite for this is that the equivalence partitioning was done correctly, what is in the responsibility of the (human) user of the CTM.

Please note:

- Equivalent for the test does not necessarily mean that the result of the test (e.g. a calculated value) is the same for all values in a class.
- Equivalence partitioning must be complete in mathematical sense: Every possible value of a test relevant aspect must be assigned to a class.
- Equivalence partitioning must be unique in mathematical sense: A value of a test relevant aspect must be assigned to a single class, and not to several classes.

Example for equivalence partitioning: Ice Warning

An ice warning indication in the dashboard of a car shall be tested. This ice warning indication depends on the temperature reported by a temperature sensor at the outside of the car, which can report temperatures from -60°C to +80°C. At temperatures above 3°C the ice warning shall be off, at lower temperatures it shall be on.

It is obvious that the temperature is the only test-relevant aspect. To have an reasonable testing effort, we do not want to have a test case for every possible temperature value. Therefore, all possible temperature values need to be classified according to the equivalence partitioning method.

It is best practice to find out if invalid values may be possible. In our case a short circuit or an interruption of the cable could result in an invalid value. Therefore, we should divide the temperature in valid and invalid values first. The invalid values can relate to temperatures that are too high (higher than 80°C) and to ones that are too low (lower than -60°C). It is tempting to form two classes out of the valid temperatures: The first class shall contain all values that result in the ice warning display being on (from -60°C to 3°C) and the other class shall contain all values that result in the ice warning display being off (from 3°C to 80°C).

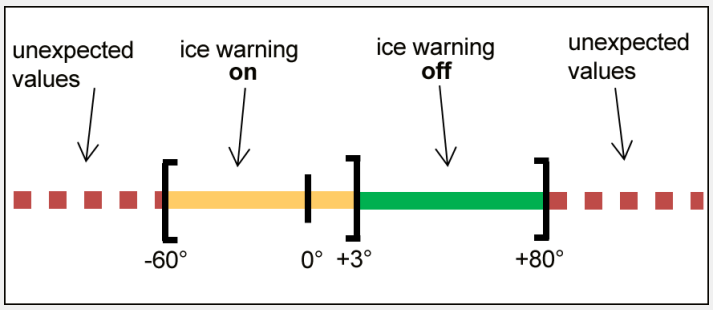


Figure 3.1: Initial equivalence partitioning for “ice warning”

The equivalence partitioning in the figure above leads to at least four test cases, because we need to take a value out of each class for the tests.

d. Repeating equivalence partitioning

An equivalence class can be sub-divided according to additional aspects. This equivalence partitioning on several levels reduces the complexity of equivalence partitioning, because you can consider each class isolated from the other classes and decide, if and how it needs to be sub-divided or not. Furthermore, this equivalence partitioning on several levels documents the thoughts resp. stages of work until the final equivalence partition. This serves understandability and traceability of the result. Also it allows easily reverting steps if the final

equivalence partition has become too fine granulated.

Example for repeated equivalence partitioning

For the example ice warning, the classification of the valid values is not detailed enough, because according to the equivalence partitioning method, it would be sufficient to use a single, arbitrary value out of a class for all the tests. This could be for instance the value 2°C out of the class of temperatures, for which the ice warning display is on. In consequence, no test with a minus temperature would check if the ice warning display is on. To avoid this consequence, you could divide this class further according to the sign of the temperature:

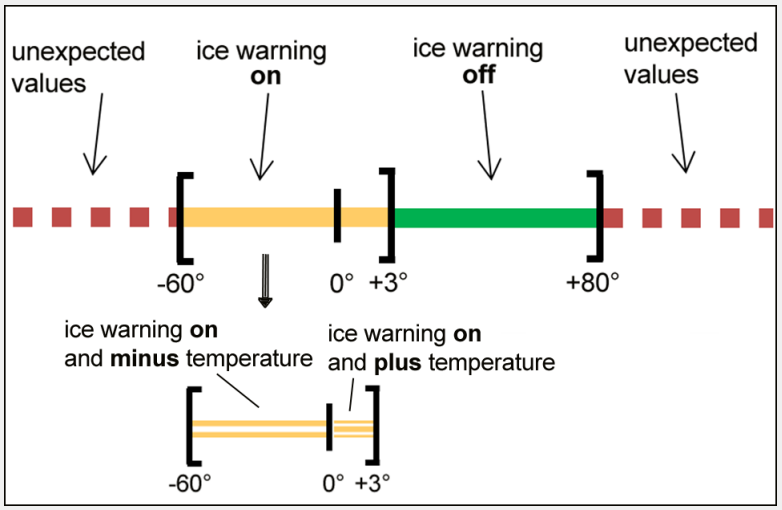


Figure 3.2: Repeated equivalence partitioning for “ice warning”

Result: Classification tree

Using the CTM, the result of the repetition of equivalence partitioning for all test relevant aspects is depicted in the CT. The root represents the functional problem, the test relevant aspects. Test relevant aspects (classifications) are drawn in nodes depicted by rectangles. Classes are ellipses.

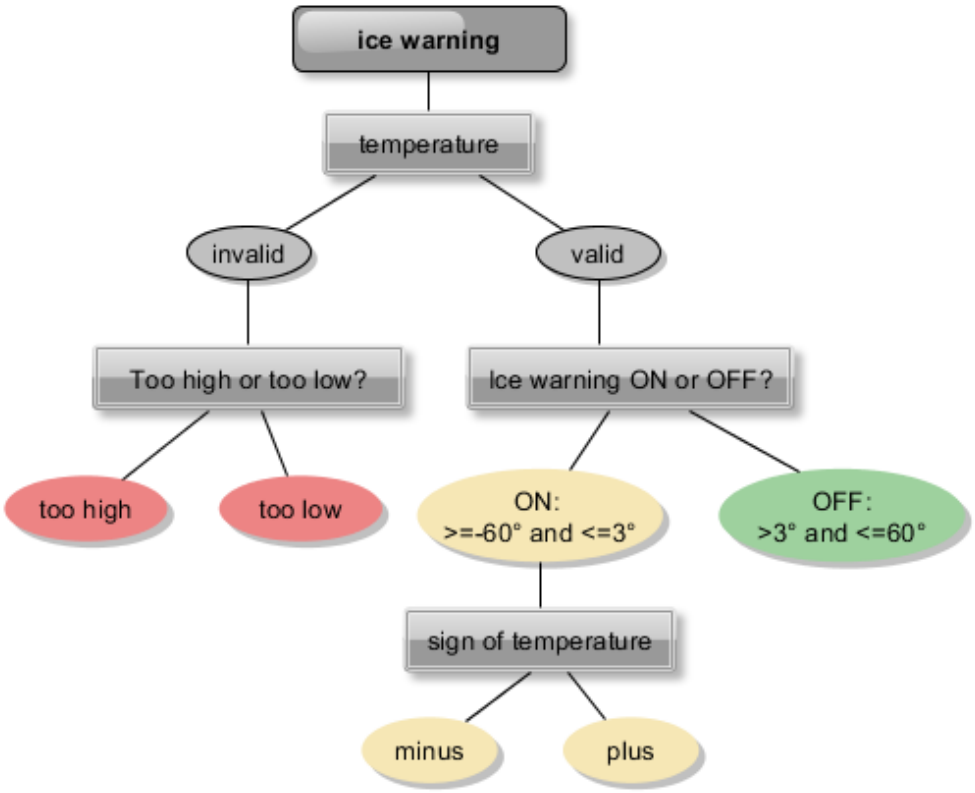


Figure 3.3: A possible CT for “ice warning”

e. Using boundary values

The idea behind using boundary values is that values at the borders of a range of values are better suited to form error-sensitive test cases than values in the middle. The idea behind boundary values analysis is contrary to equivalence partitioning, because one method takes a set of values as equivalent and the other method prefers special values in such a set. Despite the fact that the idea behind boundary values analysis is exactly the opposite of equivalence partitioning, both approaches can be expressed in the CTM.

f. Testing a hysteresis

The current problem specification of the ice warning-example does not mention hysteresis. It may be tempting to extend the current problem specification in that fast changes in the state of the ice warning display shall be avoided. For instance, the ice warning display shall be switched off only after the temperature has risen to more than 4°C. This could be realized by a hysteresis function. The necessary test cases for such a hysteresis function can be specified by the CTM.

g. Specifying test cases

Test cases are specified in the so-called combination table below the CT. The leaf classes of the CT form the head of the combination table. A line in the combination table depicts a test case. The test case is specified by selecting leaf classes, from which values for the test case shall be used. This is done by the user of the method, by setting markers in the line of the respective test cases in the combination table.

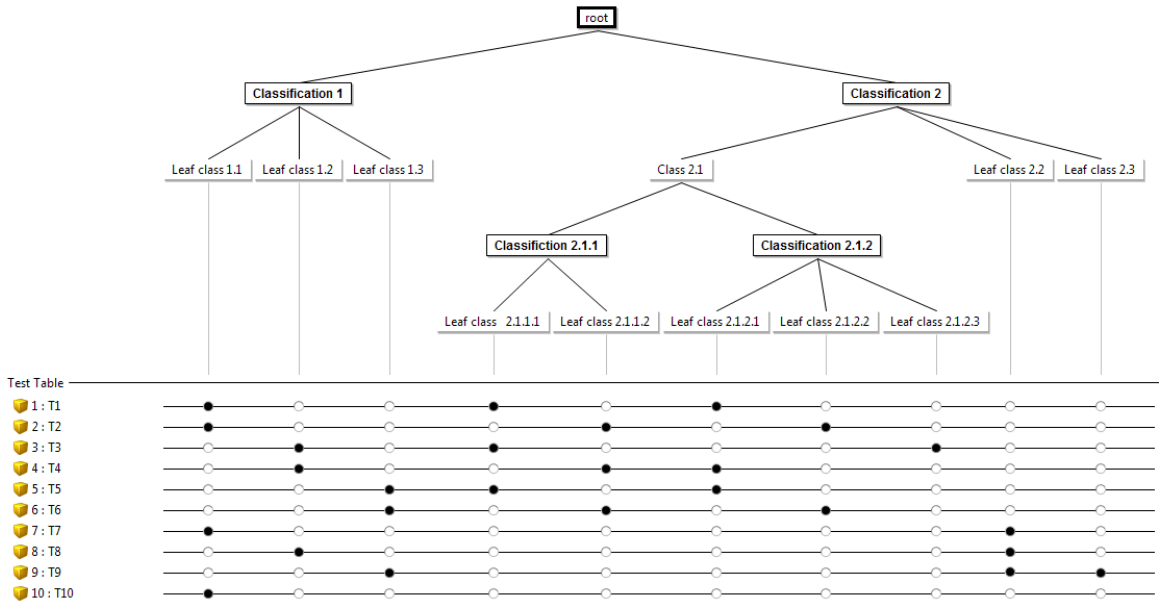


Figure 3.4: Result of the CTM: tree (above) with combination table (below)

It may be tempting to combine every class with every other class during the specification of the test cases. Besides the fact, that not every combination might be possible for logical reasons, it is not the intention of the CTM to do so, it could be done automatically by a tool. This would lead to many test cases, with the disadvantages of loss of overview and too much effort for executing the test cases.

The objective of the CTM is to find a minimal, non-redundant but sufficient set of test cases by trying to cover several aspects in a single test case, whenever possible. Similar to the drawing of the tree, it depends on the appraisal and experience of the user of the method, how many and which test cases are specified.

Obviously the size of the tree influences the number of test cases needed:

A tree with more leaf classes naturally results in more test cases than a tree with less leaf classes. The number of leaf classes needed at least for a given tree is called the *minimum criterion*. It can be calculated from the consideration that each leaf class should be marked in

at least one test case, and that some leaf classes cannot be combined in a single test case, because the classes exclude each other.

Similar a *maximum criterion* can be calculated, which gives the maximal number of test cases for a given CT. A rule of thumb states that the number of leaf classes of the tree gives the order of magnitude for the number of test cases required for a reasonable coverage of the given tree.

3.2.3 Example is_value_in_range

Problem definition:

A start value and a length define a range of values. Determine if a given value is within the defined range or not. Only integer numbers are to be considered.

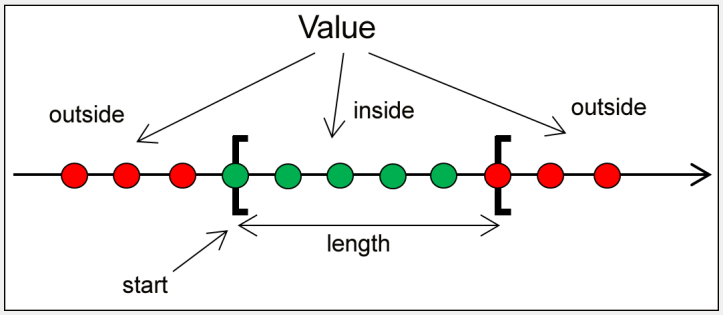


Figure 3.5: The problem “is_value_in_range” depicted graphically

It is obvious, that completed testing is practically impossible, because we get 65536 * 65536 * 65536 = 281.474.976.710.656 test cases, even if we assume only 16 bit integers. If we would assume 32 bit integers …well, we better do not.

3.2.3.1 Test-relevant aspects

The start of the range and the length can be regarded as test relevant aspects. This is convenient since, according to the problem definition, a range of values is defined by a start value and a length. It reflects the intention to use different values for the start and the length during testing.

We should have some test cases, which result in inside, and other test cases which result in outside. We call the corresponding aspect *position*, because the position of the value under test with respect to the range determines the result. So the three test-relevant aspects to be used for classifications are initial *value*, *length* and *position* and they thus form the basis of the CT:

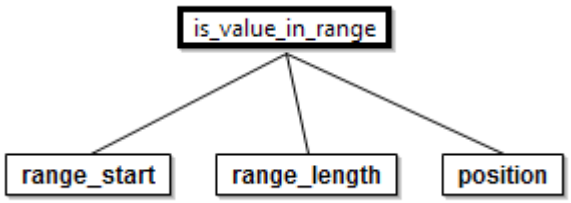


Figure 3.6: The initial CT with three test-relevant aspects

3.2.3.2 Forming classes

Now classes are formed for the base classifications according to the equivalence partitioning method. Usually, the problem specification gives us hints how to form the classes. E.g. if the problem specification would state: “If the start value is greater than 20, the length value doubles”, then we should form a class for start values greater than 20 and a class for start values smaller or equal to 20.

Unfortunately, the problem specification at hand is too simple to give us similar hints. However, since the start value can take on all integer numbers, it would be reasonable to form a class for positive values, a class for negative values, and another class for the value zero. It would also be reasonable to form just two classes, e.g. one class for positive start values including zero and the other class for negative start values. This depends on ones emphasis having zero as value for the start of the range in a test case or not.

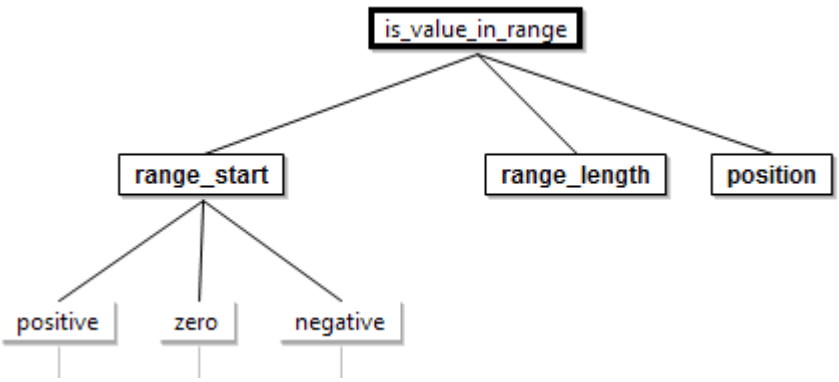


Figure 3.7: The CT for is_value_in_range, 2nd step

Because of the systematic inherent in the CTM, and because *range_length* is an integer as well as *range_start*, it is stringent to use for *range_length* the same classes as for *range_start*. This results in the following tree:

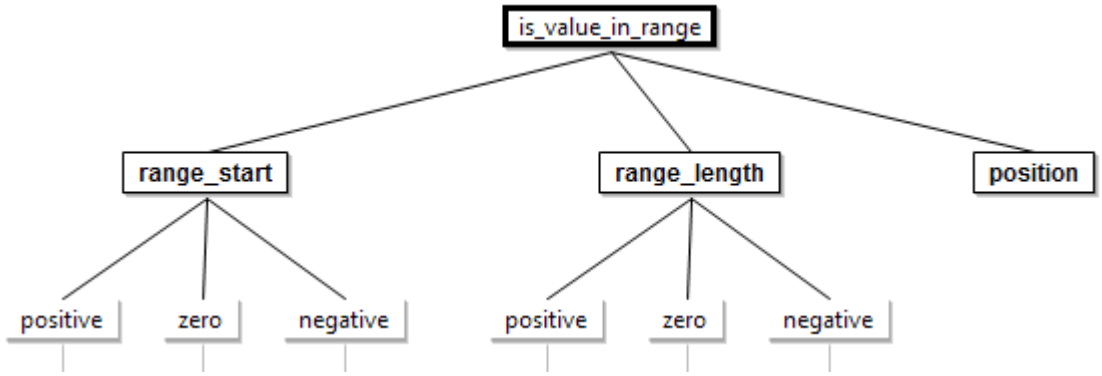


Figure 3.8: The CT for is_value_in_range, 3rd step

3.2.3.3 A first range specification

To specify a first range (to be used in the first test case), we have to insert a line in the combination table and to set markers on that line:

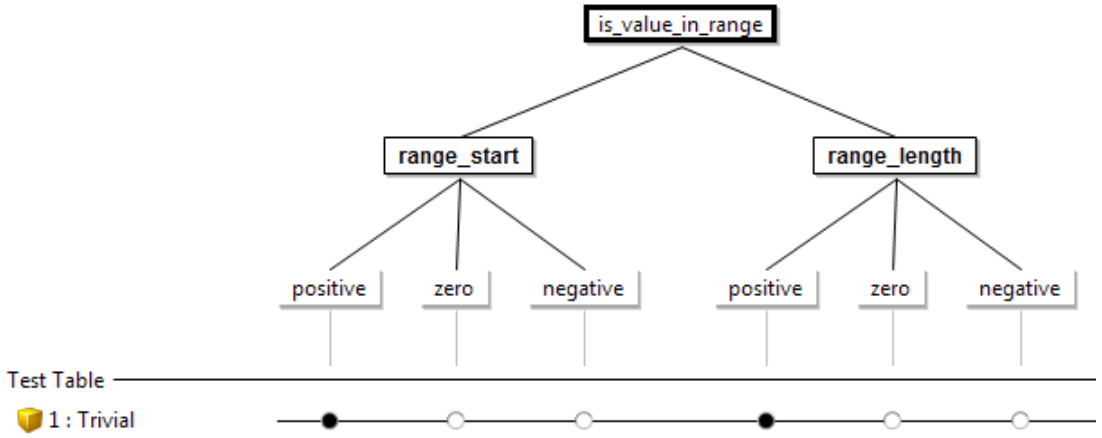


Figure 3.9: A first specification for the range in the combination table

Two markers are set on the line for the first specification. One marker selects the class *positive* for the start of the range. The other marker selects the class *positive* for the length of the range. A range with the start value of, say, 5 and a length of 2 would accord to the specification. This first specification was named *trivial*.

3.2.3.4 A second range specification

We can insert a second line in the combination table and specify a much more interesting tests case:

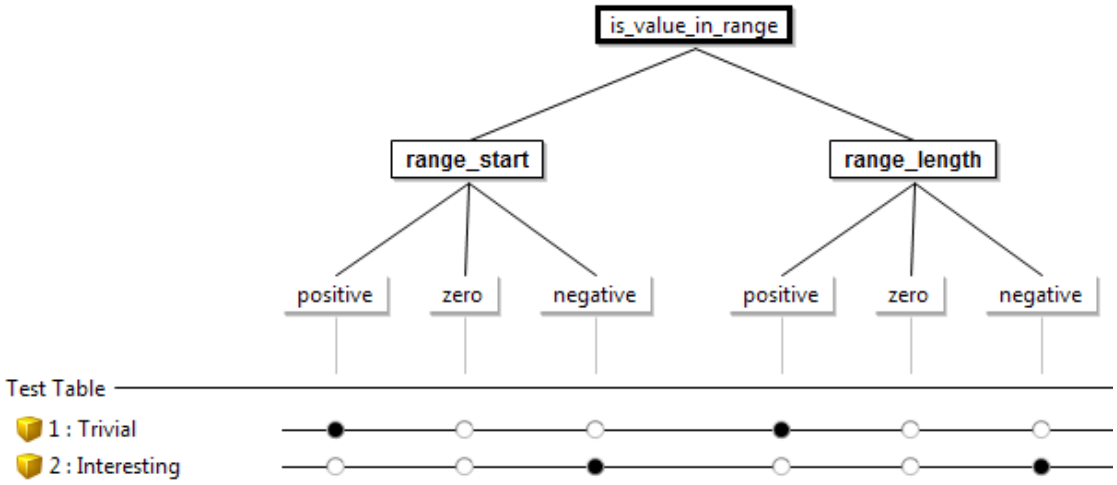


Figure 3.10: A second specification for the range in the combination table

For the second specification again two markers are set. They specify that a negative value shall be used both for the start and for the end of the range. Hence a range with the start value of -5 and a length of -2 would accord to the second specification. But this value pair raises some questions: Shall the value -6 lie inside the range? Or shall the value -4 lie inside the range? Or shall no value at all lie inside the range, if the length of the range is negative? Each opinion has its supporters and it is hard to decide what is to be considered correct. Actually, at this point it is out of our competence to decide what is correct. We have found a problem of the specification!

It is a valuable result to find a problem (omission or contradiction) in the functional problem specification, and that it was achieved in the case during test case specification for the functional problem. It is generally more likely to detect a problem in the functional specification if the test case specification is systematic. The CTM is a systematic method for test case specification. Hence, the CTM provides good means to detect problems in the functional problem specification.

Probably a test case using a negative length would not have been used if the test case specification would have been done spontaneous and non-systematic. But a negative length is completely legal for the functional problem specification that was given above. If you consider

that the problem specification at hand was a very simple one, you may imagine how likely it is to overlook a problem in a more comprehensive and complicated problem specification.

3.2.3.5 Extending the tree by a boundary class

In case we are not satisfied with the fact that a fixed single positive value, e.g. 5, may serve as value for the start of the range in all test cases, we can sub-divide the class *positive* according to a suitable classification. In our example, we classify according to the size. The idea behind this is to have a class containing only a single value, in our case the highest positive value existing in the given integer range. We use this value because it is an extreme value, and as we know, using extreme values (or boundary values) in test cases is well-suited to produce error-sensitive (or interesting) test cases.

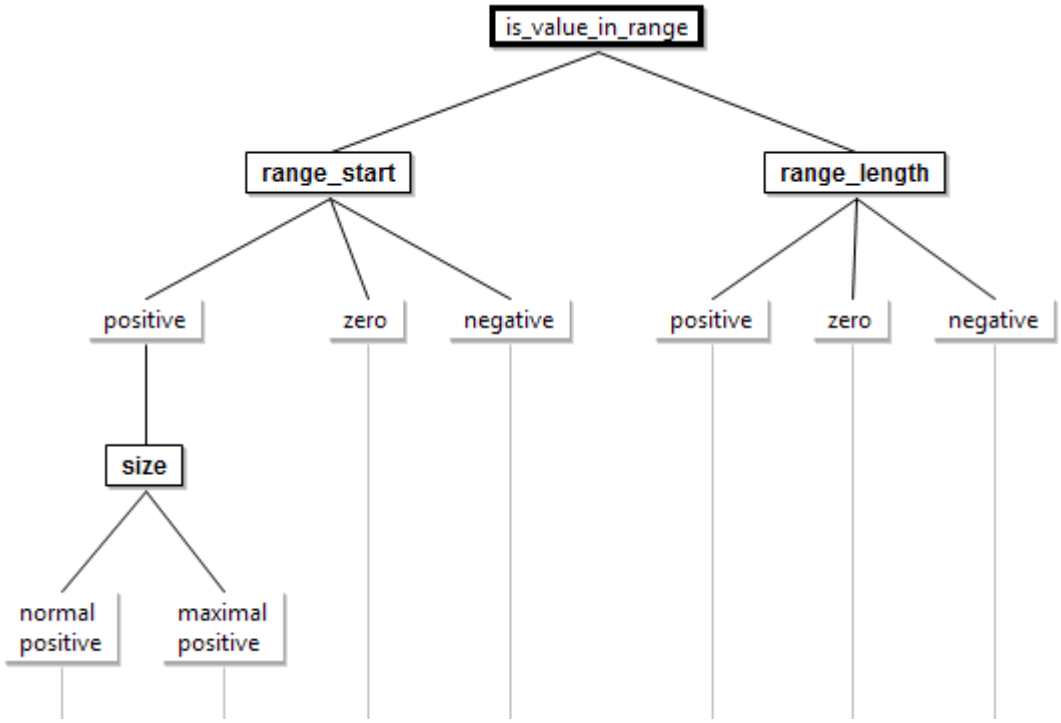


Figure 3.11: The CT for is_value_in_range, 4th step

In the figure above, the positive values for the start of the range are subdivided according to their size.

This results in the two classes *normal positive* and *maximal positive*. The class *maximal positive* holds the highest possible positive value (i.e. MAX_INT), and the class *normal positive* holds all other positive values. This satisfies mathematical completeness.

Remark 1: Another possibility to classify the positive start values would have been for instance to classify in odd and even values. This would have been completely legal. This would have been probably also sensible for e.g. a problem of number theory, but not target-oriented for the problem at hand.

Remark 2: Please note that for the moment we do not know and we need not to know the size (in bits) of the integers used in the problem at hand. We simply specify "the highest positive value in the given integer range". This keeps our test case specification abstract! E.g. our test case specification is appropriate for any integer size. As soon as we assume we use e.g. 16 bit integers, and therefore parameterize our test case by specifying 32767 as value in the class *maximal positive*, we loose this abstraction. E.g. if we port the parameterized test case to a 32 bit integer system, the test case looses its sense. This is not the case if we port the abstract test case specification.

3.2.3.6 Another interesting test case specification

With the CT extended according to figure 3.11 The CT for is_value_in_range, 4th step, we can insert an additional line in the combination table and specify again an interesting range for a third test case:

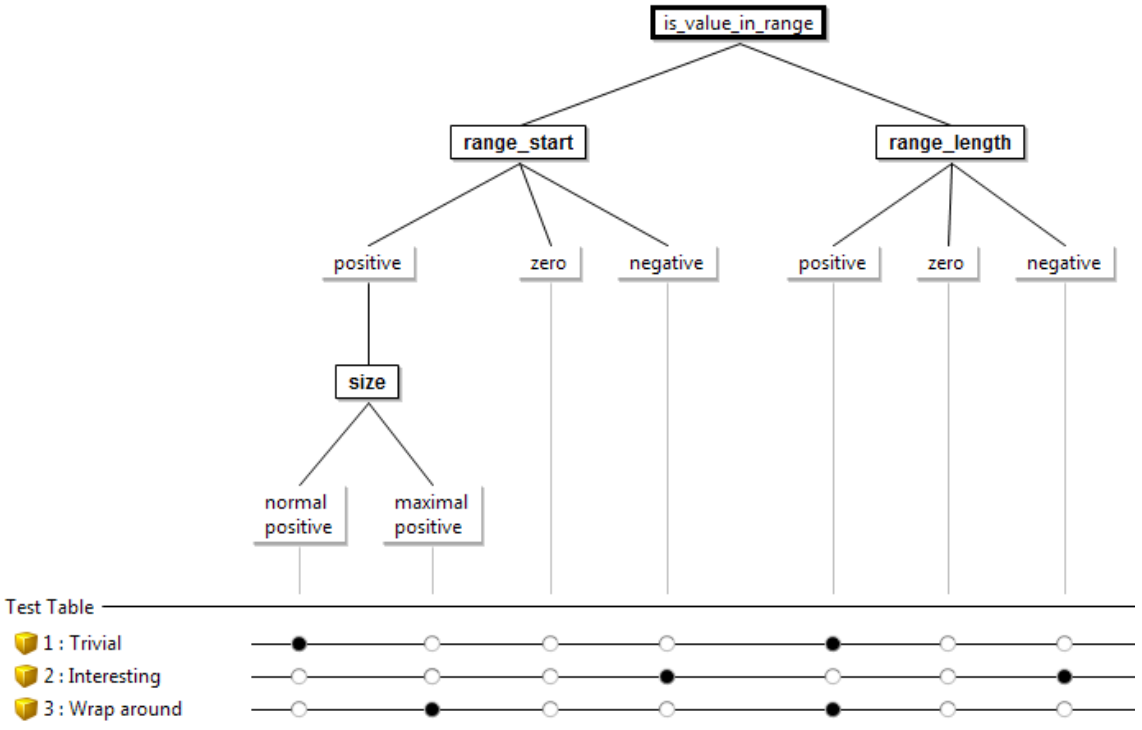


Figure 3.12: The third range specification provokes a wrap-around

The third range specification in the figure above combines the highest positive number for the start value of the range with a positive length, i.e. the range exceeds the given integer range.

The situation with the third range specification is similar to the situation depicted in the figure above. The situation raises some questions: Will the situation be handled sensible and gracefully by the test object? Or will it crash due to the overflow? Will the negative values on the left hand side be accounted to lie inside the range or not? And what is correct with respect to the last question? The problem specification above does not give an answer to the latter question, again we have found a weak point in the problem specification.

To sum up, designing test cases according to the CT method has revealed two problems of the problem specification and has lead to interesting test cases so far.

3.2.3.7 The completed classification tree

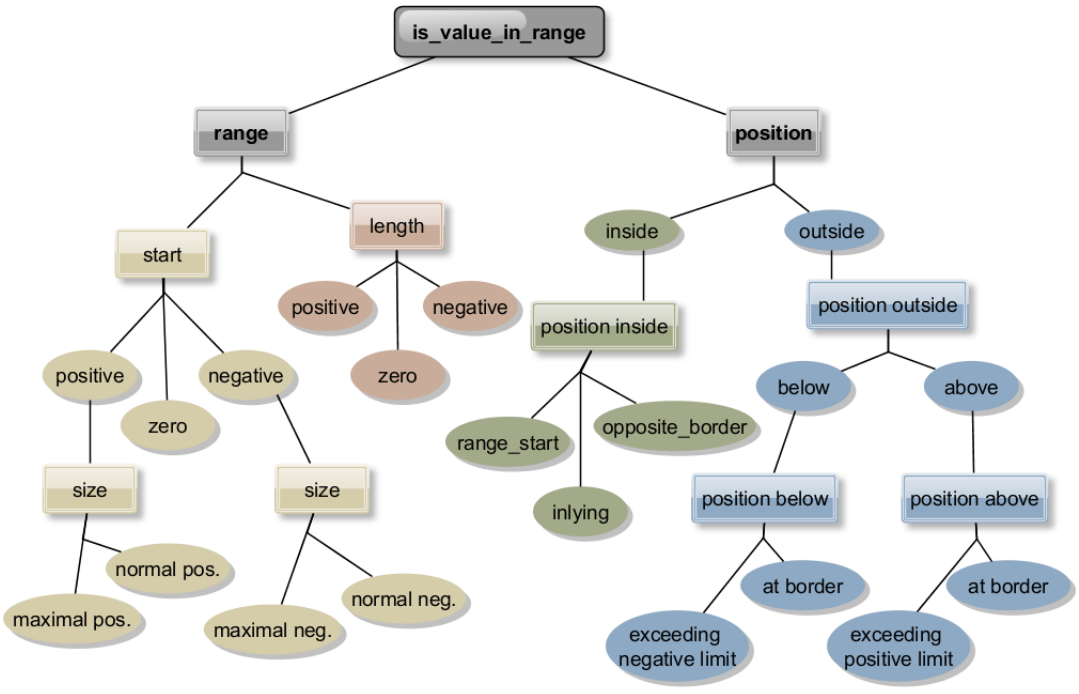


Figure 3.13: The completed CT for is_value_in_range

In the figure above, one possible completed CT is depicted. Classifications are depicted by rectangles, classes by ellipses. The “range” node is a composition with two classifications as child elements.

This tree is discussed in the following:

- Analogous to the class *maximal positive* for the start value of the range, a class *maximal negative* is introduced. The idea behind this class is to combine the maximal negative start value with a negative length of the range, what shall provoke an underflow or negative wraparound. This idea comes from the systematic in the CTM: If a positive wrap-around is seen as an interesting test case, also a negative wrap-around should be exercised.
- An example for a composition is given by *range*. A composition may be used for a relation “consists of”. In our case, the range consists of a *start* value and a *length*.
- The final tree features still the three initial classes *positive*, *zero*, and *negative* for the length of the range. It is important to note that the tree reveals at a glance that nothing like *maximal positive* length or similar is considered to be useful for the testing problem at hand.
- It is obvious that a position can either be *inside* or *outside* the range, hence this classification suggests itself. Furthermore, it is obvious that there are two different areas outside the range: *below* the range and *above* the range. This is reflected in the classification *position outside*. (If the tree would miss such a classification, it may well be considered incorrect).
- The class *inside* of the classification *position* could well be a leaf class of the classification tree. However, in the CT in the figure above, this class is subdivided further in the sub-classes *range_start*, *opposite_border*, and *inlying*. This is done to force the use of boundary values in the test cases. If a test case specification selects the class *range_start*, the value that shall be checked if it is inside the range or not shall take the value of the start of the range, that is the lowest value that is considered to be inside the range, a boundary value. The class *opposite_border* is intended to create an analogous test case specification, but using the highest value that is considered to be inside the range. The class *range_start* and the class *opposite_border* both contain only a single value. All other values inside the range are collected in the class *inlying*; this class exists mainly because of the requirement for completeness of equivalence partitioning. A similar approach to use boundary values is visible in the classes *at border* for positions outside the range.

3.2.3.8 The completed test case specification

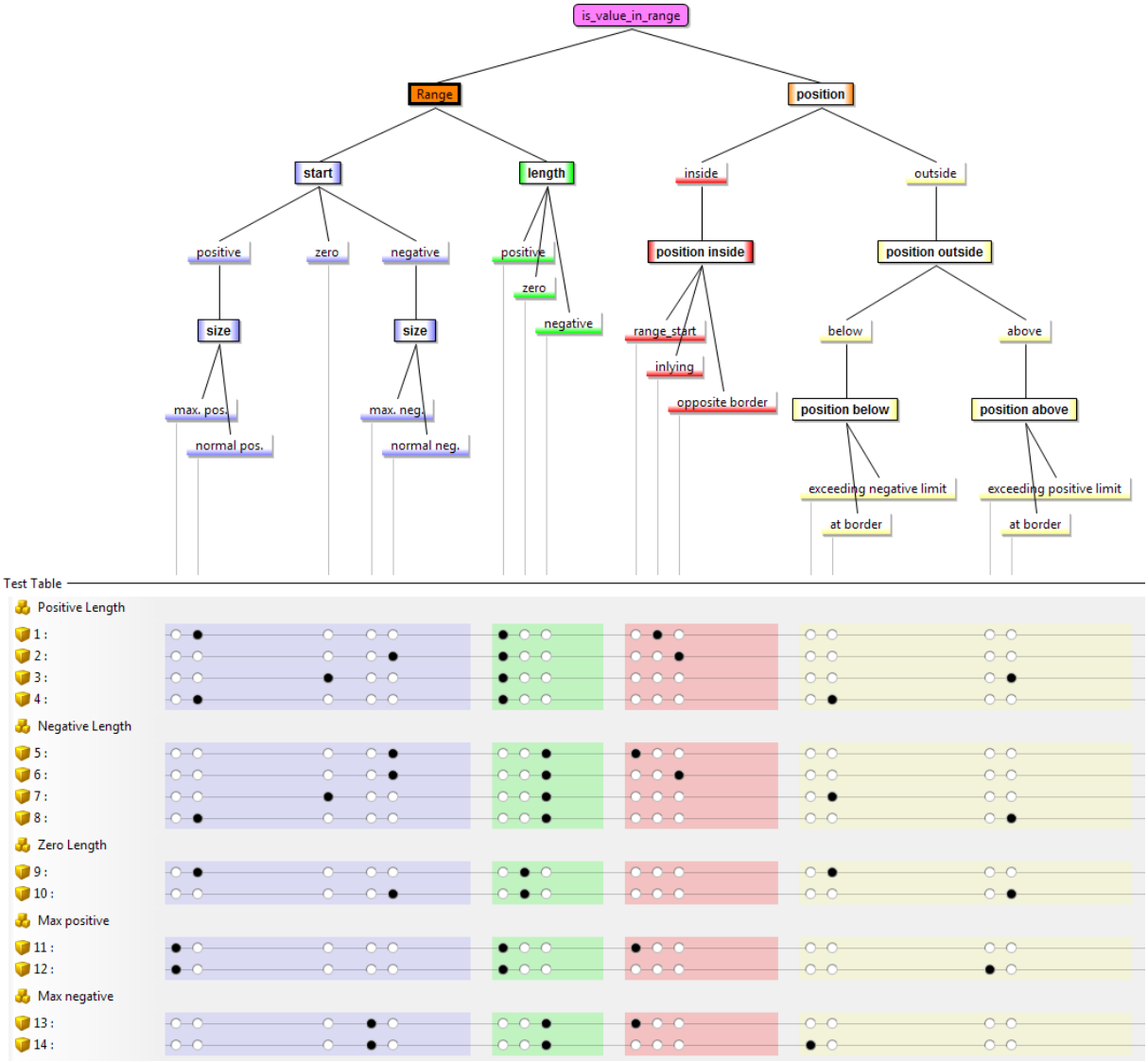


Figure 3.14: The completed test case specification

The test case specification above lists 14 test cases. Please note that these are specified by the user and depend on its judgment. Based on the CT it is possible for some values to be determined that provide clues to the number of test cases required.

The first value is the number of test cases, if each leaf class is included at least once in a test case specification. This number is known as the minimum criterion. In our example, the largest amount of leaf classes, namely seven, belong to the base classification *position*. Seven is thus the value of the minimum criterion. The maximum criterion is the number of test cases that results when all permitted combinations of leaf classes are considered.

In our example, the maximum criterion amounts to 105 (i.e. 5 * 3 * 7). The maximum criterion takes into account that it is not possible to select e.g. a negative length and a positive length for the same test case specification, because this is impossible by the construction of the tree. The maximum criterion does not take into account that it is not possible to select e.g. a zero length and *inlying*, because this is not impossible by the construction of the tree, but by the semantics of the function problem.

A reasonable number of test case specifications obviously lies somewhere between the minimum and the maximum criterion. As a rule of thumb, the total number of leaf classes gives an estimate for the number of test cases required to get sufficient test coverage. In the test case specification, the CT has 15 leaf classes, what fits well to 14 test cases.

By the test case specification in the figure above, you can deduct how the functional problem specification was extended with respect to the questions raised in sections "A second range specification" and "Another interesting test case specification":

- "If the length of the range is negative, are there values that can be inside the range?" The answer is "yes", because in test case specification no. 5 and no. 6 a negative length shall be used and the position of the value shall be inside the range.
- "If the length of the range exceeds the given integer range, shall negative values be inside the range?" Test case specification no. 12 clarifies that this should not be the case.

The leaf class *inlying* is selected for only one test case specification (no. 1). This reflects the fact that this class exists only because of the requirement for mathematical completeness of equivalence partitioning, and not because the inlying values are considered to produce error-sensitive test cases.

3.2.3.9 Another test case specification

Here is an alternative test case specification to the functional problem specification at hand depicted:

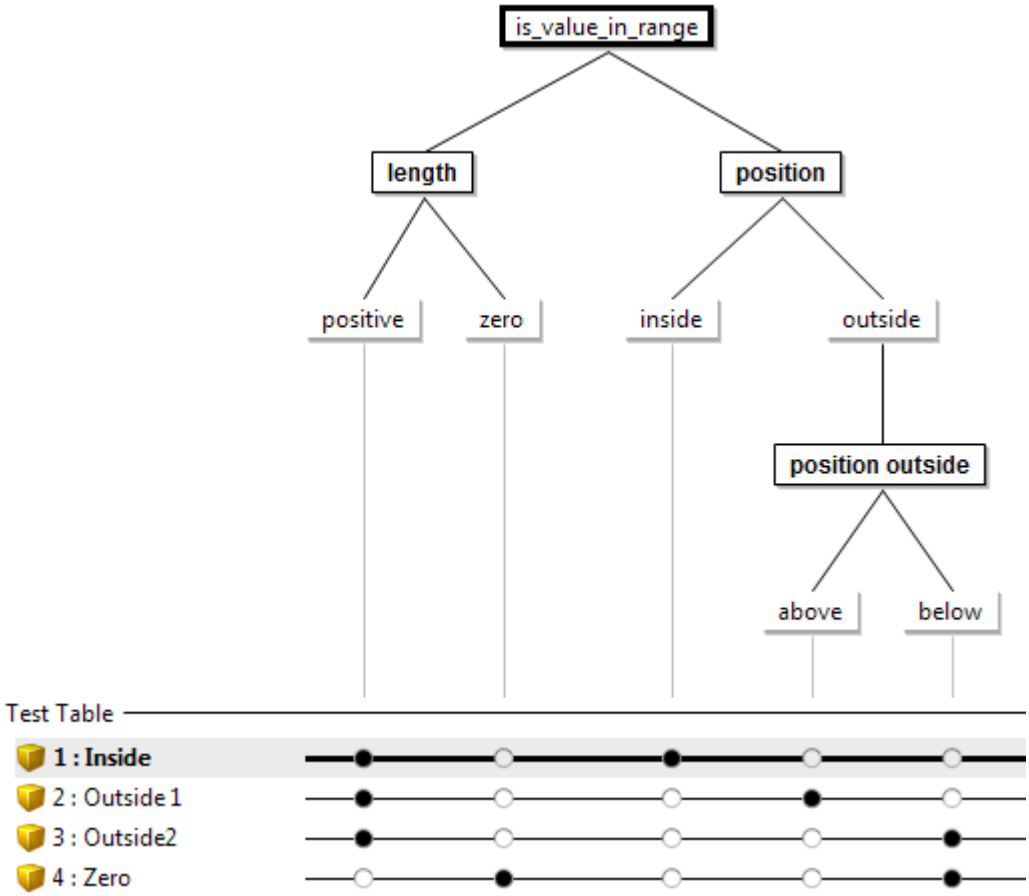


Figure 3.15: An alternative test case specification

What are the differences to the more elaborated test case specification in the section above?

- The start value of the range is not mentioned in the CT. This means, the start value is not considered to be a test-relevant aspect by the user of the CTM. In consequence, any arbitrary value can be used as start value in the four test cases. This value can be fix for all test cases, but does not have to be.
- The problem of a negative length is completely neglected. For the problem specification from section Problem which specifies a length to be an integer and hence also the length to be negative, this is a serious flaw.

- The problem of wrap-around is neglected. This may be considered to be an esoteric problem, and therefore it could be accepted that it is not mentioned in the alternative test case specification.
- The usage of boundary values is not forced by the alternative test case specification. This is questionable, because boundary values produce error-sensitive test cases. The alternative test case specification minimizes testing effort (by specifying only four test cases), but this is at the cost of thoroughly testing.

But the point is not which test case specification is better. The main point is:
Test case specification according to the CTM visualizes testing ideas!

4 Tutorial: General handling

This chapter explains how to create databases for your test, how to work with the different files and the graphical user interface of TESSY and provides some information about useful shortcuts to work more efficient.

4.1. Creating databases and working with the file system **53**
- 4.1.1. Creating a project database . . . 54
- 4.1.2. Creating, importing, cloning, editing, deleting a project . . . 60
- 4.1.3. Creating a template project . . . 61
- 4.1.4. Moving the project directory . . . 62
- 4.1.5. Handling with equally named projects . . . 62
- 4.1.6. Using a specific environment setting . . . 64
- 4.1.7. Updating the database . . . 64

4.2. Understanding the graphical user interface **66**
- 4.2.1. Menu bar . . . 67
- 4.2.2. Tool bar . . . 67
- 4.2.3. Perspectives and perspective (tool) bar . . . 67
- 4.2.4. Views . . . 68
- 4.2.5. Status bar . . . 72

4.3. Using the context menu and shortcuts **73**
- 4.3.1. Context menu . . . 73
- 4.3.2. Shortcuts . . . 73

4.1 Creating databases and working with the file system

The following table provides a fast overview about TESSY's file system and databases:

Database element	Function
Workspace	Contains all Eclipse-related settings for TESSY (layout/size of windows/perspectives/views) and the list of projects (the file projects.xml). You can close TESSY, move the projects.xml somewhere else, delete the entire directory, restart TESSY and restore the projects.xml.
tessy.pdbx (file)	Project file for the location of a TESSY project. The project can be opened via double click. Contains the basic settings of a project and can be renamed.
Project root	Specifies the root directory of your project, so that all paths (e.g. sources, includes, etc.) can be related to this root. Every project will have an own project root. The project root as an absolute path is intentionally not saved within the project file (tessy.pdbx) which allows you to transfer projects to other computers. The location of the project root will be detected automatically by TESSY when opening a project. This is done by matching the current absolute location of the PDBX file with the relative path entry of the database location stored within the PDBX file.
Source root	Optional directory to specify source and include paths to this source root independently from the project root (e.g. if you want source files to reside in another directory outside the project root). The source root as an absolute path is intentionally not saved in the project file (tessy.pdbx), only its existence is indicated. Therefore the source root needs to be selected when opening a project on a different computer. When opening such a project using the command line, the source root needs to be provided as command line argument. For more information about the CLI mode please refer to section 6.16 Command line interface.
Configuration file	Contains appropriate target environments for the project database which can be defined with the Test Environment Editor and will be stored by default under `[project root]/tessy/config`.

continue next page

Database element	Function
persist (folder)	Contains the databases for the project, one for requirements and test collections, the other one for test data.
work (folder)	Contains all temporary files generated during the test process. This entire directory can be deleted without loosing any data of the TESSY project.
TMB files	Format to store your backups, usually under `[project root]/tessy/backup`.

Table 4.1: File system and databases of TESSY

4.1.1 Creating a project database

→ Start TESSY by selecting "All Programs" > "TESSY 4.x" > "TESSY 4.x".
Loading TESSY will take a few seconds.
The Select Project dialog will open. At top you can see the path of your workspace (see figure 4.1).

If you already have opened a project before, TESSY will automatically open the last project again on startup. To create a new project, select "File" > "Select Project" in the menu bar.

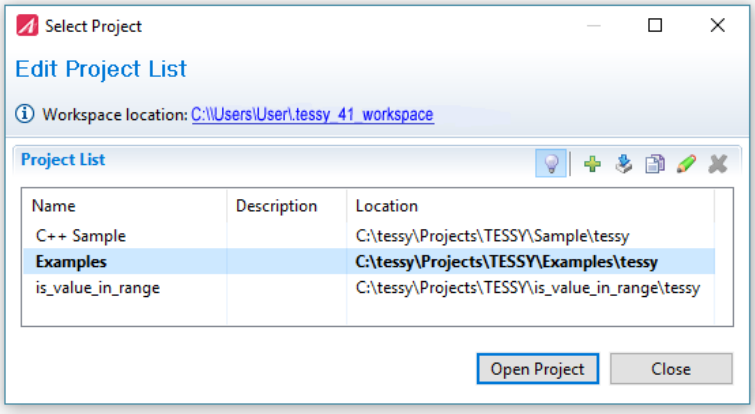


Figure 4.1: Path of the workspace

→ Click on (Create Project).
The Project Configuration dialog opens (see figure 4.2).

Creating a new project

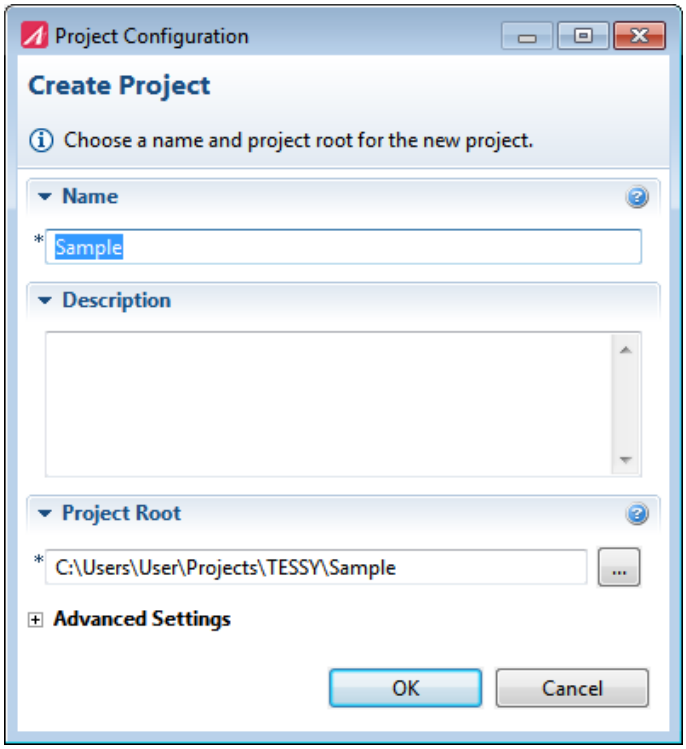


Figure 4.2: Creating a new project

→ Enter the name of your project, e.g. `Sample`.
→ Optional: Enter a description of your project.
→ Select a project root: Click on "…" and select a directory where your development project resides, i.e. where source and header files are located and where a sub directory "tessy" containing the test project shall be created (see figure 4.3).

All the project-related TESSY databases containing information on the test environment, referenced source files, compiler, debugger, etc. will be stored within a sub folder of the project root and all paths into your project will relate to this root.

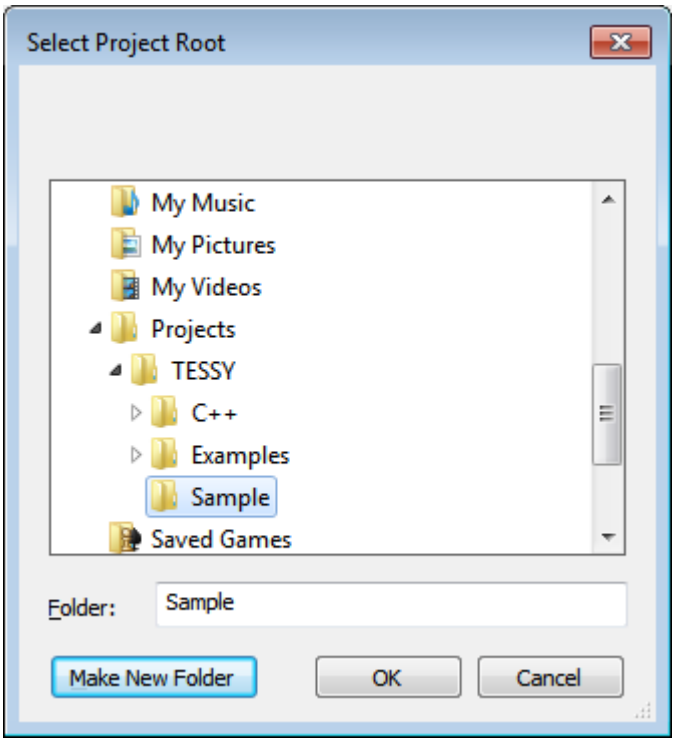


Figure 4.3: Selecting a folder for the project root.

→ Optional: Extend the “Advanced Settings” by clicking on the plus (see figure 4.4).

Advanced Settings

You have the following advanced option settings:

Source Root	Use this option to provide an alternative root location for source files if you want your source files to reside in another location than the project root. In this case all path name references to source files and include paths will automatically be collapsed using the source root location. **Important:** The source root location will be remembered locally on each computer and the given absolute path will by default **not** be stored into the TESSY project file (tessy.pdbx). This means that if this option is enabled and you want to open this project on another computer, you will always be asked for the source root location during the startup process.

continue next page

Configuration File	Enter the path to a specific configuration file or leave the field blank to use the default configuration. TESSY will create a new configuration file containing only the GNU/GCC compiler. Refer to section 6.5.6 Configuration files about how to customize this configration file.
Project Location	You can choose a different location if you would like to locate the test project files into another sub directory of the project root. **Important:** The location has to be within the project root directory!
Database Location	It is recommended to use the same directory as the project location but you can choose another sub directory of the project root if you want to separate the project database files from the other configuration files of your test project. If you tick the box “Store database in user profile” the database will be stored in a directory named using the project identifier GUID located within the “`.tessy_persist`” directory within the user profile.
Backup Location	This directory will be used to store all backup and settings files of your project which are vital for your project in order to restore it on another computer. Refer to section 6.15 Backup, restore, version control for information about files that are relevant for version control. It is recommended to use the default location but you can also choose a different location preferably within the project root. It will be used as standard for the backup and restore dialog.

Table 4.2: Options of the Project Configuration dialog

The project root and source root

By default the project root contains your development project, i.e. the source files, and one sub folder “tessy” that contains all TESSY related files.
Additionally you can specify the source root to locate source and header files outside the project root.
TESSY will use paths relative to those root paths for all files, e.g. references to source and config files. This ensures that you can move whole projects to new locations.
Please keep in mind that the source root location will always be remembered locally on each computer and the given absolute path will not be stored into the TESSY project file (tessy.pdbx). If you transfer a project with an indicated source root to another computer, you need to provide the source root (e.g. as command line argument when running in CLI mode). When opening such a project with the GUI, TESSY will remind you and ask for the source root location.

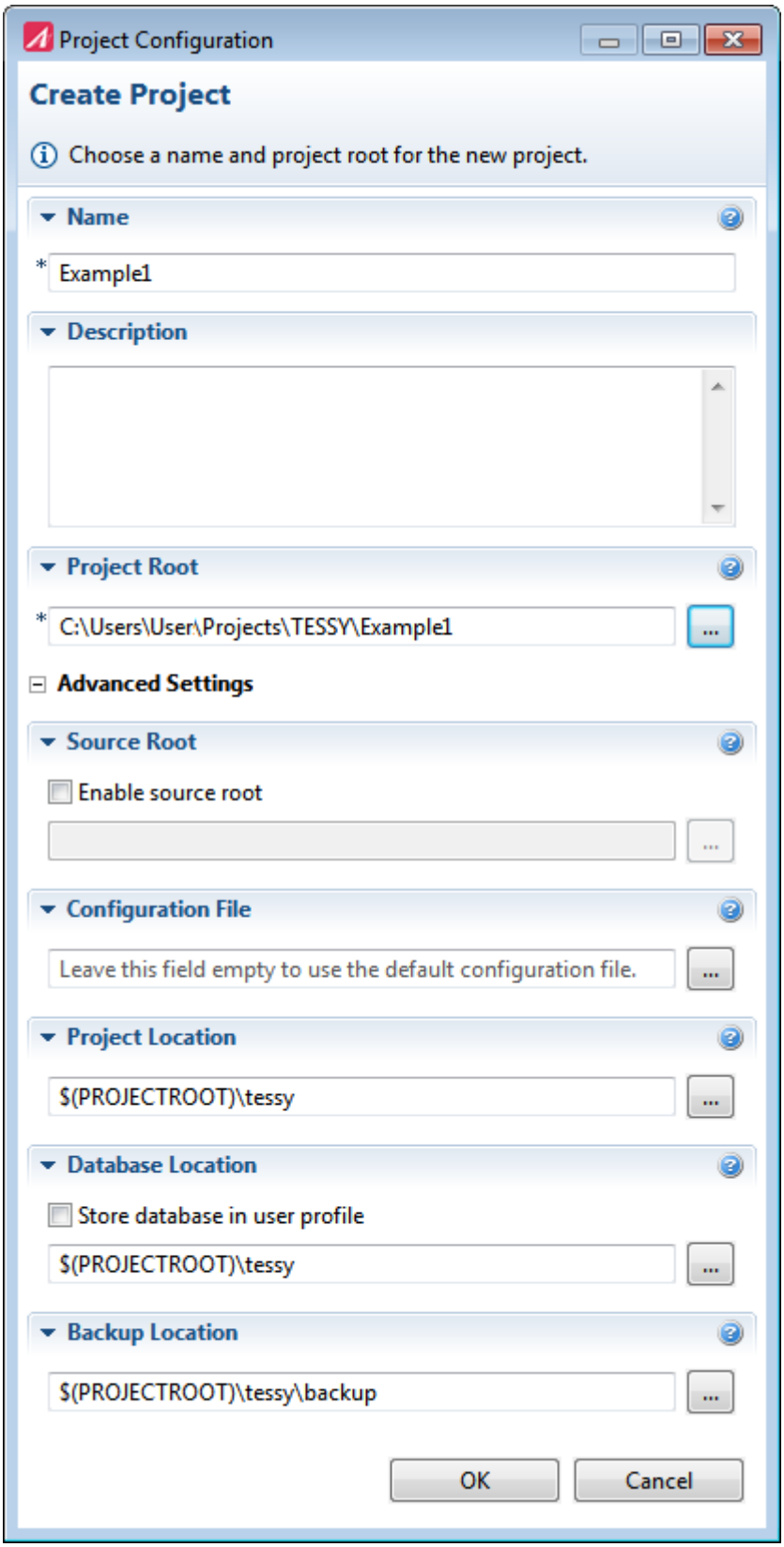


Figure 4.4: Creating a new project

→ Click “OK”.
Now TESSY creates automatically a sub folder “tessy” within the project root directory. This folder contains (within sub folders) the configuration file and the persistence databases. This will take a few seconds. Afterwards TESSY will restart (if another project was open before) and open the newly created project automatically.

4.1.2 Creating, importing, cloning, editing, deleting a project

In the Select Project dialog you can create, import, clone, edit and remove or delete your project with selecting the project and click on the icon in the tool bar:

	Creates a new project.
	Imports an existing project.
	Clones a project: TESSY creates a complete copy of a project and adds it to the project list. A new name is required.
	Opens the Project Configuration dialog. **Important:** Project root, project location and database location cannot be modified subsequently!
	Removes a project from the workspace. If you want to delete all data including project and database location, tick the box “Delete project contents on disk”. **Warning:** Deleted files cannot be restored! Before deleting make sure the project is not needed anymore or backup the project as described in section 6.15 Backup, restore, version control.

Table 4.3: Tool bar options of the Select Project dialog

With a right click you can open the context menu for further options:

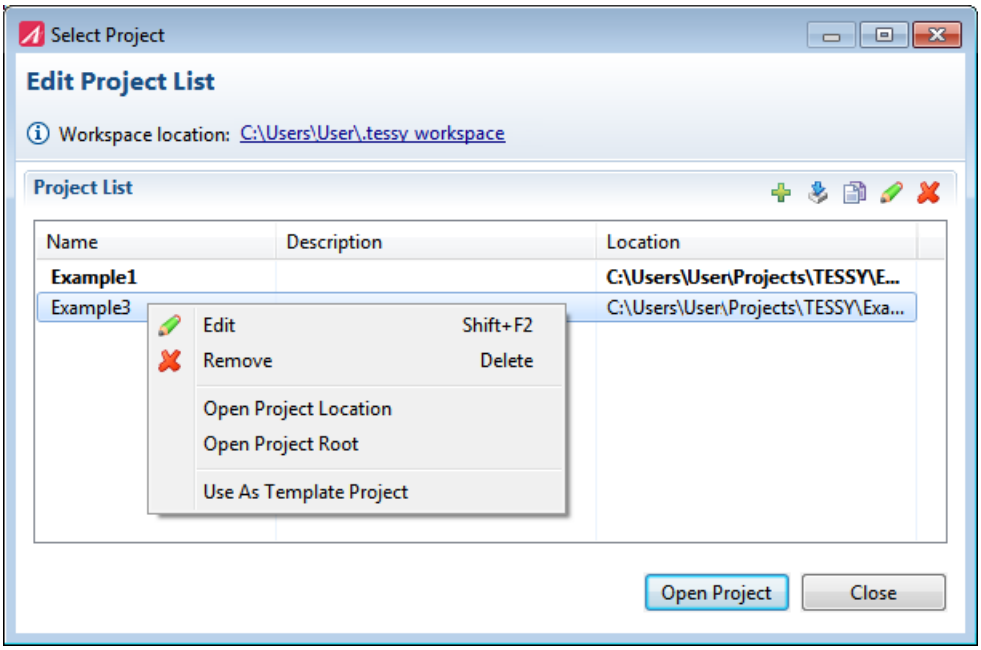


Figure 4.5: Context menu of the Select Project dialog

4.1.3 Creating a template project

With a right click on a project in the project list you can mark a project as “Template Project” (see figure 4.5).

This will have the following effects:

- The project cannot be opened anymore!
- Doubleclicking the project or marking the project and click “Open Project” will start the Project Configuration Dialog and the “Clone Project” command (see section 4.1.2 Creating, importing, cloning, editing, deleting a project).
- Doubleclicking the PDBX file of the template project will start the Project Configuration Dialog and the “Clone Project” command.

At anytime you can remove the mark as template project. The project will then be a normal project and you can open it as usual.

4.1.4 Moving the project directory

You can move your whole project directory and then import the project again:

→ Either double-click on the tessy.pdbx file or use the Import Project button. TESSY will ask you, if the project was moved or copied (see figure 4.6).

→ Answer this question correctly and click “OK”.
If the project was copied, e.g. you want to create a new project as a copy of the original one, a new project identifier needs to be assigned to distinguish the new project from the original one. TESSY will do this automatically.

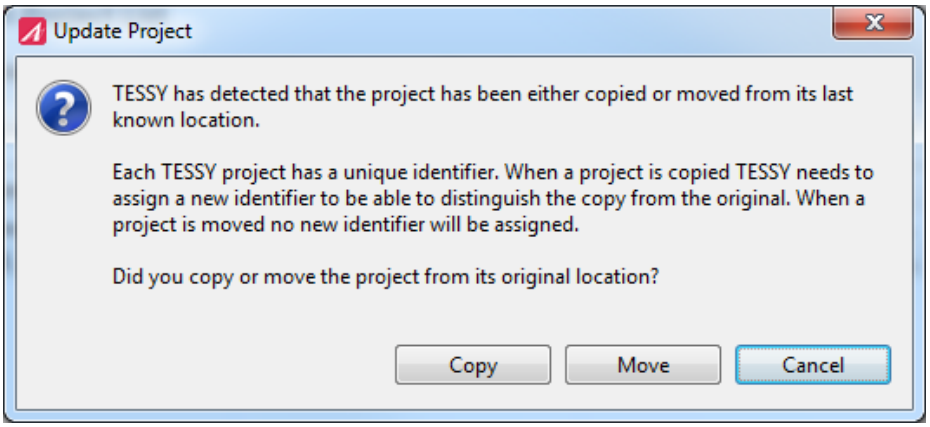


Figure 4.6: Project identifier handling

4.1.5 Handling with equally named projects

In the “Select Project” dialog all projects are listed with name and local path.

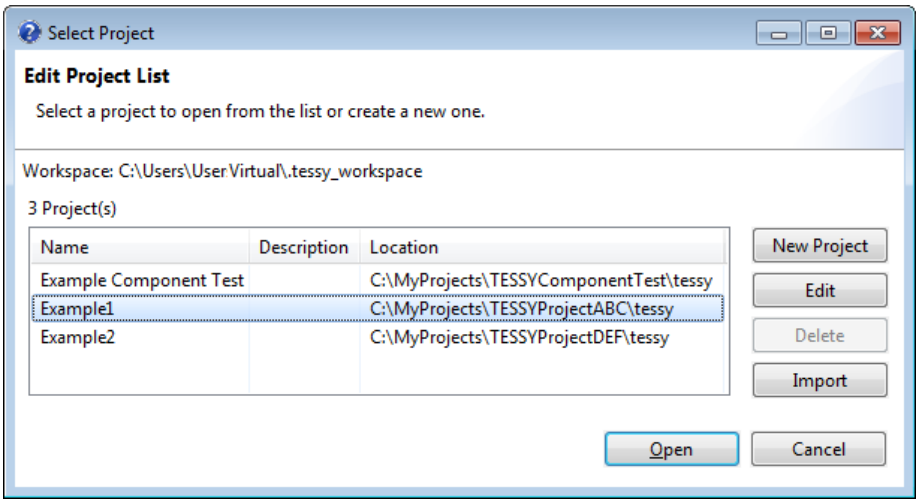


Figure 4.7: Project Example1 is created.

It is possible to handle projects with equal names. The table below explains in which way TESSY will replace projects within the projects list if they have identical names:

Status	**Operation**	**Behavior and tip**
Project named 'Alpha' exists in location 'xy'.	You create a new project 'Alpha' in **another location**.	You will get a warning: "Project with identical name will be removed from the project list." The new project appears in the project list, the old project will be removed from the list (but not deleted!). If you want to open the old project again, → click on "Import" and open the old project. In that case the newer project will again be removed from the list.
Project 'Alpha' exists in location 'xy'.	You try to create a new project 'Alpha' in the **same location**.	You will get an error, it is not possible to create the project, because two projects cannot share the same location. → Change the location of the new project.
Project 'Alpha' exists in location 'xy'.	You open an existing project 'Alpha' (either with click on "Import" or within the command line interface) in **another location** 'Beta'.	You will get a note "Information: Project 'Alpha' replaced within the project list." The second project appears in the project list, the first project will be removed from the list (but not deleted!). If you want to open the other project again, → click on "Import" and open the other project.

Table 4.4: Handling of projects with equal names

4.1.6 Using a specific environment setting

Important: This section is only recommended for advanced users that have already worked with TEE. For basic handling we recommend to continue with section 4.2 Understanding the graphical user interface and following sections and then return to this section.

TESSY will create a specific configuration file for each project database. This way you can share the environment settings with other members of your development team. The configuration file is stored within your project root together with other project related files. Such a configuration file contains only the compiler/target environments you want to use. All other environment configurations are not visible for the user as long as this file is assigned to a given project database.

To **customize the configuration file** within the Test Environment Editor (TEE)

→ refer to section 6.5.6 Configuration files.

4.1.7 Updating the database

After the update you cannot open the project in previous versions of TESSY!

TESSY will recognize, if an upate of the database is necessary (see figure 4.8).

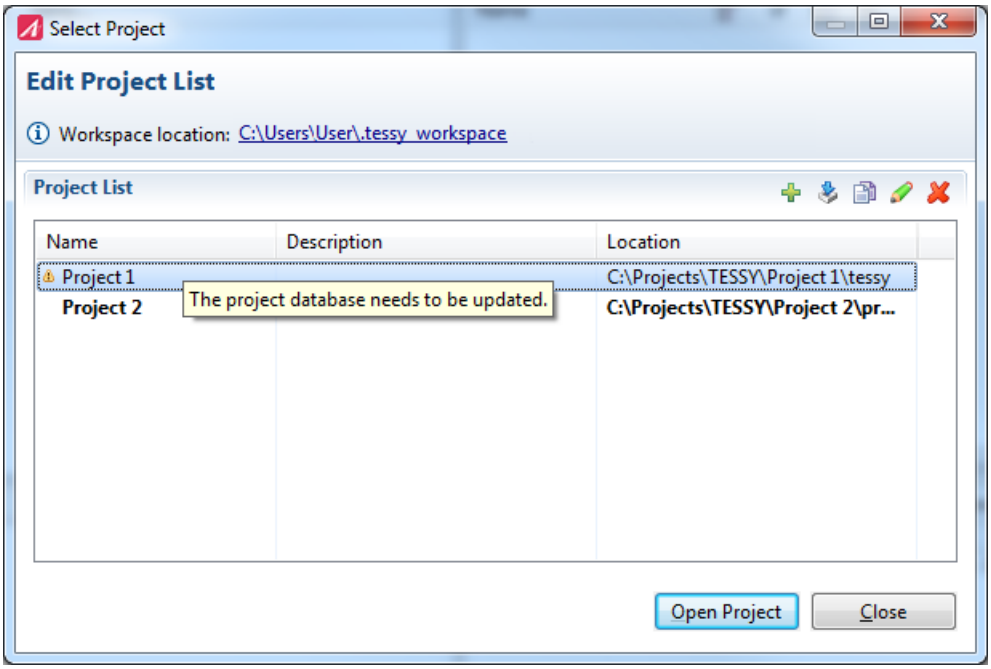


Figure 4.8: TESSY notifies, that a database update is necessary

When you want to open the project, you will be asked to update the database (see figure 4.9):

→ Click “OK”.

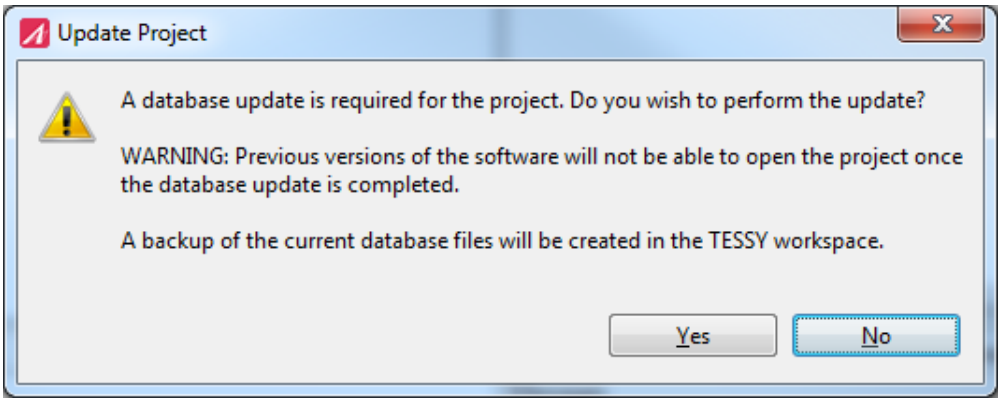


Figure 4.9: A database update is necessary

4.2 Understanding the graphical user interface

When TESSY starts the first time, the graphical user interface (GUI) will open within the Overview perspective.

Please check the terminology shown in the figure “TESSY interface” and the explanations beneath. This terminology will be used throughout this manual.

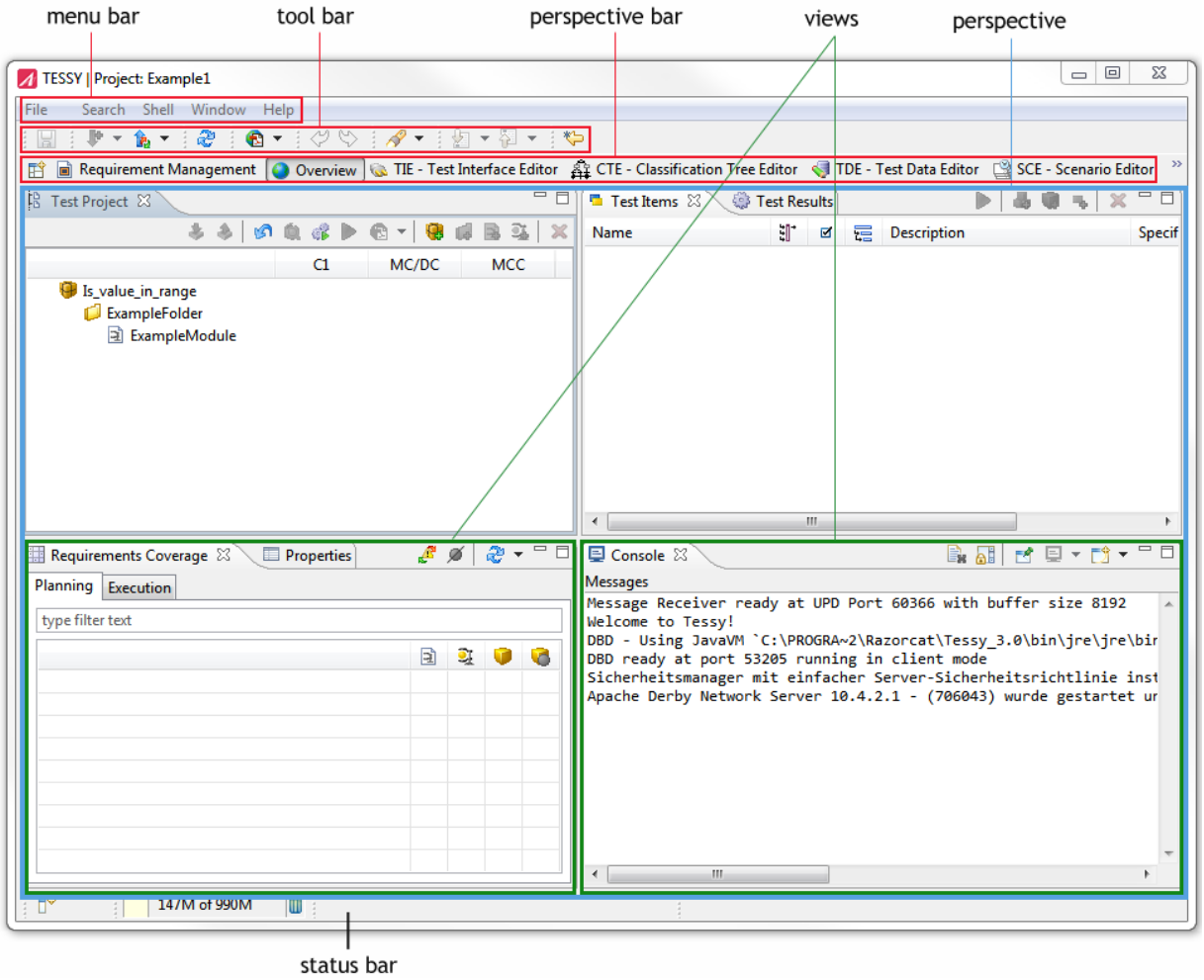
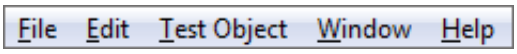


Figure 4.10: TESSY interface and its terminology

4.2.1 Menu bar



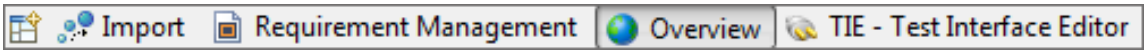
The menu bar provides actions as “File”, “Window” etc. Within the “Help” you find the online help of TESSY. Many of these actions may also be available within the context menu of individual views, if the actions are applicable for the items within the view.

4.2.2 Tool bar

At the global tool bar of TESSY interface you can select a project, save changes etc. By rolling over an icon with the cursor a tooltip will tell you the purpose of each icon. There may also be individual tool bars within the views. Here you find the tools for operating, e.g. to start the test execution .

Save all changes in any views or editors by clicking the save icon .

4.2.3 Perspectives and perspective (tool) bar



TESSY contains several perspectives to present information based on different tasks in the test workflow (“Requirement Management”, “Overview”, “TIE” etc.). Each perspective offers several views.

In the perspective bar (containing the perspective names) you can switch between the different perspectives. The perspectives - from the left to the right - follow the actions taken while preparing, running and evaluating a test.

Every perspective name has several right-click menu options (the context menu).

By clicking on the left symbol you can open other perspectives (see figure 4.11):

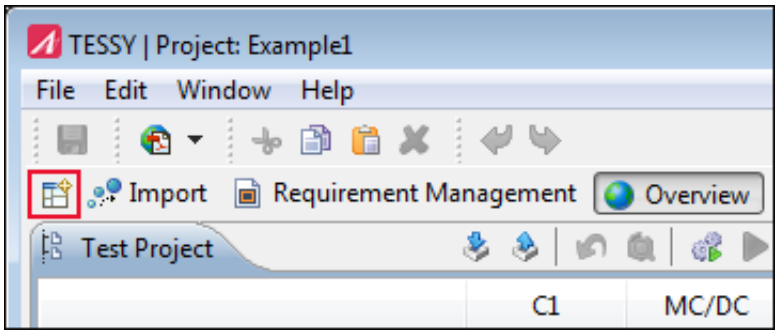


Figure 4.11: Open other perspectives

4.2.4 Views

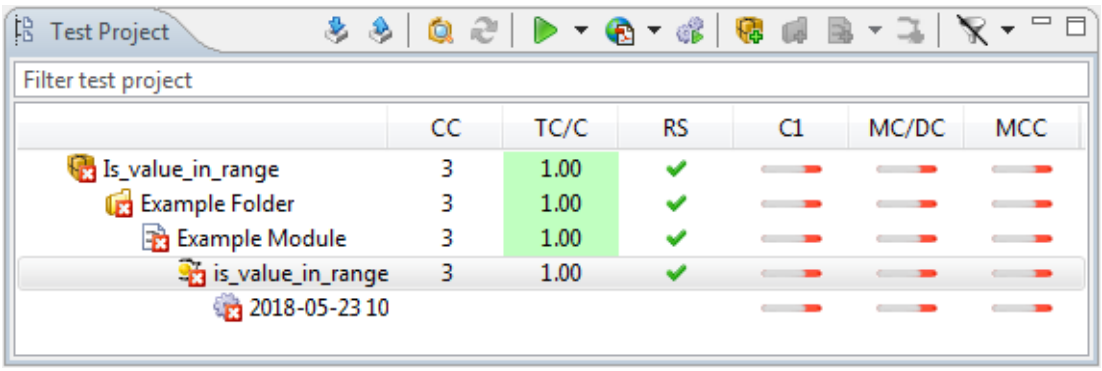


Figure 4.12: Test Project view within the Overview perspective

Every view in each perspective contains possibilities of configurations or provides presentations of data. Some views are common to several perspectives (such as the properties of sources), and some are specific to one perspective (such as the plots in the TIE).

Notice that the views appear differently combined with other views, e.g. the view Test Results within the Overview perspective is combined with the Test Items view, but within the TDE perspective it is combined with the Test Project view. The reason for the different combinations is to give you a fast overview and comparison between various information within each project step.

You can change the appearance of views for you own needs and open views of one perspective into another perspective:

Adding views to a perspective

→ Activate (open) the perspective where you want to add a view.
→ Click “Window” > “Show View”.
A window displaying all views will open (see figure 4.13).
→ Select the view you want to open.
→ Click “OK”.
The selected view is added to the active perspective.

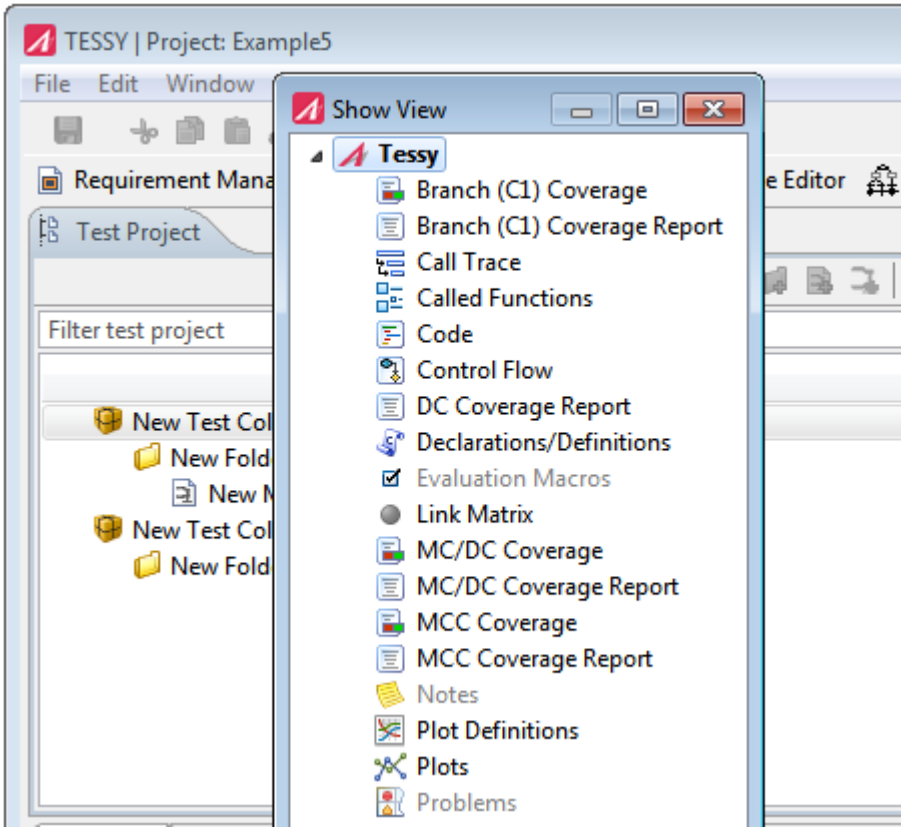


Figure 4.13: Adding views to a perspective

Change the position of views with drag and drop:

Changing view position

→ Click on a name of a view and move it where you like: You can move views to another location within the same group of views or into another group of views or even outside the current window.
→ Right-click on the perspective switch and choose “Reset” to switch back to the original positions of all views of the respective perspective (see figure 4.14).

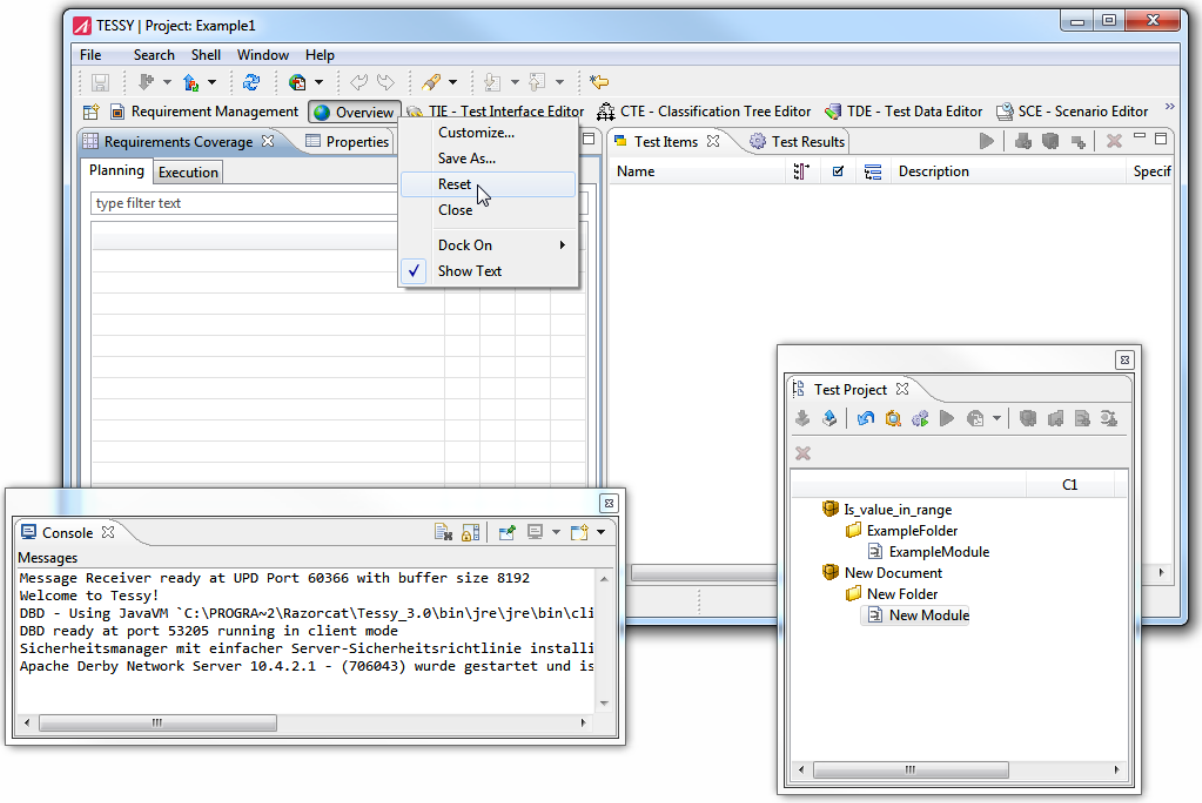


Figure 4.14: Move the views separately. To switch back, use “Reset”.

Resetting workbench

To switch back all positions of perspectives and views:

→ Choose “Window” > “Reset Workbench” from the menu bar (see figure 4.15).

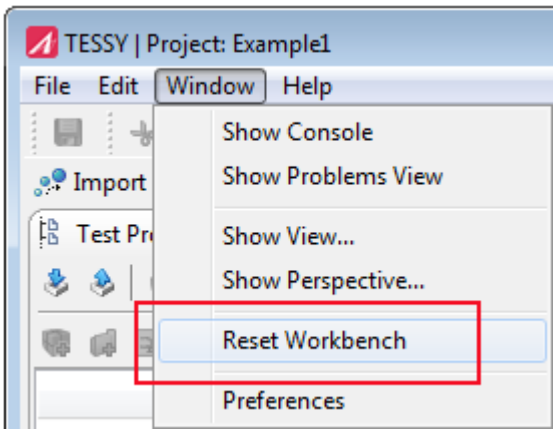


Figure 4.15: To switch back all positions of views and perspectives use “Reset Workbench”.

Maximize and minimize views

You can maximize and minimize views for better visibility.

To maximize a view,

→ use the button within the right corner (see figure 4.16) or double-click on the tab of the view.

Figure 4.16: Minimizing and maximizing views

The view will be maximized and other views of the perspective will be minimized, displayed by the view symbol on the left and the right (see figure 4.17).

To restore all views,

→ click on the “Restore”-button on the upper left side.

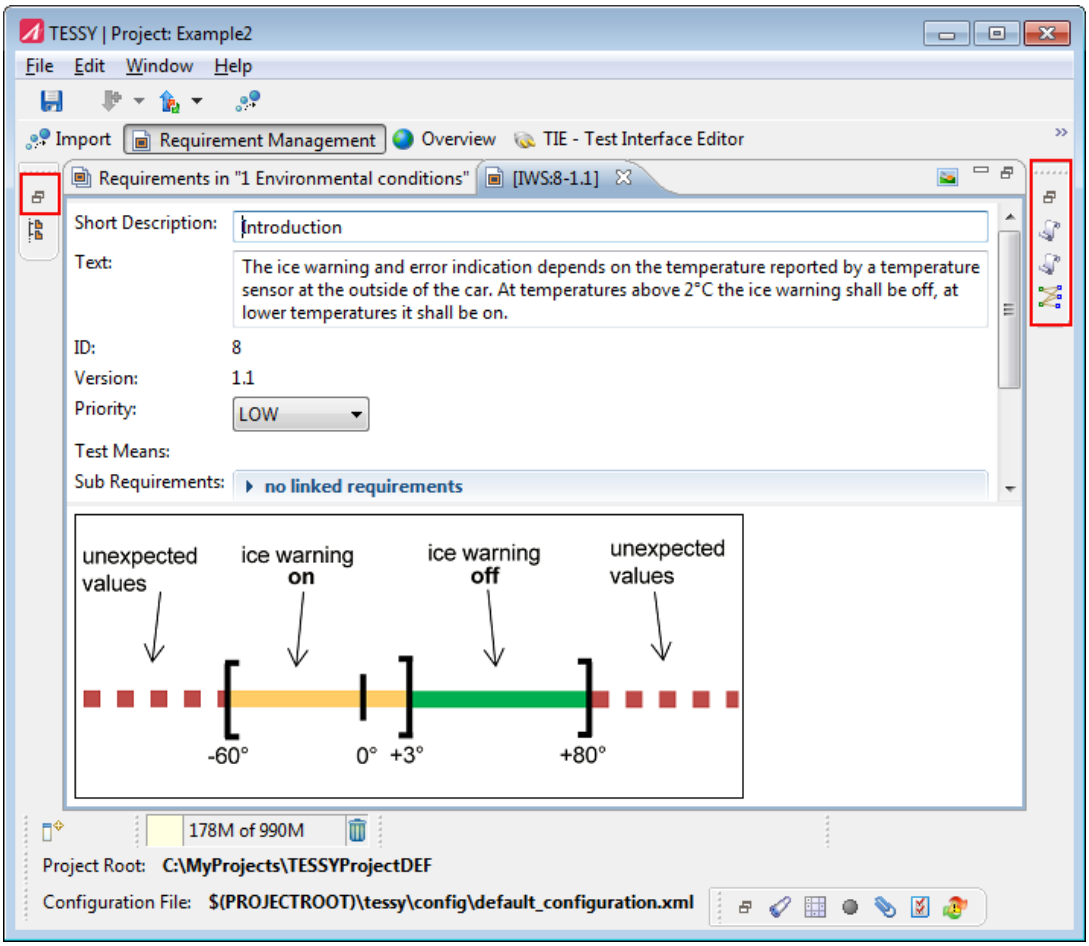
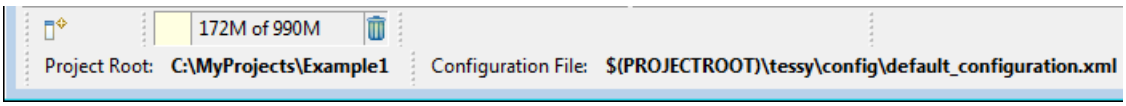


Figure 4.17: Maximized view with minimized views on the right and the restore-button on the left

There are navigation views that present hierarchical structured data. Selections on such tree items may cause other views or the editor pane to change the information being displayed.

All views are context sensitive: If you select an item within one view, another view might display other information. If something is not displayed correctly, make sure you selected the desired item.

4.2.5 Status bar



The status bar provides status information about the application and current status, e.g. the directory of the project root and the configuration file.

4.3 Using the context menu and shortcuts

4.3.1 Context menu

Most contents, tabs etc. have options that are displayed in the context menu, which is retrieved with a right click. It shows main operations as “Copy”, “Paste”, “Delete” etc.

The context menu is context sensitive and changes as different items are selected (see figure 4.18).

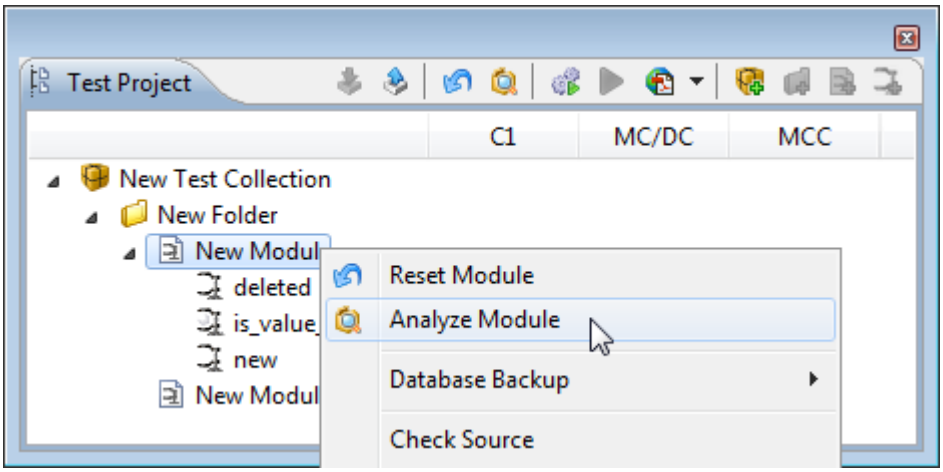


Figure 4.18: Using the context menu with a right click.

4.3.2 Shortcuts

You can operate within TESSY with several shortcuts by using the keyboard:

Important: For using shortcuts make sure to be in editing mode of the current view! Otherwise the shortcut will not work.

Warning: You cannot reverse the deleting of data. Before deleting make sure this database is really not needed anymore.

Function	Shortcut / Keys	Comment
Copy	Ctrl+C	
Cursor positioning right	Tab *alternatively* Ctrl+right arrow key	Moves the cursor to the next input section on the right side of the line. Only within TDE.
Cursor positioning left	Shift+Tab *alternatively* Ctrl+left arrow key	Moves the cursor to the input section on the left side of the line. Only within TDE.
Cut	Ctrl+X	Only possible for folders, modules and (synthetic) test objects, not for test items.
Delete	Del	Only possible if the item to delete does not contain any data or folders or modules! Only manually created test cases can be deleted! You cannot delete test cases created by CTE. This prevents possible inconsistencies within the CTE document.
Generate test report	Ctrl+R	
New folder	Shift+Ins	
New module	Ins	Only possible, if a test collection or folder is selected.
Paste	Ctrl+V	
Rename	F2	
Start test execution	Ctrl+E	Executes a test object.

Table 4.5: Shortcuts and key functions

Drag & Drop

You can also operate with “drag & drop”, which is the same as “cut & paste”.

Shortcuts for certain views are described in the related section within chapter 6 Reference book: Working with TESSY.

5 Tutorial: Practical exercises

This chapter will show you on the basis of prepared exercises how to work with TESSY:

- The Quickstart 1: Unit test exercise is_value_in_range is a very basic example of how to operate with TESSY.
- The Quickstart 2: The Classification Tree Editor (CTE) gives a short and easy introduction of operating with the Classification Tree Editor (CTE). It continues the Quickstart 1.
- The Quickstart 3: Component test exercise interior_light shows on the basis of a simple example how component testing works with TESSY.
- The Quickstart 4: Exercise C++ gives a short and easy introduction of handling with a C++ source file.
- The Quickstart 5: Test driven development (TDD) gives a short introduction into test driven development with TESSY. You should be familiar with the overall handling of TESSY before doing this exercise.

5.1. Quickstart 1: Unit test exercise is_value_in_range **78**

5.1.1. Creating a test project 79
5.1.2. Specifying the target environment 81
5.1.3. Adding the test object and analyzing the C-source file 83
5.1.4. Editing the test object interface 85
5.1.5. Designing test cases 86
5.1.6. Adding test cases and test steps 86
5.1.7. Entering test data 88
5.1.8. Executing the test 91
5.1.9. Repeating the test run with coverage instrumentation 93
5.1.10. Analyzing the coverage 94
5.1.11. Creating a Test Details Report 98
5.1.12. Repeating the test run with requirements 102
5.1.13. Reusing a test object with a changed interface 114

5.2. Quickstart 2: The Classification Tree Editor (CTE) 123
5.2.1. The CTE tree elements . . . 124
5.2.2. Working with the CTE . . . 126
5.2.3. Entering test data . . . 126
5.2.4. Creating test cases . . . 128
5.3. Quickstart 3: Component test exercise interior_light 136
5.3.1. Creating the test project . . . 138
5.3.2. The heartbeat function . . . 140
5.3.3. Preparing the test interface . . . 143
5.3.4. Adding test cases . . . 145
5.3.5. Editing data . . . 146
5.3.6. Configuring the work tasks . . . 148
5.3.7. Designing scenarios . . . 148
5.3.8. Executing the scenarios . . . 154
5.3.9. Evaluating the scenarios . . . 155
5.4. Quickstart 4: Exercise C++ 156
5.5. Quickstart 5: Test driven development (TDD) 160

5.1 Quickstart 1: Unit test exercise is_value_in_range

In this exercise we will get to know the basic functionality of testing with TESSY. We will operate with the example “is_value_in_range” which will give you a fast introduction and an overview as well as the terms of importance.

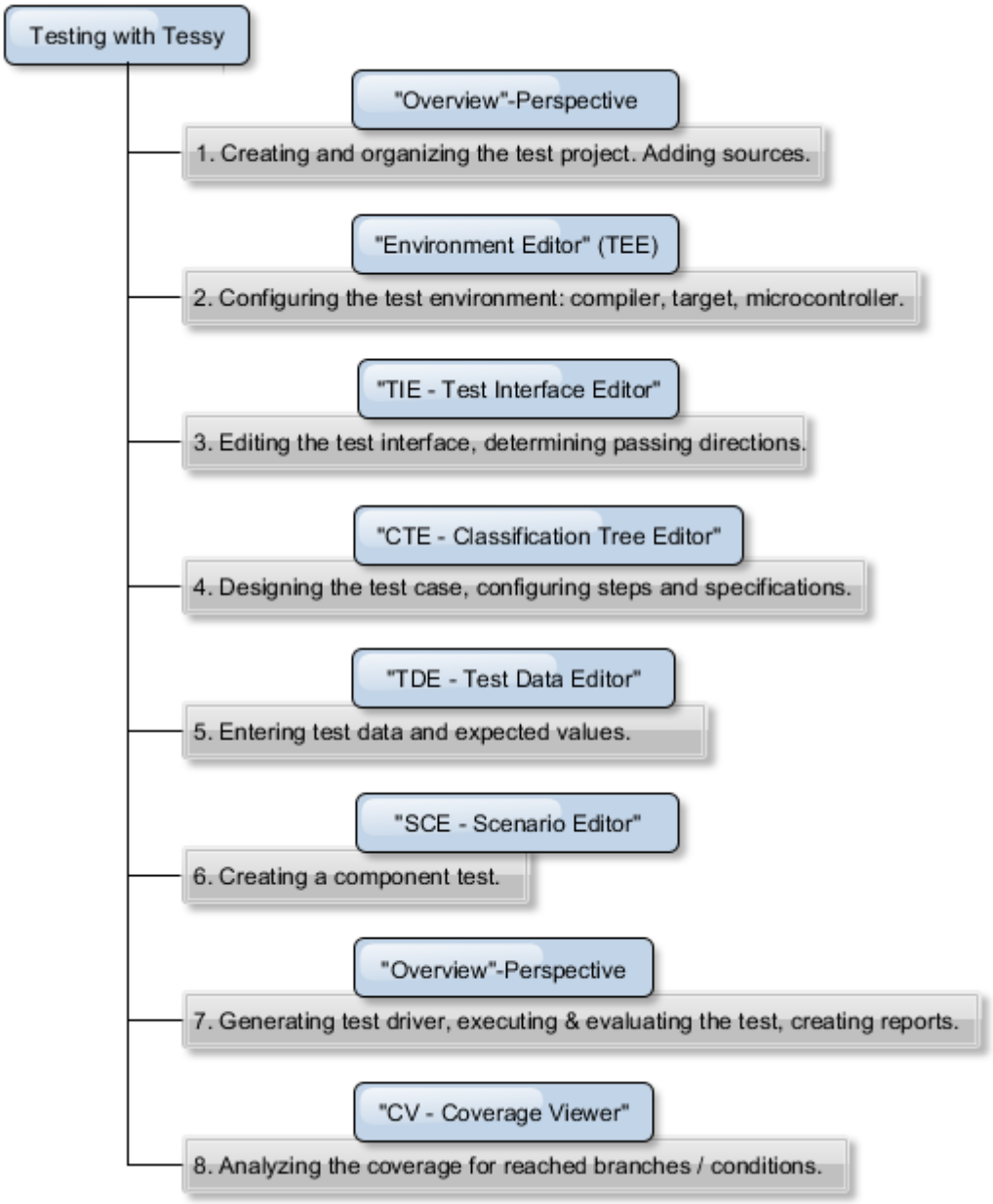


Figure 5.1: Operational sequences in TESSY

A unit test in TESSY is divided into the following central test activities:

Central test activities

- Determining test cases.
- Entering test data and expected values.
- Executing the test.
- Evaluating and documenting the test.

Usually you would import your requirements first. To keep this exercise understandable for beginners, we will first exercise a simple project, then import some basic requirements and restart the test!

We will now follow a simple source code example to show how to exercise those activities with TESSY.

Example “is_value_in_range”

A start value and a length define a range of values.

Function: Determine if a given value is within the defined range or not. Only integer numbers are to be considered.

5.1.1 Creating a test project

If you have not created the project “Example1” yet, do as follows:

To understand TESSY´s file system and databases, consult section 4.1 Creating databases and working with the file system.

→ Start TESSY by selecting “All Programs” > “TESSY 4.x” > “TESSY 4.x”.
→ If the “Open Project” dialog will open, click on (Create Project).
If another project is opened within TESSY, click “File” > “Select Project” > “New Project” and then click on .
The Project Configuration dialog opens (see figure 5.2).

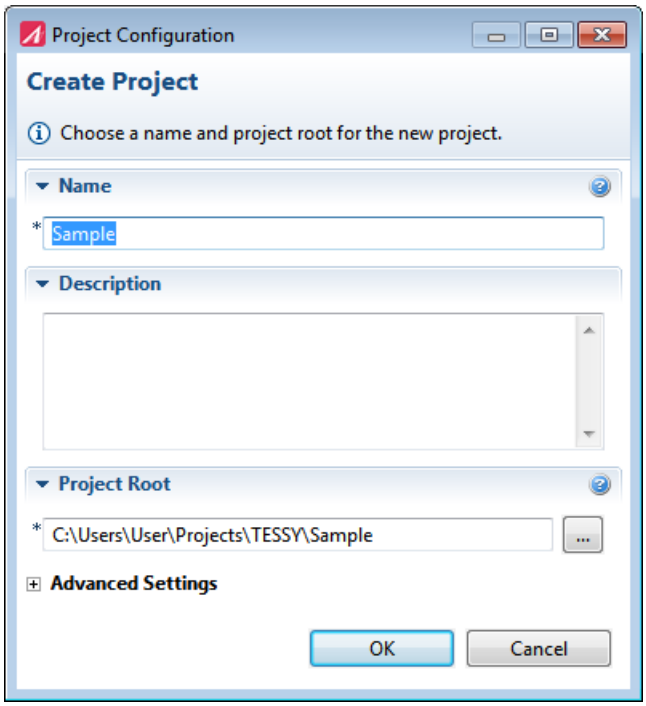


Figure 5.2: Creating the new project “Example1”.

→ Enter *Example1* as name of the project.

→ Leave the automatically created Project Root as it is (default `C:\[User]\Projects\TESSY\[Projectname]`) and click “OK”.
TESSY now creates the project “Example1” (see figure 5.3). This will take a few seconds.

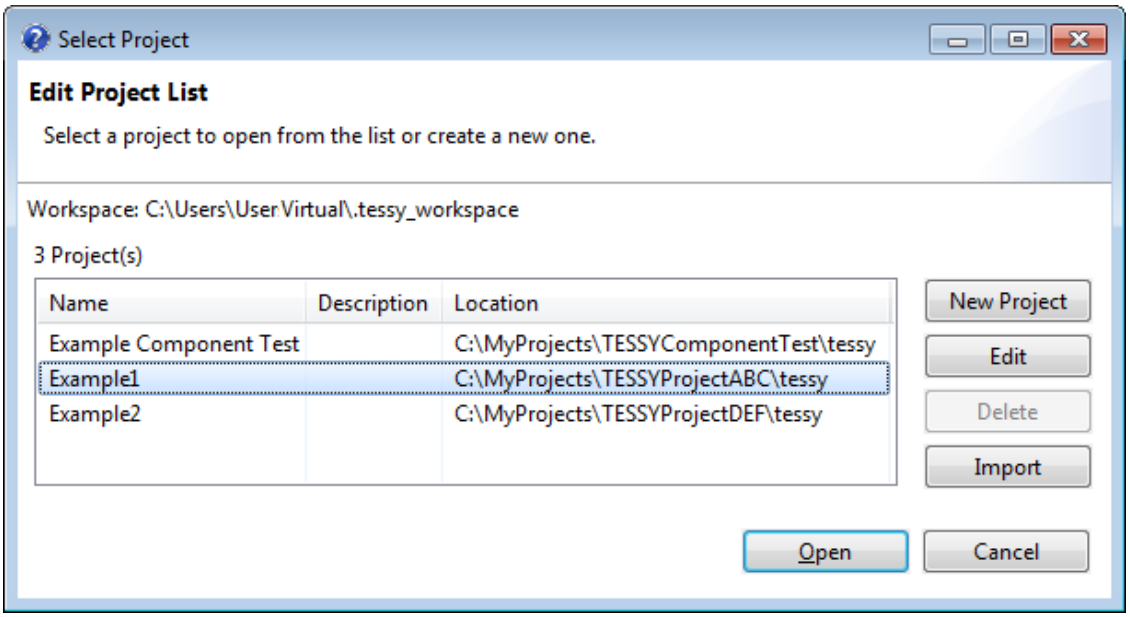


Figure 5.3: New project “Example1” is created

→ Click “Open Project”.
TESSY now opens your project. This will take a few seconds.

Organizing “Example1”

The project “Example1” is opened within the Overview perspective. You can create different folders within a test collection, each containing modules with various test objects. To keep it simple, we will create now one test collection with one folder.

We start within the view “Test Project”:

→ In the Test Project view click on the icon (New Test Collection) in the tool bar of the view.
→ Enter *Is_value_in_range* and press the “Enter”-key.
→ Click on (New Folder), enter *ExampleFolder*, click “Enter”.
→ Click on (New Module), enter *ExampleModule*, click “Enter”.
The module relates to one or many source files which are to be tested.

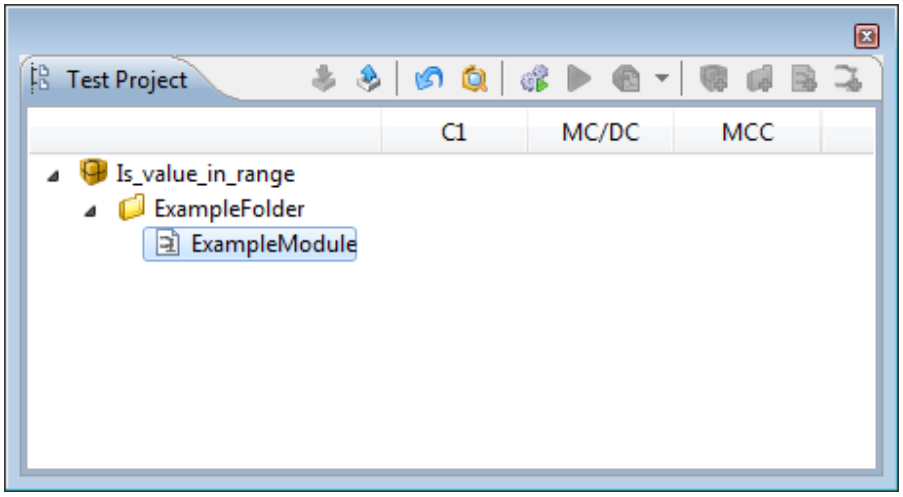


Figure 5.4: Test collection “Is_value_in_range” with an example folder and module

Rename or delete a module or a folder by using the context menu (right-click > “rename” or “delete”) or the key F2.

5.1.2 Specifying the target environment

Usually at this point you will have to specify the target environment, that is to determine the compiler, the target and the microcontroller. You will do that in the “Test Environment Editor” which we will get to know later.

Please notice beneath in the Properties view at tab “General” that the GNU GCC compiler is already selected for this module (see figure 5.5), which is enough for our example.

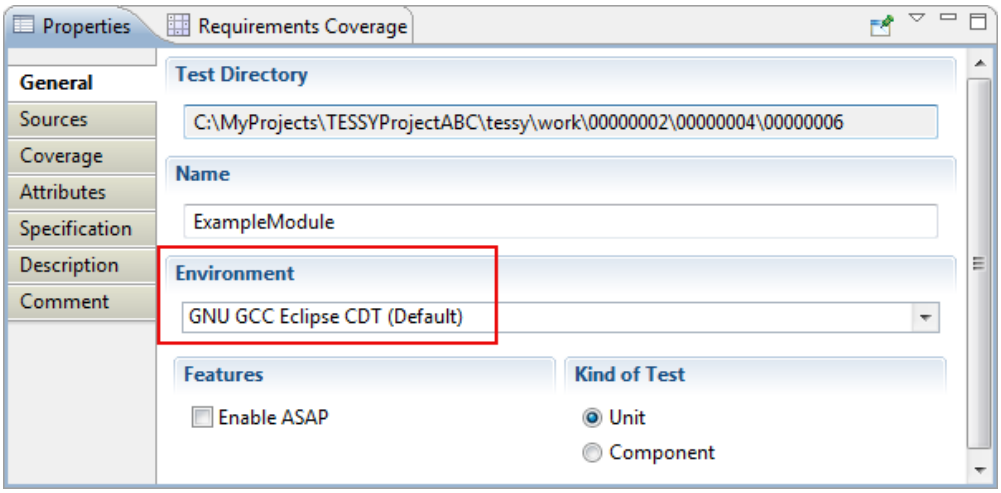


Figure 5.5: GNU GCC compiler is selected by default.

C-source file

Now we will add the source file to the module. The source file contains the C-function to be tested:

```
01 // This is a very basic program to demonstrate the integration
02 // of Tessy and HiTOP, the debugger from Hitex.
03 // (c) Hitex Systementwicklung GmbH 2001, www.hitex.de
04 //
05 // $Revision: 2$
06
07 struct range {int range_start; int range_len;};
08
09 typedef int value;
10
11 typedef enum {no, yes} result;
12
13 // Checks if r1.start <= v1 < (r1.start + r1.len),
14 // e.g. r1.start = 5, r1.len = 2
15 // ==> v1 == 4 ---> no
16 //     v1 == 5 ---> yes
17 //     v1 == 6 ---> yes
18 //     v1 == 7 ---> no
19 // However, the implementation is intentionally
20 // erroneous: v1 == 7 results "yes" instead of "no"!
21 //
22
23 result is_value_in_range (struct range r1, value v1)
24 {
25     if (v1 < r1.range_start)
26         return no;
27
28     if (v1 > (r1.range_start + r1.range_len))
29         return no;
30
31     return yes;
32
33 }
```

Figure 5.6: The source code of the C-Function to be tested

5.1.3 Adding the test object and analyzing the C-source file

We will use the example C-source file "`is_val_in_range.c`" which is stored under "`C:\Program Files\Razorcat\TESSY_4.x\Examples\IsValueInRange`".

Copy the C-source file, paste it in the project root an add it to the module:

- → Open the Windows Explorer.
- → Copy the source file "`is_val_in_range.c`" in a folder which is located in the project root, e.g. "`c:\MyProjects\TESSYProjectABC\sources`".

It is useful to relate all sources, includes etc. to the project root. You have a better overview about all sources, includes etc.

- → Switch back to TESSY.
- → In the Test Project view select the module ("Example Module").
- → In the Properties view switch to tab "Sources".
- → Click on (Add Source).

Adding the C-source

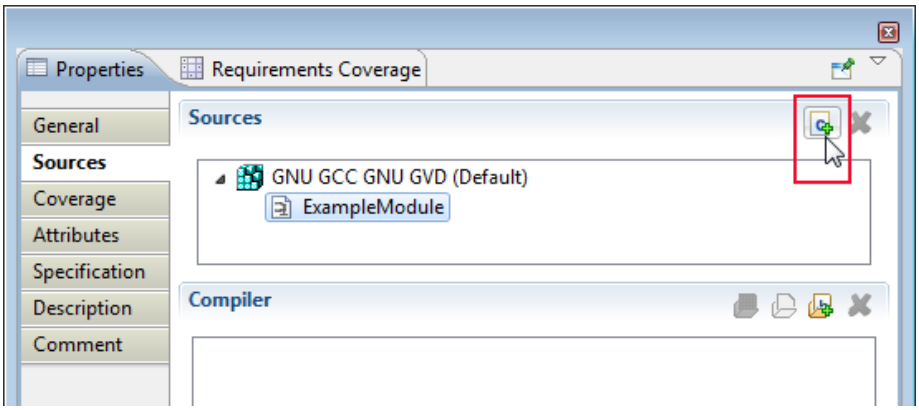


Figure 5.7: Adding the C-source file.

- → Select the source file "`is_val_in_range.c`" from the folder where you just pasted the source.
- → Click "Open". The C-source file will be added.
- → In the Test Project view above click on (Analyze Module) to start the module analysis (see figure 5.8).

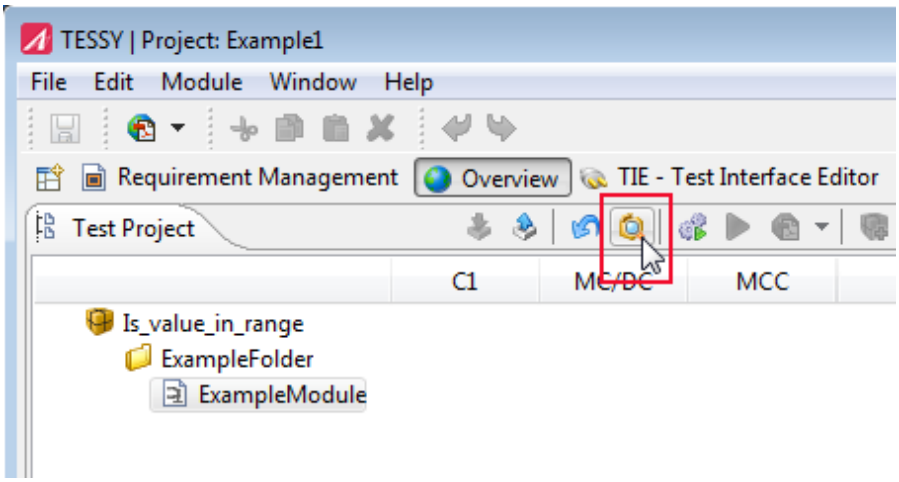


Figure 5.8: Analyzing the module, that is the C-source file.

TESSY now analyzes the C-source file, this will take a few seconds. After successful processing,

→ click on the white arrow in front the module: .

TESSY will as well analyze the C-source file by just clicking on the white arrow after adding the C-source file.

Now all functions which were defined in the C-source file are displayed as children of the module above within the Test Project view (see figure 5.9).

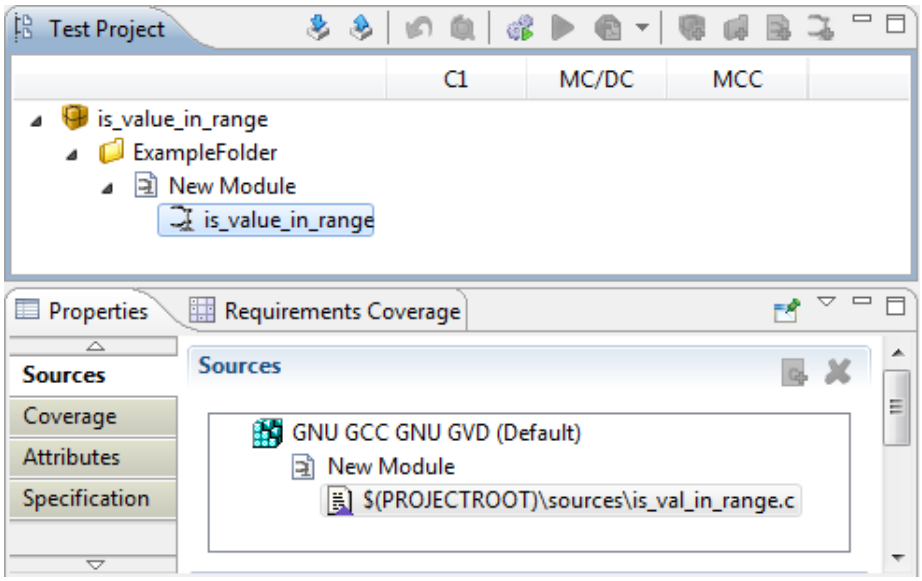


Figure 5.9: The function of the C-source is displayed as child of the module.

Our sample C-source file contains only one function, our **test object** “is_value_in_range”.

The term “test object” indicates the functions within the module we are attempting to test.

5.1.4 Editing the test object interface

→ Switch to the TIE (Test Interface Editor).

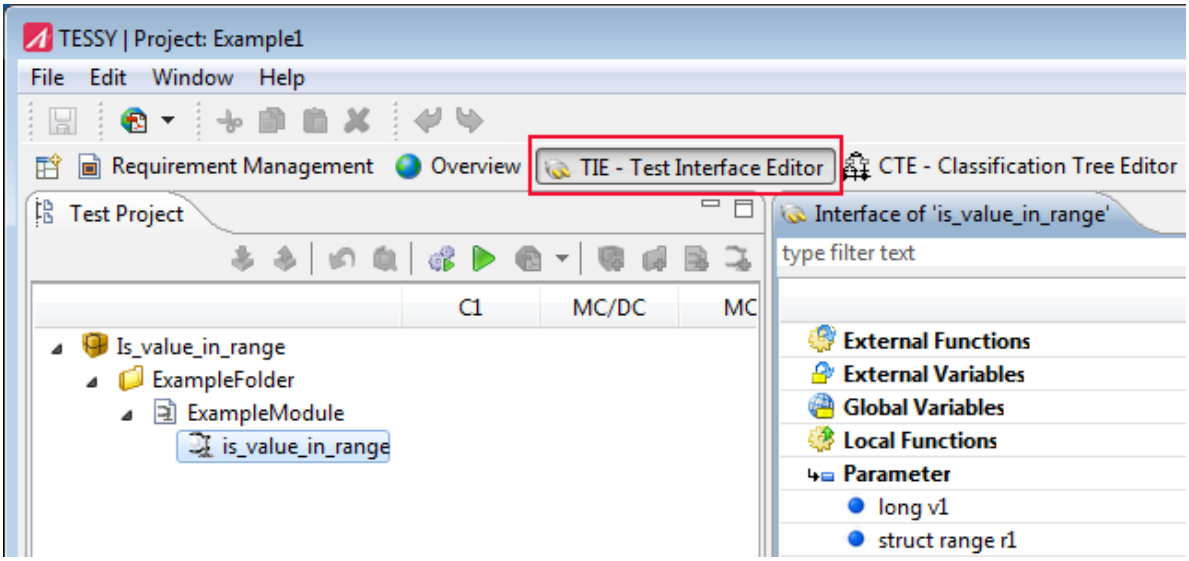


Figure 5.10: Perspective TIE - Test Interface Editor

Determine passing directions

Now we can edit the interface information for every test object and determine which values are input and which ones are output variables. **Input values** are all interface elements that are read by the test object. **Output values** are written by the test object.

Upon opening the module, TESSY will try to set the default passing directions (input or output) automatically. You can change these default interface settings to your needs.

In our sample the passing directions are already defined, you do not have to take actions.

→ In the Interface view open the Parameter paragraph to see the inputs and output values that are already defined in our example (see figure 5.11).

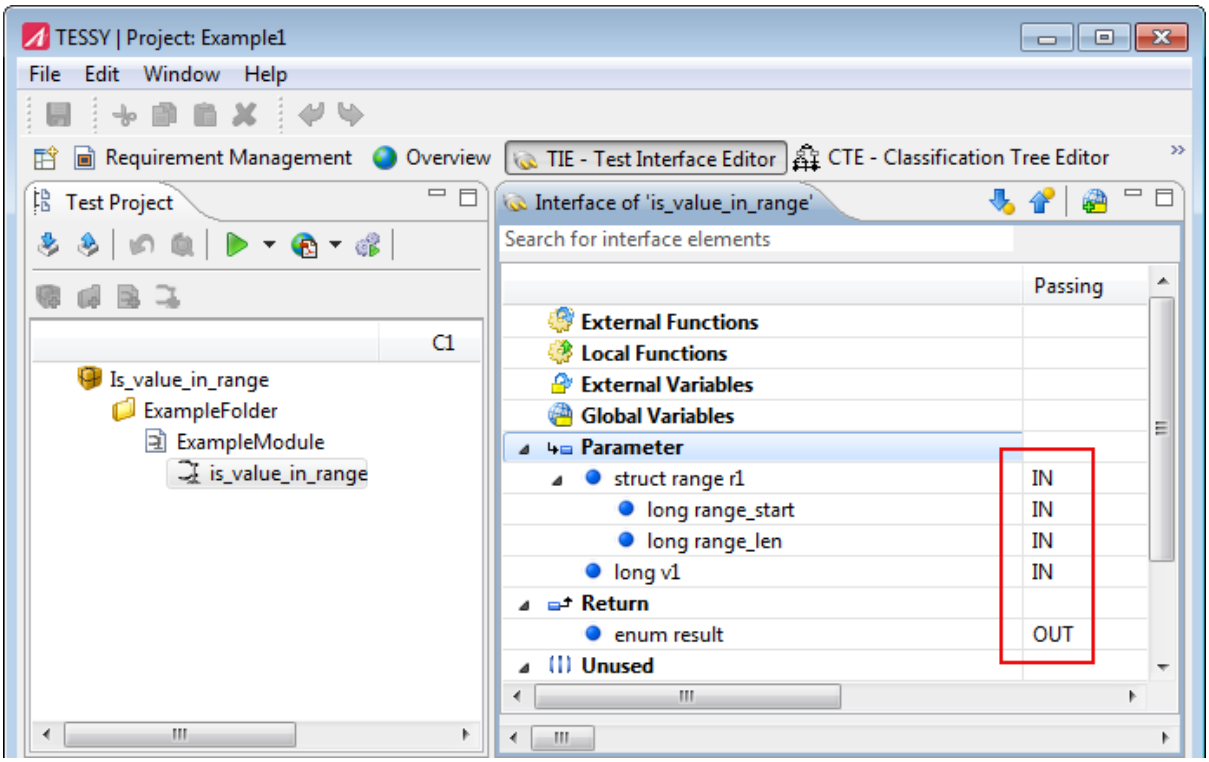


Figure 5.11: The inputs and outputs are already defined

5.1.5 Designing test cases

Usually now you would design the test cases, either manually or within the Classification Tree Editor (CTE), based on specifications of your test object.

Since the CTE is a subject for its own, we will not make use of the CTE in this example, but simply enter some ad-hoc test data manually.

To learn about the CTE refer to section 6.8 CTE: Designing the test cases or follow the Quickstart 2: The Classification Tree Editor (CTE).

5.1.6 Adding test cases and test steps

Now we will add three test cases each with one test step within the Test Items view:

→ Switch to the Overview perspective and the Test Items view.

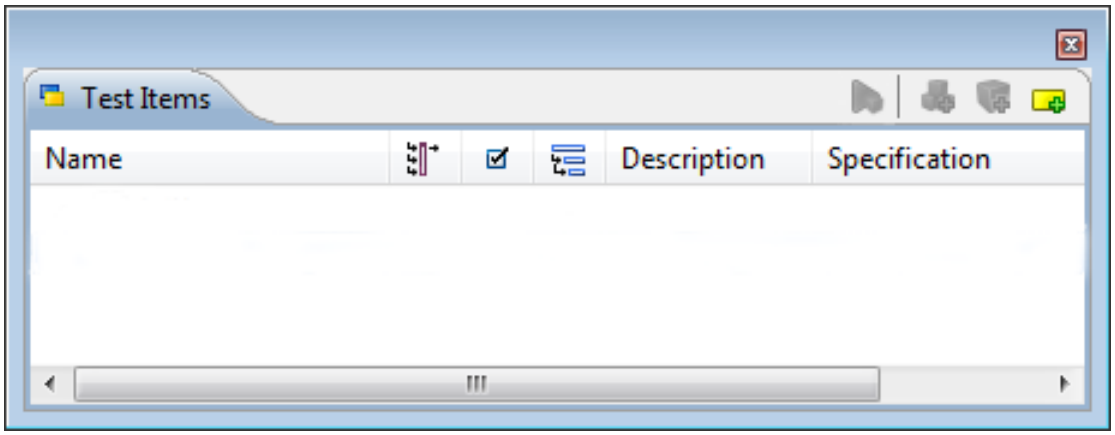


Figure 5.12: Test Items view

→ Select the test object in the Test Project view.
→ In the Test Items view click on (New Test Case).
The first test case is created and a test step is automatically added.

In TESSY every test case has at least one test step.

→ Add two further test cases.
→ Expand the test cases by clicking on the arrows in front of the test cases.

Adding test cases

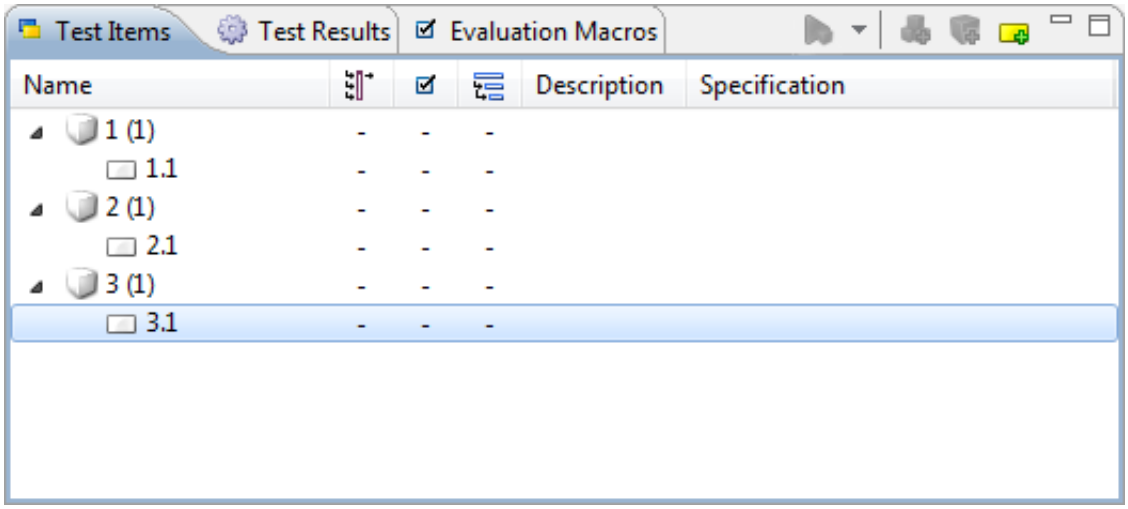


Figure 5.13: Three test cases were added in the Test Items view

Please notice the following habits of this view:

- The first number is the number of the test case, the number in brackets shows the quantity of the test steps included.
- Test case numbers will be counted continuously: If you delete test cases, new test cases will get a new number and existing test cases will not be renumbered.
- If you cannot click on “New Test Case” or “New Test Step” because the icons are inactive, you might be in the wrong selection: Select the test object within the Test Project view, then select the Test Items view.

5.1.7 Entering test data

Now we will define some input and output values:

→ Switch to the perspective “TDE - Test Data Editor”. The TDE will also open with a double click on a test case or a test step.
In the Test Data view you can see the test cases and steps in tabular form.
→ Under “Inputs” click on the arrow to open “struct range r1”.
→ For test case 1 (1.1) enter *3* for “range_start”.
→ Enter *2* for “range_len”.
→ Enter *4* for “v1”.
→ Click on (Save) to save your inputs.
After saving, the symbol of the test object in the Test Project view as well as the symbol of the test case in the Test Items view turns yellow to indicate that the test case is ready to run (see figure 5.14).
→ Under “Outputs” click on the arrow ahead “Return”.
→ Enter “yes” for the return value.

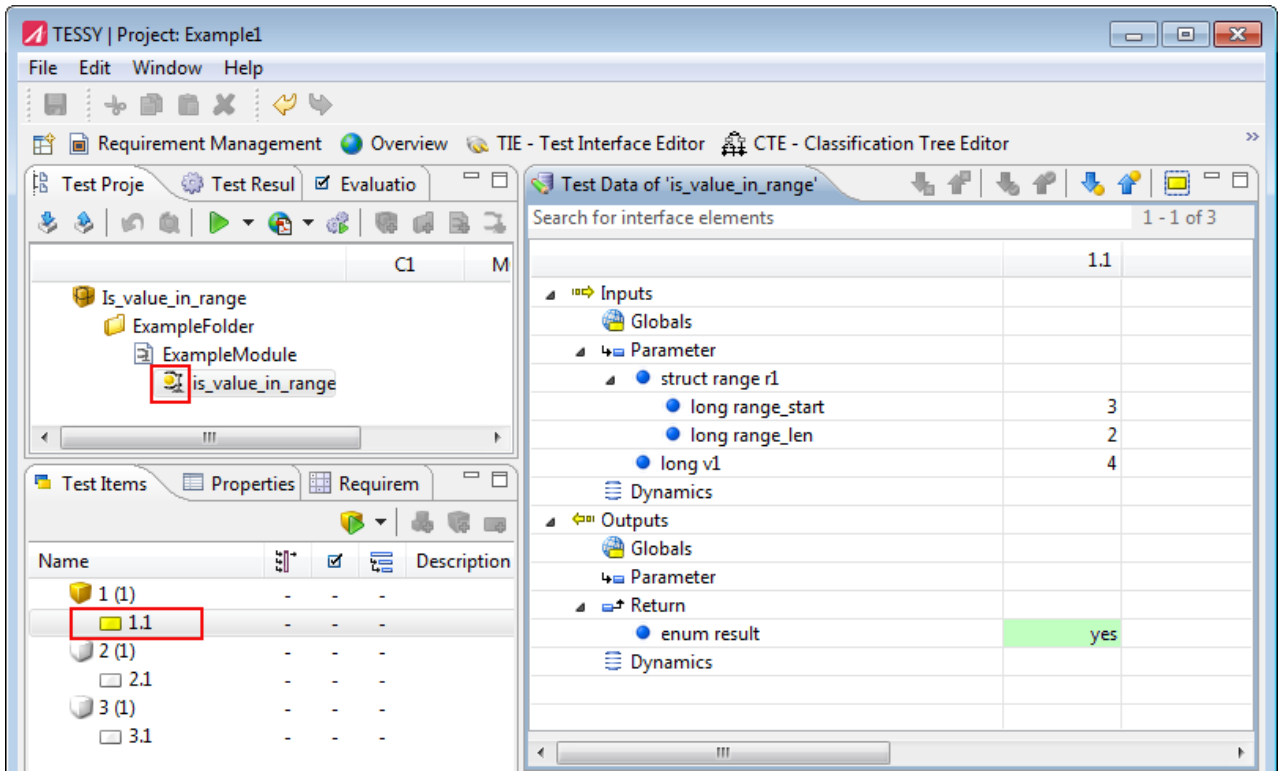


Figure 5.14: Data is entered, test step turns yellow and test case is ready to run.

Please notice the changes of the test object icon to indicate different conditions:

- Empty gripper: Test object is analyzed but has no test case.
- Gripper with white object: Test object has test cases but no data.
- Gripper with yellow object: At least one test step is ready to be executed.

→ Now enter data for the other two test cases as shown in table 5.1.

test step:	1.1	2.1	3.1
range_start:	3	20	0
range_length:	2	8	5
v1:	4	22	6
expected return:	yes	no	no

Table 5.1: Entering data for test object is_value_in_range.

- Test case 1.1: The range starts at **3** and has a length of **2**. Therefore, the range ends at 5 and the given value 4 is supposed to be inside of the range (**yes**).
- Test case 2.1: The range starts at **20** and has a length of **8**. Therefore, the range ends at 28 and the given value 22 is supposed to be inside of the range. Because we want to force an incorrect output, we state this to be not inside of the range (**no**).
- Test case 3.1: The range starts at **0** and has a length of **5**. Therefore, the range ends at 5 and the given value 6 is supposed **NOT** to be inside of the range (**no**).

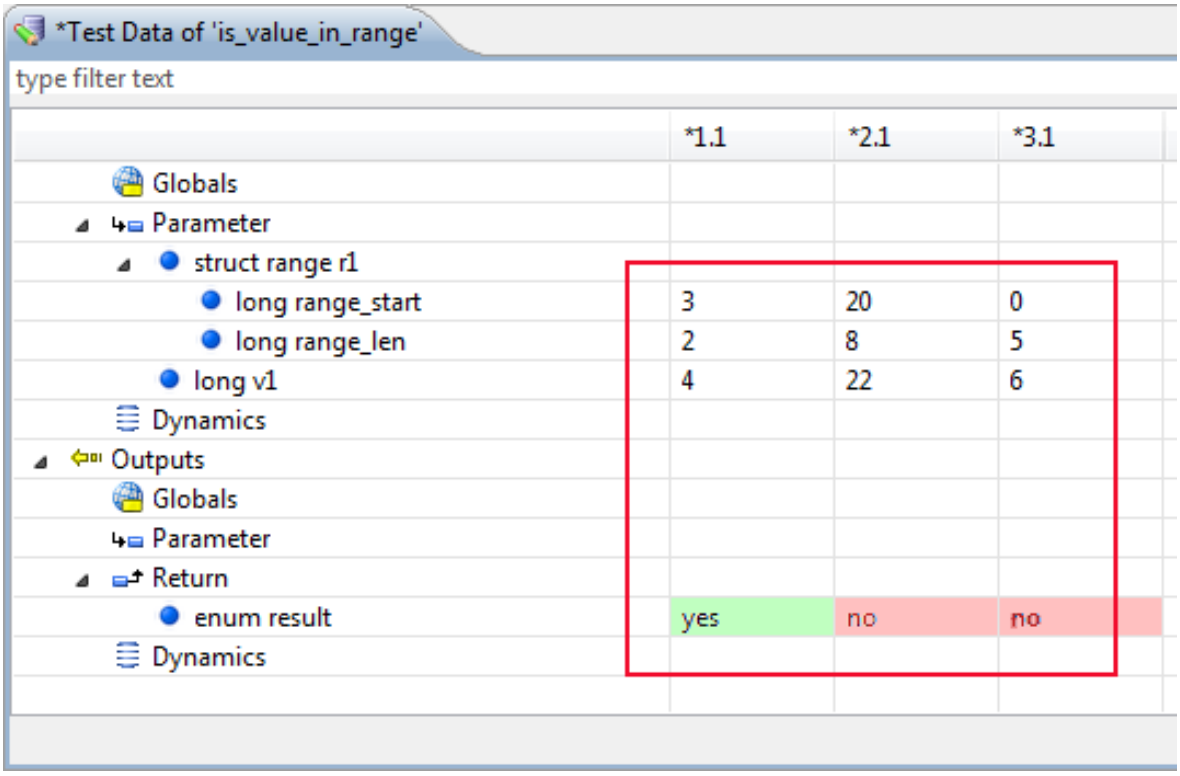


Figure 5.15: Entering data for test object is_value_in_range

The test step icons in the Test Items view will now turn to yellow (see figure 5.16). This indicates that we are now ready to run the test.

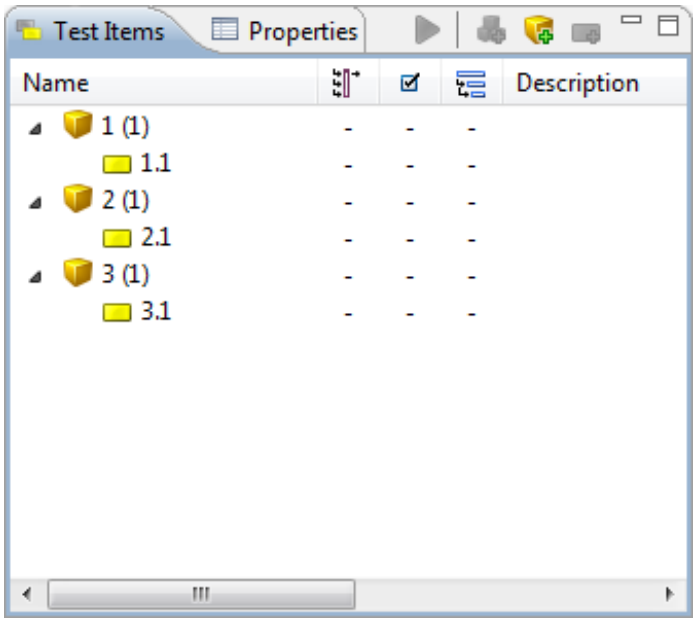


Figure 5.16: The test cases are ready to test

5.1.8 Executing the test

→ Click on (Start Test Execution) in the tool bar of the Test Project view.
A progress dialog will be shown while TESSY generates, compiles and links the test driver and runs the test. This will take a few seconds.

After the test run, test case icons (within TDE) should be (see figure 5.17):

- Within the Test Data view the second test step is marked with a red cross and the expected result “no” is marked red to indicate, that the result did not match the expected result (the actual result is “yes”).
- Within the Test Project view the test collection, folder, module and test object are marked with a red cross to indicate, that not all results did match the expected results.
- The Test Items view indicates with a red cross, that test case 2 did not match the expected result.

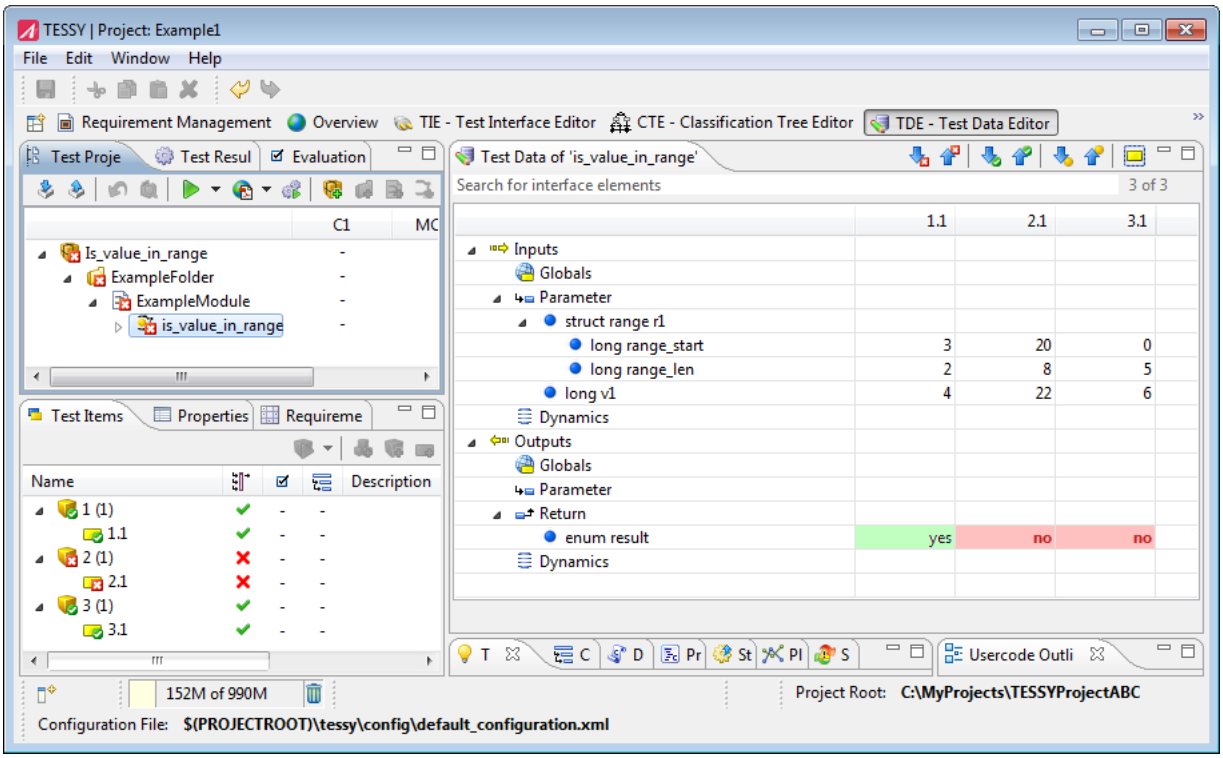


Figure 5.17: TDE after test run is_value_in_range

→ Switch to the Overview perspective.

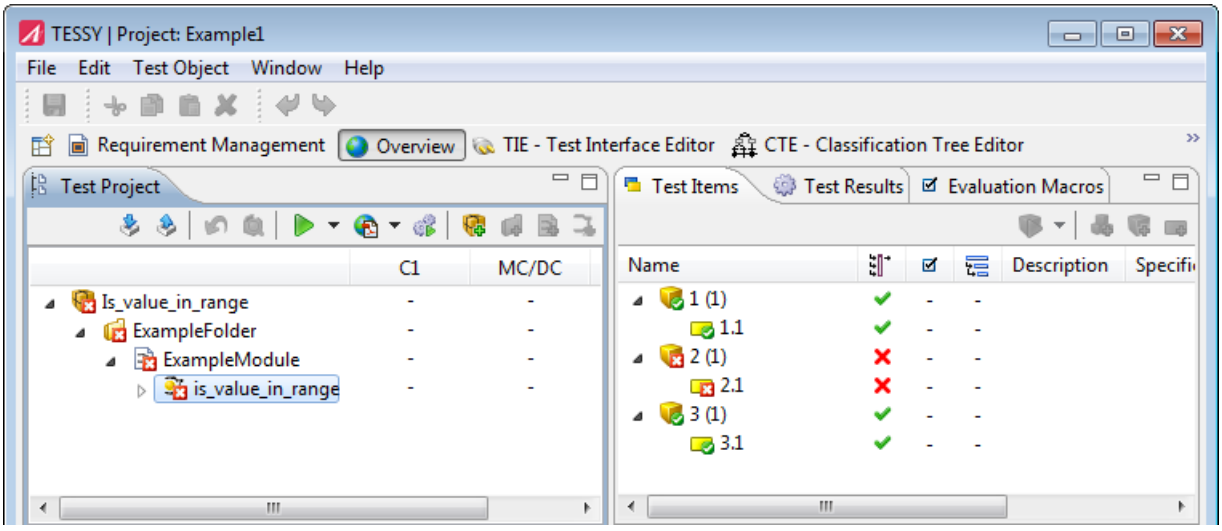


Figure 5.18: Test results of is_value_in_range

You can see the results of every test step within the Test Results view.

5.1.9 Repeating the test run with coverage instrumentation

To analyze the source code coverage of the test, repeat the test run with the branch, MC/DC and MCC-coverage instrumentation:

- → In the tool bar of the Test Project view click on the arrow next to the Execute Test icon and select “Edit Test Execution Settings…” .
- → In the following dialog tick the boxes “Run” (default) and “Create New Test Run”.
- → Choose the instrumentation “Test Object” and untick the box “Use preselected coverage”.
- → Tick the boxes for “Branch Coverage (C1)” and “Modified Condition / Decision Coverage (MC/DC)” (see figure 5.19).
- → Click on “Execute”.

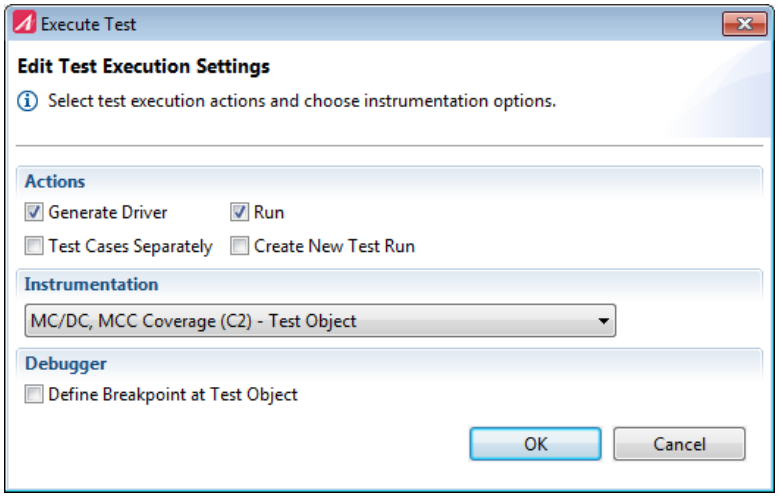


Figure 5.19: Selecting Branch and MC/DC Coverage for test run

A progress dialog will be shown while TESSY generates, compiles and links the test driver and runs the test. This will take a few seconds.

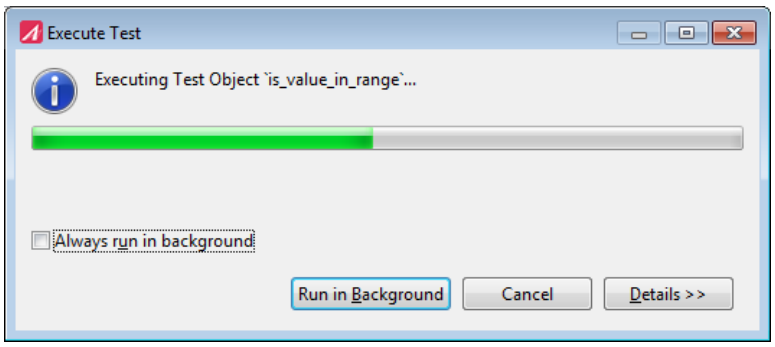


Figure 5.20: Execute Test dialog while running the test

5.1.10 Analyzing the coverage

→ Switch to the Coverage Viewer (CV) perspective.

Analyzing with the CV

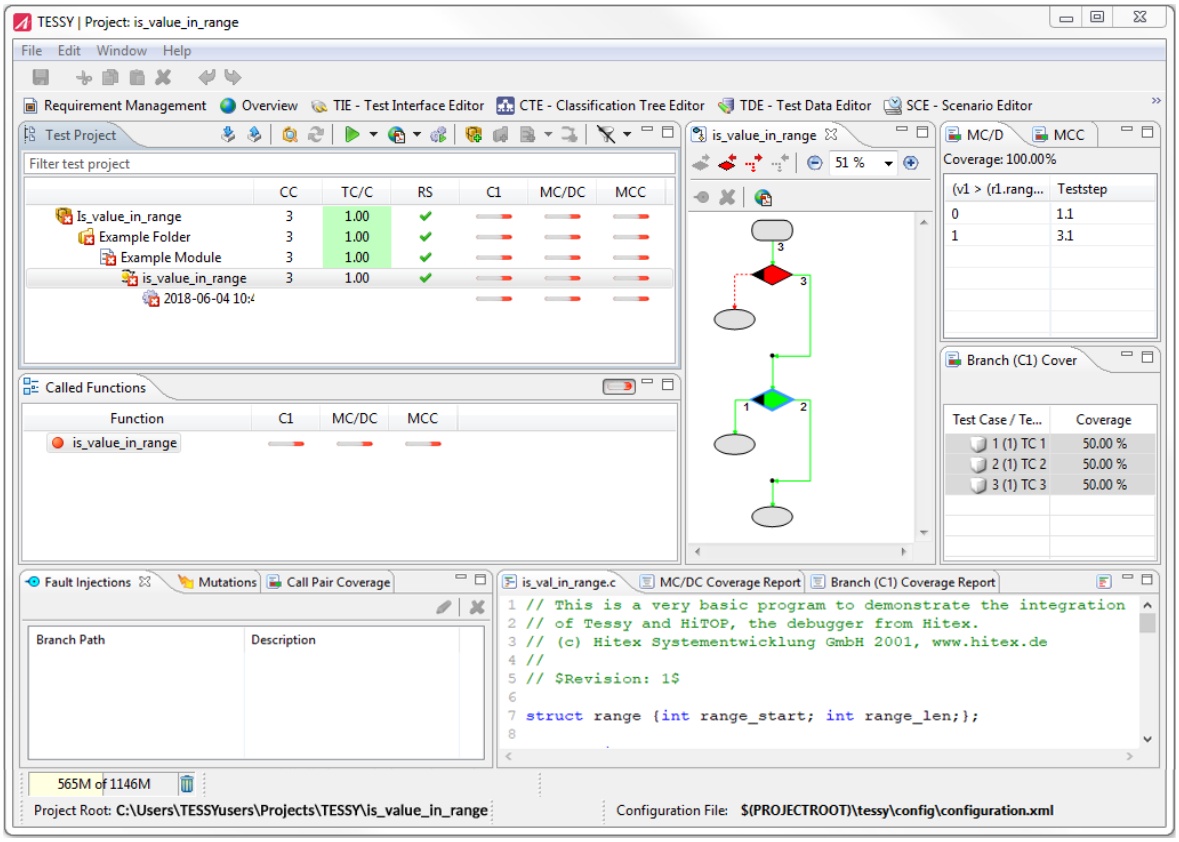


Figure 5.21: The Coverage Viewer displays the coverage of is_value_in_range

The CV shows the results of the coverage measurement of a previously executed test.

You can select a default coverage measurement as well for your whole project or any specific module or test object. Refer to section 6.2.3.4 Coverage tab.

5.1.10.1 The Flow Chart view

The Flow Chart view displays the code structure and the respective coverage in graphical form. Within each flow chart, you will see the decisions and branches of the function being displayed. Green and red colors indicate whether a decision has been fully covered or a branch has been reached.

5.1.10.2 The Branch (C1) Coverage view

The Branch C1 Coverage view (see figure 5.22) displays the branch coverage for each individual test case and test step as well as the total coverage for all test cases and test steps.

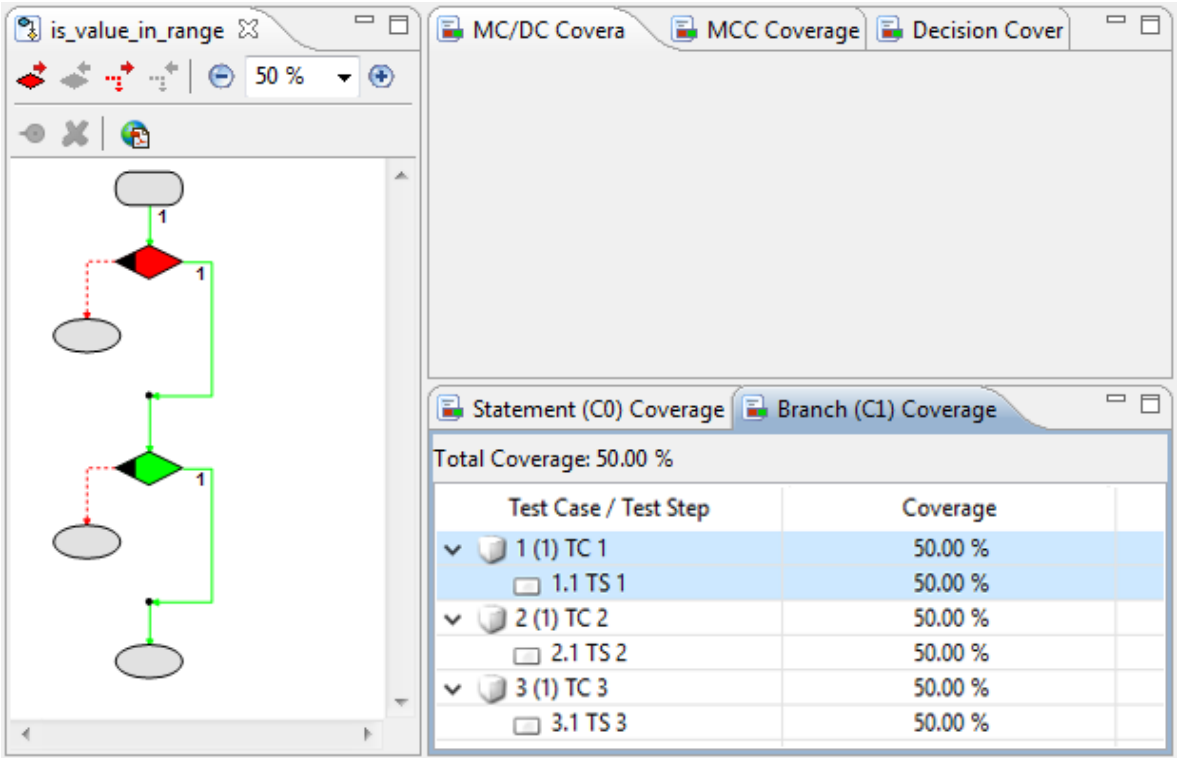


Figure 5.22: Branch coverage is_value_in_range

5.1.10.3 The MC/DC Coverage view

The MC/DC-Coverage view displays the coverage of the currently selected decision within the Flow Chart view (see figure 5.23). If no decision is selected (as initially when starting the CV), the MC/DC Coverage view is empty.

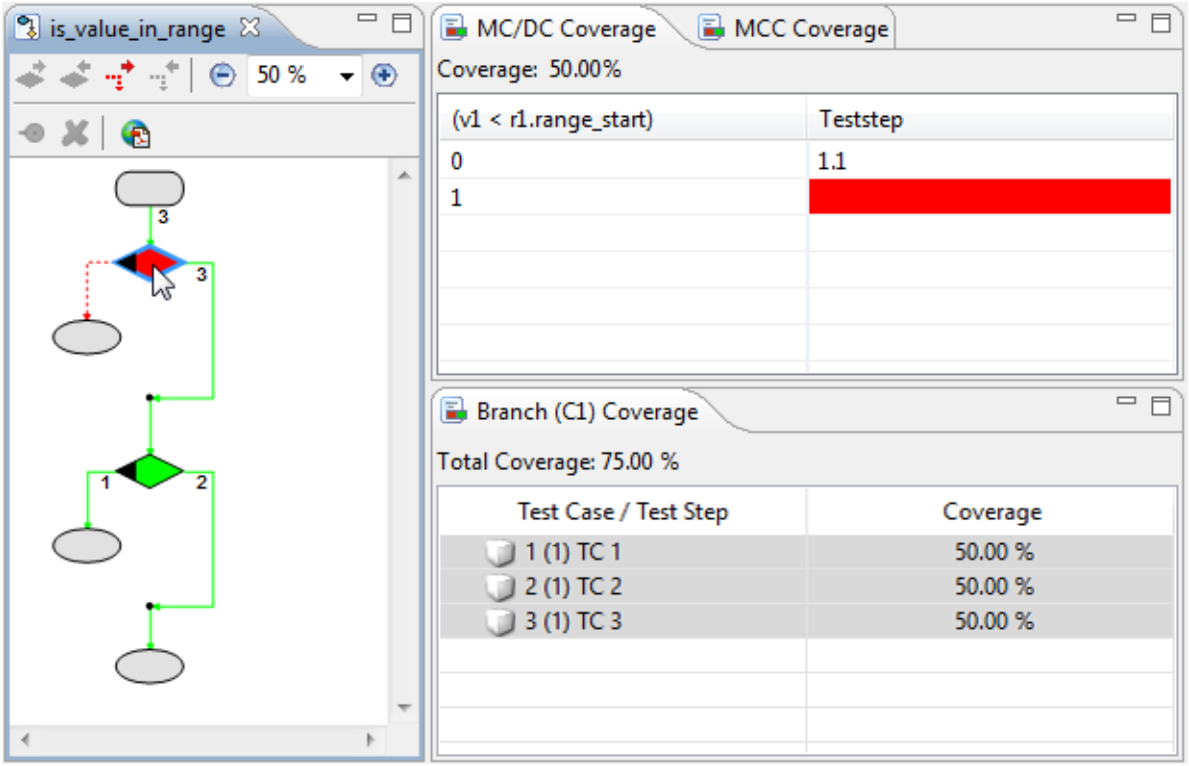


Figure 5.23: Decision coverage is_value_in_range

The current example is_value_in_range has only simple decisions, for which MC/DC is the same as branch coverage.

5.1.10.4 Analyzing

In our example you can see in the flowchart, that

- three test cases were executed (each with one test step).
- the if branch on the left of the first decision was not reached and is therefore marked red.
- the first decision was not fully covered, so it is marked red.
- the second decision was fully covered and is therefore marked green.
- the else branch on the right of the second decision was reached two times, the else branch was reached once.

→ Select the red branch of the first decision (the if branch).
The respective code section is highlighted within the source code view (see figure 5.24). This allows finding out the execution path of the selected test step.

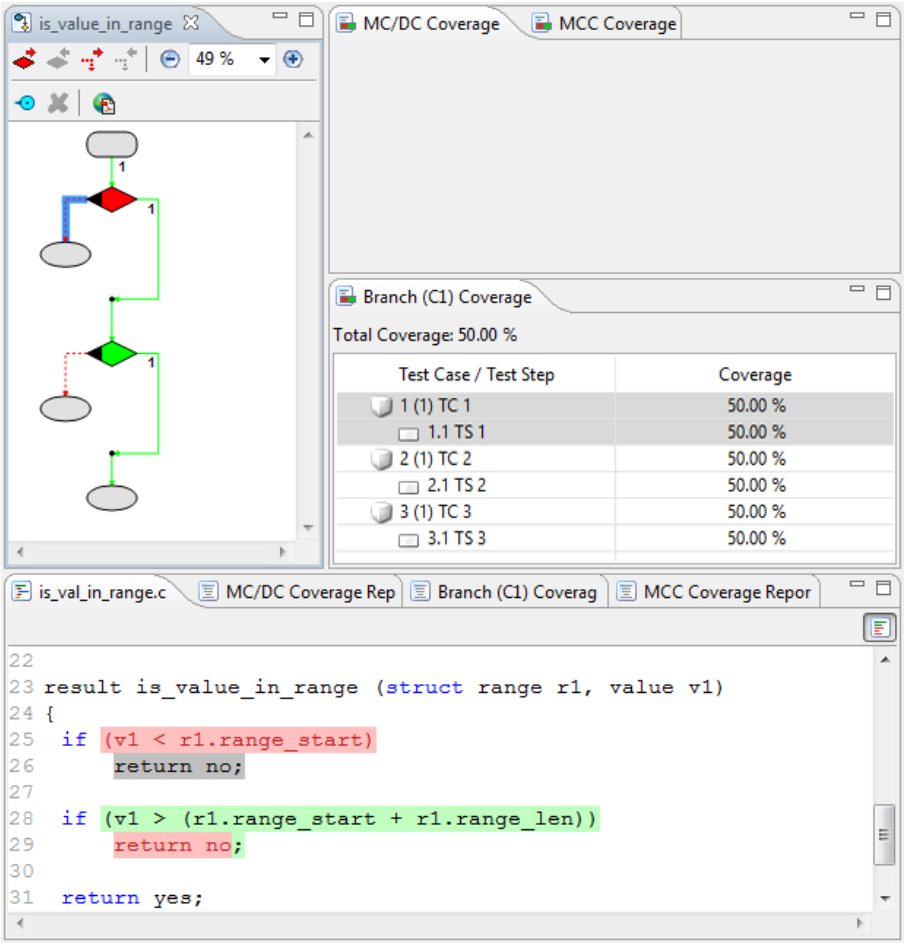


Figure 5.24: Code section of the if branch of the first decision

→ Select the second decision.
The respective code section is highlighted within the source code view (see figure 5.25).

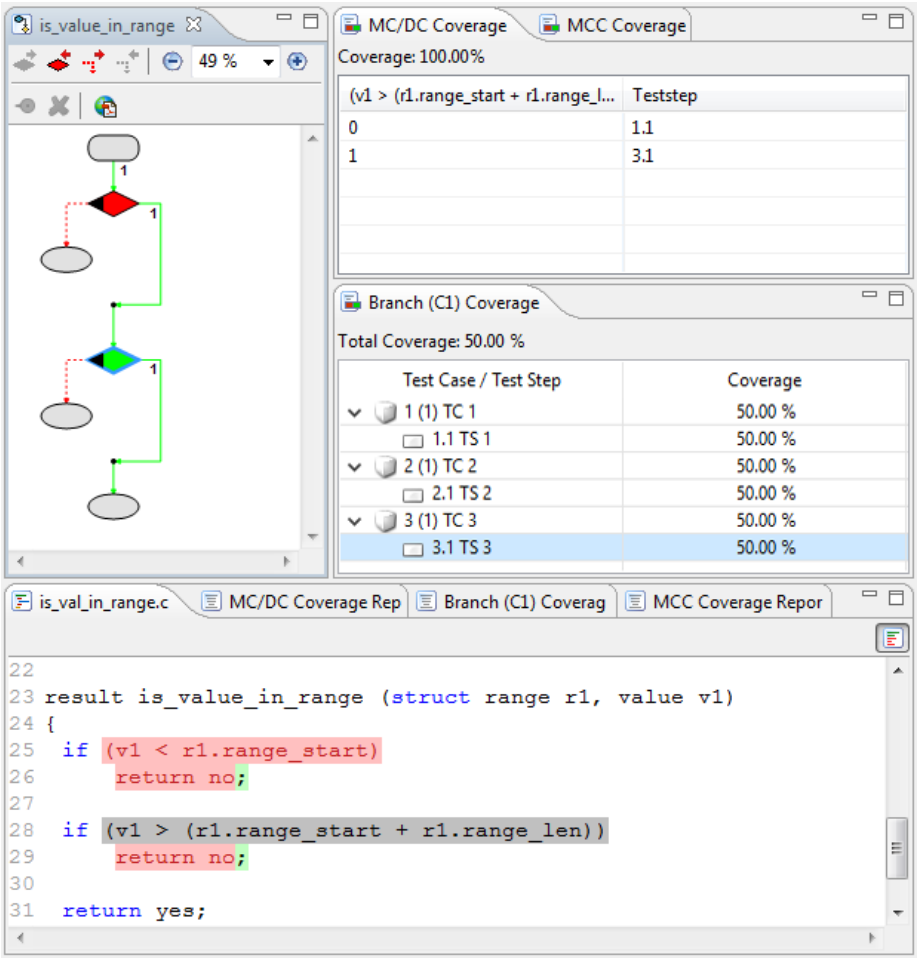


Figure 5.25: Code section of the second decision

5.1.11 Creating a Test Details Report

→ In the Test Project view of the Overview perspective click on the arrow next to the Generate Report icon and select “Edit Test Details Report Settings…” .

→ In the dialog you can select the Report Options you need.

→ You can also change the Report Output Directory by clicking on “…” and create a new folder by clicking “Make New Folder” in the additionally opened window (see figure 5.26).

The selection of the new folder will only be effective for this report. Any further report generations will again be based on the settings within the Test Report Options in the Preferences.

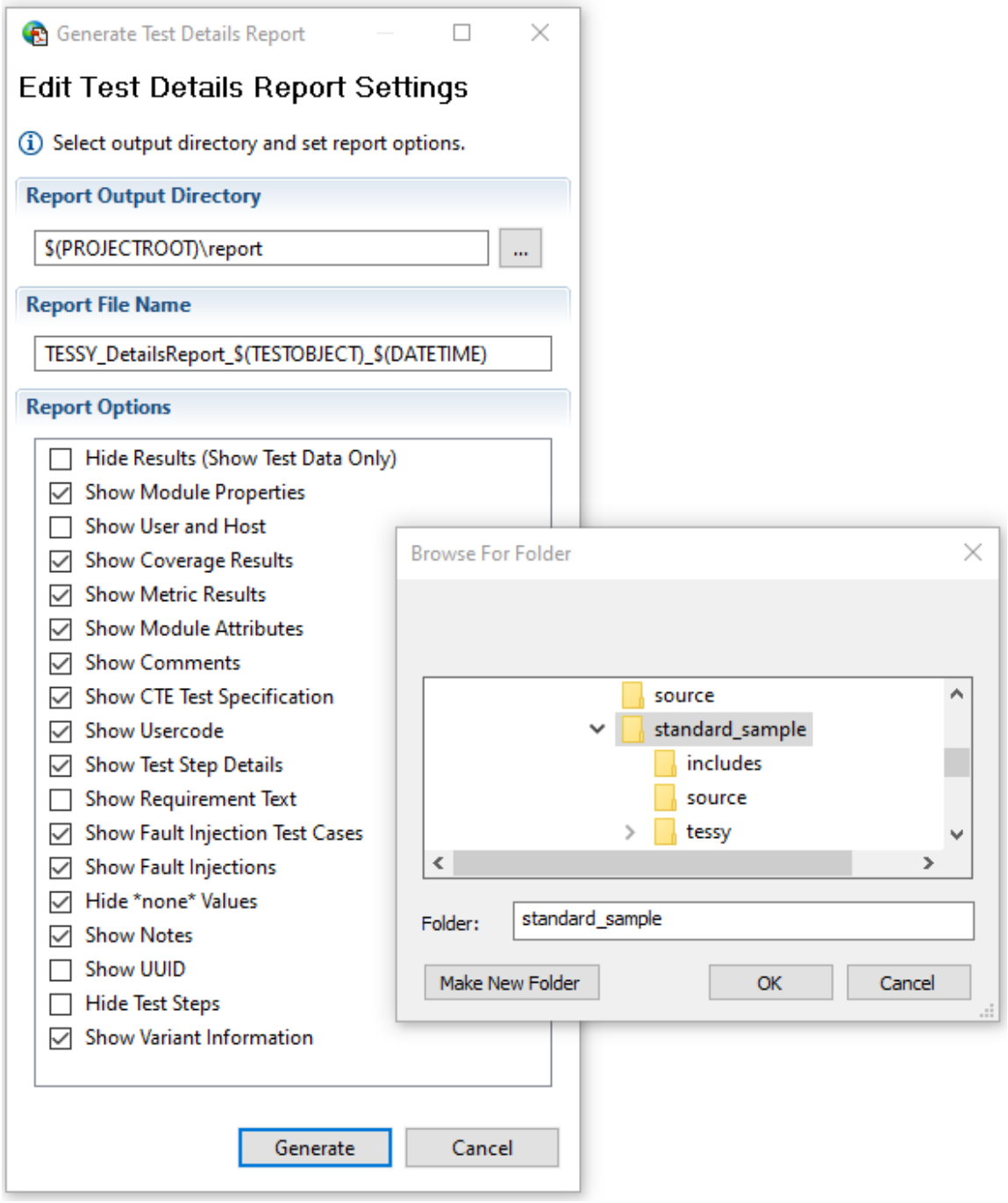


Figure 5.26: Selecting a folder or creating a new folder for Test Details Reports

→ Click on “OK” and “Generate”.
TESSY will now create the report within the new folder. This will take a few seconds.
When finished, TESSY will open the report file (PDF) automatically.

Important: The generation and opening of reports requires a third party PDF viewer. If you get the error message “No matching program found for the file…”, no such viewer was found.

This either means that there is no suitable PDF viewer installed on your computer. The other possible reason is that you need to associate pdf files with your third party PDF software in Windows 10.

After the source of error is found and fixed you need to generate the report again.

A Test Details Report basically looks like this:

TEST DETAILS REPORT *Aug 8, 2012, 7:38:48 A*

is_value_in_range

Project	Example1
Module	ExampleModule
Test Object	is_value_in_range

Statistics

Total Testcases	1
Successful	1
Failed	0
Not Executed	0

Module Properties

Directory	C:\MyProjects\Example1\tessy\work\00000002\00000004\00000006
Target Environment	GNU GCC GNU GVD (Default)
Kind of Test	Unit Test
Source Files	$(PROJECTROOT)\tessy\sources\is_val_in_range.c
Compiler Options	

Figure 5.27: Content of the test report is_value_in_range

Now you have completed your first test project!

If you use a Version Control System (VCS) providing keyword expansion to embed version control information in your source files, TESSY will display such expanded keywords within the test report.

The following keywords are available: $Revision$ (Revision number), $Author$ (User who checked in the revision) and $Date$ (Date and time stamp for the revision).

5.1.11.1 Change the default Test Report Options for your project

It is possible to permanently change the default report settings for your project, e.g. Output Directories, File Names and Logo Image. This applies for all possible reports, not only the Test Details Report.

You may also change the default Razorcat logo within the reports to your own company logo with your logo image file (PNG, JPG or GIF).

To set the Output Directory for the Test Details Report for your project permanently:

→ In the menu bar select “Window” > “Preferences” > “Test Report Options”.

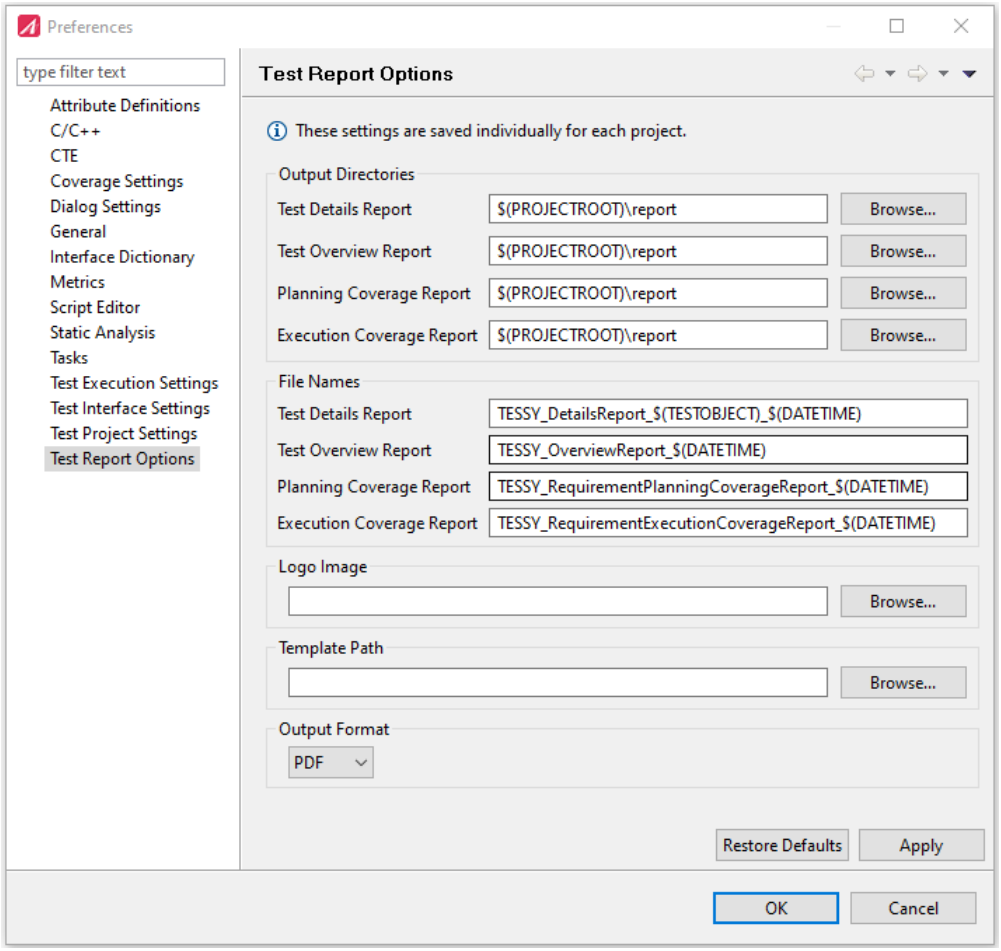


Figure 5.28: The Test Report Options in the Preferences

→ Click on “Browse…” in the line “Test details Report” to select the desired directory or change the path manually.

→ Click “OK” to safe your settings.

Important: It is not recommended to use HTML or Word as an output format because of potential layout issues. They are provided as complementary helper formats only and without further support.

For more information about creating various reports please refer to section 6.2.2.16 Creating reports.

5.1.12 Repeating the test run with requirements

Requirement Management

We will now import some very basic requirements and repeat some steps of this exercise. This way you get to know the feature of requirement management and you can consolidate the just learned workflows.

5.1.12.1 Importing requirements

→ Copy the example requirements document “Is Value In Range Requirements.txt” into a folder of your test project “TESSYProjectABC”. It is located within the TESSY installation directory
`C:\Program Files\\Razorcat\TESSY_4.x\Examples\IsValueInRange`.
→ Switch to the Requirement Management perspective.
→ Right-click within the blank RQMT Explorer view and select “Import” from the context menu (see figure 5.29).

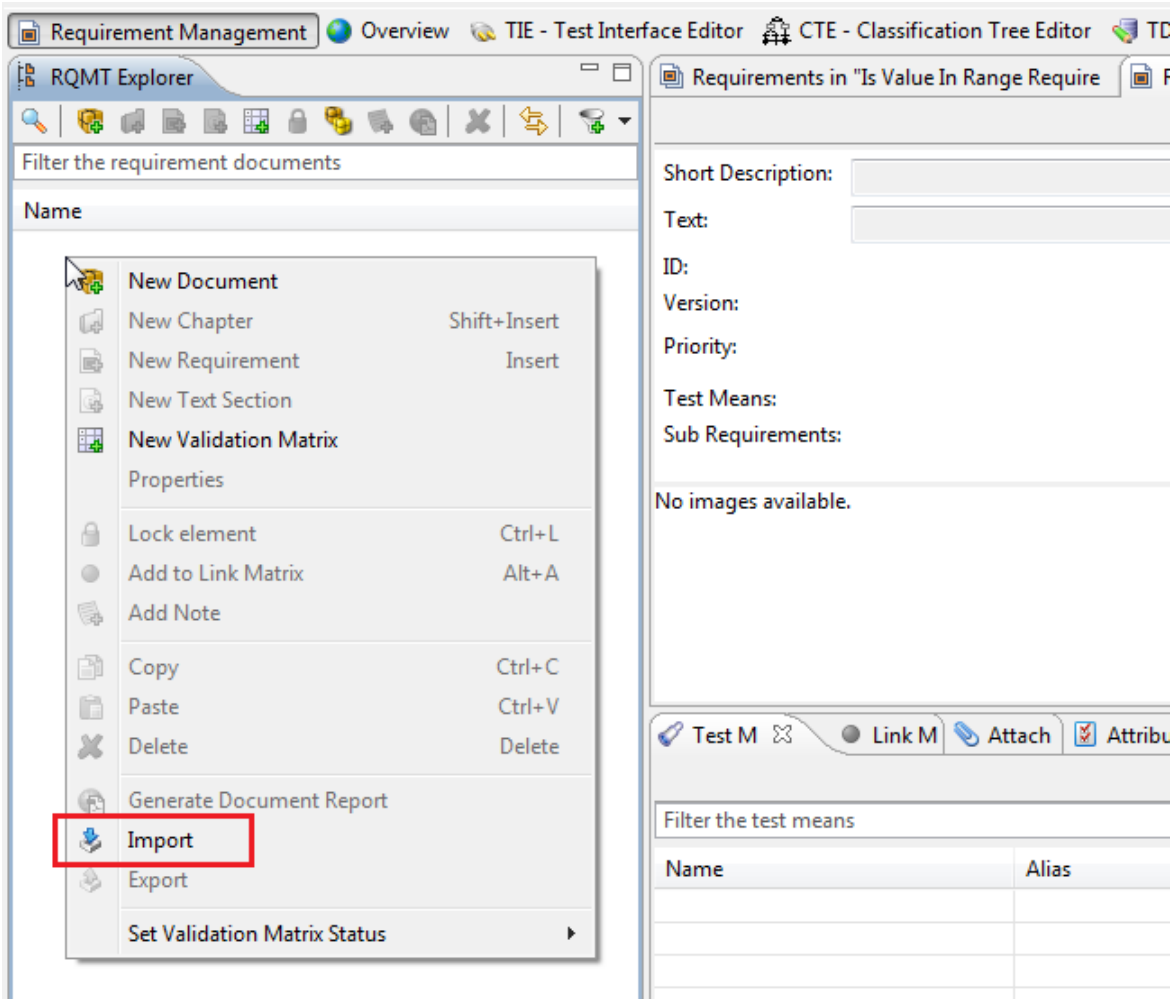


Figure 5.29: Importing a requirement

In the following Import dialog you can import various file formats. In our example we select the file we just copied into our project:

→ Click on "…" and select the file "Is Value In Range Requirements.txt" from your project.

→ Leave the File Content Type and the Target Document as it is and click "OK" (see figure 5.30).

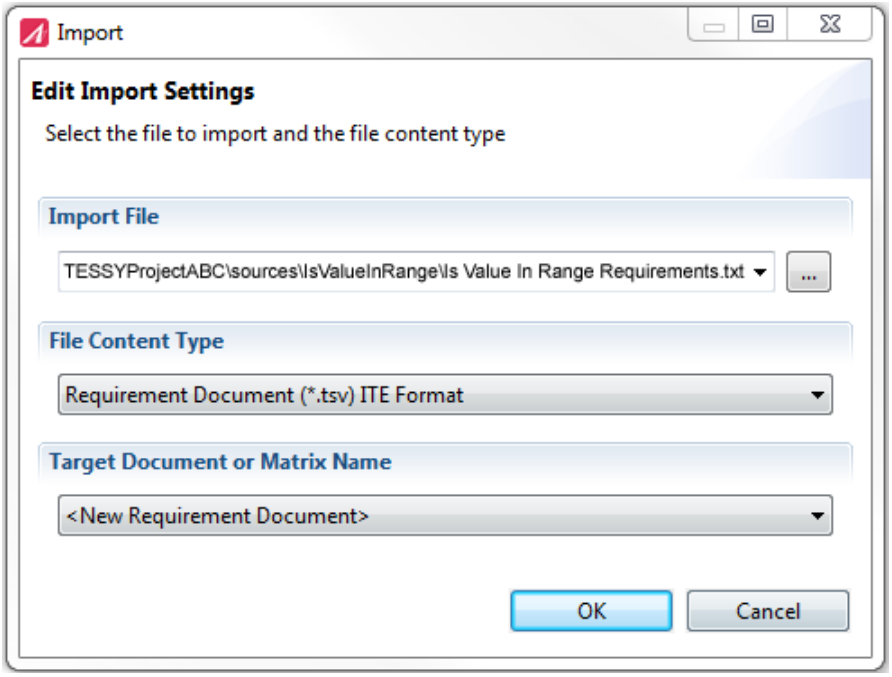


Figure 5.30: Import dialog

The newly imported requirement document will be displayed in the RQMT Explorer view (see figure 5.31).

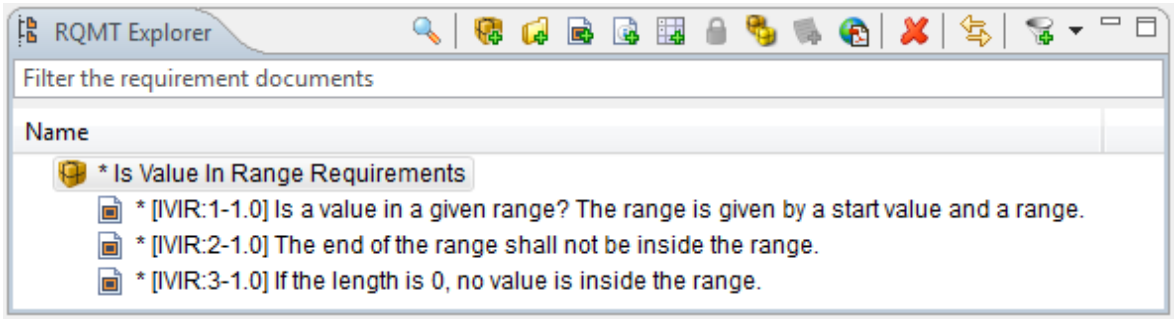


Figure 5.31: The new requirement document

→ Right-click the document and select “Properties” from the context menu.

→ Change the alias to “IVIR” and click “OK” (see figure 5.32).

The document alias will be used for reporting, in order to have an abbreviation of the document name when building the requirement identifier, e.g. IVIR-[1.0] in our example.

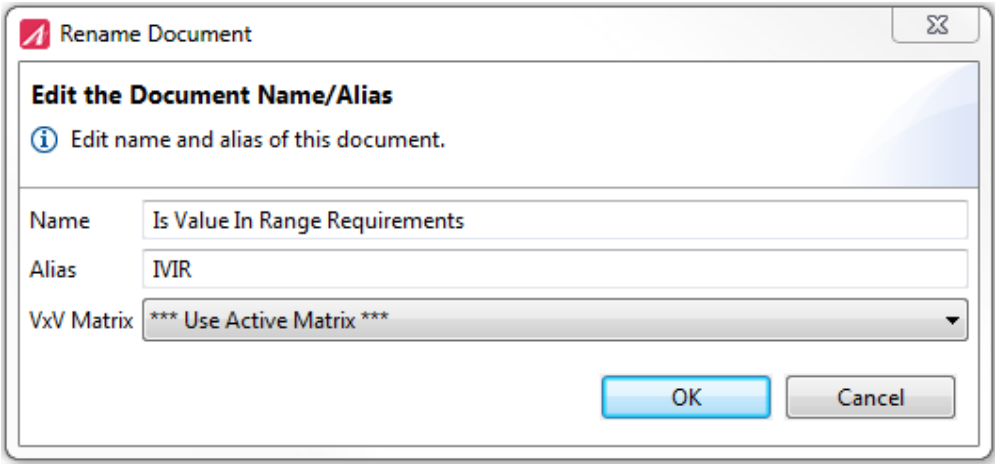


Figure 5.32: Changing the alias of the new requirement document

5.1.12.2 Committing the requirements document

Committing RQMTs document

Before linking any tests to a requirement, the respective requirements document needs to be checked in as initial revision:

→ Select the document and click on (Commit Changes) in the global tool bar.
→ Enter “Initial revision” as commit comment and click “OK” (see figure 5.33).
An initial revision of the requirement document will be created.

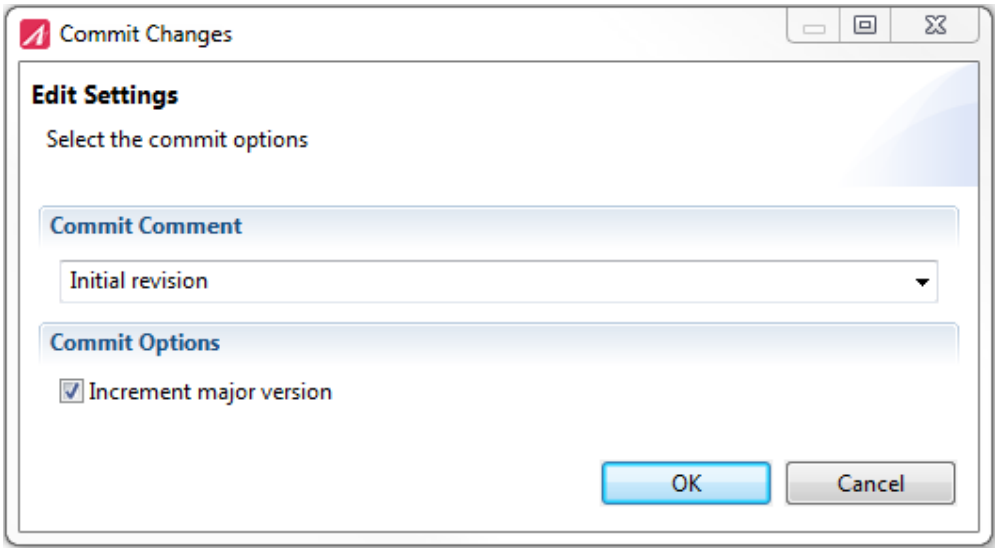


Figure 5.33: Comment for the initial revision of the commit

TESSY manages different versions of a requirements document. You can track any changes either from importing updated versions or from any modifications that you did directly within TESSY.

5.1.12.3 Linking test cases with requirements

→ Switch to the Overview perspective and select the test object.
→ Select the Requirements Coverage view.
→ Click on (Always show unlinked Requirements) in the Requirements Coverage view.
The view shows the imported requirements and the module, test object and test cases in a tree-based arrangement.

Important: If the view says “No requirements available”, select the test object in the Test Project view!

Use the toggle buttons on the right to link modules, test objects or test cases to requirements:
→ Link the first test case with the first requirement.
→ Link the second test case with the second requirement (see figure 5.34).
The third requirement is not linked.

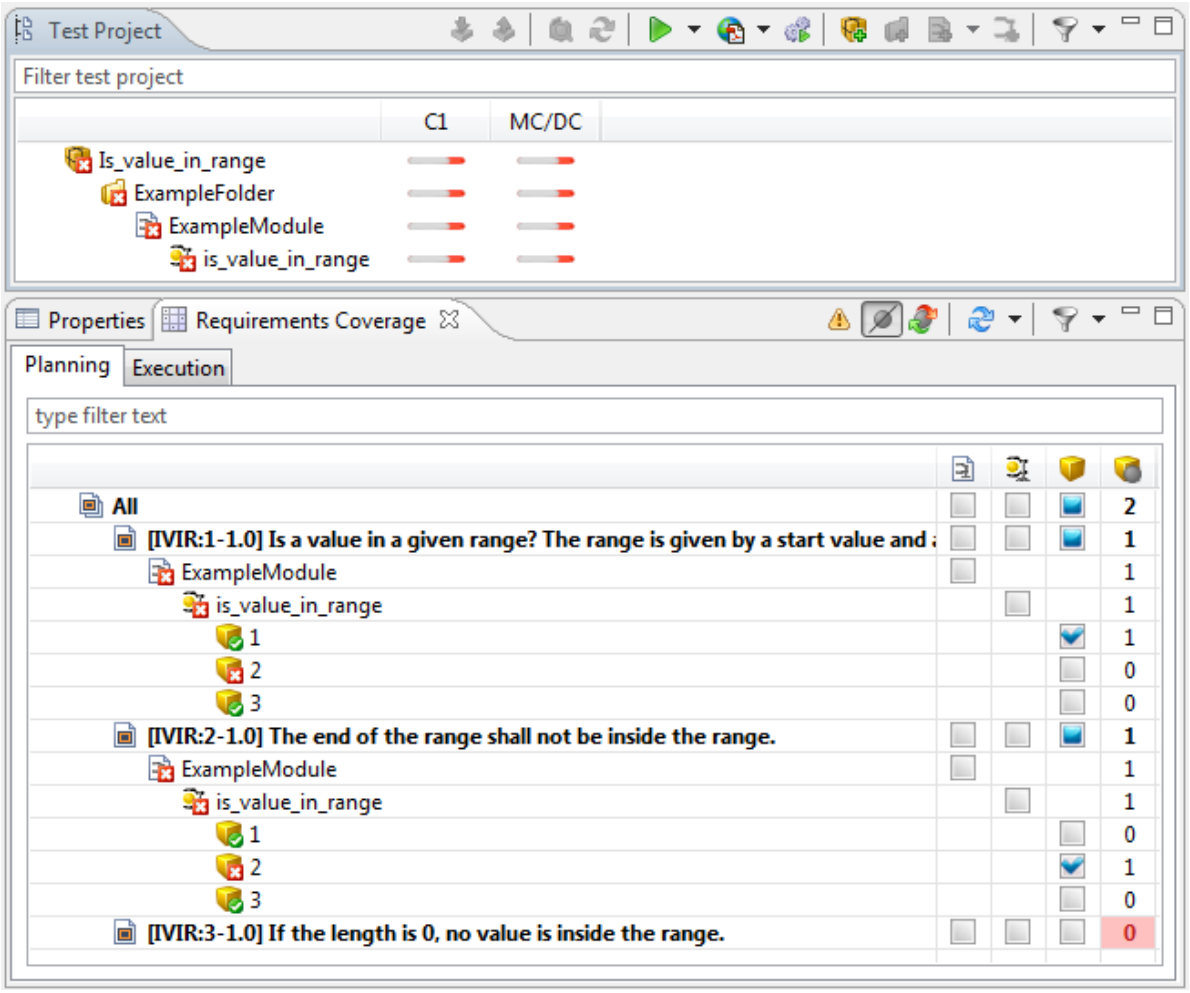


Figure 5.34: Linking test cases with requirements

Remember the values of our test cases:

- Test case 1.1: range start 3, length 2, given value 4, supposed to be inside of the range (yes)
- Test case 2.1: range start 20, length 8, given value 22. Because we wanted to force an incorrect output, we stated this to be not inside of the range (no)

→ Switch to the TDE perspective.

→ In the Test Item view select the first test case and have a look at the Test Definition view: It shows the requirements we just linked with our test cases.

→ Select the second test case.
The second requirement will be displayed (see figure 5.35).

Since we did not link any requirement to the third test case, the “Linked Requirements” will be empty when selecting the third test case.

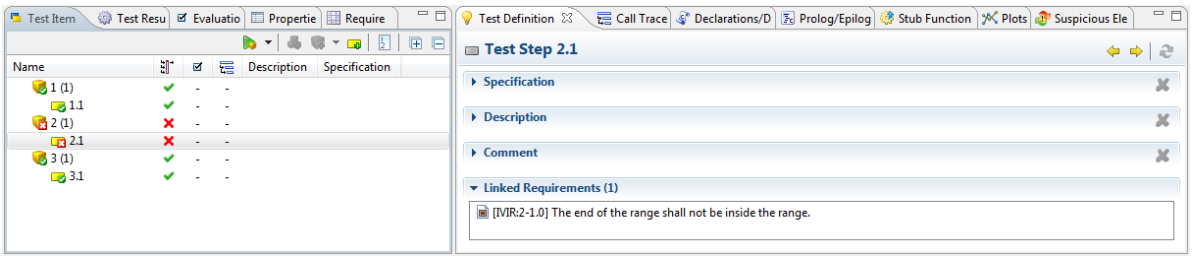


Figure 5.35: Test Definition view within TDE with linked requirement

5.1.12.4 Creating a planning coverage report

At this stage we can already generate a report showing the planned test case for our requirements:

→ Switch to the Test Project view of the Overview perspective and click on the arrow next to the Generate Report icon .

→ Select “Edit Planning Coverage Report Settings…” (see figure 5.36).

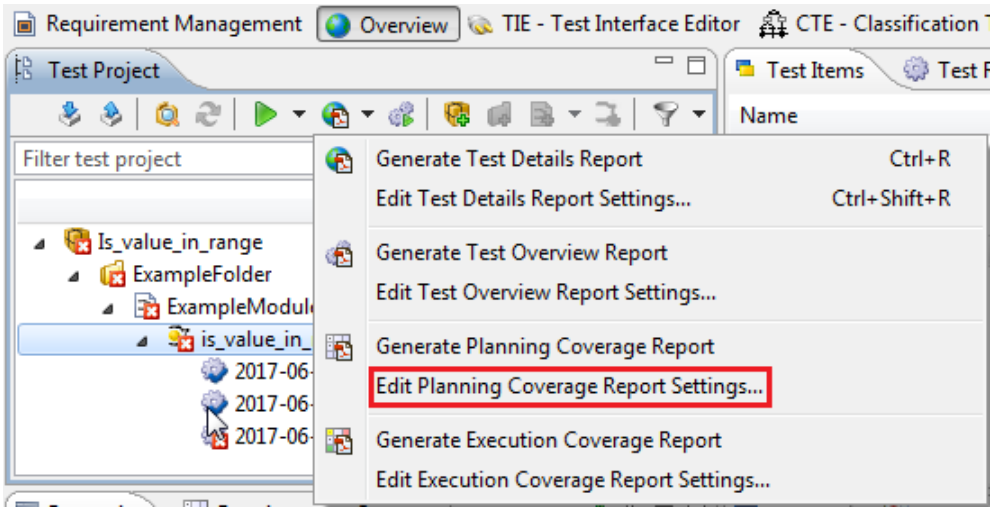


Figure 5.36: Editing the settings of a Planning Coverage Report

→ A dialog for the settings for the Planning Coverage Report will open.

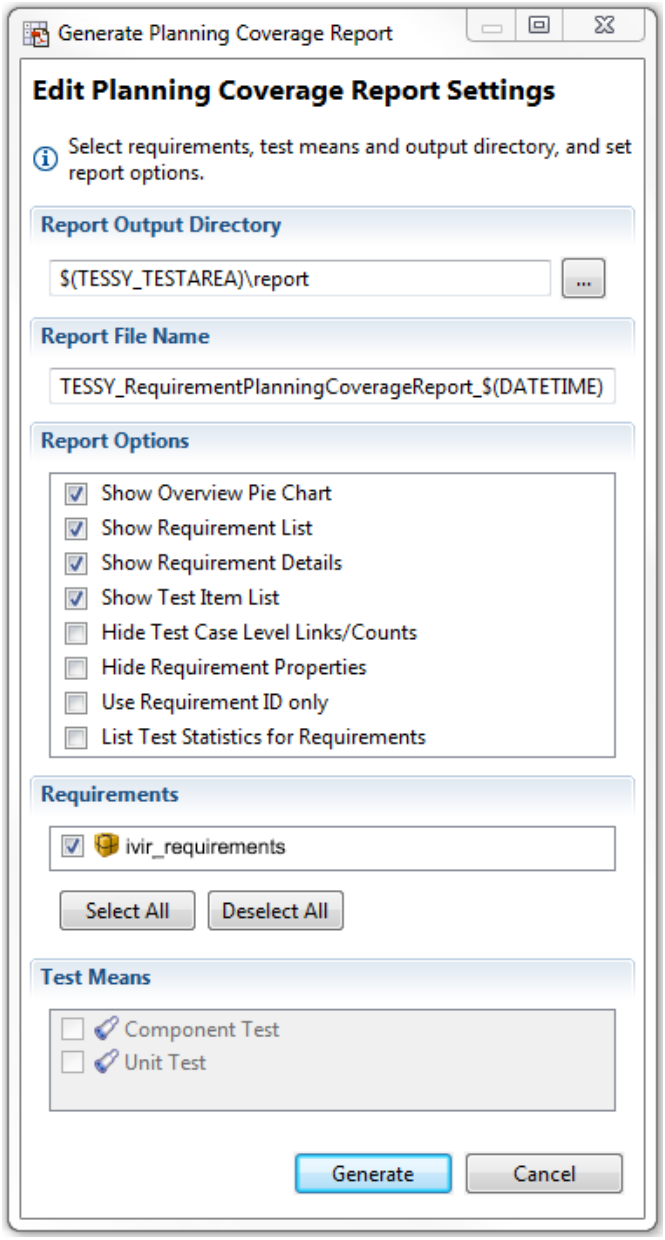


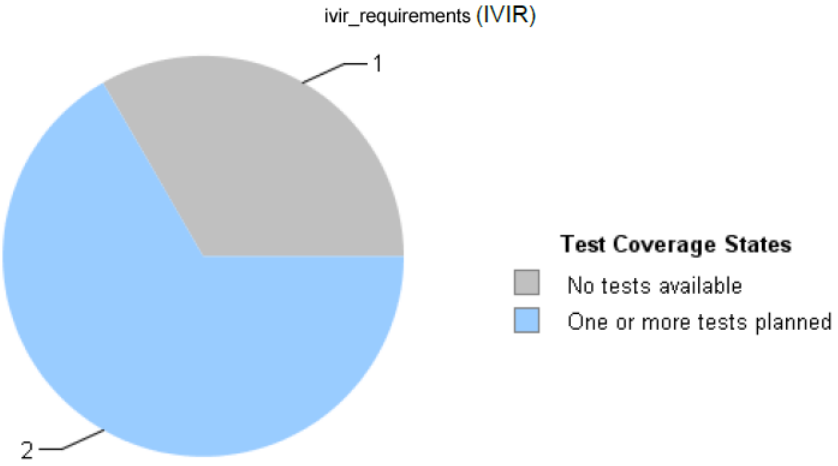
Figure 5.37: Dialog of the settings for the Planning Coverage Report

Planning Coverage Report

→ Select an output directory for the report (default: `C:\TESSY\report`).
→ Select the first four Report Options (default).
→ Select the Requirement.
→ Do NOT select any Test Means.
→ Click on “Generate”.
A planning coverage report will be created.

PLANNING COVERAGE REPORT *2017-09-04,16:24:12+0200*
Project: Example1

Requirement Test Coverage Overview



Coverage Overview Arranged by Requirement

Identifier | Text	*State*	*Number of Tests*
[IVIR:1-1.0] Is a value in a given range? The range is given by a start value and a range.	Tests planned	1
[IVIR:2-1.0] The end of the range shall not be inside the range.	Tests planned	1
[IVIR:3-1.0] If the length is 0, no value is inside the range.	No Tests available	0

Figure 5.38: Planning coverage report of the IVIR requirement document

The report shows the available requirements and the linked test cases. It provides an overview about the planned tests if all requirements are covered by at least one test case.

Since we have links to two of our requirements, the resulting requirement coverage should be as shown above.

Notice the usage of the requirement document name and alias within the report! It is important to select an appropriate alias in order to get useful report outputs.

We have planned test cases for the first two requirements, whereas the third requirement is not yet linked with any test case, because there are no tests available to validate this requirement.

5.1.12.5 Executing the test and examining the coverage

We will now execute our tests again to see the results of the test cases with respect to the linked requirements within the execution coverage report.

→ Switch to the Overview perspective and execute our test object is_value_in_range again: Click on the Execute Test icon .

→ Generate a test details report to review the results on test object level: Click on the arrow next to the Generate Report icon (see figure 5.39).

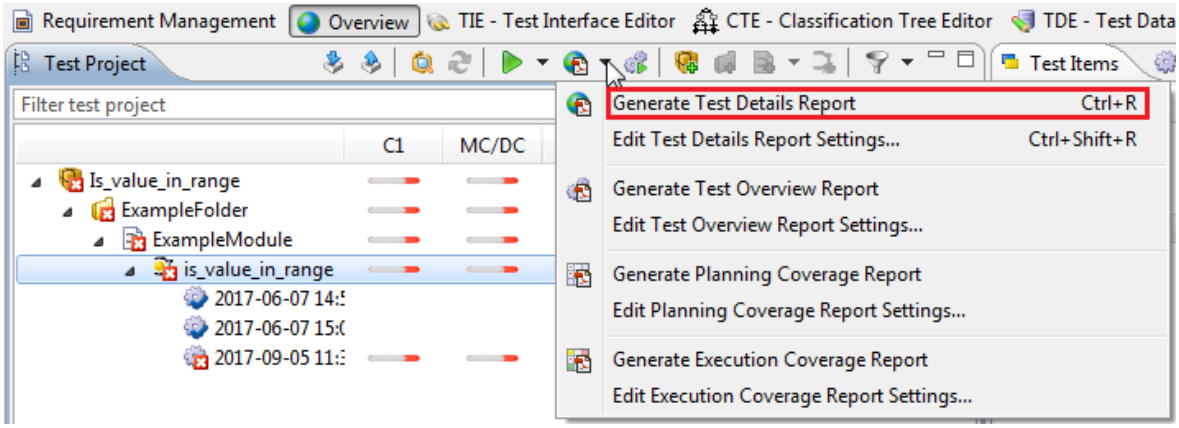


Figure 5.39: Generating a Test Details Report

The report will open automatically.

→ Have a look at the second page.
The report will show additional paragraphs with the linked requirements after the overview pages and for each test case (see figure 5.40):

Test Details Report

TEST DETAILS REPORT 2017-09-05, 11:33:33+0200

is_value_in_range

Test Case 1

Linked Requirements [IVIR:1-1.0]

Test Step 1.1 (Repeat Count = 1)

Name	Input Value
r1.range_start	3
r1.range_len	2
v1	4

Name	Actual Value	Expected Value	Result
is_value_in_range()	yes	yes	✔

Test Case 2

Linked Requirements [IVIR:2-1.0]

Test Step 2.1 (Repeat Count = 1)

Name	Input Value
r1.range_start	20
r1.range_len	8
v1	22

Name	Actual Value	Expected Value	Result
is_value_in_range()	yes	no	✖

Figure 5.40: Part of the generated test report of is_value_in_range

Execution Coverage Report

Now we will generate a coverage report showing the test case results with respect to our requirements:

→ In the global tool bar click on the arrow next to the Generate Report icon and select “Generate Execution Coverage Report” (see figure 5.41).

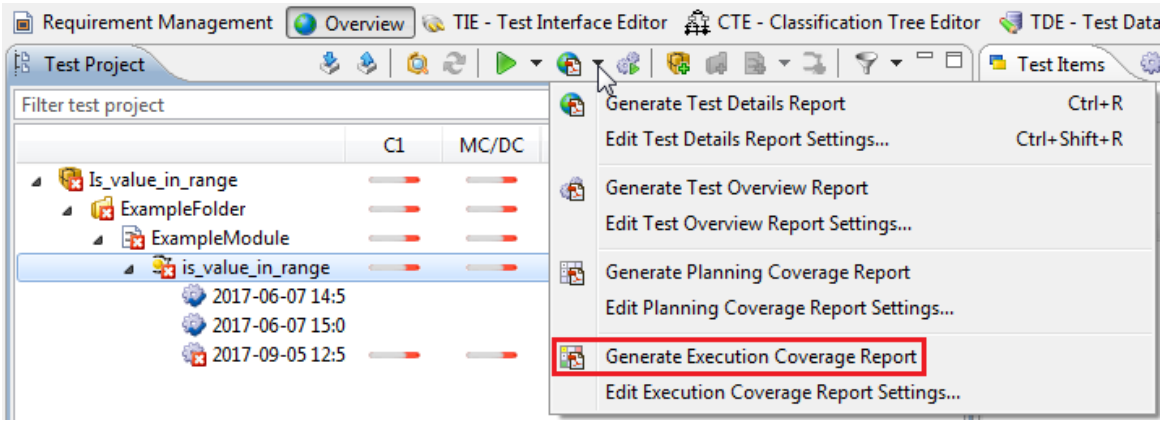


Figure 5.41: Creating an Execution Coverage Report

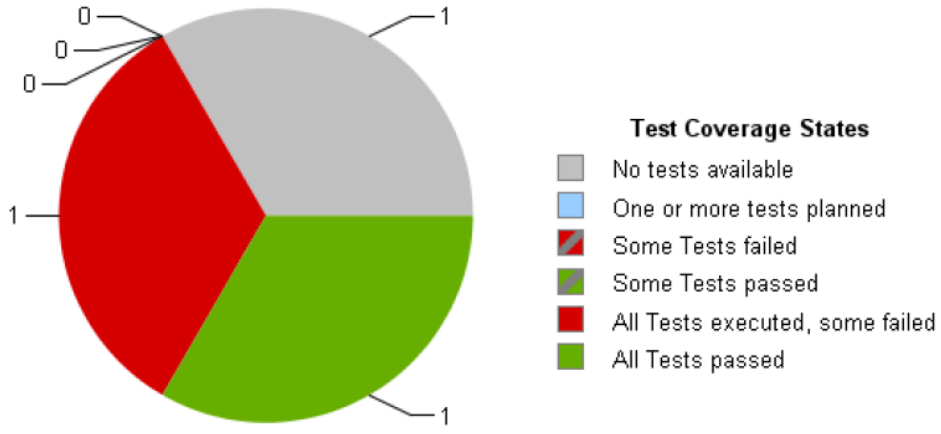
TESSY creates the coverage report showing the available requirements and the results of the linked test cases. It provides an overview about the current test status, e.g. if tests for any requirements are failed.
Since one of our test cases was passed while the other one was failed, the resulting requirement coverage should be as in figure 5.42.

Test Coverage Report

TEST COVERAGE REPORT
Project: Example1
2017-09-05,13:49:49+0200

Requirement Test Coverage Overview

ivir_requirements (IVIR)



Coverage Overview Arranged by Requirement

Identifier | Text	*State*	*Number of Tests*
[IVIR:1-1.0] Is a value in a given range? The range is given by a start value and a range.	All Tests passed	1
[IVIR:2-1.0] The end of the range shall not be inside the range.	All Tests executed, some failed	1
[IVIR:3-1.0] If the length is 0, no value is inside the range.	No Tests available	0

Figure 5.42: Coverage Report of is_value_in_range

The first requirement has one test case linked which was successfully executed, the second requirement has also one test case linked, but this one failed. The third requirement has still no test case assigned.

Test Coverage States

Test coverage state	Meaning
No tests available	No test linked to this requirement.
One or more tests planned	At least one test is linked to this requirement, but none of them have been executed.
Some tests failed	Some of the tests linked to this requirement have been executed and there were failed results.
Some tests passed	Some of the tests linked to this requirement have been executed and all of them yield passed results.
All tests executed, some failed	All tests linked to this requirement have been executed but some of them have failed.
All Tests passed	All tests linked to this requirement have been executed and all yield passed results.

Table 5.2: Meaning of the Test Coverage States

Now you successfully finished the exercise is_value_in_range.

5.1.13 Reusing a test object with a changed interface

Using IDA

If the interface of the test object changes, TESSY will indicate the changes with specific test readiness states. With the Interface Data Assigner (IDA) you can assign the elements of a changed (new) interface to the elements of the old one.

In this section we will change the interface of the test object by editing the C-source and exercise a reuse operation within the IDA.

Important: Make sure to keep the original C-source file “is_val_in_range.c” and edit a copy. Do not change the original file in folder “C:\Program Files\Razorcat\TESSY_4.x\examples\IsValueInRange”!

5.1.13.1 Changing the interface of the test object

The target of this section is to show you the three different test readiness states “changed”, “deleted” and “new”.

Therefore we will first change a test object and add two new test objects called “delete” and “new”. In a second step we will remove the “delete” object so it appears as deleted. The names are chosen to illustrate the test readiness states.

→ Switch back to the Overview perspective.

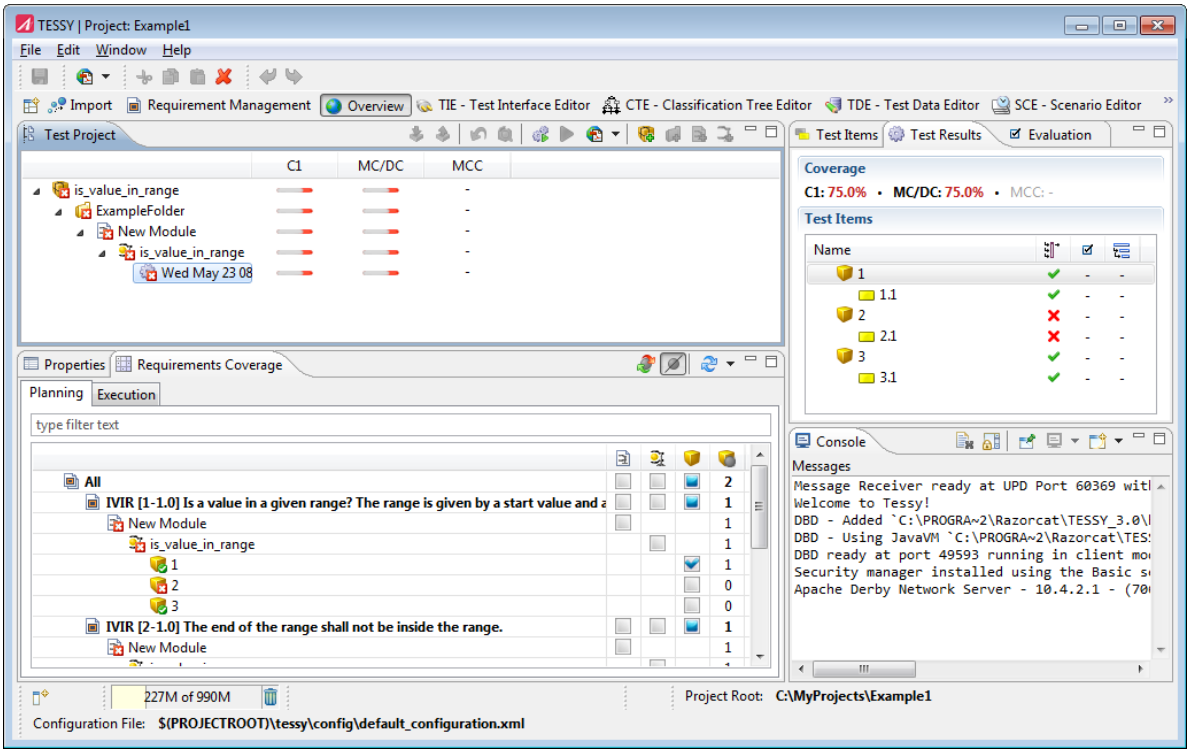


Figure 5.43: Overview perspective after test run (with requirements)

→ Select the module and “Edit Source” from the context menu (see figure 5.44).

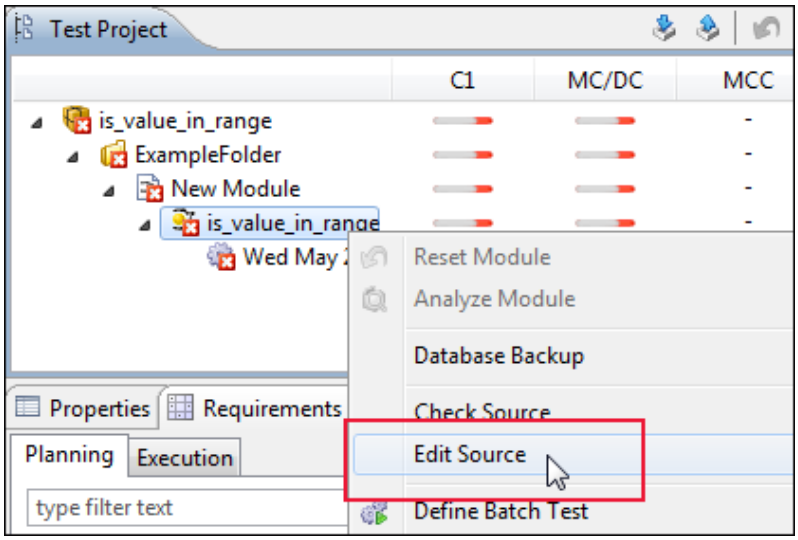


Figure 5.44: Use the context menu to edit a source

The C-source is opened.

→ Select the line "*result is_value_in_range (struct range r1; value v1)*" (see figure 5.45).

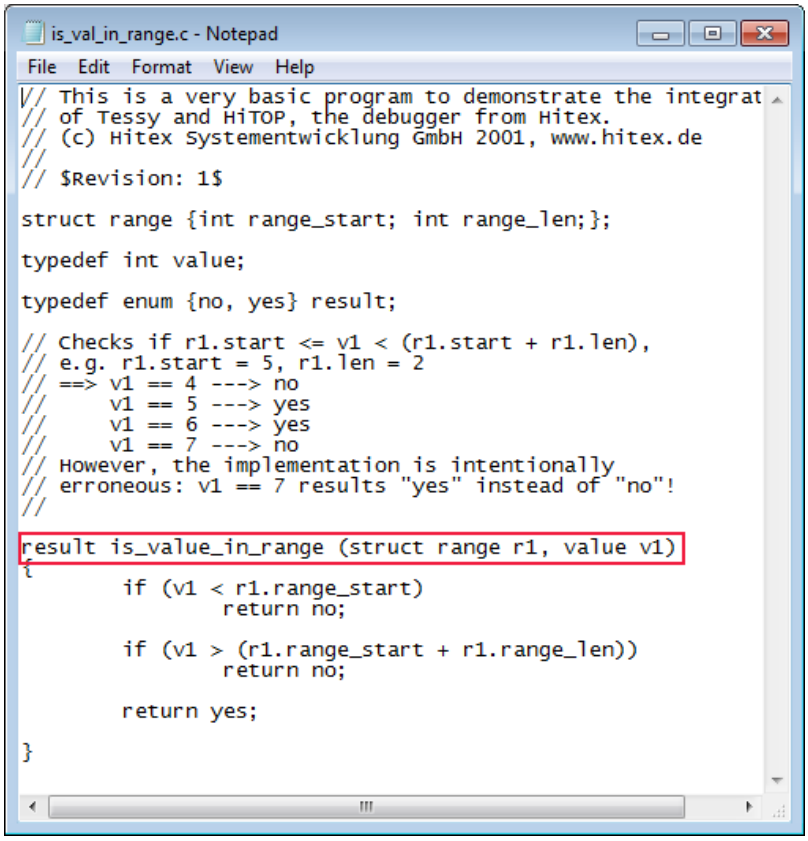


Figure 5.45: Editing the C-source file is_val_in_range.c

→ Edit the line as shown in figure 5.46

```
//      v1 == 6 ---> yes
//      v1 == 7 ---> no
// However, the implementation is intentionally
// erroneous: v1 == 7 results "yes" instead of "no"!
//

value v1;

result is_value_in_range (struct range r1)
{
        if (v1 < r1.range_start)
                return no;

        if (v1 > (r1.range_start + r1.range_len))
                return no;

        return yes;
```

Figure 5.46: Changed C-source file of is_value_in_range

→ Now add a “delete” object and a “new” object as shown in figure 5.47

Changing the C-source

```
//
value v1;
result is_value_in_range (struct range r1)
{
        if (v1 < r1.range_start)
                return no;

        if (v1 > (r1.range_start + r1.range_len))
                return no;

        return yes;

}

void deleted(int p)
{}

void new()
{}
```

Figure 5.47: Adding a “delete” and “new” object

→ Save the changes with “File” > “Save” and close the file.

→ Click on to analyze the module.
In the Test Project view you can see now three test objects with different test readiness states (see figure 5.48):

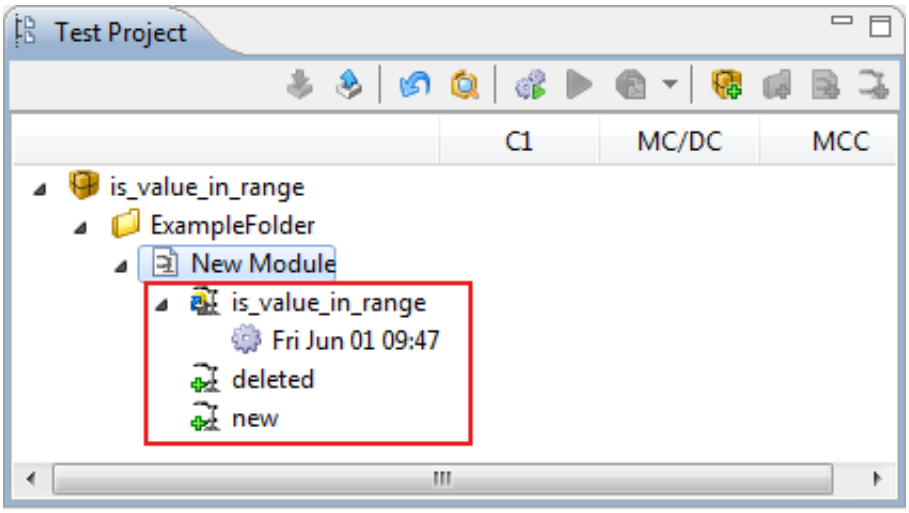


Figure 5.48: Changed and new test objects of is_value_in_range

The test object is_value_in_range has **changed**. You see the test object, but there is no operation possible. You have to start a reuse operation.

The test objects “deleted” and “new” are **newly** available since the last interface analysis. You have to add test cases, test steps and enter data for a test.

Deleted test objects that did not contain any test cases and test steps are not displayed anymore because they are considered as not important. If you want to display a deleted test object, you have to add at least one test case and one test step!

Before deleting the test object “deleted”, we will have to add some test cases with test steps:

→ Switch to the Test Item view and add a test case and a test step.
→ Switch to the Overview perspective and to the Test Project view.
→ Select the module and “Edit Source” from the context menu.
→ Remove the test object “deleted” as shown in figure 5.49.

```
//
value v1;
result is_value_in_range (struct range r1)
{
        if (v1 < r1.range_start)
                return no;

        if (v1 > (r1.range_start + r1.range_len))
                return no;

        return yes;

}

void deleted(int p)
{}

void new()
{}
```

Figure 5.49: Remove the code for test object “deleted”.

→ Save the changes with “File” > “Save” and close the file.

→ Click on to analyze the module.

In the Test Project view you can see now three test objects with three different test readiness states (see figure 5.50):

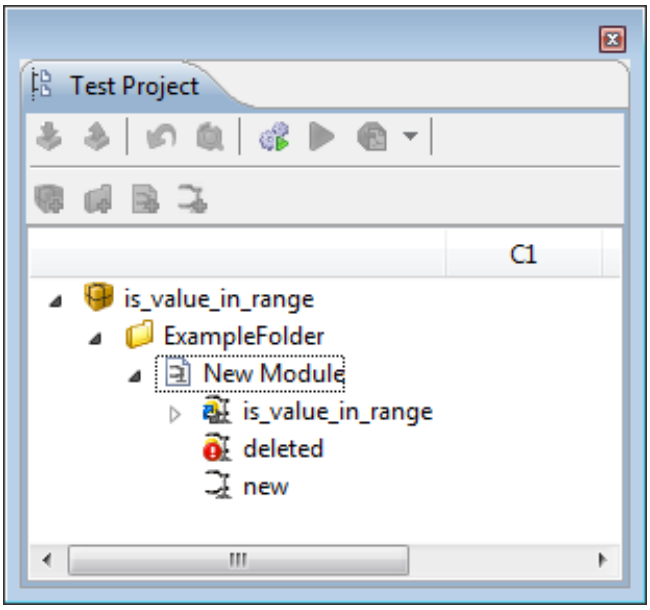


Figure 5.50: Changed and new test objects of is_value_in_range

The test object is_value_in_range is still displayed as **changed** since there was no reuse operation yet.

The test object “deleted” has been removed. You still see the object, but there is no operation possible.

The test object “new” is not shown anymore as “newly available”, because the last interface analysis already detected the object as new.

Important: Note that removed and changed test objects require a reuse operation before you can further operate on them!

5.1.13.2 Assigning the changed interface of the test object

Warning: If you do not assign the interface object, you will loose the test data entered for parameter v1 and the global variable v1 will have no values after the reuse operation!

→ Switch to IDA perspective.
→ Double-click the test object “is_value_in_range” in the Test Project view to assign its interface.

Please notice (see figure 5.51):

- On the right side within the IDA perspective you see the Compare view with the test object is_value_in_range.
- Within the Compare view you can see the old interface of our test object is_value_in_range and the new one. The red exclamation mark within the new interface indicates the need to assign this interface object before starting the reuse.
- The title of the view shows the old name versus the newly assigned name. In our case the names are the same since only the interface did change.

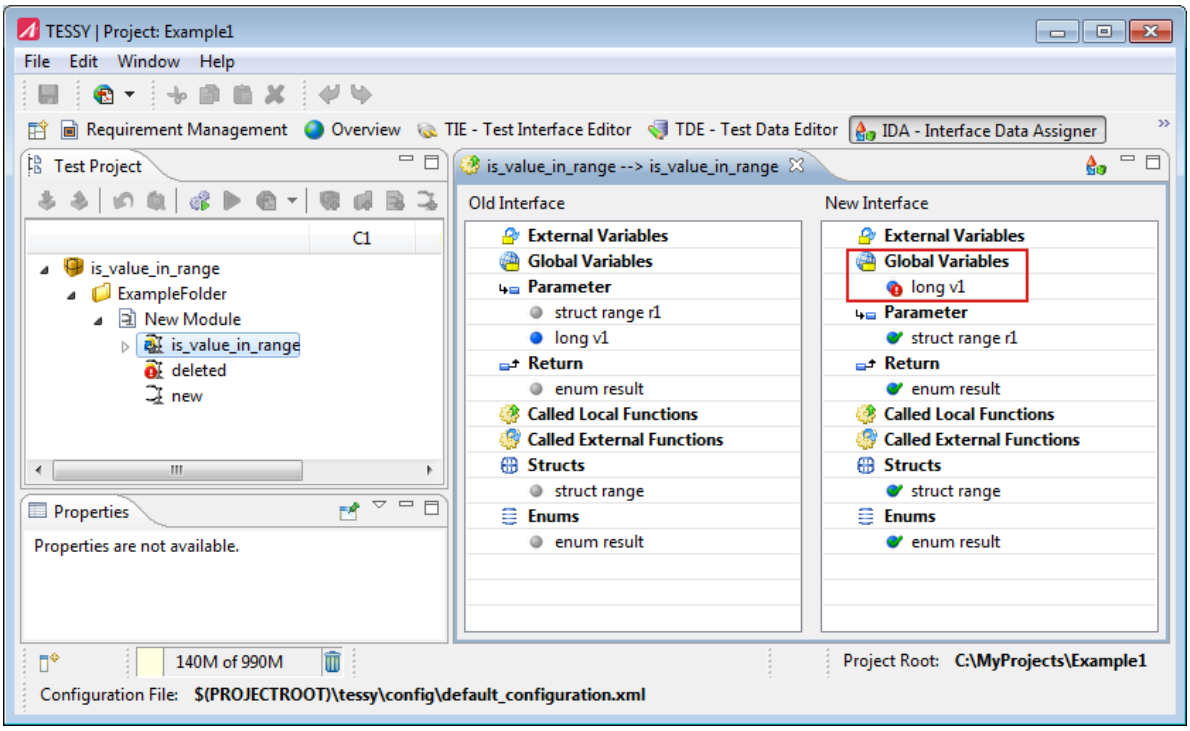


Figure 5.51: Changed, deleted and new test object of is_value_in_range

→ Assign the interface object “long v1” either by using the context menu or just drag and drop from the left side (see figure 5.52).

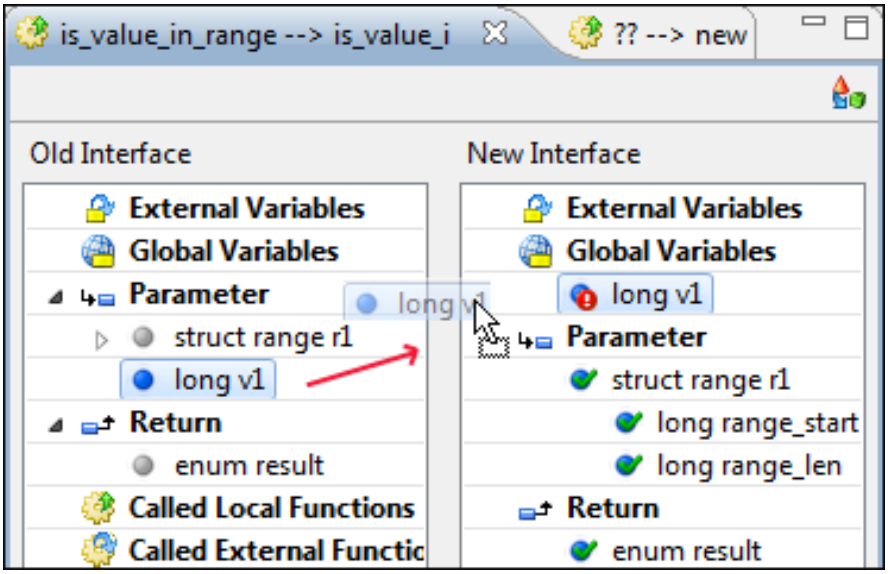


Figure 5.52: Use drag and drop in IDA

The red exclamation mark turns to a green check ✔.

→ Commit the assignments by clicking on (Commit) in the menu bar of the Compare view.
The data of all test cases and test steps will be copied from the old interface to the current test object interface.
The test object changes to yellow to indicate that all test cases are ready to be executed again.

Please notice the following habits:

- Removed and changed test objects require a reuse operation before you can further operate on them.
- Those test objects that remained unchanged will automatically be reused, e.g. they will be ready to use without further activities required.
- Removed test objects will only be displayed as “removed”, if they did contain any test cases and test steps.

5.2 Quickstart 2: The Classification Tree Editor (CTE)

To understand the handling and create a simple classification tree we consider some aspects from the Quickstart 1: Unit test exercise is_value_in_range.

3.2 The Classification Tree Method (CTM)

Important: In this chapter we will continue with the quickstart example “is_value_in_range”. If you have not done the exercise, proceed with the Quickstart 1: Unit test exercise is_value_in_range up to section Designing test cases.

This manual provides general information about the The Classification Tree Method (CTM) in the chapter ’Basic knowledge’ as well as a detailed description of the CTE in section 6.8 CTE: Designing the test cases within ’Working with TESSY’.

Switch to the CTE perspective to get to the automatically generated tree of the quickstart example “is_value_in_range”.

6.8 CTE: Designing the test cases

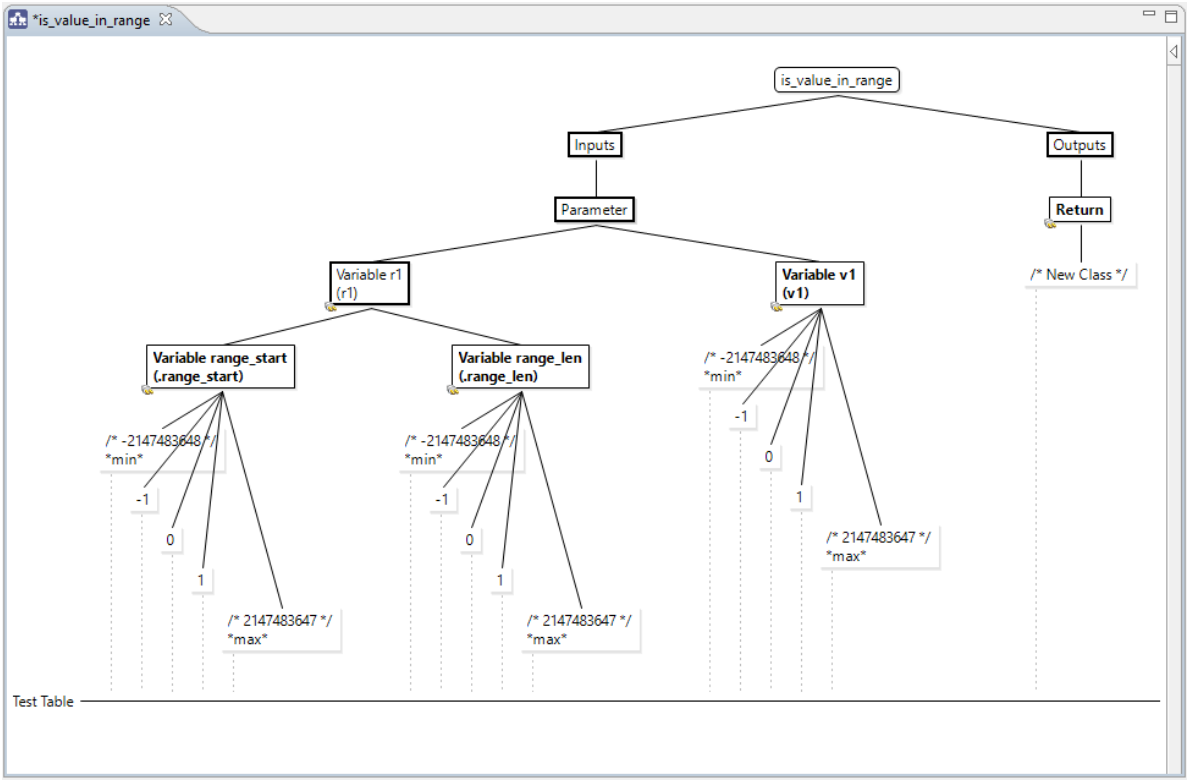


Figure 5.53: Automatically generated tree with the root “is_value_in_range” in the CTE perspective

This tree within the CTE perspective (see figure 5.53) is generated based on the relevant interface elements.

Important: Please note that Interface elements can be set to IRRELEVANT in the TIE perspective and these elements will be omitted in the generated tree.

5.2.1 The CTE tree elements

The tree elements of the automatically generated tree follow a basic structure. First of all, it is categorized into ”Input“ and ”Output“ (see figure 5.54).

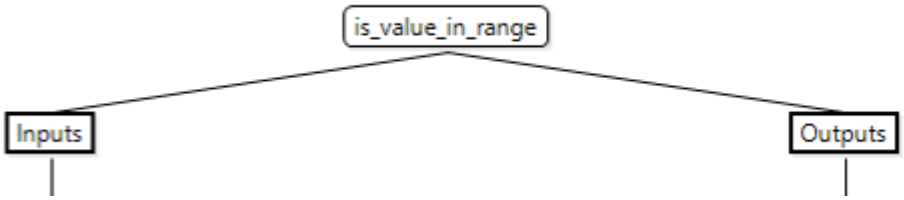


Figure 5.54: Interface elements categorized into ”Inputs“ and ”Outputs“

On the next level the interface elements are further subdivided into parameters, globals etc. With the following levels the composite types, i.e. structures, unions or pointers, are broken down into atomic types such as integers, floating point numbers, enumerations, etc.

More information about the automatically generated tree can be found in subsection 6.8.6.5 Automated tree generation based on function interface.

Within the “Inputs” subtree:
The leaves of this part of the tree are always child elements of atomic types and those are common values with specific meaning, like zero, negative value next to zero, positive value next to zero, minimal value of datatype and maximal value of the datatype (see figure 5.55).

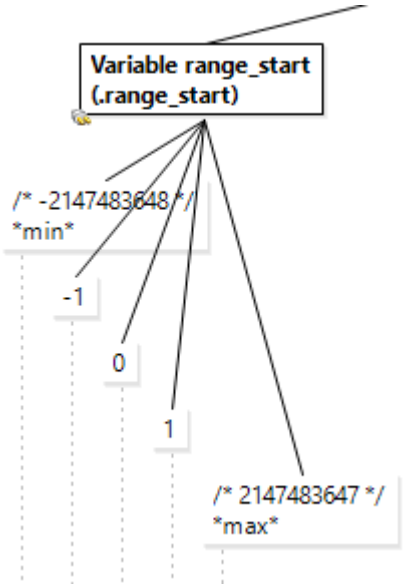


Figure 5.55: Child elements of an atomic type on the inputs side of the subtree

Within the "Outputs" subtree:
Values of the atomic types will not be predicted on this side of the tree but there will be a simple place holder (see figure 5.56).

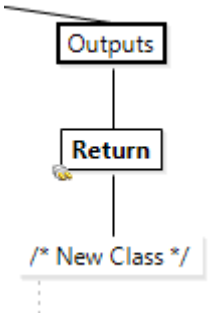


Figure 5.56: The outputs subtree

Some interface elements are marked as attached interface elements with a small TIE icon (see figure 6.188) as they had been automatically attached.

For more information about attaching interface elements to a CTE node go to subsection 6.8.7.4 Attach selected interface element to CTE node.

5.2.2 Working with the CTE

The basic idea of the Classification Tree Method is to provide a systematical approach to create test case definitions based on the functional specification of the function or system to be tested. The TESSY included Classification Tree Editor CTE assists you in creating low redundant and error sensitive test cases.

For more information about working with classes as well as with classifications and test cases please refer to subsection 6.8.6.4 Creating classifications, classes and test cases.

6.8 CTE: Designing the test cases

After preparing a test in the TIE, well designed test case specifications need to be created. A test case is formed through the combination of classes from different classifications. For each test case exactly one class of each classification is considered.

The combined classes must be logical compatible, otherwise the test case is not executable. You should choose and combine as many test cases as needed to cover all aspects that should be tested.

Within the CTE tree area it is possible to move the classifications and other elements with drag and drop: Just left click the element, hold the mouse button and move it to the desired place. You may also select a group of elements and move them the same way.

The tree layout will be arranged automatically by clicking in the tool bar.

5.2.3 Entering test data

You may use the CTE to create or edit test cases manually. Making those kind of changes is also possible within the TDE but in both perspectives values are always entered in the Test Data view.

Entering values

It is possible to assign test data using the tree nodes of the classification tree. For each tree node values to variables can be assigned (see figure 5.57).

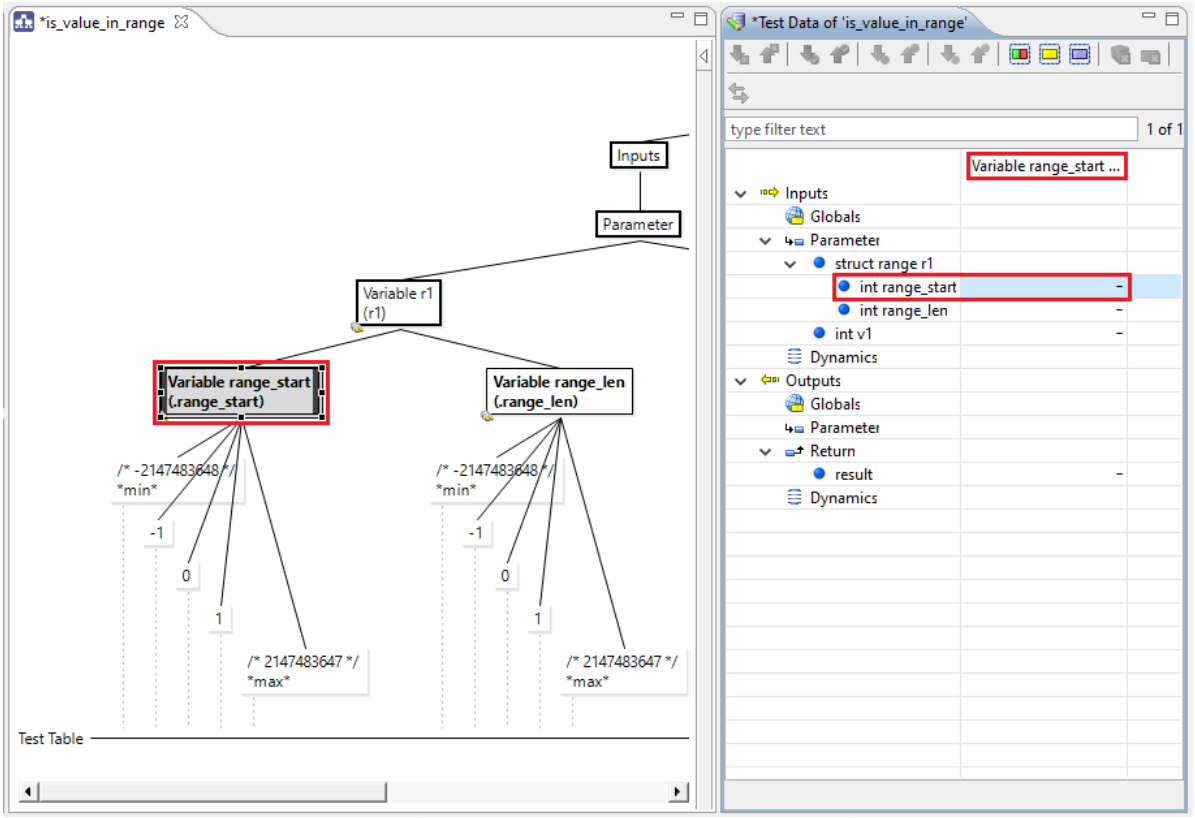


Figure 5.57: CTE tree area and Test Data view

Important: To enter the data it might be necessary to enlarge the window or to change the width of the first column containing the interface elements.

Please note that some of the operations and overviews are only available within the TDE perspective.

More information about entering test data within the CTE perspective can be found in subsection 6.8.7.1 Assigning test data to the CTE.

As mentioned above some classifications are automatically attached to interface elements. This attachments allow TESSY to automatically generated test data based on the names of the respective classes.

TESSY will generate the test data every time the CTE document is saved. Do so and select the classes in the tree to examine the generated values.

Important: Values automatically generated by TESSY can be edited but will be overwritten when saving the CTE document. Other interface elements can be edited and will be kept, this is reasonable for example for interface elements which are also output variables.

Always keep in mind that attaching arbitrary test data to nodes can lead to very confusing situations. So please be cautious and stick to conventions made by your team or company or those proposed by Razorcat.

Notice the following habits:

- The name of the selected item will be displayed as column header in the Test Data view.
- When selecting a tree item, you will see the test data entered for this item within the Test Data view.
- All CTE tree items with any assigned test data will be shown with a yellow square .
- When selecting an interface element within the Test Data view, the respective CTE tree items with test data assigned for this interface element will be shown with a blue square .

5.2.4 Creating test cases

The CTE method offers a graphical representation of the recursive partitioning of classifications and classes in shape of a classification tree. Classifications are drawn as named rectangles, respective classes are arranged below. To specify the test cases the classification tree is used as the head of a combination table. In this Test Table the classes which are to be combined can be marked.

A test case is formed through the combination of classes from different classifications. For each test case exactly one class of each classification is considered. In this example all classes get automatically generated test data.

In this way it is necessary to select one class from every classification in the input subtree to compose an executable test case, i.e. a test case with a complete set of input values.

Important: Test units without dynamic data types are not generated with a complete set of test data, the pointer's dynamic object must be initialized and set by the user. Afterwards TESSY will also generate values in the dynamic object.

More information about classifications, classes and test cases are provided in subsection Creating classifications, classes and test cases.

Important: For sake of simplicity the functions of the CTE Test Table within this quickstart exercise please delete all *min* and *max* elements from the automatically generated tree. You should also modify the output side of the tree.

In general, the automatically generated tree always needs a review and adjustments from a tester. To demonstrate this procedure we delete classes and add more in this example.

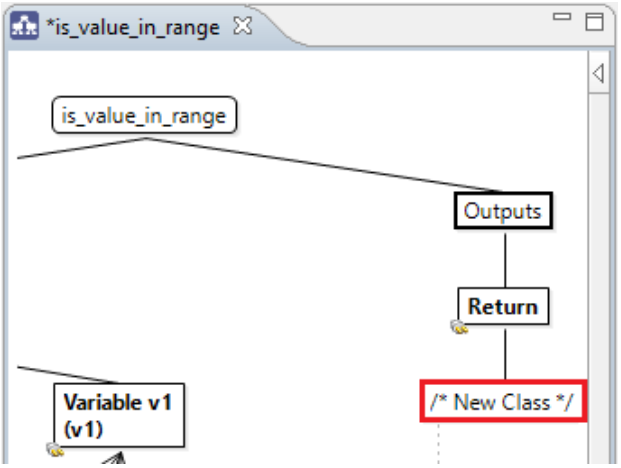


Figure 5.58: Modify class elements

→ Double-click on the red marked class in figure 5.58 or press F2 after selecting the figure.
This will allow you to edit the class name.
→ Enter "yes" and press "Enter" on your keyboard.
→ Then Right-click on "Return" to open the respective context menu.
→ Choose "Add Class" to create a new class.
→ Name it "no" (like explained above).

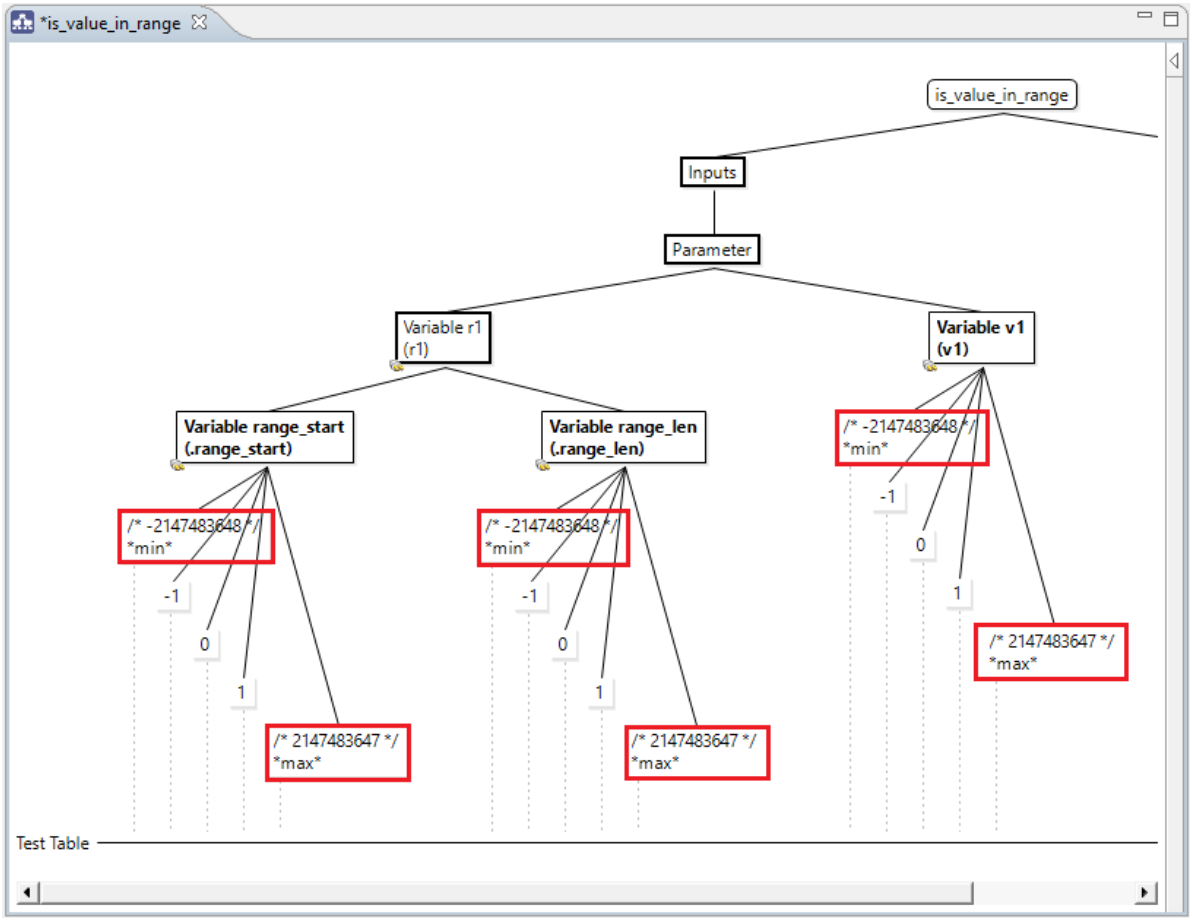


Figure 5.59: Deleting elements

→ Click on one of the red marked elements in figure 5.59.
→ Press “Delete” on your keyboard.
→ Do so for all the red marked elements.

Important: The classes names are mapped to values of the attached enumeration type. In this way it is necessary to use *exactly* the values “yes” and “no”. Comments after a pair of forward slashes “//” is allowed.

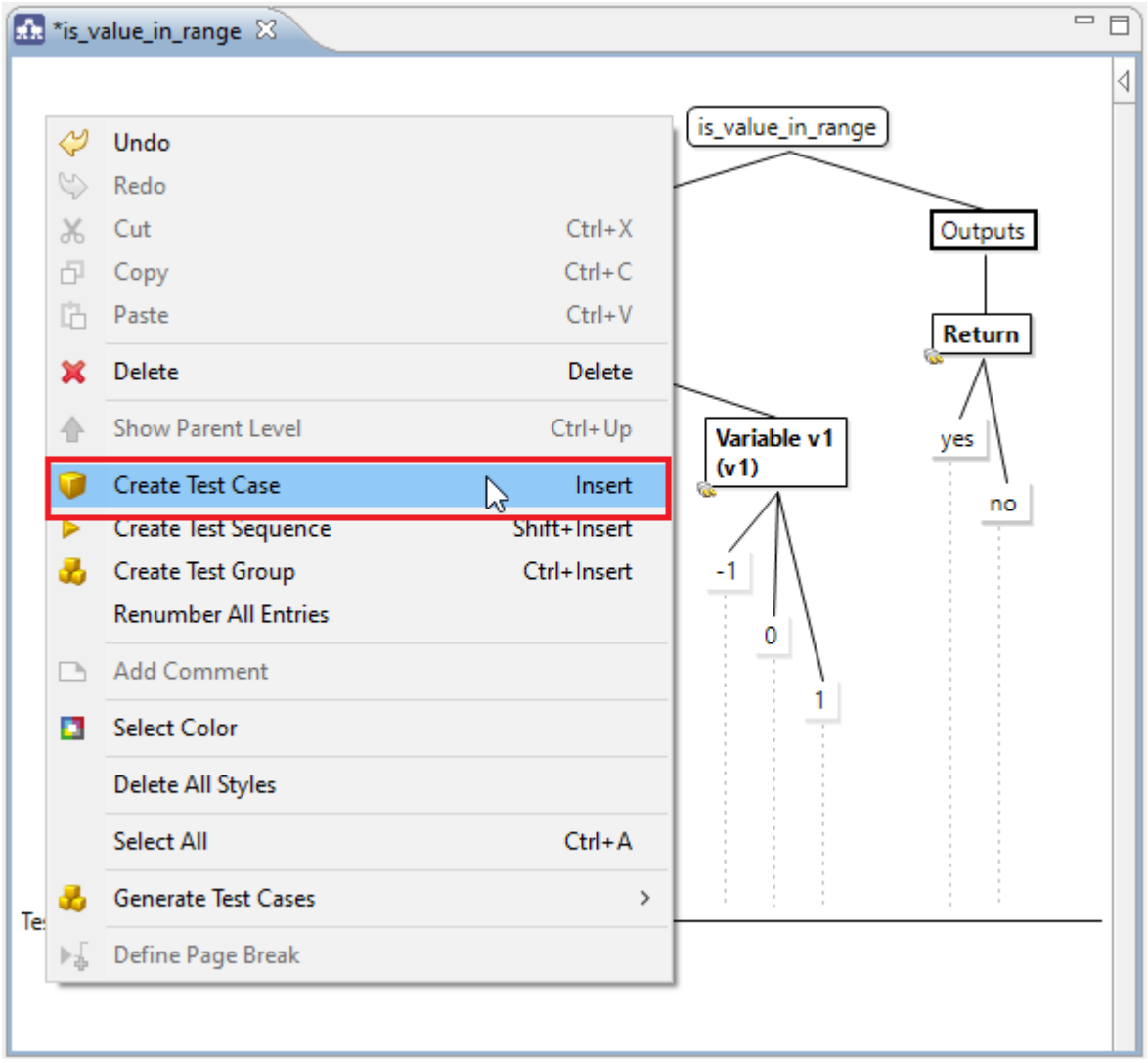


Figure 5.60: Creating test cases in the CTE

→ Right-click in the Test Table at the bottom of the CTE.
→ Select “Create Test Case” within the context menu, see figure 5.60, or press Insert on your keyboard.
→ Create 9 test cases in this way.

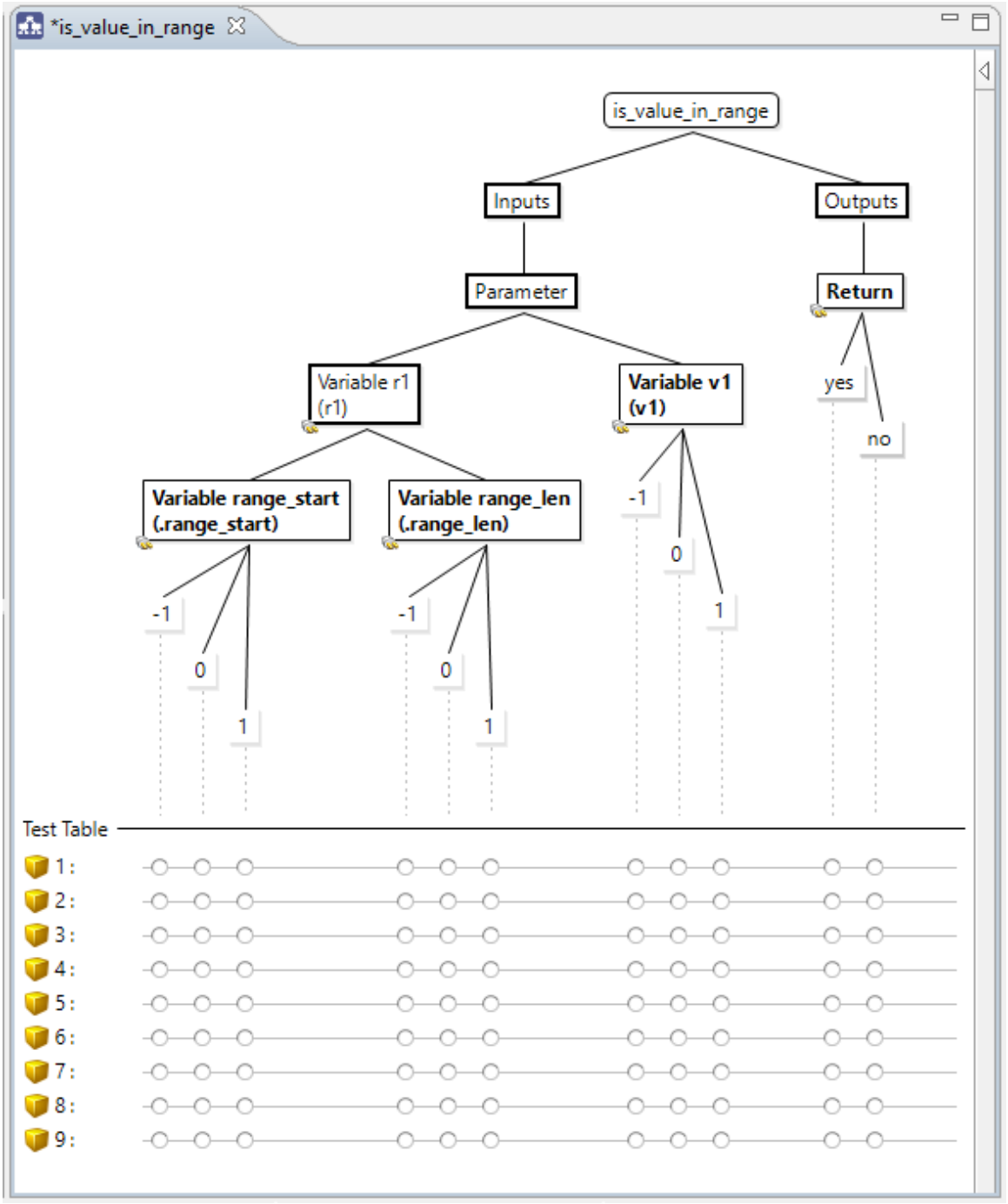


Figure 5.61: Automatically generated tree with 9 Test Cases

Setting marks

Test cases are defined by setting marks in the Test Table:

→ Select the first test case in the test item list.

→ Move the mouse over the line of the first test case.

→ Click on the circles that connect the first test case with the two positive classes.

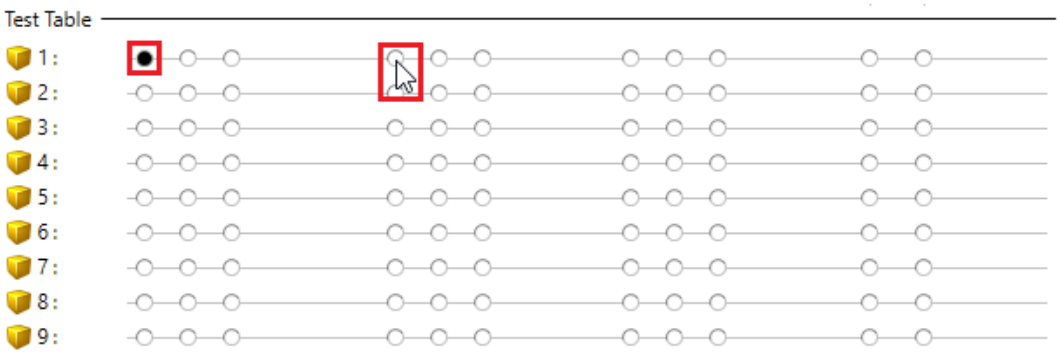


Figure 5.62: Defining test cases in the combination table of CTE

Important: Marks can also be set with the keyboard:

- Navigate with the cursor keys.
- Set or remove marks with the space key.

→ Create more marks within the combination table for the other test cases to cover completely all classes of the classifications “range_start” and “range_length” with your test cases.
For example the CTE could look like in figure 5.63.

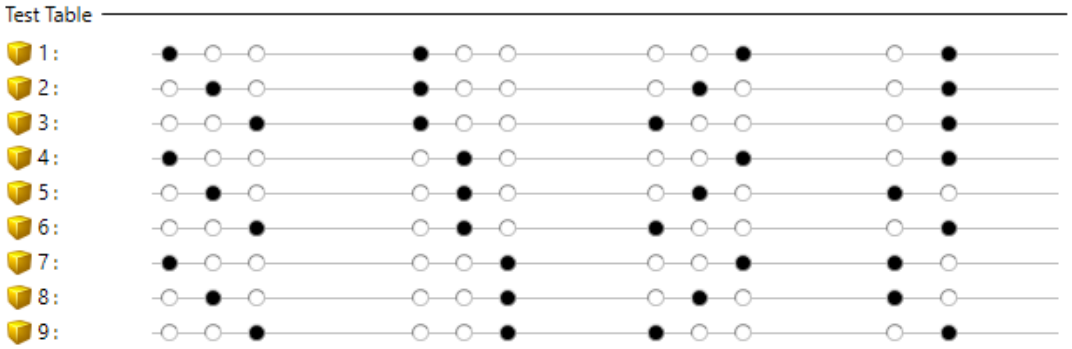


Figure 5.63: Completed table with all test cases for example “is_value_in_range”

6.8.6.4 Creating classifications, classes and test cases

Important: As you can see there are various possibilities of combining the classes within the test cases.

In this case it was decided to choose all possible combinations for the classifications. In real testing you would need to select the most interesting combinations only in order to get a reasonable number of test cases.

→ Select the test cases one after another and review the test data resulting from your mark settings being displayed within the Test Data view (see figure 5.64).

In figure 5.64 the test data for test case 7 is displayed within the Test Data view. The test data is read-only because it is defined by the marks set within the combination table.

→ Click on to save the classification tree.

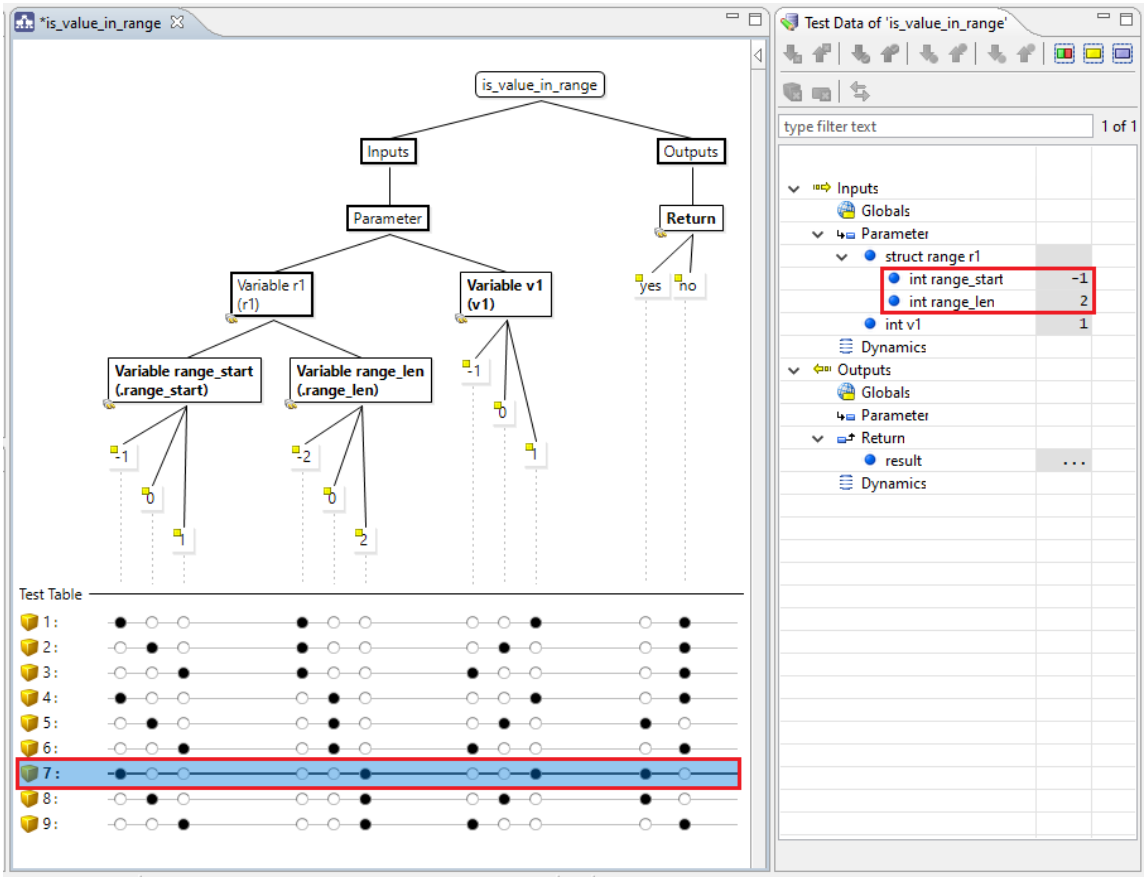


Figure 5.64: Test data is displayed when selecting a test case in the combination table

→ Switch to the TDE perspective.

Test data in the TDE

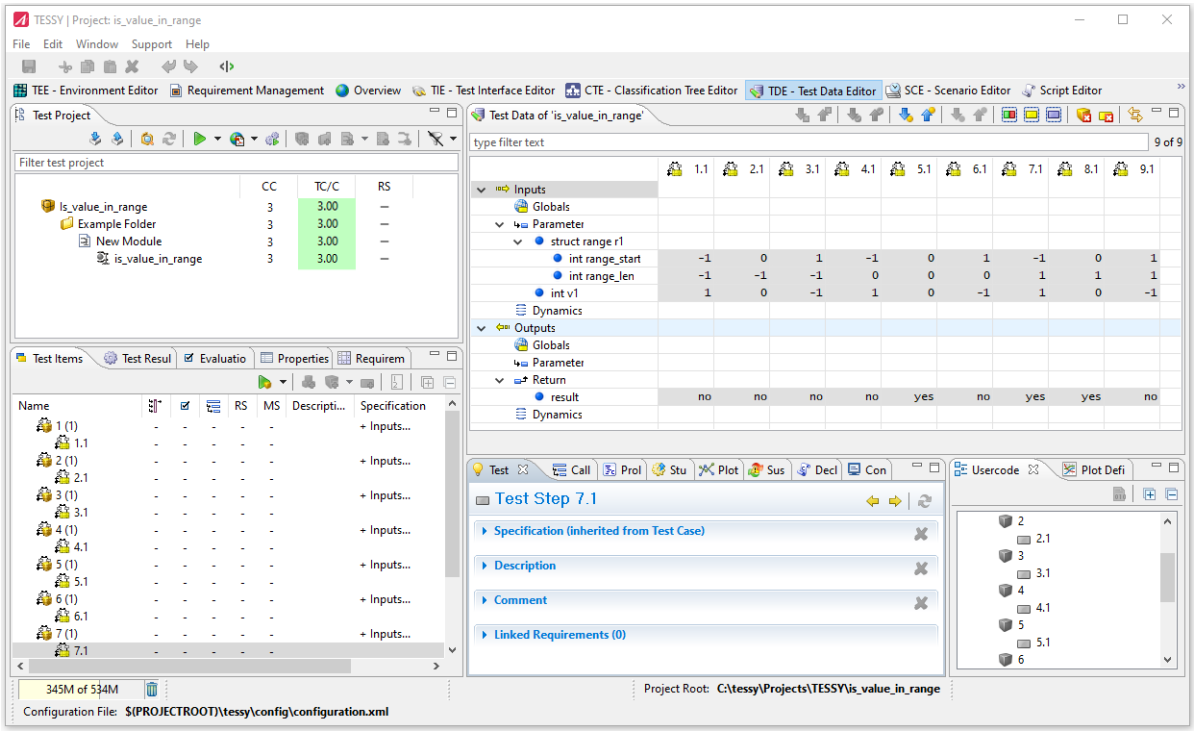


Figure 5.65: Test data displayed within TDE

You will see the test cases updated with the test data values entered within the CTE perspective.

Please notice the following habits:

- Test items with data stemming from the CTE perspective are marked with special status indicators: (test case) and (test step).
- The indicators will appear light gray when there is no data entered, dark gray when there is some data entered and yellow when the entered data is completed.
- Data stemming from the CTE are read-only. If you want to change them, switch back to the CTE perspective and do your changes there.

5.3 Quickstart 3: Component test exercise interior_light

To understand the handling in this chapter we assume you have worked with TESSY in unit testing. If not, please proceed with the Quickstart 1: Unit test exercise is_value_in_range first and come back to this chapter afterwards.

If you have already worked with former versions of TESSY, this chapter may help you to learn the differences in handling with TESSY 3.x!

Integration testing of functions = component testing

Isolated testing or testing of isolated functions is unit testing.

Testing a component of interacting functions, consisting e.g. of push() and pop() calls is a very simple example for **Integration testing**: Functions that do not necessarily have a calling relation but work together, e.g. operating on common data, with the objective to achieve a common goal.
At least one of the functions combined in a component must be callable from the outside of the component to stimulate the functionality of the component.
Normally several functions are callable from external. We call these functions "component functions." **A test for a component is no longer a single function call but a sequence of calls to the component functions.**

The calls to the component functions stimulate the component. Like testing of a single function, a test case for a component also comprises of input and output data. A component may have internal functions that can not be called from the outside of the component, but only from functions inside the component.
Relevant for the result of a component test is the sequence of the calls from within the component to (callable) functions in other components. This is with respect to the number of the calls, the order of the calls, and the parameters passed by the calls to other components.
Obviously, the functionality of the functions in a component and the interfaces between them are tested by component testing, at least for the most part. Hence, component testing can be considered well as integration testing for the functions in the component.

You already know the basic functionality of a unit test. We will now follow a simple source code example to show how to exercise component testing with TESSY.

Example interior_light

The interior light of a car shall be controlled by two inputs: The door and the ignition of the car.

The functional specification comprises of three simple requirements:

- If the door is closed, the interior light shall go on.
- The interior light shall go off after 5 seconds at the latest.
- If the ignition is switched on, the interior light shall go off immediately.

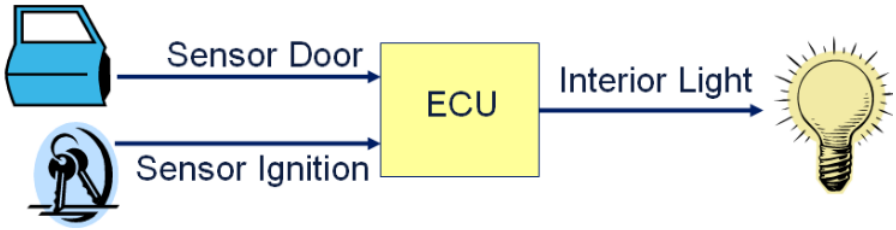


Figure 5.67: Example interior_light; ECU = Electronic Control Unit

The specification above is obviously not complete. Especially the initial state of door, ignition, and light is not given. It is not specified what shall happen e.g. if the ignition is switched off after it was switched on, etc. But this simple specification is sufficient to demonstrate temporal component testing with TESSY.

For simplicity, we assume that the initial state shall be

- Door = open
- Ignition = off
- Light = off

5.3.1 Creating the test project

→ Create a test project “interior_light” (if you need any help, consult section Quick-start 1: Unit test exercise is_value_in_range).

→ Copy the C-source file “`interior_light.c`” which is stored under “`C:\Program Files\Razorcat\TESSY_4.x\Examples`”.

→ Paste it into your project root and add it to the module.

→ Open the module.
The C-source code will now be analyzed, afterwards the test project view displays the functions of the source (see figure 5.68).

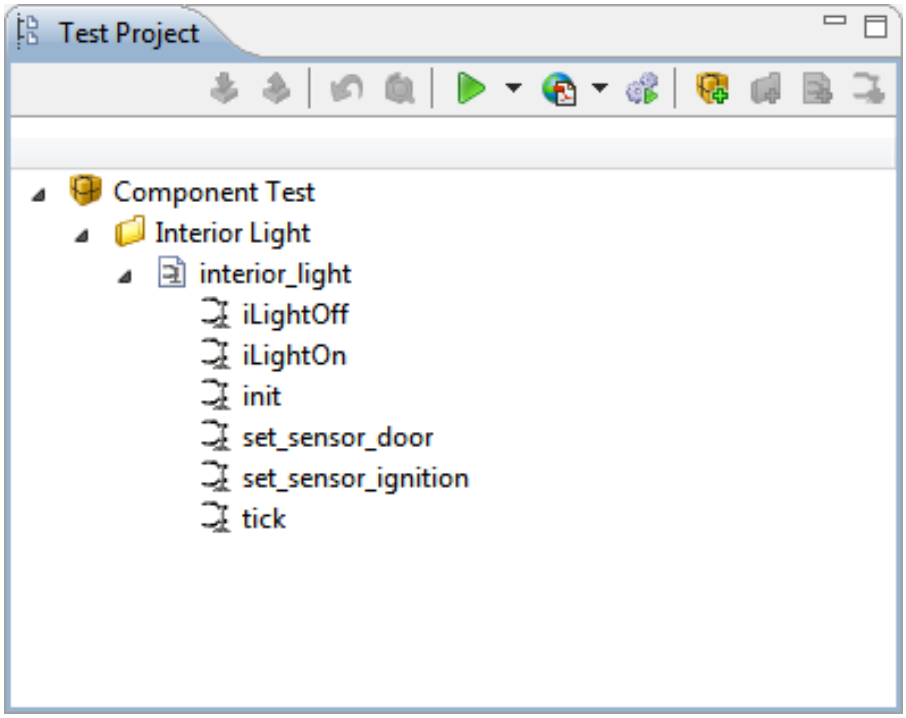


Figure 5.68: Test Project view with new project interior_light

→ In the Properties view switch the Kind of Test to “Component” (see figure 5.69).

Important: By default, “Unit” is selected. This is the point where unit testing and component testing begin to part!

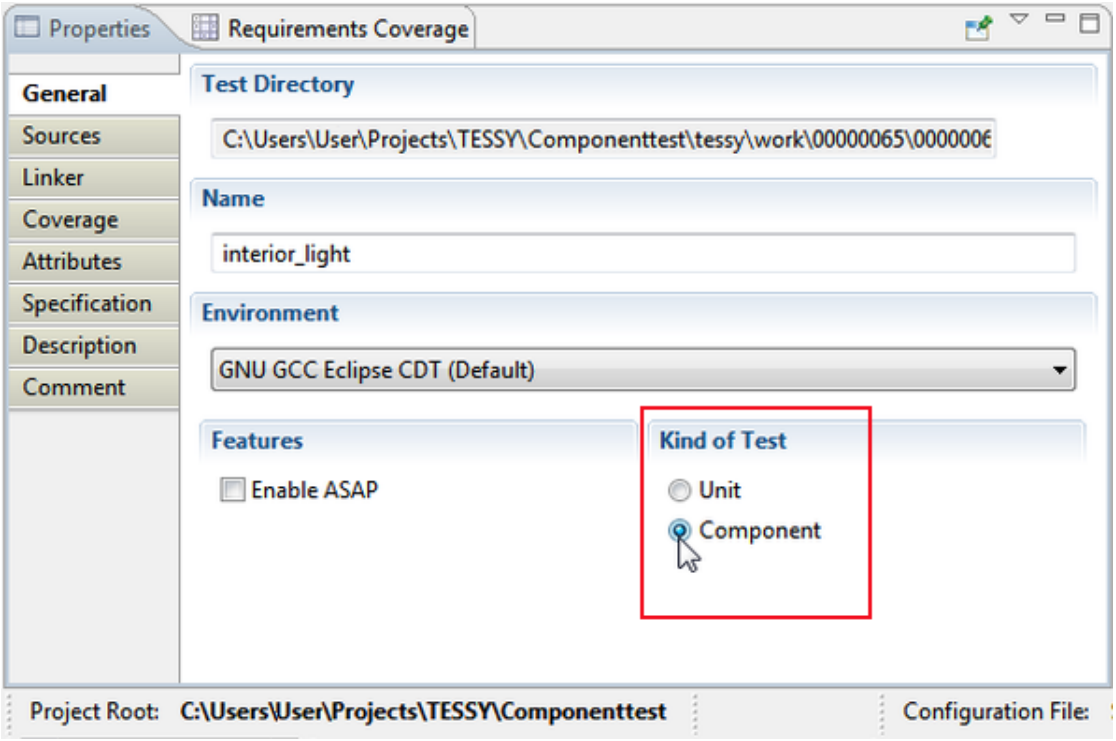


Figure 5.69: Selecting “Component” in the module properties

Now the only test object displayed has the default name “Scenarios” (see figure 5.70).

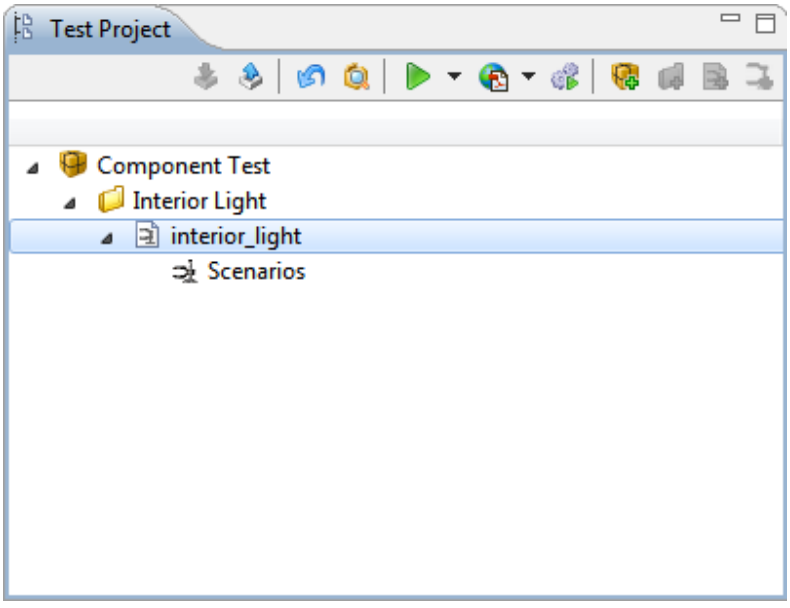


Figure 5.70: Scenario of a component test

This is different to unit testing with TESSY, where the names of the possible test objects in interior_light.c (i.e. the functions) would be listed instead.

5.3.2 The heartbeat function

You can open the C-source file with a rightclick onto the module and choosing “Edit Source” from the context menu. The C++-Perspective will open and the file will be displayed (see figure 5.71).

interior_light.c

```
//=====================================================================

typedef enum
{
   closed, open
} co_states;

typedef enum
{
   off, on
} oo_states;

co_states sensor_door, state_door;
oo_states sensor_ignition, state_light;

extern void LightOn(void);
extern void LightOff(void);

void init(void)
{
   sensor_door = open;
   sensor_ignition = off;
   state_door = open;
   state_light = off;
}

void set_sensor_ignition(oo_states i)
{
   sensor_ignition = i;
}

void set_sensor_door(co_states d)
{
   sensor_door = d;
}

static void iLightOn(void)
{
   if (on == state_light)
      return;
   else
   {
      state_light = on;
      LightOn();
   }
}

static void iLightOff(void)
{
   if (off == state_light)
      return;
   else
   {
      state_light = off;
      LightOff();
   }
}

// heartbeat function
/* ****************************************************************/
void tick()
{
   static int Timer = 0;

   if ((state_door == open) && (sensor_door == closed))
         { //Door was closed ---> start timer!
      Timer = 500;   // 5 seconds = 500 cycles of 10 ms
      iLightOn();
   }
   else
   if (on == sensor_ignition)
         {
      iLightOff();
   }

   if (Timer > 0)
      Timer--;

   if (0 == Timer)
      iLightOff();

   // Store current value for next tick
   state_door = sensor_door;
}
```

Figure 5.71: C-source code interior_light

This implementation features a heartbeat function, the tick() function. The implementation assumes that tick() is called every 10 ms. Based on this assumption, for the timer the value “500” is calculated.

Heartbeat Function

To be able to test the temporal behaviour of a component in a simulated environment, a simulated time base needs to be available. This means, a certain function inside the component is called in known equidistant times (e.g. every 10 ms). The calls to this function represent the “heartbeat” of the component. They provide a time reference for the testing of the temporal behaviour of the component.

The heartbeat function is usually called “handler function” or “work task” or simply “tick”.

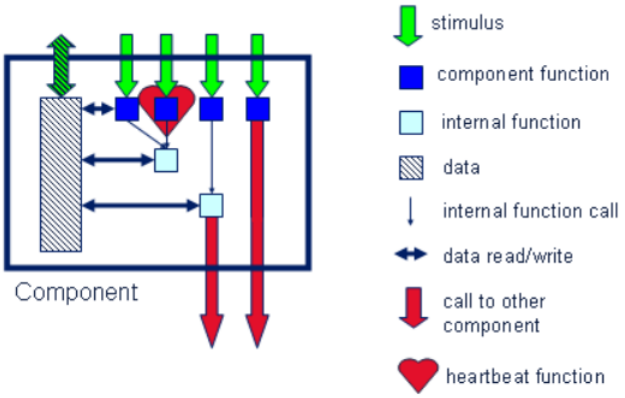


Figure 5.73: If a heartbeat function exists, timely behavior can be tested.

5.3.3 Preparing the test interface

→ Switch to the TIE perspective.

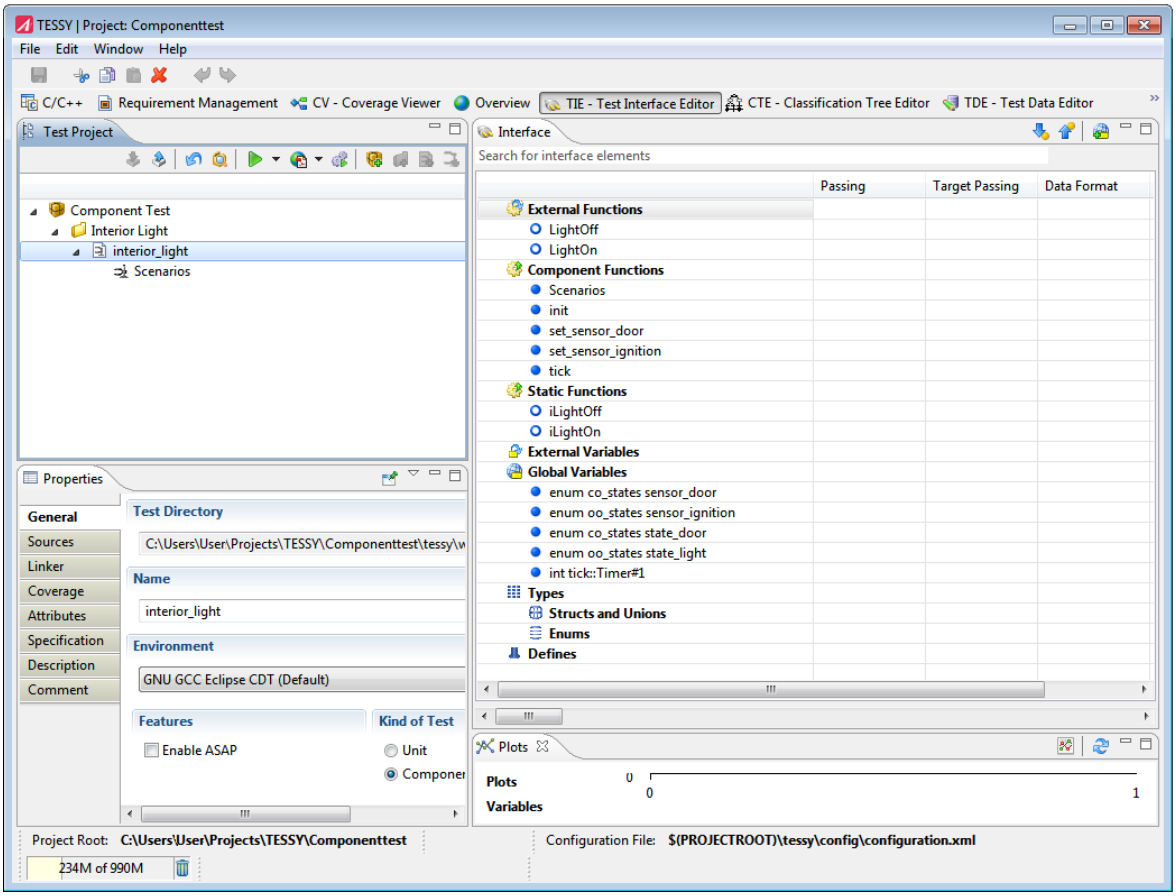


Figure 5.74: The initial interface

In the section “External Functions” of the interface, the two functions LightOff() and LightOn() are listed. These two functions are used (called) by the component “Interior Light”, but these two functions are not implemented in interior_light.c.

The component “Interior Light” expects these two functions to be provided by another component of the application.

However, we want to test the component “Interior Light” without that other component, i.e. isolated from the rest of the application. Therefore, we direct TESSY to provide replacements, i.e. stub functions for these two functions.

To create the stubs:

→ Rightclick the function “LightOff()” and choose “Create Stub” (see figure 5.75).

→ Repeat for the function “LightOn()”.
The dots with the blue border will change to blue filled dots ●.

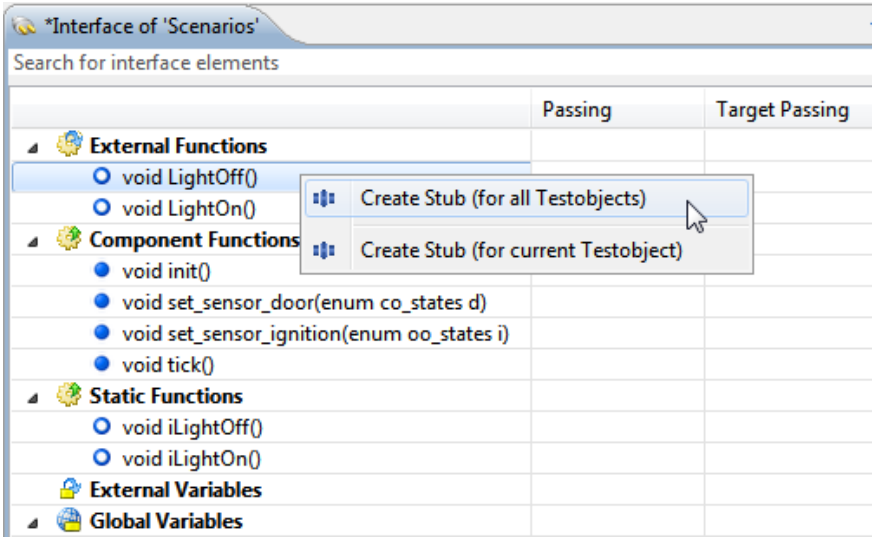


Figure 5.75: Creating the stubs

In section “Component Functions”, all functions of the component “Interior-Light” are listed.

The passing direction of the variable timer is “Irrelevant”, because it is not used by init(). Since we are only interested in the variables “sensor_door”, “sensor_ignition”, and “state_door” as input, we manually set the passing direction for these variables to “In”:

→ Click in a cell and choose “In” for “sensor_door”, “sensor_ignition”, and “state_door” (see figure 5.76).

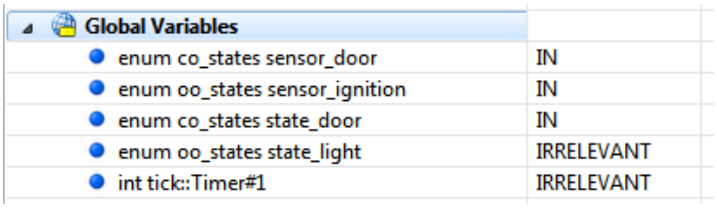


Figure 5.76: The final passing directions of variables used by init()

→ Save the changes by clicking on .

5.3.4 Adding test cases

→ Add two test cases in the Test Items view.

→ Switch to the SCE perspective (or doubleclick a test case, in this case the SCE perspective will open automatically).

→ The bracket of the scenarios in the Test Project view will open to indicate that it contains test cases but no data (see figure 5.77).

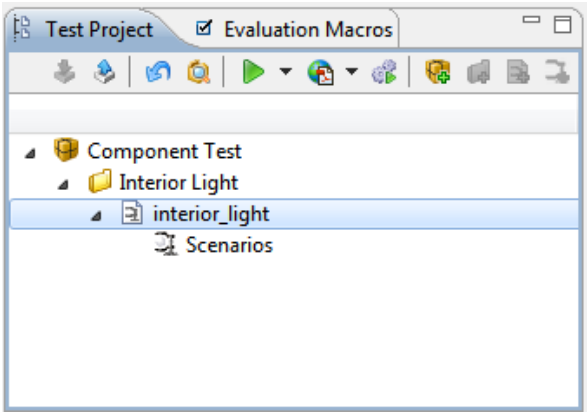


Figure 5.77: Test Project view with a component test

You can add a description to the test cases (see figure 5.78):

1. Door closed: Light off after 5 seconds.
2. Door closed and ignition on: Light off immediately.

These are the two scenarios we are going to test.

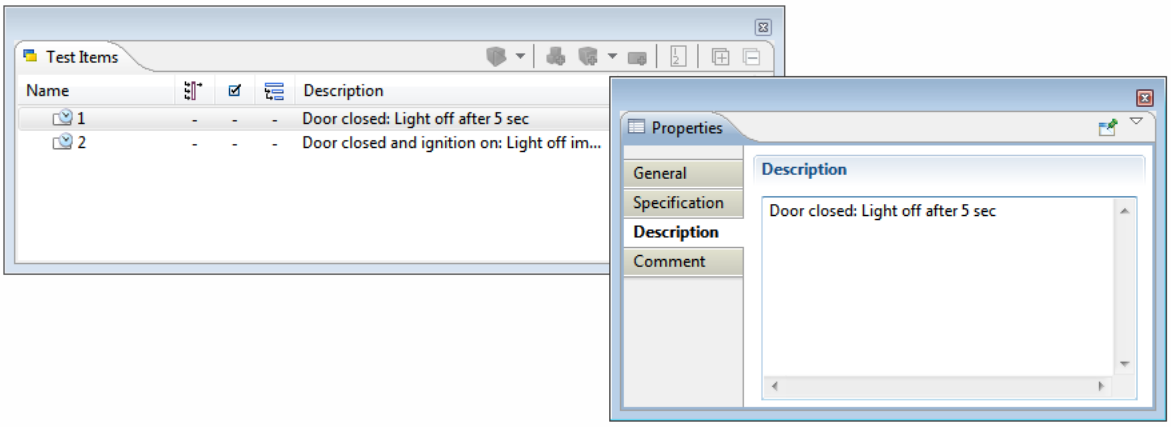


Figure 5.78: Adding a description to test cases

5.3.5 Editing data

The view Test Data of Scenarios is similar to unit testing:

→ Go to the view Test Data of Scenarios within the SCE perspective.

→ Doubleclick the value cells in the INIT column to open the inline editor (see figure 5.79).

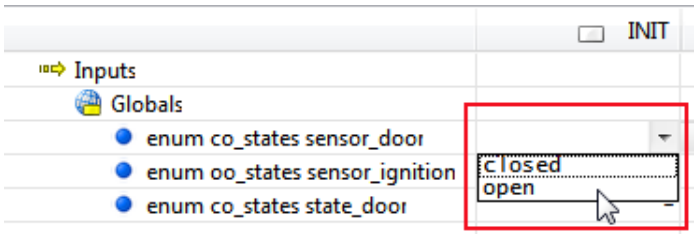


Figure 5.79: Inline editor of the view Test Data of 'Scenarios'

→ Choose the following selections:
sensor_door = open
sensor_ignition = off
state_door = open

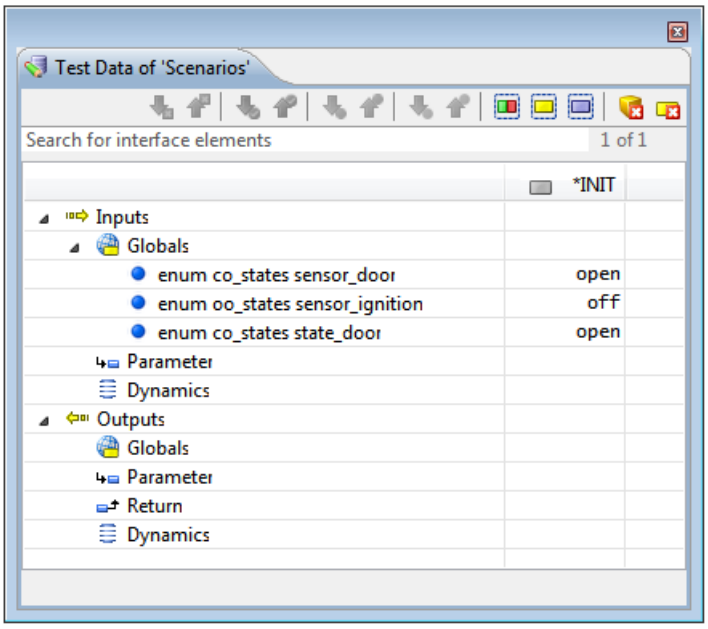


Figure 5.80: View Test Data of 'Scenarios'

→ Use the same data for the second scenario.

→ Save by clicking on .

The view **External Function Calls** displays the two functions LightOn() and LightOff() that the component “Interior-Light” supposes in another component. TESSY provides stub functions for these two functions during component testing.

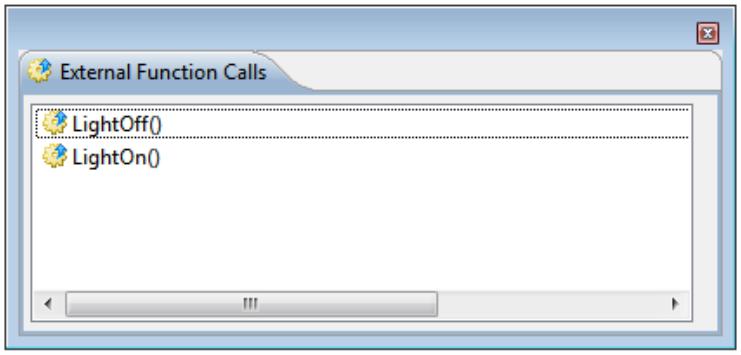


Figure 5.81: View External Function Calls

Renaming Test Cases

Please notice that you can rename the test cases:

→ Rightclick a test case and choose “Rename” from the context menu.
The new name will be displayed in the center of the SCE perspective (see figure 5.82).

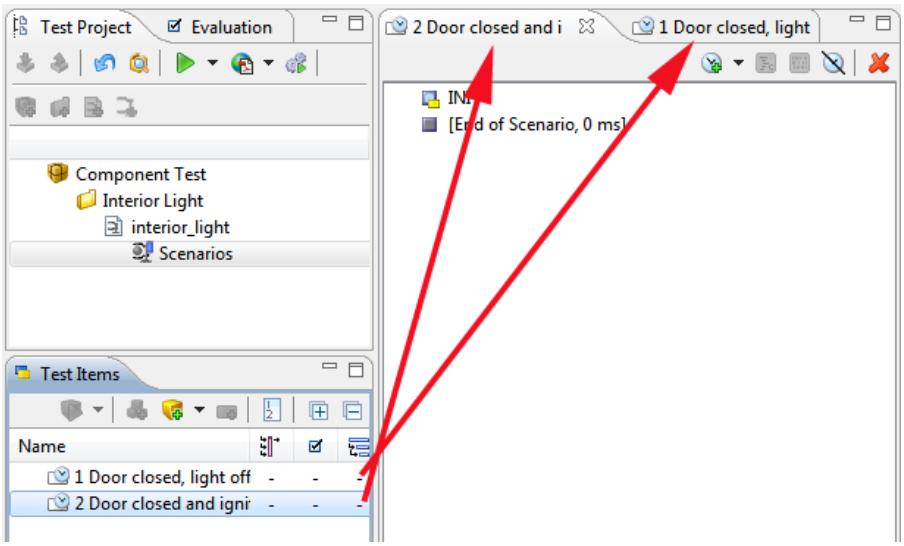


Figure 5.82: The names of the test cases are displayed in tabs of the view Work Task

5.3.6 Configuring the work tasks

As we know, the implementation of the component Interior Light assumes that the function tick() is called every 10 ms, i.e. the function tick() is the work task or handler task or heartbeat of the component. To enable TESSY for temporal component testing, Tessy must know about this, i.e. we must specify tick() as work task for the component manually.

To specify tick() as work task:

→ Select the tick().
→ Click on (Set as Work Task) in the tool bar (see figure 5.83).
The icon of the function will change from to .

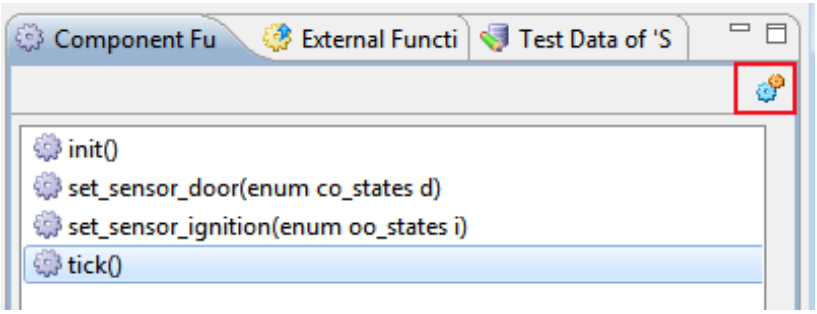


Figure 5.83: Setting the Work Task

5.3.7 Designing scenarios

5.3.7.1 First scenario

Since we want to test the temporal behavior of the component, we need to establish a time base:

→ Within the Scenario view in the center of the SCE perspective click on (Insert Time Steps).
→ Click three more times to include all in all 4 time steps.

The scenario consists of 4 calls to the work task, in our case tick(). The calls occur at 0 ms, 10 ms, 20 ms, 30 ms simulated time.
To stimulate the component besides the calls to tick(), you can simply drag and drop a function to the scenario.

→ Drag the init() from the Component Function view to the Scenario view (see figure 5.84).

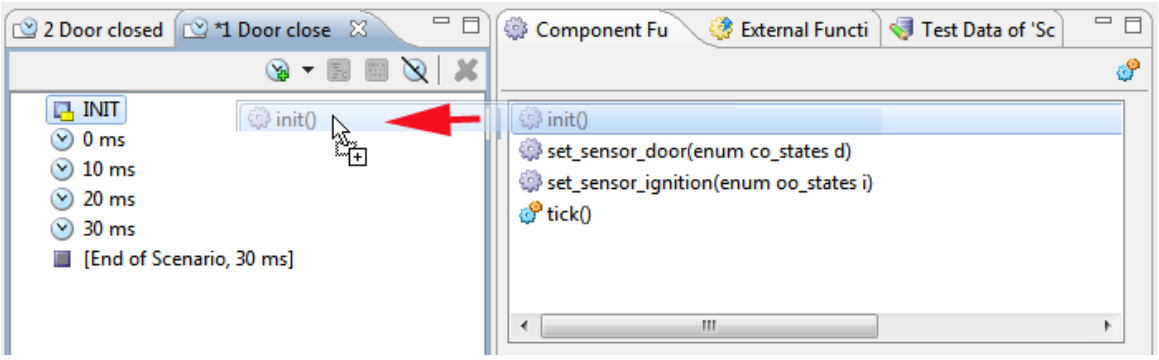


Figure 5.84: Dragging the function onto the scenario

Now we stimulate the component actually. We drag the component function set_sensor_door() to 30 ms simulated time:

→ Drag the component function set_sensor_door() to 30 ms simulated time (see figure 5.85).

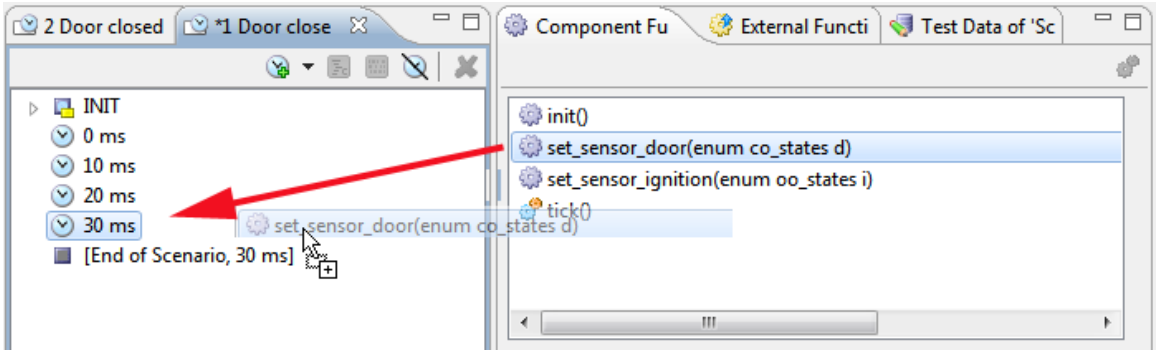


Figure 5.85: set_sensor_door() is dragged to 30 ms simulated time

Now you have to specify a value for the parameter of set_sensor_door() in the properties of set_sensor_door():

→ Open the Properties view and click in the cell under “value”.

→ Enter “closed”.
The value will be displayed in the parameter of the time step function (see figure 5.86).

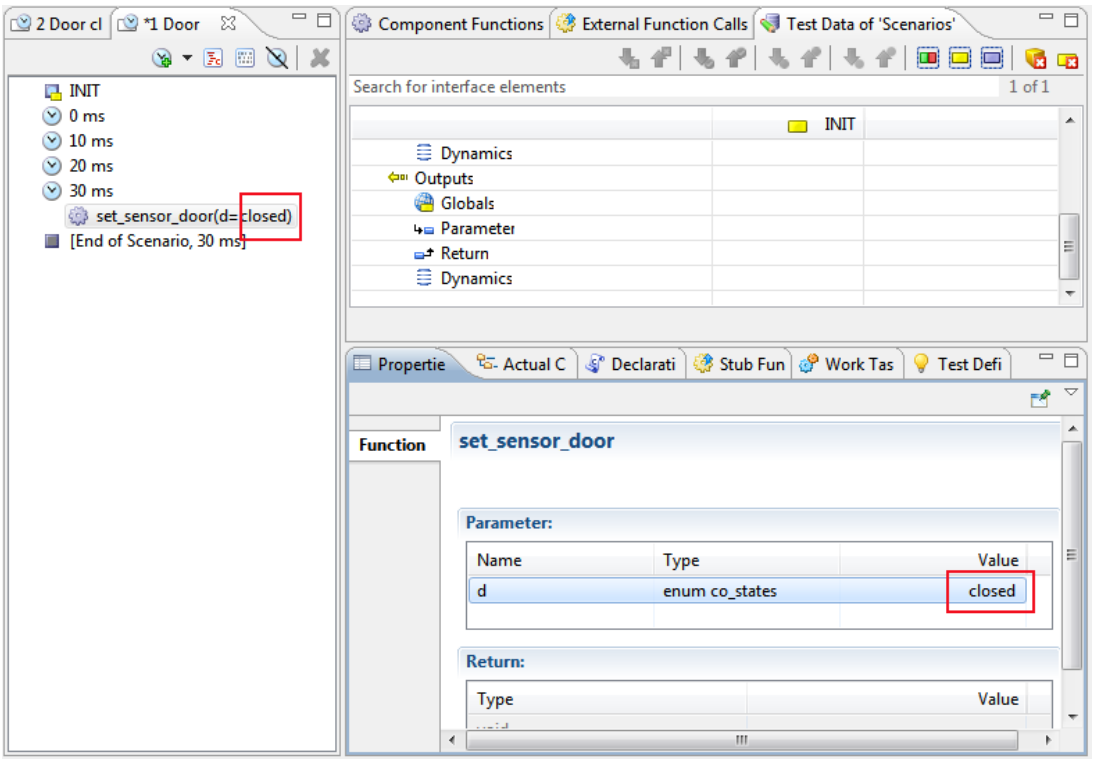


Figure 5.86: The parameter of set_sensor_door() is set

In the scenario above, after the fourth call to tick(), TESSY calls the component function set_sensor_door() with the parameter value “closed”. This should cause the component to react by calling LightOn(). We know from the implementation of Interior Light that this call will happen one tick later, i.e. after TESSY has called tick() a fifth time, at 40 ms simulated time.

→ Specify the expected result by dragging the function LightOn() from the External Function Calls tab to 30 ms simulated time (see figure 5.87).

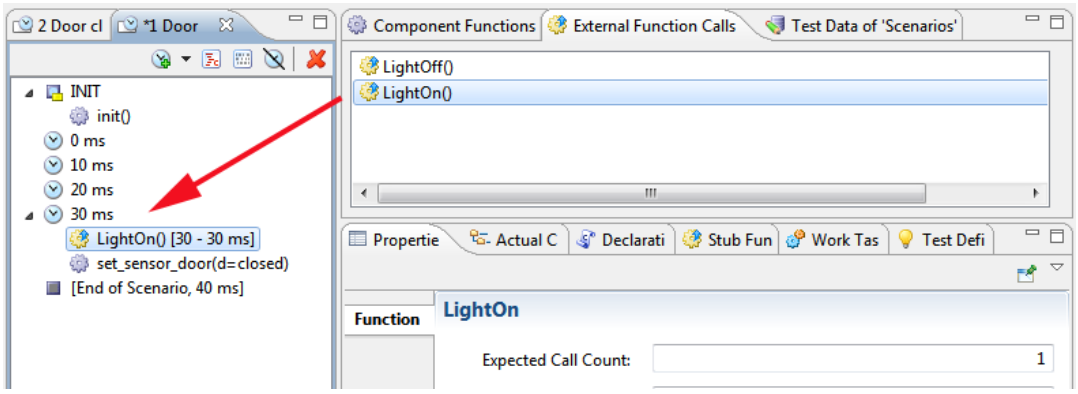


Figure 5.87: Dragging the function

The scenario above expects the call to LightOn() to happen at 30 ms simulated time, i.e. prior to the fifth call to tick(). This is indicated by the time frame “30 - 30 ms”. We know that the call will occur one tick later, and we want to treat this behavior as correct:

→ Extend the time frame in the properties to 10 ms (see figure 5.88).

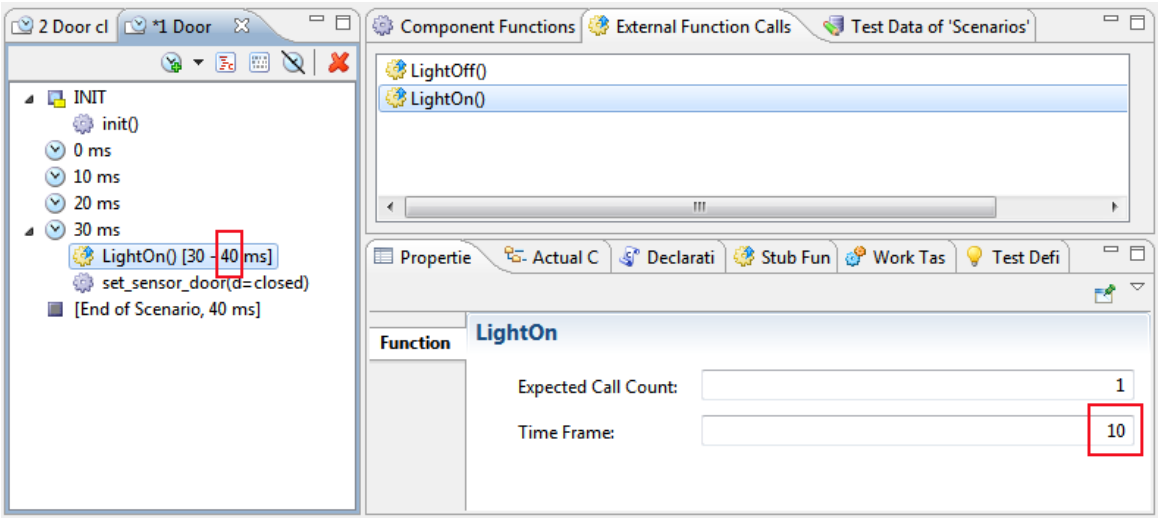


Figure 5.88: Extending the time frame

Notice that the icon of the test object has changed to yellow, indicating that the scenario is executable (the purple sign indicates a comment or description).

Besides the call to LightOn(), we also expect a call to LightOff(). Since the ignition will not be operated in this scenario, we expect the call to LightOff() to occur 5 seconds after the call to set_sensor_door() at the latest:

→ Drag LightOff() to 30 ms and extend the time frame by 5000 ms (see figure 5.89).
→ Save the data.

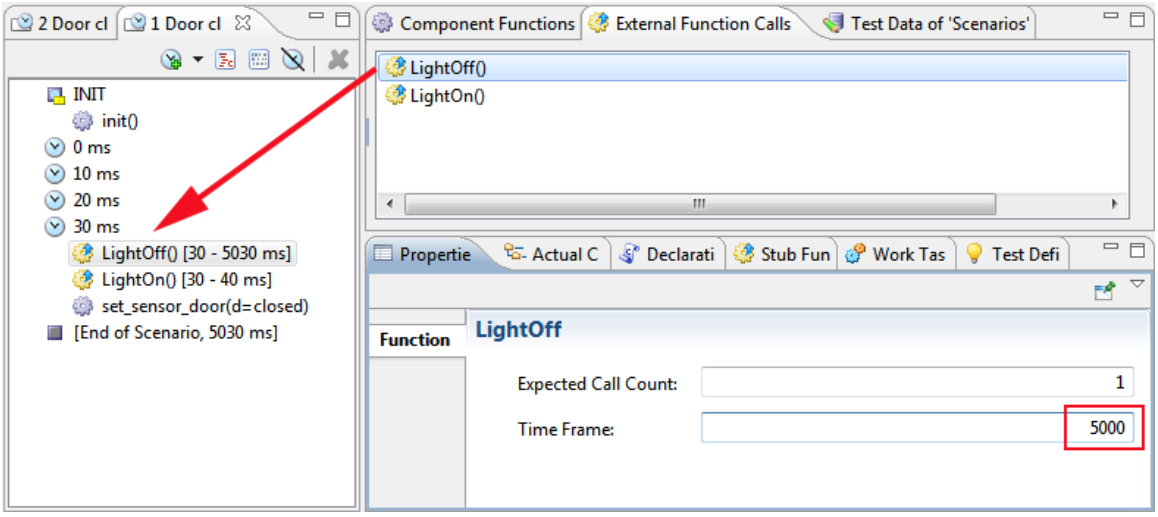


Figure 5.89: Setting the call “LightOff” and extending the time frame

In the scenario above, we have specified that we expect the call to LightOff() after 5 seconds at the latest, i.e. we would take a call to LightOff() after, say, 4 seconds as a correct result.

If we would want to accept only the occurrence of the call to LightOff() at 5030 ms of simulated time as correct, we then could use the context menu’s **Insert Time Step At…** command to create this point in time and assign the expected call to LightOff() to 5030 ms of simulated time, of course with the property “Time Frame” set to 0 and not to 5300.

5.3.7.2 Second scenario

Alike the first scenario

- → Insert some time steps into the scenario.
- → Drag and drop the component function set_sensor_door() to 10 ms simulated time.
- → Set the value of the parameter to “closed”.
- → To specify the expected behavior, drag and drop the function LightOn() from the External Function Calls tab to 10 ms simulated time.
- → Increase the time frame for LightOn() by 10ms.
- → Drag and drop the function “set_sensor_ignition” to 40 ms simulated time.
- → Select the 40 ms time step: The test data view will show a new column named “40 ms”. Set the value of the variable “sensor_ignition” to “on” and set the other values to be ignored (see figure 5.90).

→ Save the data.
The time step will now get a small yellow icon indicating that test data is fully available.

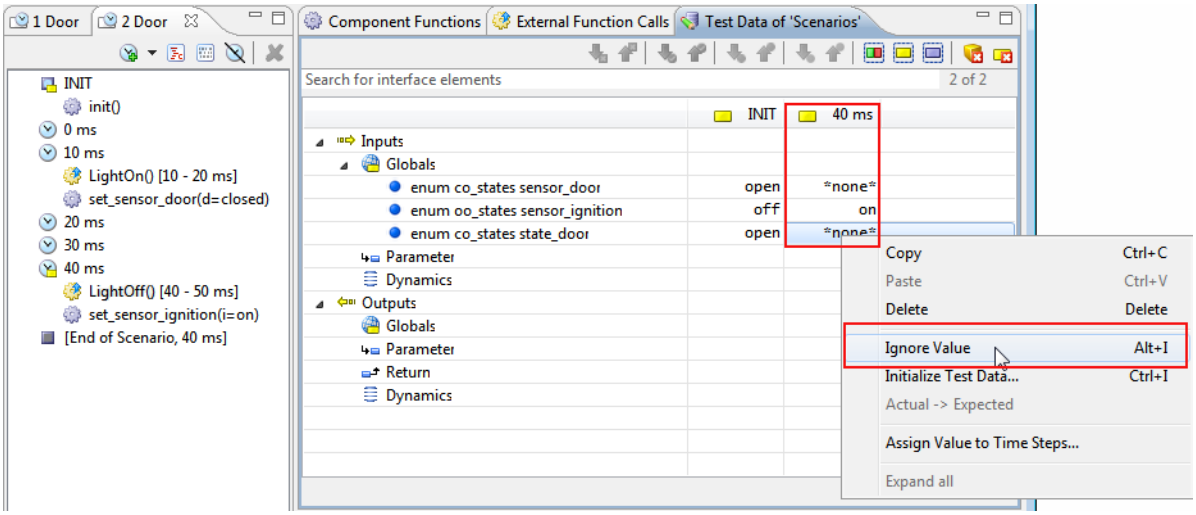


Figure 5.90: Setting values to “ignore”

→ To specify the expected behavior, drag and drop the function LightOff() from the External Function Calls tab to 40 ms simulated time.
The second scenario is now complete (see figure 5.91).

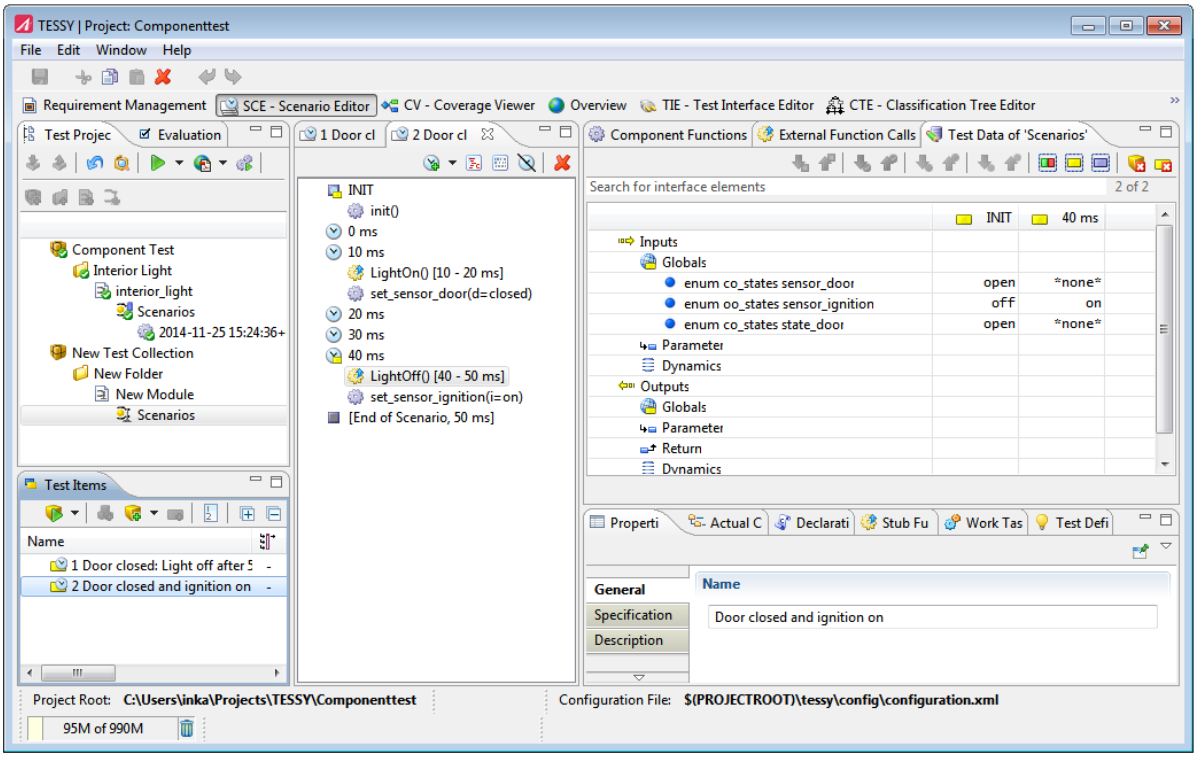


Figure 5.91: Designing the second scenario

The expected result of this action is the interior light going off immediately, i.e. after the next call to the work task tick(). This is specified in the scenario by dragging and dropping the function LightOff() to 40 ms simulated time and by increasing the time frame by 10 ms.

This is performed very similar as in the first scenario.

5.3.8 Executing the scenarios

To execute the scenatios,

→ click on ▶ in the tool bar of the Test Project view.

5.3.9 Evaluating the scenarios

After the scenarios have been executed, the color of the scenario icons changes to green. This indicates an overall “passed” verdict (see figure 5.92).

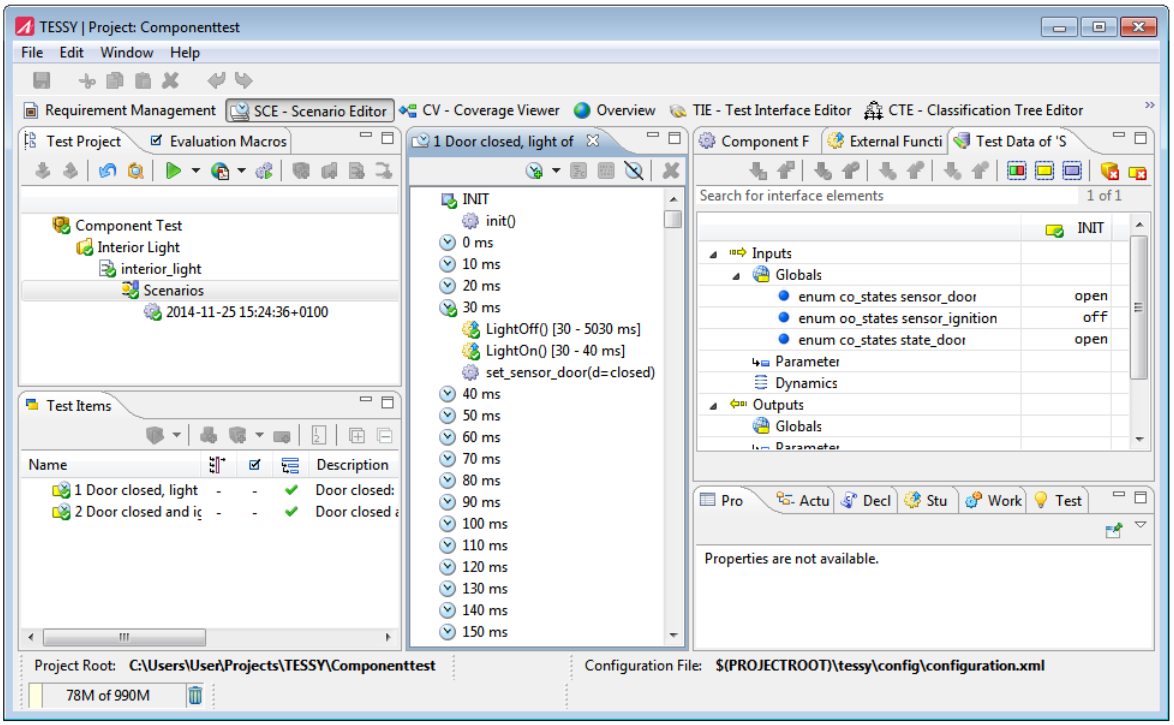


Figure 5.92: The scenarios of the component “passed” the test

5.4 Quickstart 4: Exercise C++

To understand the overall handling and create a simple classification tree we consider some aspects from the Quickstart 1: Unit test exercise is_value_in_range. Follow the steps below to import a C++ project with sample test cases:

→ In TESSY click on “File” > “Select Project”.
→ In the “Select Project” dialog click on the icon (Import Project).
→ Select the TESSY Project File under
“C:\Program Files\Razorcat\TESSY_4.x\Examples\C++”

→ Click “Open”.

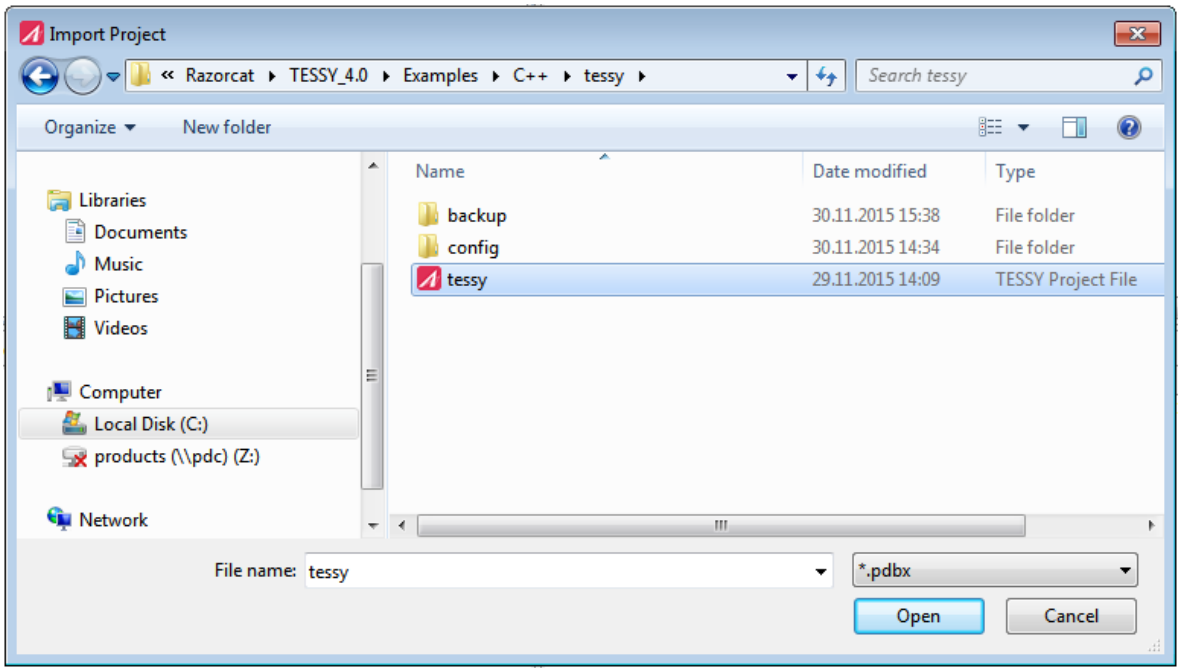


Figure 5.93: Importing a project

TESSY will show an example project within the project list. Such an example project will be copied completely to a user defined location when being opened. (see figure 5.94).

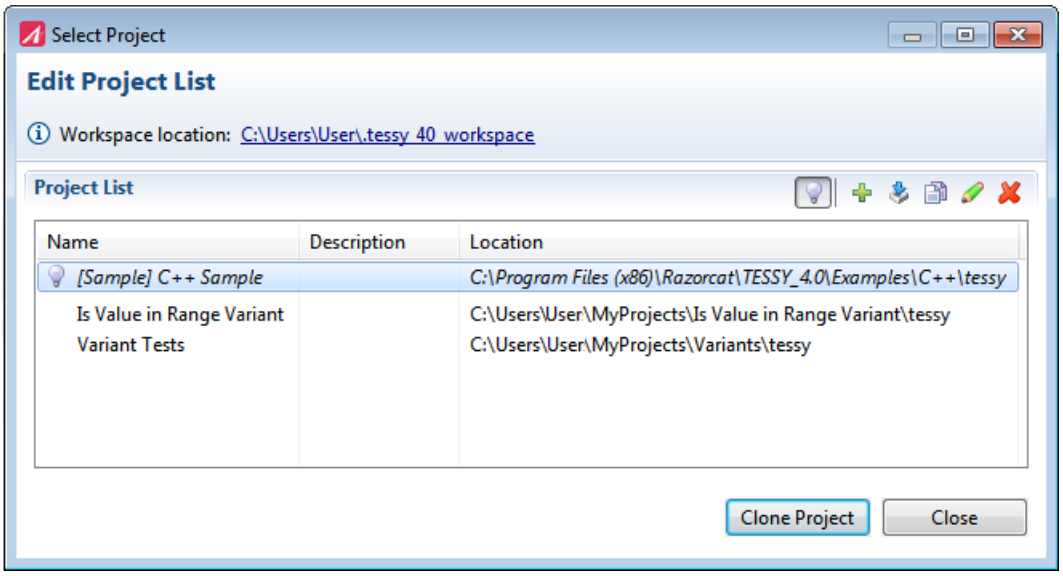


Figure 5.94: Cloning the project.

→ Double-click the C++ example project and choose an appropriate location.

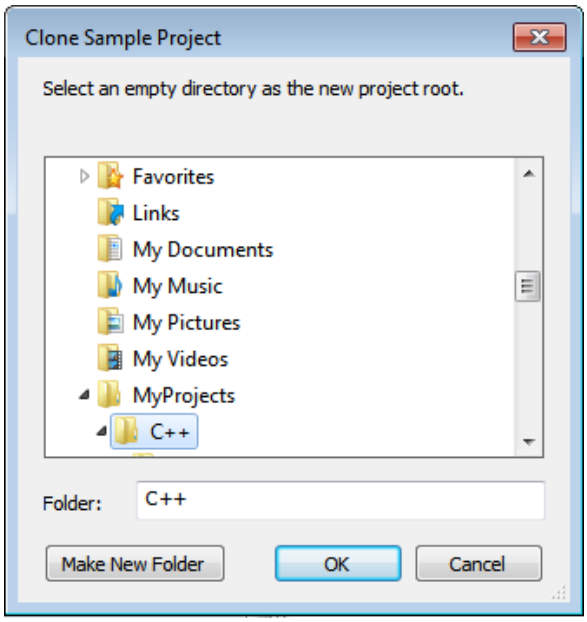


Figure 5.95: Cloning the project.

The project root will be displayed within the bottom line of TESSY.

→ Click “OK”.

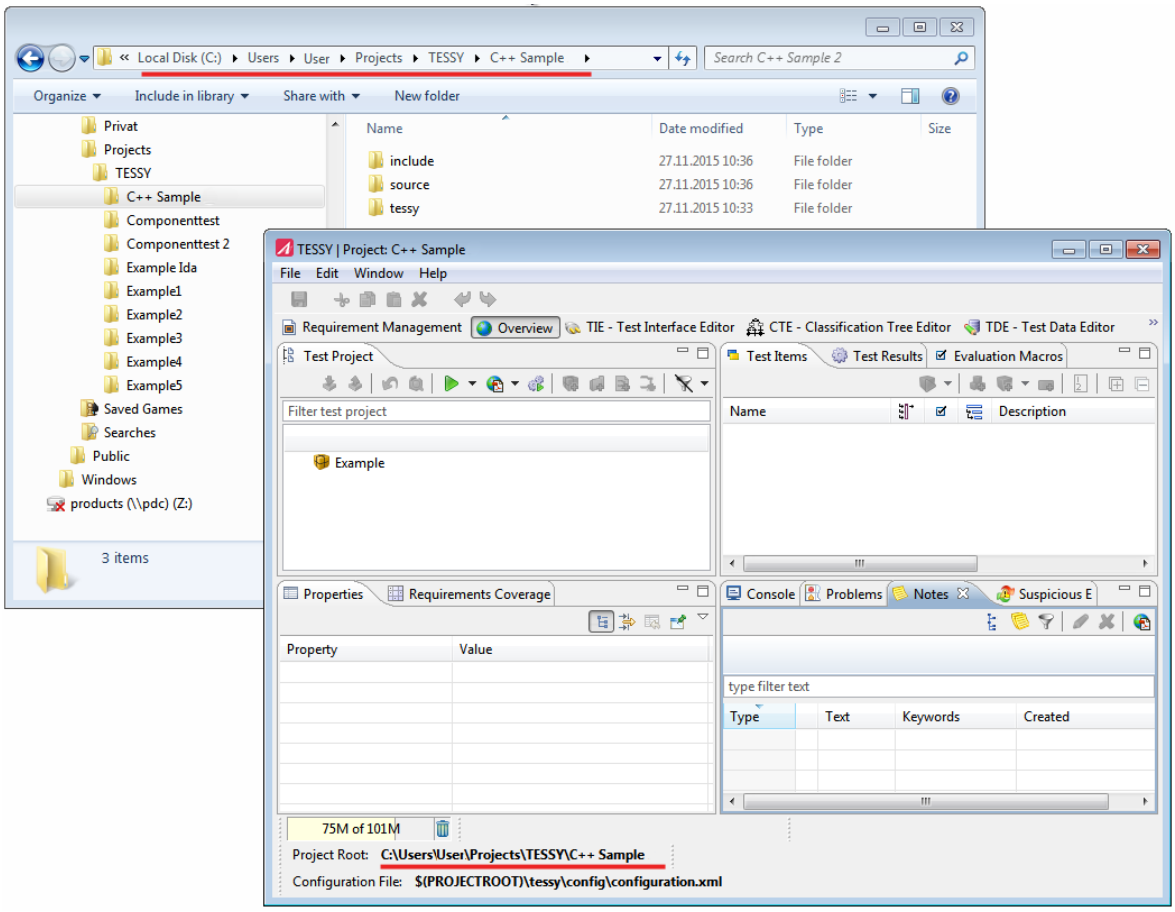


Figure 5.96: The project root is displayed within the bottom line of TESSY.

TESSY needs to restart and will open the project automatically.

After the restart the window “Restore Database” is displayed.

→ Click “OK”.

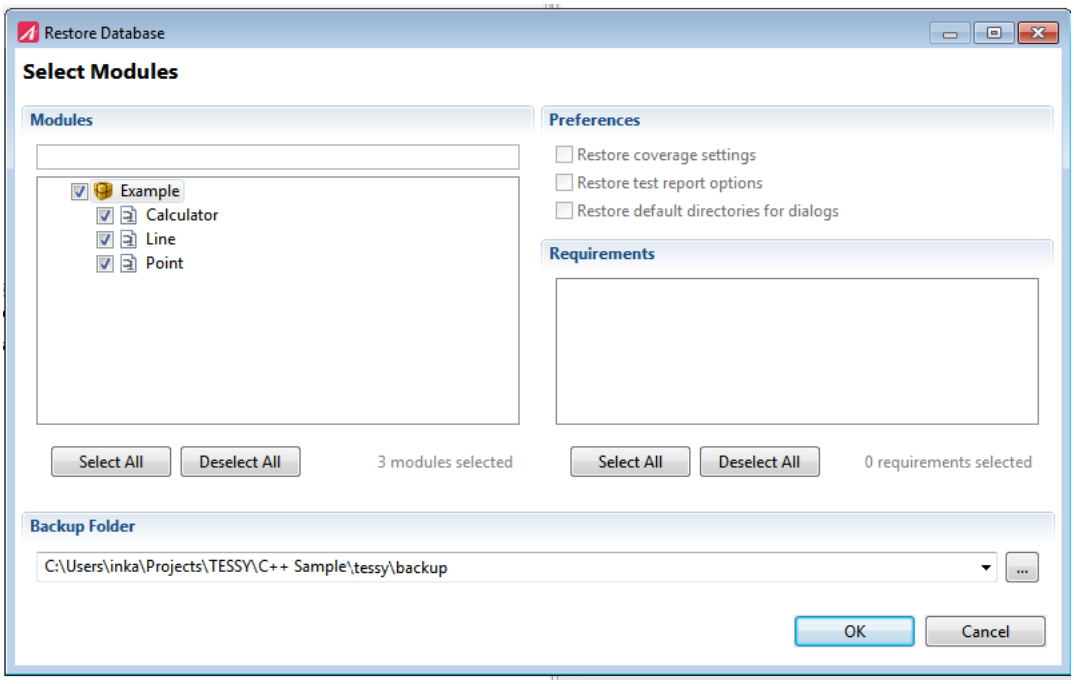


Figure 5.97: Restoring the database

TESSY will now restore the files.

→ After a few seconds the new project will be displayed within the Test Project view of TESSY.

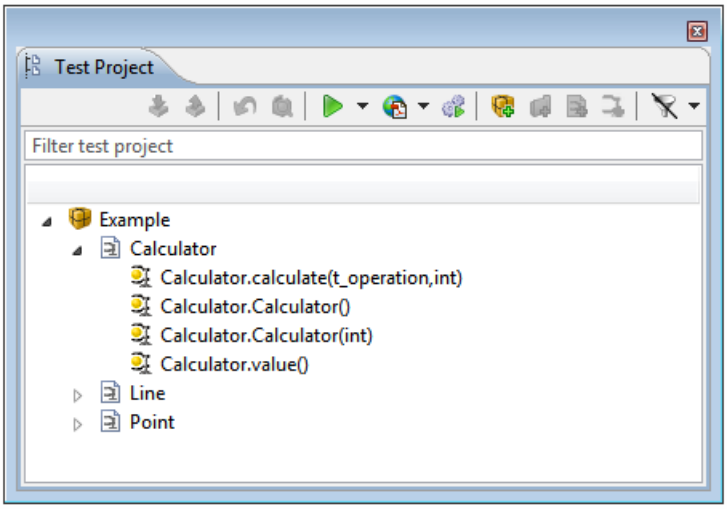


Figure 5.98: Test project view with the C++ project

Please review the test cases defined for some selected test objects. You will find more explanation and comments on the specifics of C++ testing within the application note “Using C++”.

5.5 Quickstart 5: Test driven development (TDD)

The test driven development support in TESSY allows to prepare tests already before any source code is available. If you have specification of what the individual software functions should do, you can manually create (synthetic) test objects and their (synthetic) interface variables. Afterwards you can create test cases and fill them with data as for normal test objects. Once the first version of the software is available, you need to assign the test objects to their respective implementation functions and you can start running the tests against the implementation. To create synthetic test objects for test driven development:

→ Create a module without assigning a source file. Only modules in this initial state allow to create synthetic test objects.

→ Add new test objects by clicking on the icon (New Test Object). Rename the newly created test objects.

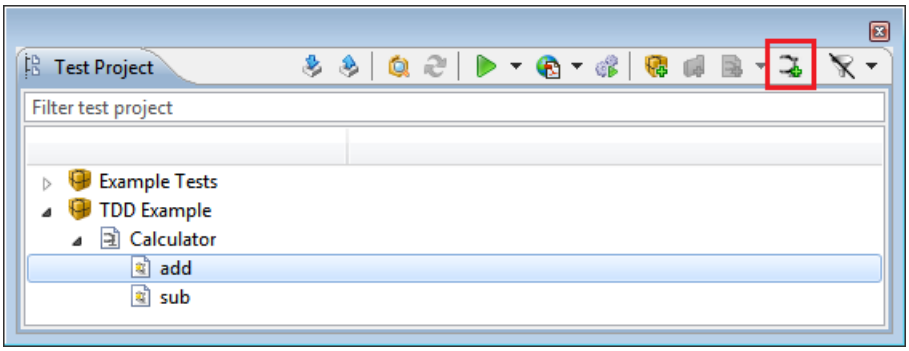


Figure 5.99: Adding synthetic test objects.

Each synthetic test object has an initially empty interface. To add variables:

→ Change to the TIE perspective.

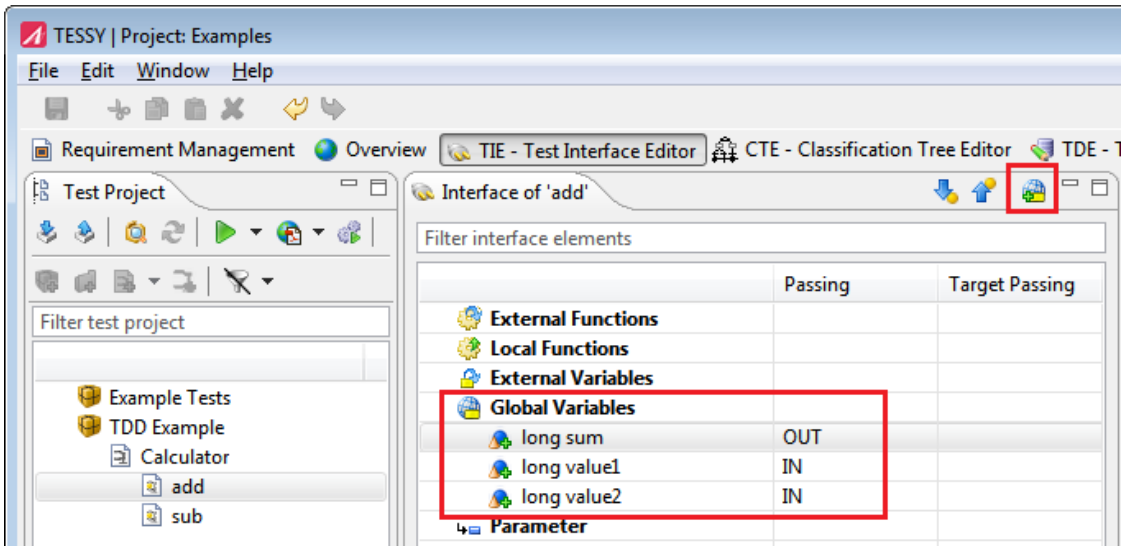


Figure 5.100: Adding synthetic variables.

→ Click on the icon (New Variable) to create input and output variables. You can create variables of all basic types to build the necessary interface. Afterwards you can create test cases.

→ Change to the CTE perspective.

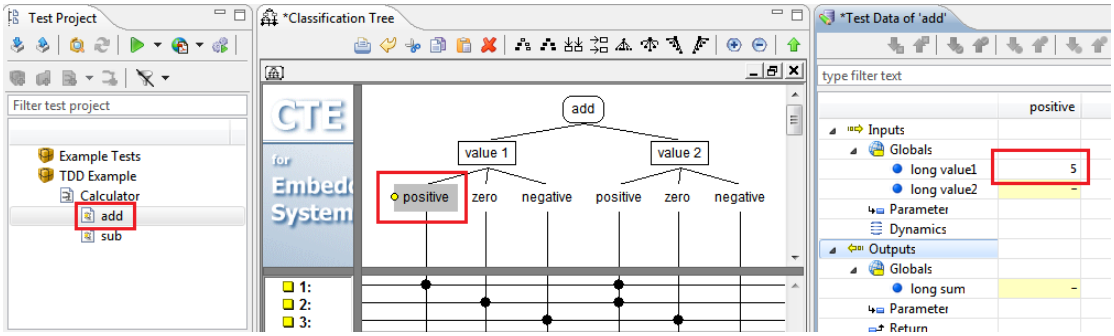


Figure 5.101: Creating test cases for TDD.

→ Assign values to CTE tree nodes as with normal variables.

→ Change to the TDE perspective.

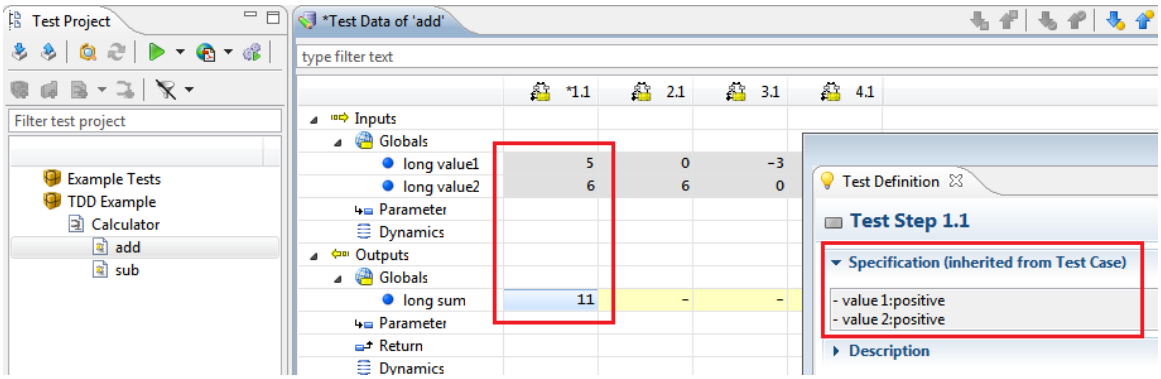


Figure 5.102: Filling test cases for TDD.

→ Enter all required test data and expected values.

The test cases will get a yellow icon if they are completely filled with test data. You cannot execute the tests because there is no source file to be tested available yet. But you can create a test details report showing the specification of all test cases:

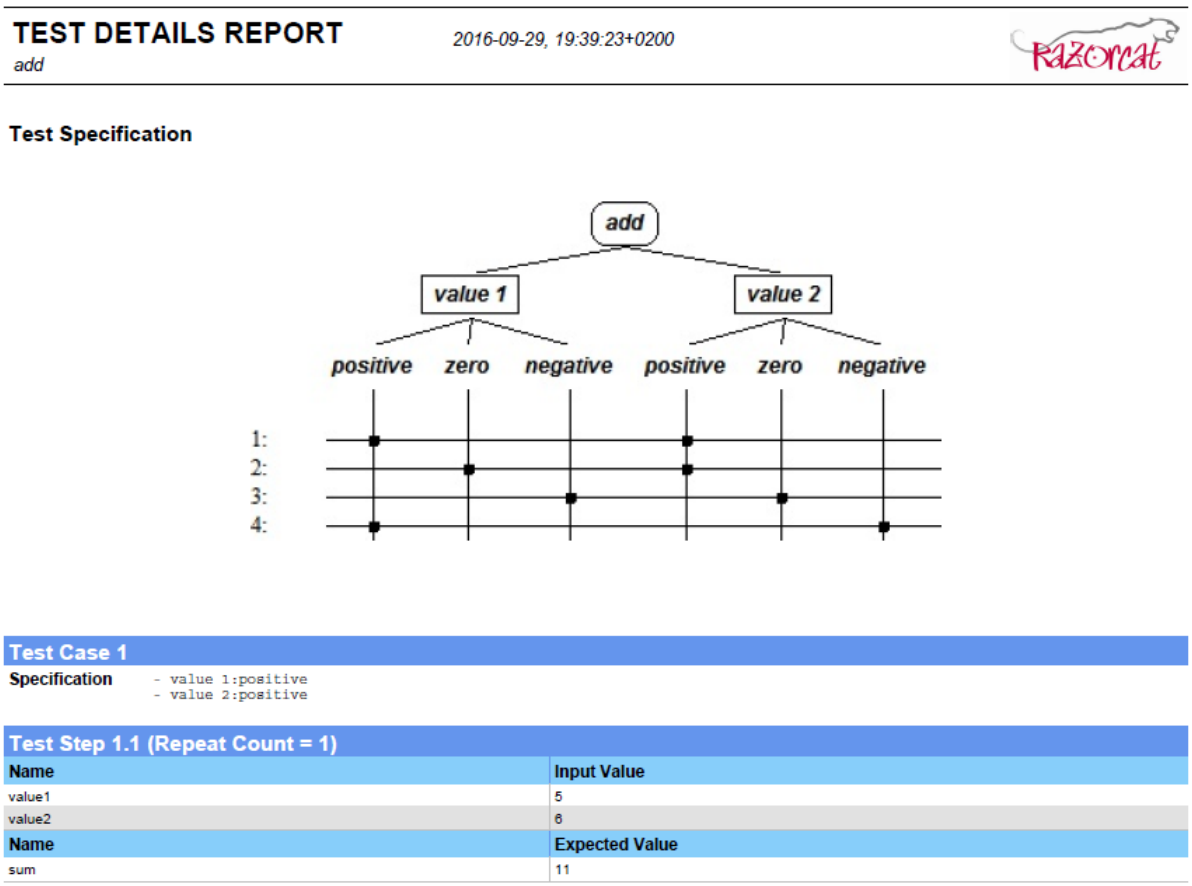


Figure 5.103: Test specification report for TDD.

When a first version of the source file is available, add this file to your module. Now you can analyze the module and you will see the functions contained within the source file and the test objects that you prepared for test driven development. It doesn't matter if the implemented test objects have other names, you can assign test objects within the next step.

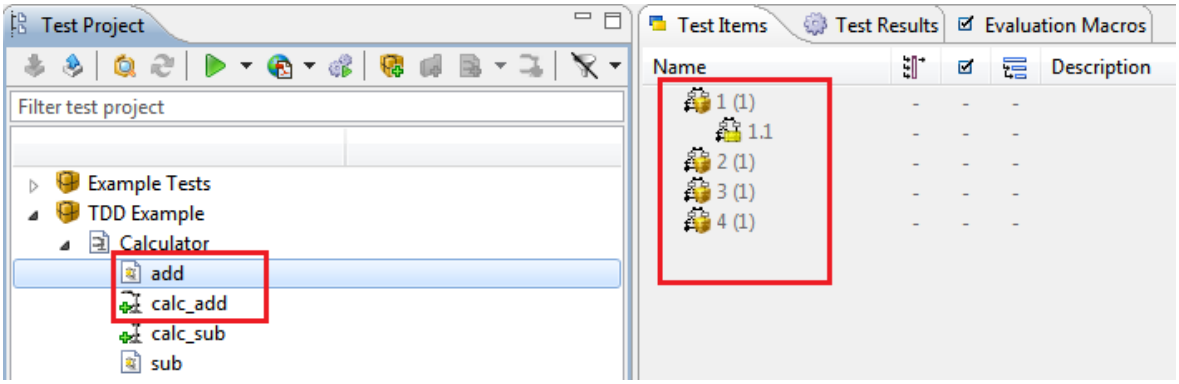


Figure 5.104: Implementation source file available for TDD.

To assign or reuse test objects:

→ Change to the IDA perspective.

→ Assign test objects as described within Compare view.

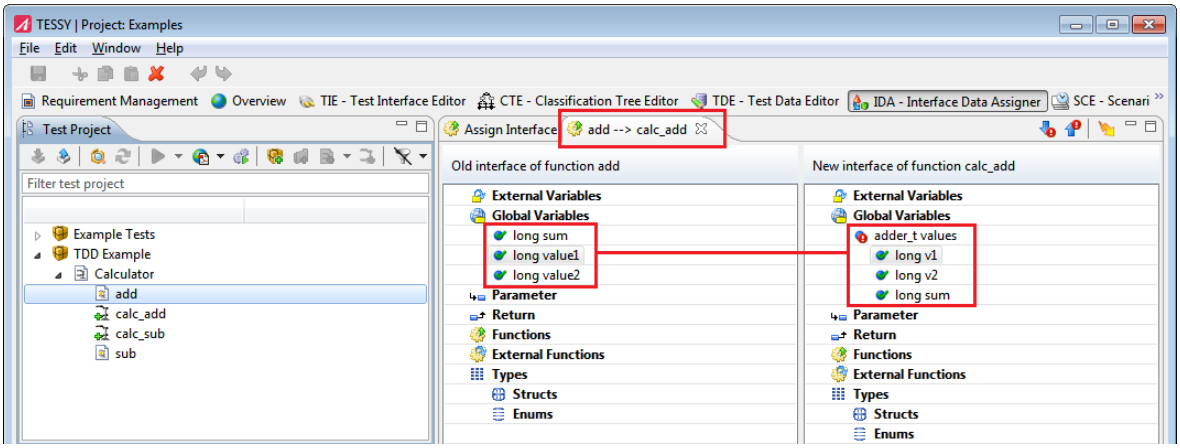


Figure 5.105: Assignment of synthetic test objects to the implemented functions.

→ Commit changes within the Compare view.

Important: When assigning synthetic test objects to the real implementation, the synthetic variables will not appear within the new interface because the purpose of this assignment is to assign the values of all synthetic variables to their respective implemented variables.

In the example shown within figure 5.105 you can see that the implementation uses a struct to hold all parameters whereas the synthetic test object has all values within scalar variables. Such differences can be resolved using the IDA assignment as it would be done for normal interface changes.

The test object “calc_add” is now ready for execution, all prepared test cases and data are available.

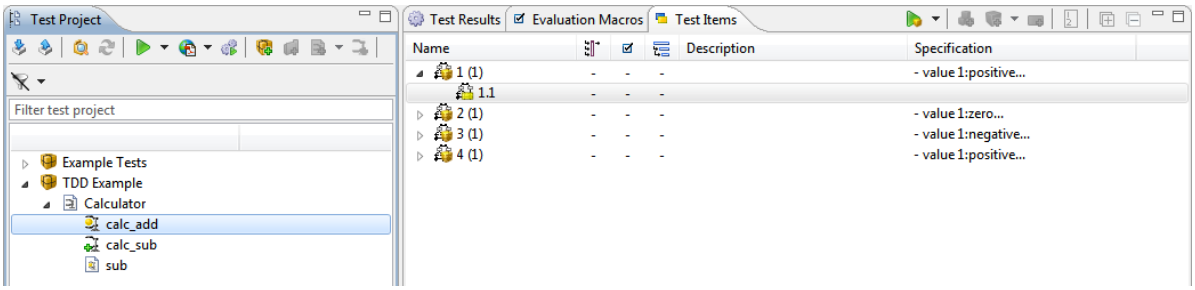


Figure 5.106: Readily assigned TDD test cases.

6 Reference book: Working with TESSY

This chapter provides detailed information of the test activities possible with TESSY. The headlines of the sections follow the actions taken during a test and refer to the corresponding perspectives and views, e.g. “CTE: Designing the test cases”.

The subsections describe the views of each perspective, displaying used icons and status indicators and giving quickly answers to your questions of “What can I do within this view?” and “How do I do …?”. So if you need help at some point, ask “Where am I?”. You should find the answer easily within this chapter. If you have questions about the workflow, consult chapter 5 Tutorial: Practical exercises.

Some views are displayed within various perspectives. Because views are context sensitive, not every operation is possible within every perspective. In this case the manual will refer to the respective section and perspective, where all operations of the view are possible.

6.1. Menu Bar Entries: Setting up the basics **171**

6.1.1. File menu 171

6.1.2. Window menu 172

6.1.3. Static Analysis Settings 177

6.1.4. Coverage Settings 178

6.1.5. Metrics Settings 179

6.1.6. Interface dictionary 180

6.1.7. Help menu 182

6.2. Overview perspective: Organizing the test **183**

6.2.1. Structure of the Overview perspective 183

6.2.2. Test Project view 184

6.2.3. Properties view 228

6.2.4. Requirements Coverage view . . . 238
6.2.5. Test Items view . . . 238
6.2.6. Test Results view . . . 248
6.2.7. Evaluation Macros view . . . 249
6.2.8. Console view . . . 249
6.2.9. Suspicious Elements view . . . 251
6.2.10. Problems view . . . 251
6.2.11. Variants view . . . 252
6.3. C/C++: Editing the C-source **256**
6.3.1. Opening the C/C++ perspective . . . 256
6.3.2. Structure of the C/C++ perspective . . . 257
6.3.3. Editor view . . . 258
6.3.4. Project Explorer view . . . 260
6.3.5. Outline view . . . 260
6.3.6. Properties view . . . 261
6.3.7. Console view . . . 262
6.4. Requirement management **263**
6.4.1. Structure of the Requirement Management perspective . . . 264
6.4.2. RQMT Explorer view . . . 265
6.4.3. Requirements List view . . . 274
6.4.4. Requirement Editor view . . . 275
6.4.5. Validation Matrix view / VxV Matrix view . . . 278
6.4.6. Test Means view . . . 279
6.4.7. Link Matrix view . . . 280
6.4.8. Suspicious Elements view . . . 285
6.4.9. Attached Files view . . . 290
6.4.10. Attributes view . . . 291
6.4.11. History view . . . 294
6.4.12. Differences view / Reviewing changes . . . 295
6.4.13. Related Elements view . . . 296
6.4.14. Problems view . . . 298
6.4.15. Document Preview . . . 298
6.4.16. Requirements Coverage view . . . 301
6.5. TEE: Configuring the test environment **306**
6.5.1. Starting the TEE . . . 307
6.5.2. Structure of the TEE . . . 308
6.5.3. Icons of the tool bar . . . 310

6.5.4. Status indicators . . . 310
6.5.5. Viewing all available configurations . . . 312
6.5.6. Configuration files . . . 315
6.5.7. Customizing the configurations . . . 315
6.6. THAI: TESSY Hardware Adapter Interface 323
6.6.1. The THAI Configuration file . . . 324
6.6.2. Environment Editor (TEE) Settings for THAI functionality . . . 325
6.6.3. Signals within the interface . . . 327
6.6.4. Entering test data for signals . . . 328
6.7. TIE: Preparing the test interface 329
6.7.1. Structure of the TIE perspective . . . 330
6.7.2. Test Project view . . . 330
6.7.3. Properties view . . . 330
6.7.4. Interface view . . . 331
6.7.5. Plot Definitions view . . . 351
6.8. CTE: Designing the test cases 356
6.8.1. The basic idea . . . 356
6.8.2. Structure of the CTE perspective . . . 357
6.8.3. Test Project view . . . 357
6.8.4. Properties view . . . 357
6.8.5. Outline view . . . 358
6.8.6. Classification Tree editor . . . 358
6.8.7. Test Data view . . . 375
6.8.8. Dependencies in CTE . . . 381
6.9. TDE: Entering test data 387
6.9.1. Structure of the TDE perspective . . . 387
6.9.2. Test Project view . . . 389
6.9.3. Test Results view . . . 389
6.9.4. Evaluation Macros view . . . 389
6.9.5. Test Items view . . . 389
6.9.6. Properties view . . . 390
6.9.7. Test Data view . . . 391
6.9.8. Test Definition view . . . 415
6.9.9. Call Trace view . . . 415
6.9.10. Declarations/Definitions view . . . 417
6.9.11. Prolog/Epilog view . . . 418
6.9.12. Stub Functions view . . . 427

6.9.13. Usercode Outline view . . . 432
6.9.14. Plots view . . . 434
6.9.15. Plot Definitions view . . . 435
6.10. Script Editor: Textual editing of test cases . . . 436
6.10.1. Editing test objects, test cases and test steps . . . 436
6.10.2. Structure of the Script Editor perspective . . . 437
6.10.3. Script Editor related Icons of the main tool bar . . . 438
6.10.4. Script states . . . 438
6.10.5. The Script Editor Outline view . . . 439
6.10.6. Synchronization with the internal model . . . 439
6.10.7. Merging script contents . . . 440
6.10.8. Importing and exporting script contents . . . 440
6.11. CV: Analyzing the coverage . . . 441
6.11.1. Structure of the CV perspective . . . 442
6.11.2. Instrumentation for coverage measurements . . . 443
6.11.3. Test Project view . . . 445
6.11.4. Called Functions view/Code view . . . 446
6.11.5. Flow Chart view . . . 447
6.11.6. Fault injection . . . 456
6.11.7. Statement (C0) Coverage view . . . 456
6.11.8. Branch (C1) Coverage view . . . 458
6.11.9. Decision Coverage view . . . 459
6.11.10. MC/DC Coverage view . . . 459
6.11.11. MCC Coverage view . . . 461
6.11.12. Call Pair Coverage view . . . 461
6.11.13. Coverage Report views . . . 462
6.12. IDA: Assigning interface data . . . 463
6.12.1. Structure of the IDA perspective . . . 464
6.12.2. Status indicators . . . 464
6.12.3. Test Project view . . . 465
6.12.4. Properties view . . . 465
6.12.5. Compare view . . . 465
6.13. SCE: Component testing . . . 469
6.13.1. Creating component tests . . . 470
6.13.2. Preparing the test interface . . . 472
6.13.3. Configuring the work tasks . . . 473
6.13.4. Designing the test cases . . . 475

6.13.5 Editing scenarios . . . 476
6.13.6 Executing the scenarios . . . 482
6.14. Fault injection **483**
6.14.1 Managing fault injections in the Coverage Viewer . . . 483
6.14.2 Creating fault injection test cases . . . 484
6.14.3 Creating and editing fault injections in the Coverage Viewer . . . 486
6.14.4 Fault injections within the report . . . 488
6.15. Backup, restore, version control **489**
6.15.1 Backup . . . 489
6.15.2 Restore . . . 491
6.15.3 Version control . . . 493
6.16. Command line interface **495**
6.16.1 Starting TESSY in headless mode . . . 495
6.16.2 Invoking “tessycmd.exe” . . . 496
6.16.3 Usage of “tessycmd.exe” . . . 497
6.16.4 Commands . . . 498
6.16.5 Execution and result evaluation . . . 498
6.16.6 Headless operation . . . 499
6.16.7 Example: DOS script . . . 500

6.1 Menu Bar Entries: Setting up the basics

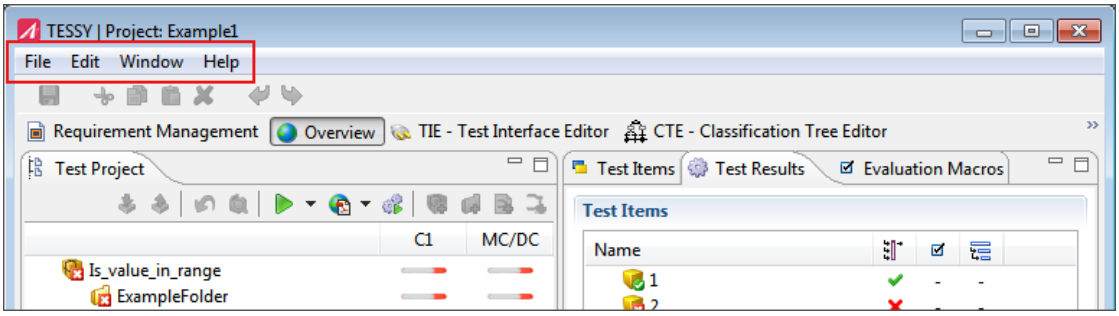


Figure 6.1: Menu bar of TESSY

The menu bar provides global operations such as project handling commands, editing commands, window settings, TESSY preferences and the help contents.

6.1.1 File menu

File menu entry	Setting options
“Select Project”	Opens the dialog “Select Project”. If you select another project, TESSY closes the current project, restarts and opens the selected project.
“New Project”	Opens the dialog “Create Project”. Refer to section 4.1.1 Creating a project database.
“Import Project”	Opens the Windows Explorer. Choose a project and click “Open”.
“Edit Project”	Opens the dialog “Project Configuration”. Refer to section 4.1.1 Creating a project database.
“Close Project”	Closes the current project. TESSY will restart and show the dialog “Select Project”.
“Edit Environment”	Opens the TEE, the Test Environment Editor. Refer to chapter 6.5 TEE: Configuring the test environment.

continue next page

File menu entry	Setting options
“Open File…”	Opens the project file.
“Database Backup”	Refer to chapter 6.15 Backup, restore, version control.
“Exit”	Quits TESSY.

Table 6.1: File menu options

6.1.1.1 Edit menu

Here you will find common actions as “Delete” or “Undo”, “Redo” etc. You can use as well the context menu. Refer to section 4.3 Using the context menu and shortcuts.

6.1.2 Window menu

Window menu	Setting options
“Show Console”	Opens the Console view on the lower right of the graphical user interface.
“Show Problems View”	Opens the Problems view on the lower right of the graphical interface.
“Show View…”	Opens a list of all available views within TESSY.
“Show Perspective…”	Opens a list of various perspectives available within TESSY to switch directly to the desired perspective. You can as well use the graphical user interface to do so. For more information please refer to section 4.2 Understanding the graphical user interface.
“Reset Workbench”	With a click you reset the positions of all perspectives and views to the default setting.
“Preferences”	Switch to the Preferences menu to be able to adapt basic functionality to your needs.

Table 6.2: Window menu options

6.1.2.1 Window > Preferences menu

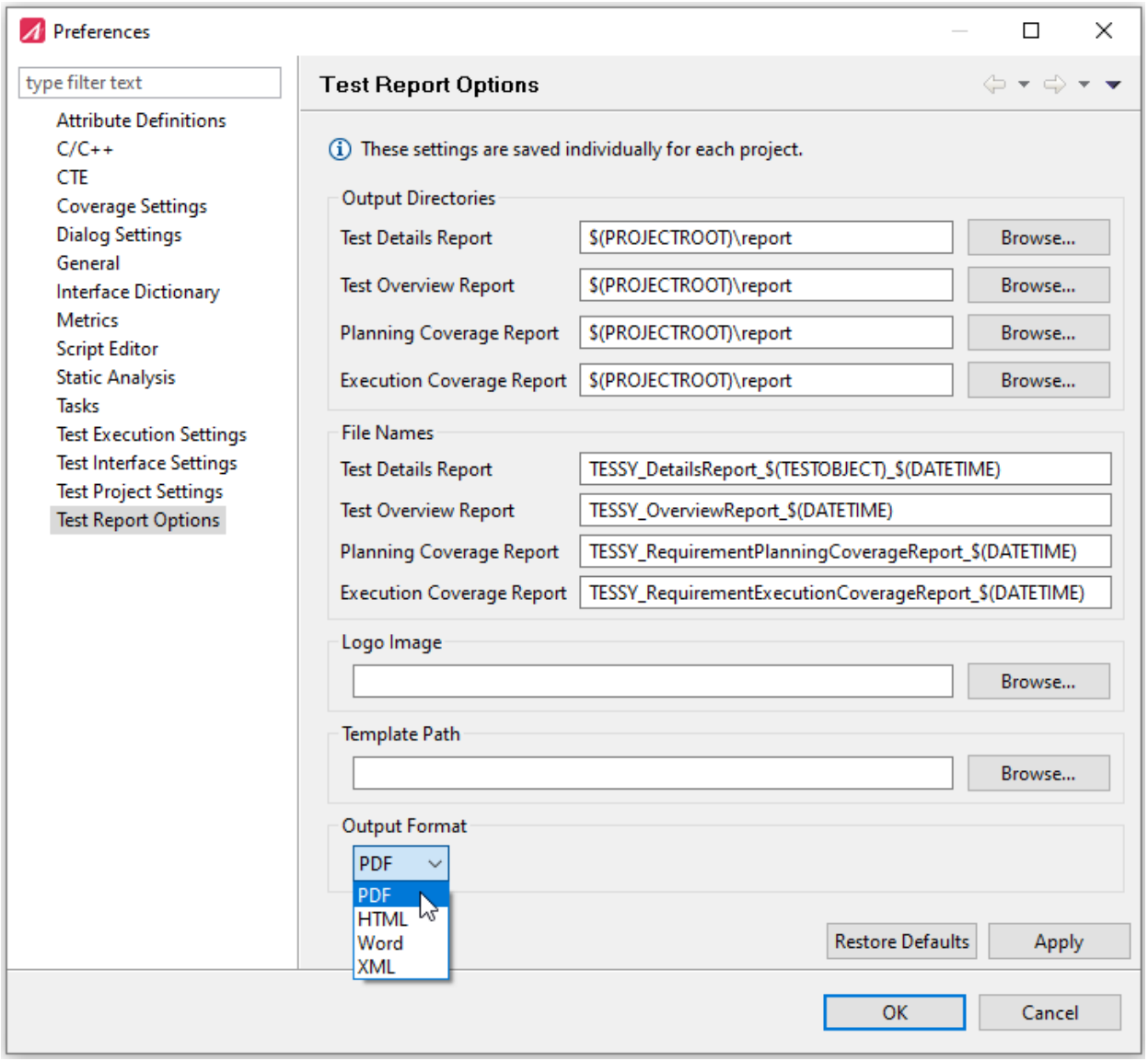


Figure 6.2: Preferences menu of TESSY

Within section “Preferences” of the Window menu you find many options for setting basic functions to your needs:

Preferences menu	Setting options
“Coverage Settings”	In this section you can setup an instrumentation for coverage measurement that will be the default for all of your projects (see figure 6.4). You can still set up a different instrumentation for every test collection, folder, module or test run. Refer so section 6.2.3.4 Coverage tab within the Properties view. TESSY includes various pre-defined coverage measurements for common known safety standards. You can as well • modify the existing selections (tick a box), • choose the test type for the selection (unit or component test), • define your own (click on “Create”) coverage selections, • import (click on “Import”) and export coverage instrumentation settings as XML file. To define a default instrumentation for your project, select from the pull-down list on top.
“Dialog Settings”	Within this section you can • set the default directories for imports of test objects, exports for test objects and modules and for adding includes and source files, • decide, which confirmation dialogs will be shown.

continue next page

Preferences menu	Setting options
“Metrics”	Within this section you can • display or hide additional measurements (e.g. the average and maximum CC measures) and specify error/warning levels for CC calculation, • choose to apply the RS and TC/C measure into the test object result calculation, • disable the calculation of coverage totals. (See figure 6.5 for details.)
“Static Analysis”	TESSY supports the static analysis tools “Cppcheck” and “PC-Lint”. To enable static analysis, • go to “Path to executable” and select your cppcheck.exe or lint-nt.exe. On the preference page you can also specify the options passed to the analysis tool, and whether the static analysis is to be run in the environment context of a module (see figure 6.3). • To perform the static analysis, right-click on a module and select “Analyze Source” from the context menu. Any issues that have been found will be printed on the console and listed in the Problems view. Please note that if you change the options on the preference page such that the message output format is different from the default setting used by TESSY, the Problems view may not be able to correctly parse and display the output.
“Test Execution Settings”	Within this section you can choose if selections and settings should be remembered, e.g. if you tick unter “Remember test instrumentation settings” the option “Globally for all test objects” the last used coverage selection will be used, see section 6.2.2.12 Instrumentation settings.

continue next page

Preferences menu	Setting options
“Test Interface Settings”	Within this section you can personalize TESSY’s behaviour within the Interface view, such as • sorting order • hide empty sections • show available types for test objects
“Test Project Settings”	Within this section you can personalize TESSY’s behaviour within the Test Project view, such as • sorting order • showing the Overview perspective when starting • showing the Problems view on error
“Test Report Options”	Within this section you can • change the output directory for the various TESSY reports • change the filenames • change the default Razorcat logo within the reports to your own company logo (PNG, JPG or GIF files are possible) (see figure 6.2) • choose an output format (PDF, HTML or DOCX) If you move the mouse over the entry, a tooltip will give you further information of the options!

Table 6.3: Preferences menu options

The following preferences will be stored within backup files when saving the whole project database as described within Backup, restore, version control:

- Coverage settings
- Dialog settings
- Metrics settings
- Test execution settings
- Test report options

6.1.3 Static Analysis Settings

You can configure the static analyzer tools “CppCheck” and “PC lint” to be executed when analyzing modules.

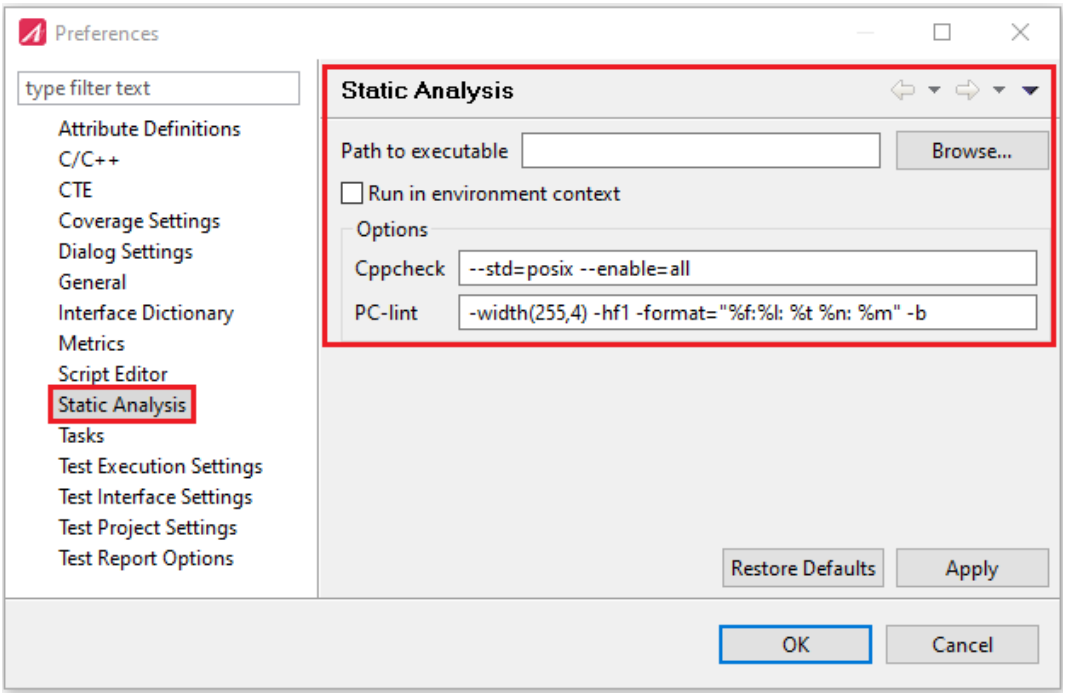


Figure 6.3: Static Analysis in the Preferences menu

The static analysis settings are used for calling the respective static analyzer tool. You need to enter the path to the binary and change the command line options to your needs.

6.1.4 Coverage Settings

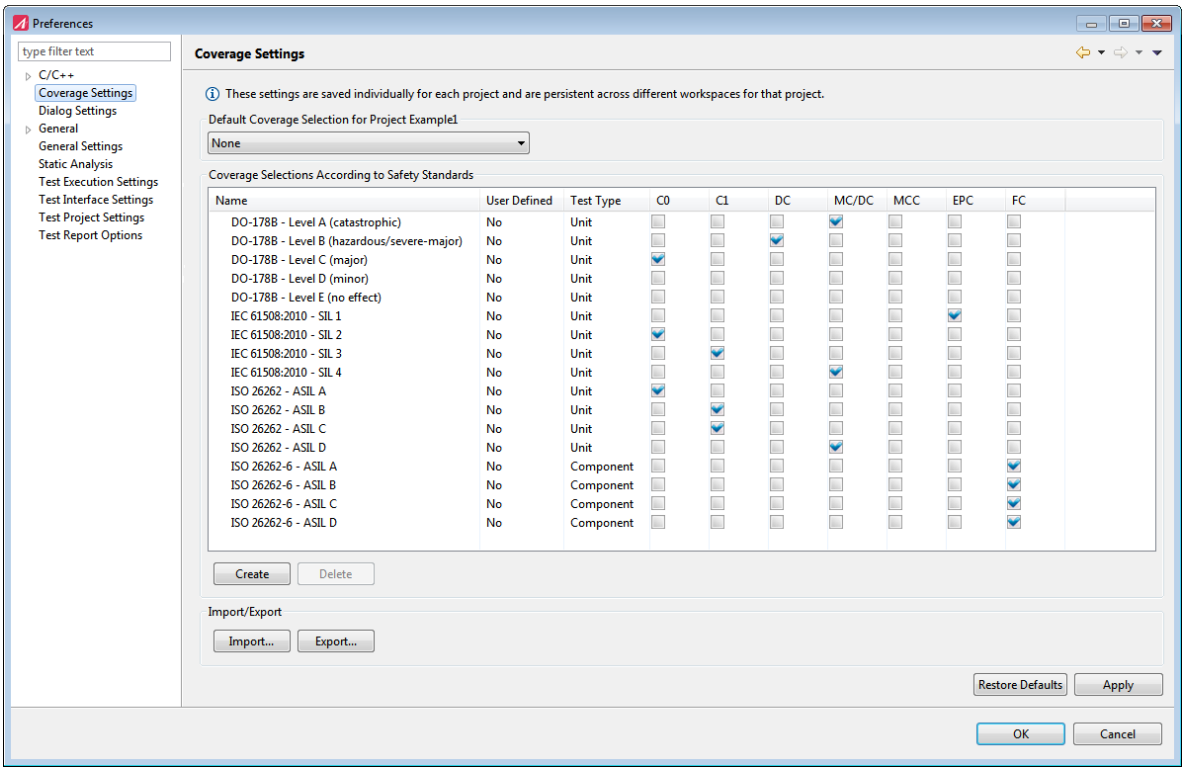


Figure 6.4: Pre-defined coverage instrumentation settings

For a selection of the most applicable safety standards, there are pre-defined coverage settings available according to the recommendations given within those standards. You can choose a coverage setting for the appropriate standard and level as default for all modules of your project. When running tests with coverage instrumentation, the respective settings will be applied automatically.

You can as well define your own coverage settings if the standard you are using is not available within the list.

6.1.5 Metrics Settings

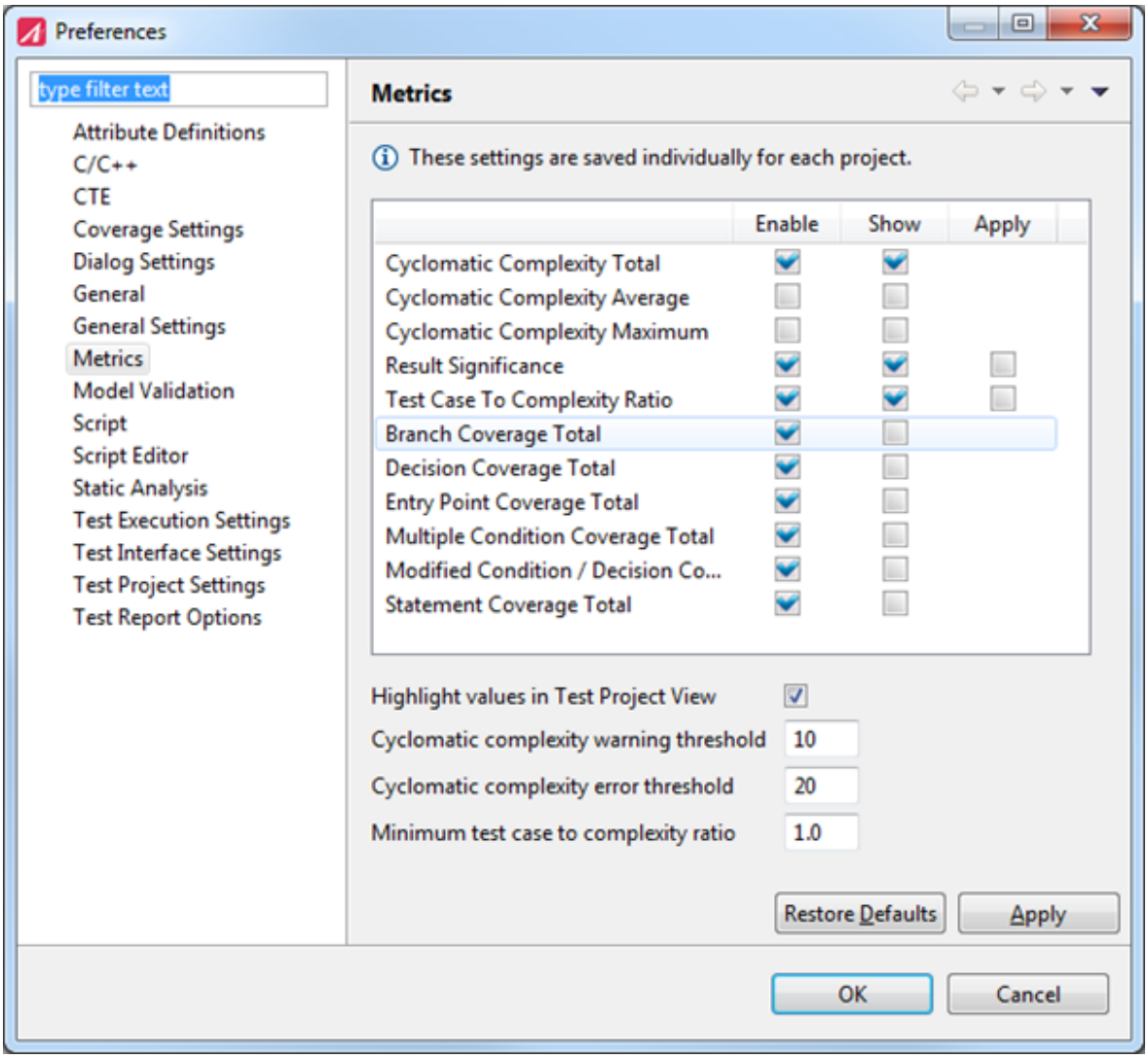


Figure 6.5: Pre-defined coverage metrics settings

You can enable the calculation of the CC average and maximum values and decide to show them within the test project view. The two measures Result Significance (RS) and Test Case To Complexity Ratio (TC/C) can be activated to be applied for the result calculation of test objects. If you tick the Apply check box, a failed value of the respective measure will lead to a failed test execution result of the test object.

The coverage totals are enabled by default in order to provide better progress information when developing and executing tests. The coverage totals will be propagated up to the test collection so that you have the total number of branches or conditions on test collection level. As soon as any test objects have been executed, the reached branches or conditions of those executed test objects will

be propagated upwards but all other test objects with missing tests or coverage data will still be taken into account.

If you don't need the metrics measurements or do not want to apply the coverage totals, you can disable all those metrics to avoid unnecessary calculations.

6.1.6 Interface dictionary

The interface dictionary is used to collect additional information about global variables. It automatically collects the global variables of all modules of a project. Whenever a module is analyzed the interface dictionary will be updated with the variables contained within the module interface.

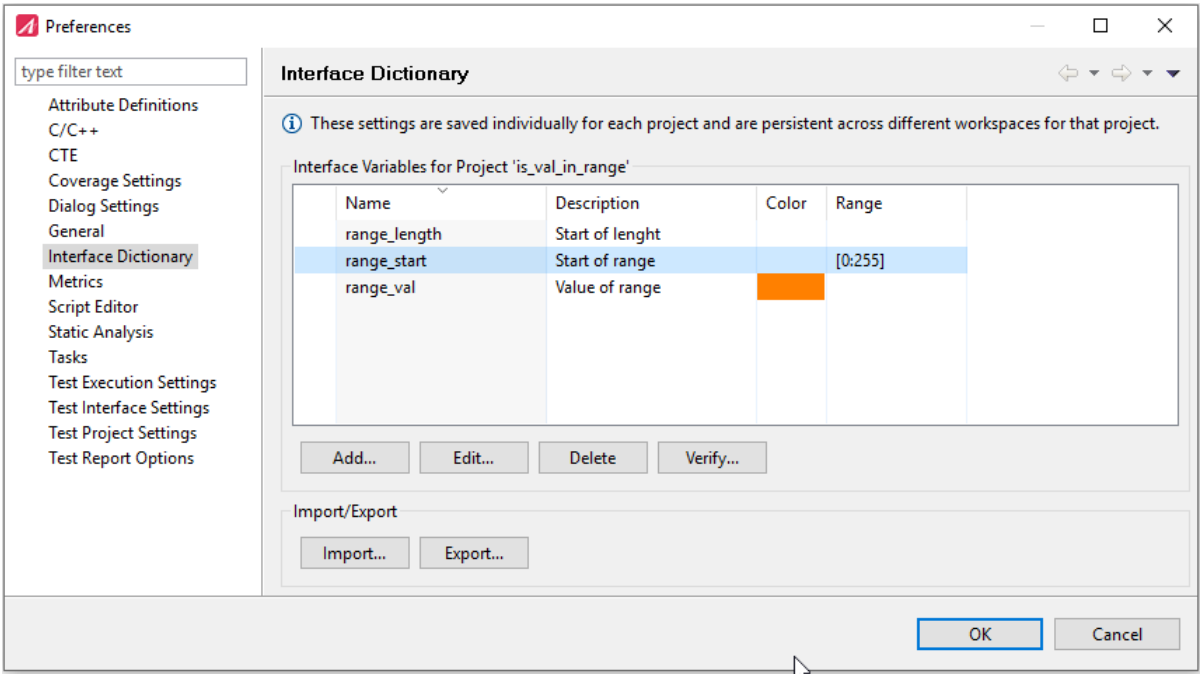


Figure 6.6: Interface dictionary within the Preferences

Double-click a variable in the interface dictionary to open the edit window:

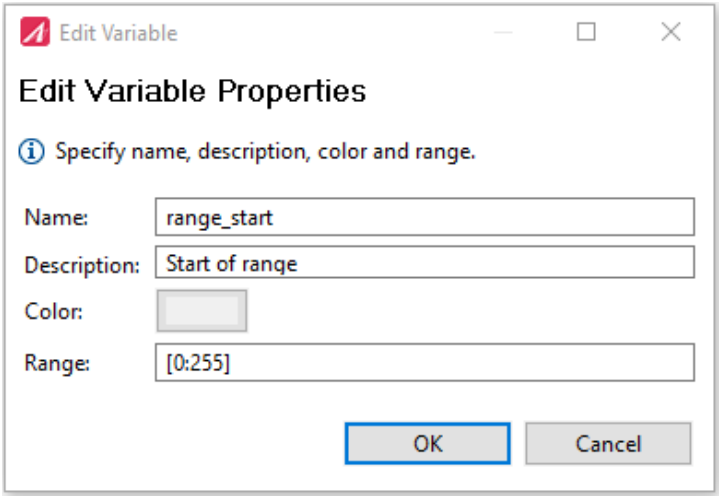


Figure 6.7: Editing variables in the interface dictionary

You can specify a description, color and a range of valid values for this variable. These information can later be used for automatic generation and update of classification trees.

With the “Add…” button new variables can be created.

Using the “Verify…” button allows you to check the edited list of variables with the available variables within all module interfaces of the project. If any variables are not existing in any module interface, such variables will be decorated with a warning icon.

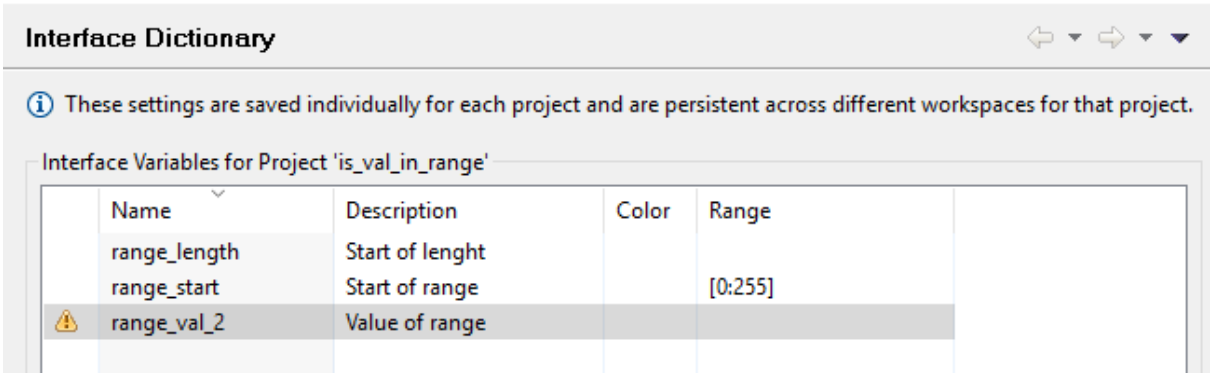


Figure 6.8: Interface dictionary variable with warning icon

If you delete variables, they will be added again when a module containing those variables is analyzed.

The interface dictionary can be exported and imported as XML file (e.g. for editing or initial creation) and it can be saved and restored during the database backup and restore operation.

6.1.7 Help menu

Help menu	Setting options
“About TESSY”	Shows information i.e. the TESSY version.
“User Manual”	Opens the TESSY User Manual.
“Safety Manual”	Opens the TESSY Safety Manual.
“Documentation…”	Contains various documentation for compiler and targets and frequently asked questions (PDF files).
“Key Assist…”	Opens a list with shortcuts.
“Create Support File…”	Creates a support file. Refer to section 7.1 Contacting the TESSY support.
“Logging…”	Opens the “Edit Settings” dialog.
“Start Shell”	Starts a bash shell that can be used to try out the command line execution of TESSY. The PATH variable is already set to the bin directory of the currently running TESSY installation, so that you can run tessycmd immediately. Refer to section 6.16 Command line interface.
“Open Log File”	Opens the external problems log 7.2.3 Opening external problem logs using the Help menu.
“Open Workspace Problems Log”	Opens an information dialog with a list of problems 7.2.3 Opening external problem logs using the Help menu.
“Open Problems Log…”	Opens the Windows file chooser to open a problems log7.2.3 Opening external problem logs using the Help menu.

Table 6.4: Help menu options

6.2 Overview perspective: Organizing the test

Important: If you have not created a project yet, do so as described in the chapter “Basic handling” in section 4.1.1 Creating a project database!

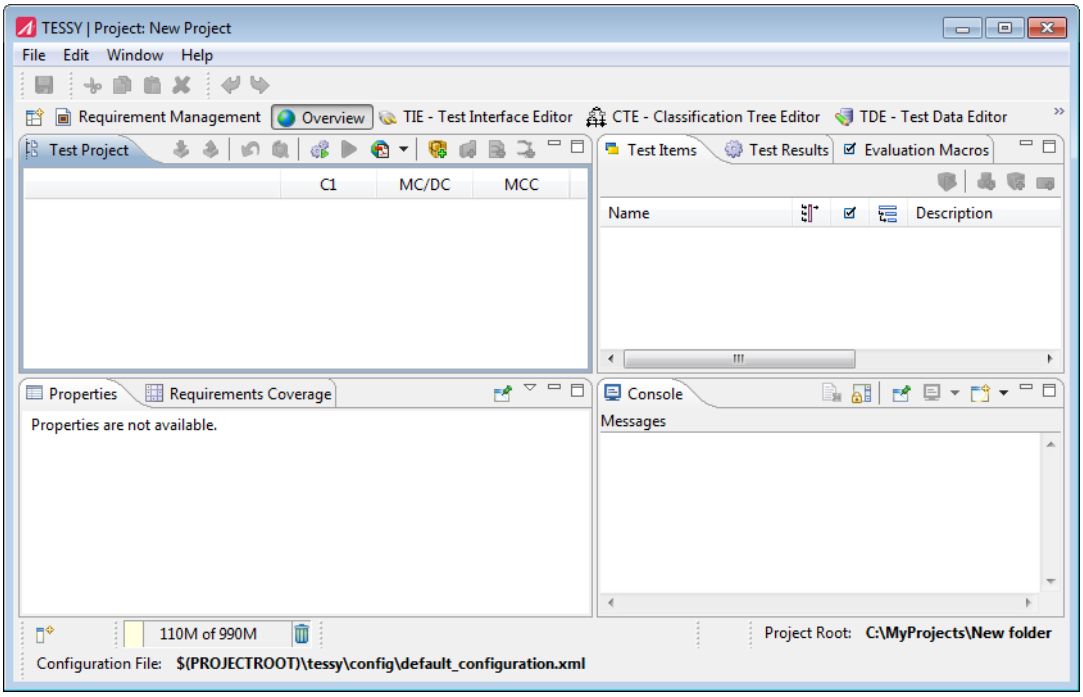


Figure 6.9: Overview perspective

6.2.1 Structure of the Overview perspective

Pane	**Location** (default)	**Function**
Test Project view	upper left	To organize the project: Create test collections, modules and test objects; execute the test, create reports and have a fast overview on your project.
Properties view	lower left	To edit all properties, e.g. adding sources or including paths to your modules.

continue next page

Pane	Location (default)	Function
Requirement Coverage view	lower left	To select and link the requirements that you managed within the Requirement management perspective.
Test Items view	upper right	To create test cases and test steps manually.
Test Results view	upper right	To view the test results.
Evaluation Macros view	upper right	To view evaluation macro results if the usercode of the test object contains any.
Console view	lower right	To display messages of sub processes invoked during test execution, e.g. compiler calls.
Suspicious Elements view	lower right	To review changes of requirements, modules or test objects that require updating the linked test cases.

Table 6.5: Structure of the Overview perspective

6.2.2 Test Project view

Test Project view

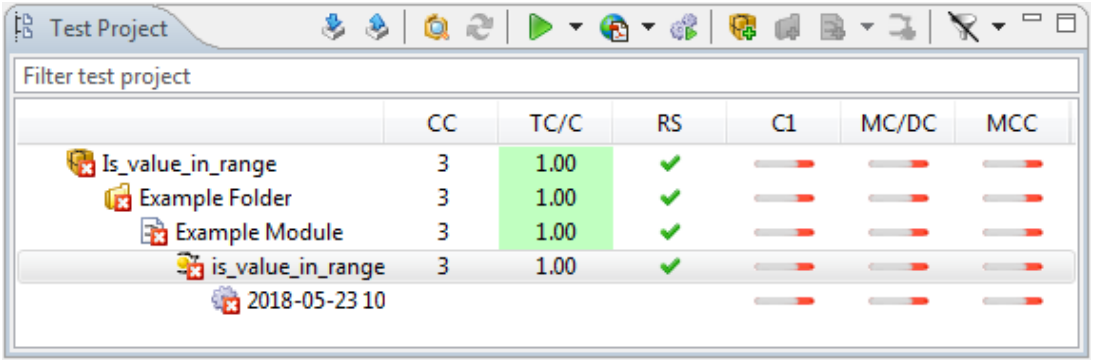


Figure 6.10: Test Project view within the Overview perspective

6.2.2.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Imports files (type depends on the selection).
	Exports files.
	Synchronizes a module.
	Analyzes the C-source file(s) of the module.
	Executes the test (Ctrl + E).
	Generates various test reports. The test details report for a test object will be generated as default (Ctrl + R).
	Defines a batch test.
	Inserts a new test collection.
	Inserts a new folder (Shift + Ins), optional for organizing your test project.
	Inserts a new module (Ins). Modules will contain the test objects available within the C-source files to be tested, i.e. C functions.
	Inserts a new task.
	Inserts a new test object (Ctrl + Insert).
	Hides irrelevant test objects.

Table 6.6: Tool bar icons of the Test Project view

6.2.2.2 View icons

Iccon	Meaning
	Test collecion containing folder(s) (optional) and module(s).
	Folder containing module(s) (optional).
	Module containing the test object(s).

continue next page

Icon	Meaning
	Module containing a comment, a description or a specification.
	Variant module that was created with the “Create Variant” option.

Table 6.7: View icons of the Test Project view

6.2.2.3 Status indicators

Indicator	Status
	Some interface settings need manual inspection (e.g. undetermined array sizes).
	The test object was analyzed but has no test case.
	The test object has test cases but no data.
	At least one test case has any data.
	At least one test case is ready to be executed.
	The test object interface has changed. A reuse operation within IDA is required.
	The C-source of the test object has changed. A new function has been added.
	The test object has been removed. You still see the object, but there is no operation possible. Only displayed when the test object contained any test cases before the removal.
	The test object is suspicious.
	The test execution has been aborted for this test object.
	The test object contains a comment, a description or a specification.
	The test result of a test run is failed. This may be either due to a mismatch of actual and expected results or if the coverage did not achieve the minimum coverage.

continue next page

Indicator	Status
	Test results and coverage of the test run are both OK.
	The coverage did not achieve the required minimum coverage. The red part of the bar indicates the missing percentage of coverage, e.g. more red means less achieved coverage.
	The coverage achieved the minimum coverage, but the minimum coverage was less than 100.

Table 6.8: Status indicators of the Test Project view

6.2.2.4 Creating tests and reviews

Creating the test

You need at least one test collection to organize your test, and within at least one module and one test object. Folders and further test collections are optional and just have the purpose to organize your test project.

Important: We recommend to do any changes of the test project structure within the Test Project view of the Overview perspective. The view layout of this perspective is optimized for this purpose!

→ Click on the icon (New Test Collection) on the tool bar of the view.
→ Enter a name and click “OK”.
→ Click on (New Folder), enter a name and click “OK”.
→ Click on (New Module), enter a name and click “OK”.

Modules need to be created for each of the source files that shall be tested with either a unit or integration test. After the module analysis, the module lists the testable functions of its source files as test objects.

Tasks

The task element provides means to protocol reviews or external tests and link them to requirements. This allows full verification coverage of requirements that are not testable with a normal unit or integration test. A task can be created within a test collection or folder.

→ In the Test project view right-click on a test collection or folder.

→ Click on “New Task” in the context menu.

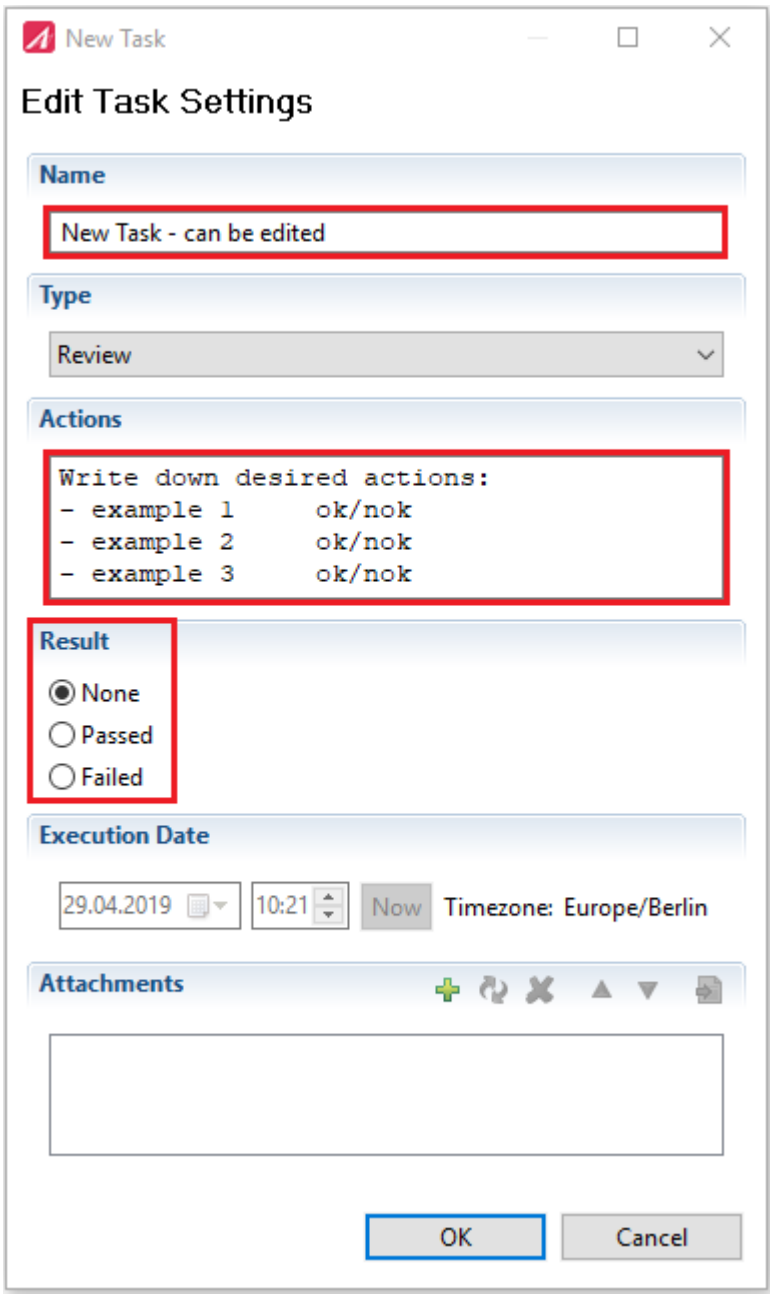


Figure 6.11: Editing the Task settings

You can edit the task name, choose the type (review or test) and write down the desired actions to be performed. Further types of tasks can be defined within the tasks preference page.

Tasks also have a result that shall be set after the task actions have been completed. Choose the execution date in order to track the completion of the task actions.

You can also attach files in PDF, ASCII text or image formats as a documentation of the review process. The contents of those files will be appended to the test details report of a task element (e.g. scanned check lists or filled PDF forms).

Executed tasks are displayed with their result within the Test Project view.

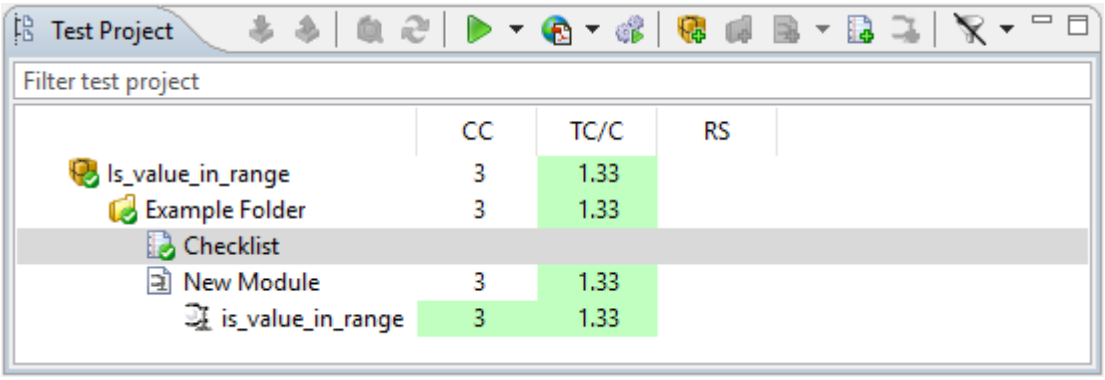


Figure 6.12: Executed task “Checklist” (Passed) in the Test Project view

When linking tasks to requirements, each task result counts as one test result compared to the test case results of normal unit testing. Refer to section 6.4.7

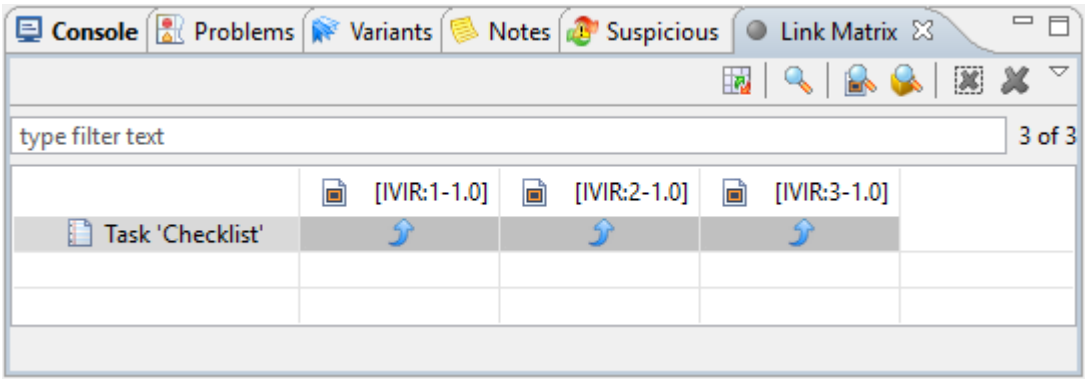


Figure 6.13: Task “Checklist” linked to multiple requirements in Link Matrix

For more information about the handling of the Link Matrix please refer to section 6.4.7.

6.2.2.5 Creating new test runs

By default test runs are overwritten by every new execution of the test. This behavior can be changed within the Preferences:

→ Click “Window” in the menu bar.

→ Select “Preferences” in the pull down menu.
The Preferences window opens.

→ Select “Test Execution Settings”.

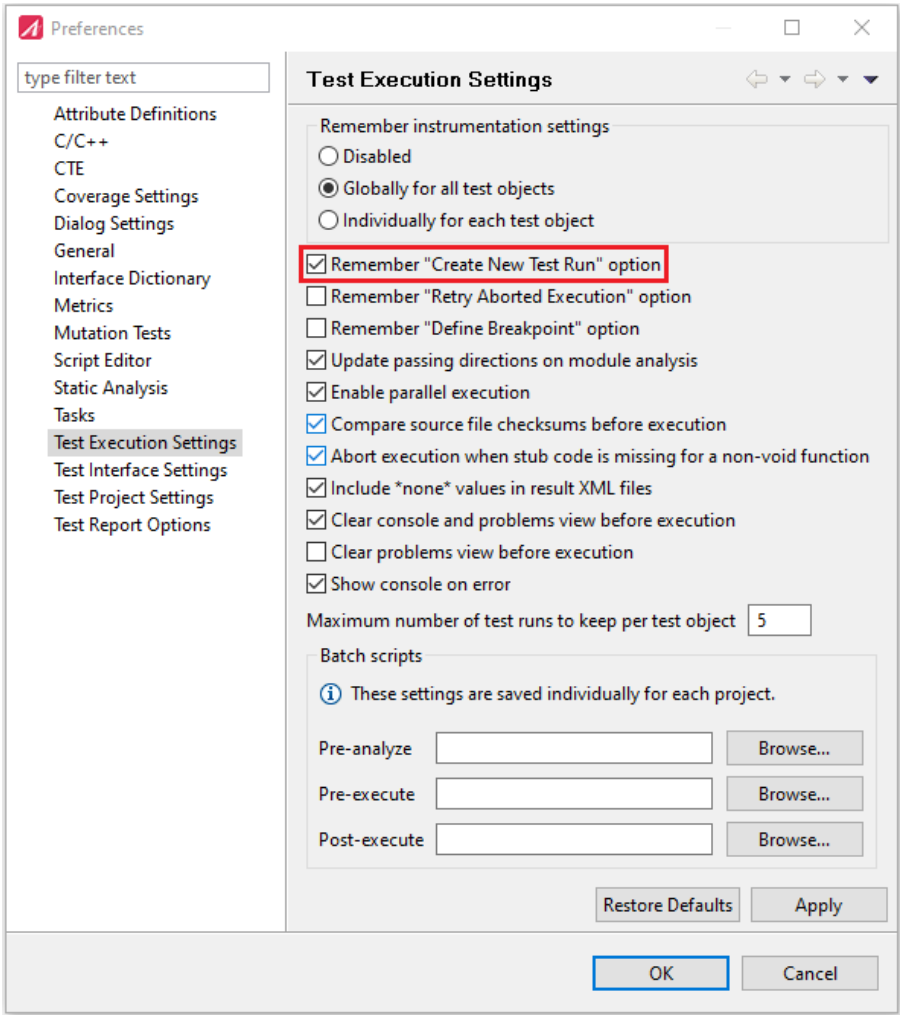


Figure 6.14: Test Execution Settings in the Preferences

→ Select “Remember “Create New Test Run””.

→ Click “OK”.

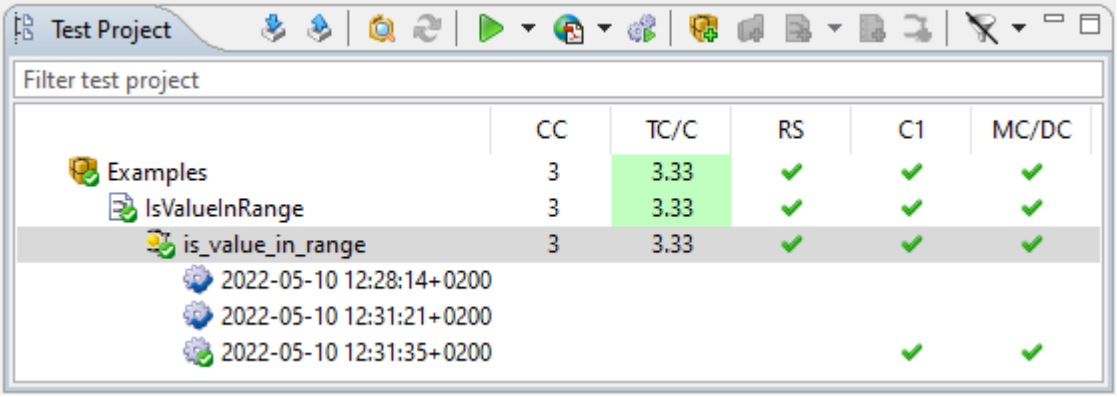


Figure 6.15: Test Project view with historical test runs

Historical test runs

Indicator	Status
	Test run completed and closed, saved as historical, will not be overwritten.
	Latest test run passed.
	Test run is suspicious, needs to be executed again.
	Test run failed.

Table 6.9: Status indicators for historical test runs

6.2.2.6 Analyzing modules

Important: To analyze a module it is necessary to select an execution environment, add at least one C-source file and required include paths for header files and add defines necessary for analyzation/compilation of the source file(s), see section 6.2.3 Properties view. If you need to learn more about this workflow, have a look at the Tutorial: Practical exercises.

To analyze a module (the C-source file),

Analyzing the C-source file

→ in the tool bar click on (Analyze Module) to start the module analysis.

TESSY now analyzes the C-source file, this will take a few seconds.
After successful processing,

→ click on the white arrow in front the module: ExampleModule.

TESSY will as well analyze the C-source file by just clicking on the white arrow next to the module after adding the C-source file.

Now all functions which were defined in the C-source file are displayed as children of the module above within the Test Project view (see figure 6.16).

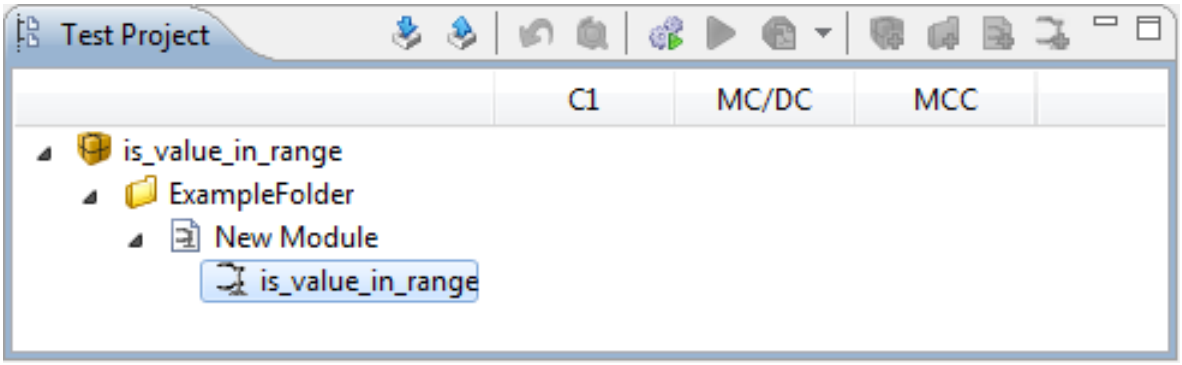


Figure 6.16: The function of the C-source is displayed as child of the module

When using the CLANG parser (which is the default since TESSY 4.0), opening older modules created with TESSY 3.2 and before will show the following warning:

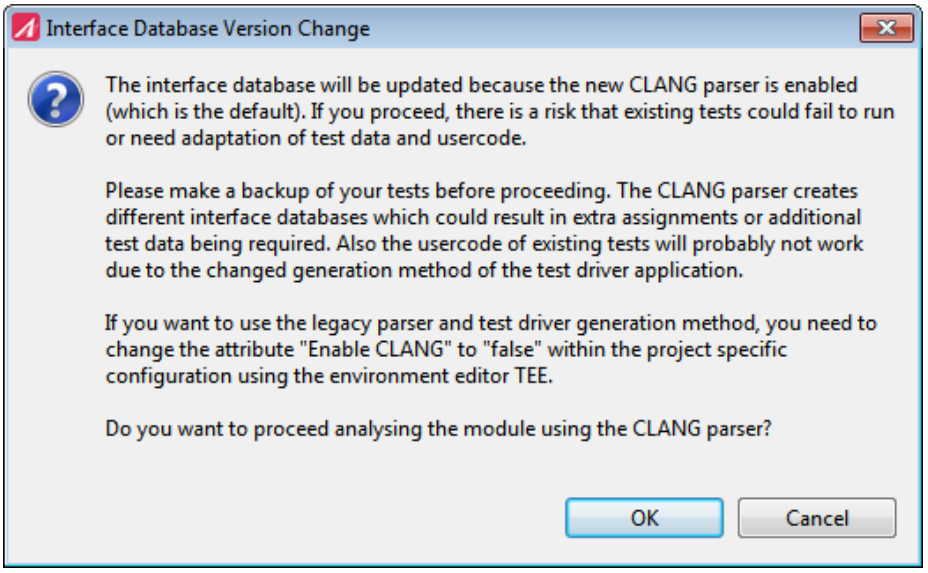


Figure 6.17: Warning when opening older modules with CLANG enabled

The following options are available in such a case:

- If you want to convert the module and start working with the CLANG mode, select “OK” which will do all necessary conversion. Refer to the restrictions that apply: Operating limits. Please create a backup of your project before converting the modules.
- If you have a legacy project and just want to execute tests, select “Cancel” and change the attribute “Enable CLANG” to false within your project configuration using the TEE Adjusting enabled configurations. If you did this successfully, there will be no more warning when opening the module.

When working with the CLANG mode there are several parser options which can all be set in the TEE:

Parser option	**Description**
Enable Create Default Constructors	If set to true, the TESSY parser creates a default constructor if it is missing.
Enable Create Function Stubs	If set to true, external functions that are called are by default marked to create stub code unless they are listed in attribute “Function Stub Exclude List”.
Enable Create Method Stubs	If set to true, undefined called methods are marked to create stub code unless they are listed in attribute “Method Stub Exclude List.”
Enable Define Variables	If set to true, external variables that are used are marked to be defined unless they are listed in attribute “Variable Exclude List”.
Enable Exceptions	If set to true, TESSY enables exceptions for the test object and generates a try-catch block around it. Also the TIE displays an artificial global variable called throws exception which can be set to OUT in order to test an exception thrown be the test object. By default the attribute is set to true.
Function Stub Exclude List	The comma separated list of functions is excluded from automatic stub creation.
Method Stub Exclude List	The comma separated list of methods is excluded from automatic stub creation.
Variable Exclude List	The comma separated list of variables is excluded from being automatically defined.

Table 6.10: CLANG parser options and descriptions

General information about editing the environment can be looked up in the section TEE: Configuring the test environment.

Important: Setting changes of the parser options made in TEE will be effective when analyzing a module. Some of the options only apply when initially analyzing modules. In this case it is necessary to reset a module before starting the analysis to see the effects of the latest changes.

For more information about the parser options and more attributes available within the environment editor TEE please refer to the application note “Environment Settings (TEE)” in TESSY (“Help” > “Documentation”).

6.2.2.7 Static code analysis and quality metrics

Cyclomatic complexity for complexity control

Calculation of the cyclomatic complexity (McCabe metric, displayed as CC) is a common measure for complexity control. It measures the complexity of source code on the basis of the control flow graph and indicates the number of linearly independent paths through the code.

If decisions within the code have more than one atomic condition (e.g. “if (A && B)”), the cyclomatic complexity will be incremented by one for each additional atomic condition within such decisions.

The metric increases linearly with the number of binary decisions within a program, however e.g. calculations are not taken into account. McCabe indicates 10 as the highest acceptable cyclomatic complexity measure which means that values higher than 10 suggest a software review.

Important: Values can only be calculated for environments working with CLANG, therefore ”Enable CLANG“ needs to be set ”true“ in the TEE. (For more information about the TEE see chapter 6.5 TEE: Configuring the test environment.) Other environments are not included in the calculation of metrics.

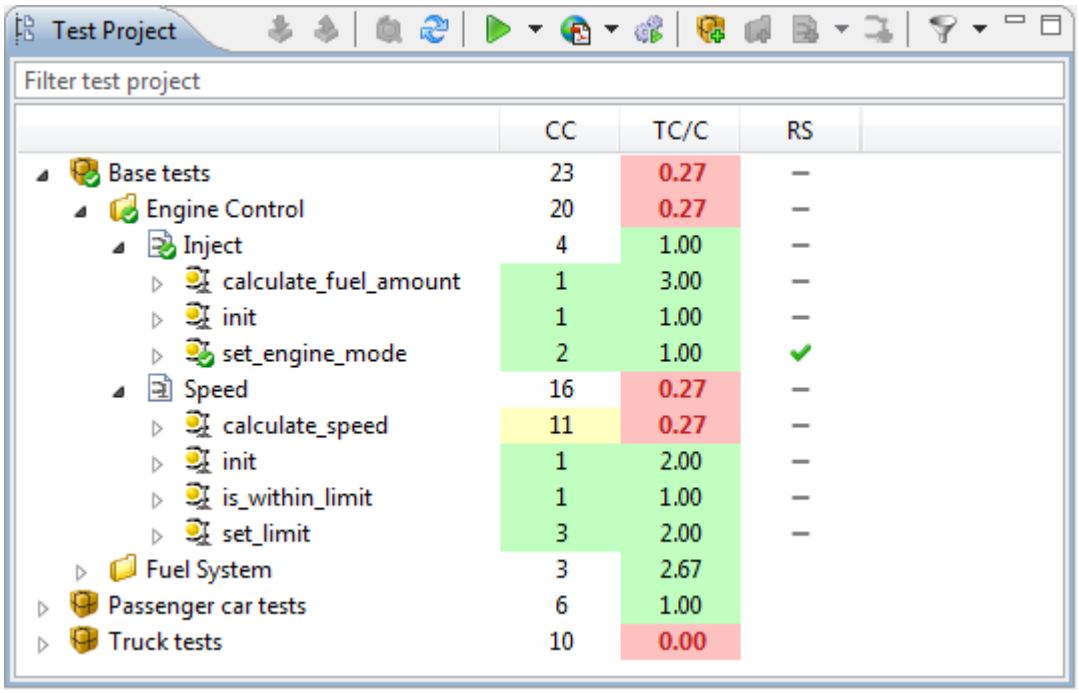


Figure 6.18: Static code analysis in the Test Project view

TESSY determines the cyclomatic complexity value for each test object on module, folder and test collection level using the sum of all values (displayed as CC - Total Cyclomatic Complexity). Optionally you can also display the average value (CC Avg) or the maximum value (CC Max).

Within the preferences it is possible to set two threshold values as limits which will be highlighted in yellow (warning) or red (error). Values below the warning limit will be marked in green.

Also important is the relation between the number of test cases and the complexity. For this purpose TESSY provides the test case to complexity (TC/C) ratio. This measure indicates if there are enough test cases available to have at least one test case for each linearly independent path. As a result you will most probably reach 100% branch coverage for a TC/C ratio greater than 1.

- A value smaller than 1 indicates that not enough test cases have been created to pass through all linearly independent paths. (The value appears highlighted in red.)
- A value greater or equal to 1 indicates that at least a minimum number of test cases has been defined. (The value appears highlighted in green.)

Important: The TC/C ratio is only a hint for the necessary number of test cases to achieve 100% branch coverage. Depending on the code to be tested there may be less or even much more test cases necessary to do a full functional test. Also this measure does not take test steps into account because test steps are only seen as helper steps to prepare a test object for the actual test.

The result significance (RS) reveals weak test cases. This measure is available after the test execution and it verifies that each test case applies at least one of the following checks:

- Has some expected results other than *none*.
- Checks the call trace.
- Uses evaluation macros.

If none of the above checks are made, the RS measure of the respective test case will be marked as failed, otherwise it will be marked as passed (after running a test).

6.2.2.8 Testing effort estimation and tracking

TESSY provides a testing effort estimation based on a customizable formula. This formula can be edited within the metrics preference page as well as warning and error level thresholds. The actual time spent for testing a test object can be tracked within a separate column of the test project view.

Both testing effort columns need to be enabled within the metrics preference page to become visible within the test project view (see figure 6.19).

The metrics preferences can be found in the Window menu of the TESSY Menu Bar.

More information about the basic settings of TESSY is provided in section 6.1 Menu Bar Entries: Setting up the basics.

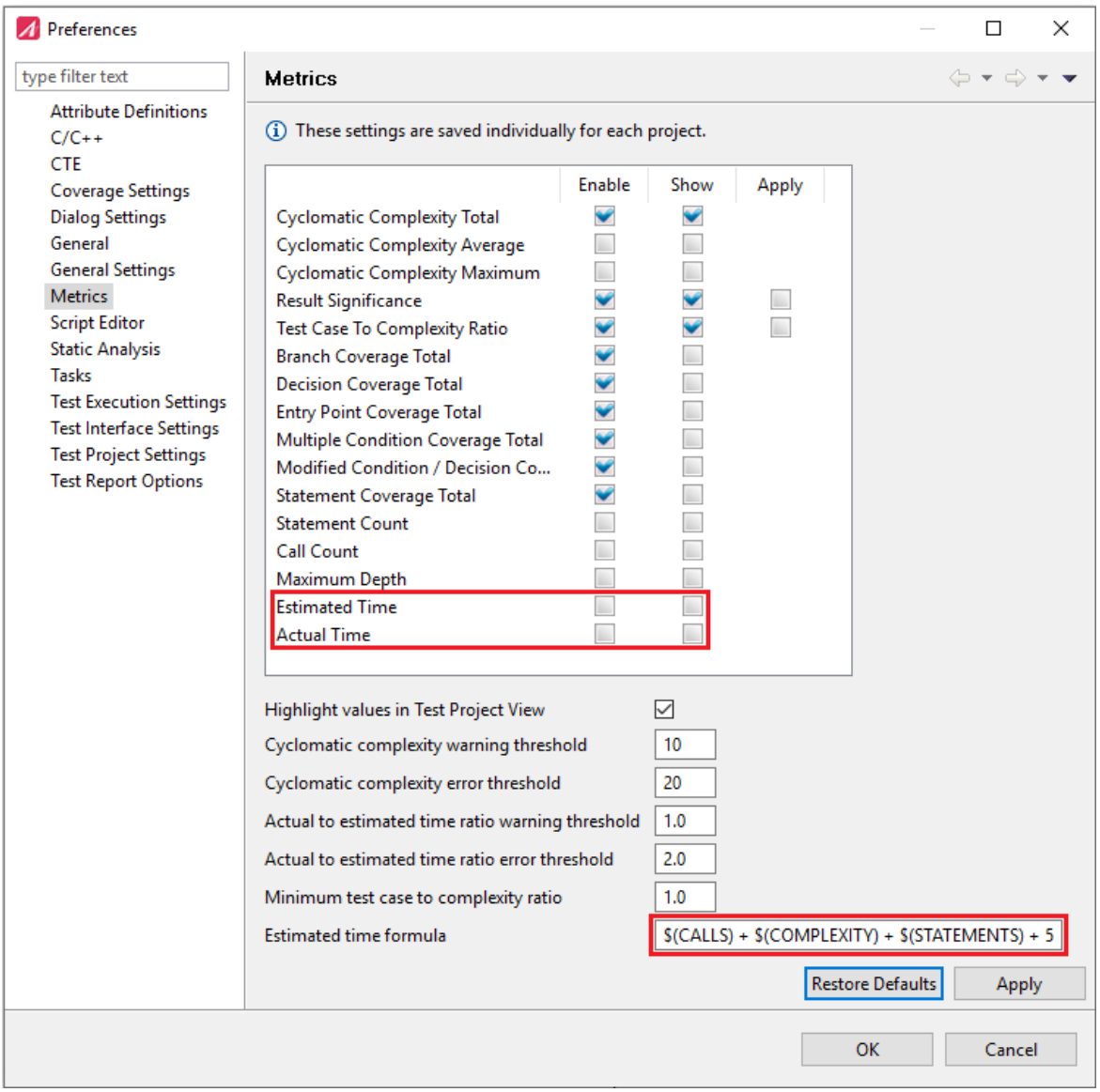


Figure 6.19: Select “Estimated Time” and “Actual Time” in the Preferences

The time estimation is an important topic for project management. In order to perform a realistic time and cost estimation for testing, the following basic conditions should be considered:

- Processes and organization of the company
- Experience of the test team
- Desired focus/level/depth of testing
- Maturity of the specification and project documentation
- Number of functions to test
- Number of test iterations (for regression testing)

The default formula prepared in TESSY is only a proposal and should therefore be reviewed and adapted for each project. The formula for the estimated time can be edited according to your needs. There are several predefined tokens available that represent values of the available metrics provided by TESSY.

The table below lists all available tokens:

Token	**Metric value**
$(COMPLEXITY)	Cyclomatic Complexity Total
$(C1_TOTAL)	Branch Coverage Total
$(DC_TOTAL)	Decision Coverage Total
$(MCC_TOTAL)	Multiple Condition Coverage Total
$(MCDC_TOTAL)	Modified Condition/Decision Coverage Total
$(C0_TOTAL)	Statement Coverage Total
$(STATEMENTS)	Statement Count
$(CALLS)	Call Count
$(MAX_DEPTH)	Maximum Depth

Table 6.11: Predefined tokens of the available metrics

If the testing effort columns are enabled within the preferences, the test project view will show the calculated estimation time for each test object after analysis of the module. Whenever the module will be analyzed again, the estimated time will be updated based on the defined formula.

The actual time can be edited within the "AT" column for each test object using the inline editor. Values entered are interpreted as minutes but you can also explicitly specify the unit, e.g. `30m` for thirty minutes or `1h` for one hour. Also combined values are possible, e.g. `1h20m`.

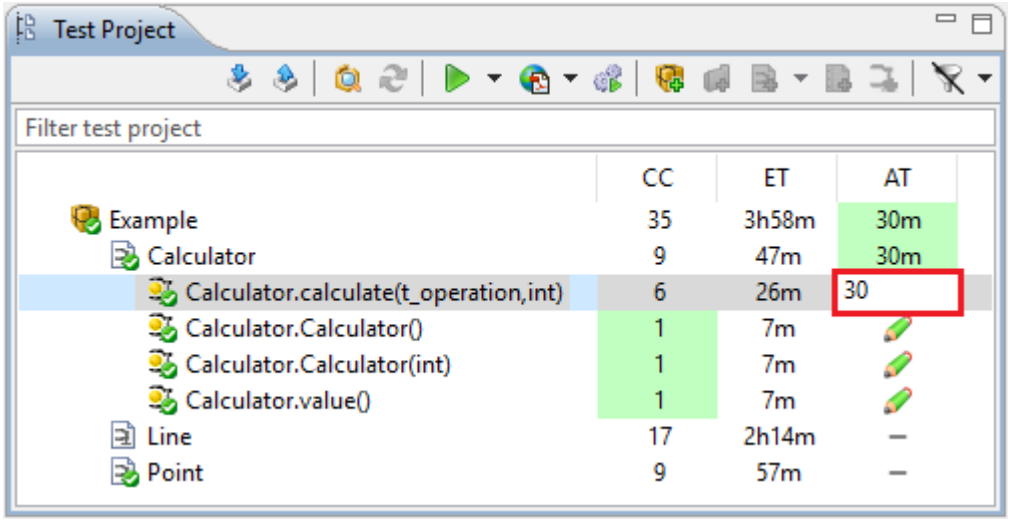


Figure 6.20: Editing the actual time within the AT

Both testing effort values are cumulated for modules, folders and test collections. The time being displayed will be rounded to avoid too long values exceeding the column width. All testing effort values will be listed within the overview report as part of the metrics table.

6.2.2.9 Creating variant modules

Module testing of software variants often require very similar tests that only differ in small parts. Therefore the comfortable reuse and adaption of existing tests reduces the testing effort for each variant of the software.

TESSY provides a variant management for test modules so that basic tests within a base test module can be inherited, altered or removed and additional tests can be added by sub modules covering the test of each software variant.

A base module serves as parent for all variant sub modules. It contains the information that will be shared with all variants. Any module can be used as base module (i.e. can be the parent of sub modules) if the parent module has changed.

If you have i.e. created a module with some test cases, you can create a variant:

→ Create a new folder i.e. with the name “Variant 1”.
→ Right-click on the folder and select “New Variant Modules” from the context menu (see figure 6.21).

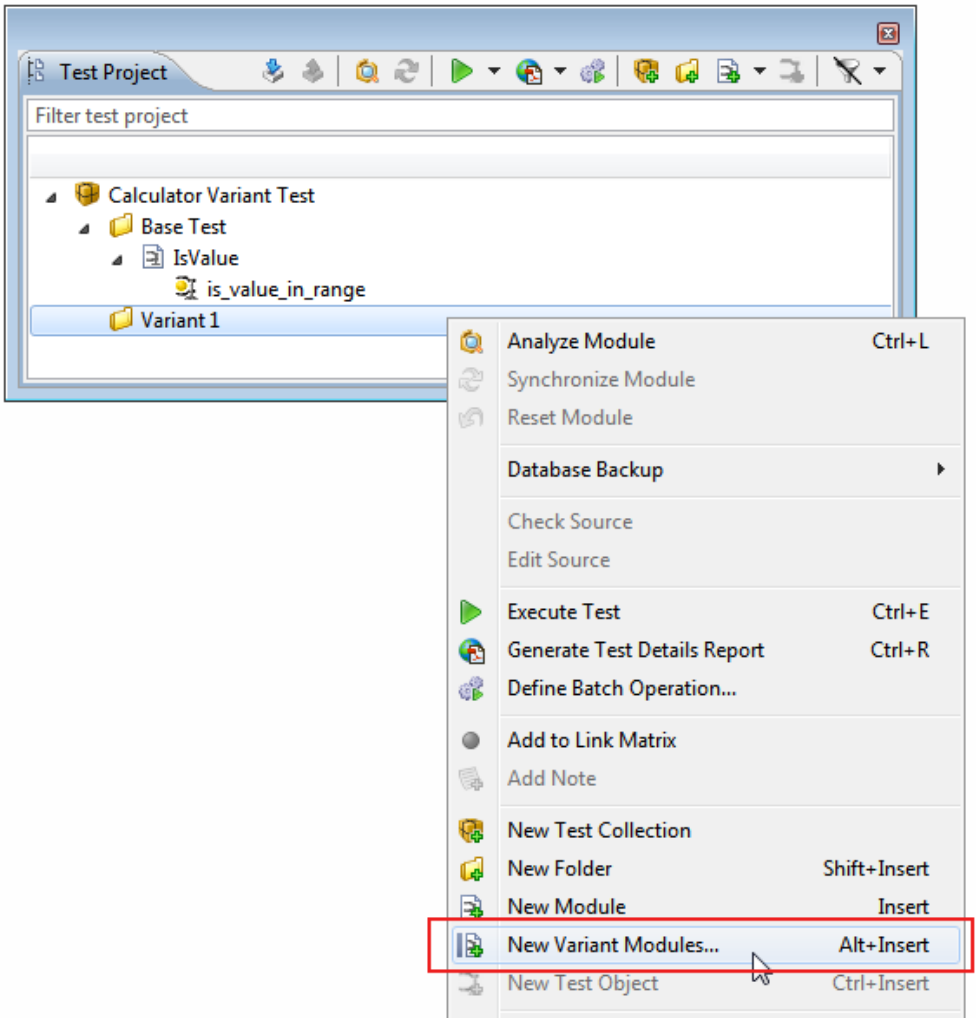


Figure 6.21: Create Variant Modules

→ Choose the parent module and click on “OK” (see figure 6.22).

→ The variant will be displayed within the Test Project view with the icon .

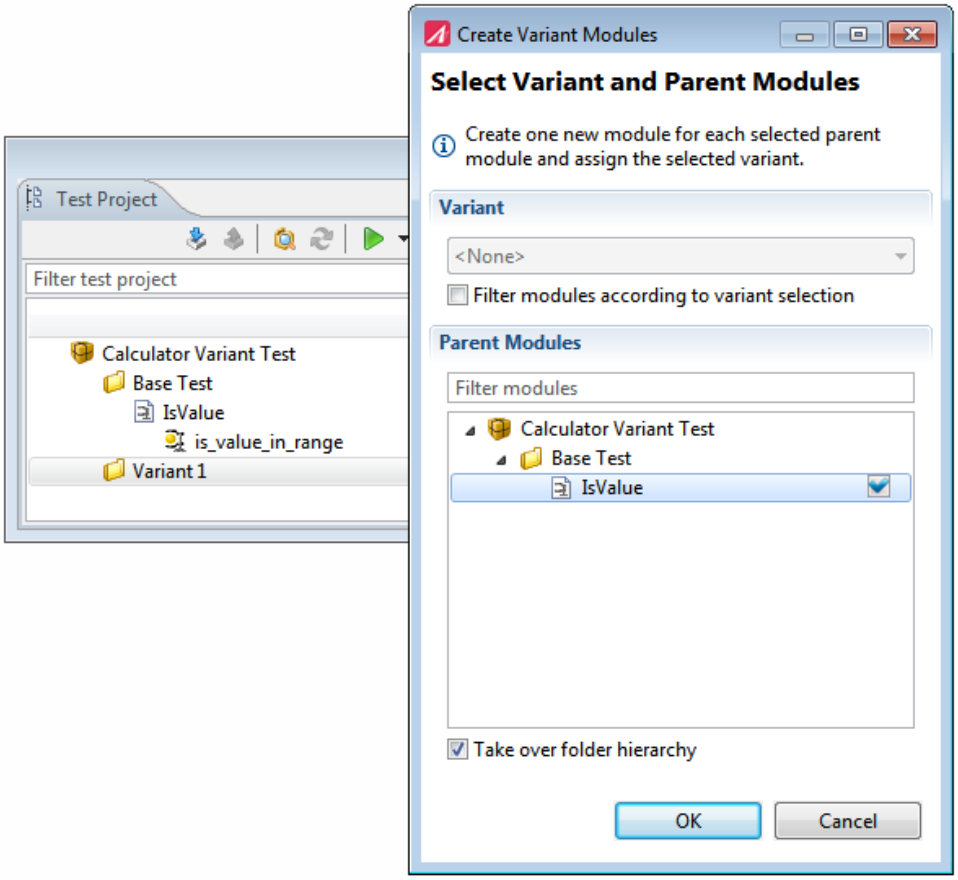


Figure 6.22: Selecting the parent module of the variant

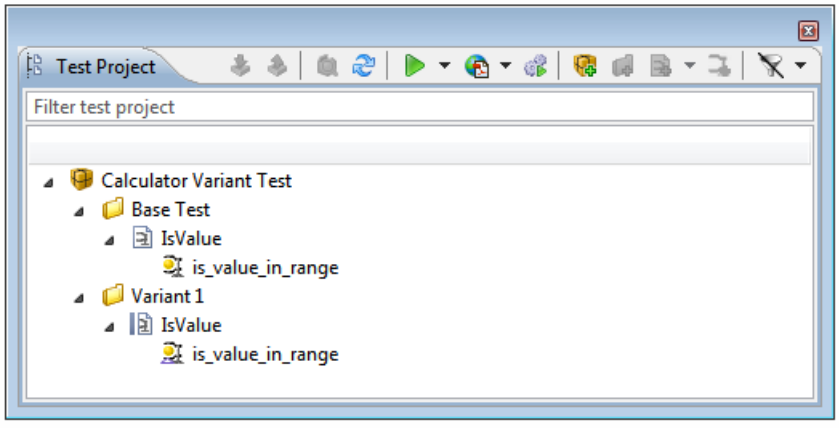


Figure 6.23: Test Project view with a module and a variant module.

If the parent module has changed, TESSY will mark the children with an exclamation mark. The mouse over states, that the module needs to be synchronized with its parent (see figure 6.24).

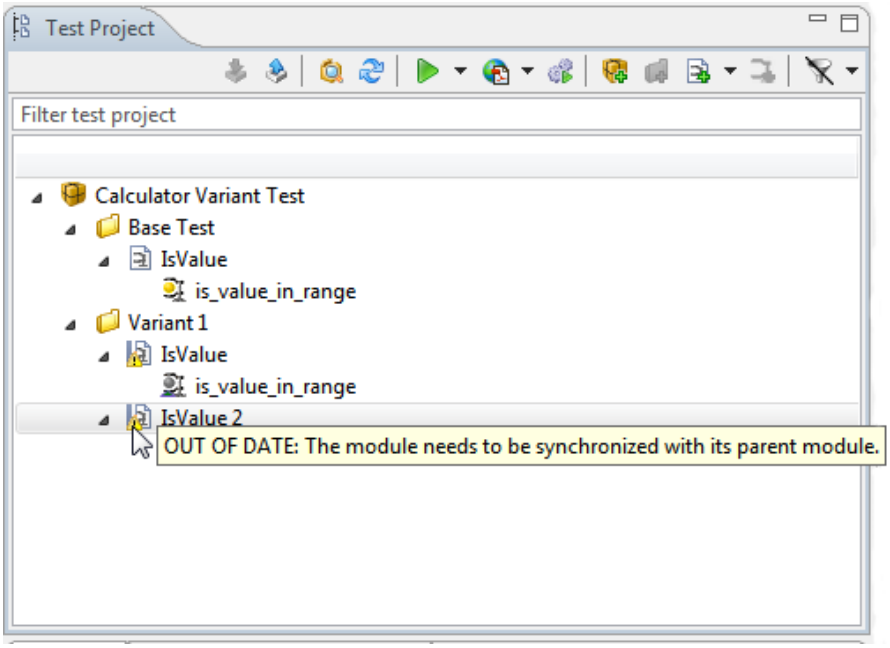


Figure 6.24: The variant module needs to be synchronized with the parent.

To synchronize the module with the parent:

→ Right-click the module and select “Synchronize Module” from the context menu.

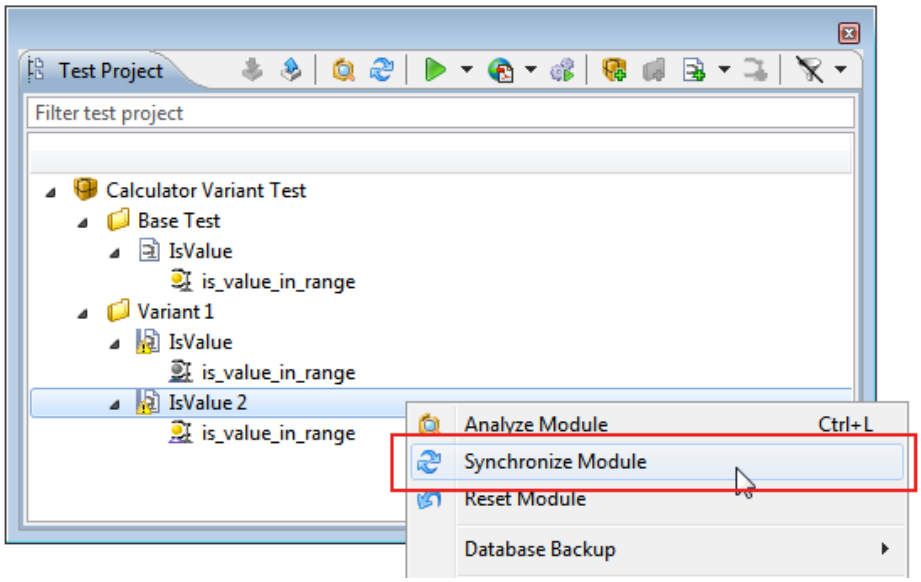


Figure 6.25: Synchronizing a module with the parent

The Synchronize Module dialog is displayed. The parent and all child modules will be shown and can be synchronized in one step.

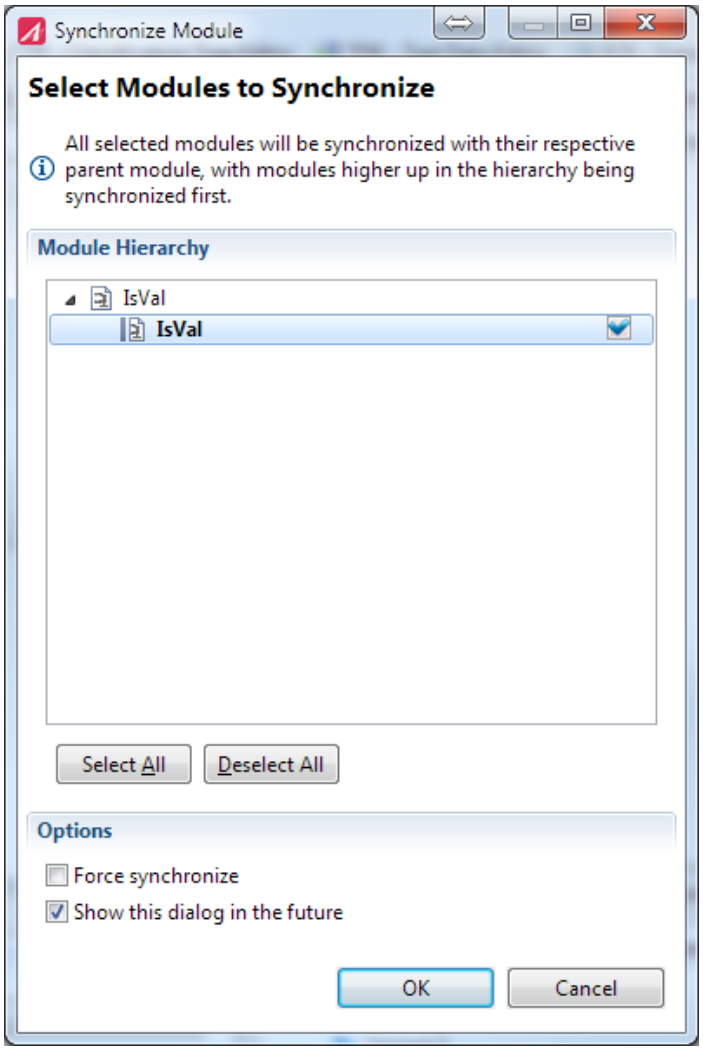


Figure 6.26: Synchronize Module dialog

For the first synchronization or if a modules interface has changed, the IDA will be opened to assign the parent modules interface to the child module interface. You will be asked if the IDA perspective shall be opened:

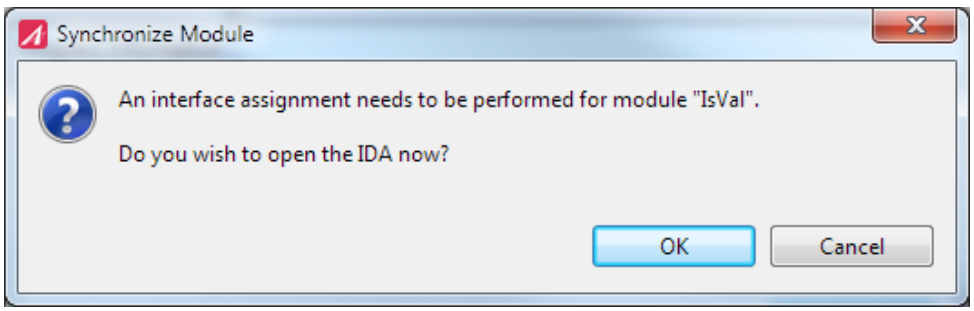


Figure 6.27: Interface assignment needed

The assignments made within IDA will be saved for future synchronizations of the child module(s). When invoking the “Synchronize Module” operation again (e.g. after changes to the parent module), the last used assignments will be applied without showing the “Synchronize Module” dialog.
If you want to force a new assignment of the parent module interface to the child module interface, you can analyze the child module which will reset the interface assignments. A subsequent module synchronization will then show the IDA again.

The following indicators display the status of the inherited test cases and test steps within the Test Items view:

Indicator	Status / Meaning
	The small triangle indicates that the test case or test step is inherited.
	The filled triangle indicates that the inherited data of the test case and test step was overwritten.
	The inherited test case or test step was deleted.
	The test case or test step was added for this variant test object.

Table 6.12: Status indicators of inherited test cases or test steps

Important: Deleted test cases/steps are only faded out within the child module. They can be made available again using “Restore Deleted” from the context menu.

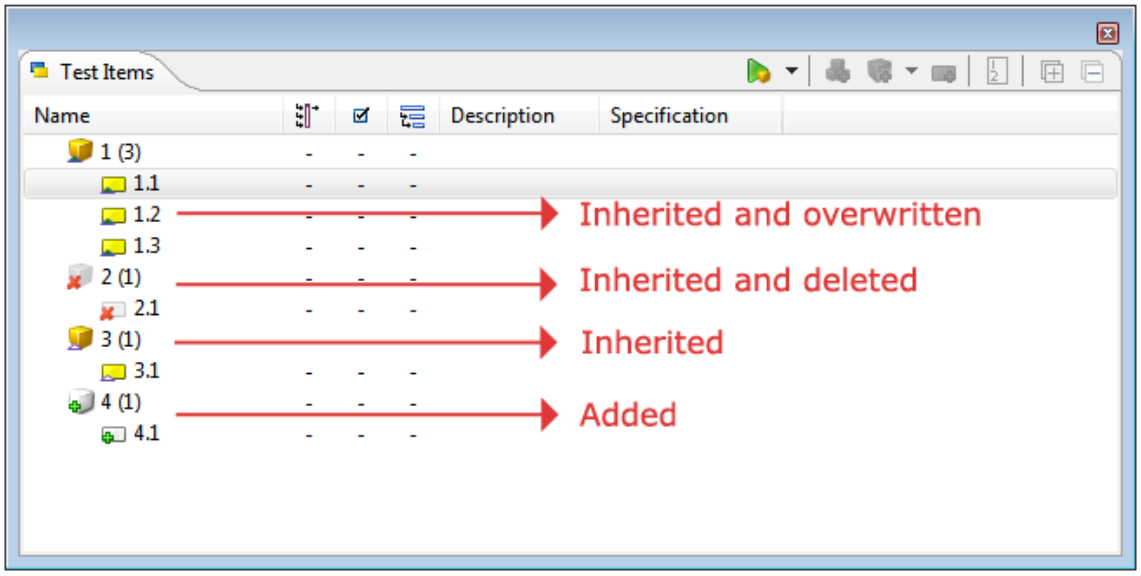


Figure 6.28: Test cases and test steps that were inherited of a variant module

6.2.2.10 Notes

In addition to specifications, descriptions and comments added or modified in the Properties view Notes can be added using the pull down menu of the respective test module, test object or test case.
Later added Notes to already executed tests will appear in the Test Details Report and in the Test Overview Report after generating new reports.

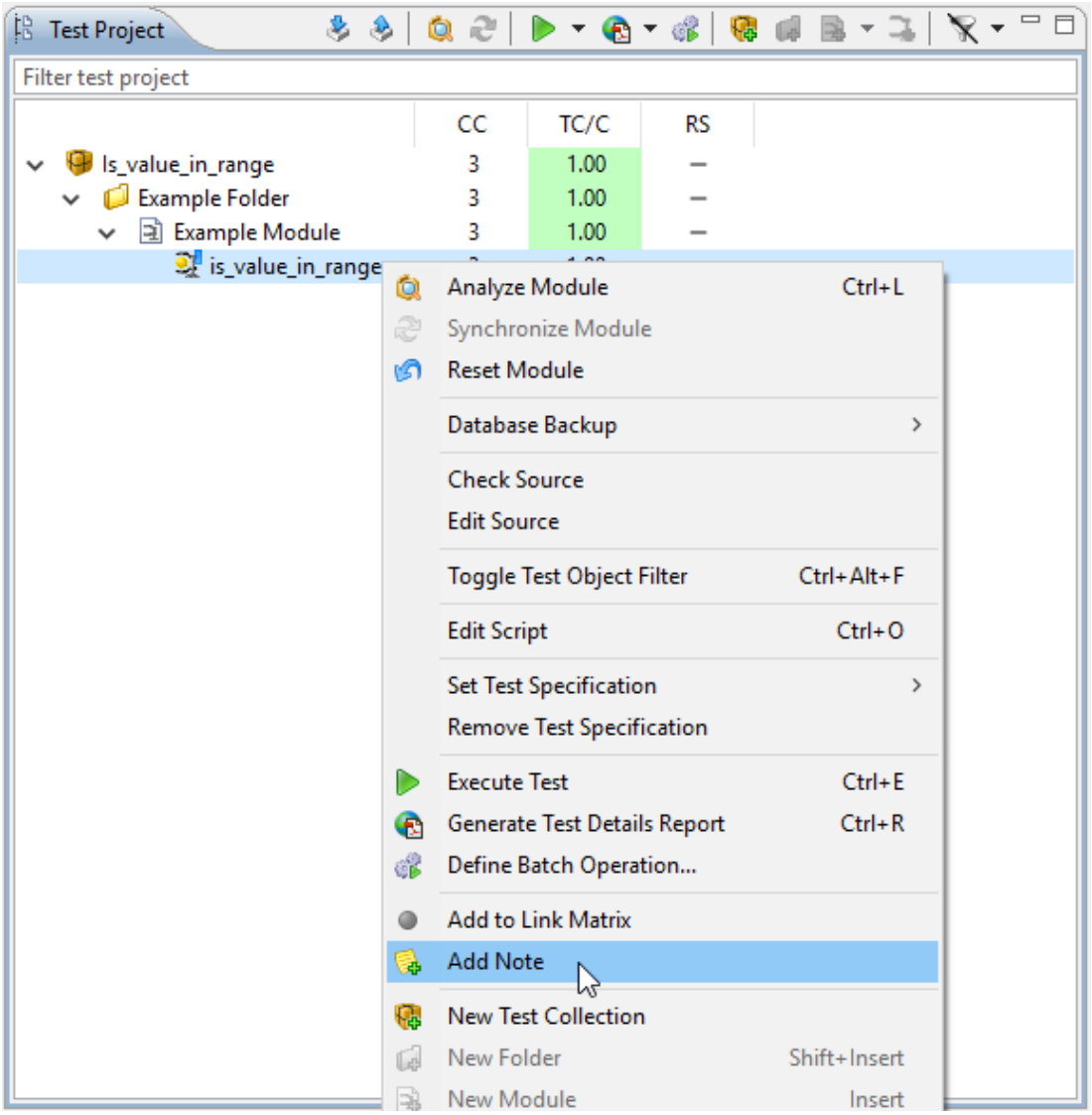


Figure 6.29: Add notes via content menu

Added Notes will appear in the Notes view on the lower right of the Overview perspective and can be edited or deleted there.

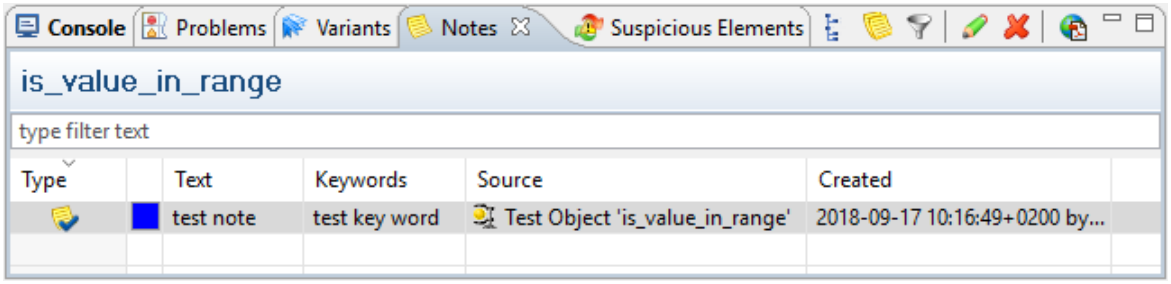


Figure 6.30: The Notes view in the Overview perspective

Right-click the respective note to open the pull down menu and choose the wanted action. You can also choose to create a Notes Report.

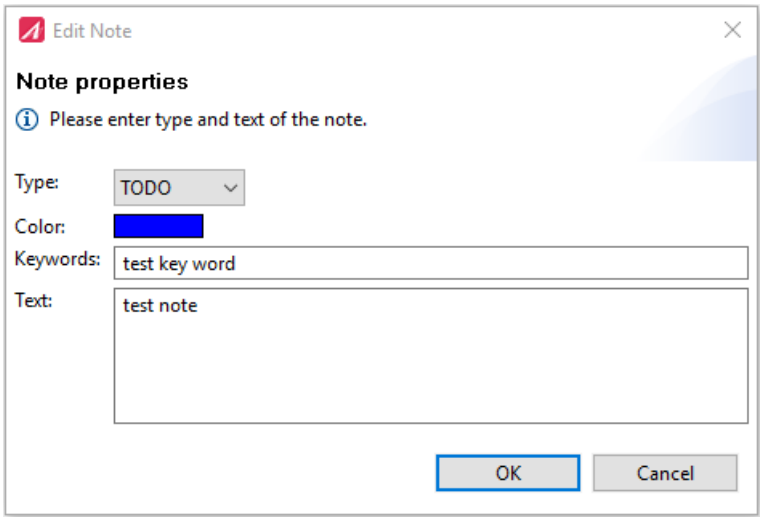


Figure 6.31: Editing notes in the Notes view

Notes will appear in the Test Details Report as well as in the Test Overview Report without any test execution. Notes will not appear in Planning Coverage Reports and Execution Coverage Reports.

6.2.2.11 Executing tests

After entering test data for a particular test object you are ready to execute the test. During this process, TESSY will perform the following steps:

- Generate the test driver based on the interface information and user code provided.
- Link the test driver to the test object to create an executable file.
- Run the test with the selected coverage instrumentation.

Generating the test driver

The test driver is necessary to call the function under test and will be generated automatically. Test driver and the function under test form a complete (embedded) test application, including the startup code for it, and will use an appropriate compiler for the particular embedded microcontroller architecture. If the function under test uses external variables that are not defined, the test driver generated by TESSY can define those variables.

Once the test driver has been compiled, it can be run as often as required. You can select a subset of your test cases and run the test again by just selecting the run option. Changes to test data and expected results might require building a new test driver. TESSY will check this automatically and generate a new driver.

Stub functions

If the function under test itself calls other functions (subroutines), TESSY can provide replacement functions (stubs) for the missing subroutines with the test driver. TESSY features two types of stub functions:

- Stub functions for which you may provide the C source code for the bodies of the stub functions.
- Stub functions for which TESSY is able to check if the expected value for a parameter is passed into the stub function and for which TESSY is able to provide any value specified by the user as return value of the stub function.

Executing the test

To execute a test:

→ Click on the arrow next to the icon Start Test Execution in the tool bar of the Test Project view.

→ Click on “Edit Test Execution Settings”.
A progress dialog will be opened (see figure 6.32).

→ Choose the desired options and click “Execute”.
A progress dialog will be shown while TESSY generates, compiles and links the test driver and runs the test. This will take a few seconds.

Meaning of the Test Execution Settings:

Action	Meaning
“Force Check Interface”	An analysis of the interface is forced.
“Force Generate Driver”	Usually TESSY recognizes if the driver has to be generated. In this case it can be forced.
With “Run” (checked)	The test will be executed.
Without “Run” (unchecked)	Only the test driver will be generated.
“Abort On Missing Stub Code”	Building the test driver application will be aborted with an error if there are non-void stub functions without any code provided to return a value. You can uncheck this action to ignore this error if you are sure that the return values of your stub functions are not used. (For more information please refer to Defining stubs for functions.)

Table 6.13: Test Execution Settings - Actions

Option	Meaning
“Create New Test Run”	A new history in the module will be generated.
“Test Cases Separately”	The download and execution process of the test driver will be started separately for each test case. This provides an initial state of memory (and variables) for each test case and is useful if the test cases shall be executed independently. The disadvantage of this approach is an increased execution time. (Due to start/stop of debugger and download of test driver.) It is recommended to set this option for dedicated test objects only.

Table 6.14: Test Execution Settings - Options

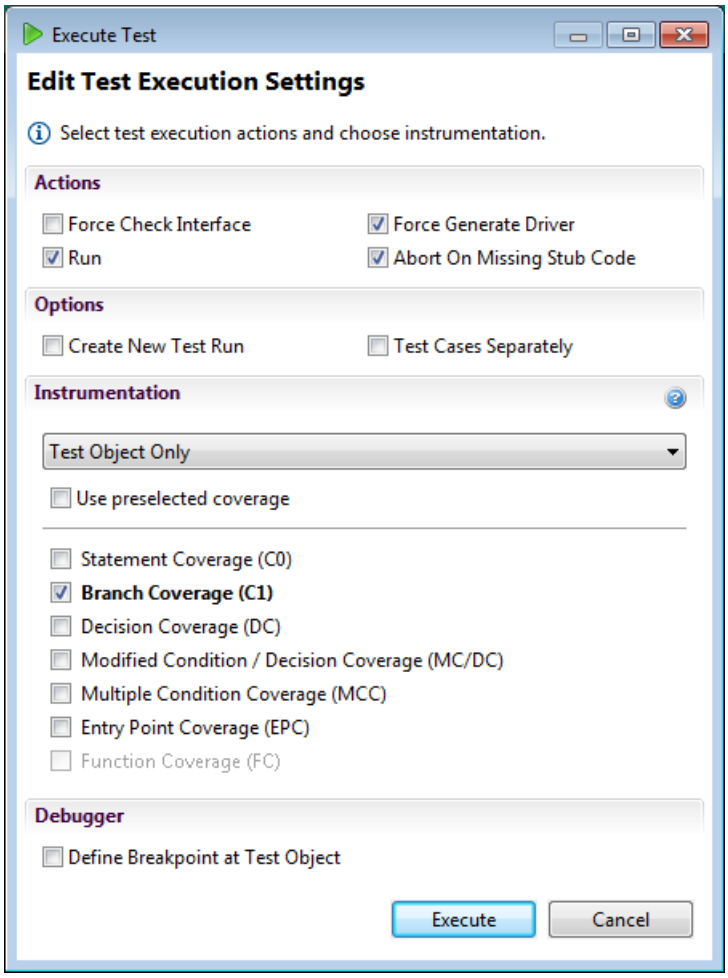


Figure 6.32: Test Execution Settings

6.2.2.12 Instrumentation settings

Within the Test Execution Settings dialog you can select the various possible coverage instrumentation for this test run (see figure 6.32):

- → elect from the pull-down menu if the coverage shall be used for the test object or the test object and the called functions.
- → Untick the box “Use preselected Coverage”.
- → Select the coverage instrumentation (more than one possible).
  The coverage instrumentation is now used for this test run, even if you have selected a different coverage instrumentation as default for your project (see section 6.1.2.1 Window > Preferences menu) or for the module or test object within the Properties view (see section 6.2.3.4 Coverage tab).

“Use preselected Coverage”

If you tick the box “Use preselected Coverage”, coverage selection will be applied according to the following rules:

- If a coverage selection is set in the Properties view (see section 6.2.3.4 Coverage tab), that selection will be used.
- If no coverage selection is set in the Properties view, but in the Test Execution Settings of the Window > Preferences menu the option “Remember instrumentation settings” is set, the last used selection will be used.
- If no coverage selection is set and the option “Remember instrumentation settings” is not set, no instrumentation will be used.

For more information about the coverage measurements refer to the application note “Coverage Measurement” in TESSY (“Help” > “Documentation”).

6.2.2.13 Viewing test results

Viewing test results

After a test run, the Test Project view gives an overview about the coverage, if selected:

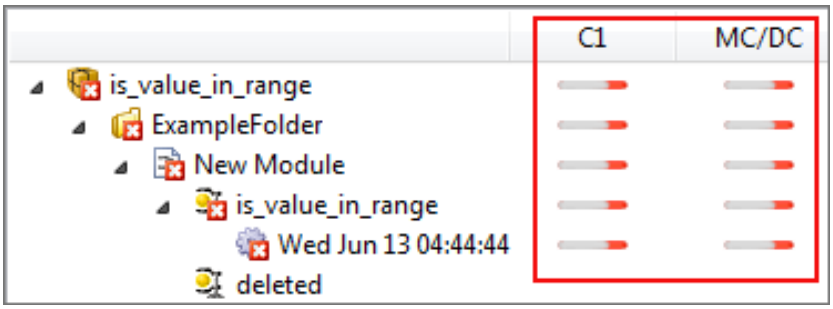


Figure 6.33: Coverage displayed within the Test Project view

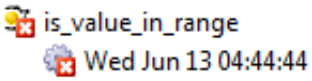
- The actual results will be compared with the expected values according to the evaluation mode. The result will be either failed or passed.
- The last step of test execution is the generation of an XML result file. This file contains all test data and the actual results. It will be used for reporting.

The results of every coverage measurement can be reviewed in the CV (Coverage Viewer) as soon as the test was carried out. For details refer to section 6.11 CV: Analyzing the coverage.

See 6.11 CV: Analyzing the coverage

Please notice the following habits:

- A green tick will indicate that all actual values comply with the expected values with respect to the evaluation modes and the coverage reached at least the minimum coverage.
- A red cross will indicate that either some actual values yield failed results or the coverage did not reach the minimum coverage.
- If the interface has changed, the test object will indicate changes with test readiness states (see Status indicators).
- The time of the test run is stated within the Test Project view:



Important: The results of the coverage measurement are also part of the test result for a test object, e.g. if all outputs yield the expected result but the coverage was less than the minimum coverage, the test result will be failed.

Warning: Using the option “Reset Module” from the context menu will delete the module with all test results!

6.2.2.14 Hiding irrelevant test objects

In the Test Project View test objects that are not relevant for the project can be hidden by setting a test object filter:

→ To modify an existing test object filter click on the arrow next to the button and select “Select Test Object Filter”.

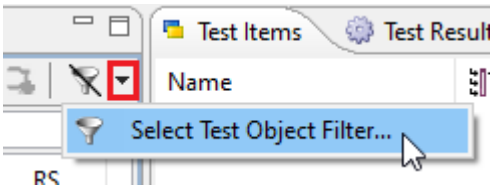


Figure 6.34: Opening the filter settings

→ The Filter Configuration dialog will open (see figure 6.35).

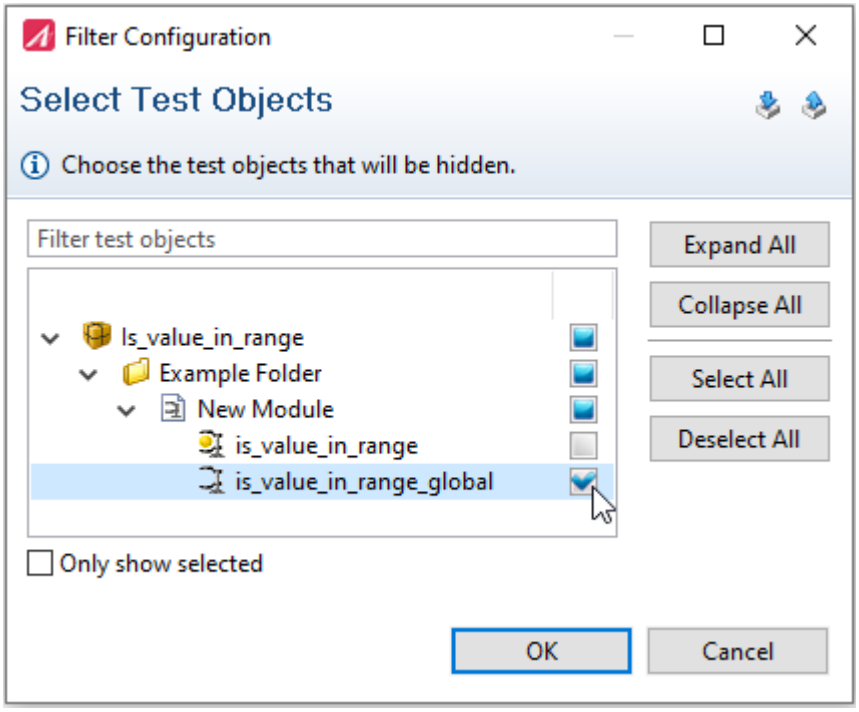


Figure 6.35: Filter Configuration Dialog

→ Select the test objects you wish to hide in the dialog that is shown.
→ Once a filter has been set, you can toggle it on and off by clicking on .

Important: The filter can only be applied to test objects without any test cases. Accordingly the Filter will automatically be removed for test objects that have at least one test case.

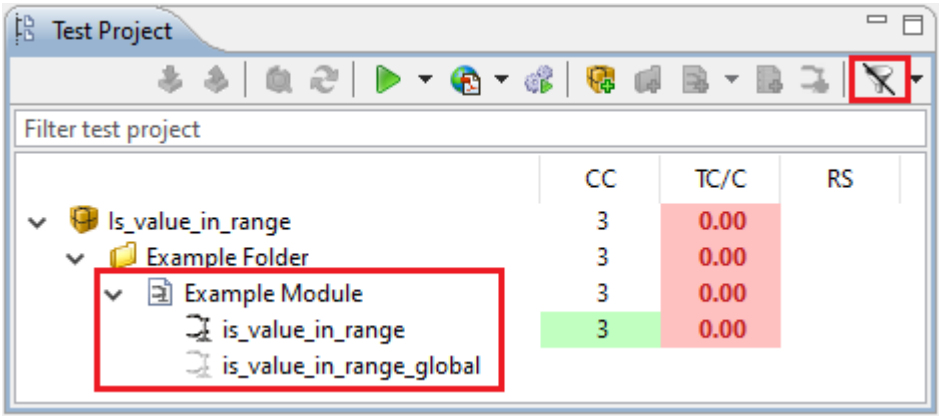


Figure 6.36: A filter has been set but is currently disabled (filtered test objects appear faded).

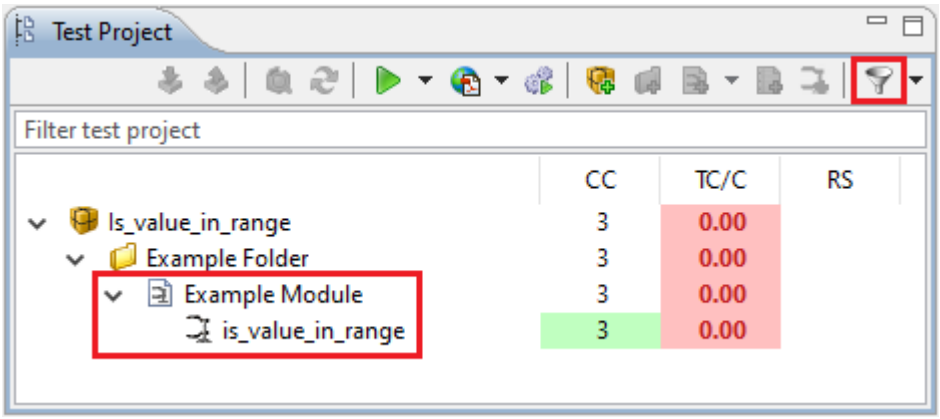


Figure 6.37: The Filter is enabled, the affected test objects are hidden.

Important: The filter setting will be saved for each filtered test object within the test database. When saving and restoring modules as TMB files, these filter settings will also be persisted and restored.

6.2.2.15 Search filter function

The search filter helps to find and select elements by their name.

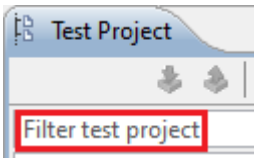


Figure 6.38: Search filter function of the Test Project view

Typing search terms into the search field will result in an updated Test Project view.

After a short delay only such matching elements and their ancestors and descendants are displayed. Elements will automatically be expanded to appear visible with the exception of modules that need to be analyzed.

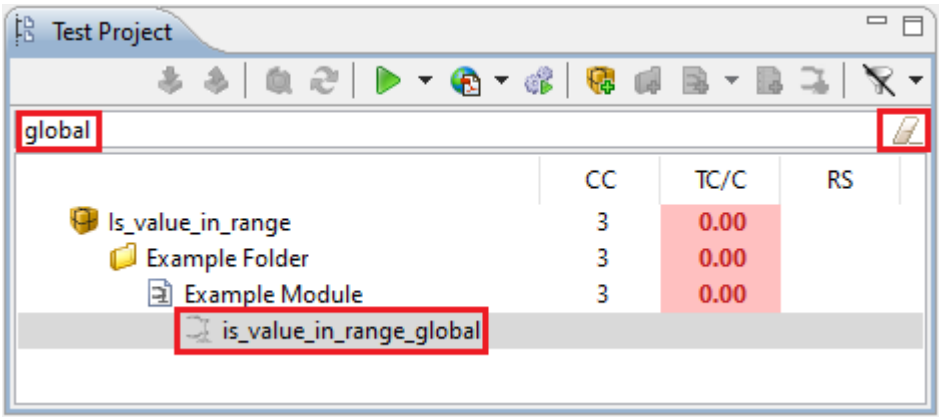


Figure 6.39: Searching for “global”

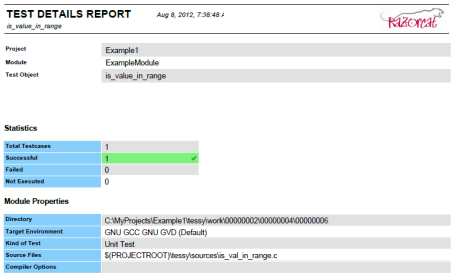
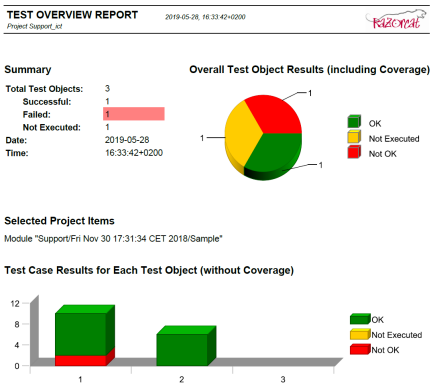
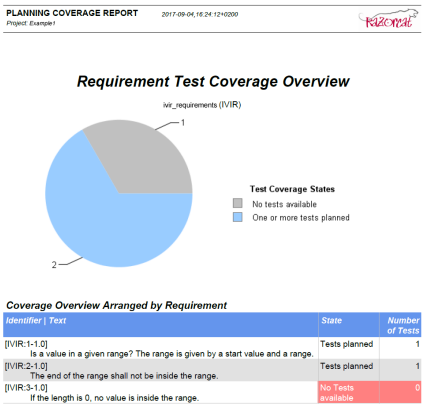
6.2.2.16 Creating reports

Important: All reports are generated as PDF files by default. To be able to open TESSY report files and enable the generation of test reports you need to install a third party PDF viewer like Adobe Reader 7.0 or higher, Sumatra PDF, Foxit etc.

Other available report formats are HTML and Word. Because of potential layout issues the usage of those formats is discouraged. Also task attachments e. g. will not appear in HTML. Therefore HTML as well as Word are provided as complementary helper formats only and without further support.

You can also create pure XML reports for further processing with your own tools.

The following reports are available in TESSY:

Test Details Report  TEST DETAILS REPORT Project: Example1 Module: ExampleModule Test Object: is_value_in_range Statistics Total Testcases: 1 Successful: 1 Failed: 0 Not Executed: 0 Module Properties Directory: C:\MyProjects\Example1\tessy\work\00000002\00000004\00000006 Target Environment: GNU GCC GNU GVD (Default) Kind of Test: Unit Test Source Files: ${PROJECTROOT}/tessy/sources/is_val_in_range.c Compiler Options:	Can be created for individual test objects or tasks. Reports for test objects contain metrics, coverage and execution results, linked requirements, information about the test cases, their properties, values and results. Task reports contain the task properties, linked requirements and the task result. The Test Details Report is based on the XML result file which is created after every test run and stored within the latest test run. A test report can be generated as soon as TESSY has finished running a test.
Test Overview Report  TEST OVERVIEW REPORT Summary Total Test Objects: 3 Successful: 1 Failed: 1 Not Executed: 1 Date: 2019-05-28 Time: 16:33:42+0200 Overall Test Object Results (including Coverage) OK Not Executed Not OK Selected Project Items Module "Support/Fri Nov 30 17:31:34 CET 2018/Sample" Test Case Results for Each Test Object (without Coverage) OK Not Executed Not OK	Contains information about the test objects and the test case results in tabular and graphical form (pie chart). The Test Overview Report summarizes the results of a selected set of test objects. The report can be generated based on already executed test objects or right after the execution of the selected test objects.
Planning Coverage Report  PLANNING COVERAGE REPORT *Requirement Test Coverage Overview* ivir_requirements (IVIR) Test Coverage States No tests available One or more tests planned *Coverage Overview Arranged by Requirement* Identifier | Text / State / Number of Tests [IVIR:1-1.0] Is a value in a given range? The range is given by a start value and a range. — Tests planned — 1 [IVIR:2-1.0] The end of the range shall not be inside the range. — Tests planned — 1 [IVIR:3-1.0] If the length is 0, no value is inside the range. — No Tests available — 0	Contains information about the requirements linked to test cases and the planned validation. Planning coverage summarizes the achieved coverage of requirements by test cases. Each link of a test case or task to a requirement will be counted. This report provides information about the planning progress of the test project, since it will show all requirements that have not yet been planned to be tested by at least one test case or task.

continue next page

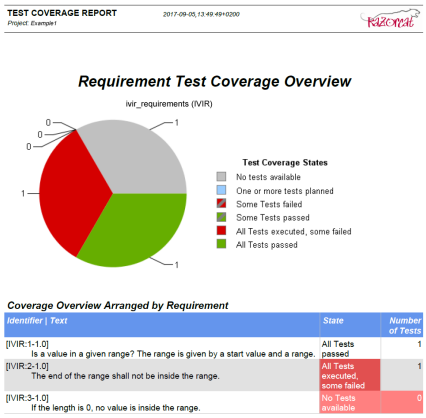
Execution Coverage Report  TEST COVERAGE REPORT Project: Example1 2017-09-05,13:49:49+0200 *Requirement Test Coverage Overview* ivir_requirements (IVIR) Test Coverage States No tests available One or more tests planned Some Tests failed Some Tests passed All Tests executed, some failed All Tests passed *Coverage Overview Arranged by Requirement* Identifier | Text / State / Number of Tests [IVIR:1-1.0] Is a value in a given range? The range is given by a start value and a range. — All Tests passed — 1 [IVIR:2-1.0] The end of the range shall not be inside the range. — All Tests executed, some failed — 1 [IVIR:3-1.0] If the length is 0, no value is inside the range. — No Tests available — 0	Contains information about the validation of requirements after test run. Execution coverage summarizes the achieved coverage of requirements based on executed tests. The results of test runs (i.e. the result of each test case) will be propagated to the list of requirements or tasks, if any link was set. This report provides an overview about the result status of the test project since it will show all requirements for which the linked test cases are failed for any reason.

Table 6.15: Reports available within TESSY

All reports are created as PDF documents based on XML data files. These XML data files can also be used for generating reports or further processing if desired.

Creating reports

To create a report within TESSY:

→ Click in the Test Project view (i.e. within the Overview perspective) on the arrow next to the Generate Report icon .

→ Select the report you wish to create (see figure 6.40).
TESSY creates the report within the new folder. This will take a few seconds. When finished, TESSY will open the file automatically.

Important: The first time you create a report, the “Edit Settings” dialog will be opened automatically. These settings are memorized and used for the following reports.

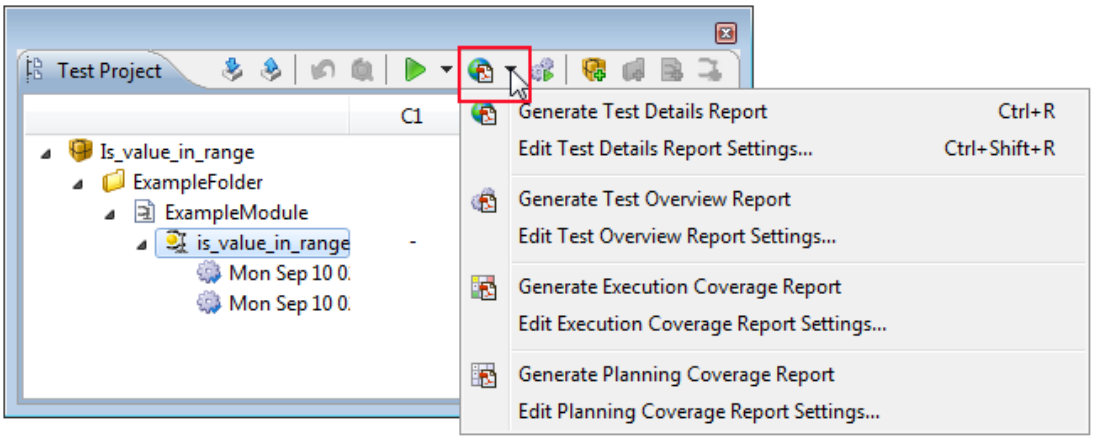


Figure 6.40: Creating a report

Comments added or modified in the Properties view will only appear in the Test Details Report after the test was executed. Notes added to test modules, test objects and test cases via content menu or modified in the Notes view on the lower right of the Overview perspective will appear in the Test Details Report as well as in the Test Overview Report without any test execution. (Test cases can be found in the Test Items view of the Overview Perspective.) Generally comments will not appear in the Test Overview Report, also comments and notes will not appear in Planning Coverage Reports and Execution Coverage Reports.

To understand the usage of notes within TESSY see section 6.2.2.10 Notes.

Change settings of a report

To change settings:

→ Click on the arrow next to the Generate Report icon .

→ Select “Edit [report] Settings…” .
The settings dialog for the selected report will be opened.

You can as well change basic settings, e.g. output directories, filenames and the logo on the reports. Refer to section 6.1.2.1 Window > Preferences menu.

The Test Details Report Settings dialog:

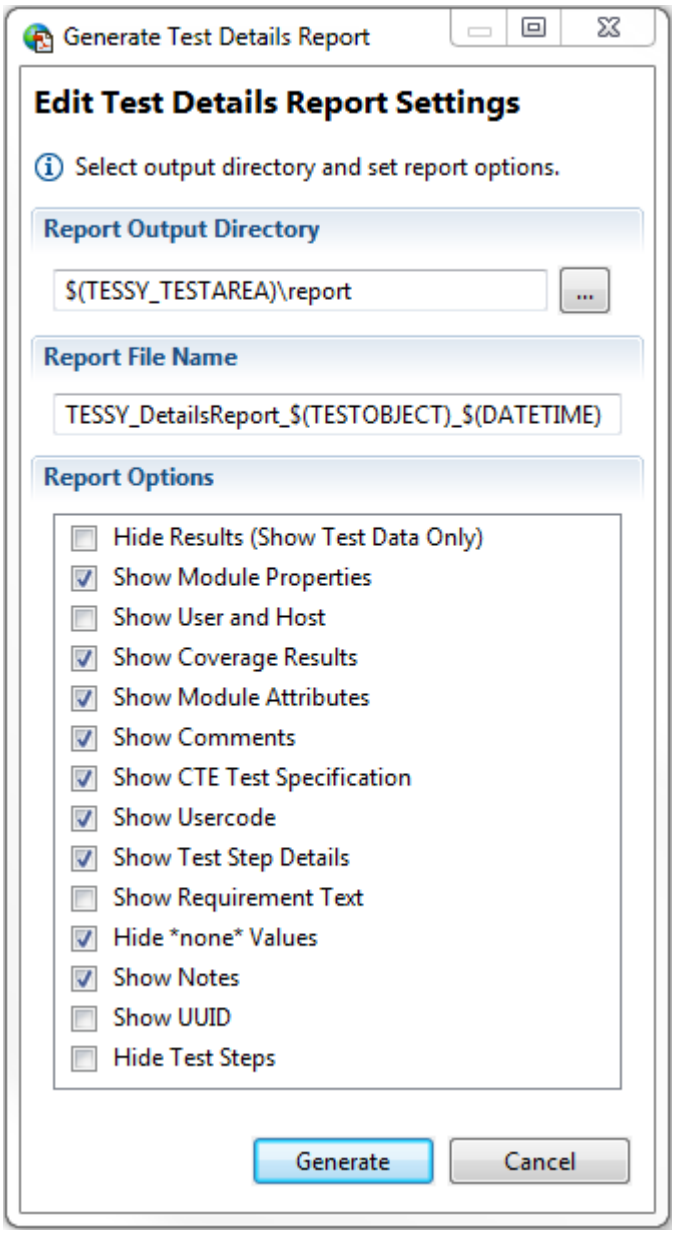


Figure 6.41: Test Details Report Settings dialog with default and optional settings

The Test Overview Report Settings dialog:

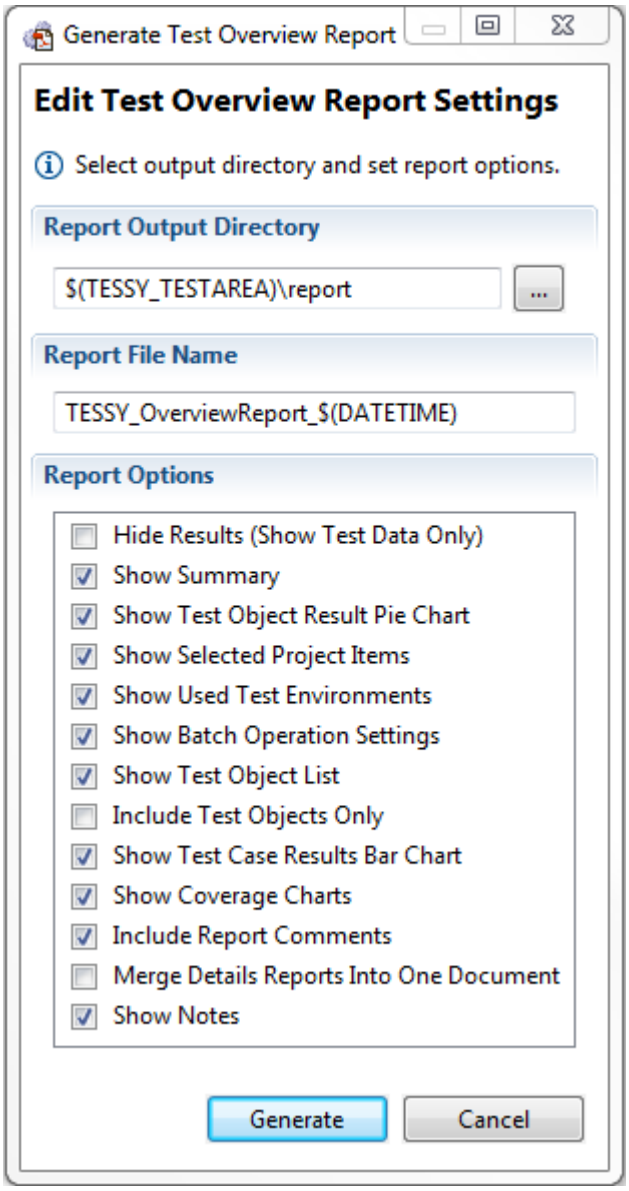


Figure 6.42: Test Overview Settings dialog with default and optional settings

Important: The option “Merge Details Reports Into One Document” is of special significance as all the other report options simply change the outline of the Test Overview Report or add or hide certain contents.
Merging the reports is only possible for PDF format and when both the Test Overview Report and the Test Details Reports are generated within the same batch operation. Therefore this option will be hidden when it is not applicable.

The Planning Coverage Report Settings dialog:

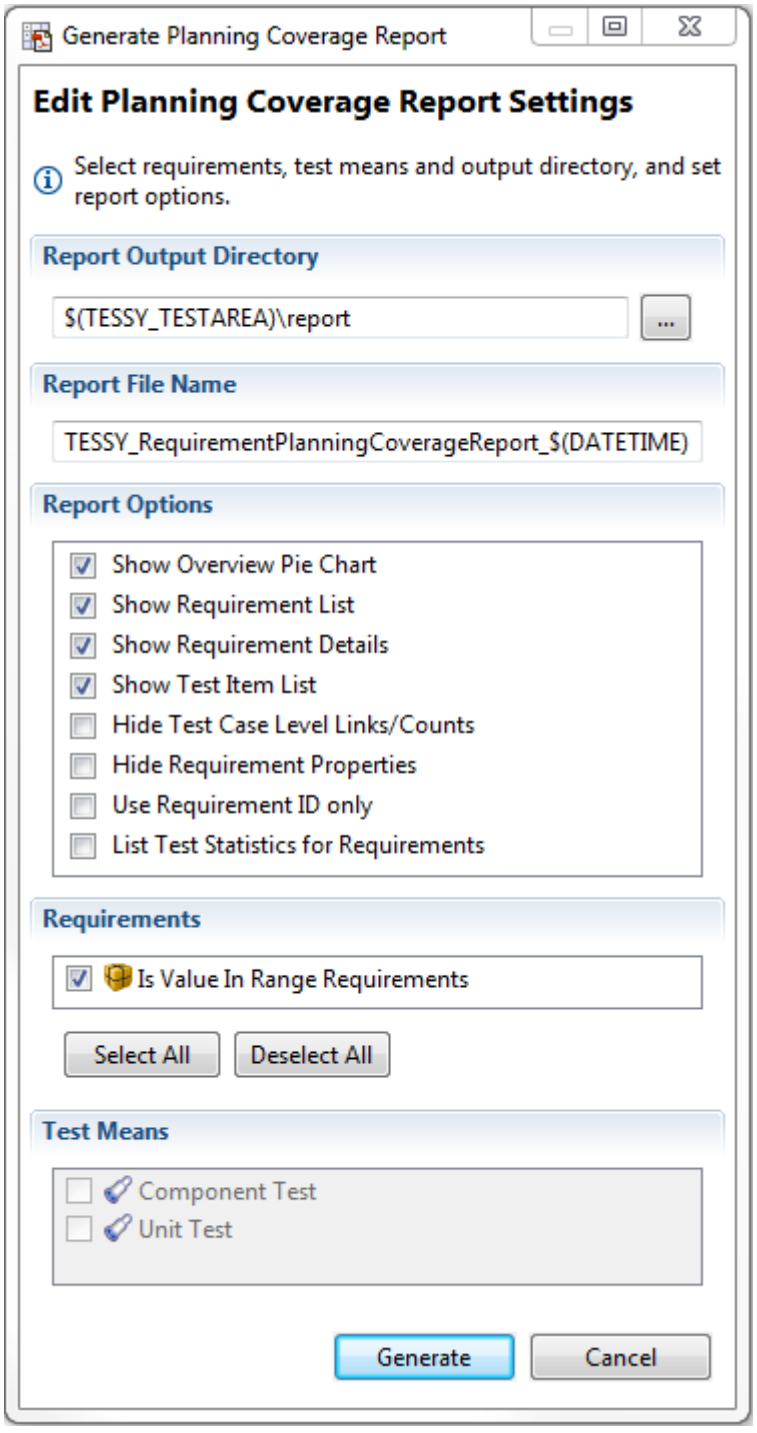


Figure 6.43: Planning Coverage Settings dialog with default and optional settings

The Execution Coverage Report Settings dialog:

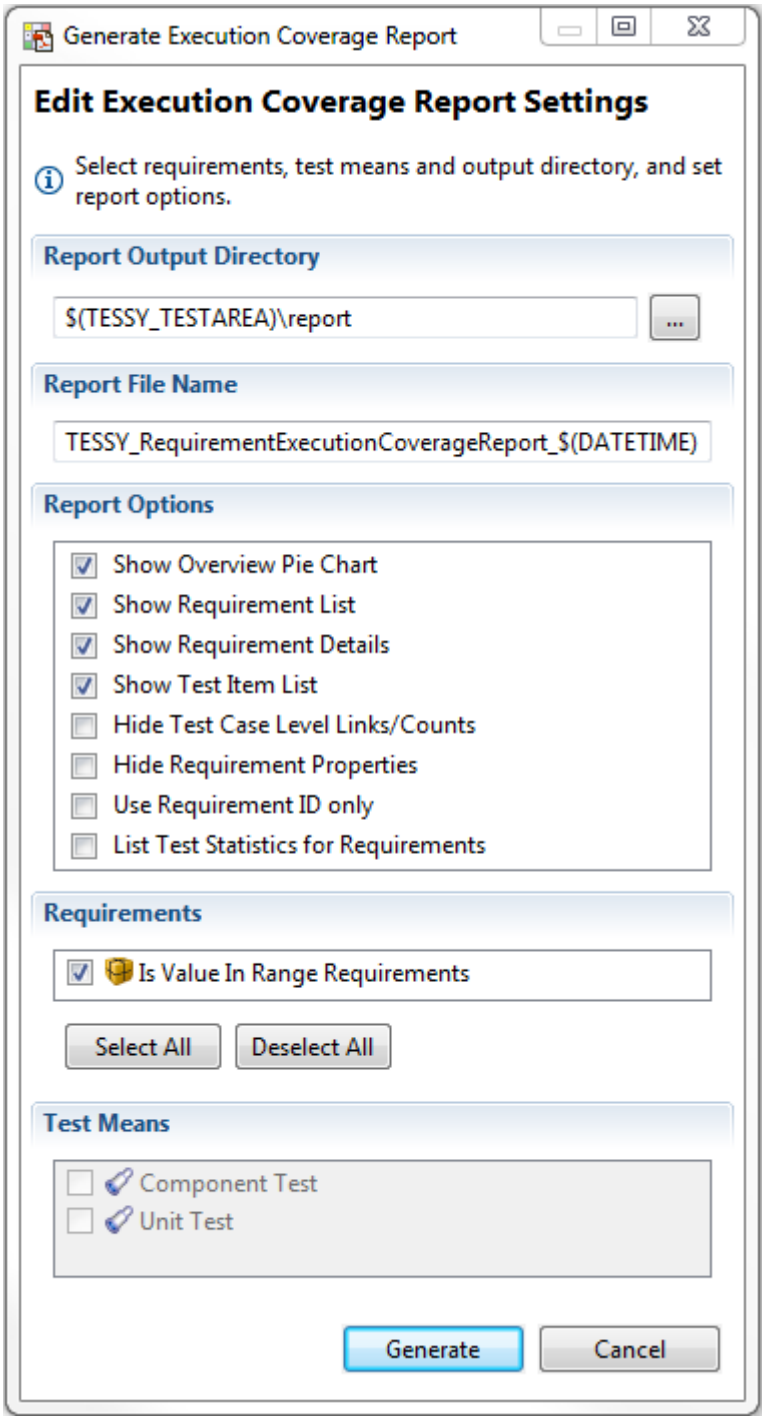


Figure 6.44: Execution Coverage Settings dialog with default and optional settings

Clear Report Output directory

To clear a report output directory:

→ Within the Windows Explorer go to the report output directory. The directory is displayed within the Test Project view under “Edit [...] Report Settings” .
→ Make sure the reports or the directory is not needed anymore!
→ Delete the reports or the directory.

6.2.2.17 Batch test operations

Batch Test

TESSY provides a batch test feature with various operations for test execution and reporting. You can define which operations shall be performed and which settings shall be used. This setup can be saved into batch script (TBS) files for test automation using the command line interface of TESSY (see chapter 6.16 Command line interface).

Open the batch operation settings:

→ In the Test Project view right-click a project, a module or a test object.
→ Select “Define Batch Operation…” from the context menu (see figure 6.45). The window “Define Batch Operation” will be opened.

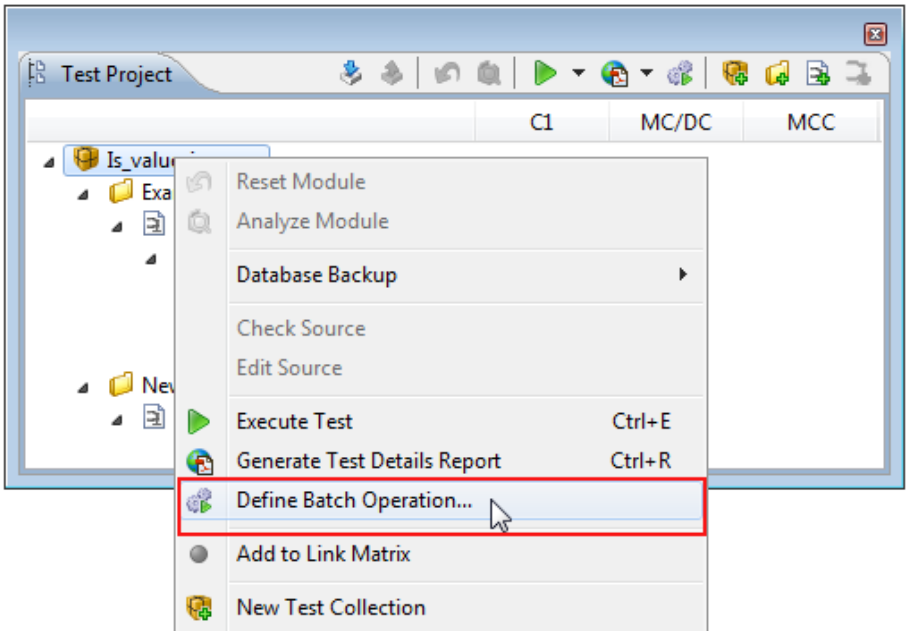


Figure 6.45: Context menu “Define Batch Operation”

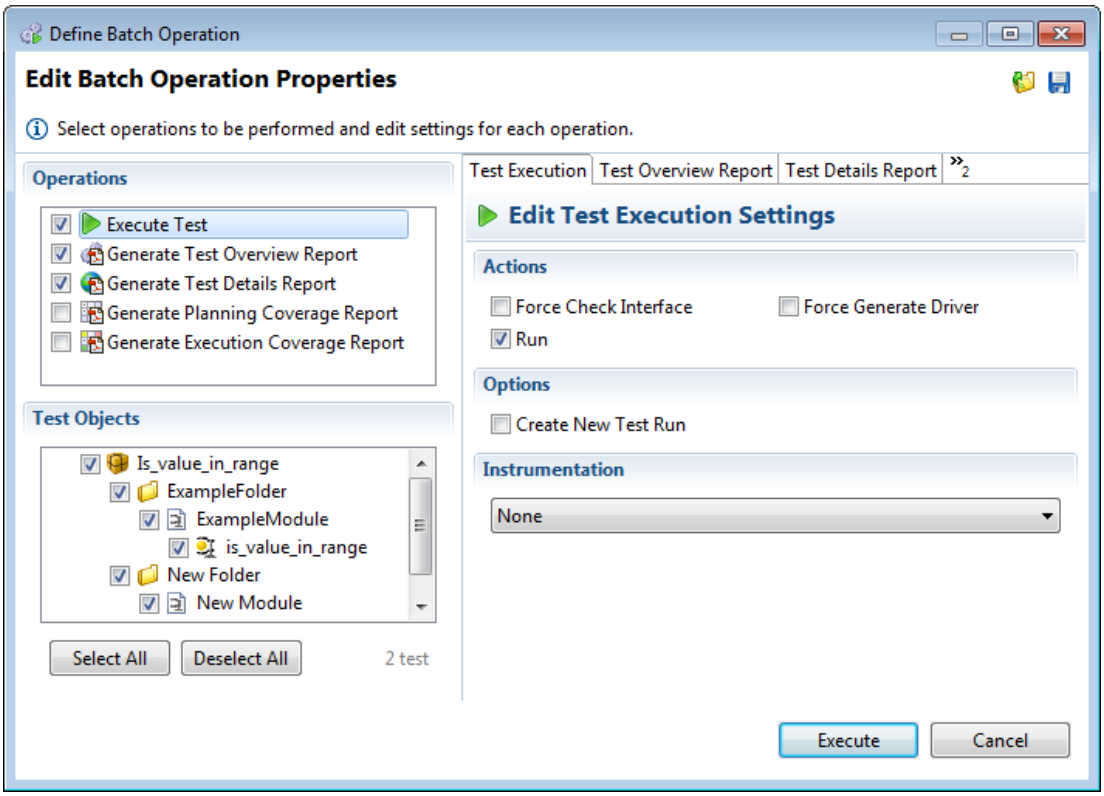


Figure 6.46: Defining the batch operation

To select the test objects for the batch test:

→ Under “Test Objects” choose the project or modules or test objects for the batch test. Click “Select All” to select all at once (see figure 6.47).

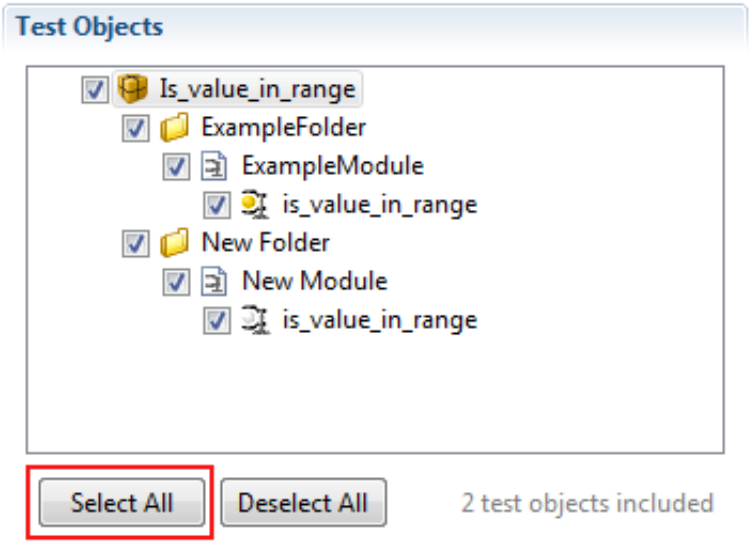


Figure 6.47: Selecting all text objects

The right side of the window is context sensitive:

→ Switch to the setting by either marking the operation on the left side or use the tabs on the upper right side (see figure 6.48). The optional settings for this operation will then be shown on the right side of the window.

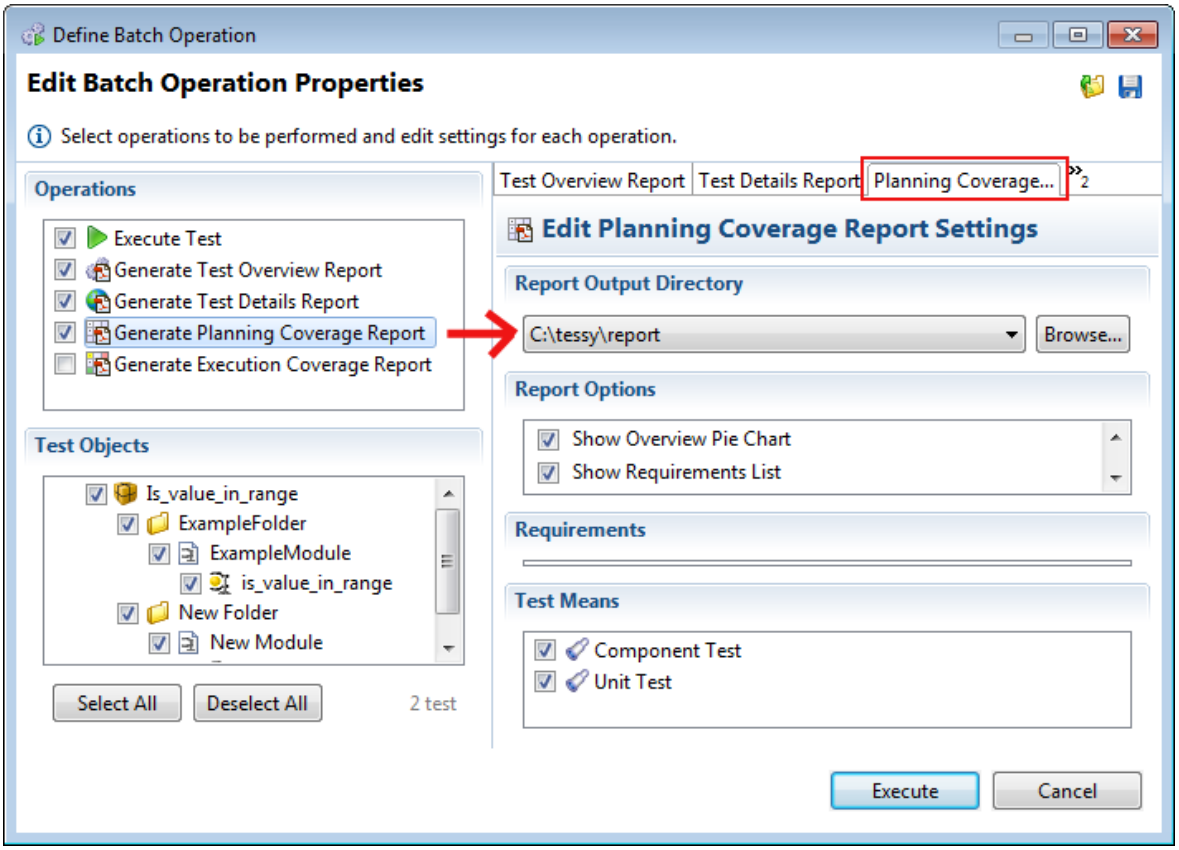


Figure 6.48: Editing the settings of each batch operation

You can create a TBS file for command line execution by saving the batch test settings:

→ In the batch operation settings window click on (Save batch file as…).
→ Choose the type of the file and click “Save” (see figure 6.49).

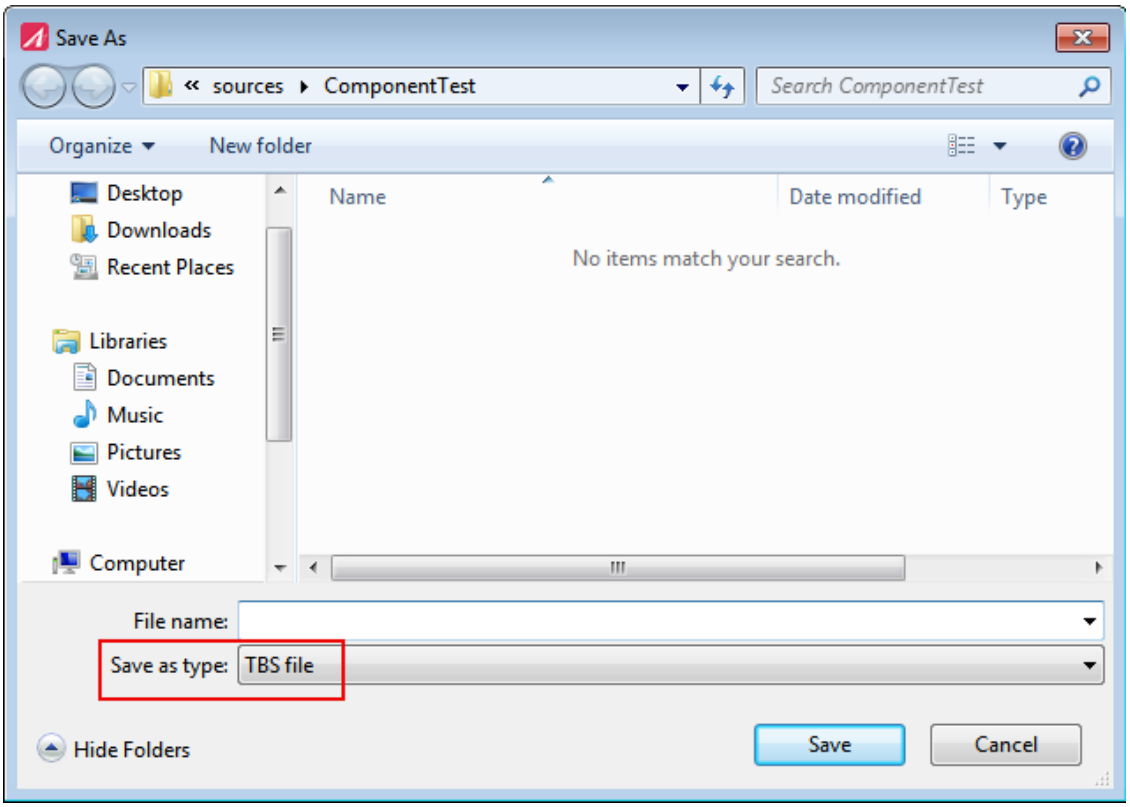


Figure 6.49: Saving the settings of a batch operation as TBS file

6.2.2.18 Importing and exporting

The Test Project view provides the import and export of test data and module backup files (*.TMB):

→ Select a folder, module or test object.
→ In the tool bar click on to import or to export the data.

The kind of import/export depends on the selection:

Selection	Type of file
Folder	Import: *.TMB files
Module	Export: *.TMB file
Test object	Import and export: test data

Table 6.16: Import/export selections

When importing test data for individual test objects there are following options (see figure 6.50):

- “Update passing directions”: If you tick the box, the passing directions of all interface variables will be set according to the passing directions specified within the import file. All other interface variables will be set to IRRELEVANT. The test object will be ready to execute when using this option because all variable with passing directions IN, OUT or INOUT will be filled with values.
- “Overwrite/append test cases”: Either delete existing test cases before importing or append any imported test cases at the end of the test case list.

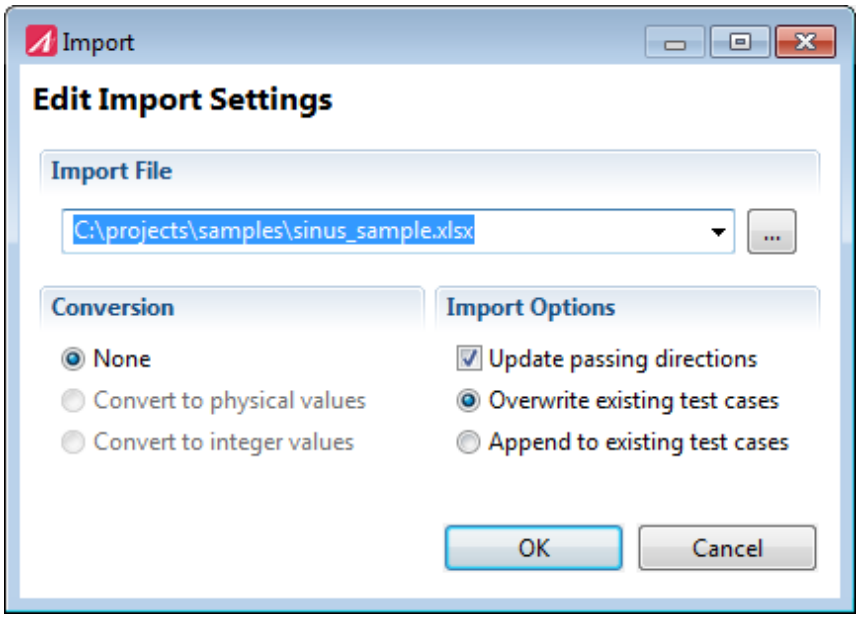


Figure 6.50: Import settings of data import

When exporting data there are following options (see figure 6.51):

- The conversion of the export settings is only applicable, if ASAP conversion is enabled! Please refer to table “General tab of Properties view” in section General tab.
- Input/Expected versus Input/Actual: Either export the expected or the actual result values. The latter is only available if the test has been executed and actual results are present.

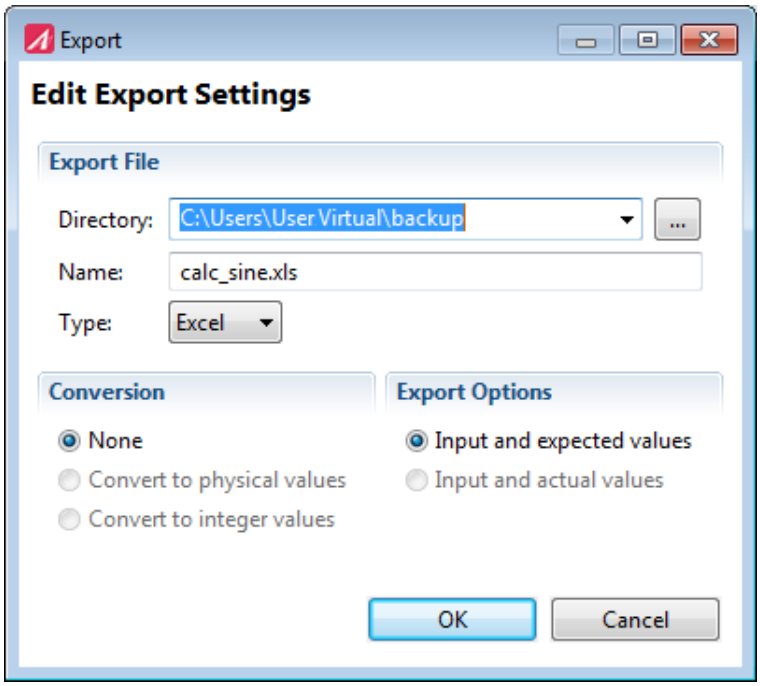


Figure 6.51: Export settings of data export

6.2.3 Properties view

Properties view

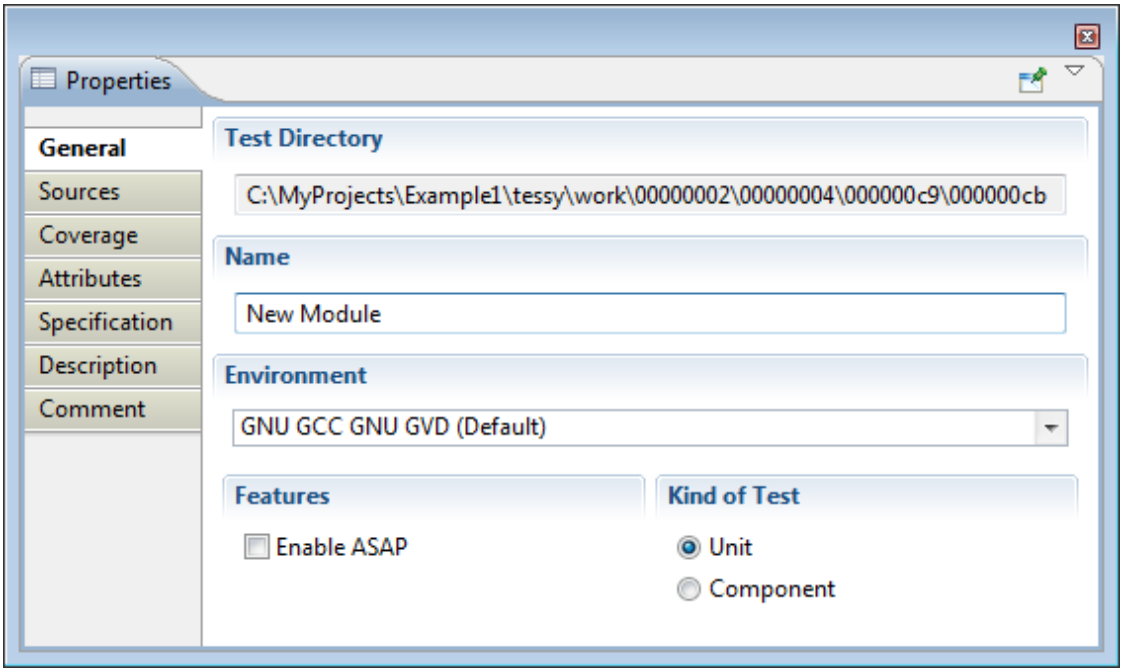


Figure 6.52: Properties view

The Properties view is divided into several tabs on the left and provides various settings which are explained in the following:

6.2.3.1 General tab

The General tab (see figure 6.52) is used to determine the test environment. Following options are available:

Option	Function
Test Directory	The path has been specified during database creation and is not adjustable here.
Name	Name of the element, e.g. test collection or module.
Environment	Specifies your target compiler (debugger/emulator/simulator) combination to be used for test execution. To enable the test environment see chapter 6.5 TEE: Configuring the test environment and section 6.5.7.1 Enabling configurations. The GNU toolset is already available by default.
Kind of Test	Unit: Enables the unit test of TESSY. Component: Enables the component test of TESSY.
Features	“Enable ASAP”: TESSY provides a close integration to the ASAP standard, allowing the usage of ASAP conversion rules for physical to integer conversion of test data. If ASAP is ticked, you will find additional attributes within the Attributes tab in which you have to specify your ASAP file. For more information refer to our application notes in the Help menu in the menu bar (“Help” > “Documentation” > “Using ASAP Information”).

Table 6.17: General tab of the Properties view

6.2.3.2 Sources tab

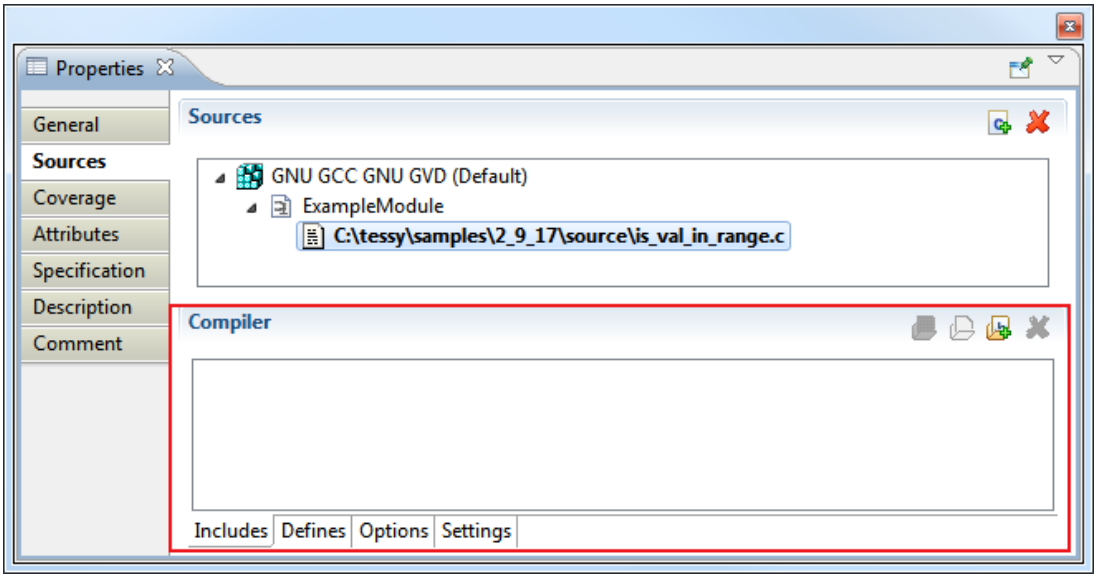


Figure 6.53: The Compiler pane in the Sources tab of the Properties view

Adding the C-source file

In the **upper pane of the Sources tab** the source files to be tested are added. All exported functions will be displayed if the module is opened. Some additional compiler options can be specified on module level by selecting the module entry, other options can be specified for each source file in the list.

To add a C-source file:

→ Click on (Add Source).
→ Select your C-source file.
→ Click “Open”. The C-source file will be added.

To remove a C-source file:

→ Select a source file and “Remove File” from the context menu.

To replace a C-source file:

→ Select a source file and “Replace File” from the context menu.
→ From the next dialog, select another source file.

The **lower Compiler pane of the Sources tab** displays information about the item selected from the upper Sources pane. Some of the displayed options (e.g. Includes) in the lower Compiler pane can be specified in the Test Environment Editor and will be inherited from there.

What kind of information is visible depends which tab is selected:

Module options apply to all source files unless otherwise specified on file level. File options apply to one selected source file and will overwrite options that are specified on module level.

Tab	Optional function within the Compiler pane
Includes	Add an include path of the headers which are included within the source file.
Defines	Define a macro for the preprocessor: → Click on and enter the name of the define without the normally used option of your target compiler, e.g. -D. TESSY will use the appropriate option automatically. Macros have to be separated by a comma or semicolon.
Options	Specify additional directives for your target compiler for your needs. Note that macros for the preprocessor and include paths have to be specified within the Defines tab respectively within the Includes tab. All compiler options added here will be used for the compilation of the source file when building the test driver application.

continue next page

Tab	Optional function within the Compiler pane
Settings	Depending on the selected item in the Sources pane the following features can be enabled (box is checked) or disabled (box is unchecked) in the Compiler pane: • Static Functions: When enabled, static functions can be tested and are added in the test object list of the module. The source code will be instrumented. • Inline Functions: When enabled, Inline Functions can be tested and are added in the test object list of the module. • Static Local Variables: When enabled, access to static local variables may be used as normal input or output variables within TESSY. The source code will be instrumented. • Hide Functions: When enabled, all functions of the selected source file are removed in the test object list of the module. This option is useful for additional C-source files needed for testing (e.g. implementation of stub functions for called functions), since they are not relevant for testing and reporting. • Enable User Includes: When enabled, all included header files of the source file(s) are included in the user code.

Table 6.18: Optional functions of the Sources tab of the Properties view

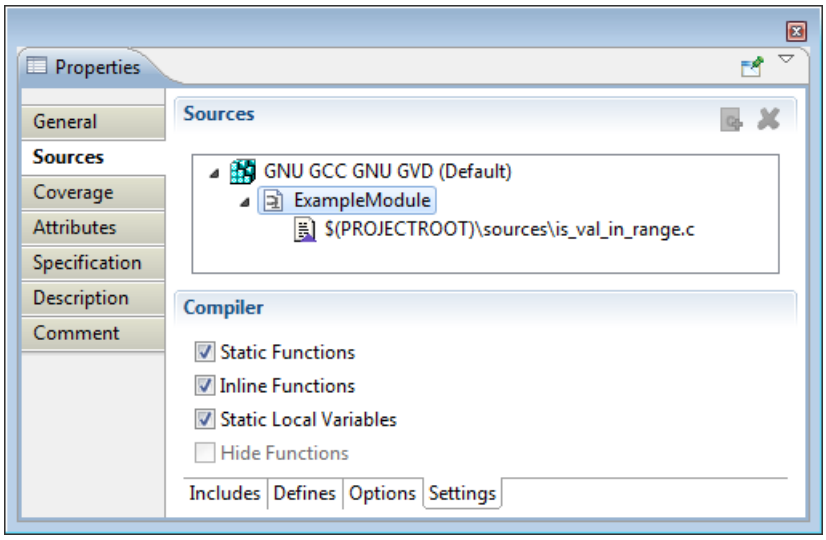


Figure 6.54: The Setting tab of the Properties view with module selected

6.2.3.3 Linker tab

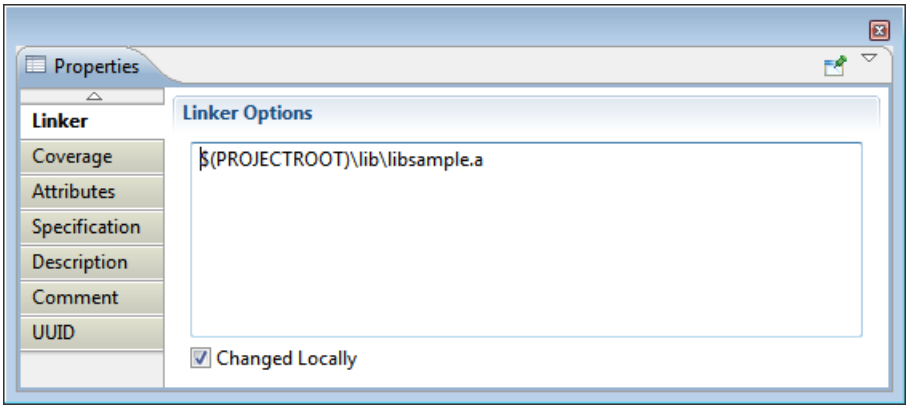


Figure 6.55: The Linker Options tab of the Properties view

Any linker options like object files or libraries can be added here. You can use predefined variables like $(PROJECTROOT) or $(SOURCEROOT) as described in section Creating databases and working with the file system. It is recommended to add such linker options using the environment editor TEE: Configuring the test environment.

6.2.3.4 Coverage tab

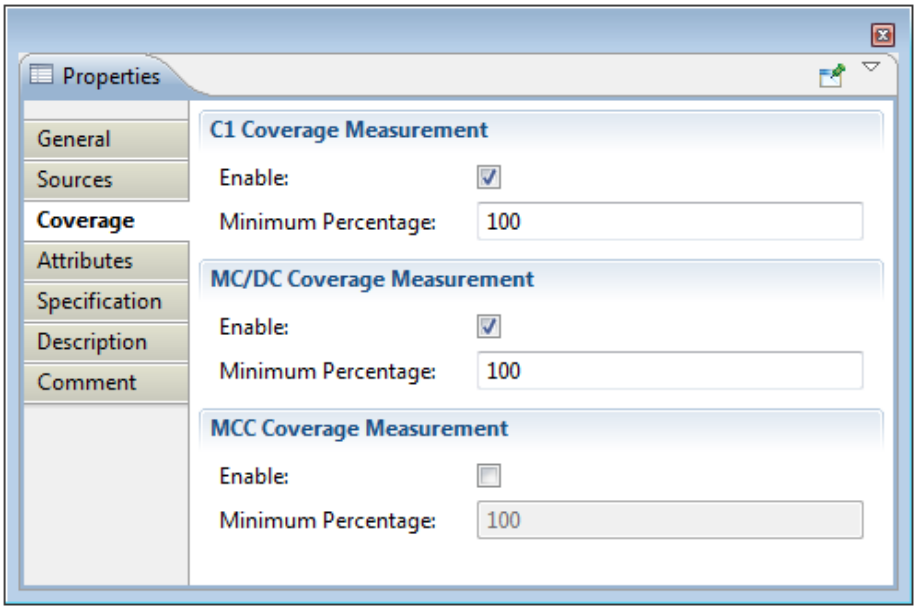


Figure 6.56: The Coverage tab of the Properties view

See 6.1.2.1 Window > Preferences menu

While you have probably chosen an instrumentation for coverage measurements as default for your whole project as described within section 6.1.2.1 Window > Preferences menu, within the Properties view you can enable a different coverage measurement on folder or test collection level or for a single module:

→ In the Test Project view select your module, for which you want to choose the coverage measurement.
→ In the Properties view select your coverage selection in the pull-down menu (see figure 6.57).

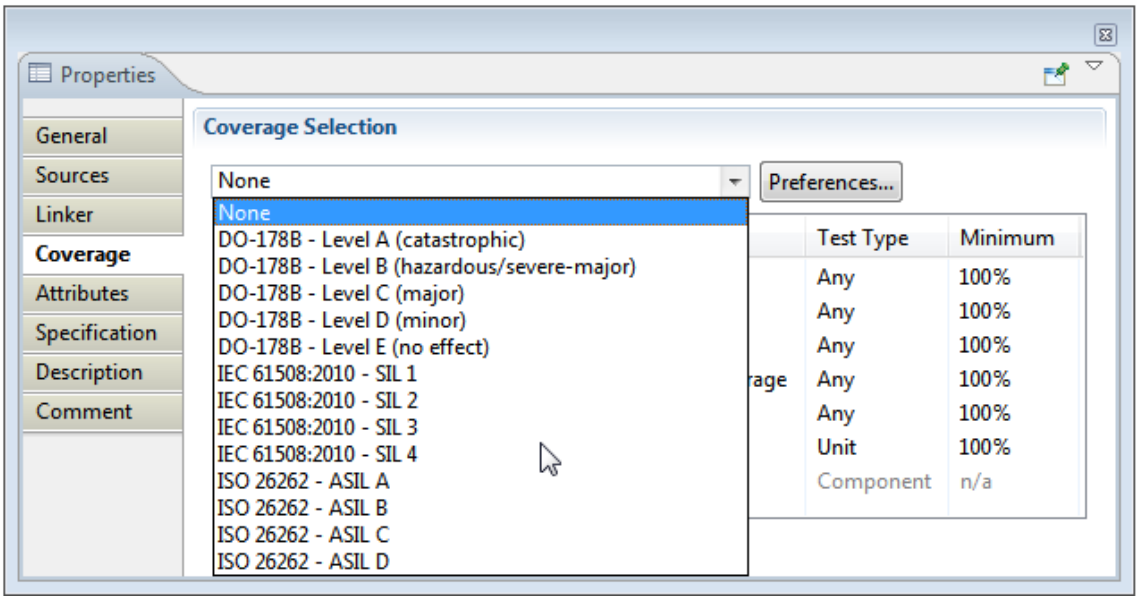


Figure 6.57: Selection of the coverage measurement in the pull-down menu

The pre-selected coverage instrumentations according to the selected safety standard level will be displayed.
TESSY supports the following instrumentations:

- Statement Coverage (C0)
- Branch Coverage (C1)
- Decision Coverage (DC)
- Modified Condition/Decision Coverage (MC/DC)
- Multiple Condition Coverage (MCC)
- Entry Point Coverage (EPC)
- Function Coverage (FC)

To edit the coverage selection of a measurement:

→ Click on the button “Preferences”.

→ Edit the instrumentation to your needs by ticking or unticking the boxes or creating a new coverage selection according to your required safety standard and level with a click on “Create”.

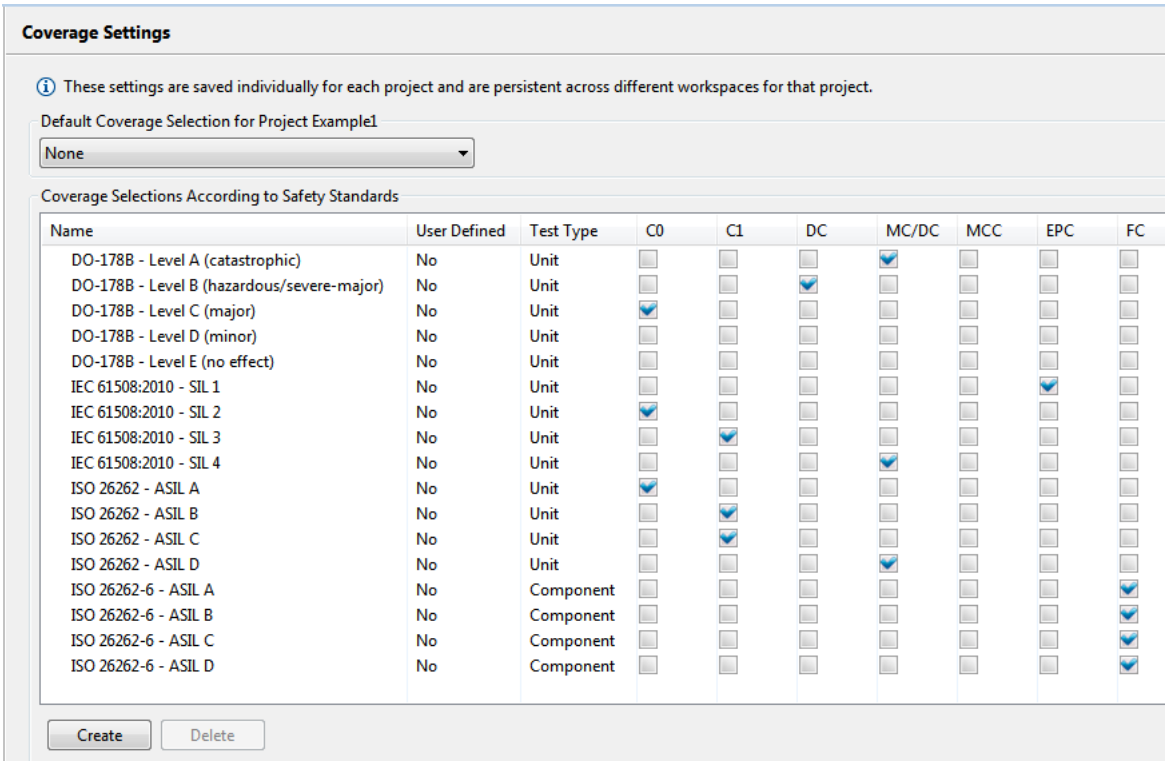


Figure 6.58: Defining coverage measurements

You can choose a different instrumentation for each test run. The options will be shown within the Execute Test dialog (refer to section 6.2.2.11 Executing tests).

See 6.2.2.11 Executing tests

To analyze the coverage refer to chapter 6.11 CV: Analyzing the coverage. For more information about coverage measurements and usage of coverage analysis refer to the application note “Coverage Measurement” in TESSY (“Help” > “Documentation”).

See 6.11 CV: Analyzing the coverage

6.2.3.5 Attributes tab

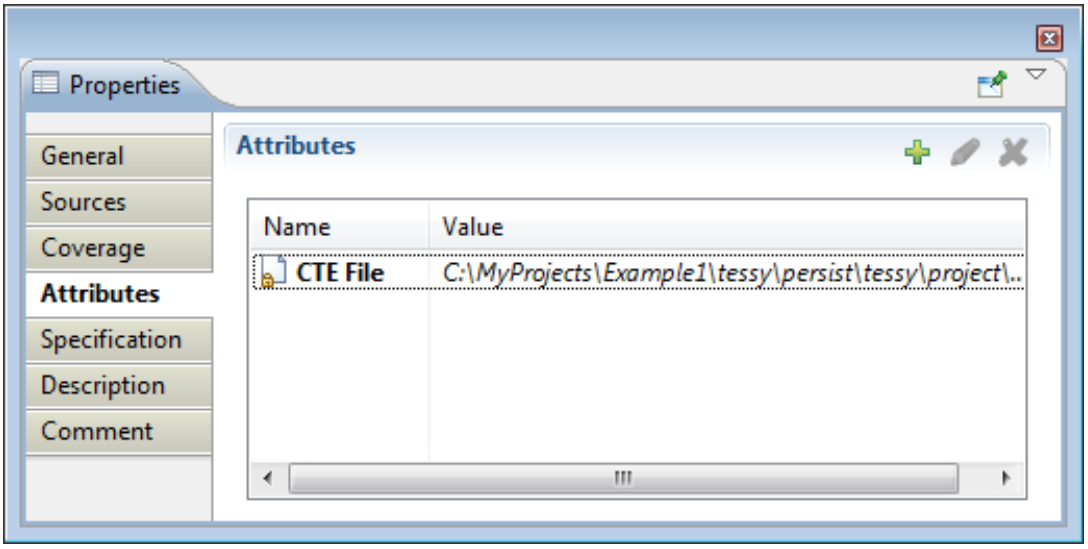


Figure 6.59: The Attributes tab of the Properties view

The Attributes tab specifies settings required by the compiler or the target environment of the module. Most attributes were preset and inherited from the Test Environment Editor (TEE).

Insert attributes

You can change the default values or add new attributes to the Attributes pane:

Changes are carried out only locally and do not influence other modules.

To create a new attribute:

→ Click on (New Attribute).
The Edit Attribute Properties dialog will be opened.

→ Enter an attribute name and select an appropriate type, e.g. String. Available types are String, Boolean, Integer, Real, File, Folder and Url.

→ Select appropriate flags, depending on the type selected.

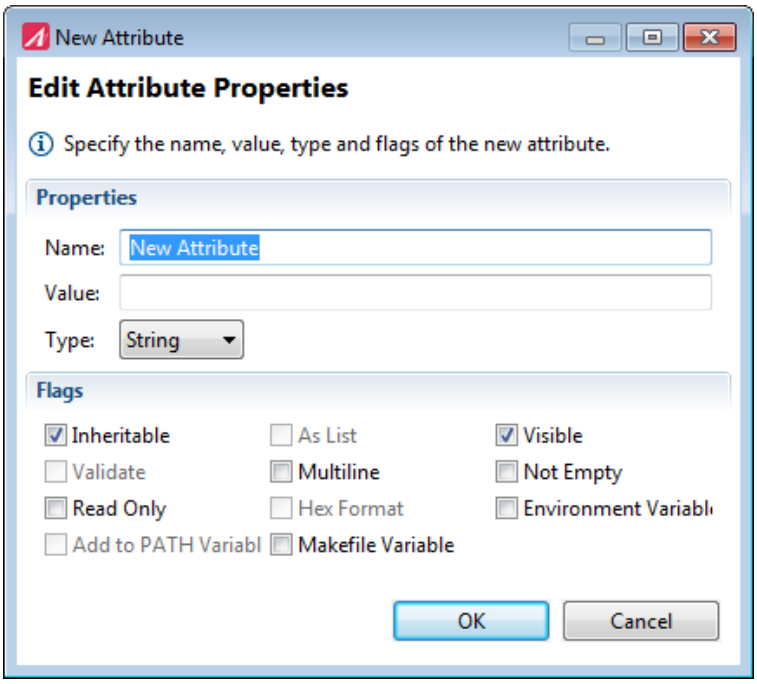


Figure 6.60: Creating a new attribute

To edit an existing attribute:

→ Click on (Edit Attribute).
The Edit Attribute Properties dialog will be opened.

You can remove user defined attributes. You cannot remove default attributes, only reset the value to its default state, if changed before.

To remove an attribute respectively reset a default attribute:

→ Click on .

6.2.3.6 Specification / Description / Comment tabs

Those tabs provide editable textboxes to be used for specifications, descriptions and comments by the tester.

If you want to add Notes, you have to use the content menu in the Test Project view. More information about notes can be found in section 6.2.2.10 Notes.

6.2.4 Requirements Coverage view

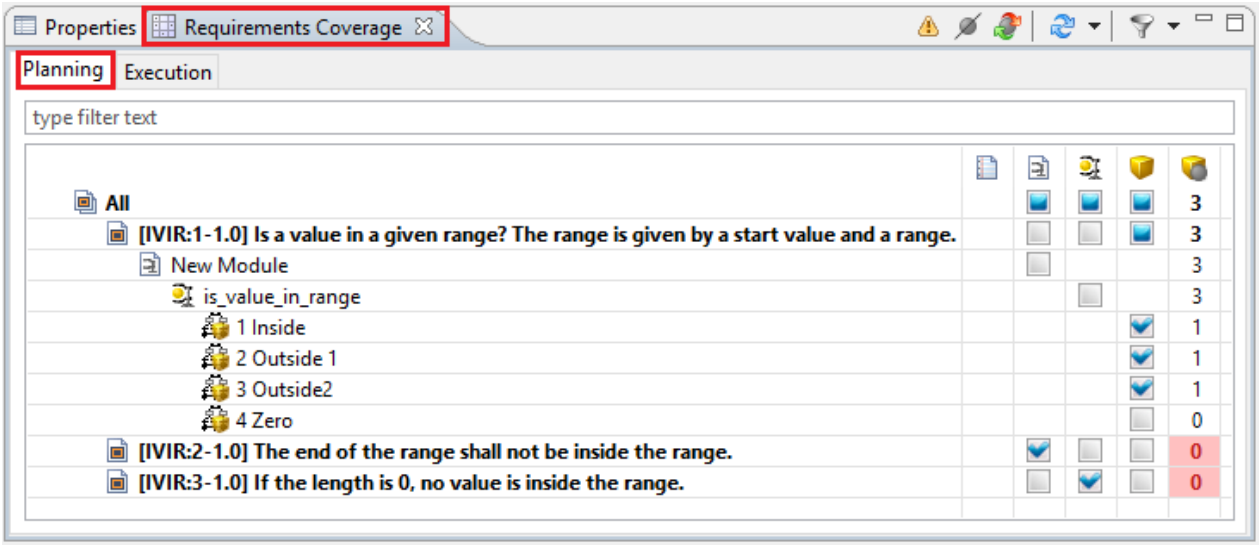


Figure 6.61: Requirements Coverage view

Within the Requirements Coverage view you can link the requirements with your test cases or tasks. We will describe this view in section 6.4 Requirement management > 6.4.16 Requirements Coverage view.

6.2.5 Test Items view

Test Items view

In the Test Items view you get an overview about your test cases and test steps, and you can as well create test cases and test steps manually without using the Classification Tree Editor (CTE, see section 6.8). This is useful for simple test objects with a few test cases that can be documented in a few words manually.

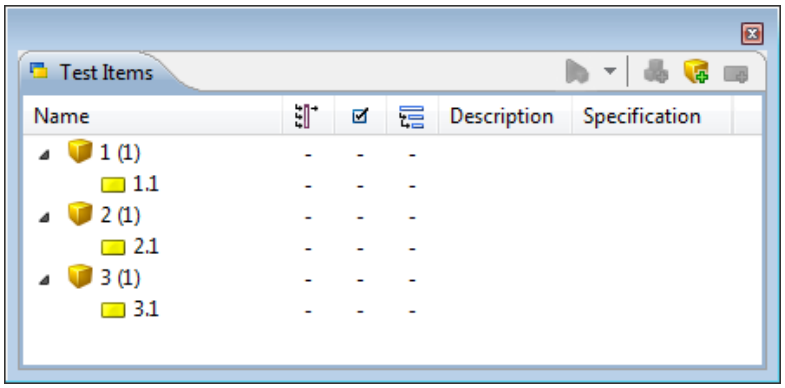


Figure 6.62: Test Items view

6.2.5.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Executes selected test cases.
	Adds new test case.
	Generates test steps automatically i.e. within a range.
	Adds new test step.
	Adds new group.
	Renumbers the test cases and test steps.
	Expands all test cases.
	Collapses all test cases.

Table 6.19: Tool bar icons of the Test Items view

6.2.5.2 Column indicators

Indicator	Status / Meaning
	Displays expected results.
	Displays evaluation macros results.
	Displays call trace results.

Table 6.20: Column indicators of the Test Items view

6.2.5.3 Status indicators

Indicator	Status / Meaning
	The test case has no data.
	The test case contains any data.
	At least one test step of the test case is ready to be executed.
	Test case passed: The actual results did match the expected results.
	Test case failed: The actual result of at least one test step did **not** match the expected results.
	Test case variant that was inherited from a parent module..
	New test step within a variant module..
	Inherited and overwritten test case of a variant module..
	Inherited and deleted test case of a variant module..
	Test Case Generator: This test case generates test steps automatically, i.e. if you enter a range. It does not contain any data yet.
	Test Case Generator with data: This test case has automatically generated test steps.
	Inherited and overwritten generator test case.

continue next page

Indicator	Status / Meaning
	The test step has no data.
	The test step contains any data.
	The test step is ready to be executed.
	Test step passed: The actual result did match the expected results.
	Test step failed: The actual result did **not** match the expected results.
	Test step variant that was inherited from a parent module.
	New test step within a variant module.
	Inherited and overwritten test step of a variant module.
	Inherited and deleted test step of a variant module.
	Inherited and overwritten generator test step.
	The test case has been created by the CTE and therefore can be changed only within CTE. The test case does not contain any data.
	The test case has been created by the CTE and therefore can be changed only within CTE. The test case does contain some data.
	The test case has been created by the CTE and therefore can be changed only within CTE. At least one test step is ready to be executed.
	The test step has been created by the CTE and therefore can be changed only within CTE. It does not contain any data.
	The test step has been created by the CTE and therefore can be changed only within CTE. It does contain some data.
	The test case has been created by the CTE and therefore can be changed only within CTE. At least one test step is ready to be executed.

Table 6.21: Status indicators of the Test Items view

6.2.5.4 Creating test cases and test steps

Creating test cases

To create test cases and test steps:

→ Switch to the Test Items view.

→ Click on (New Test Case).
The first test case is created and a test step is automatically added.

→ Add further test steps with a click on (New Test Step).

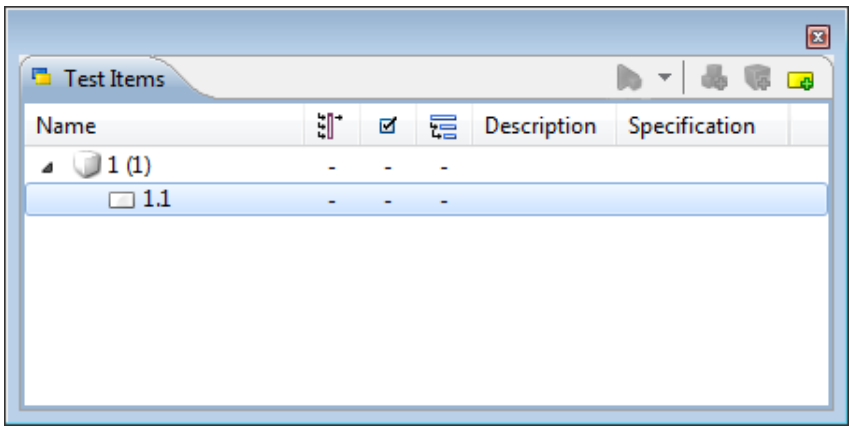


Figure 6.63: First test case with one test step

Please notice the following habits of this view:

- The first number is the number of the test case, the number in brackets shows the quantity of the test steps included.
- Test case numbers will be counted continuously: If you delete test cases, new test cases will get a new number and existing test cases will not be renumbered.
- If you cannot click on “New Test Case” oder “New Test Step” because the icons are inactive, you might be in the wrong selection: Select the test object within the Test Project view, then select the Test Items view.
- If you double-click a test case, the TDE will be opened to enter test data. Make sure to adjust or review the passing directions first in the TIE.

Every test step contains a complete set of test data. For instance, the mechanism of test steps can be used to achieve an initialization of the test object before executing the test step that checks the actual test condition of the current test case.

6.2.5.5 Creating test steps automatically

You can generate test steps automatically, i.e. with ranges of input values:

→ Click on the arrow next to the “New Test Case” icon .
→ Select “New Generator Test Case” (see figure 6.64).

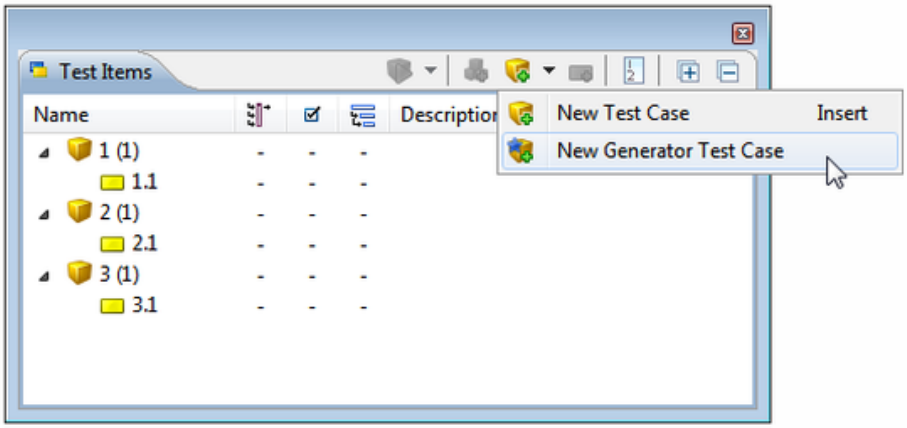


Figure 6.64: Selecting the test case generator

A new test case will be created. The star symbol indicates, that this test case is generated and you cannot add any test steps, because these will be generated automatically (see figure 6.65).

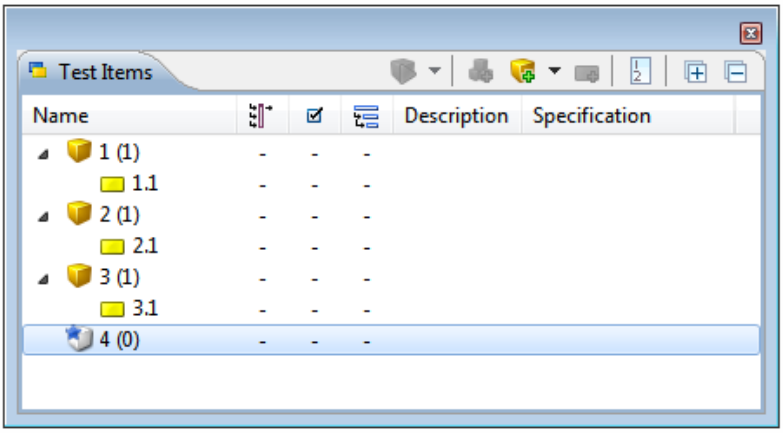


Figure 6.65: A new test case generator is created

To fill the data and generate the test steps, you will use the Test Data view within the TDE perspective:

→ Follow the description of section 6.9.7.8 Generating test steps automatically.

After generating the test steps, the icon of the test case within the Test Items view will change to yellow as well as test steps (see figure 6.66).

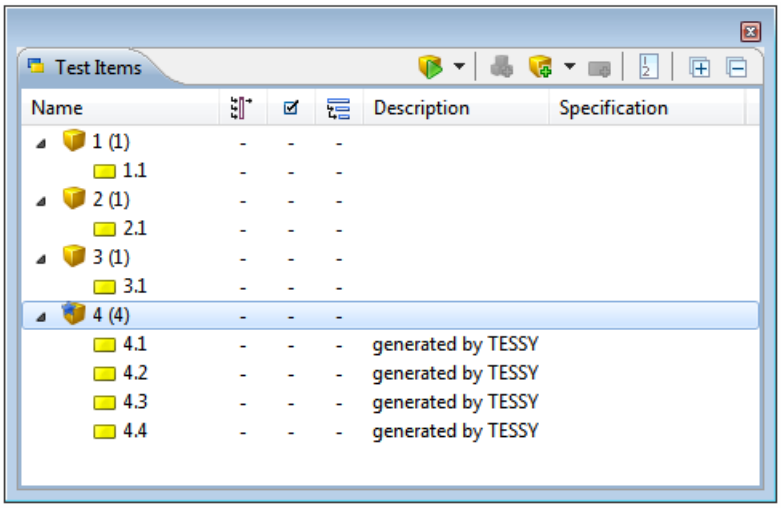


Figure 6.66: The test steps were generated and are ready to be executed

The test steps are read only! You can change the type of the test case and test steps to “normal”. That way you can edit the test steps as usual.
To change the status to normal,

→ right-click the test case and select “Change Test Case Type to Normal” (see figure 6.67).

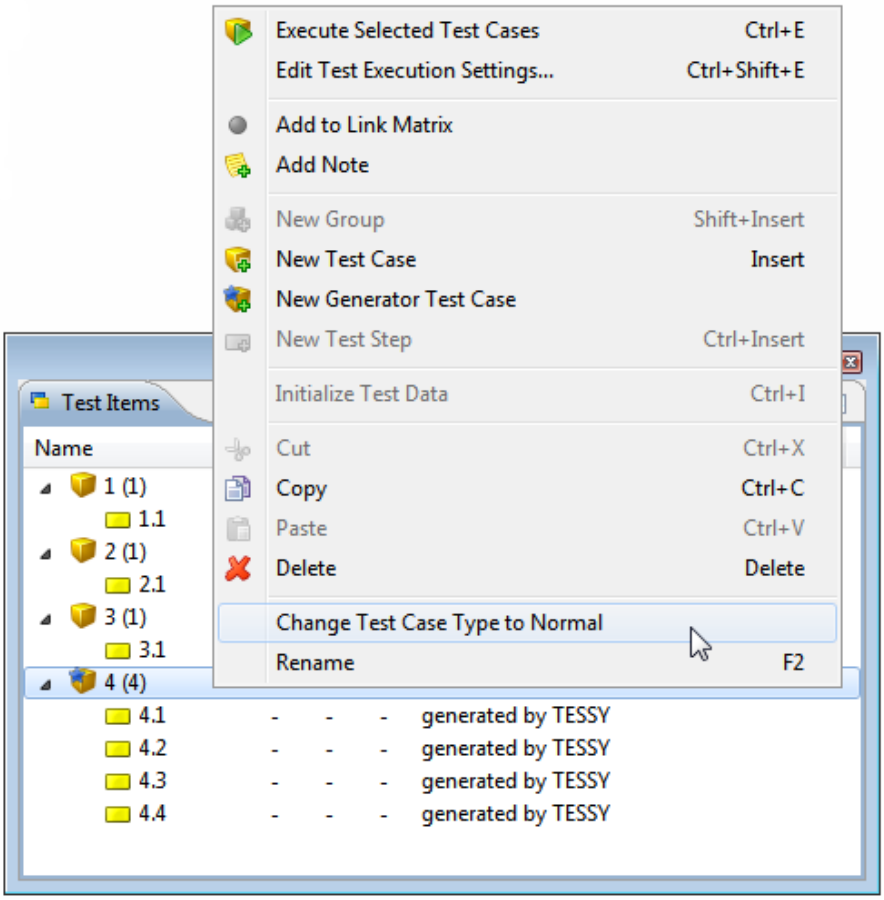


Figure 6.67: Selecting “Change Test Case Type to Normal”

Changing test case to type normal

The test case and test steps are changed to type “normal” but will indicate originally being generated with a status (see figure 6.68).

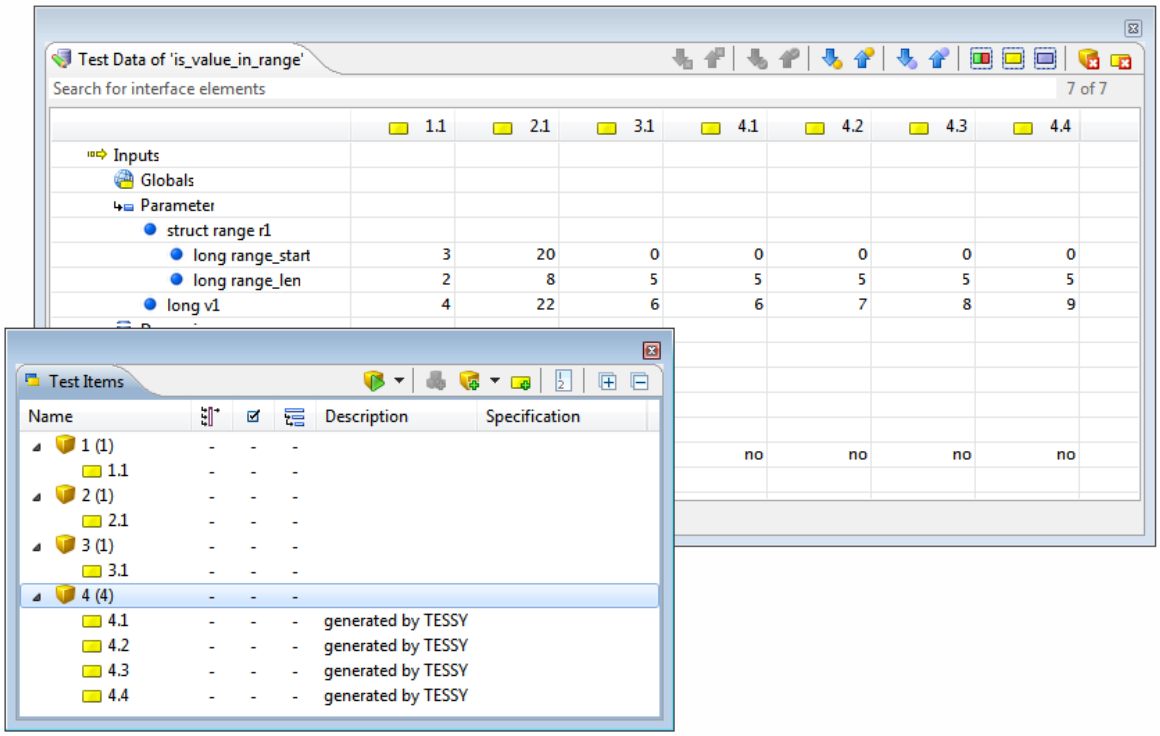


Figure 6.68: The test case and test steps originally being generated

You can reverse the action with a righclick and choose “Change Test Case Type to Generator” from the context menu.

6.2.5.6 Test cases and steps created within the CTE

If test cases and test steps were assigned within CTE, the icons of test cases and test steps within the Test Items view are displayed with a CTE symbol to indicate that you can change those test cases only within CTE (see figure 6.69).

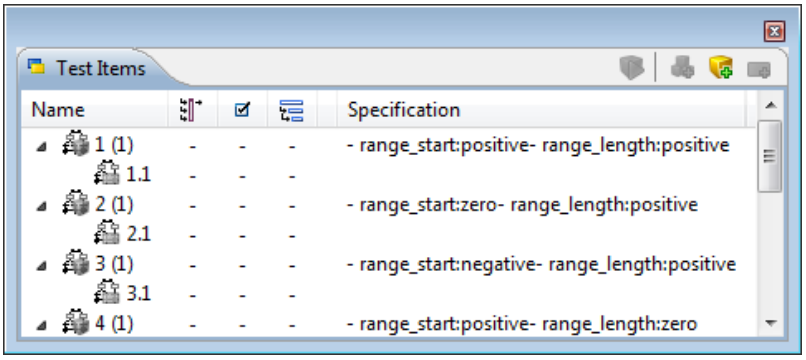


Figure 6.69: Test cases created by CTE

6.2.5.7 Test cases and steps inherited from a variant module

If test cases and test steps were inherited of a variant module as described in chapter 6.2.2.9 Creating variant modules, you can add, delete and overwrite the test steps and data. The icons will indicate each status:

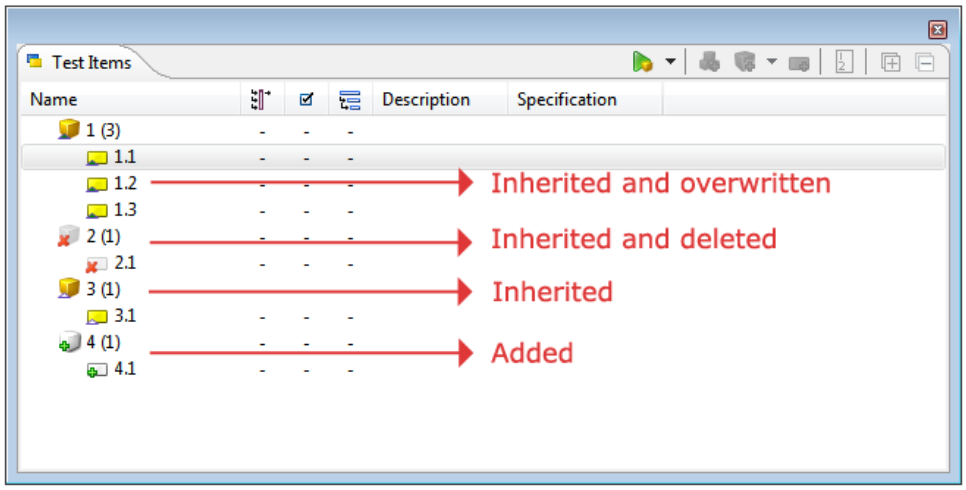


Figure 6.70: Test cases and test steps that were inherited of a variant module

Important: Deleted test cases/steps are only faded out within the child module. They can be made available again using “Restore Deleted” from the context menu.

Information about how to assign data in general and particularly to variants using the CTE is provided in subsection 6.8.7.1 Assigning test data to the CTE.

6.2.5.8 Renumbering test cases

After deleting test cases or test steps, you can renumber the existing test cases and steps:

→ Click on (Renumber Test Cases).
A notice will appear that all test cases will be renumbered (see figure 6.71).
→ Click “OK”.

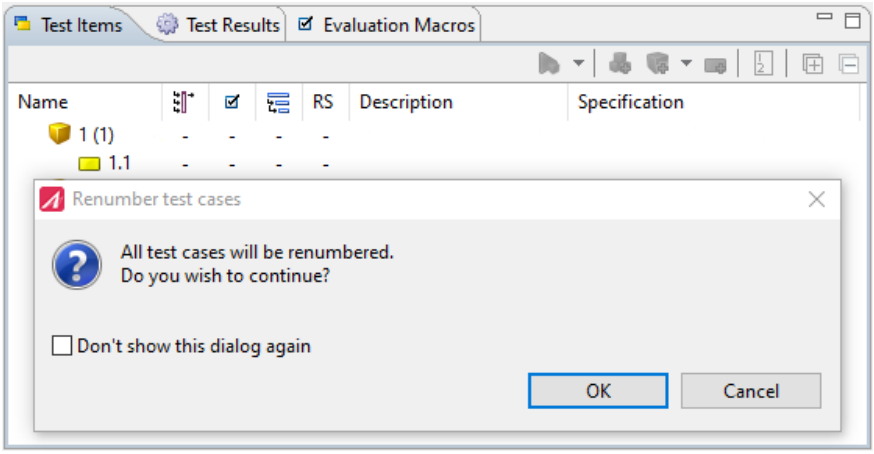


Figure 6.71: All test cases will be renumbered

6.2.6 Test Results view

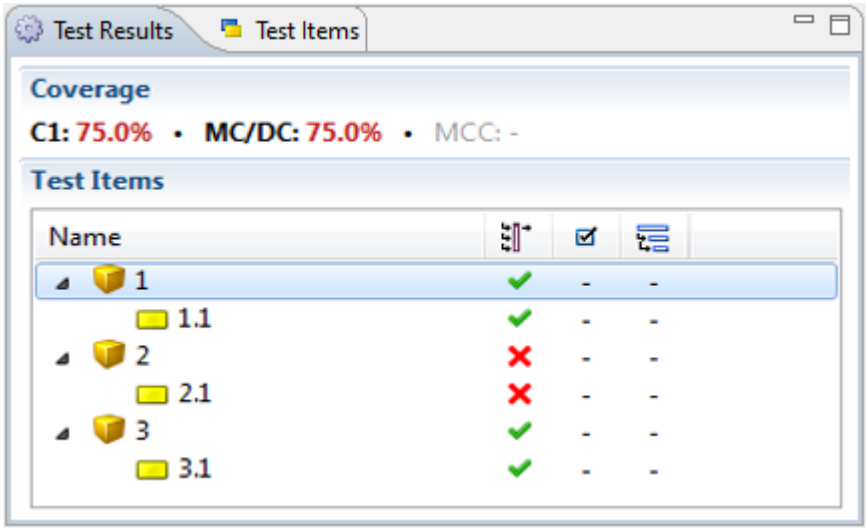


Figure 6.72: Test Results view

After a test run the Test Results view will display the coverage measurement results and the results of expected outputs, evaluation macros and call traces, if applicable.

Important: The view is context sensitive: If the Test Results view is empty, make sure a test run is selected within the Test Project view!

6.2.7 Evaluation Macros view

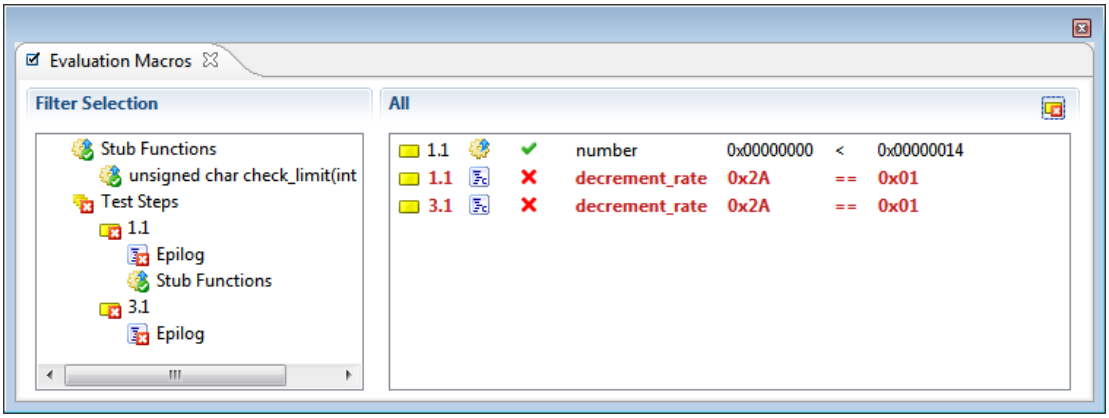


Figure 6.73: Evaluation Macros view

This view lists the detailed results of the evaluation macros if the usercode of the test object contains any evaluation macros, see 6.9.11.3 Using evaluation macros. The results are displayed wherever they occur within the usercode, e.g. within stub functions or test step epilogs. You can select the filter items on the left side to show only the evaluation macro results for e.g. the first test step. The list of results on the right will be filtered accordingly.

6.2.8 Console view

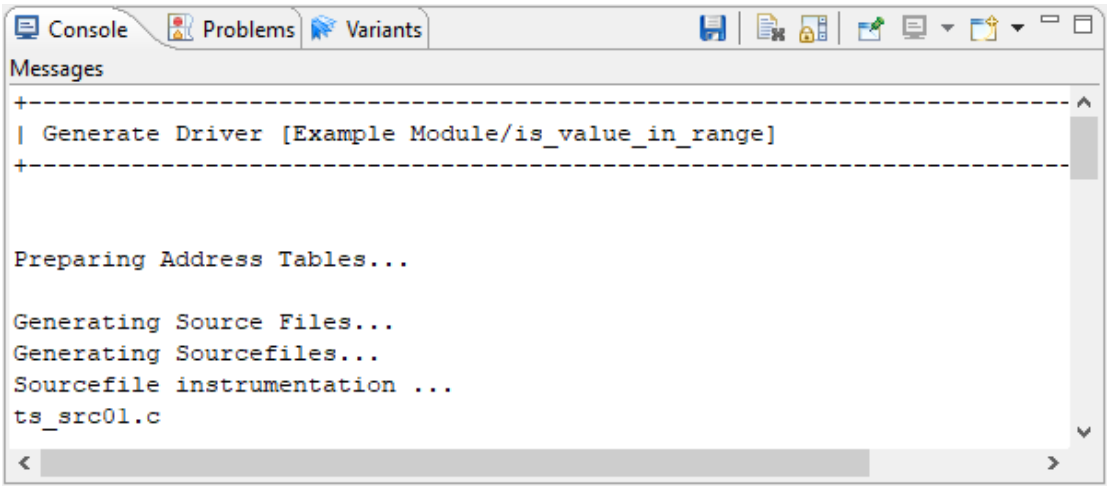


Figure 6.74: Console view

The Console view displays messages of sub processes invoked during the compilation and execution process of the test driver application. It provides a quick overview about any error messages.

6.2.8.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Saves to file.
	Clears the Console view.
	Locks the scrolling function.
	Pins the Console.
	Opens the console.

Table 6.22: Tool bar icons of the Console view

6.2.8.2 Handling

You can enable the console view to be shown whenever an error occurs during C-source analysis or test driver compilation:

→ In the menu bar click on “Window”.
→ Open the “Preferences”.
→ Click on “Test Execution Settings” and check the setting “Show console on error” (see figure 6.75.

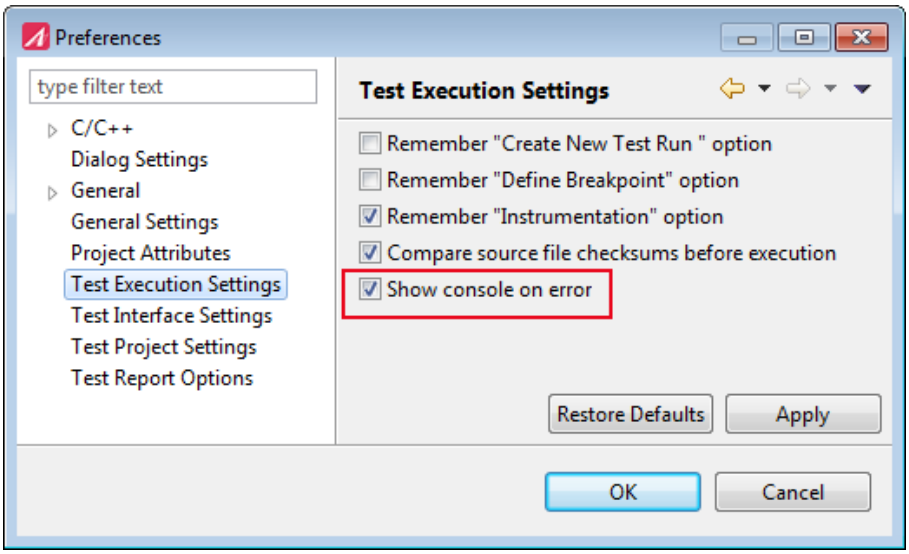


Figure 6.75: Preference “Show console on error”

6.2.9 Suspicious Elements view

Since the view refers to changes of requirements, this issue is discussed in section 6.4.8 Suspicious Elements view.

6.2.10 Problems view

In this view information about possible errors that appear e.g. in the process of test executions is displayed. It is divided into four columns: Message, Location, File and Line. The first column gives you a detailed error message with all necessary information. The other three contain all available information about where the error is located.

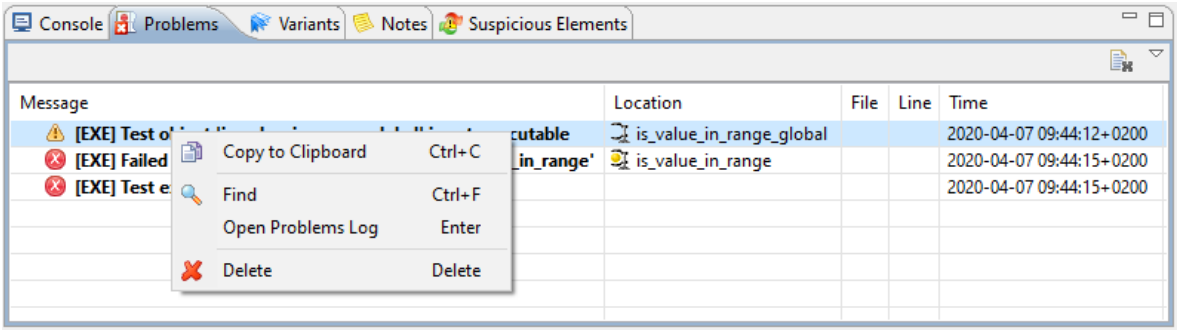


Figure 6.76: Problems view with error message

The Problems view also appears in the Requirement perspective.

Within the pull-down menu in the Problems view it is possible to find and select the respective module in the Test Project view in the Overview perspective, copy it to the clip board or open the Problems Log:

Icon	Action / Shortcut / Comment
	Copies to Clipboard.
	Finds the problematic area in the Project view.
	Deletes the message.

Table 6.23: Icons of the content menu

6.2.11 Variants view

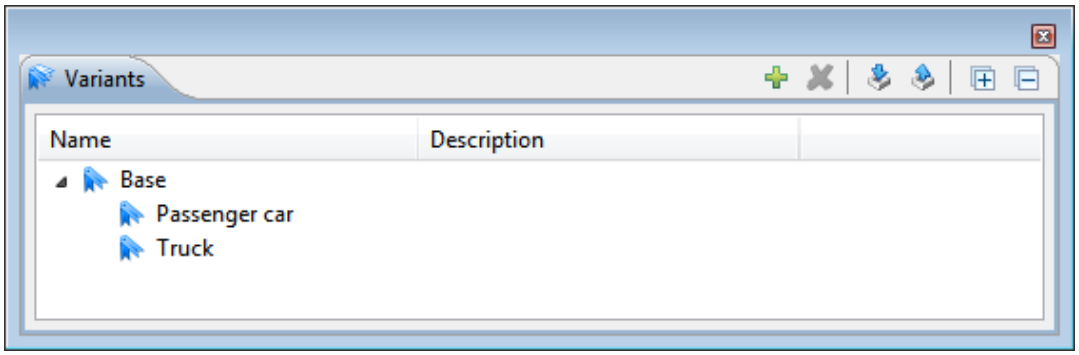


Figure 6.77: Variants View

The variants view supports the variant management in TESSY: You can create a variant tree according to the software variant structure you are going to test. These testing variants are useful for tagging TESSY modules to certain software variants which facilitates filtering and creation of variant TESSY modules.

Tip: You do not need to create a variant tree in order to create variant modules. Any module can be a parent module of another. The variant tree just helps to keep the module variant tree in sync with the actual inheritance structure of the software variants being tested.

A module can be assigned to a variant using the properties section “Variants”:

→ Select the module.
→ Switch to the “Variants” tab and select the respective variant.

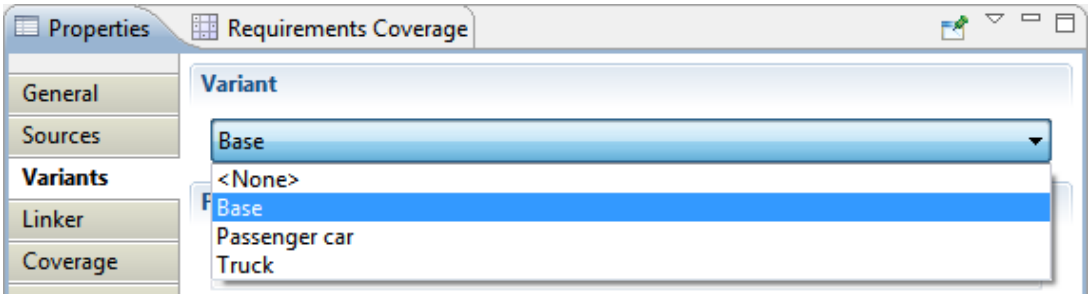


Figure 6.78: Assign a variant to a module

Let’s assume you have the following structure of base tests that shall be cloned as variant modules in order to test all software variants:

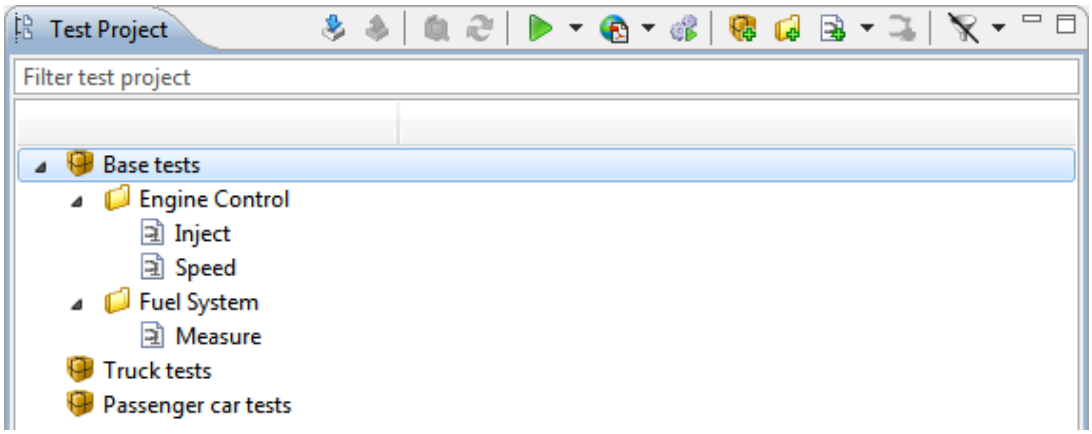


Figure 6.79: Example test collection with base modules

To create variant modules for each of the base modules do the following:

→ Create a new test collection for each variant and choose (New Variant Modules…) from the context menu.

→ Within the “Create Variant Modules” dialog select all base modules that shall be cloned as variant modules.

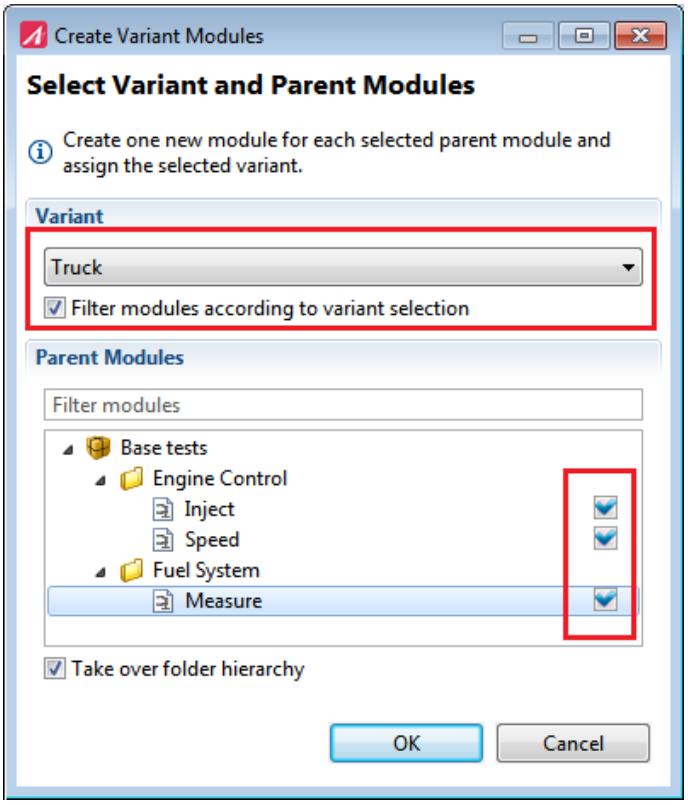


Figure 6.80: Create variant modules dialog: Filtering and selection

→ You can filter the modules being displayed by seleting the desired variant. Only potential parent modules according to the variant hierarchy will be displayed.

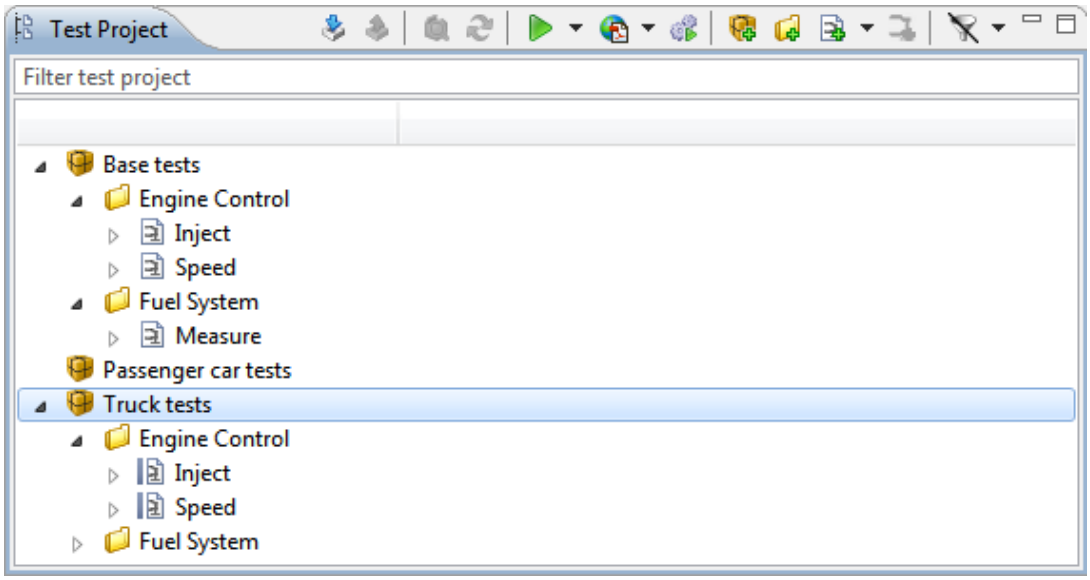


Figure 6.81: Test Project view with new variant modules

The new variant modules will be created within the selected test collection (including the folder hierarchy if the option “Take over folder hierarchy” was checked). The properties of a variant module shows the assigned variant and the parent module which can be edited using the edit button (see figure 6.82).

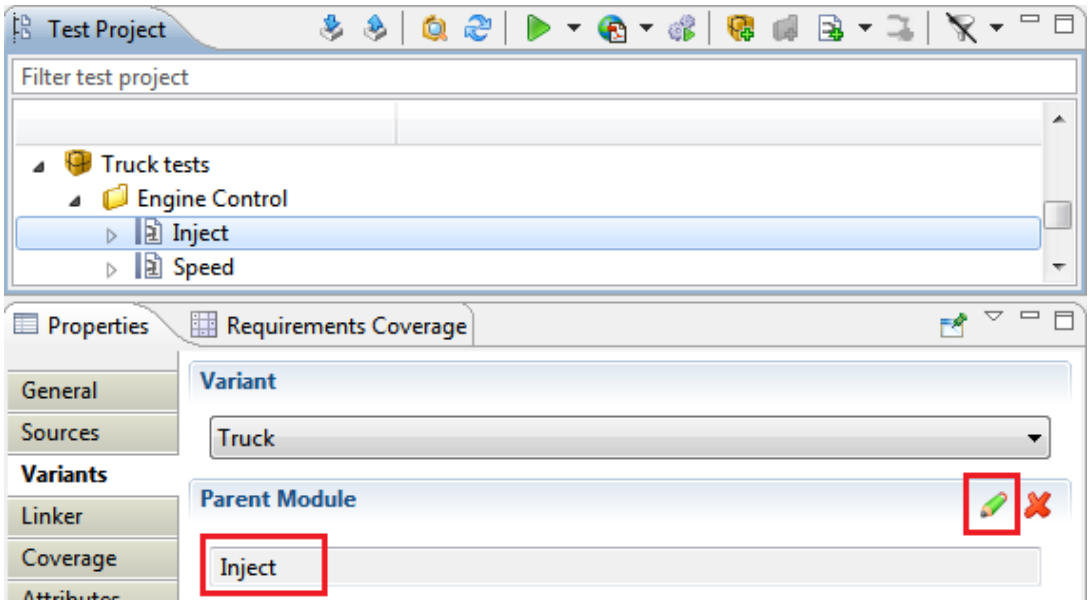


Figure 6.82: Properties view variants tab for editing the parent module

Important: All test data of a variant module will be deleted if you select another parent module.

6.3 C/C++: Editing the C-source

Important: The C/C++ perspective is not displayed by default! Open the perspective as described below!

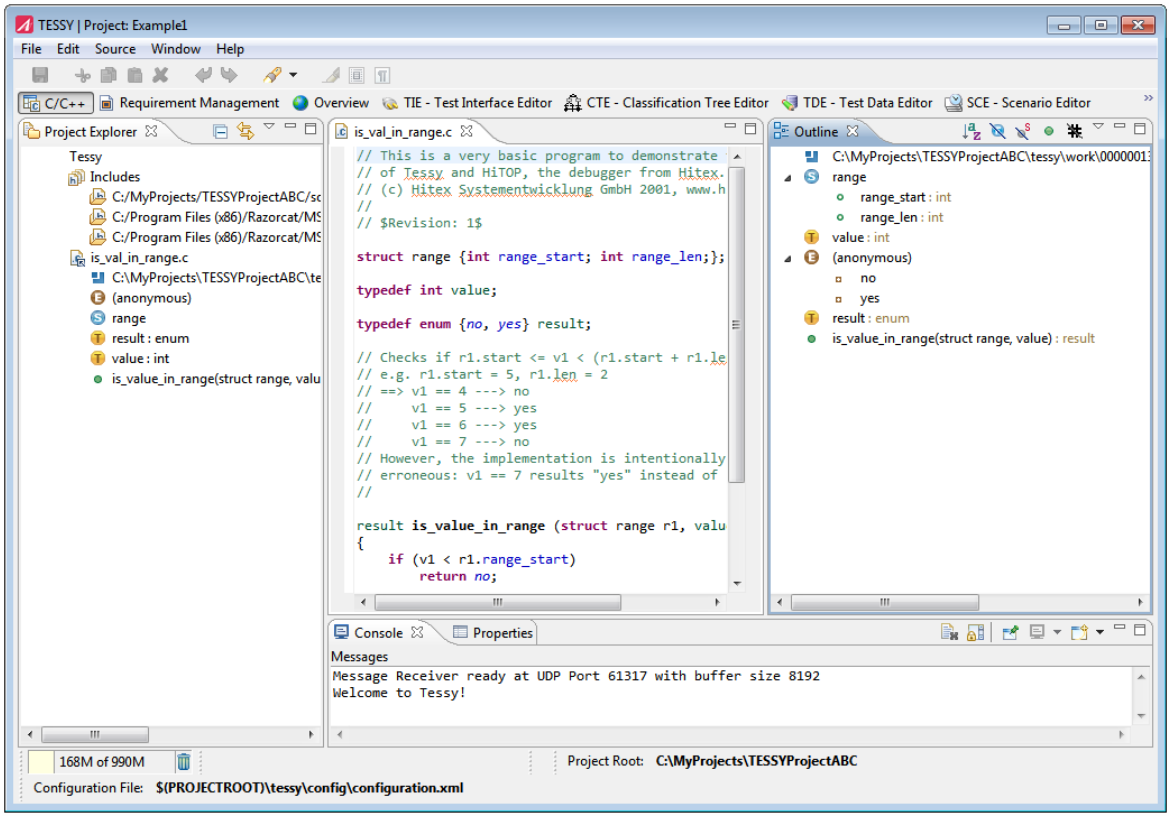


Figure 6.83: Perspective C/C++

Within the C/C++ perspective you can edit your C-source file.

6.3.1 Opening the C/C++ perspective

→ Within the Test Project view right-click the desired test object or module.
→ Select “Edit Source” (see figure 6.84).

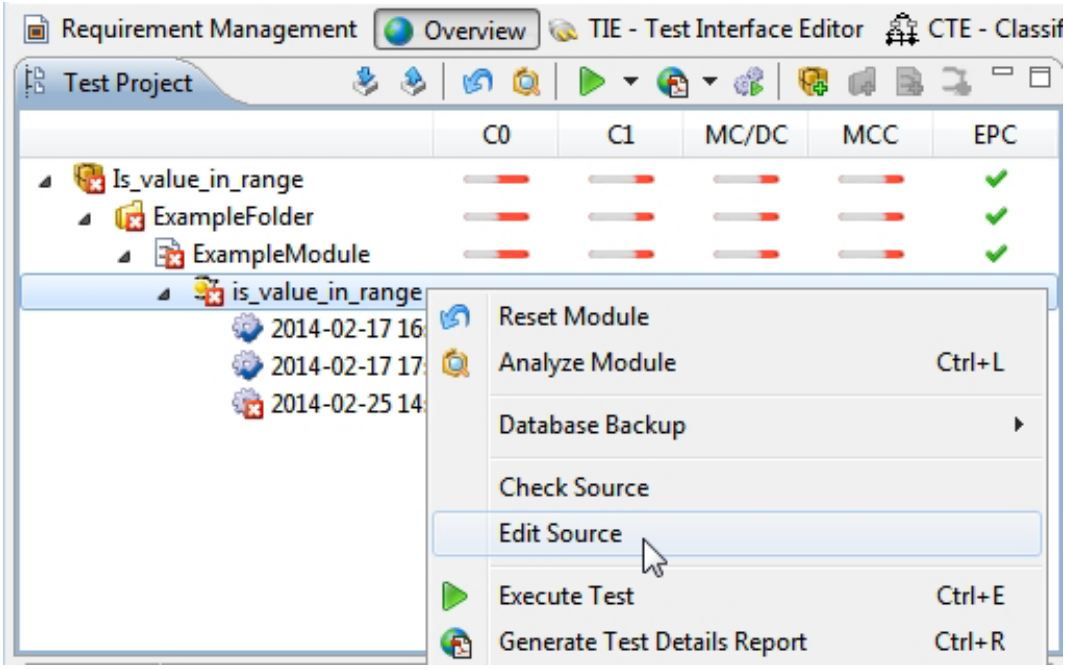


Figure 6.84: Opening the C/C++ perspective

6.3.2 Structure of the C/C++ perspective

Pane	**Location** (default)	**Function**
Project Explorer view	left	To view the includes and the functions of the C-source file.
Editor view	upper middle	To edit the C-cource file.
Outline view	right	To overview all functions of the C-source file.
Console view	lower middle	Same view as within the Overview perspective.
Properties view	lower middle	Same view as within the Overview perspective.

Table 6.24: Structure of the C/C++ perspective

Important: Most party of this view are usual Eclipse functions! Please refer to the Eclipse documentation: http://help.eclipse.org/

6.3.3 Editor view

Editor view

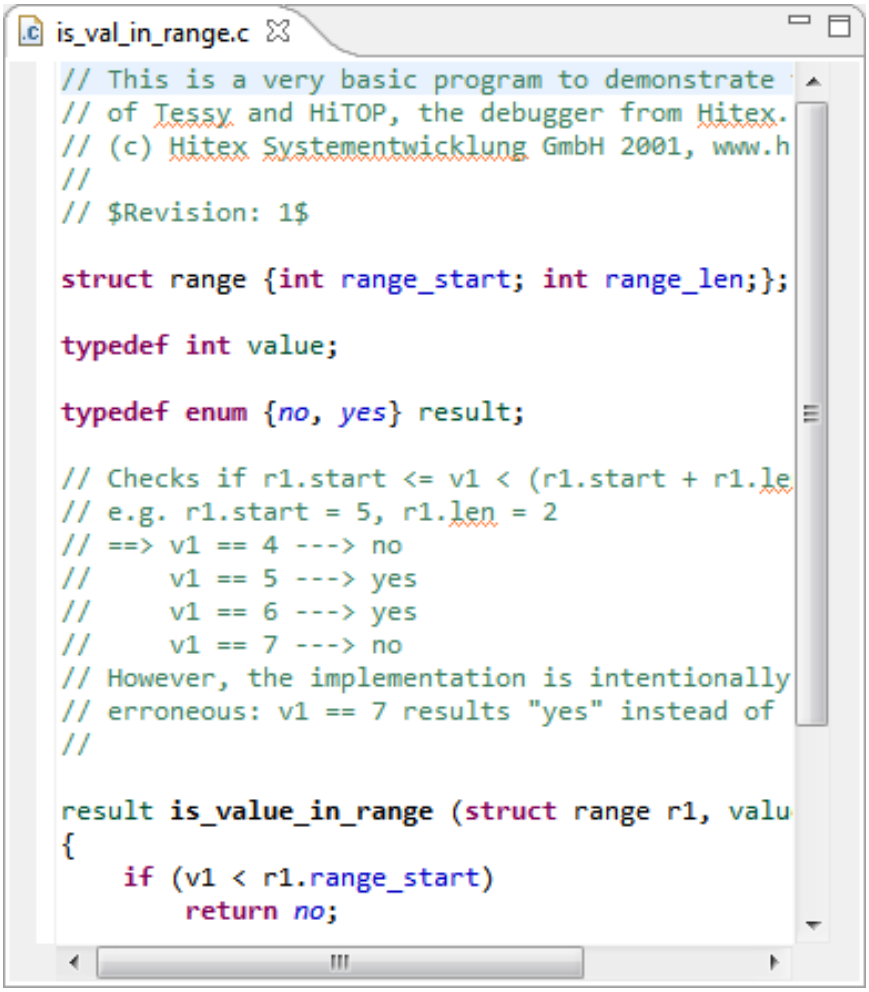


Figure 6.85: Editor view within the C/C++ perspective

Important: The Editor view is not a normal view in Eclipse sense, therefore you cannot move the view as other views of the perspectives!

6.3.3.1 Editing the C-source file

→ Open the C/C++ perspective with a right click on the desired test object or module within the Test Project view.

→ Select “Edit Source” (see figure 6.86).

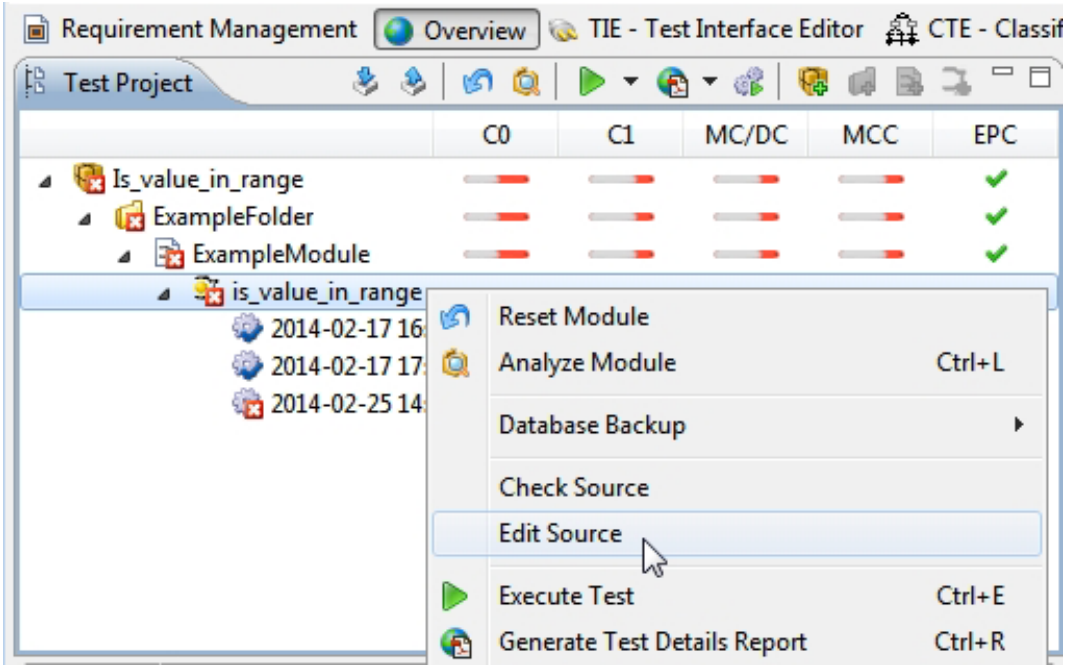


Figure 6.86: Opening the C/C++ perspective

→ Edit the C-source file.

The view is context sensitive: If you choose a function within the Outline view, the function will be highlighted within the Editor view!

6.3.4 Project Explorer view

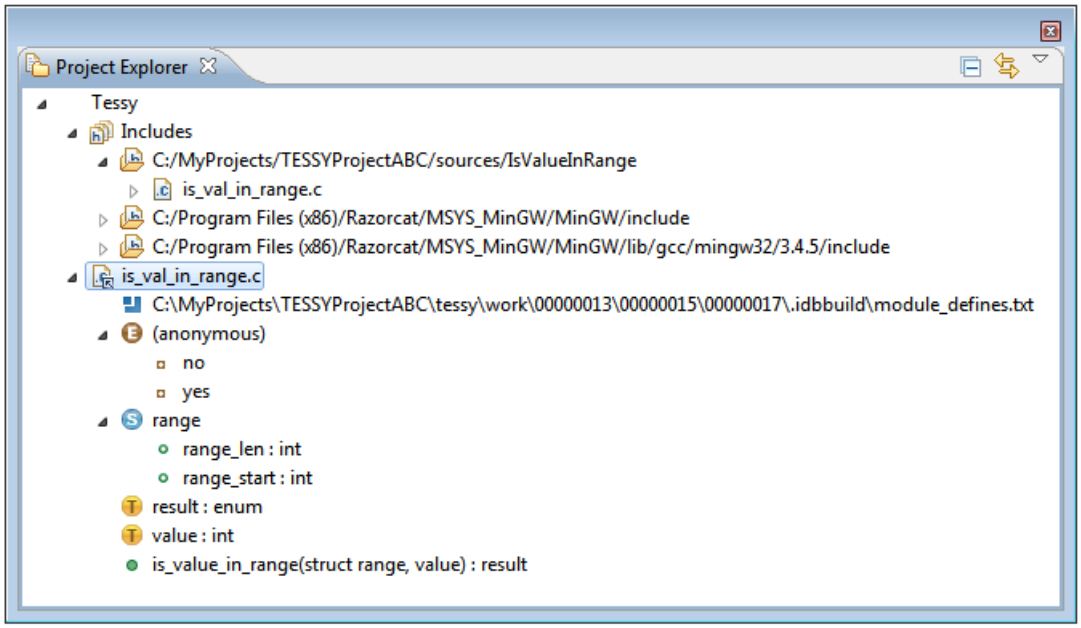


Figure 6.87: Project Explorer view within the C/C++ perspective

Within this view you can browse easily between the includes and have an overview of all functions of the C-source file.
The view is context sensitive: If you choose a function within the Project Explorer view, the function will be highlighted within the Editor view.

6.3.5 Outline view

Outline view

The Outline view displays all functions of the C-source.

The view is context sensitive: If you choose a function within the Outline view, the function will be highlighted within the Editor view (see figure 6.88).

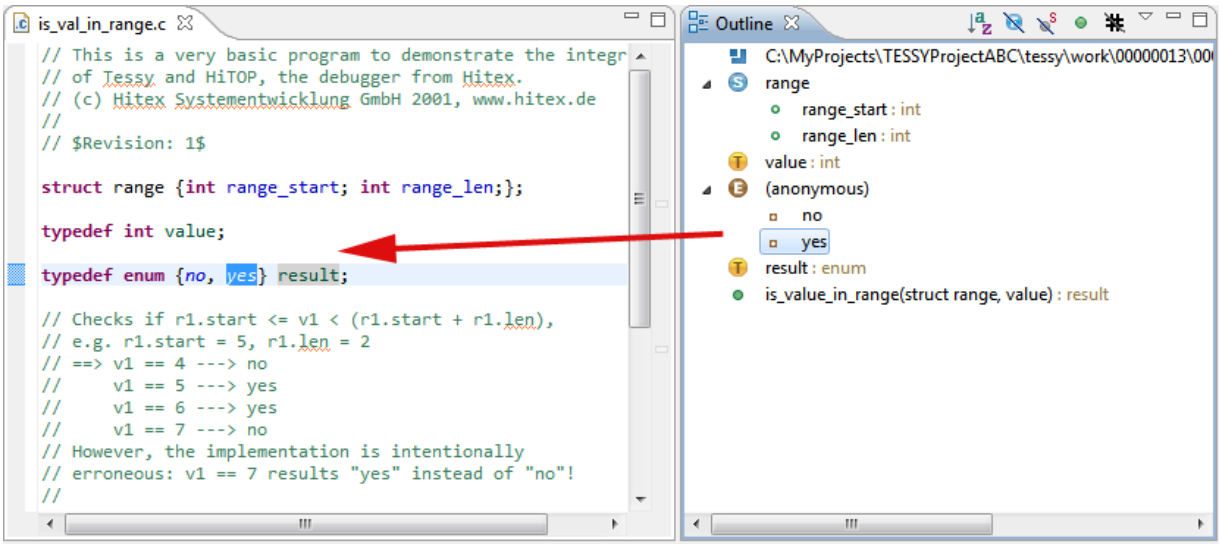


Figure 6.88: Outline view within the C/C++ perspective

6.3.5.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Sorts the functions in alphabetica order.
	Hides fields.
	Hides static members.
	Hides non-public members.
	Hides all inactive elements.

Table 6.25: Tool bar icons of the Outline view

6.3.6 Properties view

The Properties view displays all the properties which you organized within the Overview perspective. Most operations are possible.

6.2.3 Properties view

For changing a source switch to the Properties view within the Overview perspective.

6.3.7 Console view

The Console view displays messages of sub processes invoked during the compilation and execution process of the test driver application. It provides a quick overview about any error messages. Same view as within the Overview perspective, see section 6.2.8 Console view.

6.4 Requirement management

The basis for all testing activities should be a precise functional specification of the system under test. All testing activities should be caused by requirements described within the functional specification and each change of the requirements need to be tracked in order to adjust the tests if necessary. That is the reason why TESSY incorporates a requirement management solution that provides a basic environment for requirements engineering with the following features:

- Exporting and importing requirements
- Creating new requirements
- Comparing requirement versions
- Linking test cases with requirements
- Automatic versioning of requirement changes
- Adding images to requirements
- Marking different versions of requirements as semantically equivalent

There is a plugin available for the integration of Polarion. For more information please refer to the application note “Polarion Export” in TESSY (“Help” > “Documentation”)

You will use different views and perspectives for your requirement management:

1. To create and import requirements , track changes and versionize your requirements use the Requirement Management perspective.

6.4.1 Structure of the Requirement Management perspective

2. To link requirements with test cases use the Link Matrix view or the Requirements Coverage view of the Overview perspective.

6.4.16 Requirements Coverage view

6.4.1 Structure of the Requirement Management perspective

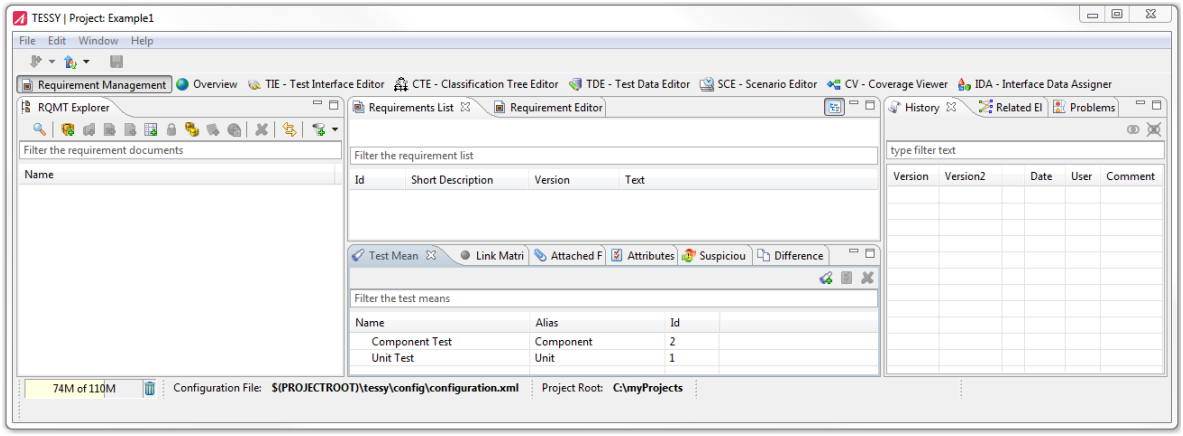


Figure 6.89: Requirement Management perspective

Pane	**Location** (default)	**Function**
RQMT Explorer view	left	To create requirements.
Requirements List view	upper center	To view imported requirements as list for a selected document or folder.
Requirement Editor view	upper center	To organize the requirements, e.g. adding information as text or images, opens only after double-clicking on a requirement in the RQMT Explorer view.
Test Means view	lower center	To list the available test means, up to unit test and component test.
Validation Matrix view / VxV Matrix view	upper center	To assign requirements to test means, only visible when there is a validation matrix and after double-clicking on it.
Link Matrix view	lower center	To link requirements with modules, test objects, test cases and other requirements.
Attached File view	lower center	To attach files to requirements.
Attributes view	lower center	To edit a list of attributes for a requirement.

continue next page

Pane	**Location** (default)	**Function**
Suspicious Elements view	lower center	To have a quick look over all suspicious (modified) elements.
History view	right	To display the version history of the selected requirement or document.
Related Elements view	right	To display linked elements for a selected requirement and compare these versions.
Problems view	right	
Document Preview	right	To edit the HTML contents of the requirements, only visible after double-clicking the respective requirement.
Requirements Coverage view	**Overview perspective!**	To select and link the requirements.

Table 6.26: Structure of the Requirement Management perspective

Important: To gather all information about managing requirements within this chapter, we will describe the Requirement Coverage view in this chapter, although the workflow makes it reasonable to place this view within the Overview perspective.

6.4.2 RQMT Explorer view

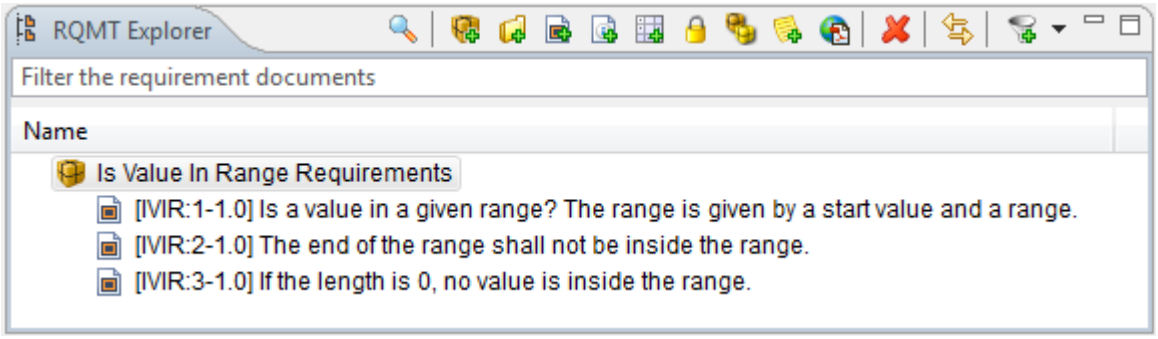


Figure 6.90: RQMT Explorer view

RQMT Explorer view

The RQMT Explorer view displays an overall view of all requirements of a requirement document. If you double-click a requirement, the requirement editor will open to display all information of the specific requirement (see figure 6.91).

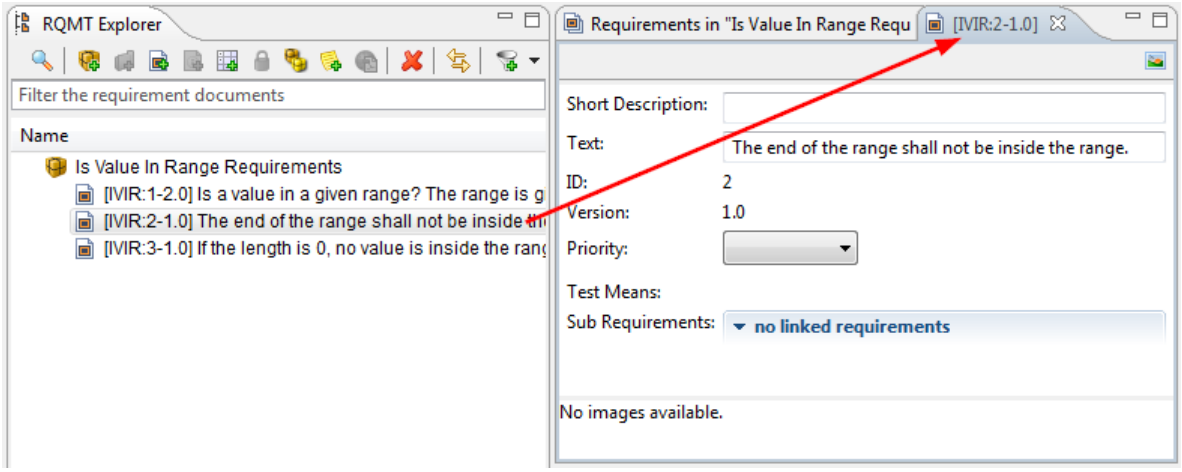


Figure 6.91: Double-clicking on a requirement opens the requirement editor

6.4.2.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Creates a new document.
	Creates a new chapter.
	Creates a new requirement.
	Creates a new text section.
	Creates a new validation and verification matrix.
	Locks the marked item.
	For editing the document hierarchy.
	For adding notes.
	Generates a document report.
	Deletes the marked item.

continue next page

Icon	Action / Shortcut / Comment
	Syncs selection with editor.
	Adds filter.

Table 6.27: Tool bar icons of the RQMT Explorer view

Please note that with a right click on an element usually a context menu opens. It contains the same buttons as shown in table 6.27. Depending on the circumstances there might be more options available.

6.4.2.2 Status indicators

Indicator	Status / Meaning
	The requirement document is locked.

Table 6.28: Status indicators of the RQMT Explorer view

6.4.2.3 Creating requirements and organizing a document structure

Creating requirements

To create requirements:

→ In the RQMT Explorer view tool bar click on (New Document).
→ Then click on (New Requirement (Insert)) to create a requirement.

The RQMT Explorer view also offers the option to organize the document structure. You can adapt it to your needs and therefore gain a better overview over sometimes numerous elements. This includes the opportunity to add chapters and text elements to documents as well as text elements to chapters (see figure 6.92).

To create new elements of all kind in the document structure use the RQMT Explorer tool bar (see table 6.27) or the context menu after a right click on the respective element. By default new elements are placed at the end of the document or chapter column and new documents appear on document level. It is possible to drag chapters, test elements and requirements into the desired position, even into other documents.

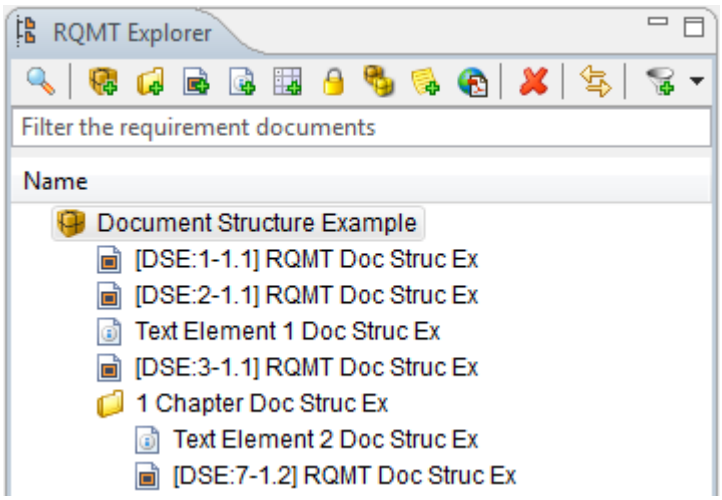


Figure 6.92: Example for the document structure within the RQMT Explorer view

6.4.2.4 Importing requirements

In the following you find a brief overview about importing requirements. For more detailed information please refer to the application note “Importing Exporting Requirements” in TESSY (“Help” > “Documentation”)

→ Right-click a document or right-click within the blank RQMT Explorer view and select “Import” from the context menu (see figure 6.93). When no document is selected, the import will create a new document.

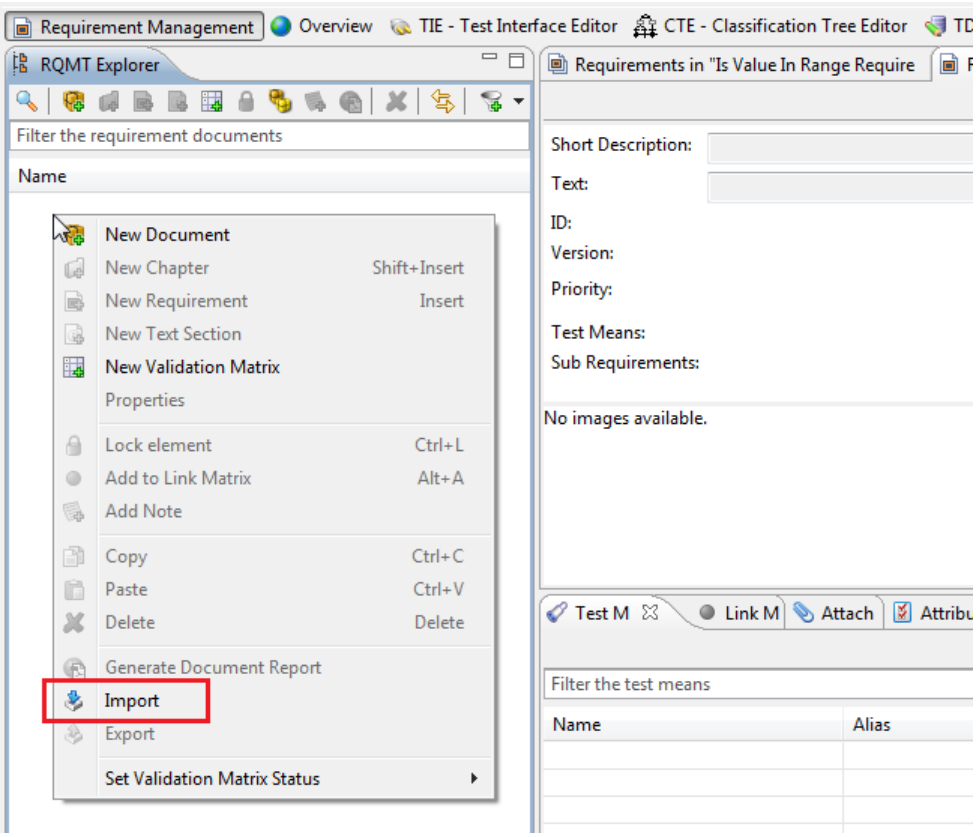


Figure 6.93: Importing requirements

→ The Import dialog opens (see figure 6.94).

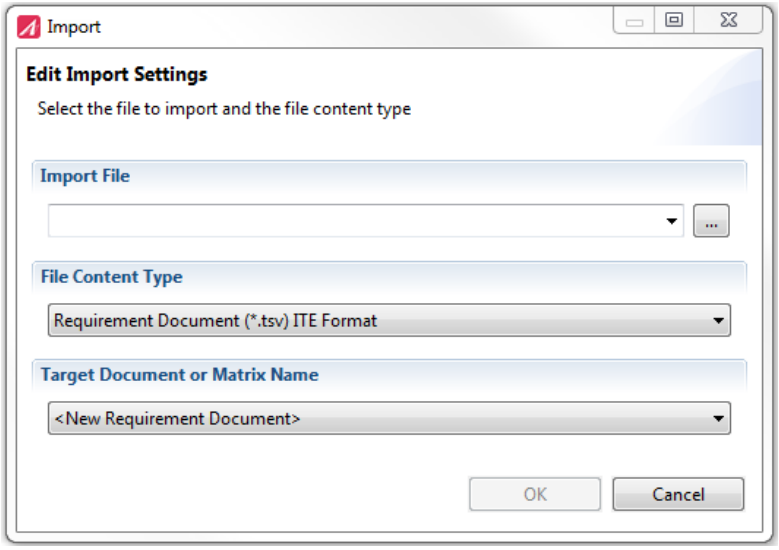


Figure 6.94: Import dialog

→ Click on "…" and select the file with the requirements.

→ Select the File Content Type. For the possible types see table 6.29.

TESSY will pre-select the content type according to the contents of the currently selected file. If the file cannot be imported, this field will be empty. You can select a content type that you think the file contains and start importing. TESSY will then show the errors within the file.

→ Select the Target Document and click “OK”.

If no existing document has been selected, the new requirements will be imported into a new document. You can also select any existing document within the list to update this document. Whether you will see existing requirement documents or validation matrices depends on the file content type.

Formats of requirement sources

You can import following formats of requirement sources:

Format	Comment
*.txt	Simple ASCII format were each line is recognized as a requirement. This is the very basic format that allows importing all sorts of text as requirements.
*.reqif	Requirements interchange format (ReqIF), used for import/export of requirements from third party requirement management tools (e.g. DOORS and Polarion). For further information see http://www.omg.org/spec/ReqIF
*.csv, *.tsv	Comma or tab separated ASCII text with a headline indicating the column names. This format allows specifying requirement id, version and all other available requirement properties.
*.xml	TESSY specific XML format which provides specifying the chapter structure of a requirement document. All available requirement properties may be specified within this format. It is the recommended exchange format when importing requirements from any other requirement management systems.

Table 6.29: Possible formats of requirement sources

The newly imported requirement document will be displayed in the RQMT Explorer view (see figure 6.95).

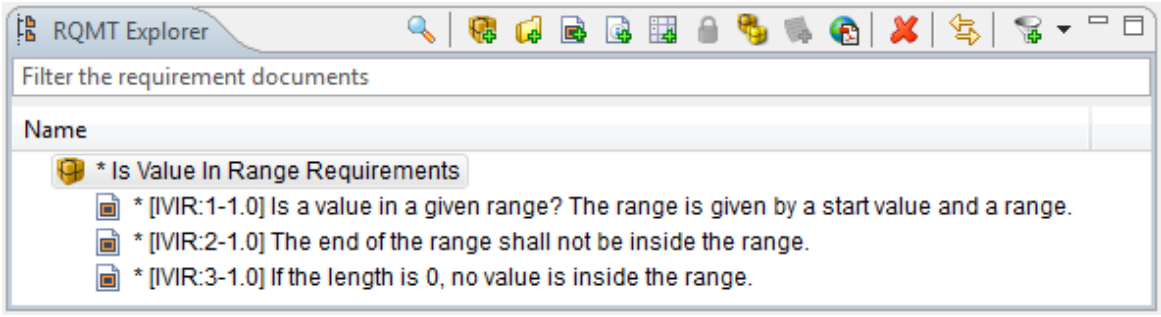


Figure 6.95: The new requirement document

New imported requirement

The asterisk (*) indicates that the requirement is new and not committed yet. A mouseover shows a tooltip (see figure 6.96).

Important: New imported requirements need to be committed (see section 6.4.2.5 Committing requirements).

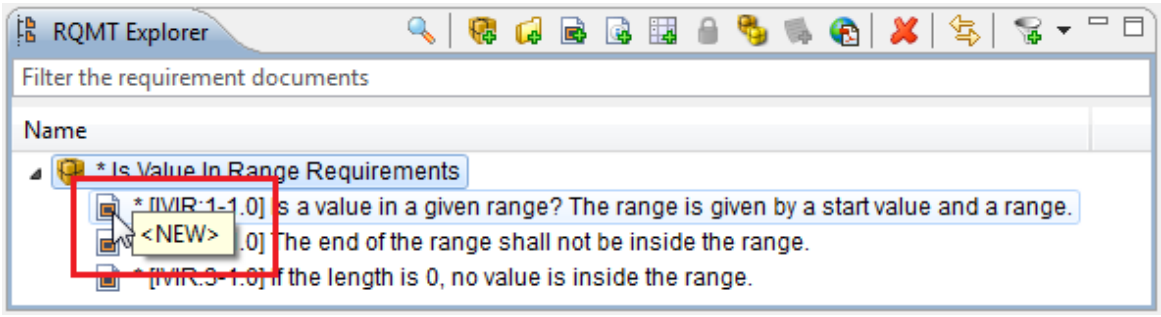


Figure 6.96: The asterix and a mouseover shows the status “new”.

6.4.2.5 Committing requirements

Committing requirements

The requirements document needs to be checked-in as initial revision:

→ Select the document within the RQMT Explorer view and click on (Commit Changes) in the global tool bar.

You can commit all changes or changes of selected elements (see figure 6.97).

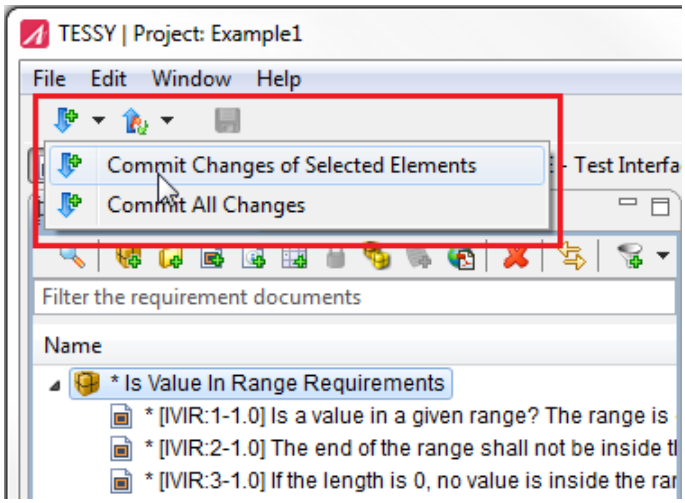


Figure 6.97: Committing options

→ Enter a comment and click “OK” (see figure 6.98).

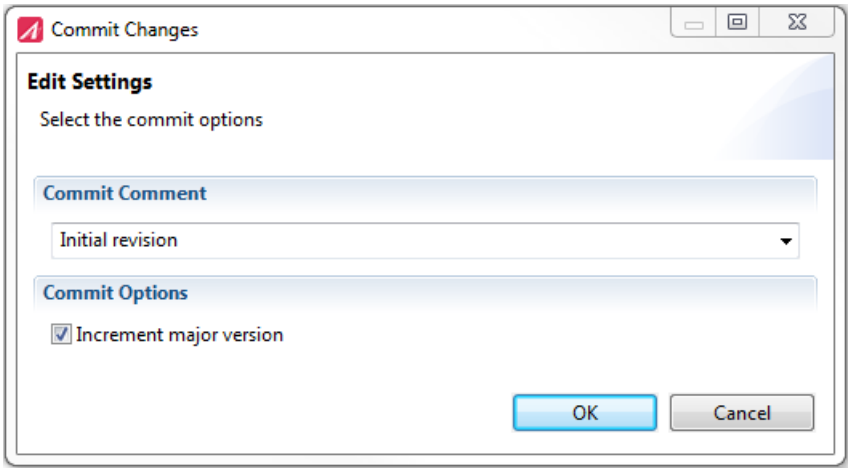


Figure 6.98: Comment for the initial revision of the commit

An initial revision of the requirement document will be created.
By ticking the box “Increment major version” a major version with a new version number will be created.

If requirements have been changed, every commit creates a new requirement version. The reason for this is traceability, only with these different requirement versions changes in the requirements can be traced.

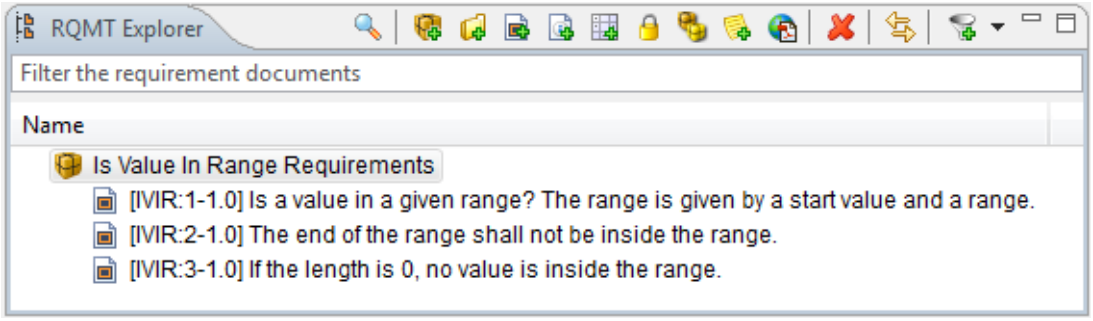


Figure 6.99: After the commit

You can also discard the changes you made by clicking on (Discard Changes) in the global tool bar. This will restore the last checked in status. With a click on the little arrow next to the icon you can set whether you want to discard all changes or changes of selected elements only.

6.4.2.6 Renaming a document / alias

You can rename a requirement document and assign an alias which is useful for the reporting, because you have an abbreviation of the document name when building the requirement identifier. The identifier will be: [document alias]:[id]-[version]. To rename or give an alias:

→ Right-click the document and select “Properties” from the context menu.

→ Change the name or choose an alias (in this Project: “Example1” it is “IVIR’) and click “OK”.

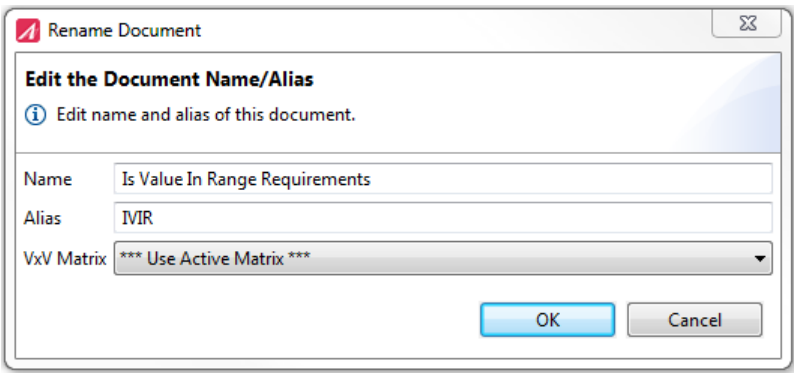


Figure 6.100: Changing the alias of the new requirement document

The new alias “IVIR” will be used within the Requirements List view and the document preview (see figure 6.101). (The document preview will only be visible after double-clicking the respective requirement.)

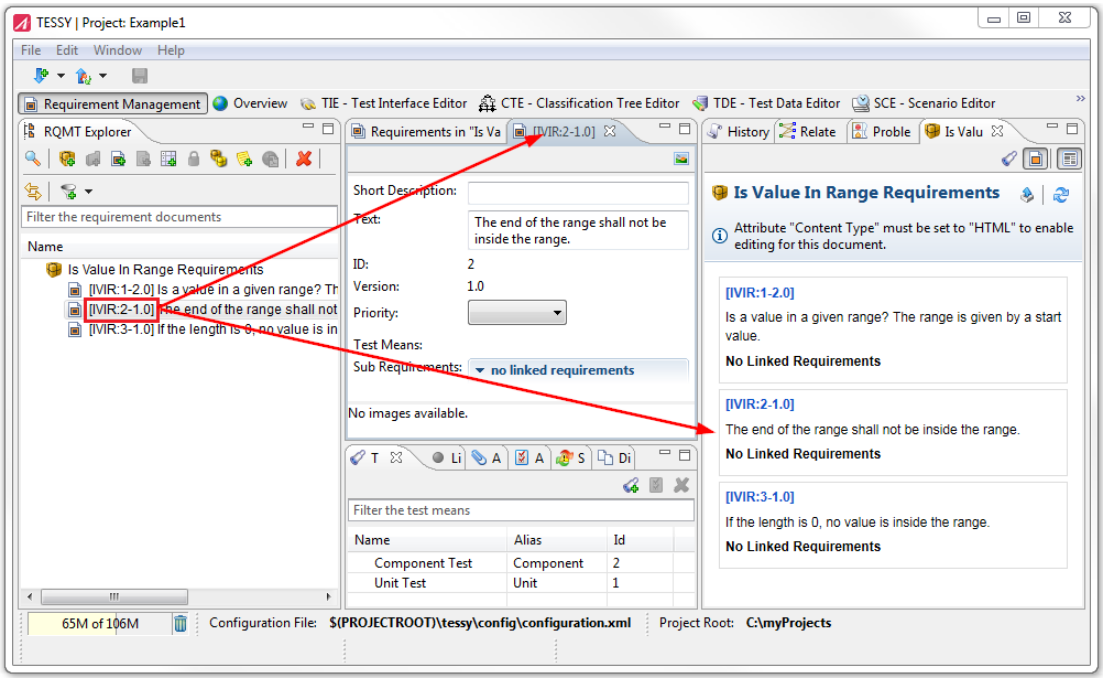


Figure 6.101: The alias of a requirement is used in various views

Information about editing requirements can be found in section 6.4.4.3 Editing requirements.

6.4.3 Requirements List view

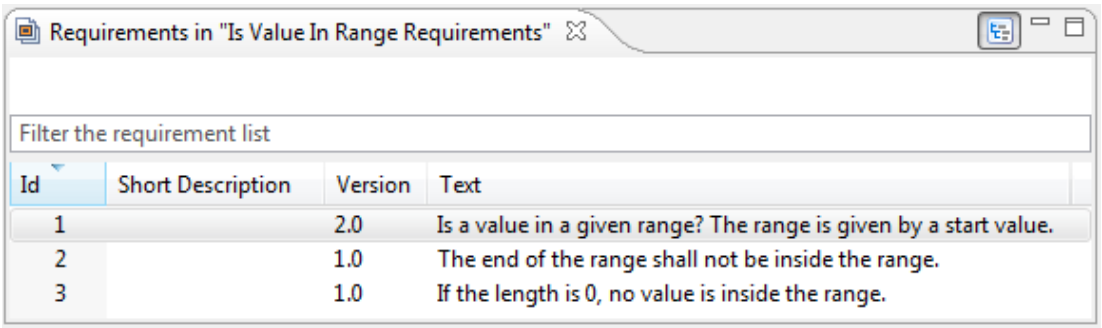


Figure 6.102: Requirements List view

6.4.3.1 Icon of the view tool bar

Icon	Action / Shortcut / Comment
	Shows all descendants.

Table 6.30: Tool bar icons of the Requirements List view

6.4.4 Requirement Editor view

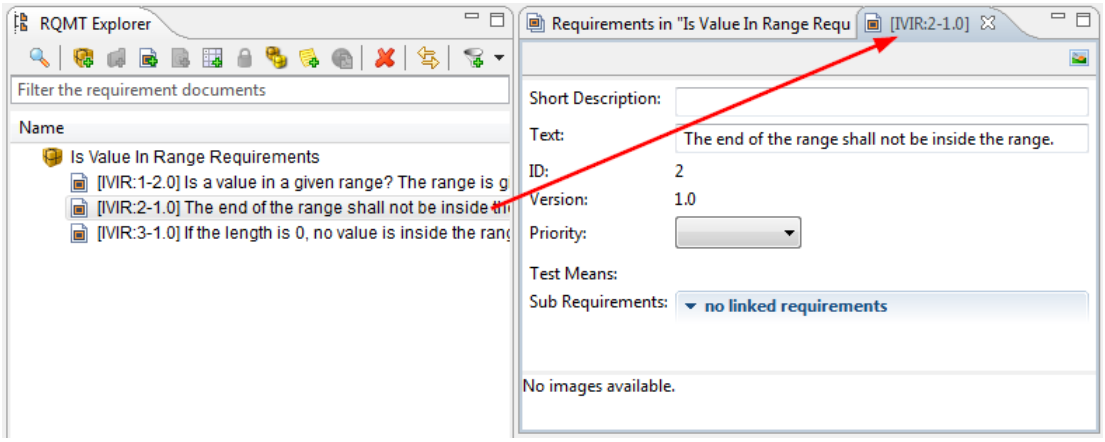


Figure 6.103: Double-clicking on a requirement opens the Requirement Editor

The Requirement Editor will also open by double-clicking in a requirement in the Requirement List view.

6.4.4.1 Icon of the view tool bar

Icon	Action / Shortcut / Comment
	Adds an image to the element.

Table 6.31: Tool bar icon of the Requirements List view

6.4.4.2 Viewing requirements, versions, IDs

Within the Requirement Editor the requirements are displayed with text, figures, if available, versions and IDs (see figure 6.104).

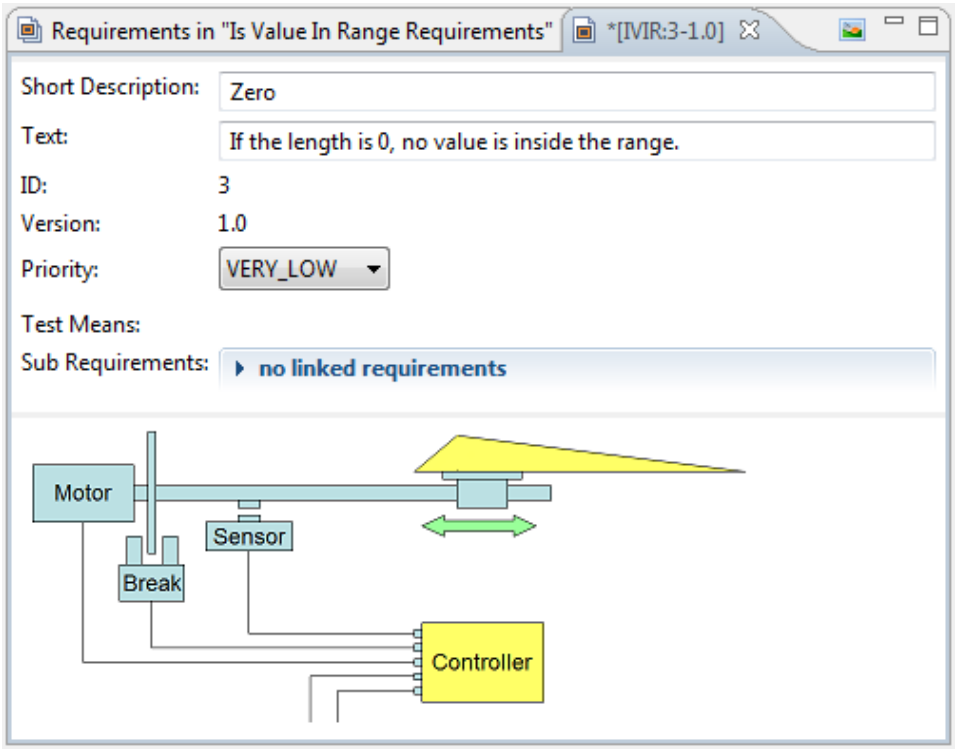


Figure 6.104: Requirements Editor with test and a figure

Requirements List view

Every requirement has an explicit ID and a version number. TESSY provides the following two mechanisms for assigning requirement version numbers:

- **Automatic**: TESSY assigns and increments version numbers automatically if the import file does not contain version numbers. This is the normal behavior when importing requirements from text files or spreadsheets. When checking-in requirements, you can decide to increment the major or minor version number (e.g. major 1.0 to 2.0 or minor 1.0 to 1.1).
- **Using external version numbers**: It is possible to use existing requirement version numbers from the import file. This is useful if you get requirements from an external requirements management tool and need to import exactly the same version numbers that where already assigned to the requirements. The only prerequisite for this kind of import is the consistency of the version numbers.

Using external version numbers

When using external version numbers, the following checks of the imported data will be performed when importing:

- If any requirement content is changed but the version number is not changed, TESSY will change the minor version number (e.g. from 1.0 to 1.1).
- If the version number was changed but no requirement content was changed, a warning will be reported.
- If the new version number is less than the highest existing version number for a requirement, an warning will be reported.

Gaps within the numbering of requirement IDs are allowed.
The numbering should be ascending.

6.4.4.3 Editing requirements

To edit a requirement:

→ Double-click on the requirement you want to edit.
The requirement will be opened within the requirement editor in the center of the Requirement Management perspective.

→ After editing, click on to save the changes.
TESSY will now create a locally modified version of the requirement which will be illustrated with a “>” in front of the requirement name (see figure 6.105).

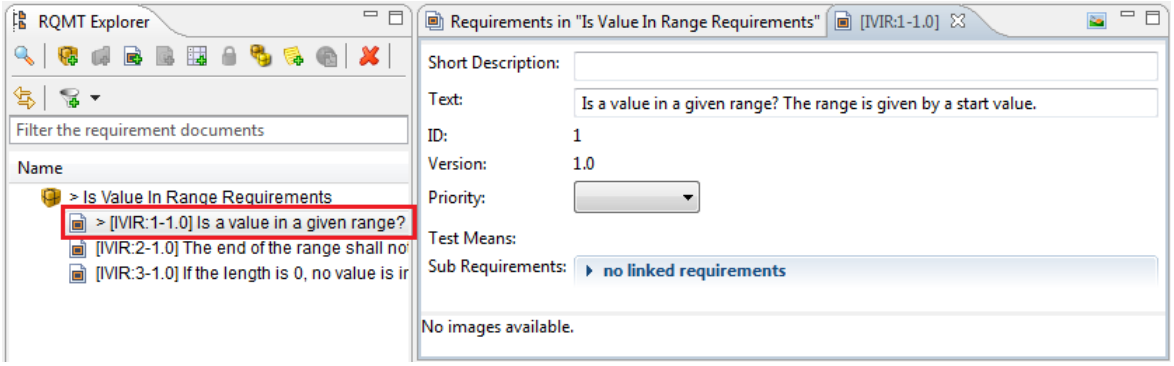


Figure 6.105: The first requirement was modified

→ Commit your changes with a click on .

→ A new version of the requirement will be created and you need to decide either to increment the major or the minor version number within the check-in dialog (see figure 6.98).
If you did only minor changes or want to commit a draft update of a requirement, you can decide to increment only the minor version. In all other cases, it is recommended to increment the major version.
After committing, the ID of the requirement will be updated to display the new version (see figure 6.106).

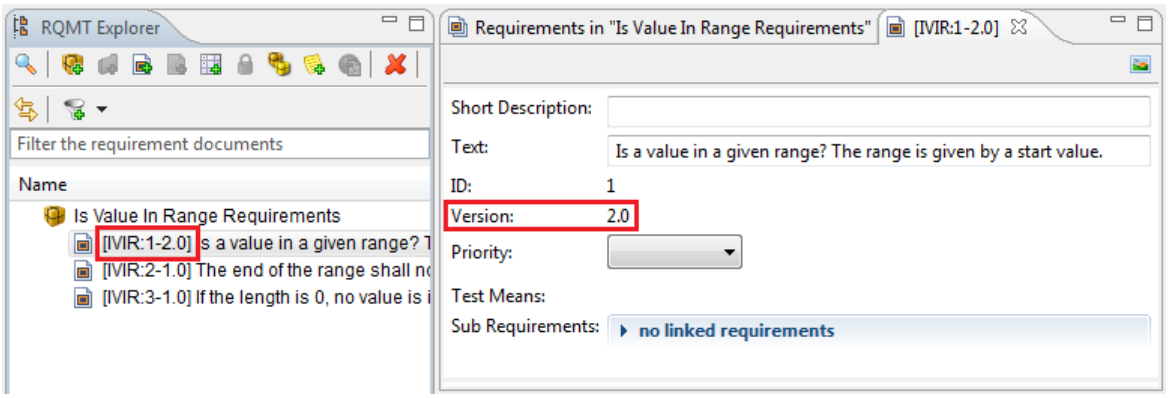


Figure 6.106: The first requirement has the version number 2.0

6.4.5 Validation Matrix view / VxV Matrix view

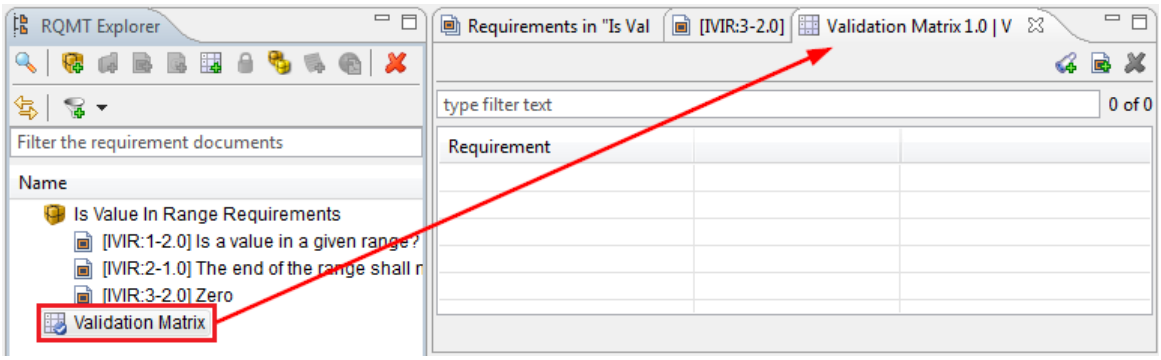


Figure 6.107: VxV Matrix view

The VxV matrix supports the assignment of requirements to the test means used for validation of the requirement. This helps filtering out those requirements that are to be tested with unit and component testing. The assignments within the VxV matrix will be used for requirement filtering for reporting.

6.4.5.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Adds all test means to the matrix.
	Adds all requirements to the matrix.
	Removes all unused items.

Table 6.32: Tool bar icons of the VxV Matrix view

6.4.6 Test Means view

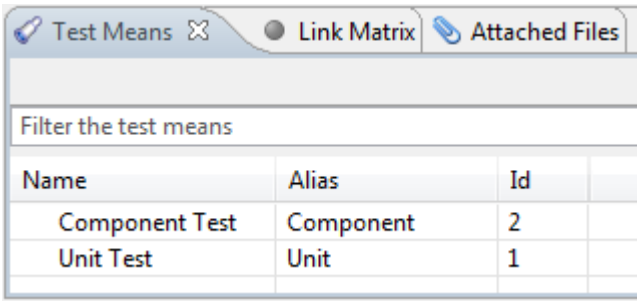


Figure 6.108: Test Means view

Requirements will be tested using different test means, e.g. unit test, system test or review. The default test means used within TESSY are for unit and component testing. You can filter your requirements by test mean for later reporting issues.

6.4.6.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Imports test means.
	Exports test means.
	Creates a new test mean.
	To edit the selected test mean.

	Deletes the selected test mean (Del). Only test means that are not used can be deleted.

Table 6.33: Tool bar icons of the Test Means view

6.4.7 Link Matrix view

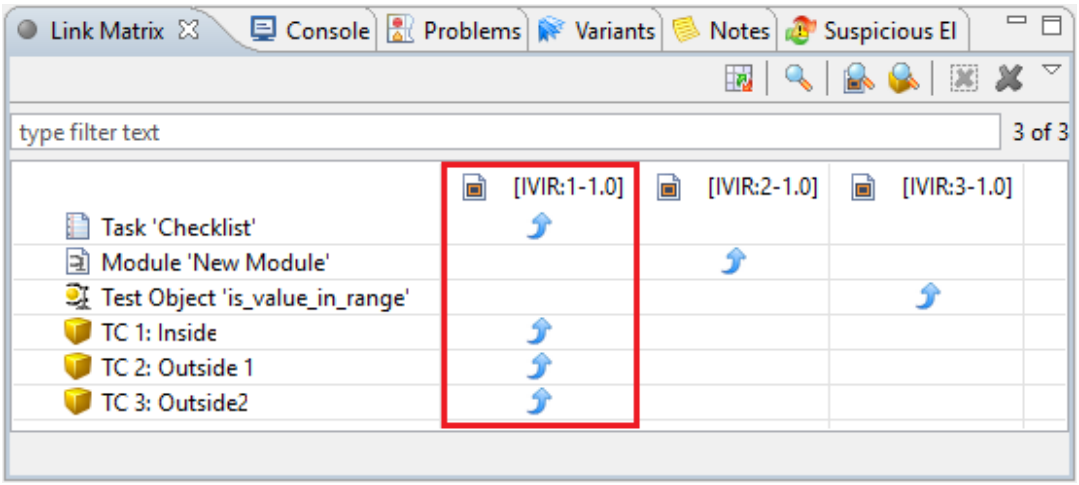


Figure 6.109: Link Matrix view

Link Matrix view

Within the Link Matrix view you can link tasks, modules, test objects and test cases with requirements. It shows the link relationship of the elements currently contained within the matrix.

In the example above (see figure 6.109) the requirement [IVIR:1-1.0] is linked with three test cases and one task, i.e. in total there are four tests linked to that requirement.

6.4.7.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Transposes the matrix, i.e. changes the rows and columns (Ctrl+T).
	Adds all currently linked elements based on the elements selected within the first column of the matrix.
	Adds all unlinked requirements.

	Adds all unlinked test cases and tasks.
	Removes selected elements (Delete) from the Link Matrix. Does NOT delete the element but removes it from the view.
	Removes all elements from the Link Matrix. Does NOT delete the element.

Table 6.34: Tool bar icons of the Link Matrix view

6.4.7.2 Status indicators

Indicator	Status / Meaning
	The link is suspicious. Indicates that changes in the respective requirement need to be updated.
	No link possible. Indicates that it is not possible to add a link here.

Table 6.35: Status indicators of the Suspicious Elements view

6.4.7.3 Adding and removing elements

To add elements to the Link Matrix view:

→ Drag & drop requirements, modules, test objects or test cases into the matrix. The elements will be shown within one of the rows in the first column if they are dropped there. If they are dropped in one of the right columns, they will appear on top of the respective rows of the matrix.

To exchange rows or columns, click on within the view tool bar.

→ Click on to add all unlinked requirement.

→ Click on to add all unlinked test cases.

→ Use the context menu entry “Add to Link Matrix” within the RQMT Explorer view, Test Project view or Test Items view.

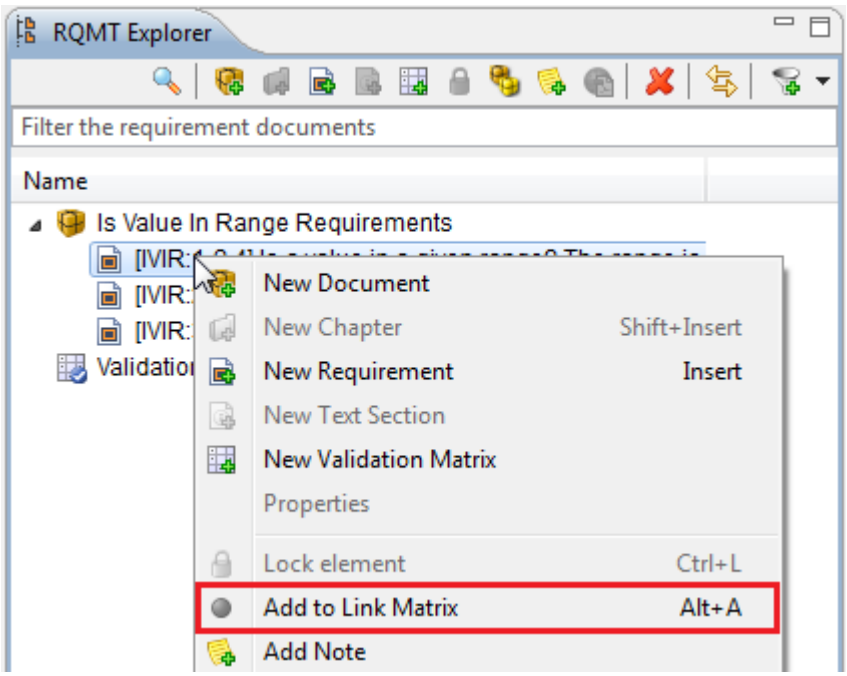


Figure 6.110: Adding elements to the Link Matrix view

To remove elements from the Link Matrix view:

→ Click on (Remove All Elements) in the tool bar to remove all currently displayed elements.

→ Click on (Remove Selected Element) in the tool bar will remove the currently selected element within a row of the matrix (if any element is selected).

This will only remove the elements from the matrix view, no changes will be made to the elements themselves. They are not deleted in the process and set links remain unchanged.

Important: Test cases can not be added to the Link Matrix view in the Requirement Management view. To do so you have to switch to the Overview perspective (see figure 6.111). Test cases can also be added to the Link Matrix in the TDE perspective or the SCE perspective.

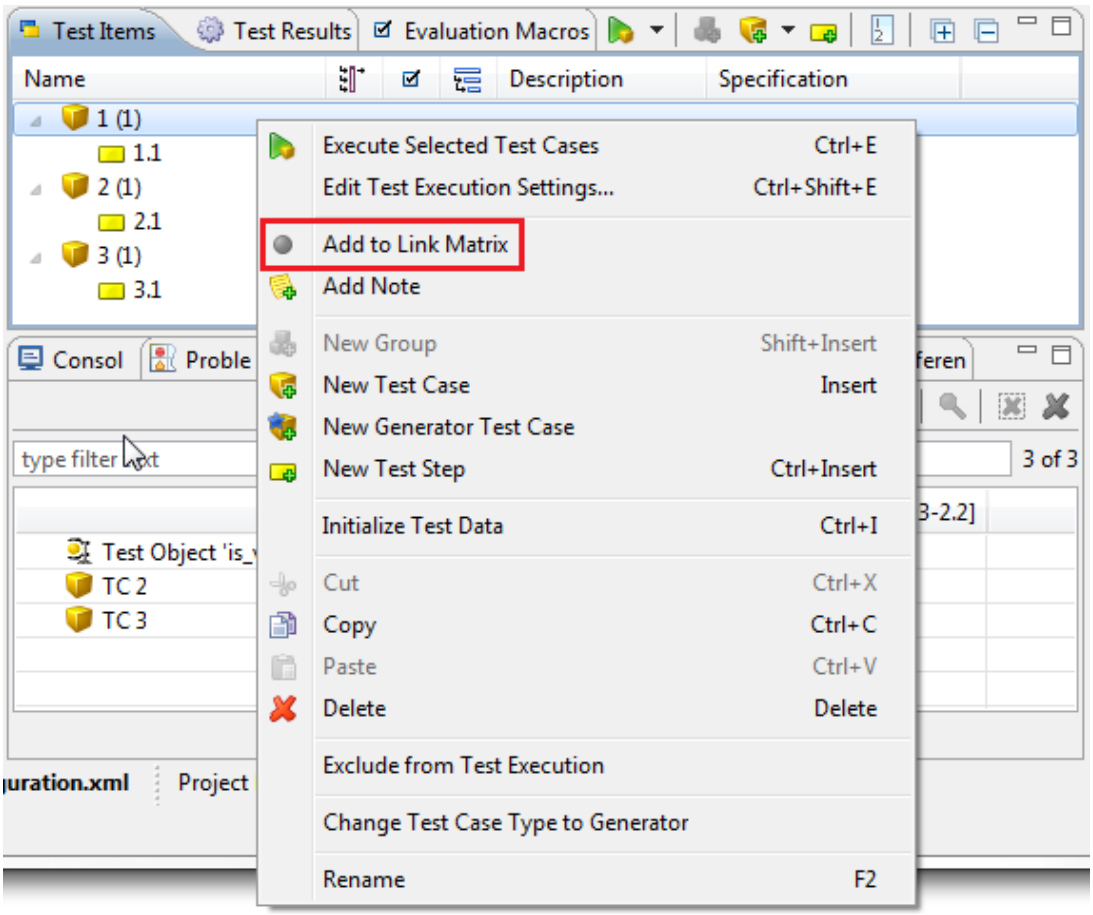


Figure 6.111: Adding Test Cases to the Link Matrix view in the Overview perspective

Please notice the following habits:

- The Link Matrix view is available within the Overview perspective and within the Requirement Management perspective.
- The Link Matrix view will also be visible within the TDE perspective and the SCE perspective if elements had already been added.
- The current contents of the Link Matrix are remembered when restarting TESSY but the matrix itself is not persisted in any way. You can add or remove elements and this will not cause any changes to the elements.
- The search button “Add All Elements Linked to Elements in Rows” allows finding and adding the elements that are linked to the elements currently displayed within the rows of the matrix.
- Setting links or changing elements will cause dependent elements to become suspicious. Please refer to section 6.4.8 Suspicious Elements view for details.

6.4.7.4 Updating requirement links

If requirements have changed, the links within the Link Matrix view will be declared suspicious with an exclamation mark (see figure 6.112).

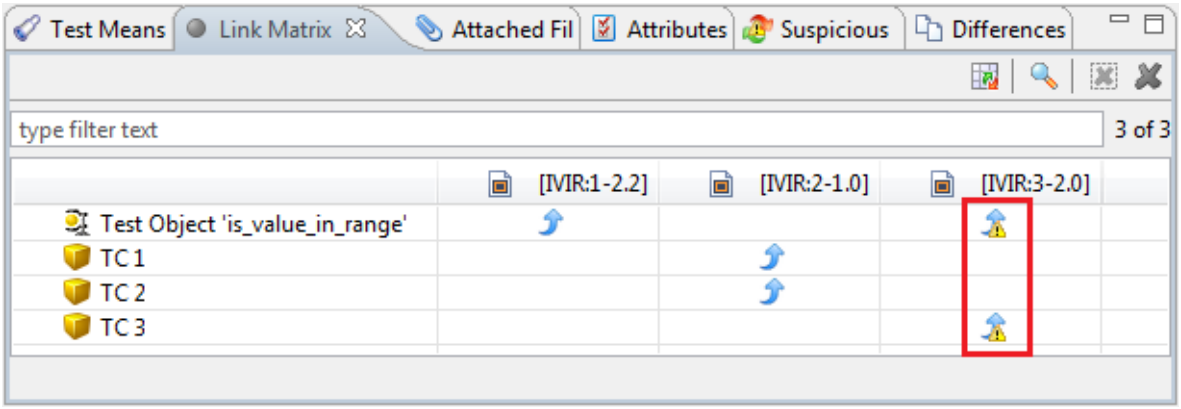


Figure 6.112: Link Matrix view with suspicious elements

General handling of links:

→ A double click on a link within the matrix will delete the link and another double click will create the link again.

Updating suspicious links

To update suspicious links:

→ Double-click a link within the matrix. Click “Yes” in the opening popup window if you want to update the selected link.

→ Right-click a cell and select “Update Selected Suspicious Link” from the context menu. The selected link will be updated.

→ Right-click a row and select “Update Suspicious Links” from the context menu. All links within the selected row will be updated.

6.4.8 Suspicious Elements view

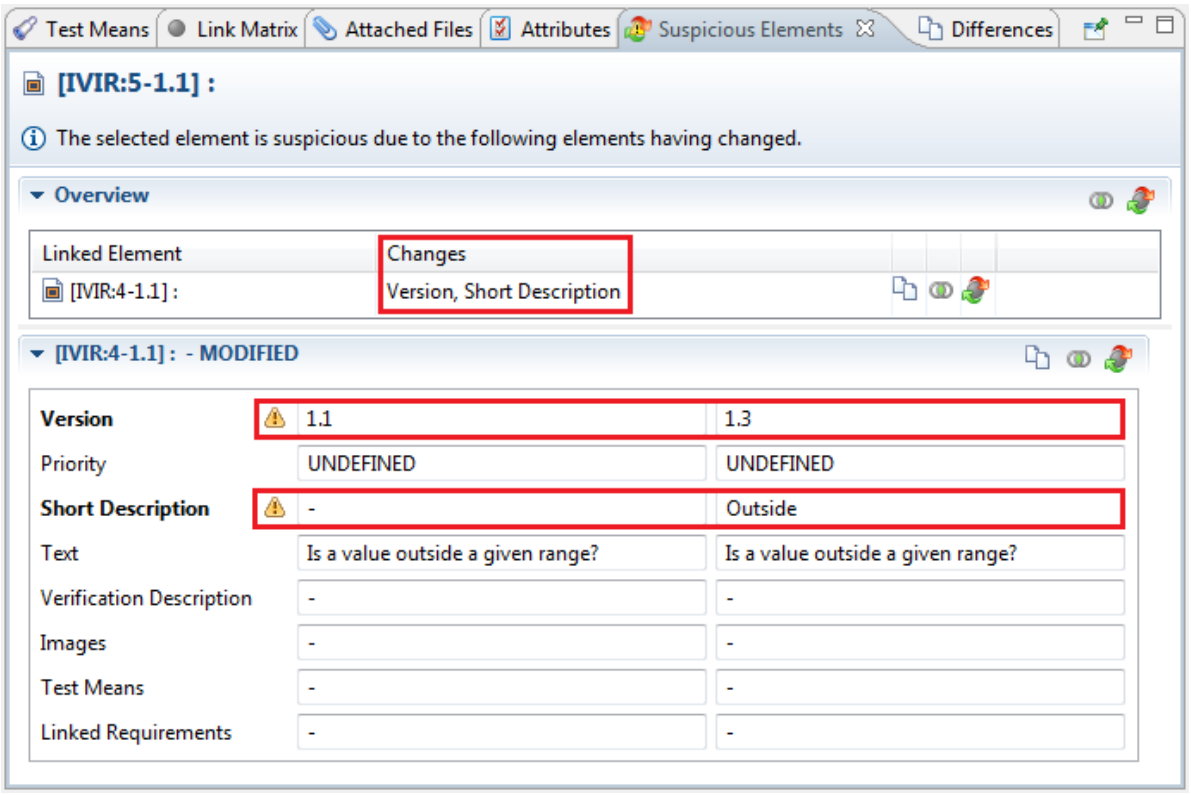


Figure 6.113: Suspicious Elements view

The Suspicious Elements view allows finding out the reason why an element is suspicious. In this case the version number has changed and a short description has been added. *Suspicious Elements view*

During the testing process tasks, modules, test objects and test cases will be linked to requirements indicating that the respective requirements are tested by the linked elements.
Whenever a requirement changes because of modifications or because a new version has been checked in, the linked elements will become suspicious and need to be reviewed. The suspicious status will be indicated by an exclamation mark icon decorator, i.e. for a suspicious test object.

6.4.8.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Sets elements semantic equal (all elements or selected element).
	Updates links (all links or selected link).
	Compares the versions.

Table 6.36: Tool bar icons of the Suspicious Elements view

"Set elements semantic equal" should only be used in situations where the change of a requirement does not change its meaning such as spelling corrections, formatting etc. In all other cases the link should be updated.

6.4.8.2 Determine changes that caused suspicious status

When you have linked the test object and some test cases, any changes to the linked requirements will cause the linked elements to become suspicious. Please switch to the Overview perspective to be able to see that.

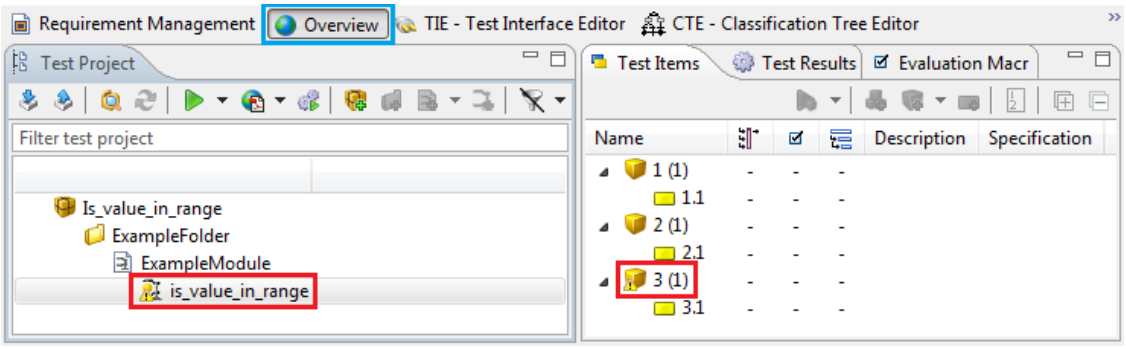


Figure 6.114: Suspicious test object and test cases in the Overview perspective

Determine the related modified requirements that causes the status of a test object being suspicious within the Suspicious Elements view:

→ Select the suspicious test object within the Test Project view. (Again you have to do that within the Overview perspective.)
The Suspicious Elements view will display the changed requirements (see figure 6.115).

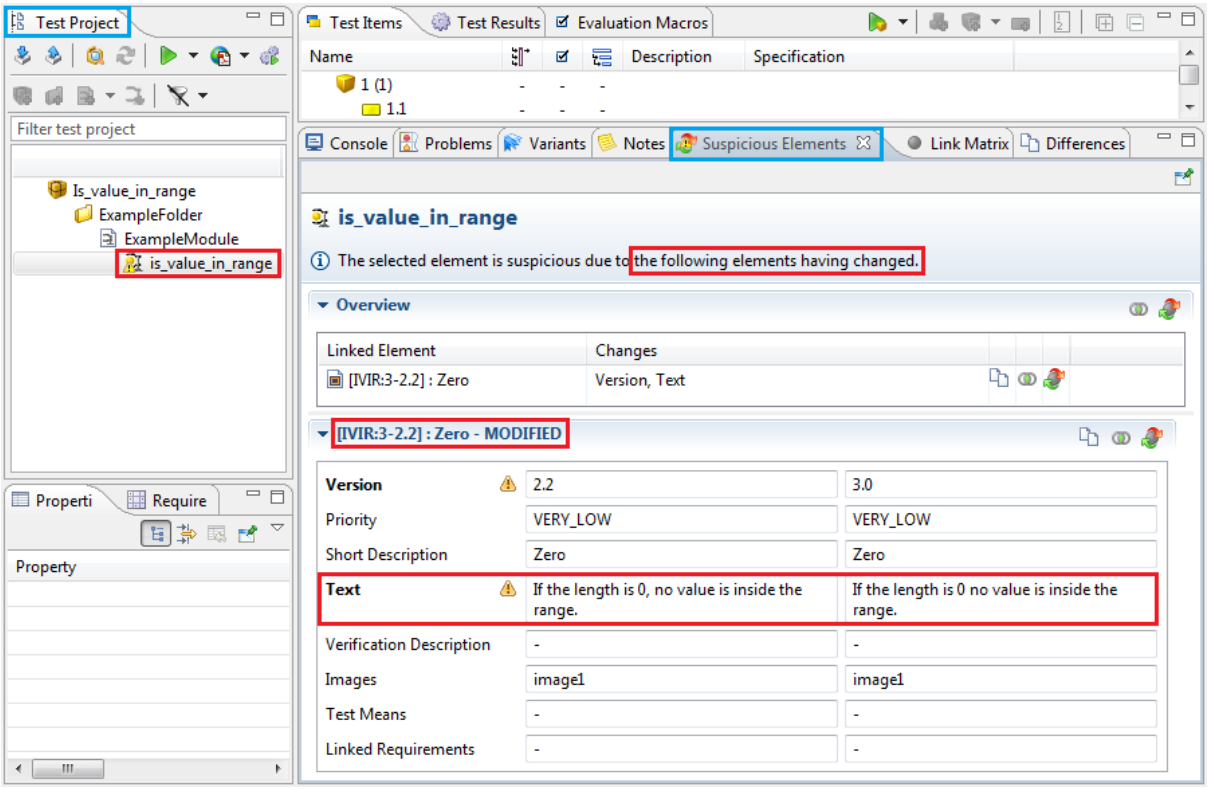


Figure 6.115: Suspicious test object and linked modified requirements

As you can see in figure 6.115 above, the requirement text of the requirement “[IVIR:3-2.2]:Zero” has been edited. Therefore it has the addition “MODIFIED”.

If you select a test case in the Overview perspective, the Suspicious Elements view will also show the changed requirement(s) (see figure 6.116).

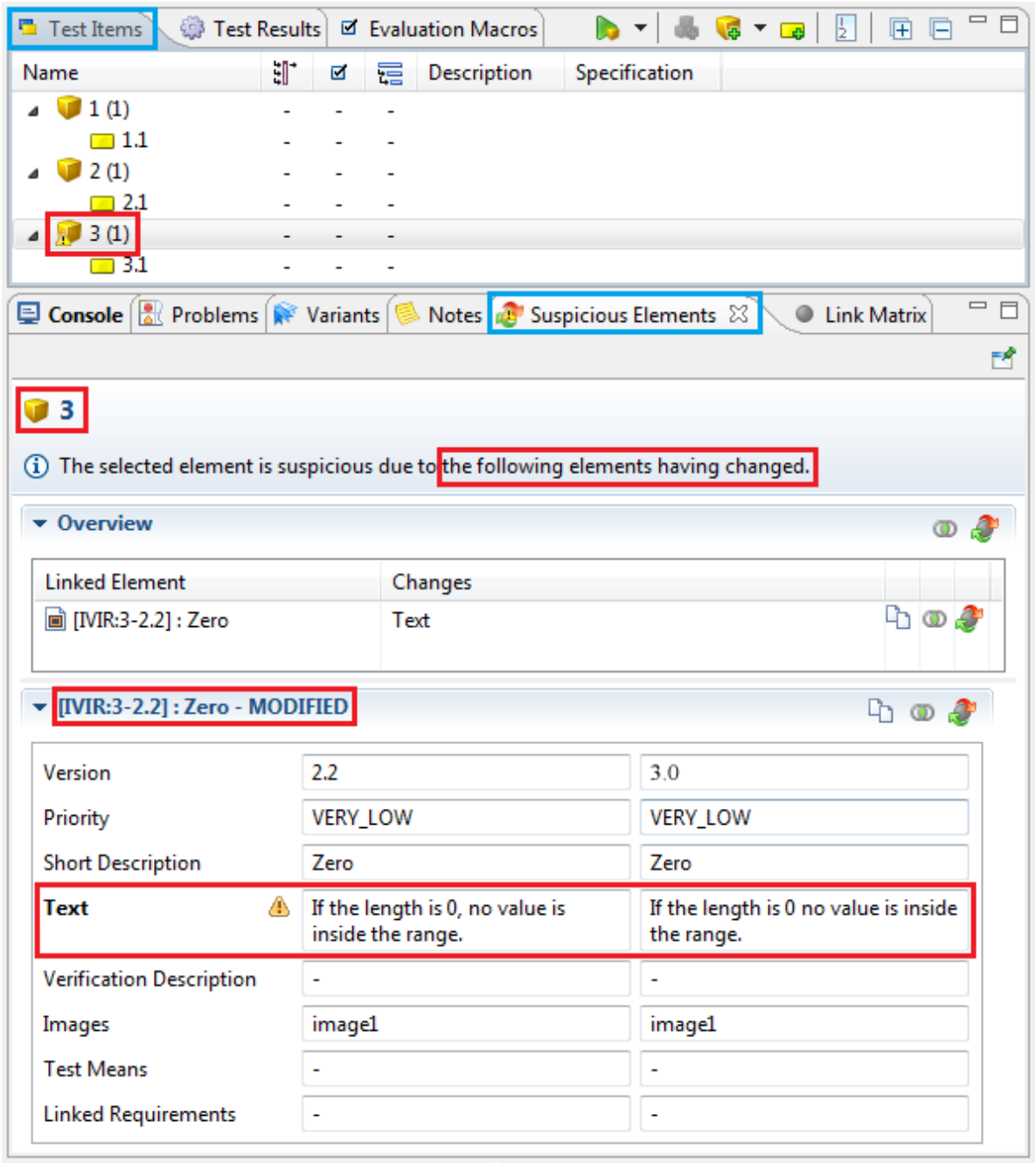


Figure 6.116: Selecting the suspicious test case shows the modified requirement(s)

Within the Differences view you can determine the exact differences:

Differences view

→ Click on (Compare Versions).
The Differences view shows all changes of the requirement (see figure 6.117).

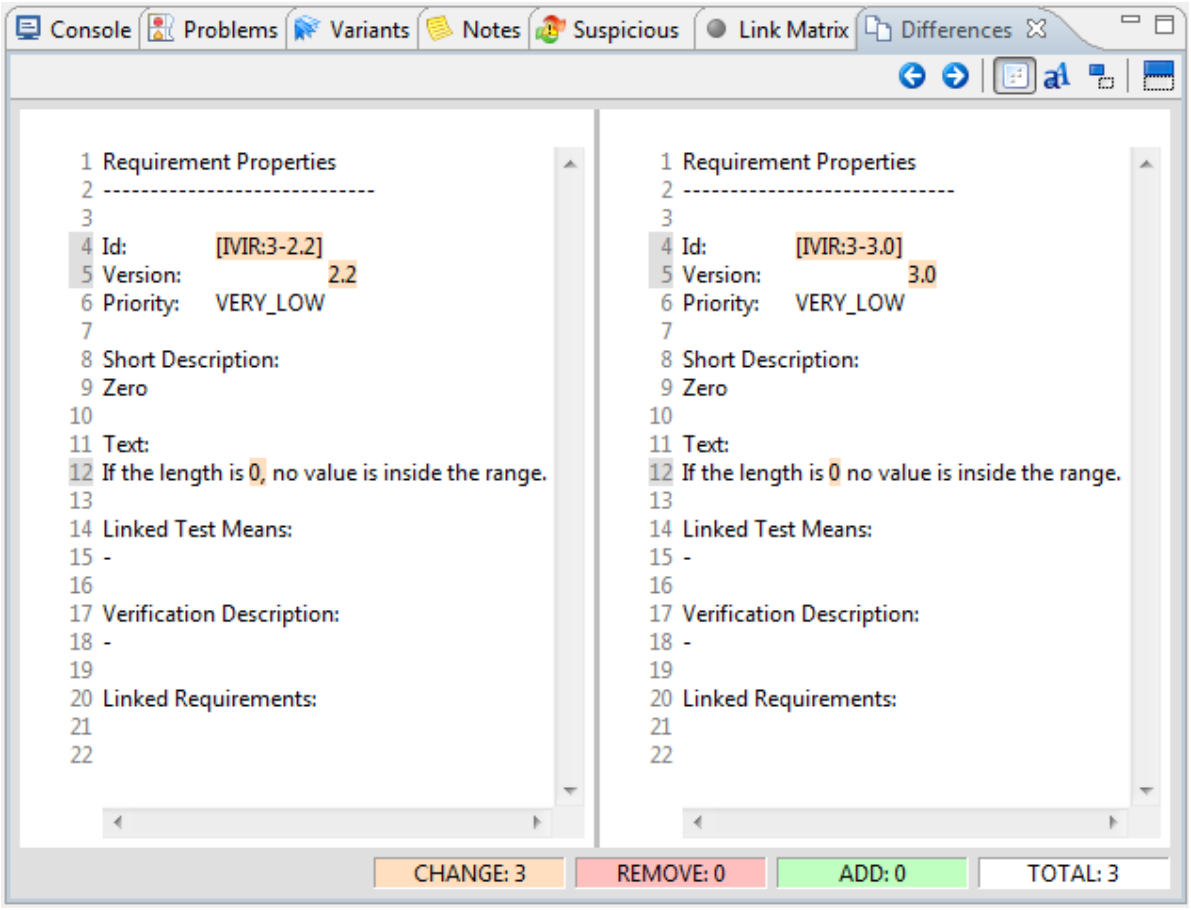


Figure 6.117: Comparing the versions of the requirement

You need to determine if the change of the requirement affects the linked test cases and adapt the test data if necessary.

If no changes to the test cases are required, update the link to acknowledge the requirement change. Therefore click on (Update Link). The suspicious icon will then disappear for the respective test case.

For more information about the Difference view go to section 6.4.12 Differences view / Reviewing changes.

You can also update requirement links in the Link Matrix view.

6.4.9 Attached Files view

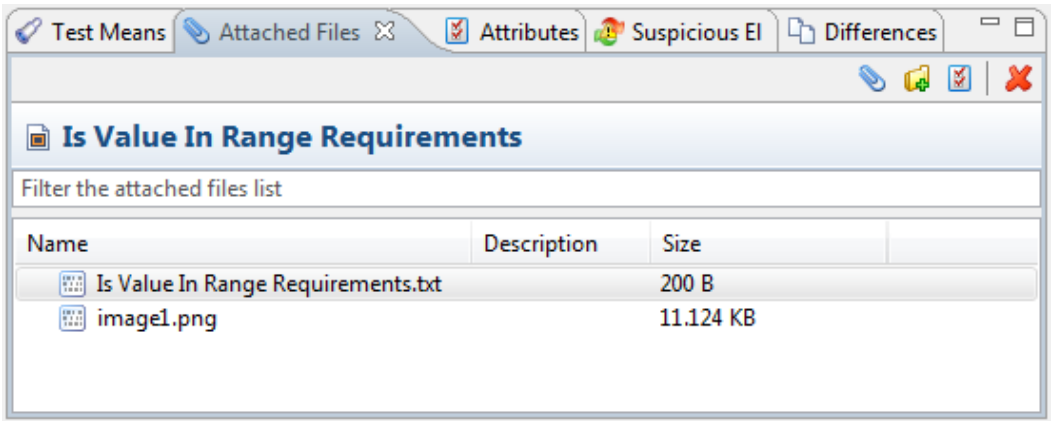


Figure 6.118: Attached Files view

Attached Files view

The Attached Files view allows adding arbitrary files to the selected requirement. You can add additional documents with detailed information about the requirement. The files will be stored within the TESSY database.

6.4.9.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Adds the selected file (Ctrl+N).
	Creates a new folder (Ctrl+B).
	To edit a name or description of the selected folder or file (Alt+C).
	Deletes the selected file (Del).

Table 6.37: Tool bar icons of the Attached Files view

6.4.10 Attributes view

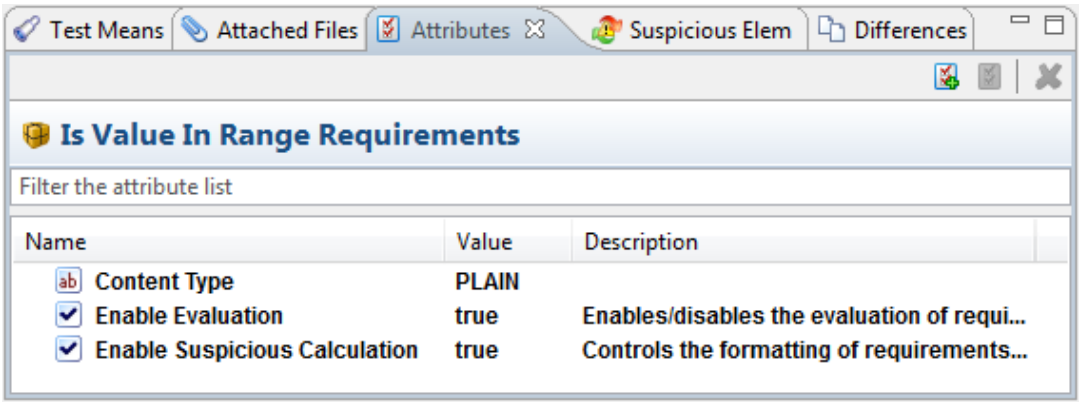


Figure 6.119: Attributes view

Attributes view

The Attributes view allows adding arbitrary attributes for the selected requirement or requirement document.

Important: New attributes should be created for the requirement document. They will then be inherited to each requirement of the document and can be overwritten on requirement level.

There are three predefined attributes named “Content Type”, “Enable Evaluation” and “Enable Suspicious Calculation” on document level that control the behavior of the requirement evaluation and suspicious calculation for elements linked to requirements.

6.4.10.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Creates a new attribute (Ctrl+N).
	Edits the selected attribute (Ctrl+E or double click).
	Deletes the selected attribute (Del).

Table 6.38: Tool bar icons of the Attributes view

6.4.10.2 Editing attributes of a requirement

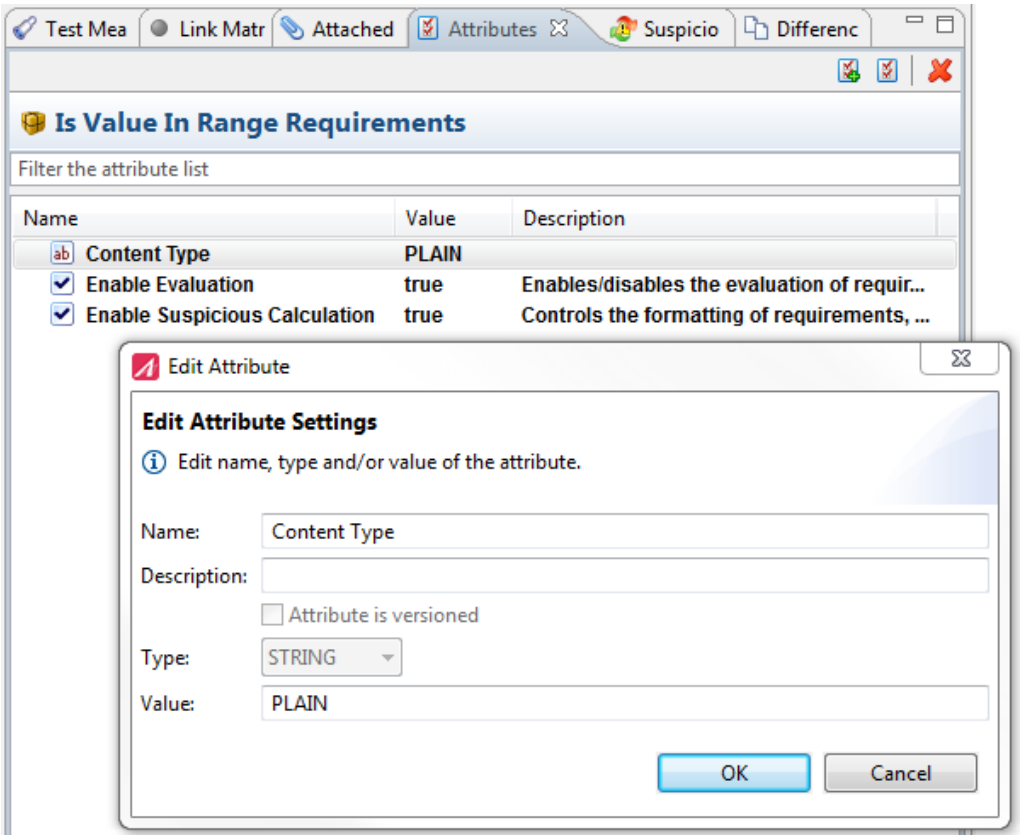


Figure 6.120: Editing the requirement settings within the Attributes view

To edit an attribute:

→ Within the RQMT Explorer view select a requirement, a requirement document, or a requirement chapter.
The Attributes view will display the attributes for the selected element.

→ Right-click the desired attribute and select “Edit” from the pull-down menu.
→ Change value, name or description of the attribute if possible or add an description.

Please note that it is not always possible to edit all of the given opportunities. Usually it is possible to edit the value but name and description can only be edited where the attribute was originally created. Type and version of a requirement can not be edited.

For example you can edit the “Content Type” version of a requirement in the Attribute Settings. This is necessary to enable the HTML Document View and the HTML Editing. The “Content Type” of a document needs to be “HTML” instead of “PLAIN”.

To do so change the “type” to HTML (instead of plain text):

→ Select the desired requirement document within the RQMT Explorer view.
→ Right-click the attribute “Content Type” within the Attributes view and select “Edit”.
→ Change the value to “HTML” and click “OK” (see figure 6.121).

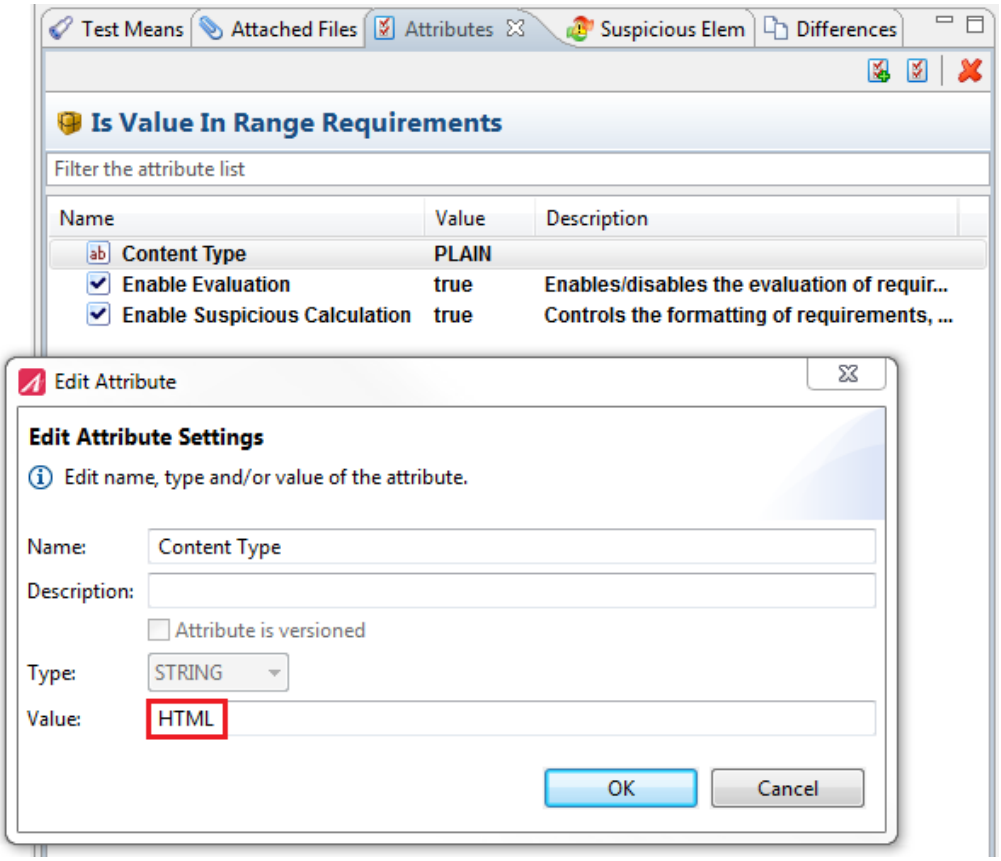
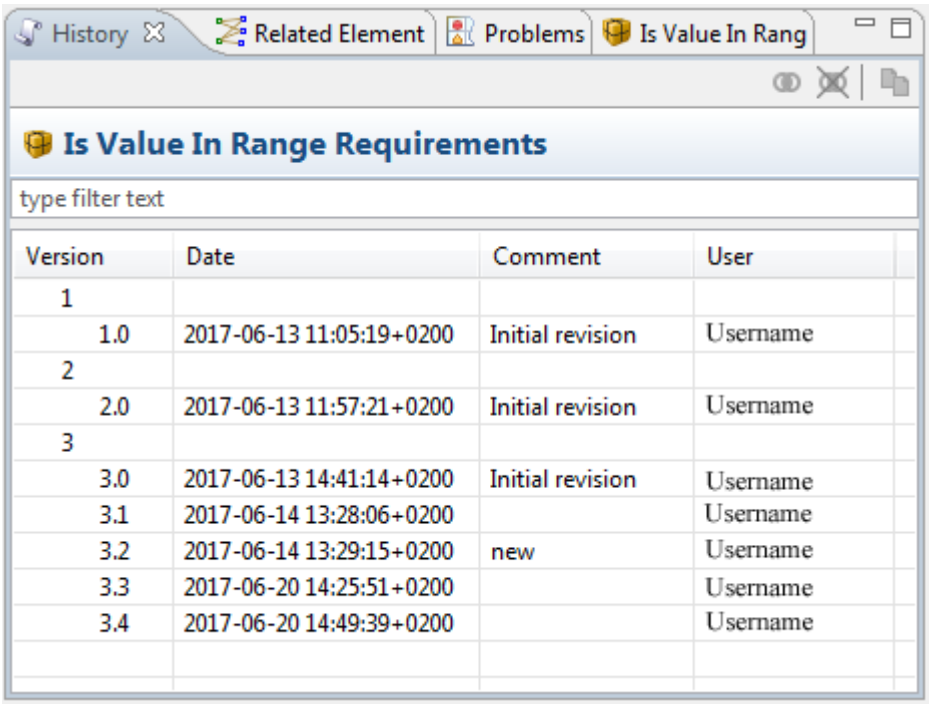


Figure 6.121: Changing the “Content Type” attribute to HTML

For more information about editing requirements in the HTML editor go to section 6.4.15.2 Editing the requirement as HTML version.

6.4.11 History view



Version	Date	Comment	User
1			
1.0	2017-06-13 11:05:19+0200	Initial revision	Username
2			
2.0	2017-06-13 11:57:21+0200	Initial revision	Username
3			
3.0	2017-06-13 14:41:14+0200	Initial revision	Username
3.1	2017-06-14 13:28:06+0200		Username
3.2	2017-06-14 13:29:15+0200	new	Username
3.3	2017-06-20 14:25:51+0200		Username
3.4	2017-06-20 14:49:39+0200		Username

Figure 6.122: History view

6.4.11.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Two versions of a requirement are set to be semantically equal if their contents reflect the same semantics but minor changes were made, e.g. spelling errors.
	Unsets semantic equal. You need to have two versions of a requirement selected to do this operation.
	Compares the versions.

Table 6.39: Tool bar icons of the History view

6.4.12 Differences view / Reviewing changes

Differences view

The Differences view will be displayed within the lower pane, which provides a direct comparison of the respective requirement versions printed as text (see figure 6.123).

Reviewing changes

Each requirement has a version history showing all of its changes.
To review the changes between any two versions of the history or between a historic version and the current version,

- → select either two versions within the history view to compare these versions or select only one version within the view if you want to compare it against the current version.
- → Click on (Compare) in the tool bar.

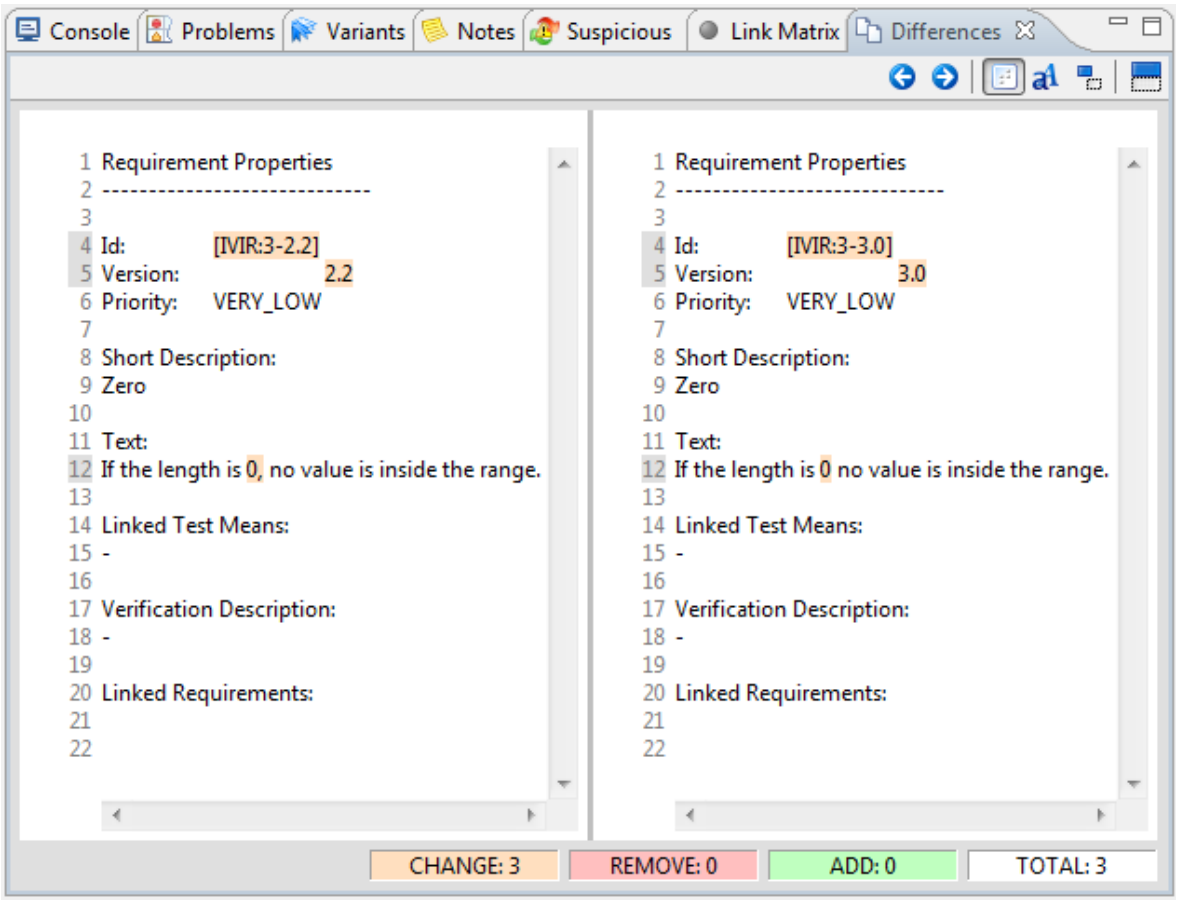


Figure 6.123: Differences view

6.4.12.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Navigates to the left.
	Navigates to the right.
	Switches on or off the matching of words.
	Switches on or off the ignoring of cases.
	Switches on or off the ignoring of whitespace.
	Sets the layout to vertical.
	Sets the layout to horizontal.

Table 6.40: Tool bar icons of the Differences View

6.4.13 Related Elements view

In this view you can see the links of requirements to other requirements, e.g. when creating refined requirements based on a given requirements document.

After selecting a requirement in RQMT Explorer this view presents all linked elements of the respective requirement. It shows all sub requirements or the linked main requirements divided into Incoming Links and Outgoing Links.

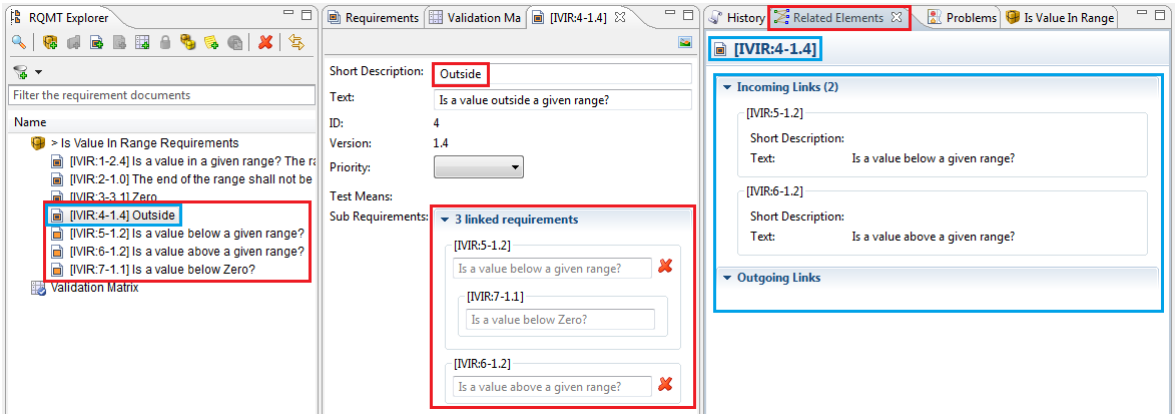


Figure 6.124: Related Elements view and its interrelations

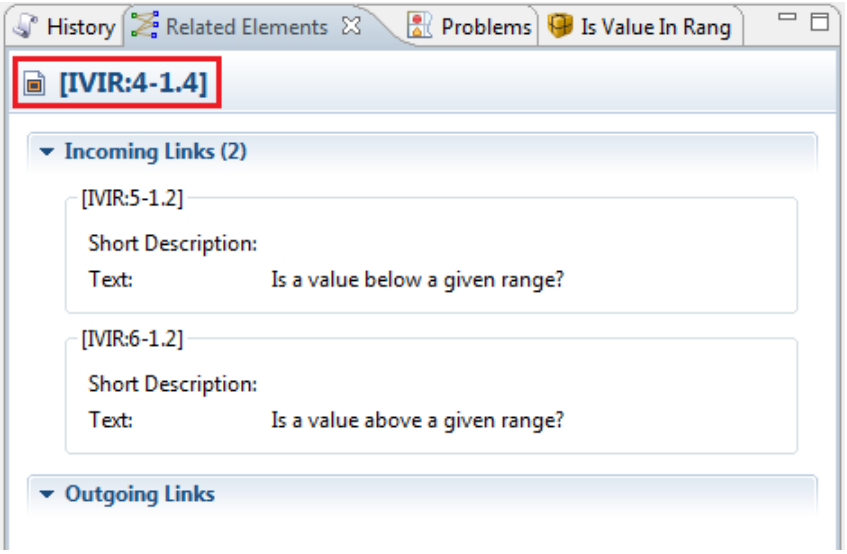


Figure 6.125: Related Elements view

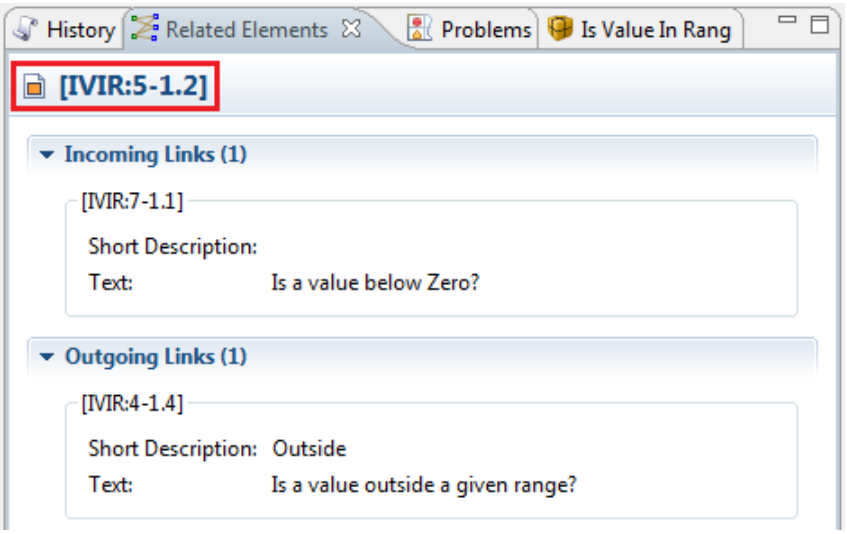
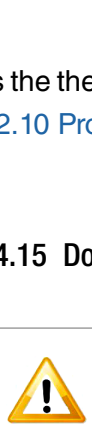


Figure 6.126: Related Elements view with Incoming and Outgoing Links

6.4.14 Problems view

As the the Problems view also appears in the Overview perspective please refer to subsection 6.2.10 Problems view.

6.4.15 Document Preview



Important: You have to open the Document Preview by double-clicking on the appropriate requirement document!

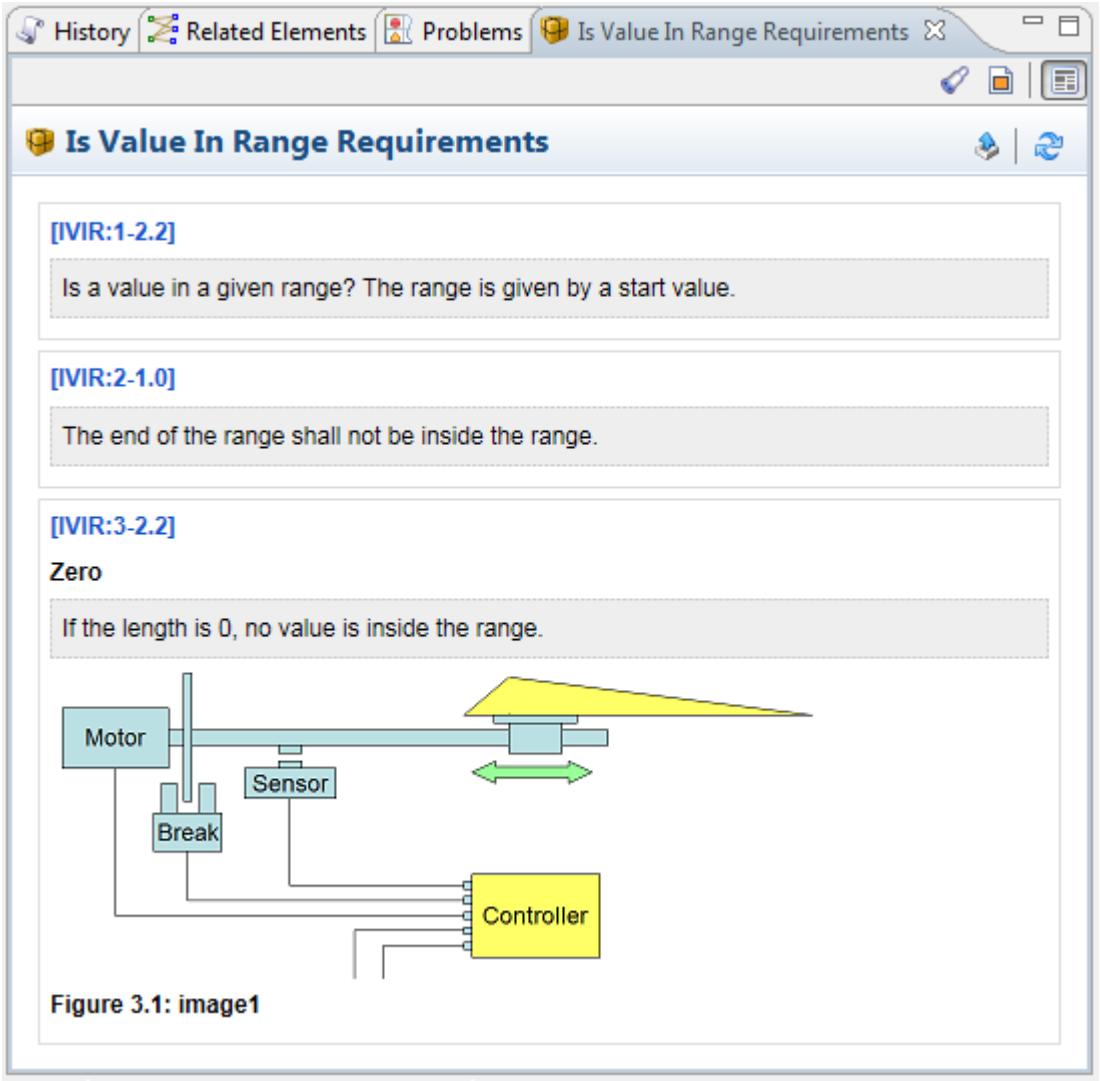


Figure 6.127: View Document Preview

It is possible to open the Document Preview in other perspectives as well. Therefore go to “Window” > “Show View…”, select “Document Preview” from the list and then click “OK”.

A new Document Preview will open where you can select the desired Requirement Document to be displayed (see figure 6.128).

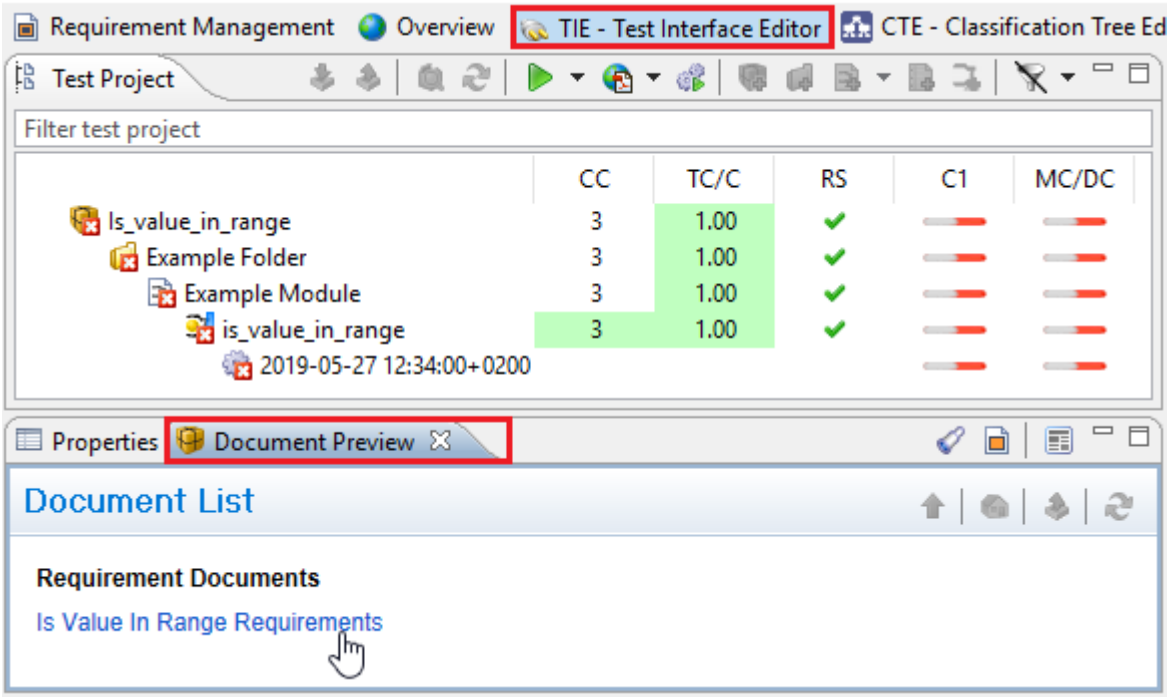


Figure 6.128: Newly opened Document Preview within the TIE perspective

6.4.15.1 Icons of the view tool bars

Icon	Action / Shortcut / Comment
	Displays the test means within the requirement document.
	Displays the sub requirements in the requirement document.
	Toggles to the HTML inline editor (only available of the “Content Type” of the document is “HTML”).
	Back to parent.
	Back home.
	Exports the HTML version as HTML file.

continue next page

Icon	Action / Shortcut / Comment
	Refreshes the view. Necessary after editing any requirement.

Table 6.41: Tool bar icons of the Document Preview

6.4.15.2 Editing the requirement as HTML version

After you created or imported requirements, you can edit them as HTML version:

- → Click on the icon (Toggle HTML Editing).
- → If a note is displayed, that the “Content Type” must be set to HTML, refer to section 6.4.10.2 Editing attributes of a requirement. After changing the content type to HTML, refresh the Document Preview with a click on .
- → Click in a field with any description of a requirement. An HTML inline editor appears.
- → With a click on the icon you can switch between the WYSIWYG editor and plain HTML (see figure 6.129).

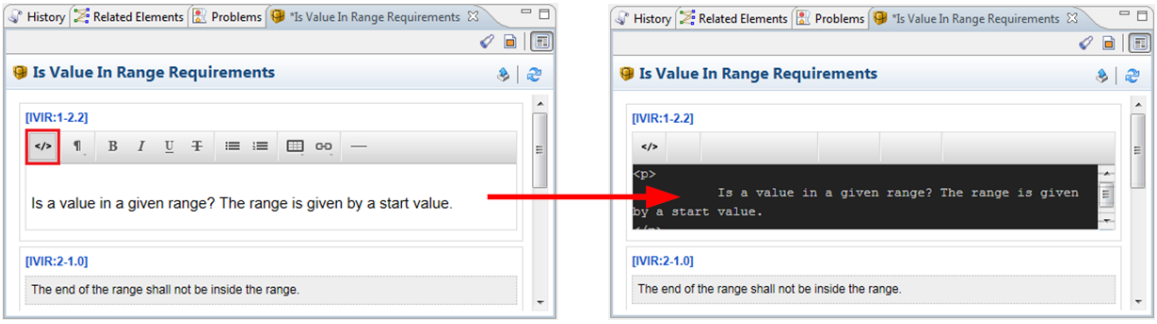
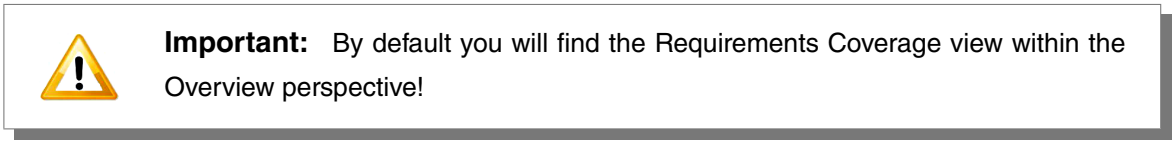


Figure 6.129: HTML editing within the inline editor (WYSIWYG and plain HTML)

6.4.16 Requirements Coverage view



Important: By default you will find the Requirements Coverage view within the Overview perspective!

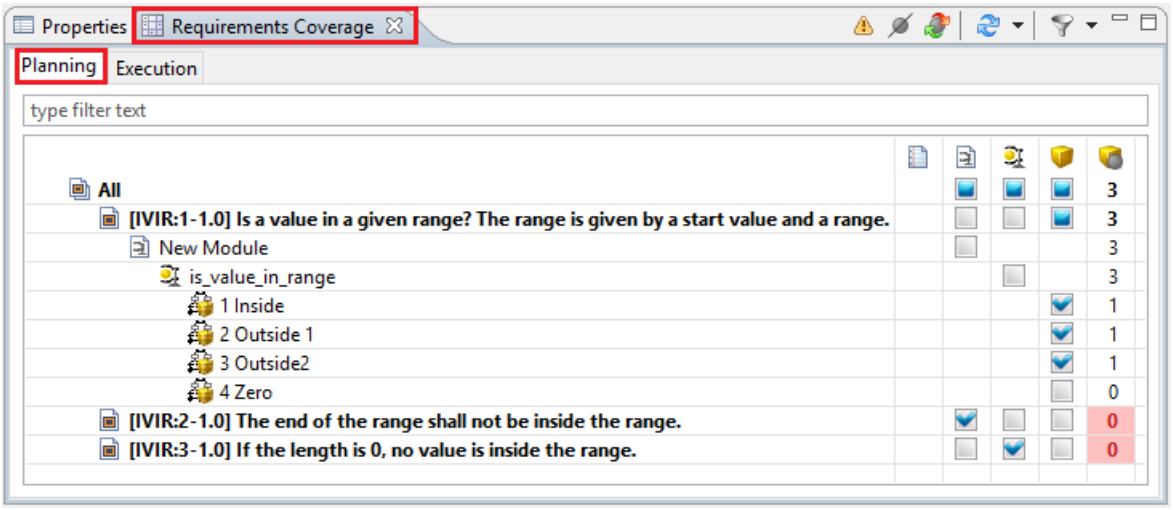


Figure 6.130: Requirements Coverage view with no linked requirements

Requirements Coverage view within the Overview perspective

Within the Requirements Coverage view you will link the test cases with the requirements. You will as well have an overview of the requirements coverage. This is the reason why you will find this view within the Overview perspective.

6.4.16.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Shows only suspicious links in the Planning tab.
	Shows all requirements, including unlinked requirements in the Planning tab.
	Updates all links in the Planning tab.
	Refreshes the view in the Planning and Execution tab. With a click on the little arrow next to the icon you can set on which selection you want to auto refresh. You can also disable the auto refresh function (see figure 6.131).

continue next page

Icon	Action / Shortcut / Comment
	Filters requirements in the Planning tab (component test, unit test, requirements without assigned test means).

Table 6.42: Tool bar icons of the Requirements Coverage view

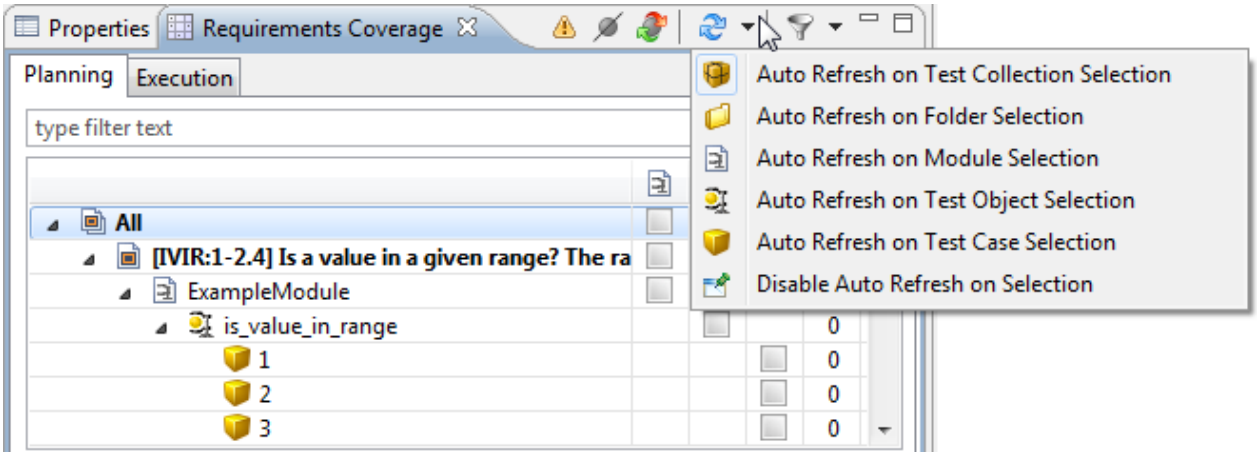


Figure 6.131: Setting or disabling the options of auto refreshing

6.4.16.2 Planning tab

The current status of the links between modules, test objects, test cases and requirements reflects the current state of your requirements coverage. This coverage can be examined on arbitrary levels of your test project.

You can also create a report that shows the currently achieved planning coverage in the test Project view.

Icons of the Planning tab

Icon	Meaning
	Requirements on test modul level and their linking status.
	Requirements on test object level their linking status.
	Requirements on test case level their linking status.

continue next page

Icon	Meaning
	Linked requirements in total.
	Task.

Table 6.43: Tool bar icons of the Planning tab

6.4.16.3 Execution tab

After execution of any tests, the test results are stored within test runs. The test result of a test run covers the requirements that were linked to modules, test objects or test cases at the time the test was executed. Therefore, the actual execution coverage result may differ from the planning coverage result. The execution coverage view is read-only, because this just displays the results. Any changes to requirement links need to be carried out within the planning coverage view.

You can create a report that shows the currently achieved execution coverage.

Icons of the Execution tab

Icon	Meaning
	Test cases with achieved test results for linked requirements.
	Total number of test cases with achieved test results for linked requirements.
	Passed test cases with achieved test results for linked requirements.
	Failed test cases with achieved test results for linked requirements.

Table 6.44: Tool bar icons of the Execution tab

6.4.16.4 Linking requirements with test cases

Linking requirements with test cases

The idea behind linking requirements to modules and test objects is based on the following process:

- First the complete list of requirements is gathered.
- Then each applicable requirement is assigned to modules that implement functionality referenced by the requirement.
- For further break down of the assignment individual test objects are linked to the requirements. This especially makes sense if the module has a large number of linked requirements.
- At last there is a small subset of all available requirements that must be verified. To be taken in consideration the requirement linking for a given test object must be further broken down to test case level.

Important: Please note that only linked requirements of test cases will be analyzed. Unlinked requirements on test case level will not be taken in consideration.

For this process TESSY provides the Requirement Coverage view within the Overview perspective. It is divided into two tabs:

- The Planning tab (see section 6.4.16.2 Planning tab) is the editor for all requirement links to modules, test objects and test cases.
- The Execution tab (see section 6.4.16.3 Execution tab) provides quick overview about the achieved test results for linked requirements.

Important: When selecting objects on upper levels of the test project, the calculation of the test planning/execution links can take a moment.

The content that is displayed in the Planning tab or the Execution tab of the Requirement Coverage view depends on the current selection in the Test Project view of the Overview perspective. If not already linked with any requirement, all available requirements will be displayed; otherwise only the linked requirements will be displayed.

If you want to display the requirements on test cases level, you need to select the respective test case in the Test Items view of the Overview perspective.

You can choose to display all available requirements by clicking on “Always show unlinked requirements” in the Requirement Coverage view. Once chosen, this remains active for other selections as well.

6.5 TEE: Configuring the test environment

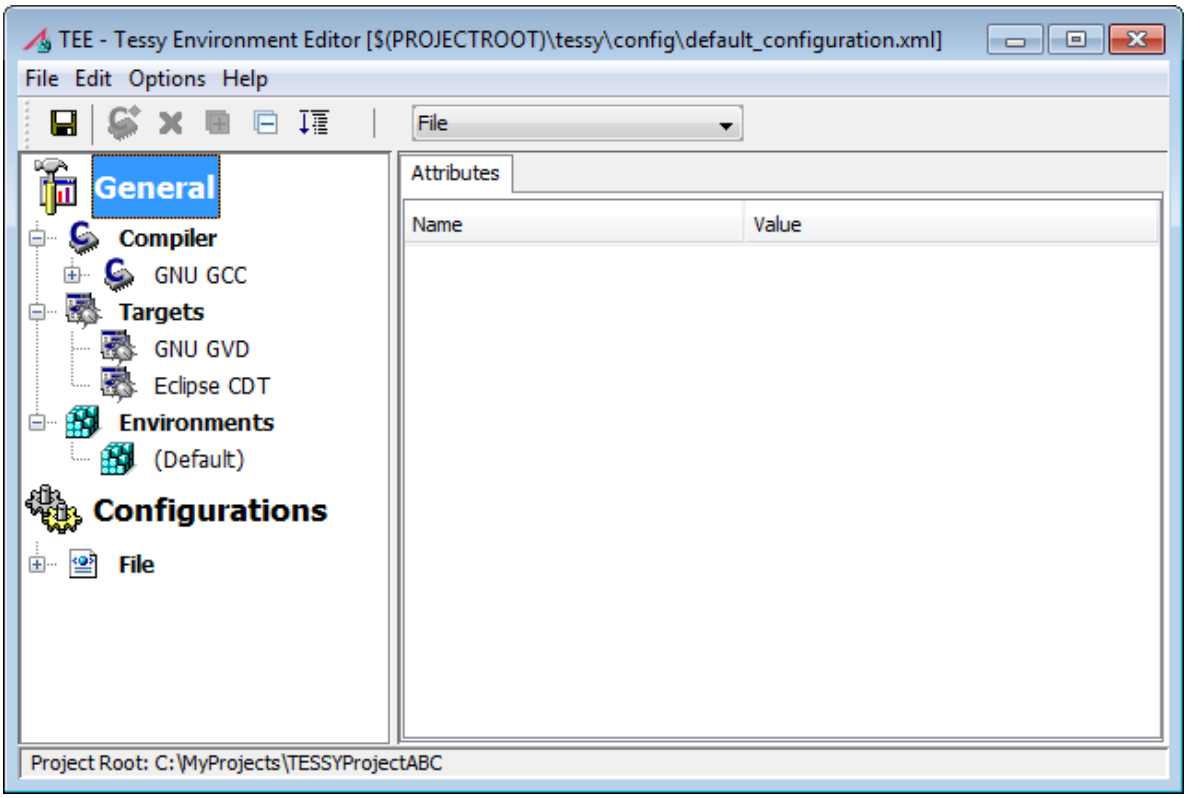


Figure 6.132: Test Environment Editor (TEE)

To execute a test, you need to create and configure a new module. The necessary settings, besides the source files that you want to test, are the following:

- include paths and defines for the source files
- the compiler of a microcontroller target and debugger, i.e. the desired test environment
- compiler and linker options
- debugger settings
- other optional module settings, e.g. ASAP conversion files

We do that within the “Test Environment Editor”, the TEE.

For a complete list of all the available attributes and possible values refer to the application note “Environment Settings (TEE)”.

With the installation of TESSY, the configurations for all supported compiler and target environments (including necessary settings and files) were copied to the TESSY installation directory. You need to enable the compiler and targets that you want to use and add them to your configuration file as described in the following sections.
Their default settings may need to be adapted to your needs, e.g. the installation path of the compiler or target debugger is one of the settings that normally need to be changed to your local values. Settings which have already been used with a previous version of TESSY were also taken over during installation.

The TEE configuration management allows you to create variants of compiler and target settings and assign them to a module. We recommend to save your settings in a specific configuration file, which is the default when creating a new project database (see section 6.5.6 Configuration files). This allows easy sharing of specific environment configurations between developers of the same development team.

As a result you have all your basic settings at one central place, i.e. include paths, additional compiler options, etc. Once configured, you can start testing immediately using the required configuration for all your modules.

6.5.1 Starting the TEE

To open the TEE:

→ In the menu bar click on “File” > “Edit Environment…” (see figure 6.133).
The TEE will start with the custom configuration file assigned to this project database.

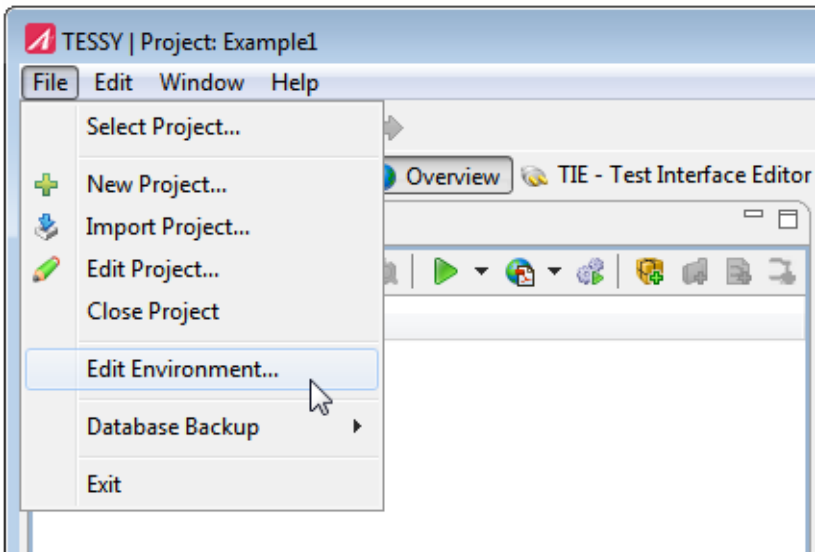


Figure 6.133: Call of the Test Environment Editor (TEE)

6.5.2 Structure of the TEE

Important: TEE is not an eclipse-based program and does not provide views and perspectives! Therefore you cannot drag and drop the panes.

Pane	**Location** (default)	**Function**
General	left	Section General lists all supported compiler and target environments which were configured with default settings.
Configurations	left	Section Configurations lists all supported compiler and target combinations. The settings of these combinations have been inherited by the paragraph General. The section displays a configuration file with default environments.
Attributes	right	Displays all attributes of an item as soon as it is selected in the left pane.

Table 6.45: Structure of TEE

The configuration file

When you have created your project database with the default settings, you will already have a configuration file assigned to the project database. The name of this file will be displayed within the lower right side in the status bar of TESSY. This configuration file will be edited when opening TEE.

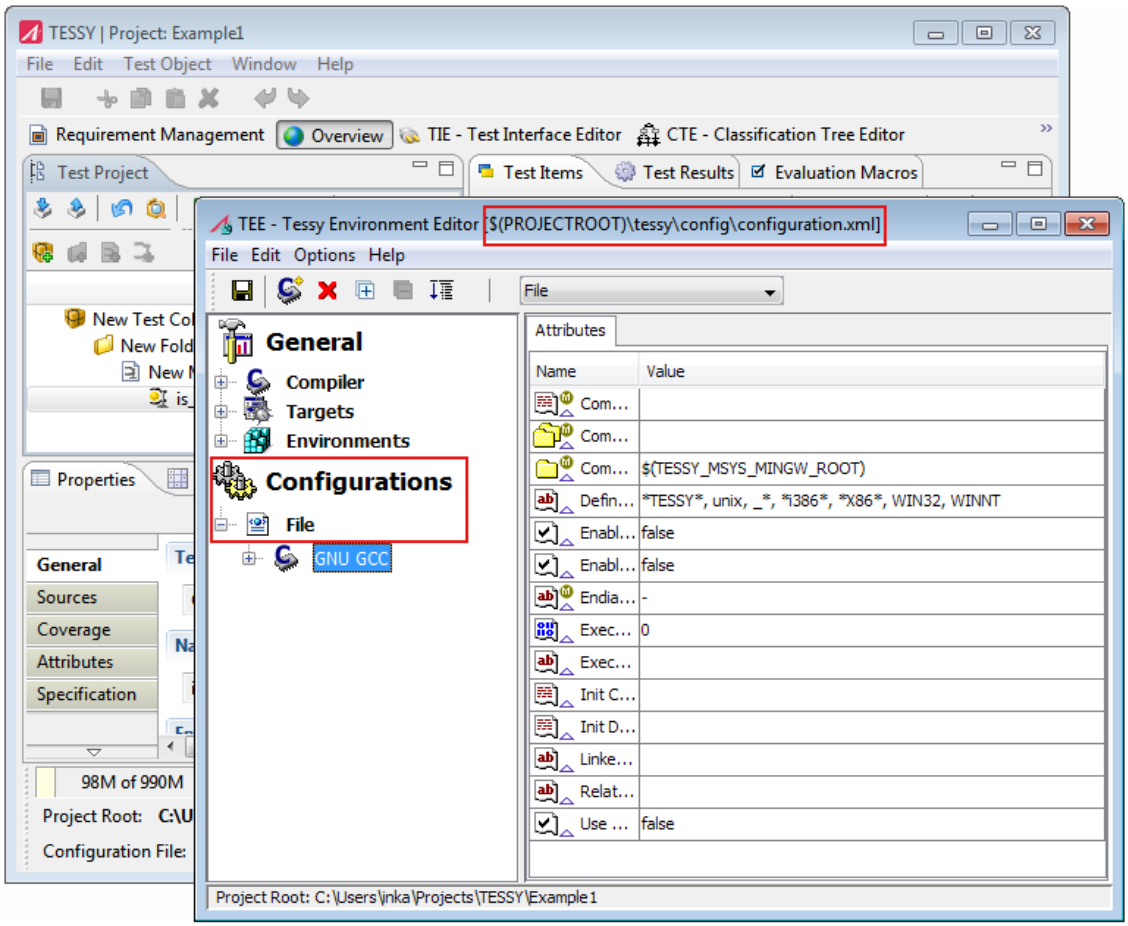


Figure 6.134: Configuration file

6.5.3 Icons of the tool bar

Icon	Action / Shortcut / Comment
	Collapses the selected item.
	Expands the selected item.
	Copies a compiler, including the target and environment sub tree. Icon is only active when a compiler is selected.
	Copies a target, including the environment sub tree. Icon is only active when a target is selected.
	Copies an environment. Icon is only active when an environment is selected.
	Deletes the selected item.

Table 6.46: Tool bar icons of the TEE

6.5.4 Status indicators

The following table shows the indicators of status and their meaning which are used by TEE.

Indicator	Status / Meaning
	Item added as Windows environment variable for all processes, i.e. the make call or the slave call, spawned using this test environment.
	Item added to path variable of the execution environment.
	Item disabled.
	Item disabled because of an inherited parent item being disabled. Example: A compiler item within section “Configurations” gets this item if the respective compiler item within section “General” is disabled.
	Error.

continue next page

Indicator	Status / Meaning
	Error, but the cause is somewhere within the attributes inherited by item. Check for errors upwards or downwards within the configuration hierarchy and fix the error there.
	Information.
	The information comes from another item upwards or downwards within the configuration hierarchy.
	Attribute inherited. The attributes in section “Configurations”, i.e. compiler, targets, environment, are normally inherited from the respective section “General”.
	Compiler active.
	Compiler inactive, probably overwritten or replaced by another item downwards the configuration hierarchy.
	Target active.
	Target inactive, probably overwritten or replaced by another item downwards the configuration hierarchy.
	Environment active.
	Environment inactive, probably overwritten or replaced by another item downwards the configuration hierarchy.
	Warning of error within the attributes of the item. This needs to be resolved somewhere else within the configuration hierarchy.
	Warning, but the cause is somewhere within the attributes inherited by this item. Check for warnings and errors upwards within the configuration hierarchy and fix the errors there.
	The directory is added to the Windows path variable and also added as Windows environment variable. The attribute has been inherited.

Table 6.47: Status indicator example

Different fonts as indicators

TEE will display the attributes in different fonts to indicate the following situations:

Normal letters	Represent factory settings respectively default settings from paragraph “General” and have been inherited.
Bold letters	Value has been defined the first time for the attribute.
Italic letters	Factory setting respectively default setting from paragraph “General” has been overwritten.

Table 6.48: Attribute fonts

6.5.5 Viewing all available configurations

The TEE provides predefined configurations for all supported compiler and target environments. By default, at least the GNU GCC and GNU GCC (C++) environments will be enabled and all your compiler/target environments used in previously installed versions of TESSY, if any, and only basic attributes are visible for the user.

Visualizing all attributes

To visualize all settings, compiler and targets:

→ In the tool bar select the option “Show All” from the pull-down menu (see figure 6.135).

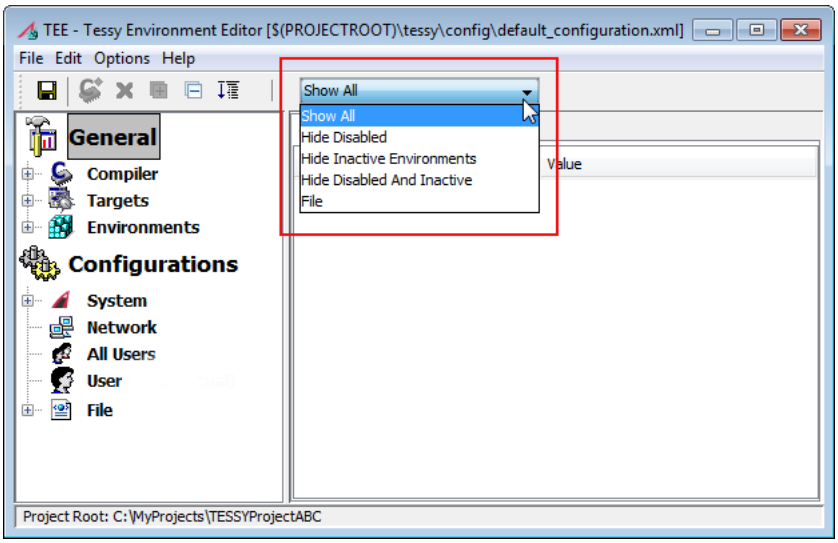


Figure 6.135: Visualizing all items within TEE

Under “Configurations” more sections are now visible, e.g. “System”, which contains the compiler and targets possible to use (see figure 6.136).

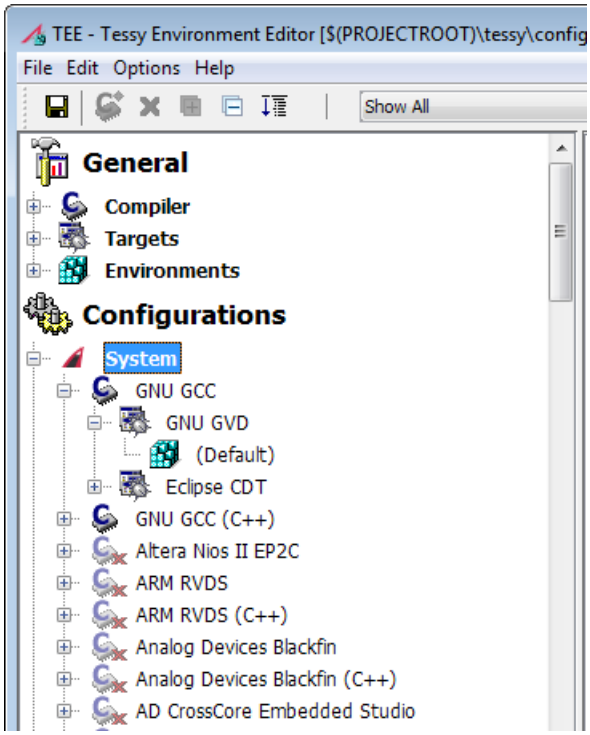


Figure 6.136: Compiler and targets possible to use in TESSY

You can see the enabled GNU GCC and GNU GCC (C++) environments. All other predefined configurations are disabled (see figure 6.137).

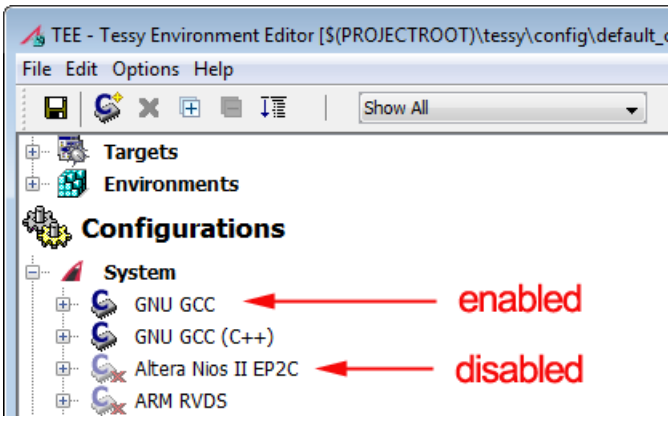


Figure 6.137: Enabled and disabled configurations within TEE

When you select any item on the left pane, the attributes will be shown in the right pane (see figure 6.138).

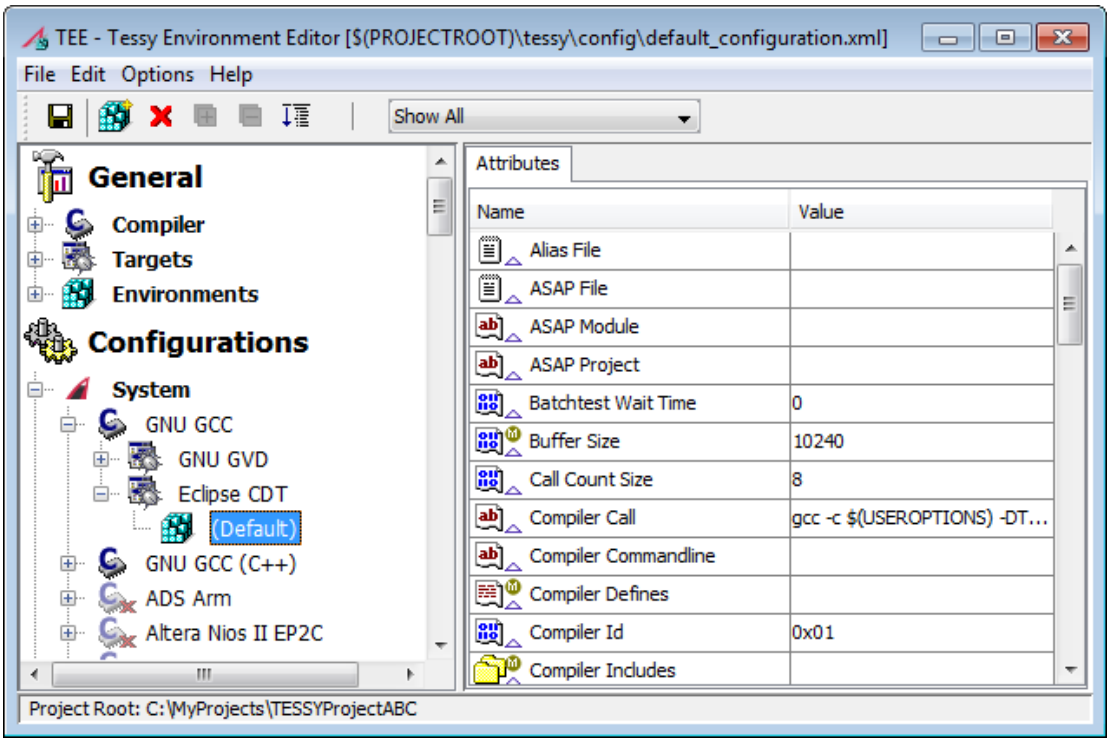


Figure 6.138: Attribute settings of the Eclipse target

You can refresh the panes in TEE by pressing F5.

6.5.6 Configuration files

A system default configuration file contains the settings for all supported compiler and target environments and has been installed with TESSY into the installation directory. The configuration file assigned to the project database contains all settings that are changed compared to the system default configuration. The contents of this file are displayed within section “File”.

Configuration files of the respective sections will be stored in following default folders:

Paragraph	Storage / File(s) / Function
System	stored under `C:\Program Files\Razorcat\TESSY_4.x\config` • `configuration.default.xml`: Contains factory settings of TESSY. This file will not be changed. • `configuration.system.xml`: Contains all your changes made in paragraphs “General” and “Configurations” and will be stored within the user profile. The System section is read only! Change your settings by drag ’n drop compiler and targets onto the file section. Changes are possible within the General section.
File	Stored within the configuration file of the project `[PROJECTROOT]\tessy\config\configuration.xml`

Table 6.49: Contents, functions and storage location of configuration files

6.5.7 Customizing the configurations

The following steps are necessary to customize the configuration file of the currently open project:

- Enable the compiler/target environments that you want to use for testing (see section 6.5.7.1 Enabling configurations).
- Add the desired compiler/target environments to the “FILE” section (see section 6.5.7.2 Adding configurations).

- Review and adjust the attributes (see section 6.5.7.3 Adjusting enabled configurations).

You need administrator privileges to change the contents of the “General” section! If you do not have enough privileges, TEE will inform you and save your changes in a user specific file.

6.5.7.1 Enabling configurations

Enable configurations

To enable a compiler or target,

→ select “Show All” from the pull-down menu to display all supported compiler and target environments.

→ Either in section General or under “Configuration > System” select the compiler or target.

→ Select “Enable Compiler” respectively “Enable target” from the context menu or press Ctrl+E (see figure 6.139).
TEE will remove the red cross from the icon to indicate, that the item is enabled.

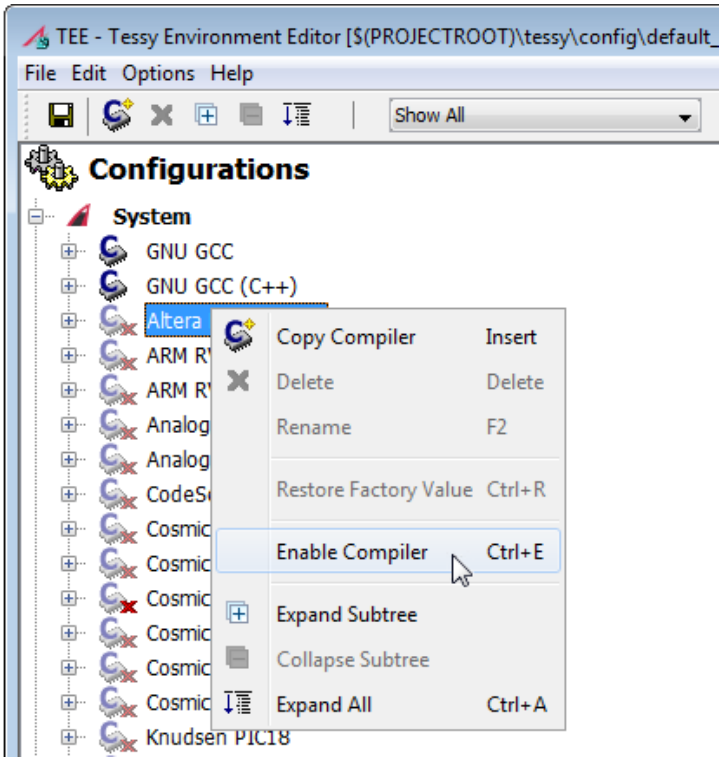


Figure 6.139: Enable compiler

After enabling your desired compiler/targets, we recommend to select “Hide Disabled” from the filter menu to have a better overview.

6.5.7.2 Adding configurations

You make a configuration available as compiler/target environment for usage within TESSY by adding it to the configuration file. To add an environment, copy an available configuration environment from section “System” onto section “File” by either using the context menu or drag and drop:

→ Rightclick the compiler and choose “Add to File Section” from the context menu (see figure 6.140) or “grap” the compiler with the mouse and drop it onto the file section (see figure 6.141).
The compiler will be added to the list under “File”.

→ Change the settings to your needs, e.g. delete some debuggers if they are not necessary for your test environment.

Important: Do all necessary settings within section “File”, because all contents of this section will be saved into your configuration file. Only changed values of attributes will be saved, not the inherited values from other sections!

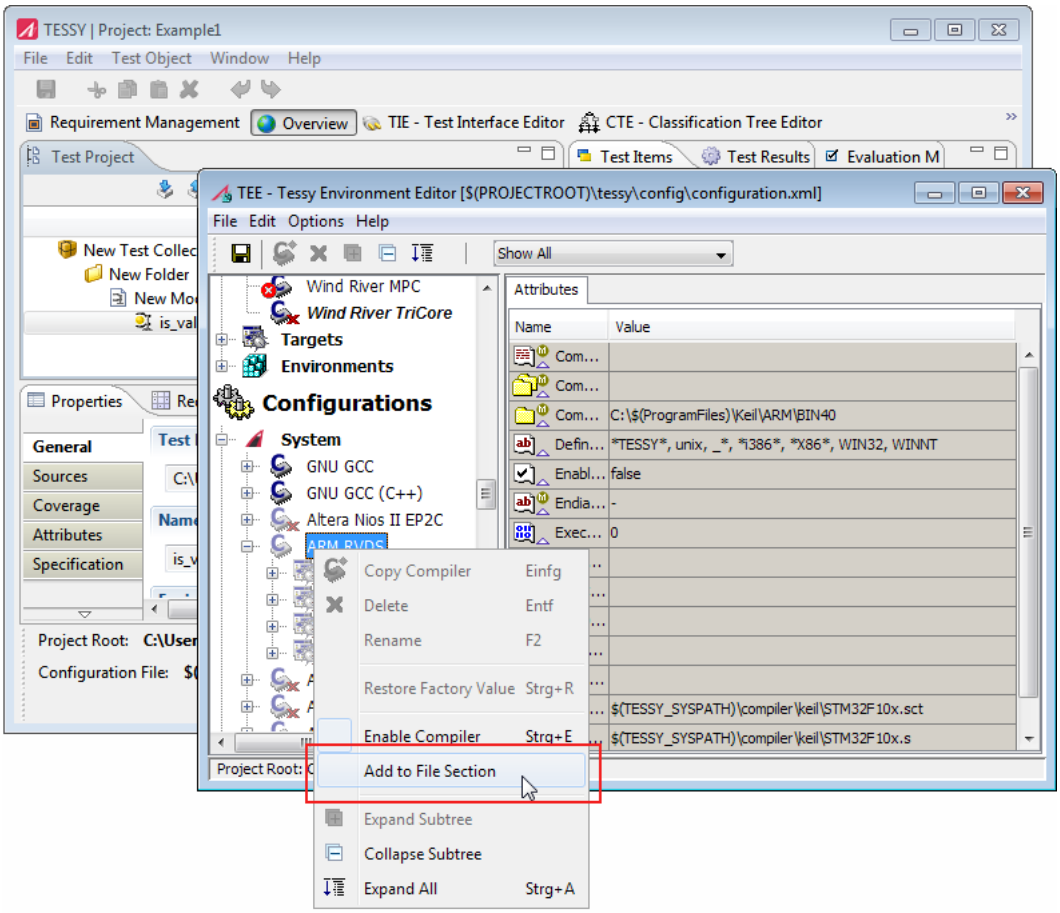


Figure 6.140: Adding an environment to a configuration file by using the context menu

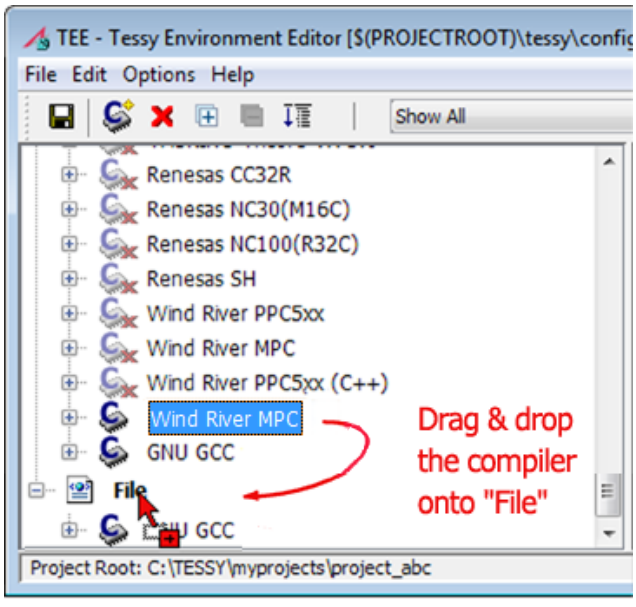


Figure 6.141: Adding an environment to a configuration file by drag and drop

6.5.7.3 Adjusting enabled configurations

Normally you need to change some settings for your specific environment. Some of the settings will be checked for validity. The TEE will check all enabled configurations and displays error and warning signs as soon as an error has been found, e.g. if the “Compiler Install Path” must be corrected.

Important: Please note that all customizations need to be be done within section “File” in the lower part (“Configurations”) of the TEE to have them saved into the custom configuration file.The only exceptions would be the compiler and target installation paths: Those path settings should be carried out within the sections “Compiler” and “Targets” in the upper part (“General”) of the TEE. Those sections are saved either into the installation directory or into a configuration file within the user profile.

If you do as explained above, computer specific path settings are kept out of the configuration file which you will probably share with other testers on different computers. On the other hand your customizations made in the “Configurations” part of the TEE are saved in the configuration file. So this part of your customizations will automatically be available to other testers as well.

Important: Be aware that customized files in the upper part (“General”) of the TEE, e.g. the “Makefile Template” and the “Linker File” amongst others that sometimes need to be customized, have to be copied into the TESSY project. The best way is to copy customized files from the “General” part of the TEE into the “config” folder of the project root. After that is done the respective attributes need be updated in the TEE.

Adding and editing attributes

TEE preserves all default settings. You can revert the default values with “Restore Factory Value” in the context menu (right-click the attribute).

Editing attributes

To edit an attribute,

→ right-click an attribute in the Attributes pane.
→ Select “Edit Attribute” from the context menu.
→ If you want to change only an attribute value, select “Edit Attribute Value” from the context menu.
Depending on the attribute type, either a standard selection dialog for that kind of information will appear (e.g. Browse for Folder in case of directories) or the inline editor for the value will be activated.
If you have changed a default value other than the factory setting, the attribute will be displayed in italics.
→ If you want to delete the attribute value, select “Reset Attribute” from the context menu or press Del.
This will either remove the local value and show the inherited value or delete the whole attribute entry, if it is only defined locally in this section.

Adding attributes

To add an attribute,

→ right-click in the Attributes pane.
→ Select “New Attribute” from the context menu.
The New Property dialog will open.
You see different attribute types available: String, Boolean, Number, Float, File and Directory.

→ Specify the desired type.

Important: The type cannot be changed once the attribute is created!

→ Check the desired specific attribute flags. This depends on the type used. For description see table below.
→ Click “OK”.

Flag	Description
Inheritable	This flag will always be ticked by default. It controls the inheritance of the attribute: The attribute will be available in all (child) section nodes. Some basic attributes are defined at the main nodes, e.g. compiler. Each supported compiler will inherit these basic attributes.

continue next page

Flag	Description
Validate	This flag may be important for directory or file types. The attribute value will be validated by TEE, e.g. whether the path or file is available. An error sign will indicate that the file or directory could not be found.
Read Only	This flag makes it impossible to change a default value by using the attribute pane of the module properties dialog.
Always Valid	Internal flag of TEE
As List	Using this flag, the attribute value will be handled as list of values (comma separated). The values may be edited using a special list dialog. This is useful for file or directory types.
Multiline	Provides a text window for multiline editing.
Internal	Internal flag of TEE.
Hex Format	This flag is useful in combination with the number type. TEE will convert all inputs (e.g. a decimal value) to a hex value, e.g. 1 > 0x01.
Visible	This flag makes the attribute visible in the attribute pane of the module properties dialog (and within the test report).
Not Empty	Checks whether the value is not empty. An error sign will indicate that the attribute does not have a value.
System	Internal flag of TEE.
Environment Variable	This flag is useful during test execution and during the make process: TESSY will create an environment variable within the process space of the process that will be used for test execution (e.g. running the slave process) and for make (e.g. building the test driver). **Important:** The environment variable will only be recognized by TESSY if a plus-sign is used in front of the Make Call value.

continue next page

Flag	Description
Add to PATH Variable	This flag is useful for attributes of type directory. Like described above for the flag “Environment Variable”, the respective directory value will be added to the PATH variable of the process space used for test execution and make.
Makefile Variable	Adds this variable to the generated makefile for compilation/linking of the test driver application. You can use this variable within the makefile for include paths or other settings required during the make process. A variable named “My Include Path” will be added to the generated makefile as MY_INCLUDE_PATH with the respective value.

Table 6.50: Meanings of flags in the attribute properties

6.5.7.4 Assigning an existing configuration file to your project

Assigning a configuration file

If you want to assign an existing customized configuration file to your project, do as follows:

→ In the menu bar of TESSY select “File” > “Edit Project…” .
The Project Configuration dialog opens.
→ Open the “Advanced Settings”.
→ Under “Configuration File” click on “…”, choose your file and click “Open”.
Click “OK”.

6.6 THAI: TESSY Hardware Adapter Interface

In order to enable hardware I/O stimulation and measurement during unit testing, TESSY provides a hardware adapter interface allowing control of external measurement hardware. This hardware device implements a configuration interface as well as reading and writing methods for hardware signal data. The following figure shows the architecture of the system within the TESSY unit testing framework using a GAMMA V based HIL system as an example implementation.

Hardware stimulation and measurement

For further information see http://www.rst-automation.com/index.php/middleware.

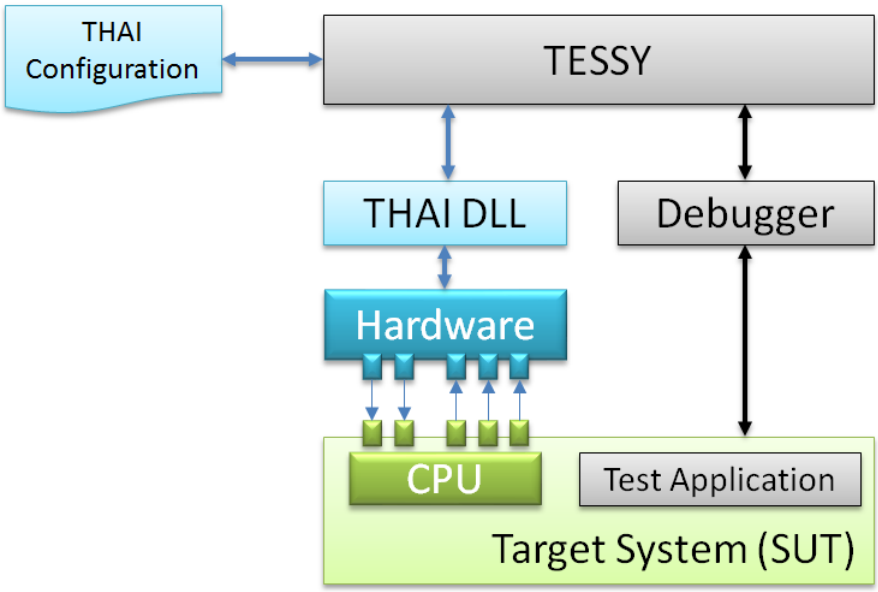


Figure 6.142: Integration of a hardware adapter (e.g. GAMMA) into the TESSY unit test execution

During module analysis: TESSY reads the configuration of the hardware device in order to determine the available interface (i.e. the available I/O signals).This list of signals (including passing directions) will appear within the interface of the TESSY module (for each test object). The signals may be edited within the Test Data Editor (TDE) as any other normal test object variable.

During test execution: The input signals will be stimulated from the input values provided within the TDE and the actual measurements will be saved to the respective output signals. The synchronization between stimulation/measurement and the unit test execution will be controlled by the debugger running the test object code. Therefore callback functions that stimulate/measure the signal values will be executed before and after calling the test object.

Timing measurement may also be supported by the hardware adapter device, this will be carried out using dedicated pins of the hardware.

6.6.1 The THAI Configuration file

The configuration file for the hardware adapter includes all necessary configuration data for TESSY as well as the configuration data necessary for the hardware device (in XML format). The configuration file will be specified as ”THAI Configuration File“ attribute within the Test Environment Editor (TEE).

TESSY extracts the available hardware signals from the following XML data structure. The required tags and attributes mandatory for TESSY are in bold.

```
<project name="sample project" description="THAI test project">

  ...more hardware-related tag entries possible

  <signals>
    <signal name="sw0" passing="IN" type="uint8" />
    <signal name="sw1" passing="IN" type="uint8" />
    <signal name="leds" passing="OUT" type="uint16" />
  </signals>
</project>
```

Figure 6.143: XML data structure for the configuration THAI

There may be additional tags and attributes for the hardware device configuration, which will be skipped by TESSY. Only the ”signals“ and the corresponding ”signal“ tags will be read and TESSY will create the available unit test interface from these entries.

For more details about THAI including a sequenced diagram for hardware device control and information about the interface DLL please refer to the application note “TESSY Hardware Adapter Interface” in TESSY (“Help” > “Documentation”)

6.6.2 Environment Editor (TEE) Settings for THAI functionality

The THAI functionality can be enabled using the “Enable THAI” TEE attribute. It is recommended to enable THAI for individual modules instead of enabling it for all modules of a whole project.

The configuration of the THAI related attribute can be done within TEE but the THAI functionality should be enabled for individual modules using the module properties as shown below.

Fill the required THAI attributes:

→ Create a new module within TESSY and go to the Properties view. The ”Enable THAI“ toggle button will be available within the Features section. *Enable THAI*

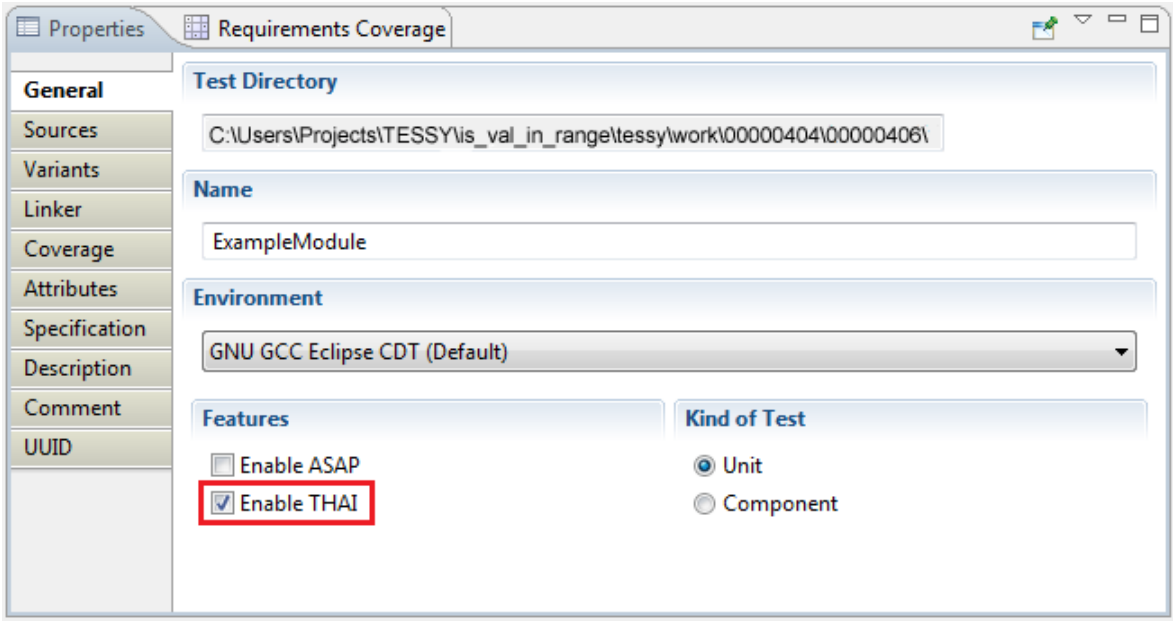


Figure 6.144: Enable THAI in the Properties view

- If you select the ”Enable THAI“ check box, TESSY will add the required THAI attributes within the attributes tab.
- If you switch to ”Attributes“ within the Properties view, you will see the required THAI attributes.

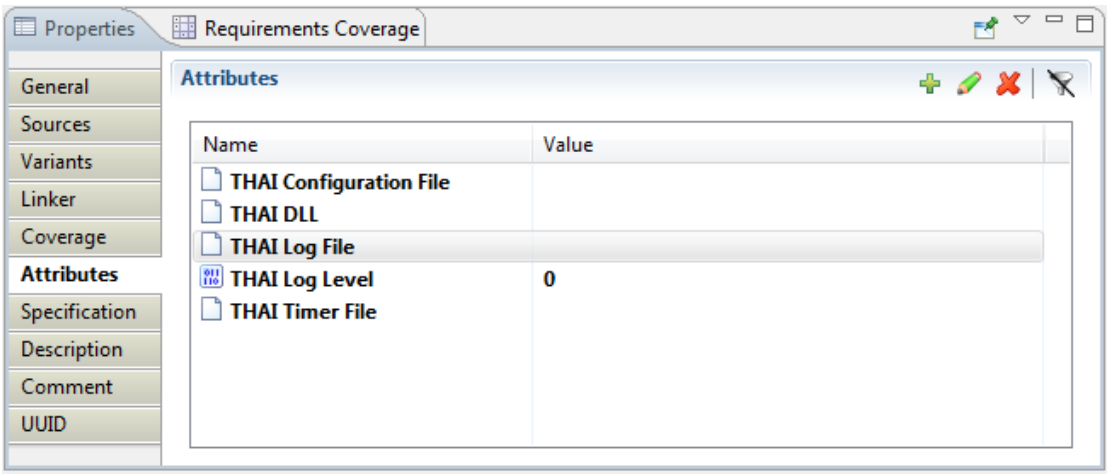


Figure 6.145: Required THAI attributes in the Attribute View

More Information about the configuration and functionality of THAI:

- You need to enter the THAI Configuration File attribute and select a suitable configuration file for your hardware device.
- The THAI Timer File attribute is filled with a default value. You will need to change this attribute, if you are using the timing measurement feature.
- The THAI DLL attribute references your implementation DLL of the THAI interface.
- The Log Level attributes are optional.

Attribute	**Description**
THAI Configuration File	This references the configuration file for your hardware device including the available hardware interface description for TESSY.
THAI DLL	References your implementation DLL of the THAI functionality.
THAI Log File	The output file for logging.
THAI Log Level	Value from 0 (no logging) to 2 (full logging). The actual log content depends on the implementation of the DLL.

continue next page

Attribute	Description
THAI Timer File	References a C source file with an implementation of the ts_start_timer() and ts_stop_timer() functions. The default implementation contains empty functions. These functions shall toggle a specific hardware pin for timing measurements.

Table 6.51: THAI attributes and their descriptions

6.6.3 Signals within the interface

When THAI is enabled for a specific module, you will see the available hardware signals as inputs and/or outputs according to the contents of the THAI configuration file.

Available hardware signals within the interface

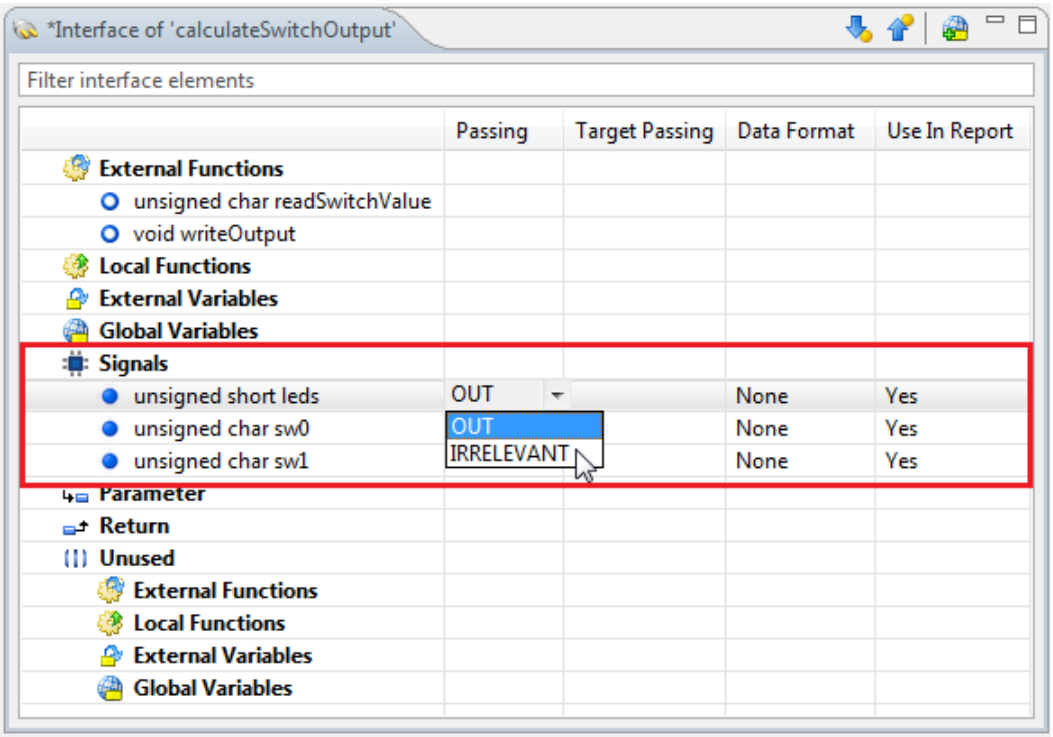


Figure 6.146: Required THAI attributes in the TIE

You can change the passing directions to IRRELEVANT in the Test Interface Editor (TIE) if certain signals are not necessary for the given module. (More information about the TIE is provided in chapter 6.7 TIE: Preparing the test interface.)

6.6.4 Entering test data for signals

Assign concrete values for each test step

The hardware signals defined within the THAI configuration file will appear within the Test Data Editor (TDE) as normal inputs and/or outputs. You need to assign values for each test step as with normal variables of the test object interface. (More information about the TDE is provided in chapter 6.9 TDE: Entering test data.)

> **Important:** You must not use special values like *none* for the hardware signals defined within the THAI configuration file in the TDE. For each signal input and output a concrete value is required.

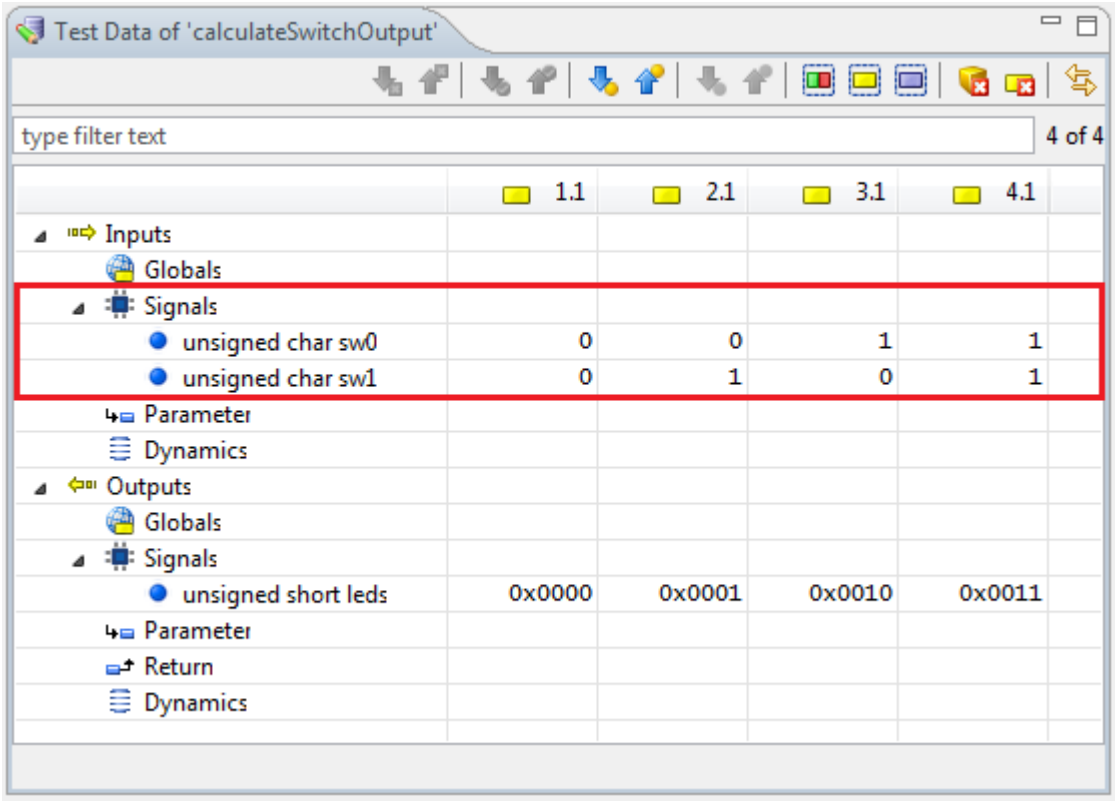


Figure 6.147: Required THAI attributes in the TDE

When evaluating hardware signals it is recommended to specify ranges or values with deviations (e.g. 20 +/- 1%) due to the possible signal measurement deviations cause by the hardware.

6.7 TIE: Preparing the test interface

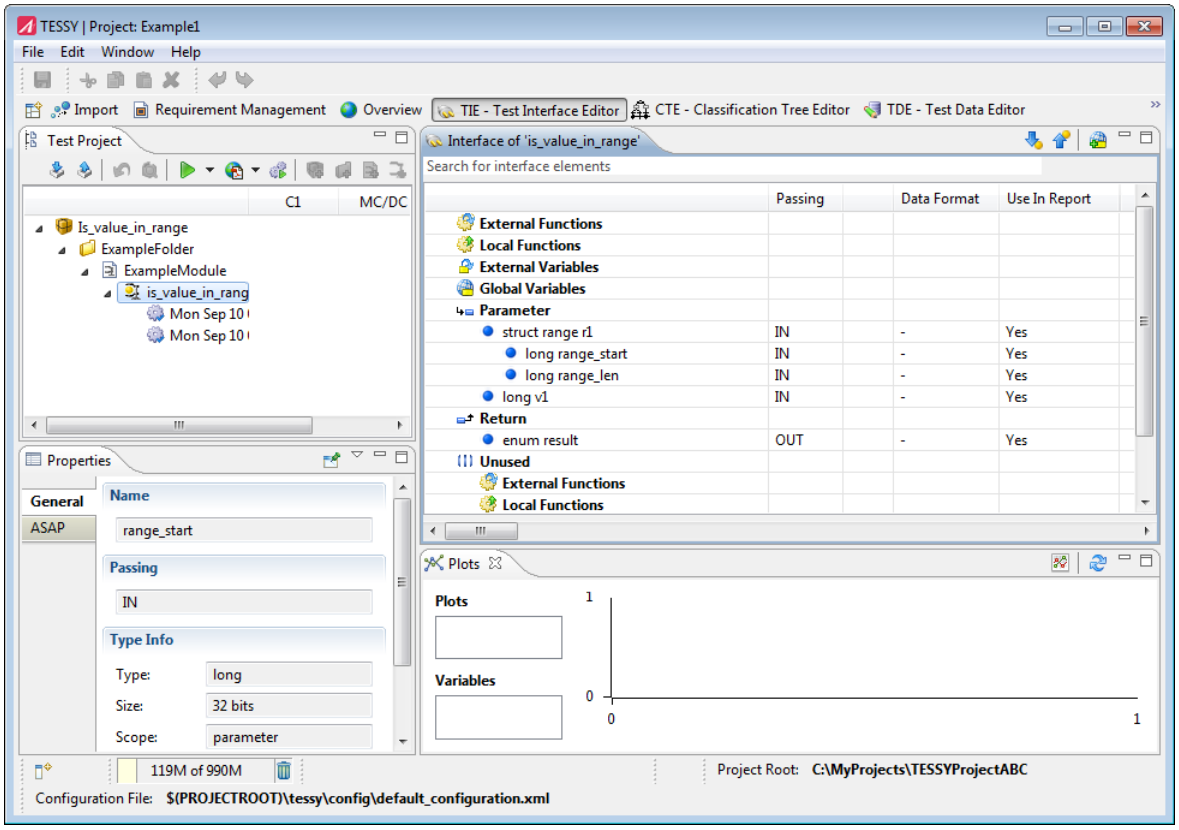


Figure 6.148: Perspective TIE - Test Interface Editor

Within the TIE you determine which values are input and which ones are output variables. **Input values** are all interface elements that have to be set before execution of a test object. **Output values** are compared with the expected values after test execution.

After configuring the test environment of a module and opening the module the analysis of the respective source files starts. The functions found within the source files will be available as test objects, TESSY will try to assign useful default passing directions automatically.

You need to specify missing information that TESSY could not determine automatically, i.e. array dimensions or values of enumeration constants. This can happen due to the usage of the “sizeof” operator when declaring arrays or enumeration constants.

6.7.1 Structure of the TIE perspective

Pane	**Location** (default)	**Function**
Test Project view	upper left	Same view as within the Overview perspective.
Properties view	lower left	Same view as within the Overview perspective.
Interface view	upper right	To display all interface elements of the test object and to provide the edit fields to enter passing directions of variables as well as additional information.
Plot Definitions view	right	To create and configure plots (same view as within the TDE perspective).

Table 6.52: Structure of TIE

6.7.2 Test Project view

6.2.2 Test Project view

The Test Project view displays your test project which you organized within the Overview perspective.

Important: We recommend to do any changes of the test project structure within the Test Project view of the Overview perspective. The view layout of this perspective is optimized for this purpose!

6.7.3 Properties view

Passing directions of a variable

The Properties view is context sensitive: You can view the passing direction of a variable (e.g. IN, OUT, IRRELEVANT) if you select the variable within the Interface view. Then the Properties view will display the passing direction and the type information (see figure 6.149).

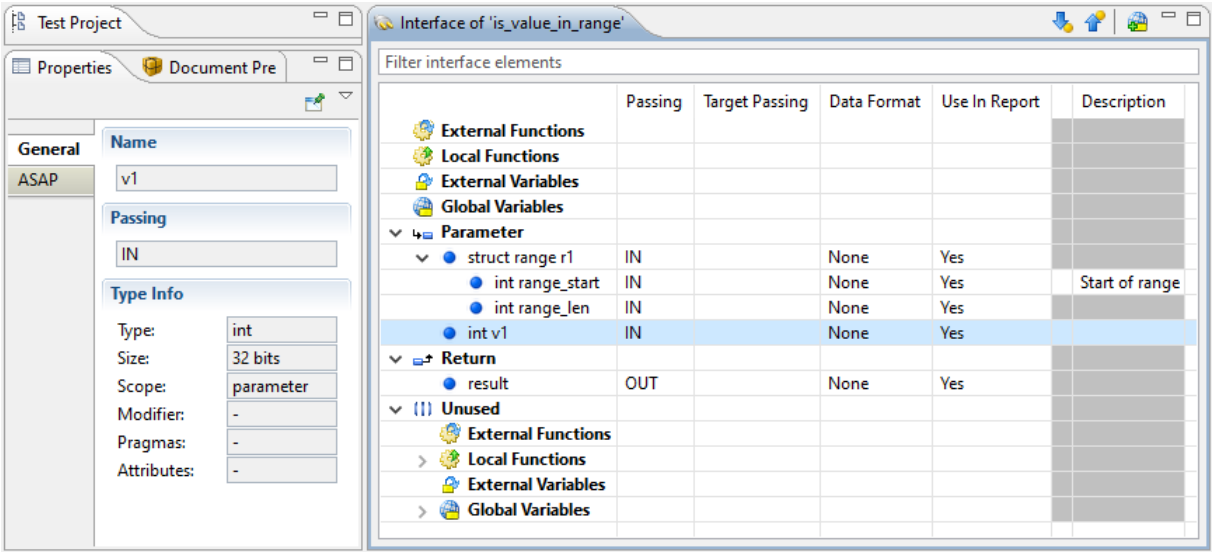


Figure 6.149: Information of passing direction and type

6.7.4 Interface view

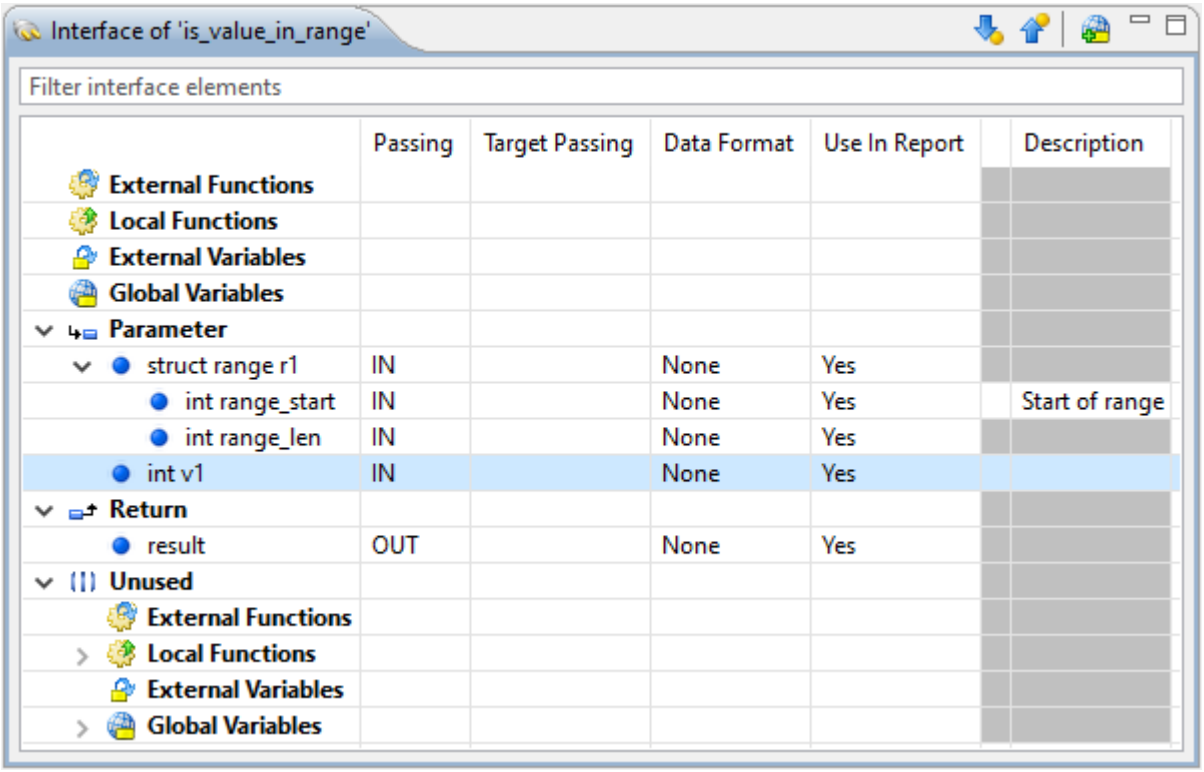


Figure 6.150: Interface view

6.7.4.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Highlights the next undefined value.
	Highlights the previous undefined value.
	Creates a new variable.

Table 6.53: Icons of the Interface view

6.7.4.2 View icons

Icon	Meaning
	External functions
	Local functions
	External variables
	Global Variables
	Parameter
	Return
	Unused

Table 6.54: View icons of the Interface view

6.7.4.3 Status indicators

Indicator	Status
	Function (not stubbed) / Undefined external variable
	Stubbed function / Defined external variable

continue next page

Indicator	Status
	Function with advanced stub function
	Function with global stub function

Table 6.55: Status indicators of the Interface view

6.7.4.4 Handling

You can browse through the interface items of the currently selected test object. An arrow in front indicates further levels (see figure 6.151).

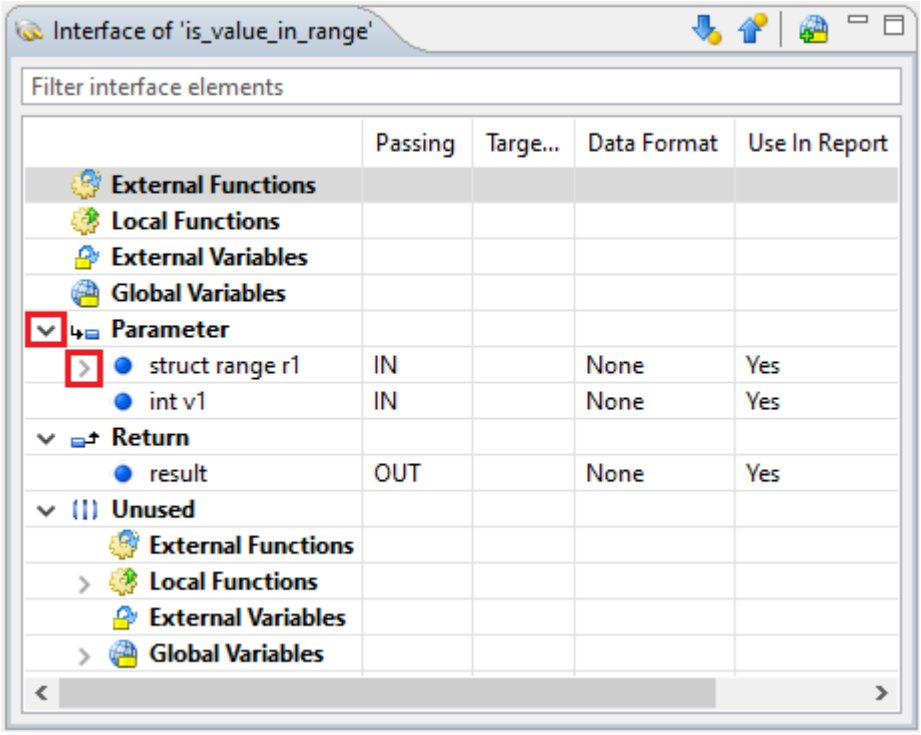


Figure 6.151: White arrow indicating further levels, black arrow when expanded

6.7.4.5 Interface elements

Interface elements

The variables are either read within the function (IN), written within the function (OUT), both read and written (INOUT), to be altered by usercode (EXTERN), or they are simply not used within the function (IRRELEVANT). The TIE classifies all recognized interface elements of the test object into the following sections:

External Functions	All functions which are not defined within the source file(s) of the module. These functions are called from the test object.
Local Functions	All functions defined within the source file(s). These functions are called from the test object.
External Variables	External declared variables which are not defined within the source file(s).
Global Variables	Global variables and module local static variables which are defined within the source file(s).
Parameter	Parameter of the test object.
Return	Return value of the test object.
Unused	Contains all sections and the related interface elements which are not used in the current test object.

Table 6.56: Classification sections of interface elements

6.7.4.6 Setting passing directions

The passing direction reflects the kind of usage for each variable while testing the test object. You can specify how TESSY treats a value for an interface variable either to provide the value before test execution (IN) or to keep the value for evaluation and reporting after test execution (OUT).

Setting passing directions

To set the passing directions:

→ Click in the relevant cell of the element in the column “Passing”.
A drop-down menu will be displayed with the available options IN, OUT, INOUT and IRRELEVANT.

→ Select a suitable passing direction.

You have to specify one of the following passing directions for each interface element:

- provide an input value for that interface element, because the element is only read by the test object (IN),
- evaluate and report the results of that interface element, because the element is only written by the test object (OUT),
- both provide a value and evaluate the result, because the interface element is both read and written by the test object (INOUT),
- provide a value within the UCE (Usercode Editor) of TESSY (EXTERN) . With this setting, the interface element is visible in the scope of the user code and may be set using C code,
- not use the interface element at all (IRRELEVANT). In this case, you will not see this variable for all further testing activities.

The following table shows possible passing directions of the different types of interface elements:

Direction: / **Element:**	**IN**	**OUT**	**INOUT**	**EXTERN**	**IRRELEVANT**
External variable	x	x	x	x	x
Global variable	x	x	x	x	x
Parameter	x			x	x
Return		x		x	x

Table 6.57: Possible passing directions of the interface elements

Automatic analysis of the passing directions

During processing when opening the module, TESSY analyzes the passing directions automatically and stores its findings in the interface database. This information is available in the TIE as default values of the passing directions. TESSY analyzes the usage of individual interface elements by the test object.

Warning: Although TESSY usually correctly recognizes all interface settings, open the TIE for every test object and make sure that the values are set correctly or do match your needs!

Depending on that usage, the following passing directions will be set as default:

Read only	IN
Write only	OUT
Read and write	INOUT
Not used	IRRELEVANT

Table 6.58: Default passing directions

In case that the passing directions or any other interface information could not be determined the respective fields in the TIE will be marked “UNKNOWN” or “?”. If TESSY could not calculate the size of an array dimension enum value (indicated with a question mark), you have to set them manually.

Reset passing direction to default

TESSY analyzes the usage of individual interface elements by the test object. Change the passing direction of an interface element to suite your needs.

Reset the passing direction for all interface elements of one section:

→ Select the respective section and click “Reset to Default Passing” from the context menu (see figure 6.152).

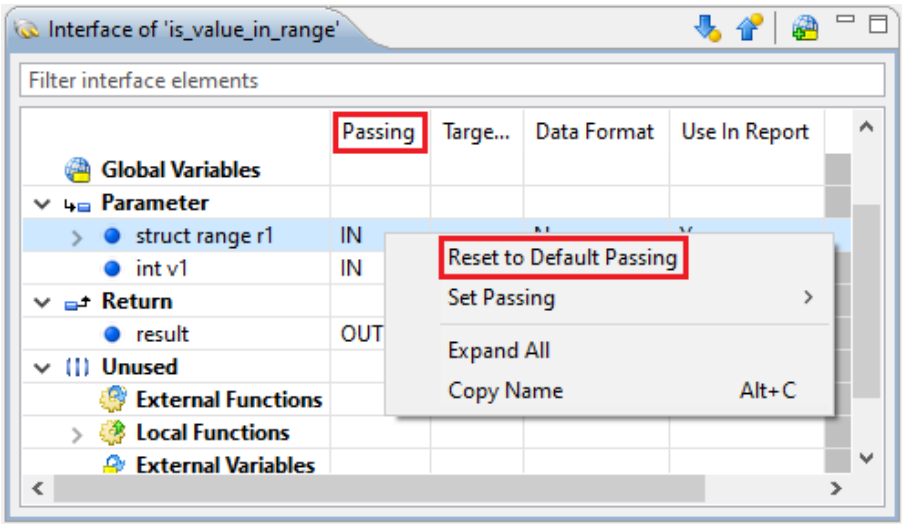


Figure 6.152: Resetting passing directions

Reset the passing direction only for an individual interface element:

→ Select the respective interface element and click “Reset to Default Passing” from the context menu.

6.7.4.7 Setting the data format

Setting the data format

You can change the data format:

→ In the row “Data Format” click into the cell to open the pull-down menu.
→ Click on a format to change it either to “Decimal”, “Hexadecimal” or “Binary” (see figure 6.153).

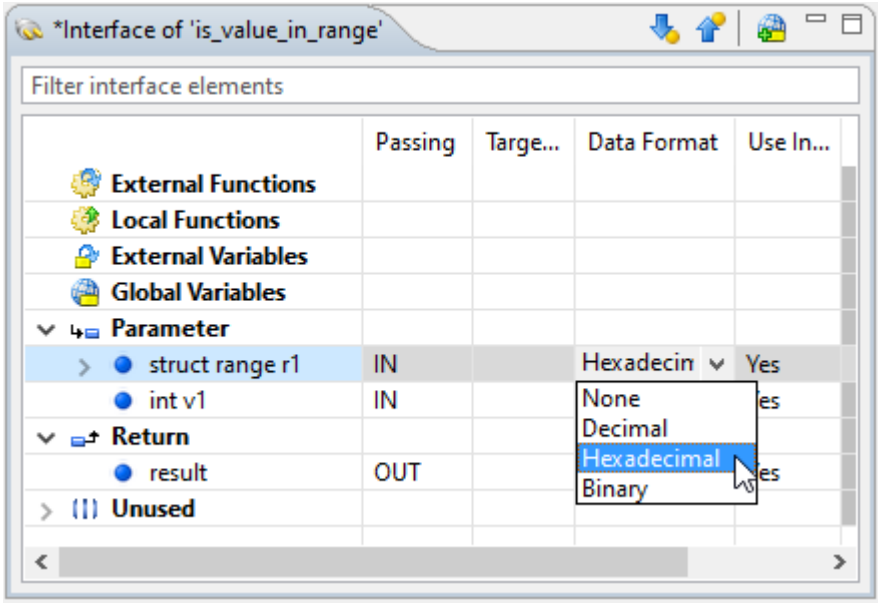


Figure 6.153: Setting the data format

Important: If you change the data format, all newly entered values within the Test Data view of the TDE will be formatted into the new format. Existing data will not be formatted!

6.7.4.8 Setting passing direction of special data types

Pointers and complex data types will be treated slightly different as normal data types.

Pointers

Interface elements of pointer type have two passing directions:

Pointers

Both the passing direction of the pointer itself and the passing direction of the target to which it points have to be specified.

The passing direction of the pointer and the target can be set independently, but they are checked or corrected by TIE to ensure valid combinations.

Complex data types

Complex data types as “Structure” and “Union” have a dependency between their passing direction of the overall structure/union and the passing directions of their components.

To avoid invalid combinations the TIE checks the setting of passing directions for these data types in the following manner:

- When the passing direction of one component is set, TIE determines the resulting passing direction for the overall structure/union and set them automatically.
- When the passing direction for the overall structure is set, all components are automatically set to the same passing direction.

Arrays

The passing direction of the data type “Array” will be set for the entire array to the same direction. Only one passing direction will be defined for the whole array and all elements. If

Arrays

the array is made up of structured array elements (e.g. structures), it is possible to define different passing directions for the components of these structures.

Array as parameters will be shown as pointers within the interface. They can be initialized with NULL or pointing to a dynamic object, synthetic variable or global variable (see figure 6.154).

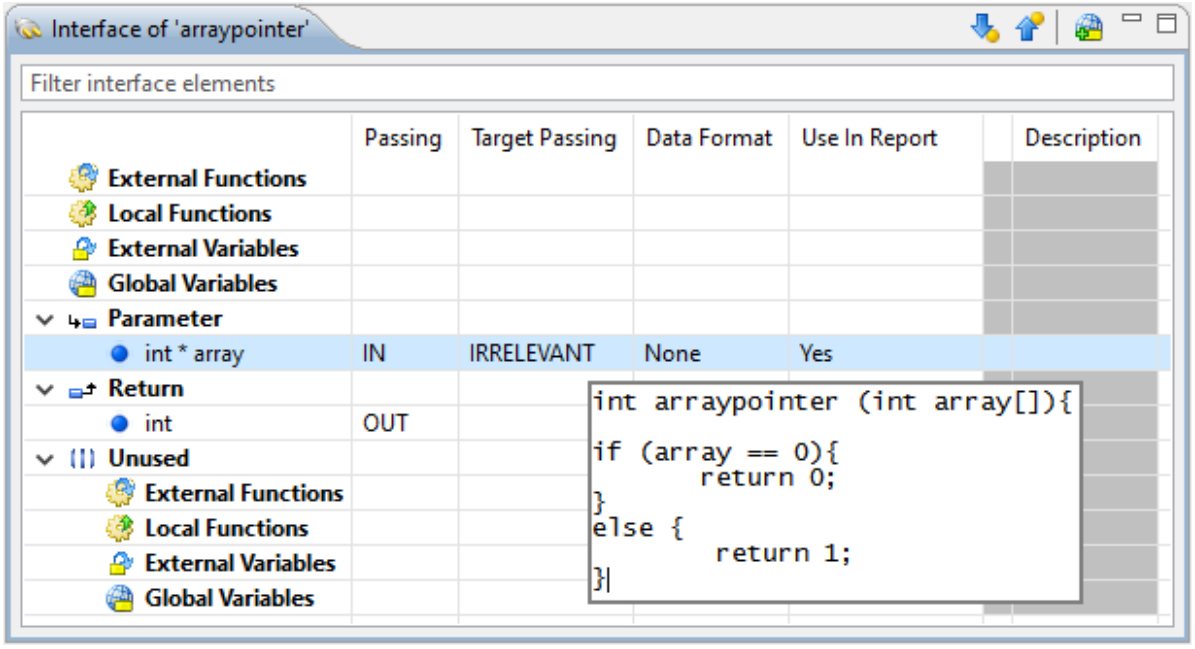


Figure 6.154: Array as pointer

6.7.4.9 Defining stubs for functions

The TIE displays all functions used by the test object either in section External Functions or Local Functions and it provides an interface to define stubs for these functions that will be executed instead of the original function. TIE distinct two different stub functions:

- A **stub function** for which you can enter C code.
- An **advanced stub function** that allows to provide values for parameters and return values of stub functions like normal variables in the TDE.

You can define stubs globally for all test objects of the module or create a stub independently of the global module setting.

The following restrictions apply to advanced stubs:

- Advanced stub variables cannot be created for arrays and pointer to arrays.
- Pointers that are components of structs or unions will always be handled as IRRELEVANT.
- Multiple calls to advanced stubs will always use the same input values and the results of the last call will be taken as output values.

For more information about entering values please refer to subsection 6.9.7.4.

Important: Stub functions returning a value should be implemented with stub code that returns a defined value. If no stub code is provided for such non-void functions, a random value will be returned which depends on the current memory layout and stack contents. Therefore TESSY aborts the test driver generation in such a case with an error. You can select to ignore this error within the test execution preference pages if you are sure that the return values of your stub functions are not used. (For more information see Executing tests.)

Warning: Due to the possibility of unforeseen side effects, please refrain from stubbing standard or system functions of your chosen compiler.

For example: A stub of “memcpy()” in a GCC configuration may provoke an access violation error or stubbing “__ARM_disabl_irq” within Keil ARM will fail the build process entirely with an error message that the IDE installation may be damaged.

Creating a stub function

To create a stub:

→ Right-click the function and choose “Create Stub” from the context menu (see figure 6.155).

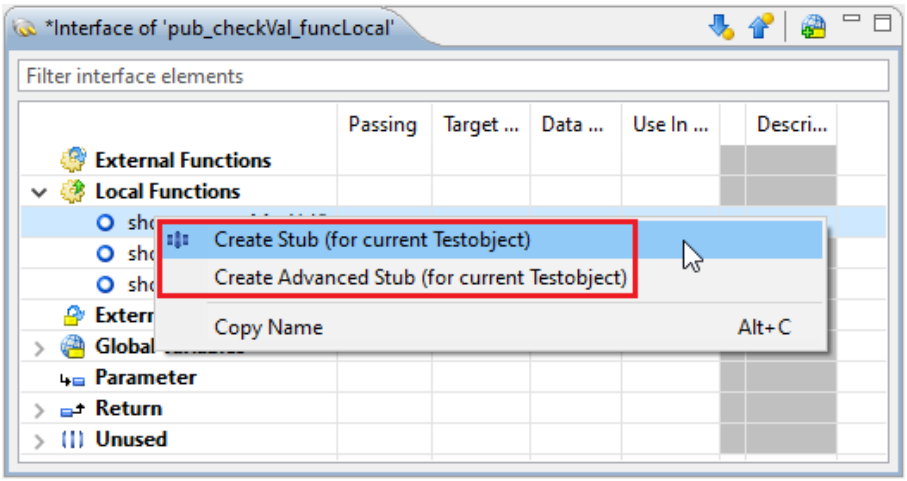


Figure 6.155: Create a stub function within the context menu

You can create stubs either for external or local functions which will be executed instead of the original functions.

There are several options available:

- Create stubs for all functions at once for all test objects of the module (global setting).
- Create stubs for a single function for all test objects of the module (global setting).
- Create stubs for the current test object.
- Use global stub settings.

The enhancement to normal stub functions are advanced stub functions, that allow to provide values for parameters and return values of stub functions like normal variables.

TESSY checks if the stub is called by the test object with the specified parameter values, otherwise the test fails. You can also provide a return value for further processing by the test object. This reveals if the test object handles the return values of the stub function correctly.

Creating an advanced stub function

To create an advanced stub:

→ Right-click the function and choose “Create Advanced Stub” from the context menu.

You can create advanced stubs either for external or local functions.

There are several options available:

- Create advanced stubs for all functions at once for all test objects of the module (global setting).
- Create advanced stubs for the current test object.
- Use global stub settings.

6.7.4.10 Other interface settings

For test execution the information on data types of the test object interface has to be complete. The dimensions for arrays, the values of the enumeration constants for enumerations, and the dimensions for bitfields have to be defined. If these values have been automatically recognized by TESSY while opening the module, the respective text field will show the calculated value for every data type. In this case, it is not possible to change these values.

If a value for an interface element has not been recognized automatically, the respective text field will be empty or contain the value -1. In case of arrays TIE will also use question marks to indicate this issue, i.e. array[?]. In all those cases you have to add values manually.

Warning: Wrong array dimensions or wrong values for enumeration constants can cause the test object to crash during test execution! TIE cannot check for plausibility of used values!

6.7.4.11 Creating new variables

You can create new (synthetic) variables for usage within your test cases based on all basic C/C++ types as well as based on all types available within your source files.

Important: Restrictions apply for synthetic variables using types that are defined within the source file only (and not within a header file): Such synthetic variables cannot be used within stub code for local functions or within fault injection code.

If synthetic variables with source file defined types shall be used (e.g. within prolog epilog) and either stubbing of local functions or fault injections are active, you may need to set the module attribute “Synthetic Declarations In Source” to false in order to prevent a compilation error.

To create a new variable:

→ click on the icon (New Variable).

Important: Creating enum variables is only possible for enum types with either a tag name or which were defined using a typedef.

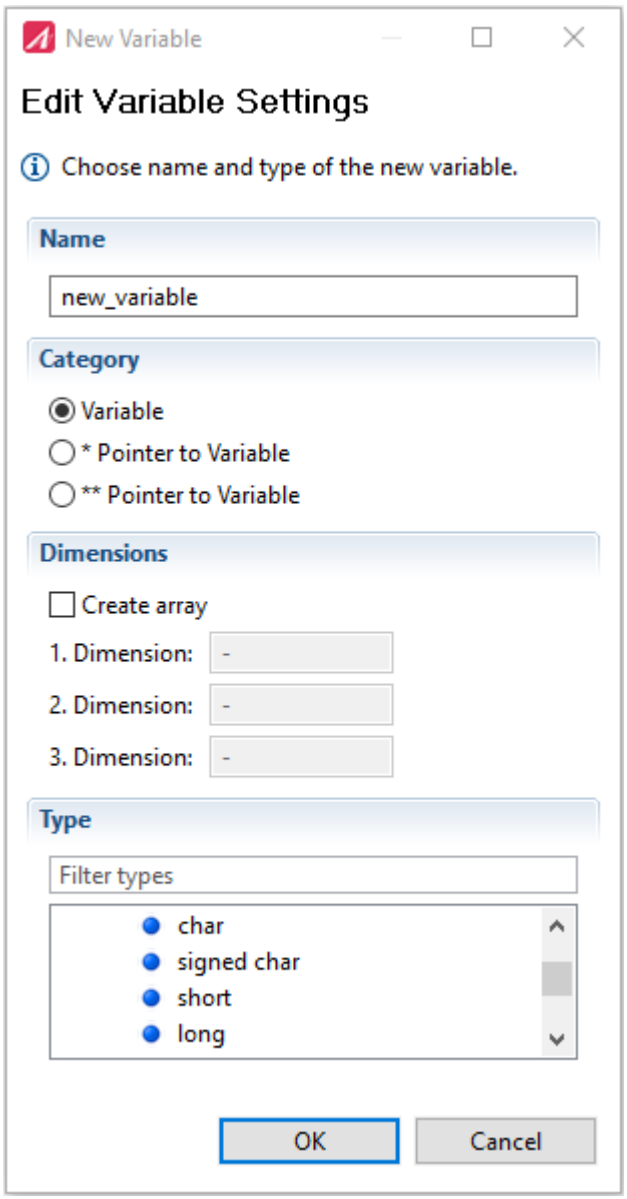


Figure 6.156: Create a new variable

Within the New Variable dialog:

- → Specify the name of the variable.
- → Select a type and whether it shall be a pointer or an array.
- → Click “OK”.

The new variable will be shown within the TIE view with the default passing direction “INOUT”. Adjust the passing direction to your needs.

Important: Variables created for one test object are global and therefore available for all test objects. This of course means that deleting a (synthetic) variable has global consequences too. The variable will be deleted in every test object where it is in use.

Synthetic variables in other test objects are by default listed in the ”Unused” section of the interface. They can be moved to “Used Variables” manually if necessary.

6.7.4.12 Using alias names

TESSY provides an alias name mechanism to mirror the usage of **#define** to access variables during the whole testing cycle (e.g. access to individual bits of common bitfield structures). You will see your variables within TESSY named exactly as the defines you are using in your code to access these variables.

```
1  struct bits
2  {
3     unsigned int   b0:1;
4     unsigned int   b1:1;
5     …
6     unsigned int   b7:1;
7     unsigned int     :8;
8  };
9
10 union char_bit
11 {
12    unsigned short port;
13    struct bits    b;
14 };
15
16 union char_bit door_light_c;
17
18 #define door_light_left_b  door_light_c.b.b0
19
```

Figure 6.157: Example code snippet for alias names

To activate the usage of alias names:

→ Change the value of the TEE attribute “Use Alias Names” to “true” (Refer to chapter 6.5.7.3).

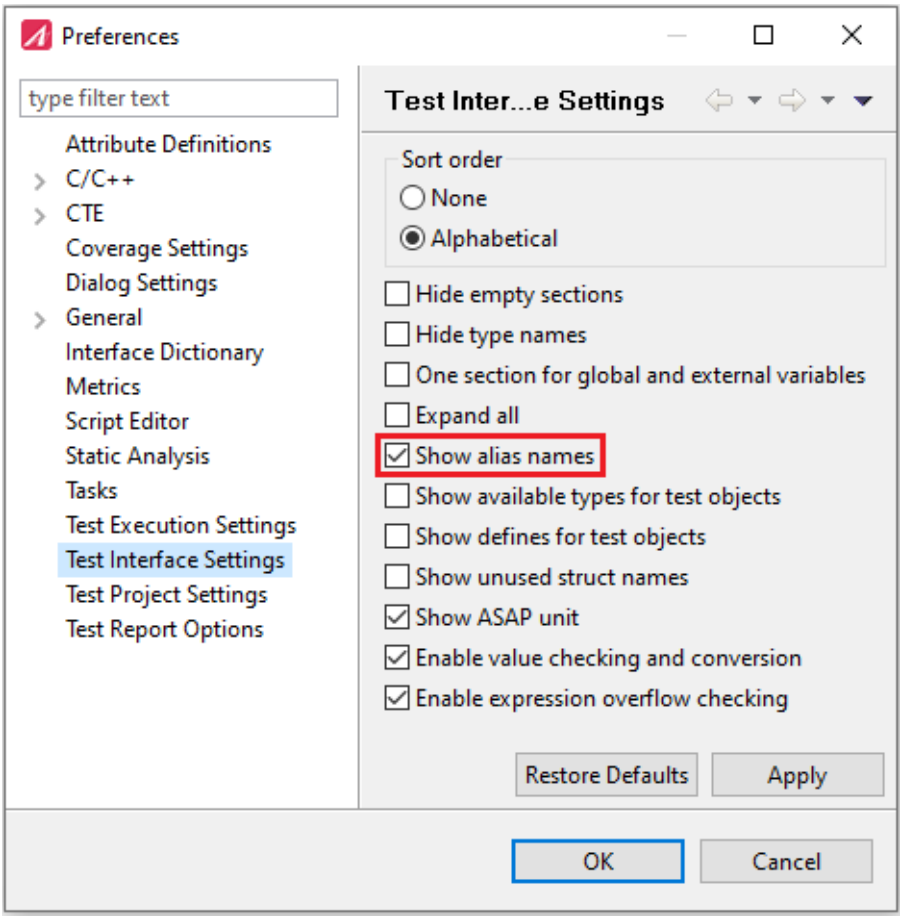


Figure 6.158: Show alias names preferences

→ Check the “Show Alias Names” preferences button.

Instead of the real variable name “door_light_c.b.b0” that you are not using within your code, you will now see the virtual name “door_light_left_b” given through the define within the TESSY interface and within the test reports.

6.7.4.13 Defining external variables

External variables are listed in the interface of the TIE and can be handled like any other variable. You can set e.g. the passing direction, the data format or add descriptions.

External variables are by default defined. This is indicated by an ●.

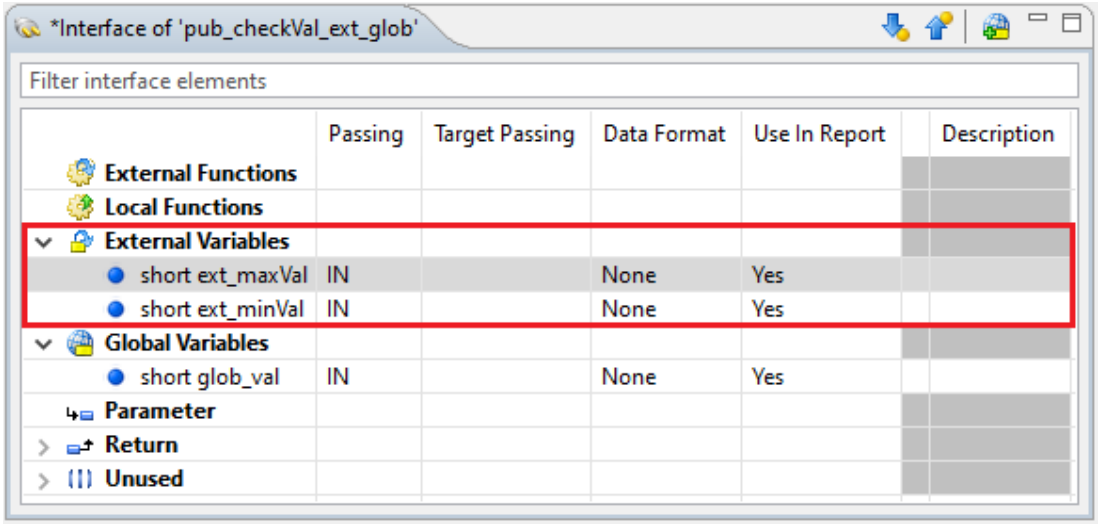


Figure 6.159: Defined external variables

To undefine an external variable:

→ Right-click the respective variable to open the context menu (see figure 6.160).
→ In the context menu click “Don’t define Variable”

Undefining external variables

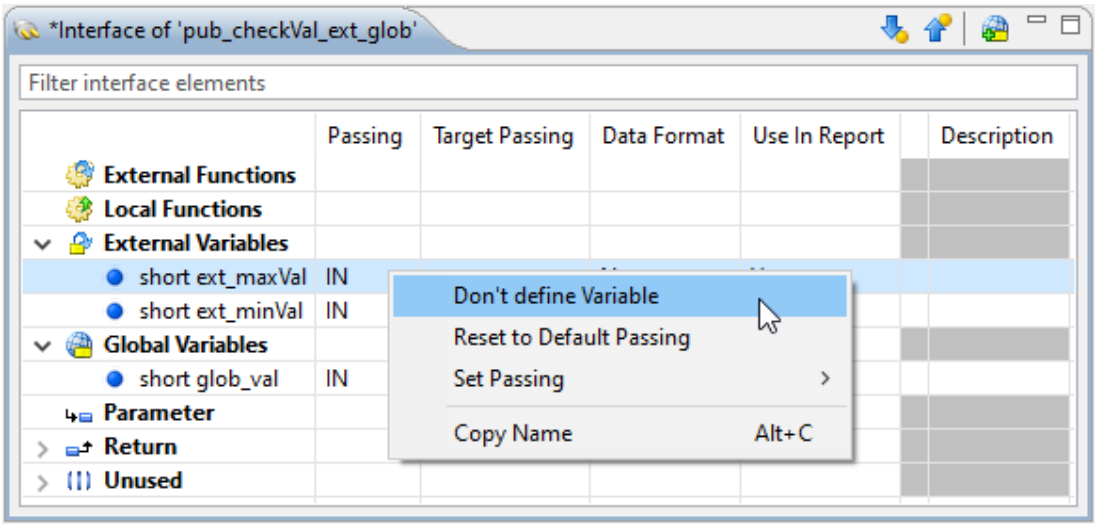


Figure 6.160: Undefining an external variable

The now undefined external variable appears with an . (See figure 6.161).

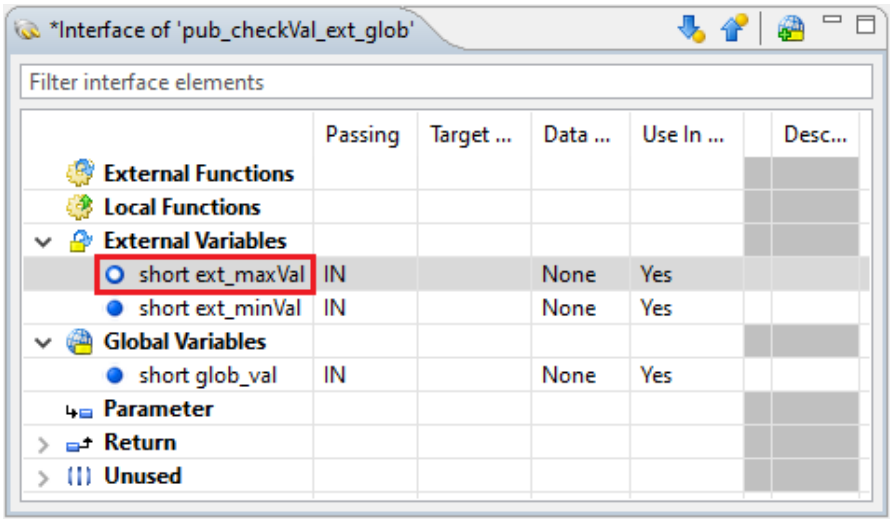


Figure 6.161: Undefined external variable

6.7.4.14 Changing the default settings in the Test Environment Editor (TEE)

The default behavior for external variables or external functions is as follows:

- External variables will be defined.
- External functions will not be stubbed.

If you want to change e.g. the default settings of the external variables or functions for your project, you should change it within the related configuration files.

Edit default settings in the TEE

To open the TEE:

→ Click “File” in the menu bar.
→ Then click “Edit environment…” .
The Test Environment Editor will open.
→ You can now change the default settings to create the desired behavior.

To edit the default settings of the external variables:

→ Set “Enable Define Variables” to “false” in your configuration file (see figure 6.162).
→ Save your settings when closing the TEE.
Changes will be active with the first opening or after a reset of the module. The interface of existing modules will not be changed.

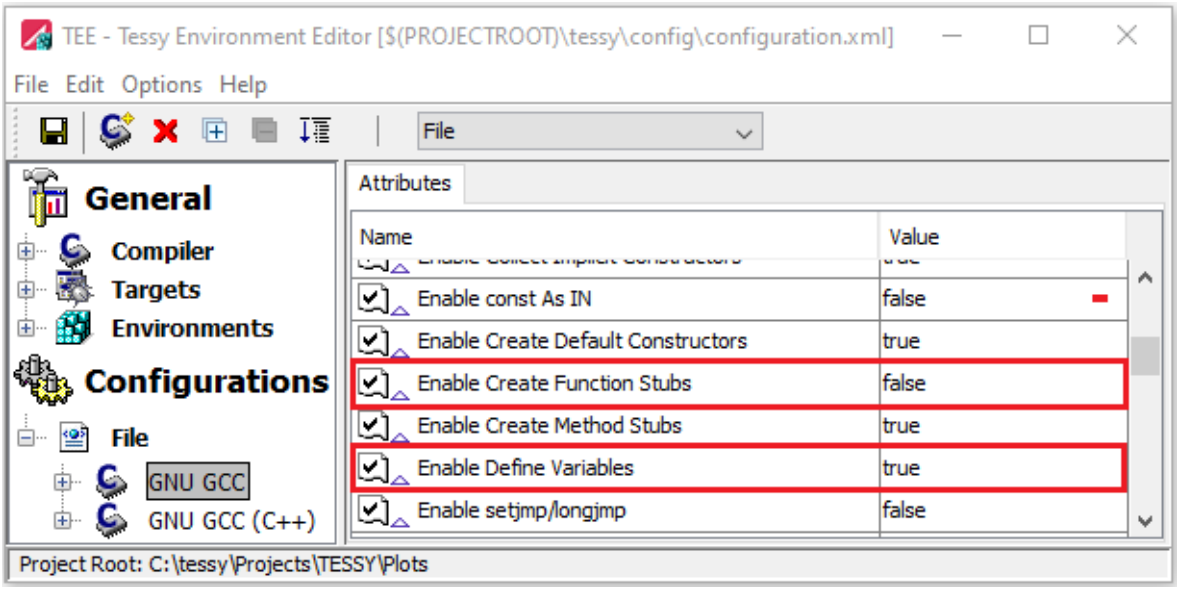


Figure 6.162: Change external variable/function settings in the TEE

To edit the default settings of the external functions:

→ Set “Enable Create Function Stubs” to “true” in your configuration file.

→ Save your settings when closing the TEE.
Changes will be active with the first opening or after a reset of the module. The interface of existing modules will not be changed.

More information about available attributes and their setting you can find in the application note “Environment Settings (TEE)” in TESSY (“Help” > “Documentation…”)

6.7.4.15 Handling unused functions or variables

The list of unused functions and variables shows all external items that are used by other test objects of the same module but not by the current test object itself. Because such items need to be defined in order to link the test driver, you need to review this section to check whether all external references have been defined or stubbed.

Please note: System functions or intrinsic functions must not be stubbed. For more information please refer to chapter 6.7.4.9 Defining stubs for functions

Local functions are available in this list to be able to move them to the used functions of a test object. Creating a stub for a local unused function does not usually make sense, because the local function is not called from test object.

One possible use case could be the usage of such local functions as a value of a function pointer. Via stub code it is possible to influence the behavior of the function.

Depending on the type of function or variable it is possible to stub the function or not define the variable. You can also copy the name or move the function or variable to used functions or variables.

To handle unused functions or variables:

→ Right-click on the respective function or variable to open the context menu.
→ Choose your desired option (see figure 6.163).

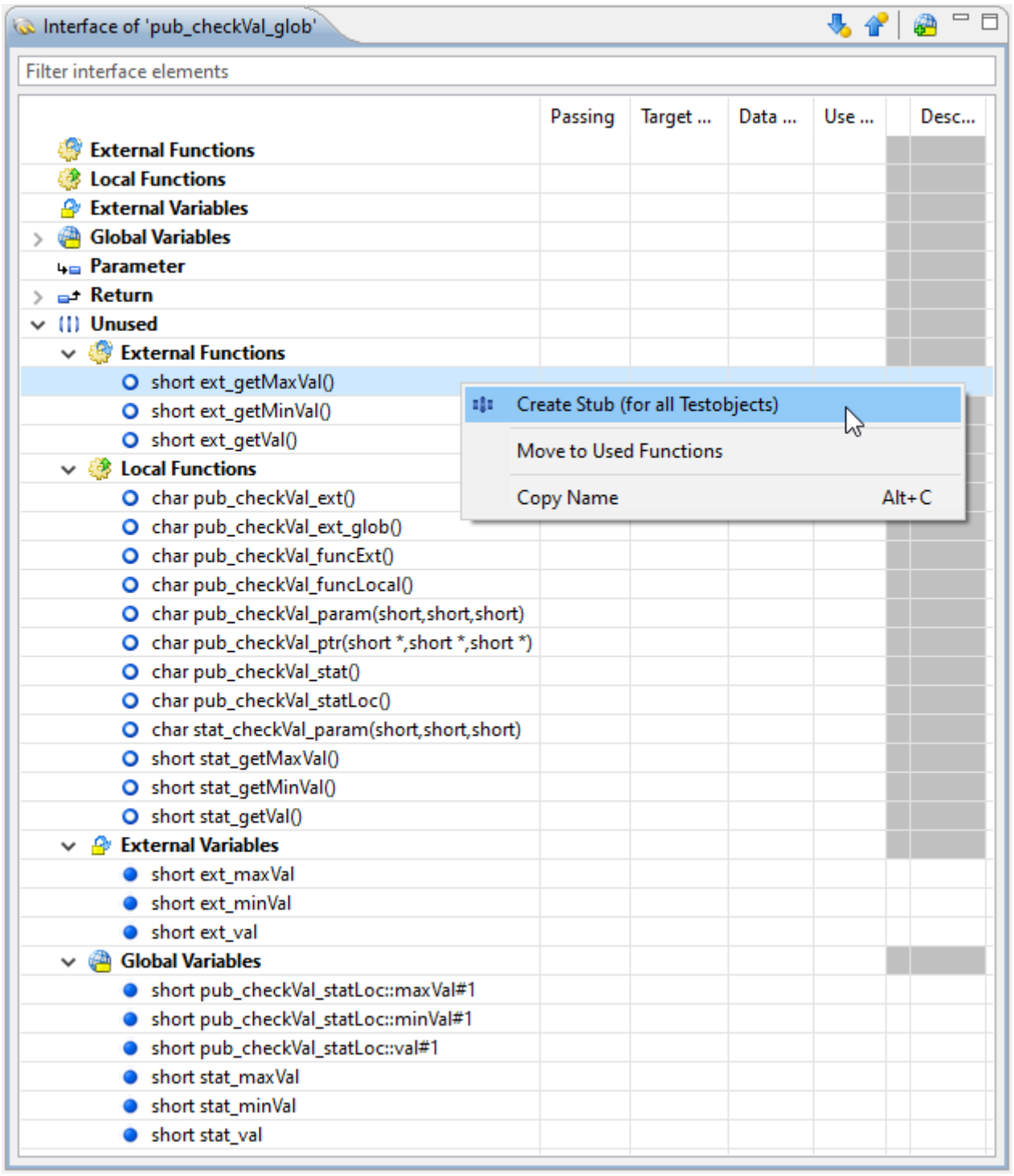


Figure 6.163: List of unused functions and variables in the TIE interface

6.7.5 Plot Definitions view

You can find more information about plot definition in chapter 6.9.14 Plots view.

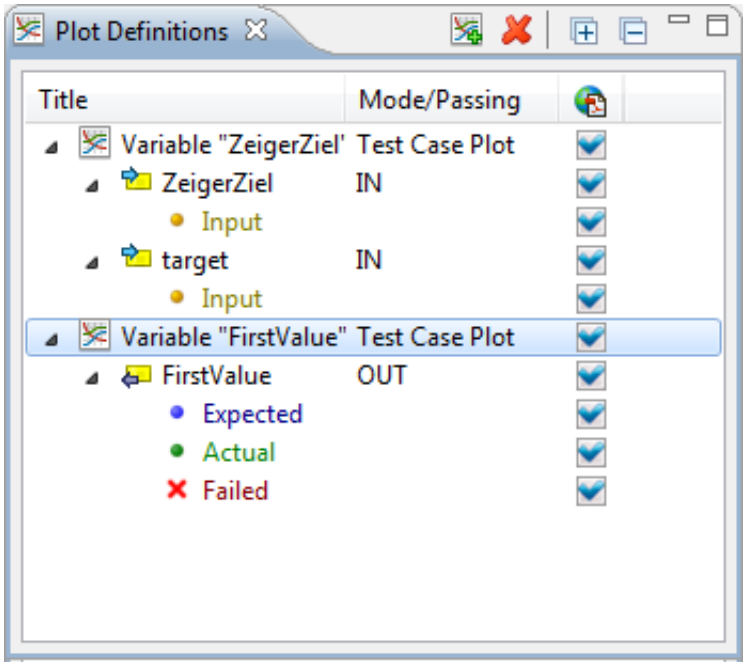


Figure 6.164: Plot Definitions view

The Plot Definitions view displays the plots for a selected test object or test run.

Within the view you can create or configure plots for a selected test object. You can also select whether plots should be used in reports.

6.7.5.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Creates a new plot with no variables.
	Deletes the selected plot respectively removes the selected variable from its plot.
	Expands the tree.
	Collapses the tree.

Table 6.59: Icons of the Plot Definitions view

6.7.5.2 Creating plots

TESSY can handle different kind of plots:

A **test case plot** spans over all values of all test cases of the selected variables.
A **test step plot** provides one curve for each test case spanning over all values of the test steps of this test case. This requires at least two test steps for each test case to define a valid curve.
An **array plot** creates plots for array type variables. There will be one curve spanning over the array values for each test step.

To create a plot:

→ Click on to create a new plot.
→ Right-click on the newly created plot to rename it or set the included test items (see figure 6.165).

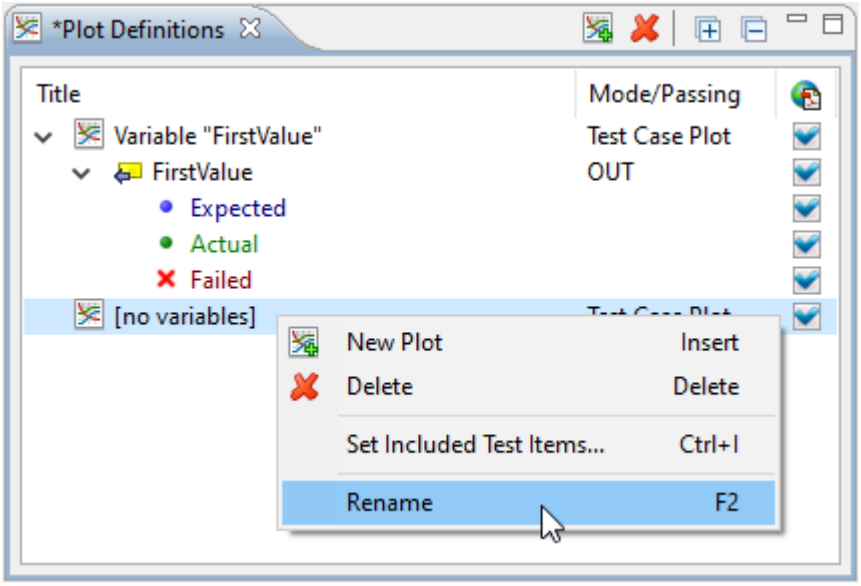


Figure 6.165: Rename a new plot

→ Click on the newly created plot to choose the desired kind of plot (see figure 6.166).

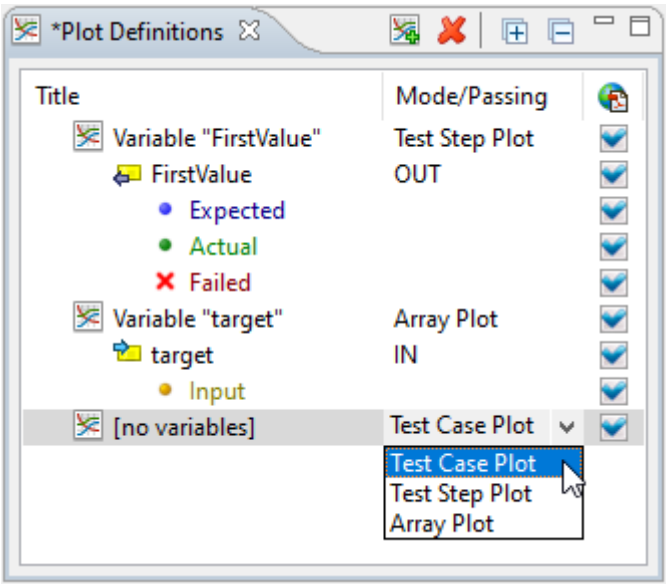


Figure 6.166: Plot Definitions menu

It is possible to drag variables from the TIE or the TDE onto the Plot Definitions view. Also plots and variables can be dragged and dropped within the Plot Definitions view. (see figure 6.167)

Adding variables to plot

To add variables to the plot:

→ Drag and drop the variables to the Plot Definitions view.

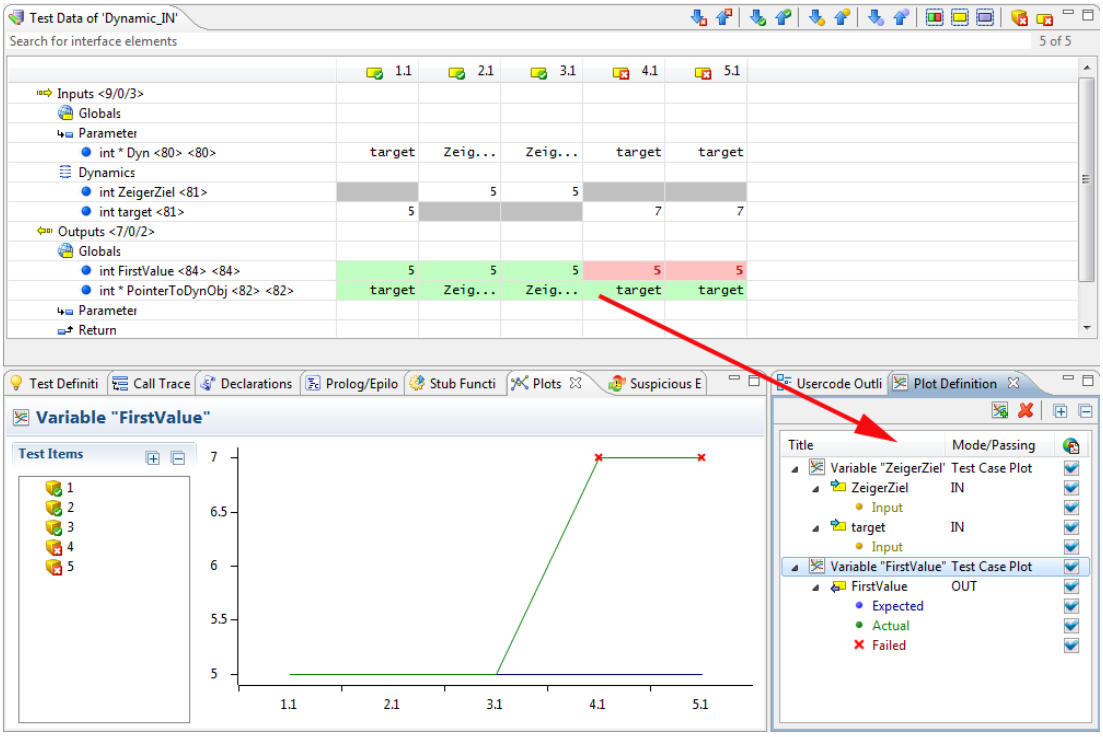


Figure 6.167: Adding variables to a plot in the TDE

The table below shows the different possibilities:

Source	Drag onto	Result
Scalar or array variable from TIE or TDE	Empty area Plot	A new plot containing the variable is created. The variable is added to the plot if possible. (1)
Variable from Plot Definitions view	Empty area Another plot	The variable is moved to a new plot. (2) The variable is moved to the other plot. (1, 2)
Plot from Plot Definitions view	Empty area	A copy of the plot is created. (3)

Table 6.60: Drag and drop handling with the Plots and Plot Definitions view

(1) Restrictions apply: A scalar variable cannot be added to an array plot, and whole arrays cannot be added to a test case or test step plot (whereas single array elements can be added to test case or test step plots).
(2) If CTRL is being pressed while dropping the variable, it will be copied instead of moved to the other plot.
(3) Only applies if CTRL is being pressed while dropping the plot.

6.7.5.3 Reporting plots

Only the plots that are ticked with “Use in Report” will be displayed within the reports (see figure 6.168).

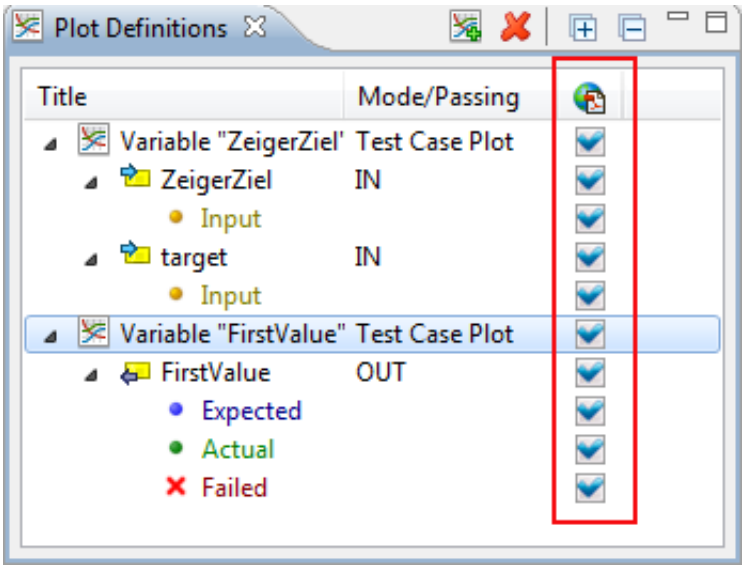


Figure 6.168: Using plots in report

6.8 CTE: Designing the test cases

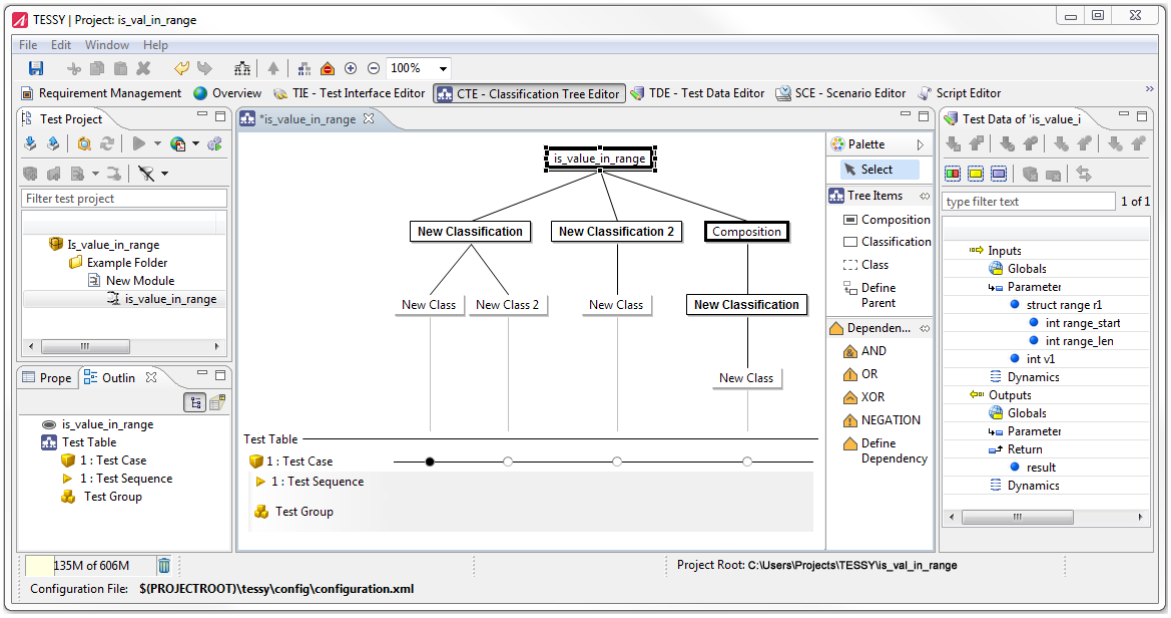


Figure 6.169: CTE perspective

6.8.1 The basic idea

3.2 The Classification Tree Method (CTM)

After preparing a test in the TIE, you need to create well designed test case specifications. The Classification Tree Method provides a systematical approach to create test case definitions based on the functional specification of the function or system to be tested. TESSY includes the specialized Classification Tree Editor CTE which assists you in creating low redundant and error sensitive test cases.

The basic concept of the Classification Tree Method is to first partition the set of possible inputs for the test object separately and from different aspects, and then to combine them to obtain redundancy-free test cases covering the complete input domain.

For further general information about the Classification Tree Method (CTM) please refer to chapter 3.2 The Classification Tree Method (CTM).

→ Switch to the CTE perspective to start the CTE in TESSY.

6.8.2 Structure of the CTE perspective

Pane	**Location** (default)	**Function**
Test Project view	upper left	Displays your test project. For editing your test project switch to the Overview perspective.
Properties view	lower left	Displays the properties of tree and Test Table items.
Outline view	lower left	Displays the structure of the classification tree and the Test Table and allows to navigate and select items in the structure.
Classification Tree editor	center	To edit the classification tree.
Test Data view	right	Allows to assign test data to classification tree elements.

Table 6.61: Structure of the CTE perspective

6.8.3 Test Project view

The Test Project view displays the test project which has been organized within the Overview perspective.

6.2.2 Test Project view

Important: We recommend to do any changes of the test project structure within the Test Project view of the Overview perspective. The view layout of this perspective is optimized for exactly this purpose.

6.8.4 Properties view

The Properties view displays all the properties which you have organized within the Overview perspective. Within the CTE perspective this view additionally provides all properties of items used in the classification tree and the Test Table. Most operations are possible.

6.2.3 Properties view

For changing any module related settings switch to the Properties view within the Overview perspective.

6.8.5 Outline view

The Outline view displays the structure of the classification tree and the Test Table and allows navigating and selecting items in the structure.

6.8.6 Classification Tree editor

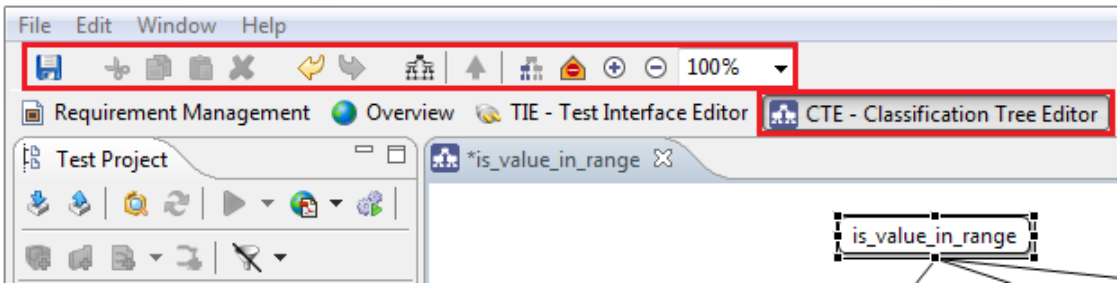


Figure 6.170: Classification Tree editor related tool bar

Classification Tree editor

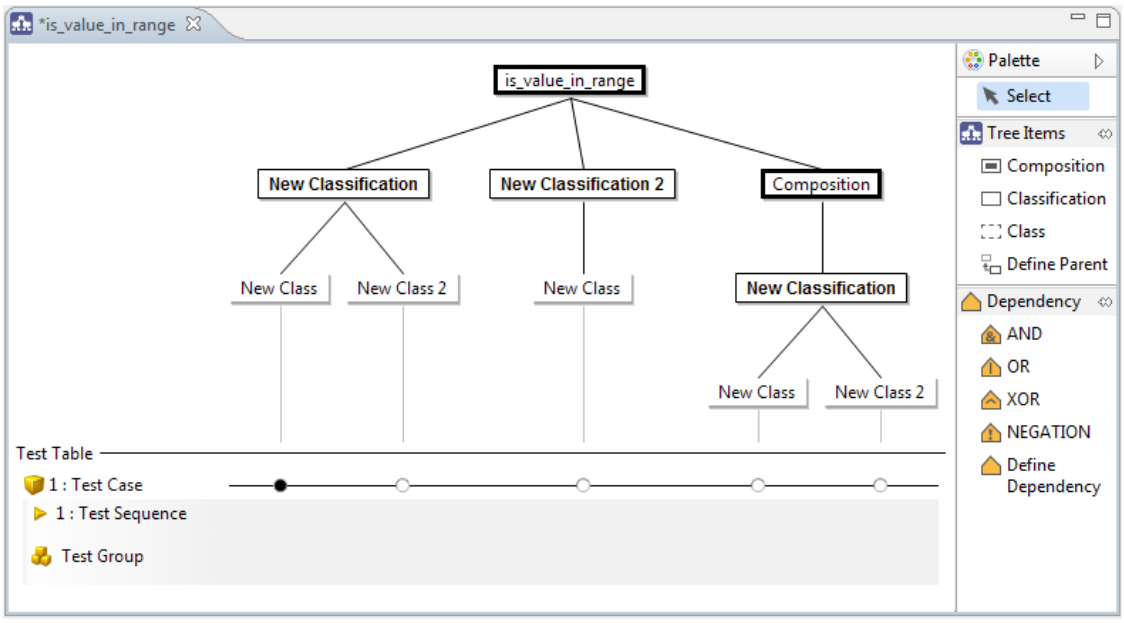


Figure 6.171: Classification Tree editor

Maximize the CTE window within the Classification Tree editor to avoid additional scroll bars and to always show the whole CTE window contents within the perspective.

6.8.6.1 CTE related Icons of the main tool bar

Icon	Action / Shortcut / Comment
	Saves the current contents (Ctrl + S).
	Cuts the selection (Ctrl + X).
	Copies the selection (Ctrl + C).
	Paste (Ctrl + V).
	Delete (Del).
	Undoes the last move or edit operation within the classification tree pane (Ctrl + Z).
	Redoes the last move or edit operation within the classification tree pane (Ctrl + Y).
	Lays out the tree.
	Aligns two or more selected graphical elements to the left.
	Aligns two or more selected graphical elements in the center.
	Aligns two or more selected graphical elements to the right.
	Aligns two or more selected graphical elements on the top.
	Aligns two or more selected graphical elements in the middle.
	Aligns two or more selected graphical elements on the bottom.
	Opens the parent level editor of a view editor (Ctrl + up).
	Selects all leaves that are children of the current selection (Ctrl + L).
	Hides the dependencies in the tree and disables the dependency entries in the palette.
	Zooms in.
	Zooms out.

Table 6.62: Tool bar icons of the Classification Tree view

See figure 6.170 to localize the CTE related tool bar.

6.8.6.2 Structure of the Classification Tree editor

Pane	**Location** (default)	**Function**
Tree area	upper left	Drawing the classification tree with a root, compositions, classifications and classes.
Test Table	lower left	Marking classes of the classification tree in order to define test cases, test sequences and test steps. Every test item creates a new line in the Test Table.
Palette	right	Tool box to create tree items and define parents as well as create and define dependencies.

Table 6.63: Structure of Classification Tree view

6.8.6.3 Palette view

Creating tree items and define the parent structure and dependencies

The Palette view on the right side of the CTE contains a tool box to select and create tree items and define the parent structure. Furthermore the palette provides tool entries to define different types of dependencies for tree items.

Icons of the Palette view

Icon	**Action / Shortcut / Comment**
	Selects elements in the Classification Tree.
	Creates a composition.
	Creates a classification.
	Creates a class.
	Defines the parent for tree items.

continue next page

Icon	Action / Shortcut / Comment
	Creates an AND dependency.
	Creates an OR dependency.
	Creates an XOR dependency.
	Creates a NEGATION dependency.
	Defines a relation between the dependencies or classes.

Table 6.64: Icons of the Palette view

6.8.6.4 Creating classifications, classes and test cases

A test case is formed through the combination of classes from different classifications. For each test case exactly one class of each classification is considered. The combined classes must be logical compatible; otherwise the test case is not executable. You should choose and combine as many test cases as needed to cover all aspects that should be tested.

The method offers a graphical representation of the recursive partitioning of classifications and classes in shape of a classification tree. The classifications are drawn as named rectangles. The respective classes are arranged below. To specify the test cases as combination of classes the classification tree is used as the head of a combination table wherein the classes, which are to be combined, are marked.

For further general information about the Classification Tree Method (CTM) please refer to chapter 3.2 The Classification Tree Method (CTM).

Creating classifications

To create **classifications**:

- → Select the root tree item.
- → Right-click to open the context menu.
- → Choose "□ Add Classification", see figure 6.172 general information about the Classification Tree Method

You can also press Insert on the keyboard to create a classification in the CTE.

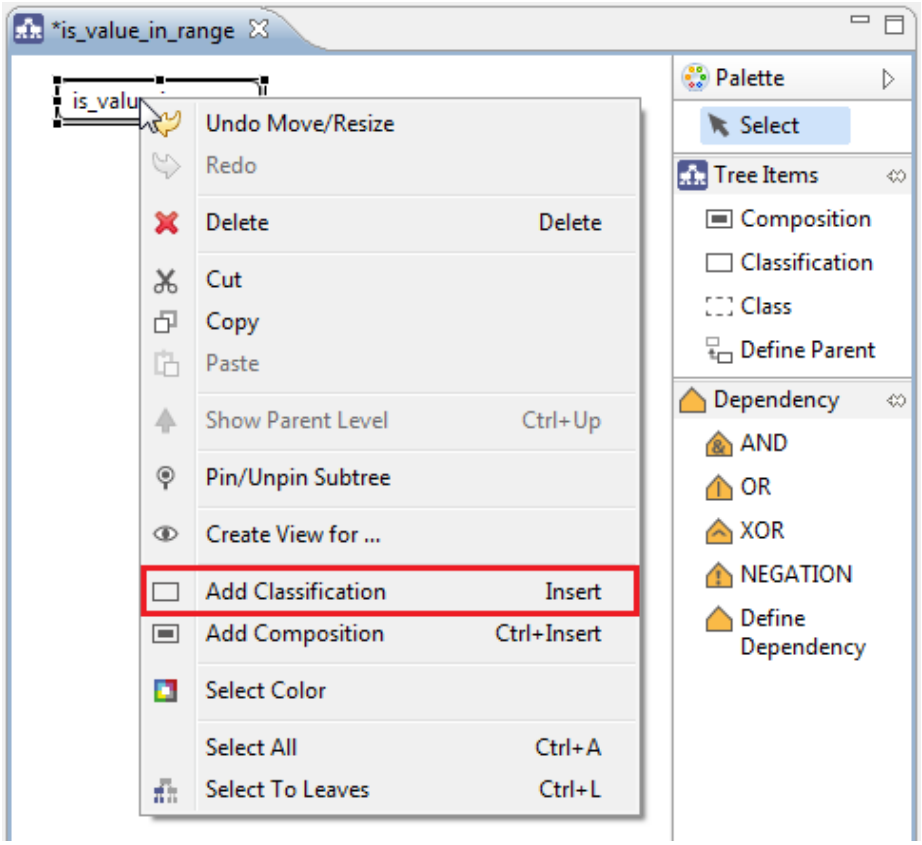


Figure 6.172: Creating a new classification with the context menu

To edit the classification:

→ Double-click the new classification or press F2 after selecting the new classification to start the inline editor for the tree item.

Within the tree area you can move the classifications and other elements with drag and drop: Just left click the element, hold the mouse button and move it to the desired place. You may also select a group of elements and move them the same way. The tree layout will be arranged automatically by clicking in the tool bar.

Creating classes

To create **classes**:

→ Select a classification as parent.

→ Right-click to open the context menu.

→ Choose "Add Class".

You can assign test data to all interface variables for each tree node of the classification tree. This speeds up testing because the test data will be assigned automatically to the test cases via the marked class nodes (refer to section 6.8.7 Test Data view).

Creating test cases

To create **test cases**:

→ Select the Test Table on the lower left.

→ Create the test cases either using the context menu (“ Create Test Case”, see figure 6.173) or press Insert on the keyboard.

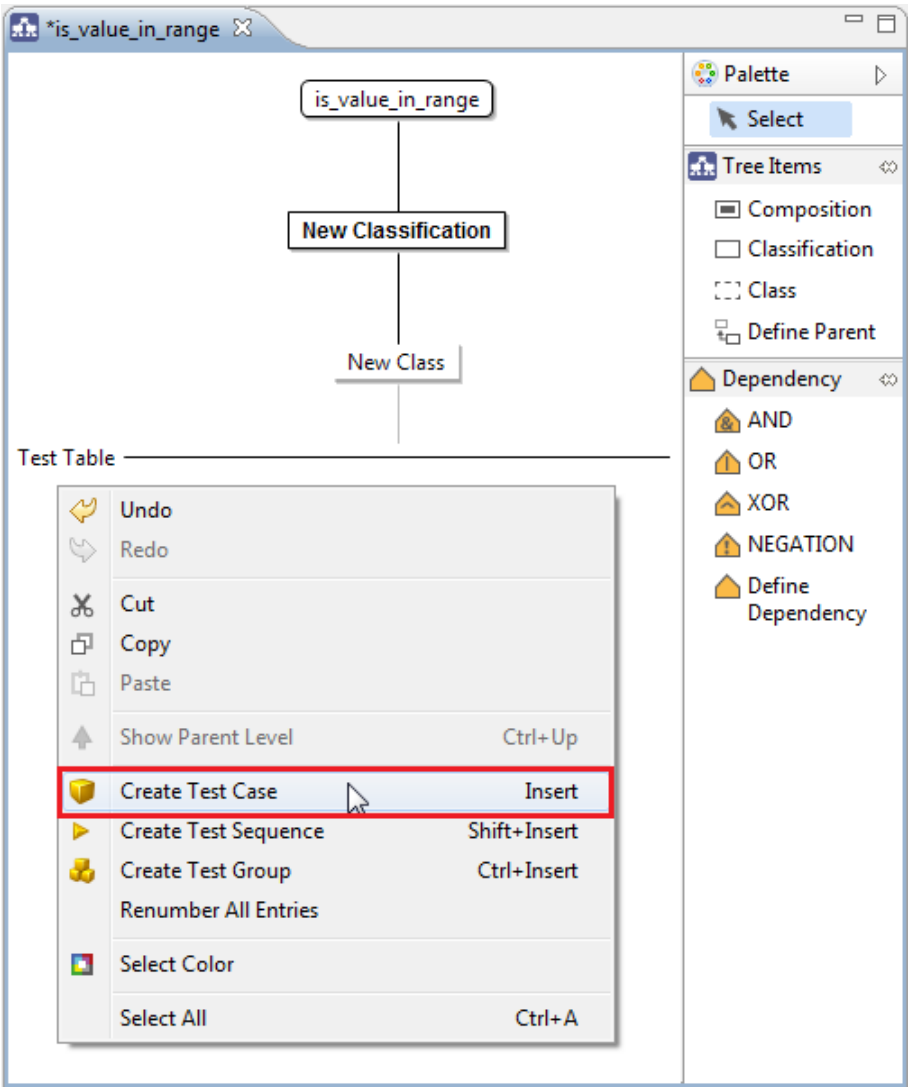


Figure 6.173: Creating test cases in the test item list

Setting marks

Test cases are defined by setting marks in the Test Table:

→ Click on one of the circles to connect a test case with a class. The empty circle will turn into a black circle.

→ Click on to save the classification tree.

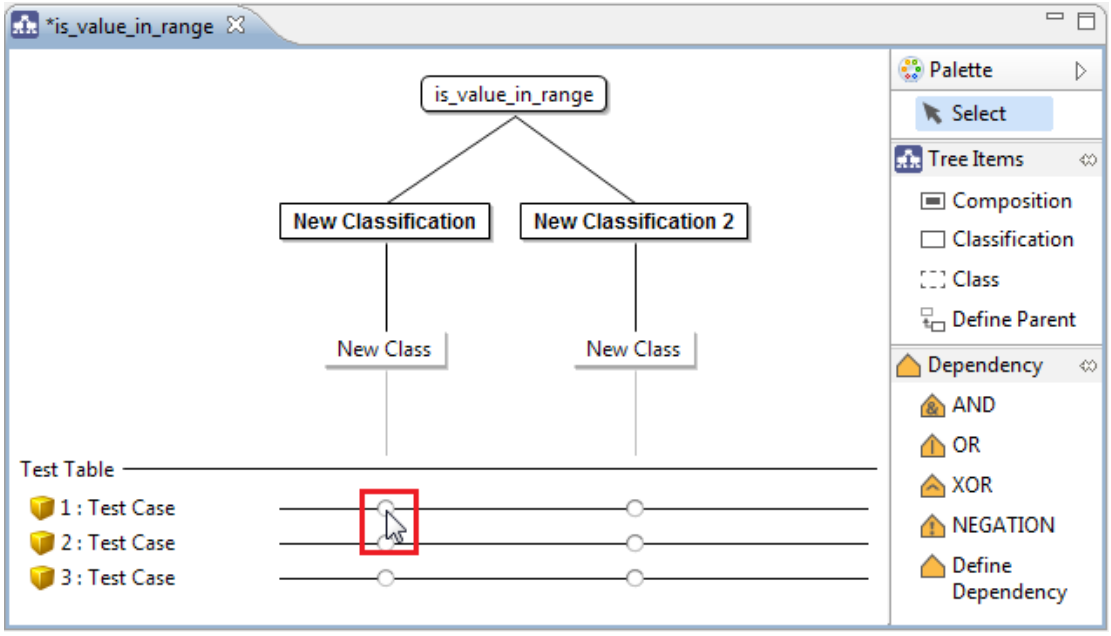


Figure 6.174: Setting marks in the Test Table

If you connect a test case with a class, the respective test data assignments of the class will be assigned to the test case. If you want to review the resulting test data assignments for the whole test case, select the test case within the test item list. The Test Data view will now display the assignments for the test case.

The test data of a test case is displayed read-only because it is defined by the marks set within the combination table and cannot be changed here.

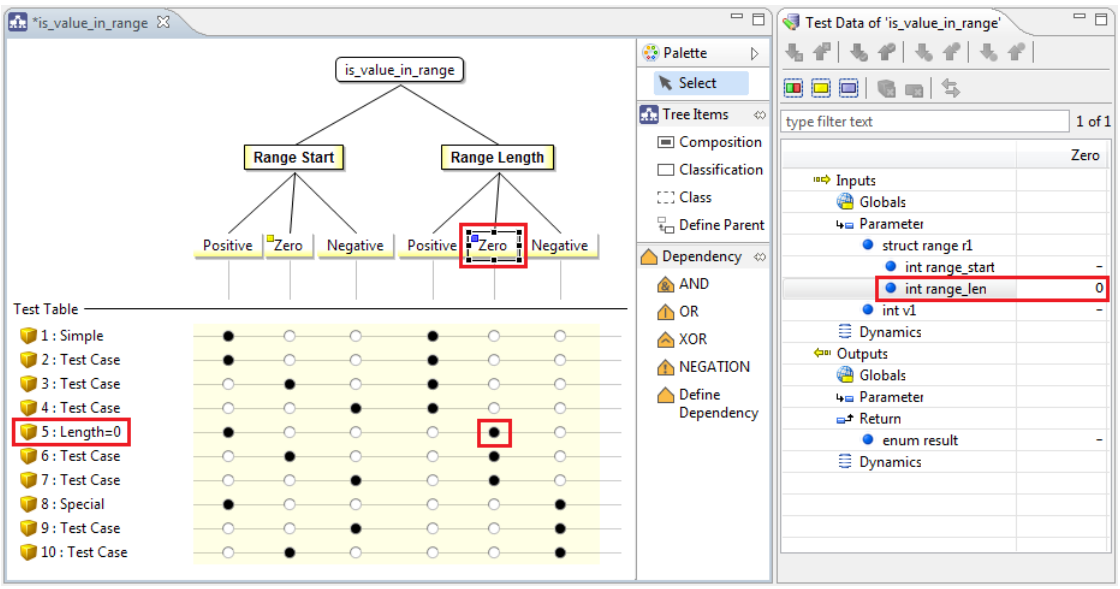


Figure 6.175: Classification Tree with test data for class “Zero”

Notice the following habits:

- All tree items with assigned test data are marked with a yellow square (■), when not selected.
- When selecting a tree item, you will see the test data entered for this item within the Test Data view.
- When selecting any interface element within the Test Data view, all classification tree elements that contain test data for this interface element will be marked with a blue square (■).

→ Switch to the TDE perspective.

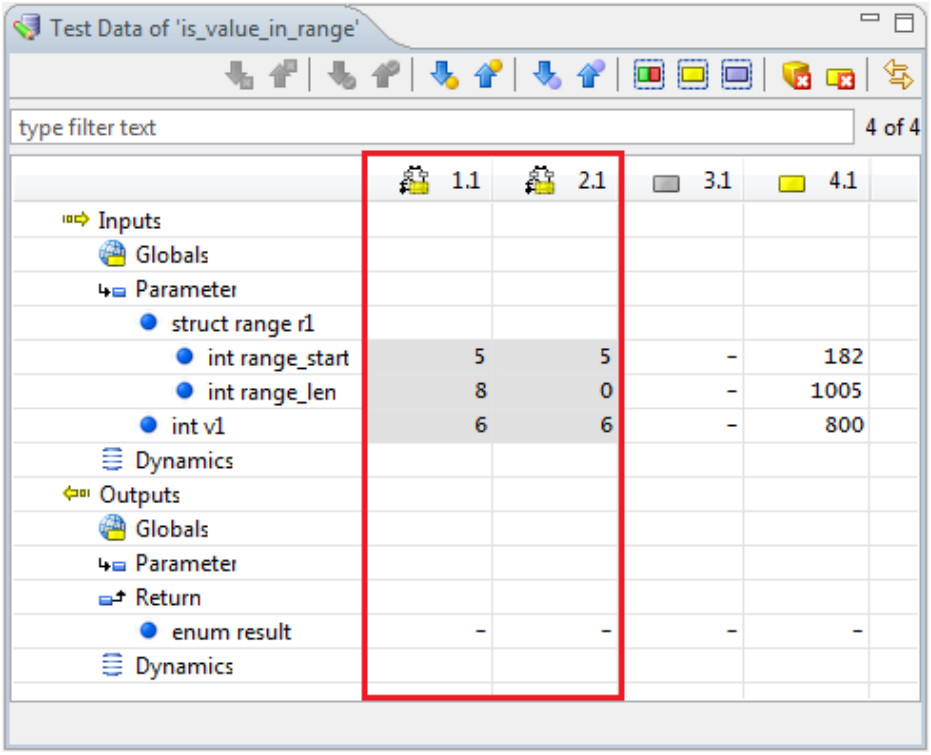


Figure 6.176: Test cases and test steps created within the CTE in the Test Item view of the Overview perspective

Please notice the following habits:

- Test items with values stemming from the CTE perspective are marked with special status indicators: (test case) and (test step).
- Indicators will appear light gray when there are no values entered, dark gray when some values are entered and yellow when the entering of values is completed.
- Values stemming from the CTE are read-only. If you want to change them, switch back to the CTE perspective and do your changes there.
- Test cases created in the Test Item view do not appear in the CTE.

6.8.6.5 Automated tree generation based on function interface

Based on information provided by the test object interface as well as the interface dictionary and the configuration in the TIE it is possible to generate a classification tree. (More information about the interface dictionary is provided in subsection 6.1.6 Interface dictionary.)

The tree always contains an “Inputs” and “Outputs” subtree and when available nodes for parameter, globals, return value, parameter and return values of called functions.

The interface information defined in the TIE is used to generate subtrees based on the different types of the interface items such as enums, arrays, scalars, pointers.

Please refer to the overview in figure 6.177, the blue marked boxes contain descriptions of the generated nodes in the respective position.

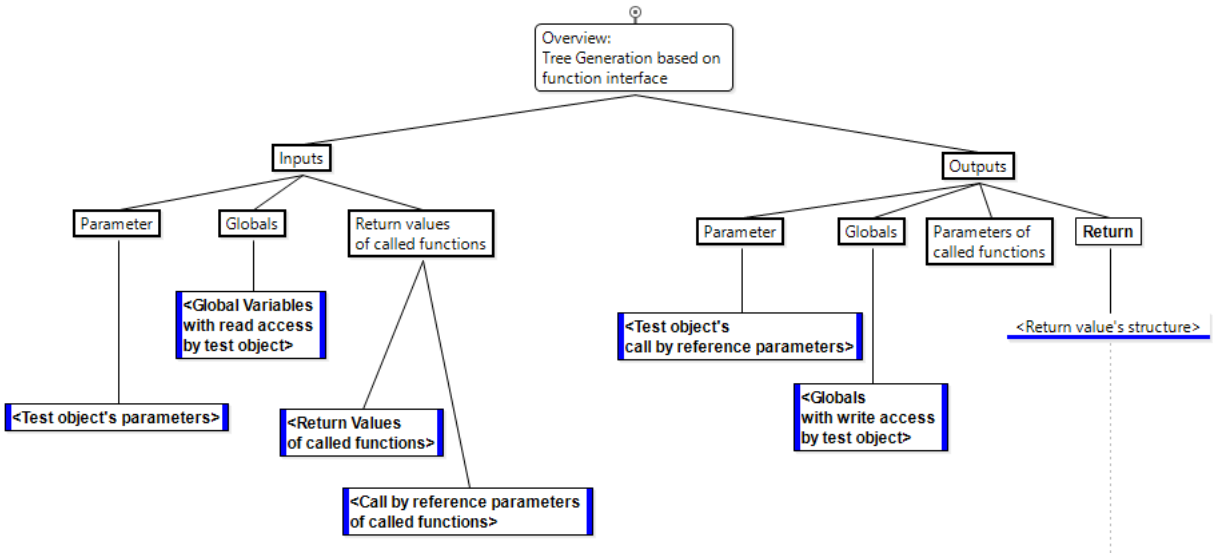


Figure 6.177: An overview on the automated tree generation based on the function interface

To generate a classification tree:

→ Open the CTE perspective.
→ Select a test object.

In general there is no saved CTE file, therefore a new classification tree will be generated automatically and the CTE will be instantly created.

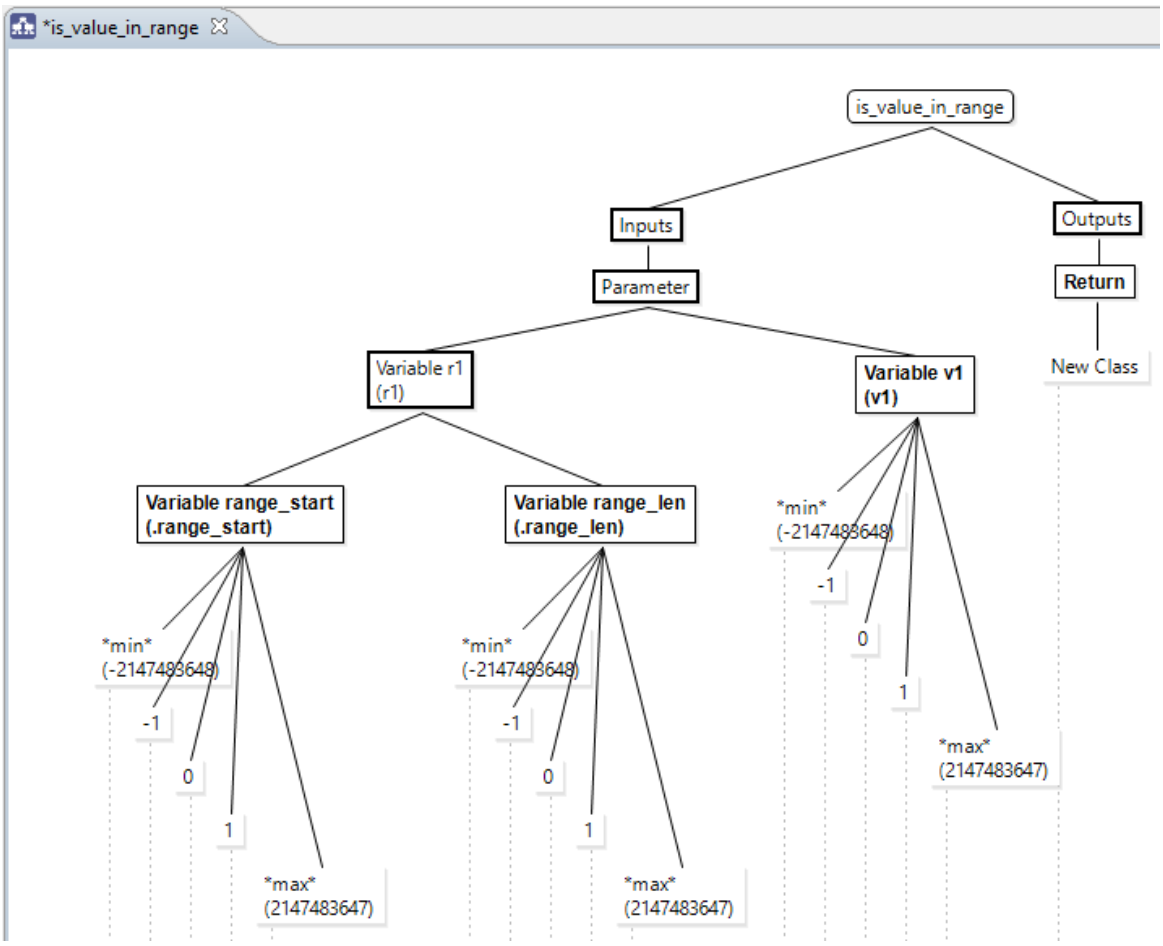


Figure 6.178: Automatically generated for the example is_value_in_range

The editor of the resulting tree is marked with an asterisk to show unsaved content. It is possible to modify the content to your specific needs and changes can finally be saved.

To check whether a CTE file was already saved:

→ Select a test object.
→ Select the Properties View > Attributes.

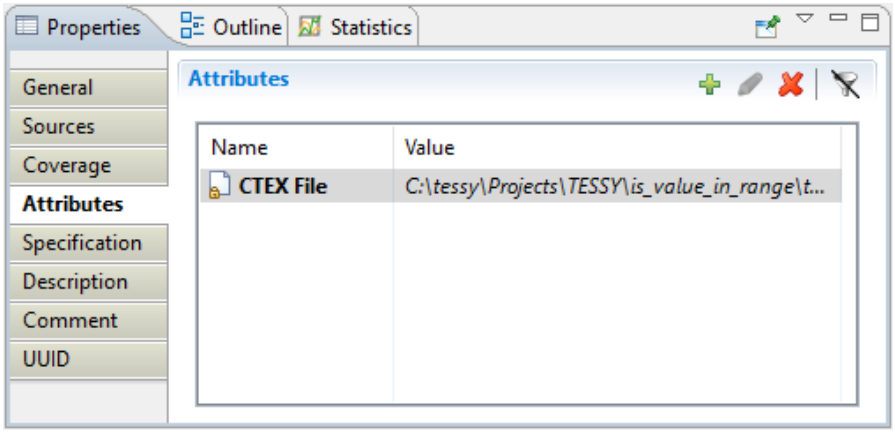


Figure 6.179: The CTEX file attribute

This CTEX file attribute only exists when a CTE was saved.

Important: By removing the CTEX file from the attributes list the respective classification tree will be voided, but associated data will remain on the hard disk.

To be able to create a new tree it is necessary to discard a generated and already saved tree. Therefore you need to remove the test specification:

→ Right-click on the test object.
→ In the pull down menu click “Remove Test Specification” (see figure 6.180).

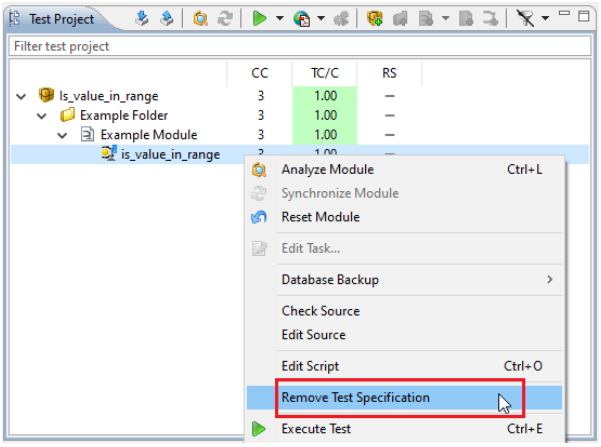


Figure 6.180: Remove Test Specification

The automatic tree generation in TESSY is enabled by default. You can disable it in the Preferences menu.

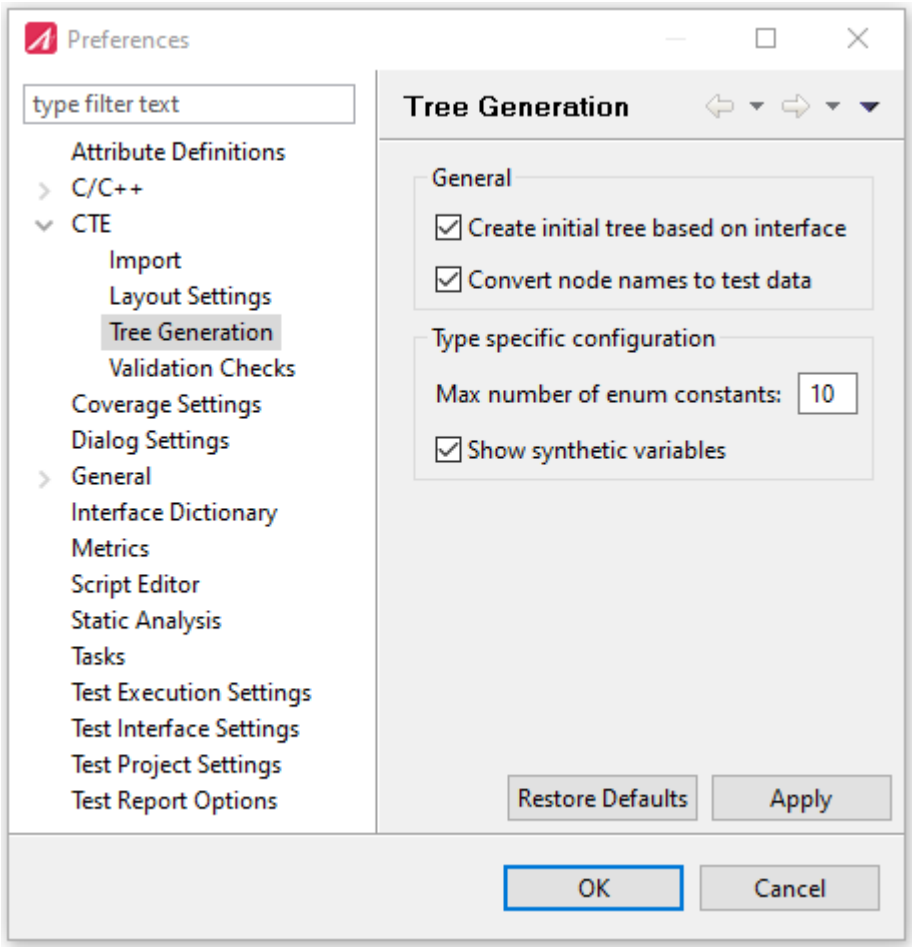


Figure 6.181: The CTE Preferences

Just untick “Create initial tree based on interface”.

Type specific tree generation configurations are also provided. Via “Max number of enum constants” you can define for how many enum constants tree nodes should be generated. This is useful for enums with many constants.

Important: Make sure that “Show synthetic variables” is ticked in order that synthetic variables will be shown and considered in newly generated trees.

This checkbox is checked by default.

6.8.6.6 Automated conversion from class node name to test data

It occurs that in Classification Trees the name of leaf classes create a clear intention for the test data attached to them. As a test engineer you are then confronted with the error prone and tedious job to attach the right test data to such nodes.

The automated conversion of test data is about CTE classes like the children below the classification “Variable range_start” in the following image (see 6.182). The names of the classes can be associated uniquely with test data for the parameter “range_start”.

To associate test data manually you have to first select each class, in this case “*min*”, “-1”, “0”, “1”, “5” and “*max*”. You need to find the parameter “range_start” in the Test Data view on the right and add the corresponding value to each class separately.

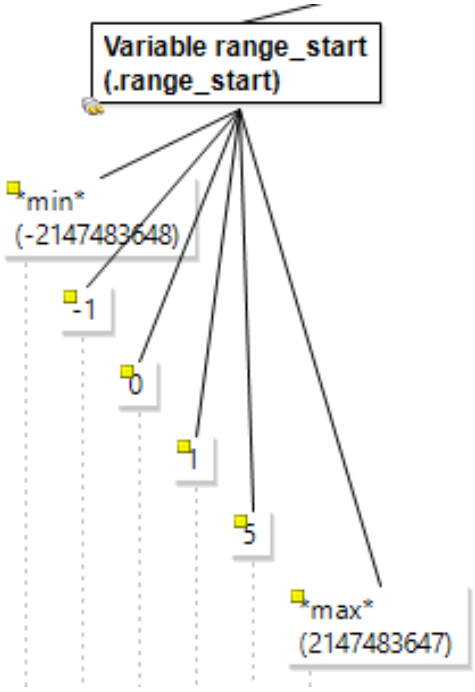


Figure 6.182: CTE class node with children associated with test data

TESSY enables you to create this test data automatically, it just needs to be informed which interface element in the Test Data view is associated with the corresponding CTE classification in the tree area. This association is done by “Attach to CTE Node”.

If your initial classification tree is generated by TESSY, the interface will be attached to the corresponding nodes automatically and TESSY will automatically parse the test data of leaf classes of this classification from the class’ name.

Please keep in mind: This automated conversion can also be deactivated (see below).

If your tree is created manually, you can associate an interface object manually.

→ Select the node to be attached in the CTE.

→ In the Test Data View on the right select the interface element you want to attach to the selected node.

→ In the context menu click “Attach to CTE Node” to associate the interface object with the selected node in the active CTE (see figure 6.183).

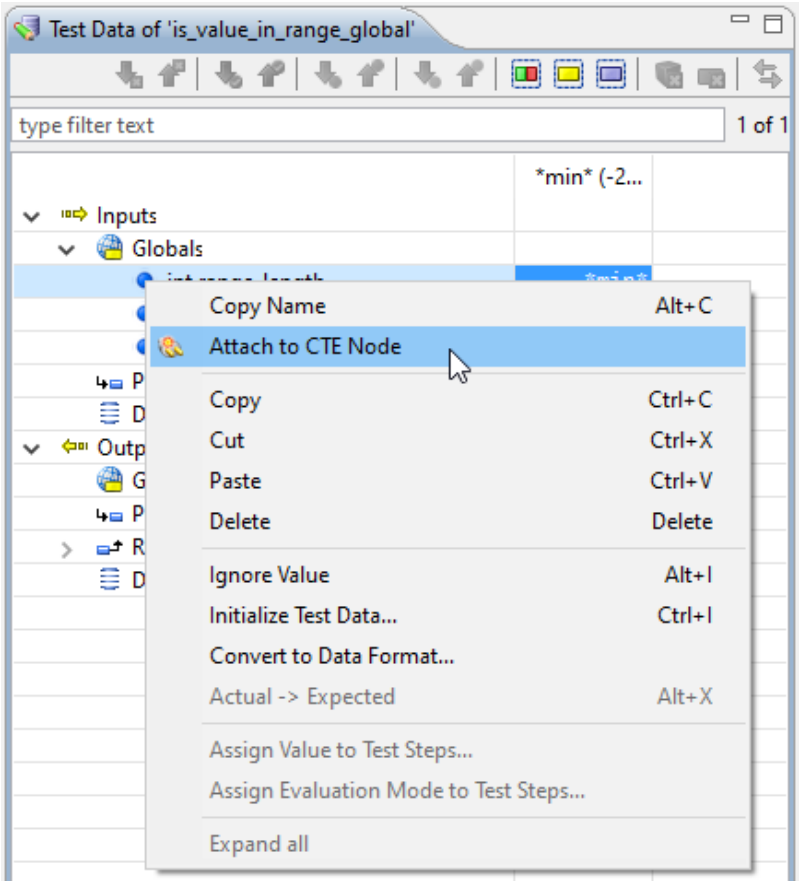


Figure 6.183: Attach the selected interface object

The Test Data value will be derived from the class node’s name. All values and expressions that can be entered within TDE for the respective interface object can be used as class name as well.

For more information about entering values and expressions refer to section 6.9.7.4 Entering values.

In particular it is possible to use special values, such as:
”*min*(-100)”, ”*max* (-100)”, ”*max-1*”, ”*none*”, ”89 // This is a special Value”, ”/* another comment */”, ”A_DEFINE”

Important: For performance reasons this derivation is only done when classification trees are saved.

Pointers must have set a dynamic object as their targets in order that TESSY will create test data. For other targets TESSY will assume that the test data is derived from other parts of the classification tree.

For array elements the index is determined from the last line of the nodes' names which are associated with the array element type. I.e. the last line of such a node must match the regular expression: `.*("["[0-9]+"]")+`

Examples for arrays are:

```
array_1[3]
array_2[5][17]
```

The derivation of the index from the name has the advantage that changes of the name will be recognized by TESSY.

Important: Be aware that as of now test data will not be automatically removed.

Deactivating the automated conversion

If you do not want TESSY to derive test data from a class node's name, there are two options:

- Deactivate the automated conversion for your whole TESSY project in the menu bar by unchecking “Window” > “Preferences” > “CTE” > “Tree Generation” > “Convert node names to test data”.

- Detach the interface from its classification node in the context menu of the respective node in the CTE by clicking the menu entry “Detach Interface from CTE Node” (see figure 6.184).

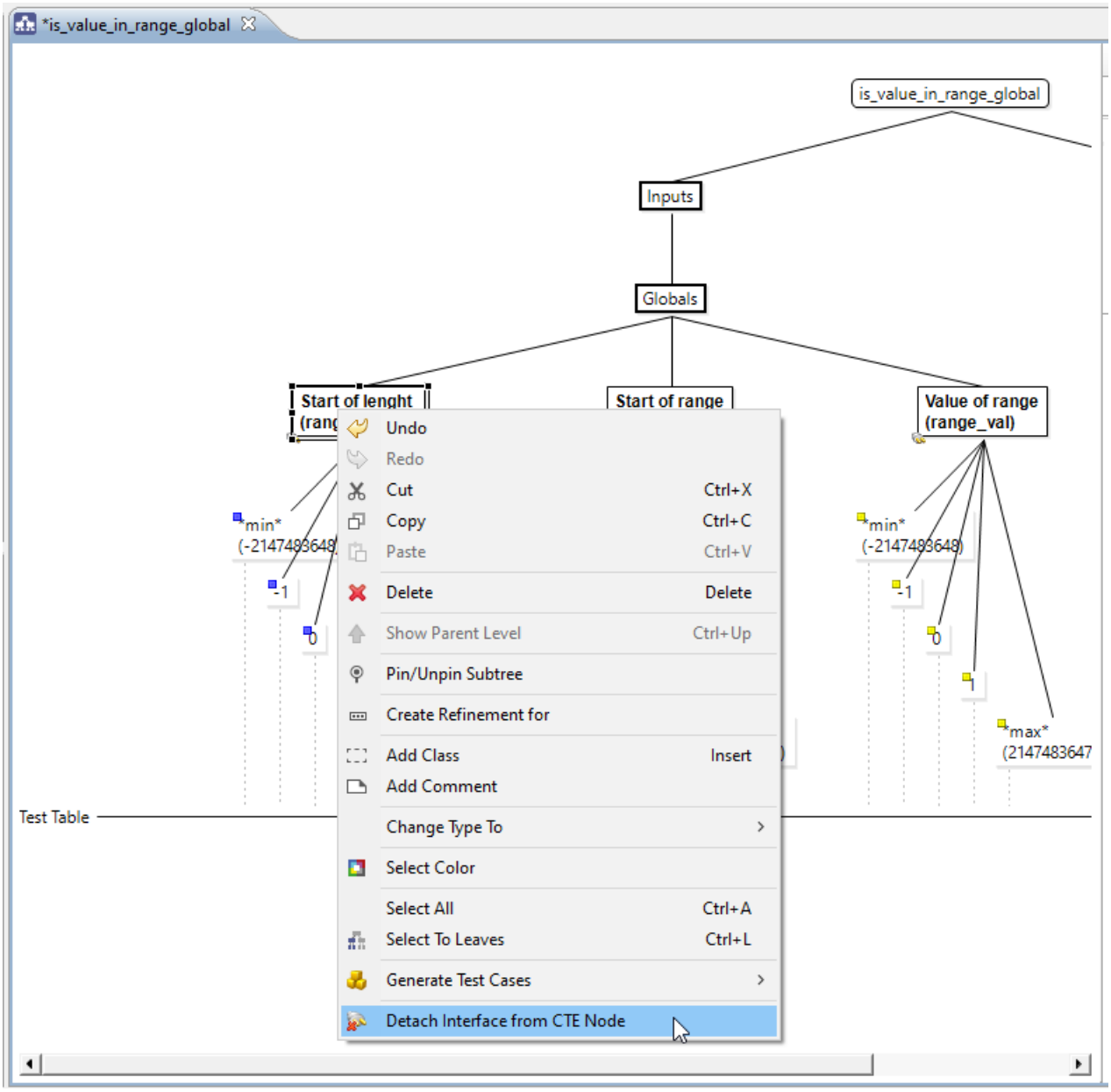


Figure 6.184: Detach the interface from an CTE note

6.8.7 Test Data view

Whether using the CTE or creating the test cases manually within the TDE perspective: Values are always entered in the Test Data view.

6.9.7 Test Data view

Some of the operations and overviews are only possible within the TDE perspective, so switch to chapter 6.9.7 Test Data view to learn how to use the Test Data view.

6.8.7.1 Assigning test data to the CTE

Instead of assigning test data directly to all variables of the test object interface for each test case, you can assign them using the tree nodes of the classification tree. For each tree node you can assign values to variables.

Assigning values to variables

Child nodes inherit the values from their parent nodes, but you can as well overwrite inherited variable values for a child tree node.

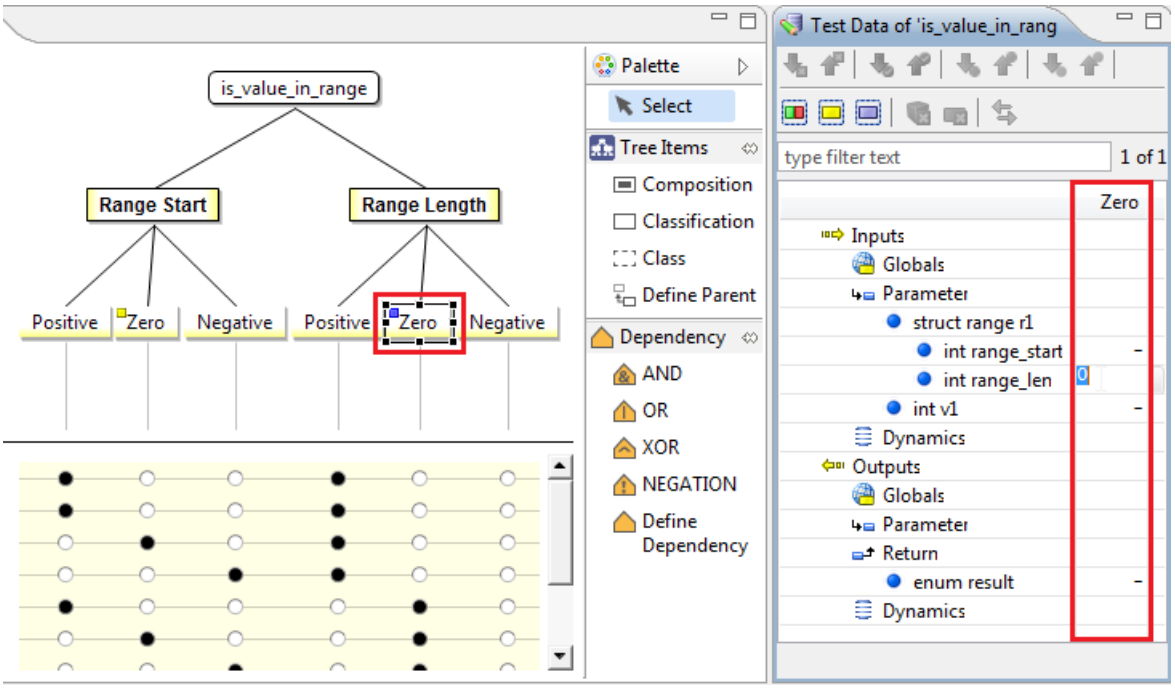


Figure 6.185: Showing data of a tree node

When combining leaf classes of the classification tree to test cases, the variable assignments of the marked tree nodes will be assigned to the respective test case. In this way, you can assign all test data within the classification tree and get your test cases automatically filled by setting marks within the combination table.

To assign test data to a variable of the test object interface:

→ Select a tree node within the CTE tree.
The Test Data view on the right will show the test object interface with the value assignments for this tree node as well as inherited values of parent nodes of the tree node.
→ Double-click in the value cell and enter the value.
→ Click on to save the entries.

When selecting a test case within the test item list you will see the resulting test data assignments according to the marks of the test case within the Test Data view.

6.8.7.2 Handling variants: Managing test data in variants

Assigning test data to variants needs particular attention. You need to be aware of the differences between test data on the parent module and test data of the variant and where exactly they are located.

TESSY offers two ways of handling test data in variants.

In general it is possible to edit test data directly within the CTE nodes.

If you do so, you need to make sure that the nodes name is not contradicting the test data. Contradictions between node names and test data lead to contradictions in the test specifications and the related test data. This, of course, will also effect your test reports.

Best practice: Assigning data in variants

Important: Classes should be named after concepts or ideas like “Highest Value” or “Highest Value +1” instead of e.g. “10”. The easiest way is to simply follow the naming of the TDE expressions which are “*max*” and “*max-1*”.

If you do not want to or for some reason cannot use such abstract class names, it is still possible to proceed as follows:

→ Assign all test data which should be equal in all variants in the CTE and organize your test cases in the CTE as usual.
→ Leave all interface variables unassigned in the CTE which will vary in the variants.
→ Save your entries and switch to the TDE perspective.
→ Add the missing values within the TDE.
→ You can later create or synchronize variants.

→ In the TDE perspective you can now enter missing or varied values in the newly created or synchronized variants.

Important: This, of course, is only possible if the respective value in the variant is not stemming from the CTE. Test data added by the CTE is indicated by a gray background within the TDE.

6.8.7.3 Test data assignment precedence rules

When assigning test data to tree nodes of the classification tree, the same variable can be assigned within different locations of the tree and each assignment can have another value for the variable. The resulting value for such a variable (for a given test case) depends on the classes being marked for a test case.

When calculating the variable assignments for a test case, CTE collects all marked tree branches where the variable is assigned. A tree path is defined as the list of tree nodes up to the root starting at the tree node where the variable is assigned. The tree paths are sorted by position of their leaf nodes: The sort order is from left to right.

The example below (see figure 6.186) shows different assignments of variable “x” within a classification tree. The resulting value for “x” is indicated for each test case.

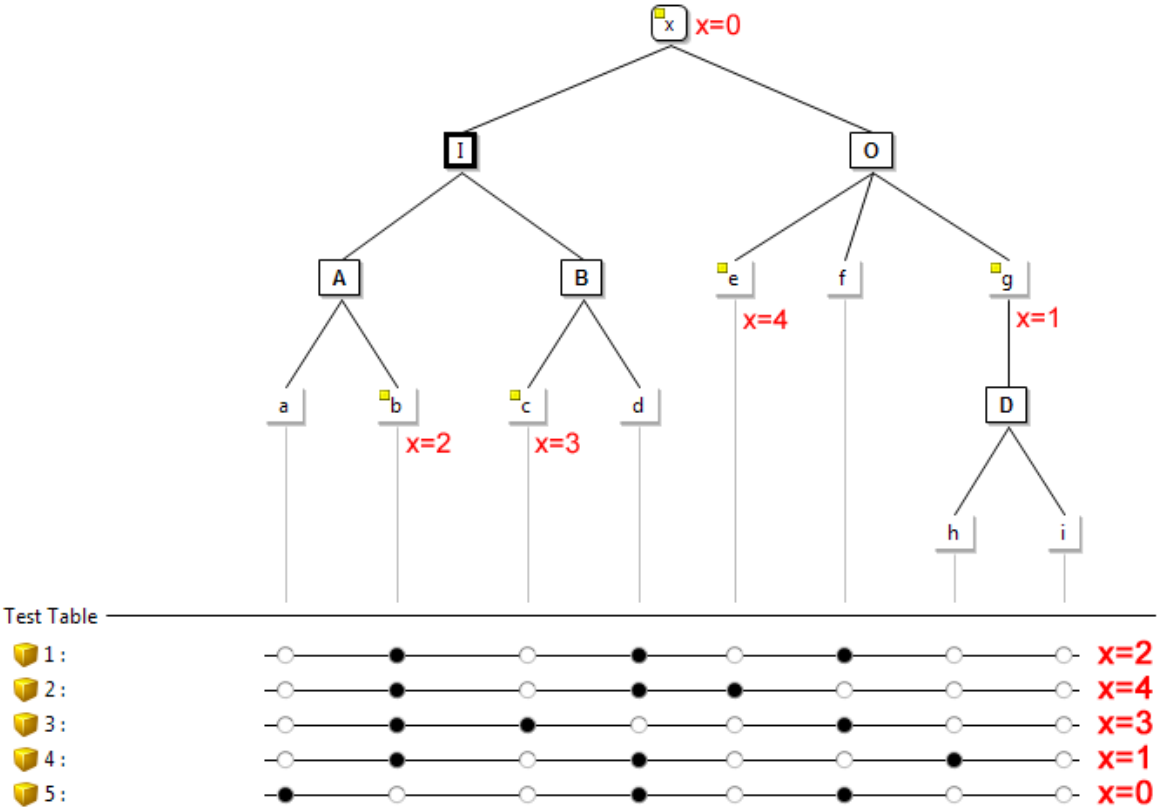


Figure 6.186: Variable assignments in classification trees

The resulting value for the test cases will be calculated like follows:

1. For the first test case the variable is assigned in class “b” which is a longer path than the assignment within the root, so the value of class “b” will be taken.
2. For the second test case we have values within class “b” and class “e”. The tree paths diverge below the root node and the classification “O” is on the right side so that the value of class “e” will be taken.
3. In the third test case there are values within the root node and within classes “b” and “c”. Both tree paths of the classes are longer than the root path and the classification “B” is on the right side so that the value of class “c” will be taken.
4. In the fourth test case we have the tree paths of classes “b” and “g” that diverge at the root. Because classification “O” is in the right side, the value of class “g” will be taken.
5. In this test case all marked classes refer to the value defined within the root node so that the value of the root node will be taken.

6.8.7.4 Attach selected interface element to CTE node

It is possible to define a connection between the selected interface element in Test Data view and the CTE node currently selected in the CTE.

Therefore you need to:

→ Select tree node element in the tree area (see figure 6.187).
→ Select the corresponding interface element in the Test Data view on the right.
→ Right-click in respective interface element in the Test Data view to open the context menu.
→ Click “Attach to CTE Node” in the context menu.

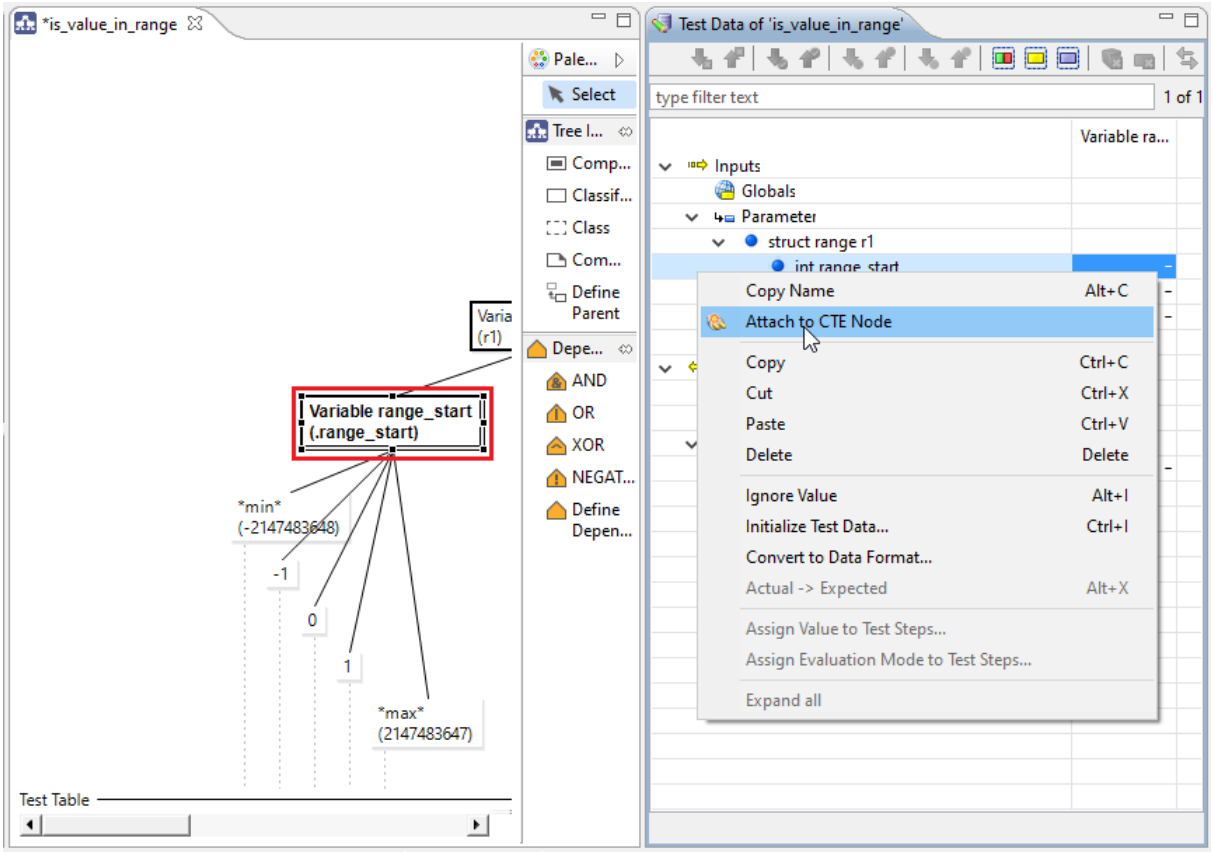


Figure 6.187: Attaching an interface element to a tree node

The connection is visualized by a small TIE icon in the bottom left corner of the respective CTE node:

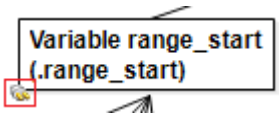


Figure 6.188: TIE icon of a CTE node

This connection enables TESSY:

- To perform an automated conversion from class node name to test data (see 6.8.6.6 Automated conversion from class node name to test data).
- To ensure a unique descriptive name for CTE nodes attached to variables which have a detailed description in the interface dictionary (see 6.1.6 Interface dictionary).

6.8.8 Dependencies in CTE

When modeling a classification tree there are situations where the tree contains classes which must or must not be marked simultaneously. Dependencies express such relations and help to force particular combinations of marks in test cases and steps.

Defining dependencies using the Palette view

A dependency is primarily a logical expression. It can only be defined for classes which are not further specialized, i.e. which do not have any child nodes.

Important: Please note that a dependency only implies an action. If the logical expression is false, no action will be triggered.

6.8.8.1 Dependencies related icons in the Palette view

Icon	Action / Shortcut / Comment
	Creates an AND dependency.
	Creates an OR dependency.
	Creates an XOR dependency.
	Creates a NEGATION dependency.
	Defines a relation between the dependencies or classes.

Table 6.65: Dependencies related icons in the Palette

6.8.8.2 Defining dependencies

The classification tree of “Value in Range” the classes “range_length -> negative” implies that the position of the value can only be outside of the interval. Thus means a test case marked “range_length -> negative” must also mark “position -> outside”.

More information about the TESSY example “Value in Range” is provided in section 5.1 Quickstart 1: Unit test exercise is_value_in_range

Due to the given situation a dependency needs to be defined which expresses this relation in the “Value in Range” example:

→ Open “Value in Range”.

→ Select (Define Dependency) in the CTE Palette.

→ Select the class “range_length -> negative” first and than the class “position -> outside”.

→ Click to lay out the tree.

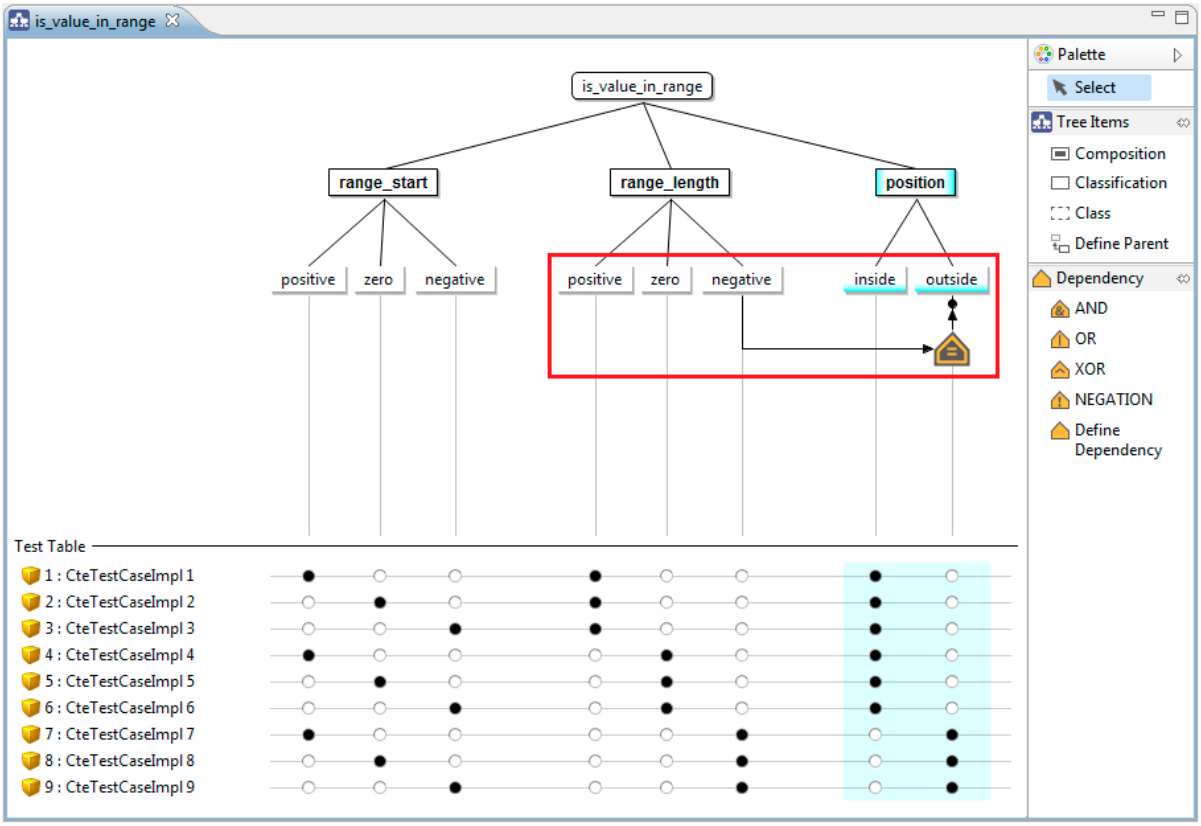


Figure 6.189: Dependency defined between “range_length -> negative” and “position -> outside”

With this a dependency between “range_length -> negative” and “position -> outside” was defined. Therefore creating a mark for “range_length -> negative” will also create a mark for “position -> outside” and as long as “range_length -> negative” has selected mark in the same test case a mark for “position -> outside” will be present.

According to the result of this logical expression a “mark action” of the class connected with this dependency is automatically triggered. An action can be selected at a class with a dependency as input.

There are two actions: “activate mark” or “deactivate mark”. In the diagram the selected action is represented by a black circle (activate mark) or white circle (deactivate mark) between the incoming arrow and the class. The Properties View of “position -> outside” will show a textual representation of the dependency behind the (dependency icon).

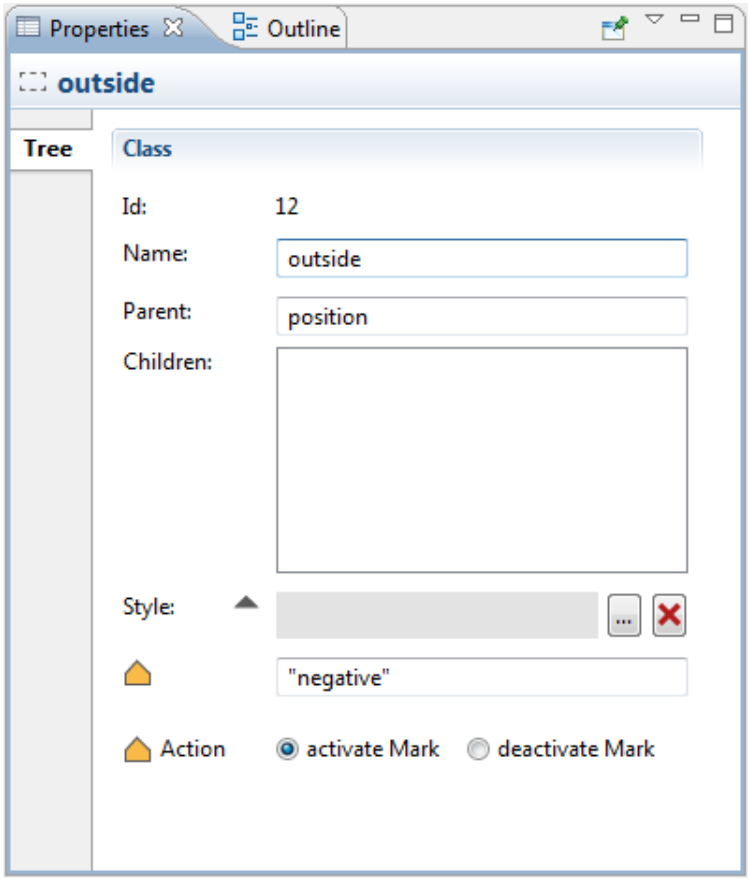


Figure 6.190: Dependencies in the Properties View

In the picture below (see figure 6.191) it is a simply “negative” triggered action which is the default “activate Mark”. In “Value in Range” you can use the deactivate mark action in the following way:

- → Create a dependency as explained above between the class “range_length -> zero” and the class “position -> inside”.
- → Select “position -> inside”.
- → Select “deactivate Mark”.

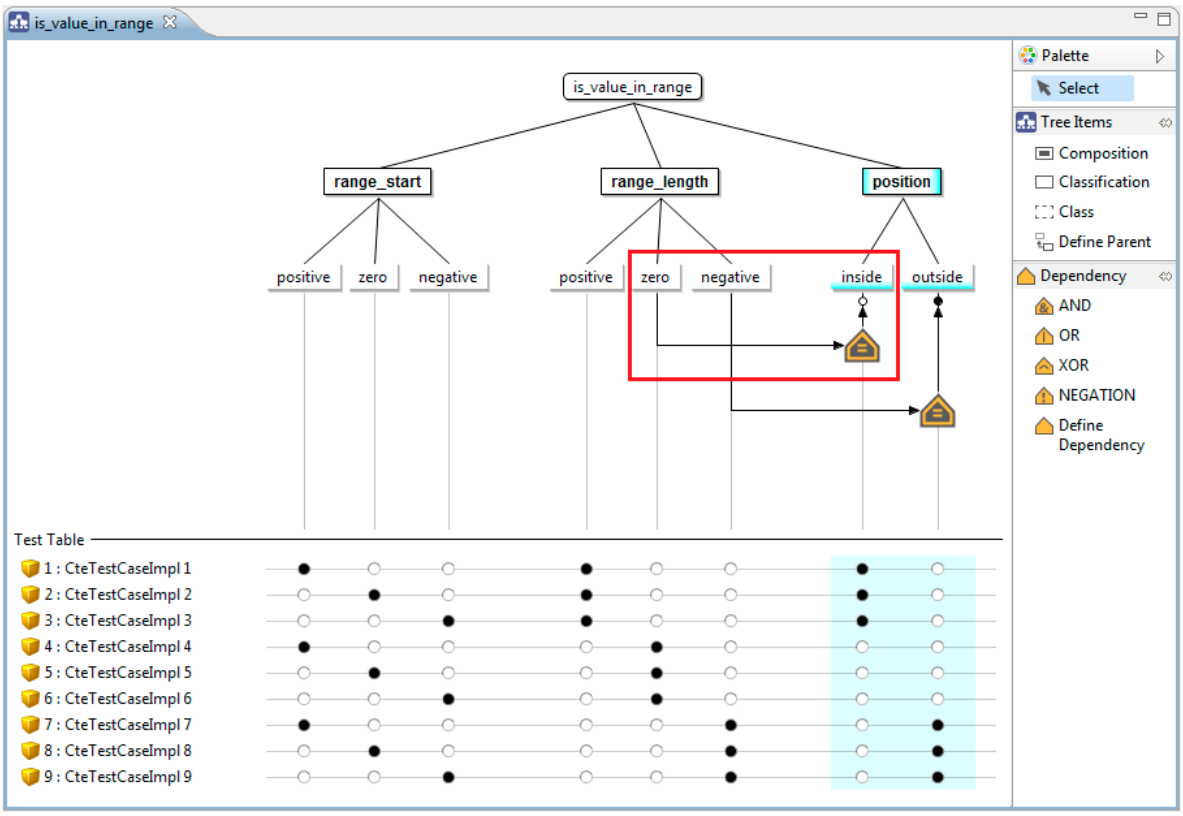


Figure 6.191: Dependency defined between “range_length -> zero” and “position -> inside”

Now a mark for “position -> inside” will be removed if a mark for “range_length -> zero” is created and it is not possible to create a mark for “position -> inside” as long “range_length -> zero” is selected.

Important: Please remark that a dependency only implies an action. If the logical expression is false, no action will be triggered. Moreover it is only possible to set marks if there are not any dependencies which exclude the mark directly or indirectly by activating another mark.

6.8.8.3 Composite dependencies

Note the following restriction in defining dependencies:

- A class can only be connected with exactly one dependency. Multiple dependencies however can be resolved by the user with the help of operators to one composite dependency.

In “Value in Range” such a situation exists, so it is reasonable to mark “position -> outside” if either “range_length -> zero” or “range_length -> negative” is selected.

Defining a composite dependency

The situation can be solved by defining a composite dependency:

- → Select (OR dependency) in the Palette and click on an empty spot below the class “position-> outside”.
- → Select (Define Dependency) from the Palette and click “position -> outside”. After that is done the dependency for “outside” is the OR operation and the former ASSIGNMENT dependency is not connected to outside anymore.
- → Select the now disconnected (ASSIGNMENT dependency) and than the (OR dependency).
- → Select the class “range_length -> zero” and the (OR dependency).

(See figure 6.192)

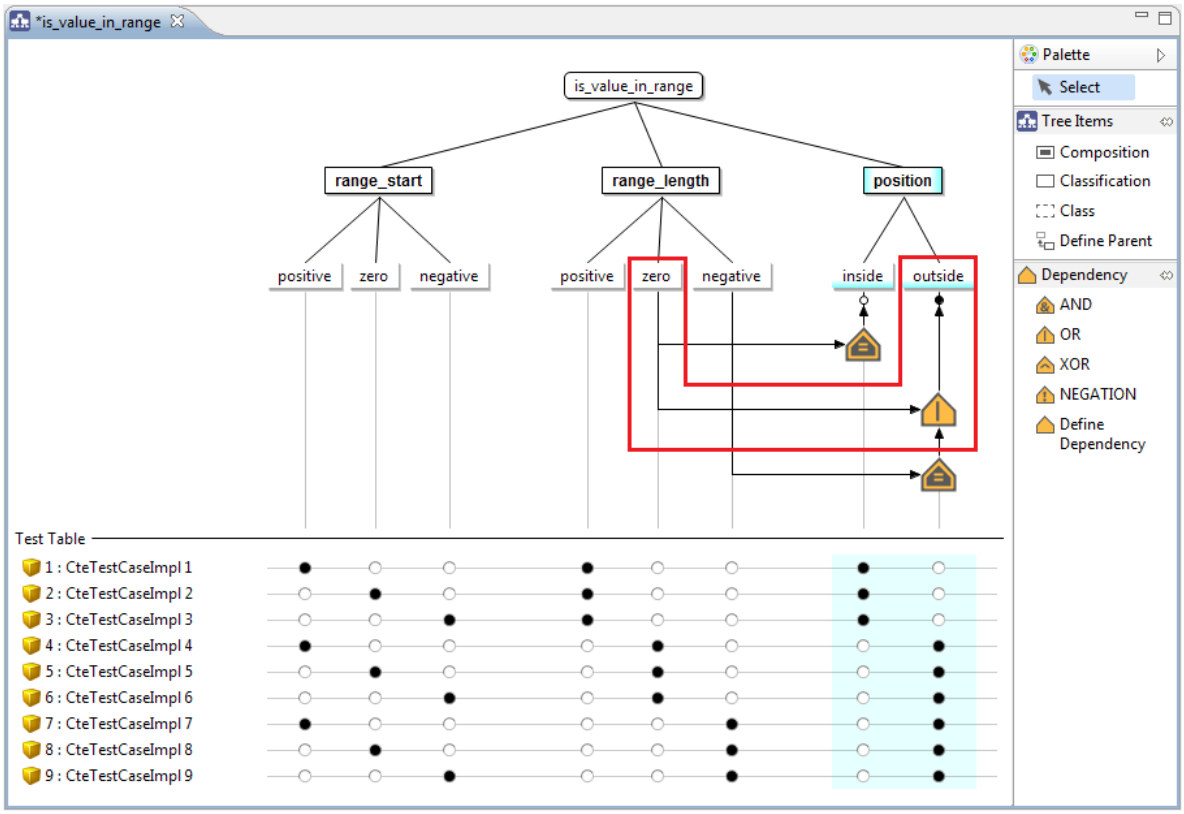


Figure 6.192: Composite dependency

Now both dependencies are modeled in one composite dependency and in the Properties View of “position -> outside” the formula “negative” || “zero” is shown.

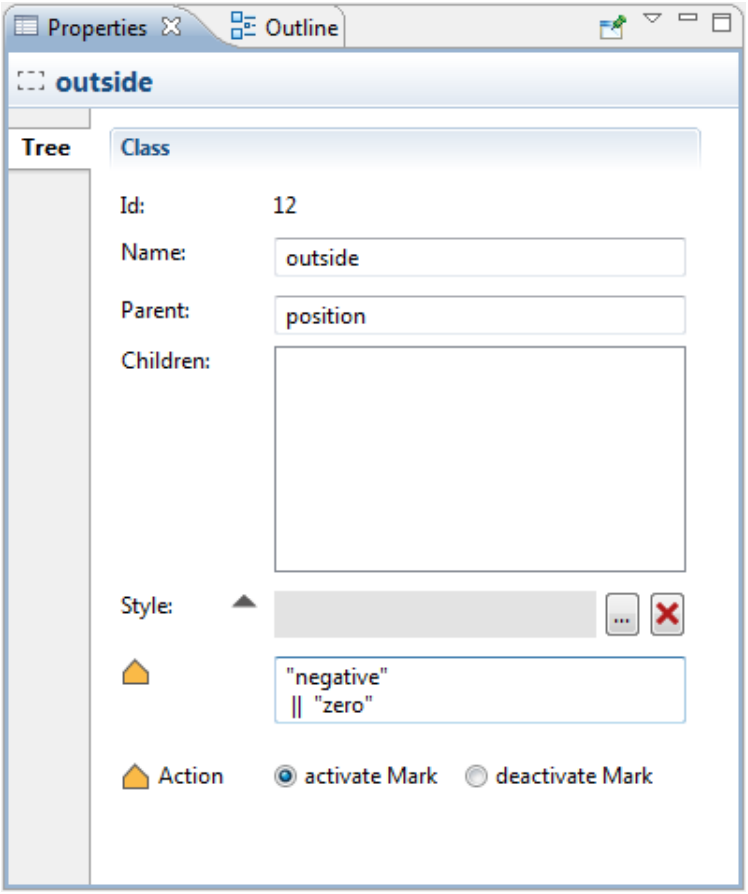


Figure 6.193: Composite dependency in the Properties View

6.9 TDE: Entering test data

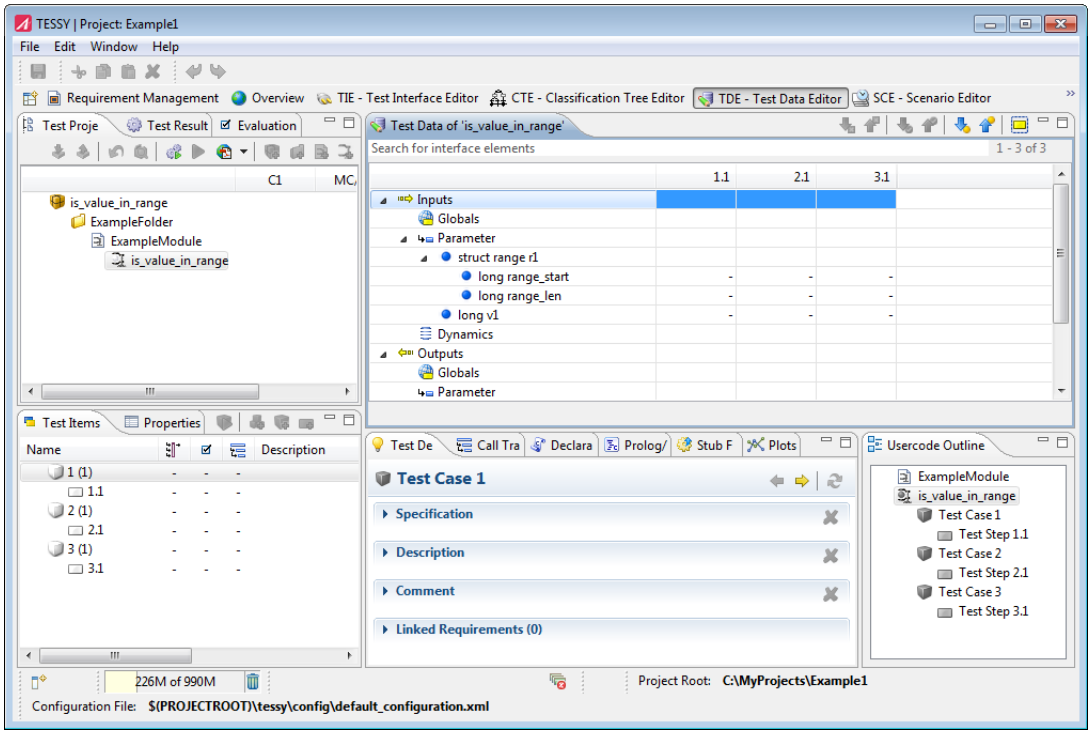


Figure 6.194: TDE perspective

6.9.1 Structure of the TDE perspective

Pane	**Location** (default)	**Function**
Test Project view	upper left	Same view as within the Overview perspective.
Test Results view	upper left	Same view as within the Overview perspective.
Evaluation Macros view	upper left	Same view as within the Overview perspective.
Test Items view	lower left	Same view as within the Overview perspective.
Properties view	lower left	Same view as within the Overview perspective.

continue next page

Pane	**Location** (default)	**Function**
Test Data view	upper right	To enter test data and expected values, after the test execution, reviewing passed, failed and undefined values.
Test Definition view	lower center	To display the test case specification, the optional description and linked requirements of the current test case.
Call Trace view	lower center	To evaluate expected calls of functions for each test step of a test object.
Declaration view	lower center	Declaration and definition for usercode e.g. for freely definable and usable variables and functions.
Prolog/Epilog view	lower center	To specify usercode that will be executed at a certain point in time during the test execution.
Stub Functions view	lower center	To display the code for all stub functions.
Plots view	lower center	To visualize graphically the test results.
Usercode Outline view	lower right	To display the usercode that will be executed at a certain point in time during the test execution.
Plot Definitions view	lower right	To create and configure plots (same view as within the TDE perspective).

Table 6.66: Structure of TDE

Usercode

TESSY provides an interface to specify the usercode that will be executed at a certain point in time during the test execution. Using the usercode views (i.e. Prolog/Epilog view and Usercode Outline view) you can specify such C code fragments or emulator scripts depending on the selected target configuration.

6.9.2 Test Project view

The Test Project view displays your test project which you organized within the Overview perspective.

6.2.2 Test Project view

Important: We recommend to do any changes of the test project structure within the Test Project view of the Overview perspective. The view layout of this perspective is optimized for this purpose!

6.9.3 Test Results view

The Test Results view displays the coverage measurement results and the results of a test run of expected outputs, evaluation macros and call traces, if applicable. It is the same view as within the Overview perspective.

6.2.6 Test Results view

6.9.4 Evaluation Macros view

This view lists the detailed results of the evaluation macros if the usercode of the test object contains any evaluation macros. The results are displayed wherever they occur within the usercode, e.g. within stub functions or test step epilogs. It is the same view as within the Overview perspective.

6.2.7 Evaluation Macros view

6.9.5 Test Items view

Within the Test Items view you get an overview about your test cases and test steps which you organized within the Overview perspective or the CTE (see section 6.8 CTE: Designing the test cases).

6.2.5 Test Items view

To create test cases and test steps manually without using the Classification Tree Editor, switch to the Test Items view within the Overview perspective.

6.9.6 Properties view

6.2.3 Properties view

The Properties view displays all the properties which you organized within the Overview perspective. Most operations are possible.
For changing a source switch to the Properties view within the Overview perspective.

Type information of a variable

The view is context sensitive: You can view the passing direction and all type information of the variable (i.e. the basic type, the size as well as any modifiers and pragmas) if you select the variable within the Test Data view (see figure 6.195).

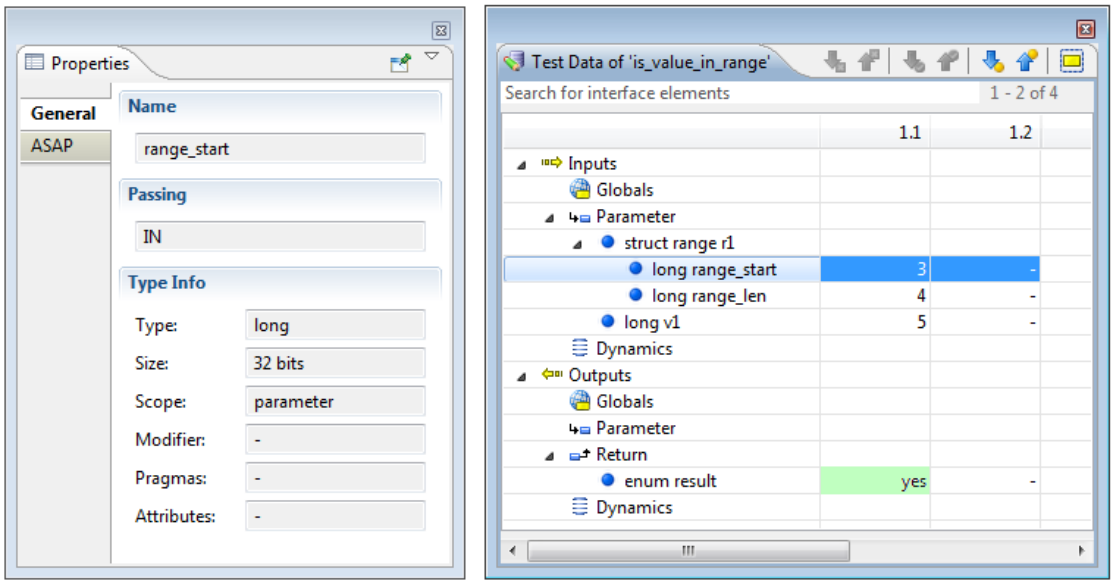


Figure 6.195: Type information of the variable long range_start

6.9.7 Test Data view

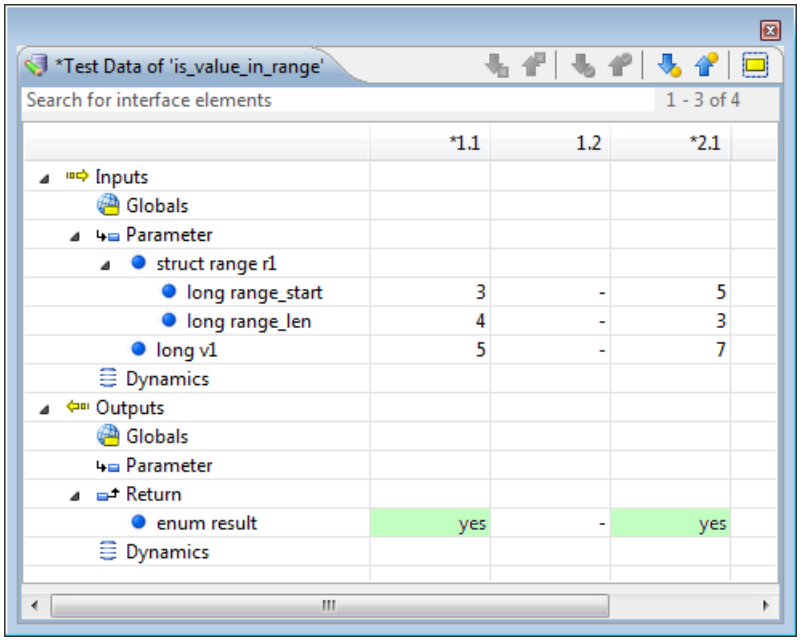


Figure 6.196: Test Data view

Whether using the CTE or creating the test cases manually within the TDE perspective, you will use the Test Data view to enter or review the input values and expected results of all test cases and test steps.

Important: CTE exported values are read-only within the TDE perspective. The cells are insensitive. Switch to the CTE perspective to change such values if necessary (respectively the underlying document).

6.9.7.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Highlights the next failed value.
	Highlights the previous failed value.
	Highlights the next passed value.

continue next page

Icon	Action / Shortcut / Comment
	Highlights the previous passed value.
	Highlights the next undefined value.
	Highlights the previous undefined value.
	Highlights the next modified value.
	Highlights the previous modified value.
	Shows all actual values.
	Highlights all undefined values.
	Highlights all modified values.
	Shows only failed test cases.
	Shows only failed test steps.
	Links with the Test Item view.

Table 6.67: Tool bar icons of the Test Data view

6.9.7.2 Status indicators

The following table shows the indicators of status and their meaning which are used by the Test Data view.

Indicator	Status / Meaning
	Test step passed: The actual result did match the expected results.
	Test step failed: The actual result did **not** match the expected results.
	Test step generated: The test step was generated by the test case generator but has no executable data yet.
	Test step generated with data: The test step was generated by the test case generator and executable test steps were generated.

Table 6.68: Status indicators of the Test Data view

6.9.7.3 Viewing the interface of the test object

The Test Data view displays the interface of the test object. On the left side of the Test Data view you see the following interface elements and icons:

Important: Interface elements with the passing direction “Irrelevant” or “Extern” will NOT be displayed within the TDE!

Element / Icon	Meaning
Inputs and Outputs	Input values are all interface elements that are read by the test object. Output values are written by the test object respectively are the expected results. Within the TIE you determine which values are Inputs and which are Outputs. TESSY tries to find out the default passing directions (input or output) automatically when analyzing the source files.
Globals	Globals are the global variables referenced by your test object. Global variables can contain global static variables and static local variables that are defined within functions.
Parameter	Parameters of the functions of the test object.
Dynamics	Pointer targets, referenced through a pointer of the test object interface.
Return	Return variables.
(white arrow)	The arrow is displayed when an element has child elements. Click on the arrow to expand. If you want to expand all child elements, use the context menu (“Expand all”).
(blue dot)	Function

Table 6.69: Interface elements and icons of the Test Data view

Every variable will be assigned to one of the interface elements described above, e.g. Parameter, Global etc. Initially, the Dynamics section will always be empty.

The columns on the right represent the test steps where the values of the variables are defined.

Please notice the following habits of this view:

- Select a column by clicking on the number of the test step. The selected column is marked in blue (compare figure 6.197).
- Move the mouse pointer over the number of the test step to see the name of the test step within a tool tip (compare figure 6.197).
- Select all values for a variable by clicking on the variable in the left column.
- If you select the icon “Highlight Undefined Values” in the tool bar, all variables that do not contain any data are marked in yellow (compare figure 6.197).

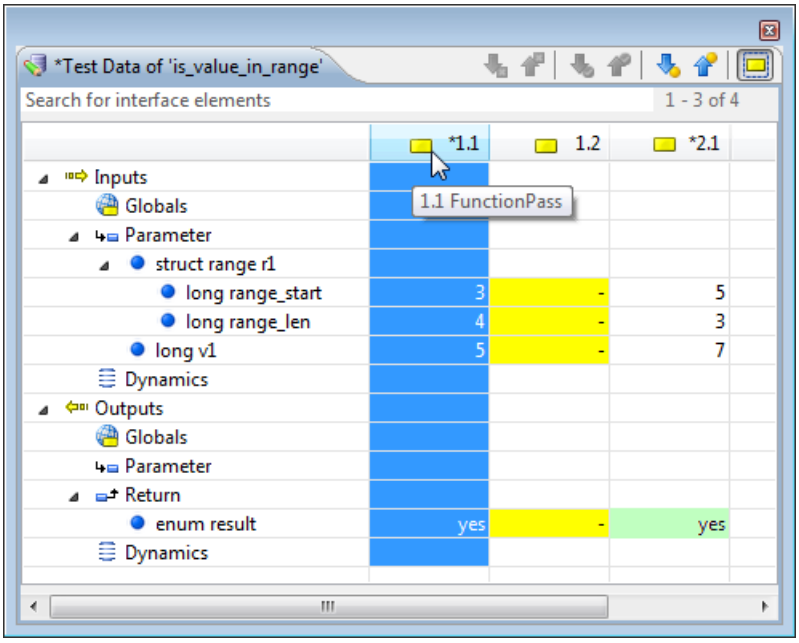


Figure 6.197: Test step 1.1 is selected and undefined values are highlighted in yellow.

To choose the test steps you want to see in the Test Data view you need to select them in the Test Item view (Ctrl + click) first. Make sure that (Link with the Test Item View) is enabled in the Test Data view tool bar. After that only the selected test steps will be displayed (see figure 6.198).

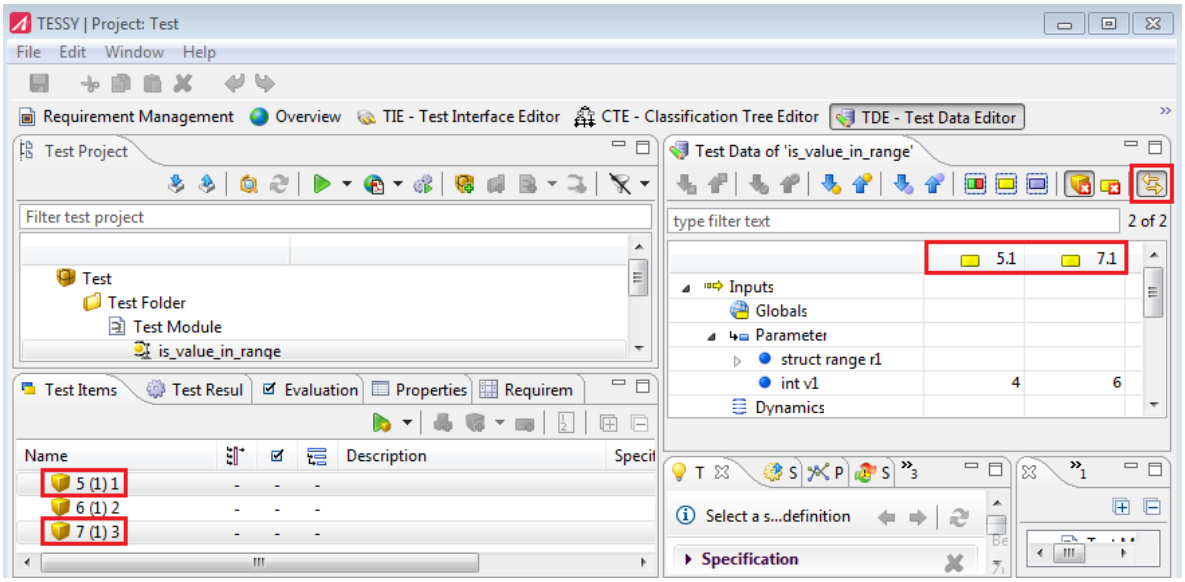


Figure 6.198: Test Data view showing selected test steps.

6.9.7.4 Entering values

Entering values

Values for interface elements are entered into the cells of the Test Data view. The values will be checked and/or converted when leaving the cell or when changing to any neighboring cell.

The TDE provides undo/redo functionality for all changes within the Test Data view!

Validation of test data values

By default, all imported or manually entered test data values are checked for syntactical correctness, truncated to the type size and optionally formatted. The truncation of values depends on which kind of number format was used:

- Decimal numbers will be checked against the minimum and maximum value of the respective data type. When entering -10 for an unsigned type you will see a value of 0 as test data. If the value is less than the minimum, the minimum value will be used, if it is more than the maximum, the maximum value will be used.
- Hexadecimal and binary numbers will be truncated to the number of bytes available for the respective data type, regardless if the type is signed or unsigned. When entering 0xF11 for an 8 bit type you will see a value of 0x11 as test data. Also when entering a binary 0b1100001111 you will see a value of 0b00001111 as test data.
- Missing leading zeros will be filled up for hexadecimal and binary values. If you enter 0x12 for a 16 bit value, you will see a value of 0x0012 as test data.
- Expressions will be validated for syntactical correctness and all values used within the expression (defines, enum constants or numerical values) will be validated as well.

After the truncation of the value to the available data type size, it will be formatted according to the data format configured within TIE. Suppose you have an 8 bit signed value with data format “Decimal” and you enter a value of 0xF05: The value will firstly be truncated to 0x05 and then formatted as decimal number so that you will see 5 as test data value.

Important: If you change the data format within TIE, only newly entered test data will be formatted according to the new format. If you want to change the format of the available test data for a certain variable, you need to use the “Convert to Data Format” menu entry within TDE. Make sure the box “Enable Value Checking and Conversion” is checked within the menu “Window” > “Preferences” > “Test interface Settings”.

Important: When running the test with undefined values, the initial value passed to the test object depends on the default initialization of the compiler.

Clicking into a cell activates the inline editing mode and you can enter arbitrary values:

→ Click in a cell and press enter.
Now you are in the inline editing mode (editing mode).
→ Enter the value.

You can navigate between the cells with CTRL + cursor left/right.

You can apply the available operations of the context menu to multiple cells depending on the current selection within the Test Data view:

- If you select a single variable of the interface tree, all values of all test steps for this variable will be affected.
- If you select a test step column, all variables of this test step will be affected.
- If you select an array, a struct or a union, all components of this element will be affected.

The current selection is highlighted in blue. You need to select a test step column before right-clicking for the context menu, because the right click will not select the test step column.

In case of union variables, the respective components can be displayed by doubleclicking the cell (see figure 6.199).

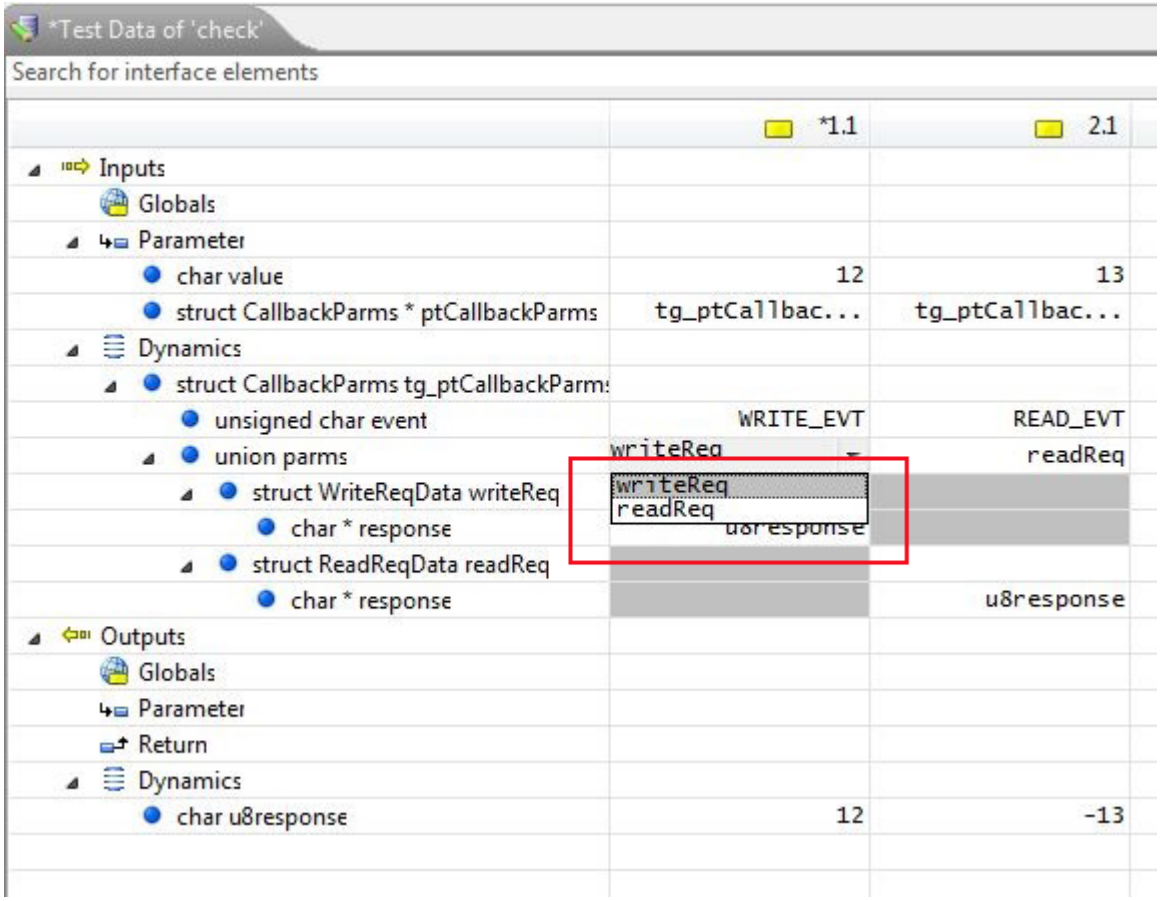


Figure 6.199: Clicking in the cell shows a combo box with the union components

Setting defines

Defines

→ Double-click in a cell or press ENTER to be in the inline editing mode.

→ Press Ctrl + Space.
A window will open with all known defines and enum constants (see figure 6.201).

In some countries you need to press CTRL + ALT + Space to get to this result, e.g. in China.

→ Choose your define with a doubleclick.

Important: If you want to select enum constants from anonymous enum declarations (e.g. enum {A=42, B=43, C=44}) that are not used within your source file, you need to set the module attribute “Collect All Enums” to “true” in the TEE.

Enums

Setting enums

→ Click in a cell.
A dropdown menu will open showing the available enum constants for this enum type (see figure 6.200).
→ Choose any constant or click into the inline editor field to enter any other suitable value.

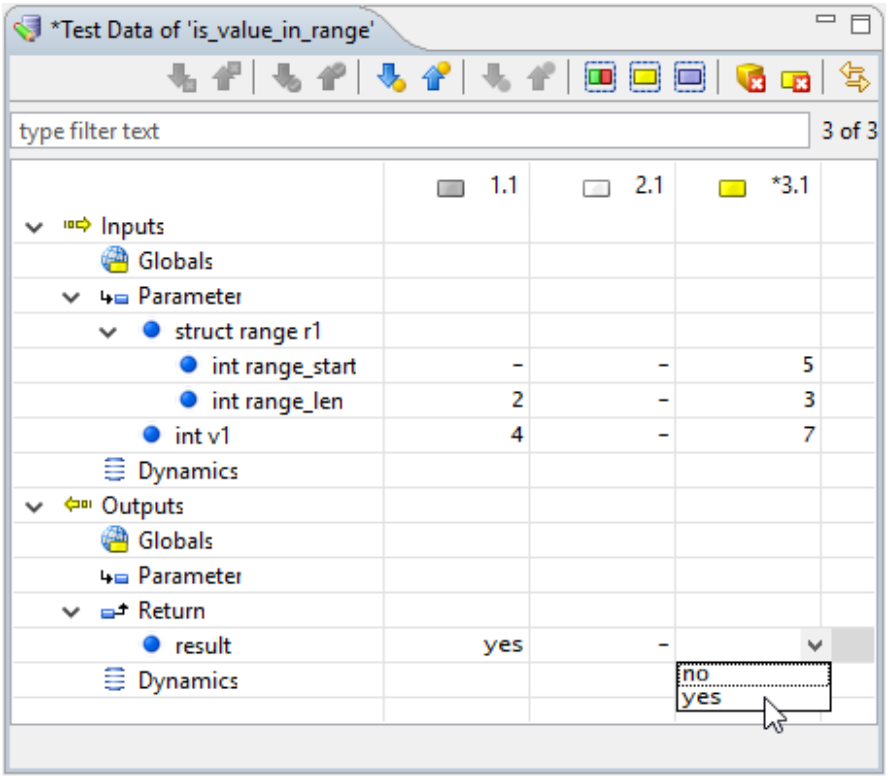


Figure 6.200: Clicking in the cell shows a combo box with the available enum constants

Arithmetic expressions

Enter expressions

→ Click in a cell.
→ Enter a valid expression.
You can press Ctrl + Space at any location within the expression to get a list of available defines or enum constants.

The following operators are supported in expressions: Addition, subtraction, multiplication, division, shift, binary or/and. The operands can be numbers, defines and enum constants. It is also possible to use parentheses, e.g. “(A + B) * 2”

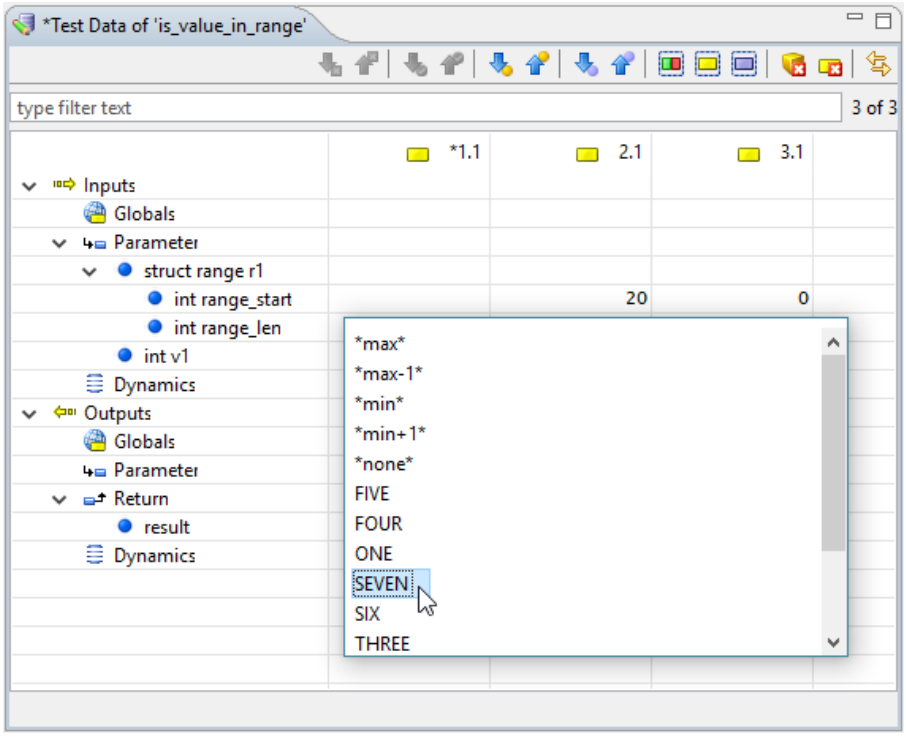


Figure 6.201: Pressing Ctrl + Space opens a list of available defines or enum constants

→ Select a define or enum constant.

→ Enter a valid operator and complete your expression within the inline editor of the cell.

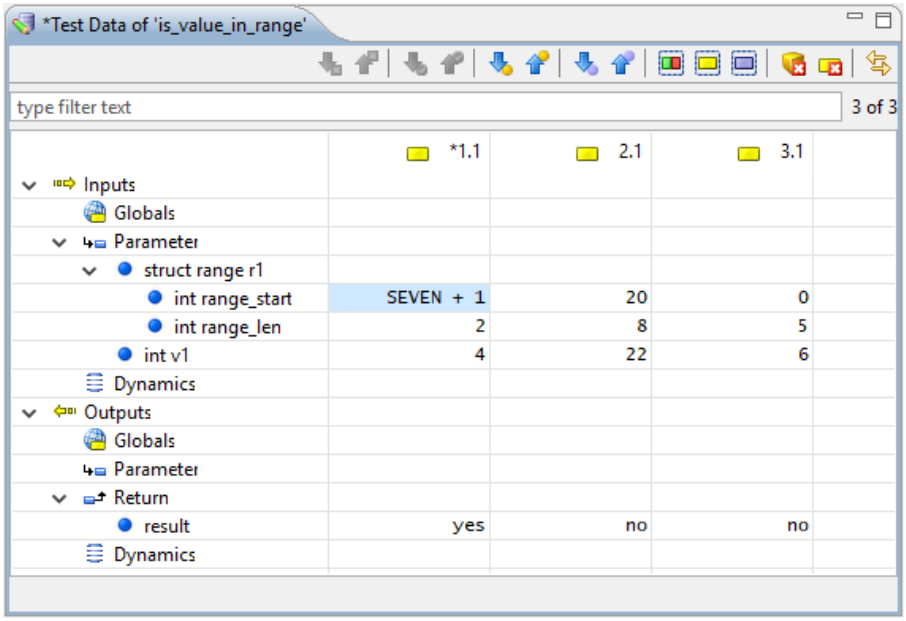


Figure 6.202: Arithmetic expression

Input values

Input values are all interface elements that need to have a defined value at the beginning of the execution of a test object. These values are included in the calculation of the output values during test execution or will be used as a predicate of a condition.

Types of input values

There are three types of input values :

- Global and external variables, used by the test object.
- Function parameters, transferred during function call.
- Dynamic objects: They represent pointer targets, referenced through a pointer of the test object interface (they are not really dynamic, variables will be created for each one within the generated test driver).

Vector values for advanced stub functions

External called functions can be defined as advanced stub functions to provide the return value and the expected parameter values within the Test Data view. If a test object calls an external function multiple times the same return value would be returned for each invocation and also the parameters would be checked against the same parameter values as specified within TDE. In order to provide different values for each invocation of the advanced stub, you can enter multiple values as a vector written within braces, e.g. `1,2` (see figure 6.203). In this example the return value of the stub will be 1 for the first call, 2 for the second call. You can also specify a vector value for the expected parameter values.

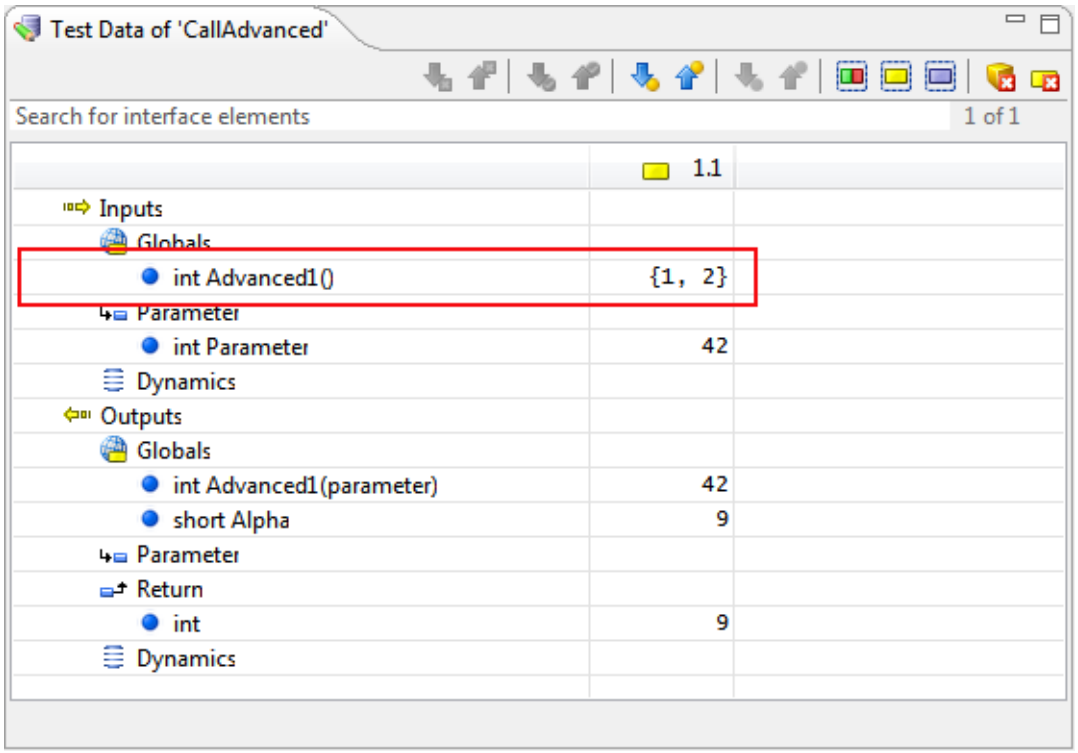


Figure 6.203: Entering values as vector for an advanced stub

Handling of Arrays

→ Right-click the array to open the array content menu.

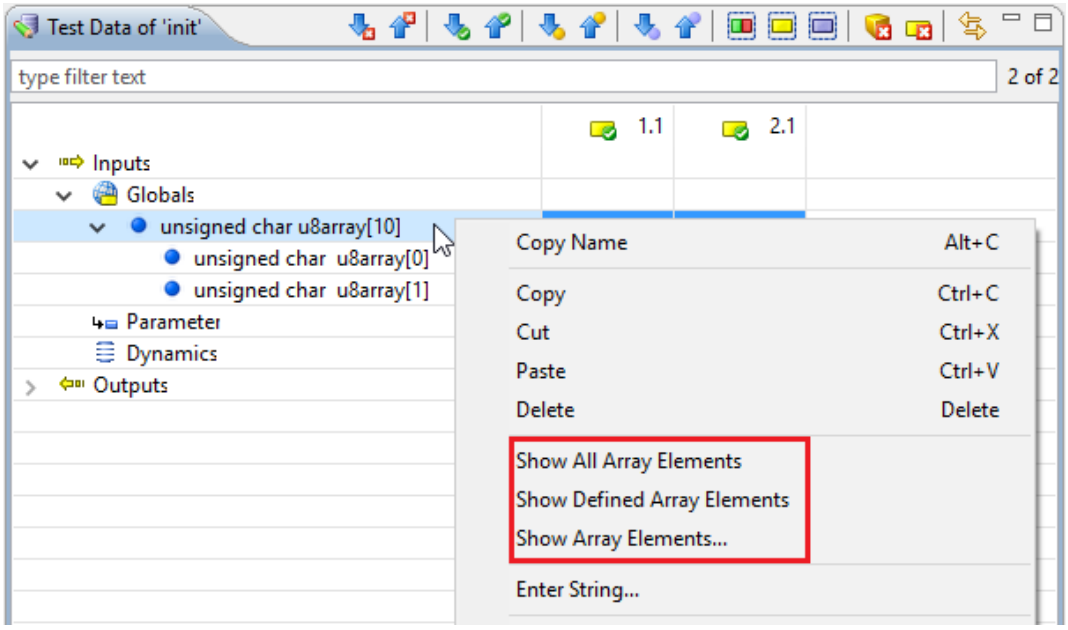


Figure 6.204: Choosing shown arrays

→ Choose ”Show All Array Elements“, ”Show Defined Array Elements“ or ”Show Array Elements…“ to decide which array elements are shown.

By choosing ”Show Array Elements…“ you can select the shown array elements individually.

Important: It is not necessary to enter test data for all array elements. Entering test data to only one array element is enough to make the test executable.

Expected values

Expected values are the calculated results regarding the input values for the test object after test execution. TESSY will compare both expected and actual values after test execution. The result will be either failed or passed.

Important: The values are compared with the evaluation mode equal (==). To change the evaluation mode refer to section Entering evaluation modes.

Types of expected values

There are three types of expected values:

- Global or external variables: They can always be expected values as they have a valid value after execution of a test object.
- The return value of a function.
- Dynamic objects: Referenced through a pointer that is either an input or expected value, dynamic objects continue to be accessible after execution of a test object. They can therefore also be expected values, just like global variables.

Initializing test data

You can initialize all interface variables of a test step at once.

→ Right-click the variable and choose “Initialize Test Data…” from the context menu. The “Initialize Values” dialog opens.

You can use following options:

Option	Meaning
Pattern	A pattern in hexadecimal format, i.e. 0xff
Value	A specific value only
Random	A range of generated values for the initialization. The random values will adhere to the min/max limits of each interface variable type
Ignore values	All input and expected values will be set to “*none*”
Initialize all array elements	All array elements will be initialized. Otherwise only visible array elements will be initialized

Table 6.70: Options of initializing values

The following table shows the initialization values for certain data types:

Type	Contents
Integer	*0x00000000* i.e. if *0x42* is entered as pattern, all int variables will be initialized with *0x42424242*.
Float	*0.0*
Char	The pattern or numerical value
Struct	All sub components are initialized according to their type
Union	The first sub component is initialized as active component
Enum	The first enum constant is used as initialization value
Array	All array elements are initialized according to their type if option “Initialize all Array Elements” is used
Pointers	Pointers are initialized with *NULL* provided that they do not point to dynamic objects

Table 6.71: Initialization values for data types

6.9.7.5 Set passing to IRRELEVANT

It is possible to set the passing direction of variables that are not needed for testing any more to “IRRELEVANT” in the Test Data Editor. Right-click the variables you want to hide and choose “Set Passing to IRRELEVANT” in the context menu. You can undo this by choosing “Restore Passing” in the same menu if necessary.

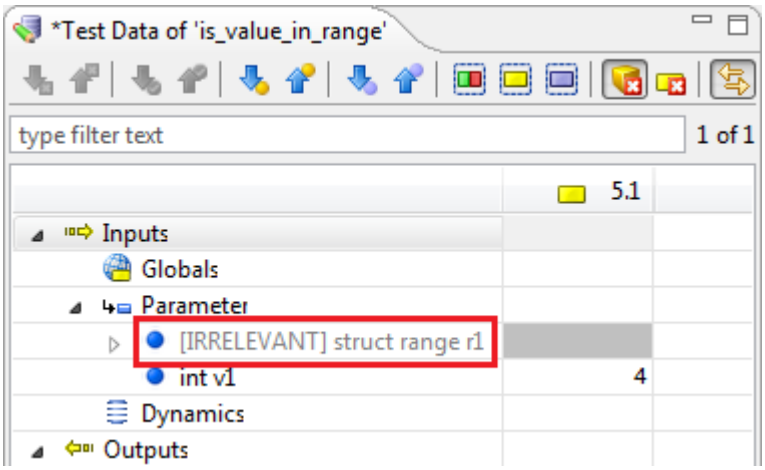


Figure 6.205: Passing direction set to irrelevant.

The chosen variable will still be displayed and marked as “[IRRELEVANT]”. When saving the test data the passing direction of this variable is updated and the variable will no longer appear within the Test Data view.

Important: Passing directions set to irrelevant that have been saved can only be restored in the Test Interface Editor (TIE) (see section 6.7.4.6 Setting passing directions).

6.9.7.6 Entering evaluation modes

Using the evaluation mode allows to specify how to compare the actual value (calculated during the test run) with your specified expected value. The evaluation mode together with the expected value will be used to process the test results after the test run.

Evaluation modes

The default evaluation mode is equal (==). To enter another evaluation mode:

→ Click in a cell.

→ Enter the desired evaluation mode within the inline editor mode (see figure 6.206).

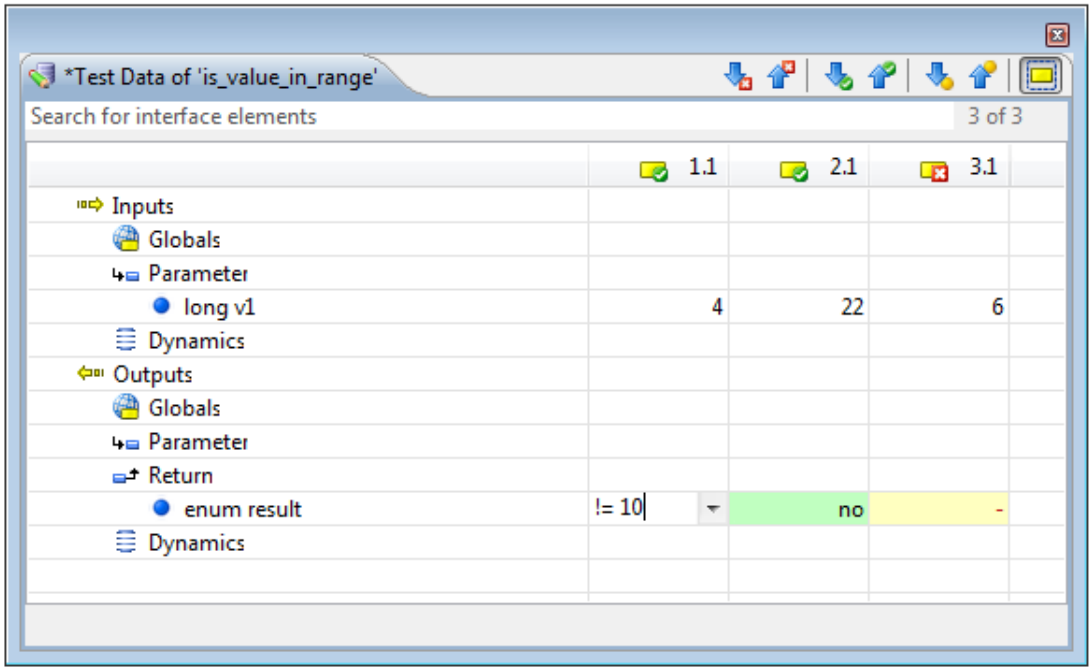


Figure 6.206: Entering evaluation mode “unequal” within the inline editor

The following evaluation modes are available:

Evaluation mode	**Written as**	**Meaning**
equal	==	Checks the expected value and actual value for Equality. **This is the default setting.**
unequal	!=	Checks the expected value and actual value for inequality.
greater	>	Checks if the actual value is greater than the expected value.
less	<	Checks if the actual value is less than the expected value.
greater or equal	>=	Checks if the actual value is greater or equal to the expected value.
less or equal	<=	Checks if the actual value is less or equal to the expected value.
range	[1:10]	Checks if the actual value is within a range, here: range 1 to 10.

continue next page

Evaluation mode	Written as	Meaning
deviation	100 +/- 1 100 +/- 1%	Checks if the actual value equals the expected value but takes into account a deviation value. The deviation can either be an absolute value or a percentage, for the example the following actual values would yield OK: 99, 100, 101.
positive/ negative deviation	100 +- +1 100 +- -1 100 +- +1% 100 +- -1%	It is also possible to specify a positive or negative value for the deviation: In the example the positive deviation would yield OK for 100 and 101 whereas the negative deviation would yield OK for 99 and 100.

Table 6.72: Evaluation modes

Important: Please note that in TESSY "Actual -> Expected" generally does not work for evaluation mode entries.

6.9.7.7 Ignoring values for a test step

By default, values have to be assigned for all variables with passing directions "IN" or "INOUT". It can be useful to not overwrite a value calculated in the last test step. In this case you can use the special value " *none* ":

→ Right-click a value and choose "Ignore Value" within the context menu.

6.9.7.8 Generating test steps automatically

You can generate test cases and steps automatically, i.e. test steps of a range of input values which you enter in the TDE:

→ Create a generator test case within the Test Items view as described in section 6.2.5.5 Creating test steps automatically.
→ Switch back to the Test Data view.
→ Enter your values and a range of an input value, i.e. [6:9] as in our example (see figure 6.207).

TESSY can generate the test cases stepwise: Enter a semicolon and the step size behind the range, e.g. [6:15;3] would give you the values 6, 9, 12 and 15.
You can also enter specific values, e.g. [1,5,8] would be the values 1, 5 and 8.
Combinations are as well possible: [2:8;2,11,15,20:22] would be 2, 4, 6, 8, and 11, 15, 20, 21 and 22.

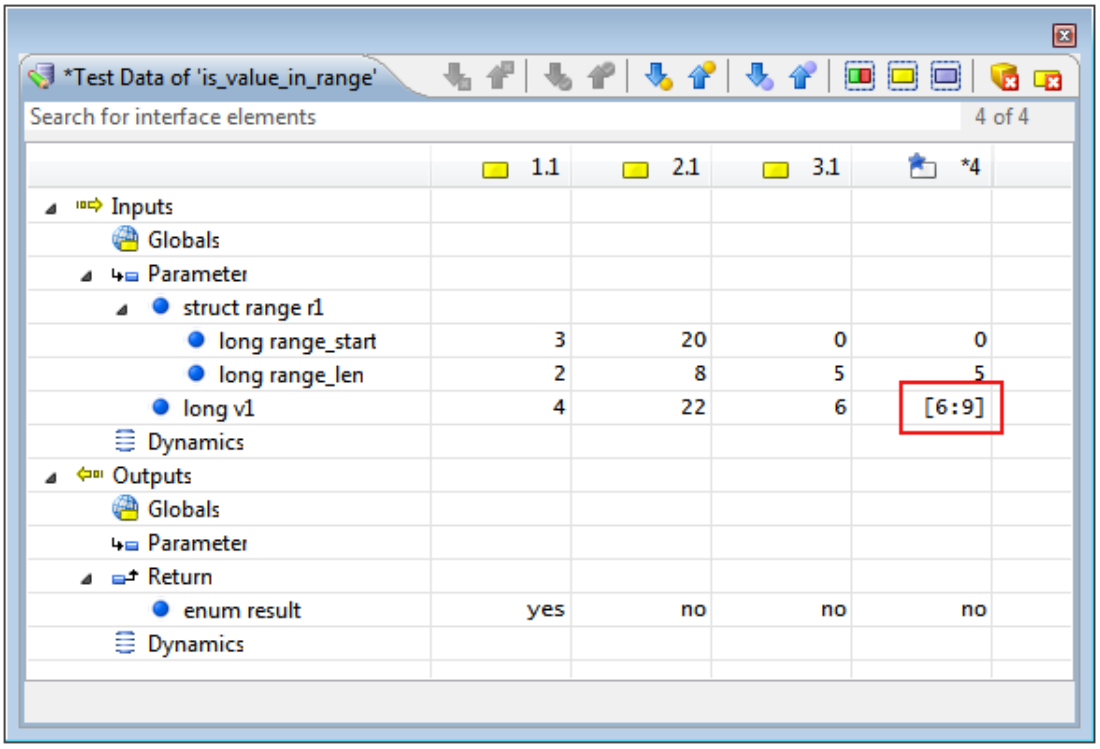


Figure 6.207: Generator test case 4 has a range value from 6 to 9 for parameter v1

→ Save your inputs with a click on in the TESSY tool bar.
TESSY will now automatically create a test step for every value within the range you entered (see figure 6.208).

You might need to expand or scroll the Test Data view to see all the test steps!

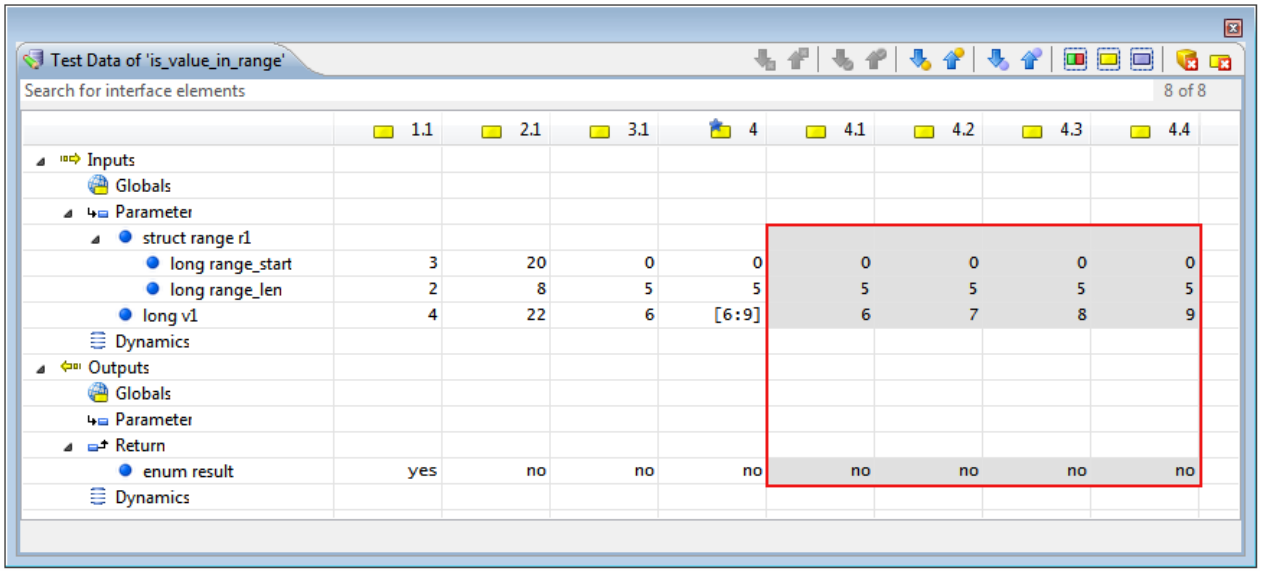


Figure 6.208: Four test steps are generated for every value within the range “6 to 9”.

The test steps are read only because they were generated!

You can change the type of the test case and test steps to “normal”. That way you can edit the test steps as usual.

To change the status to normal,

→ switch to the Test Items view.

→ Right-click the test case and select “Change Test Case Type to Normal” (see figure 6.209).

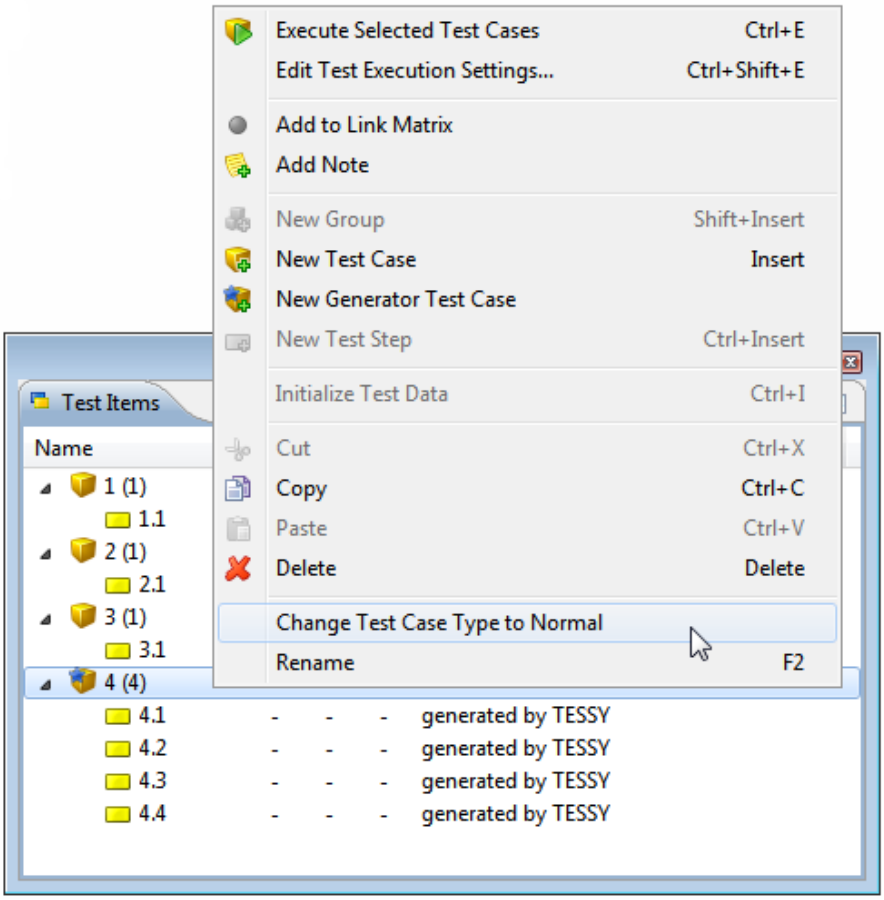


Figure 6.209: Selecting “Change Test Case Type to Normal”

Changing test case to type normal

The test case and test steps are changed to type “normal” but will indicate originally being generated with a status within the Test Items view (see figure 6.210).

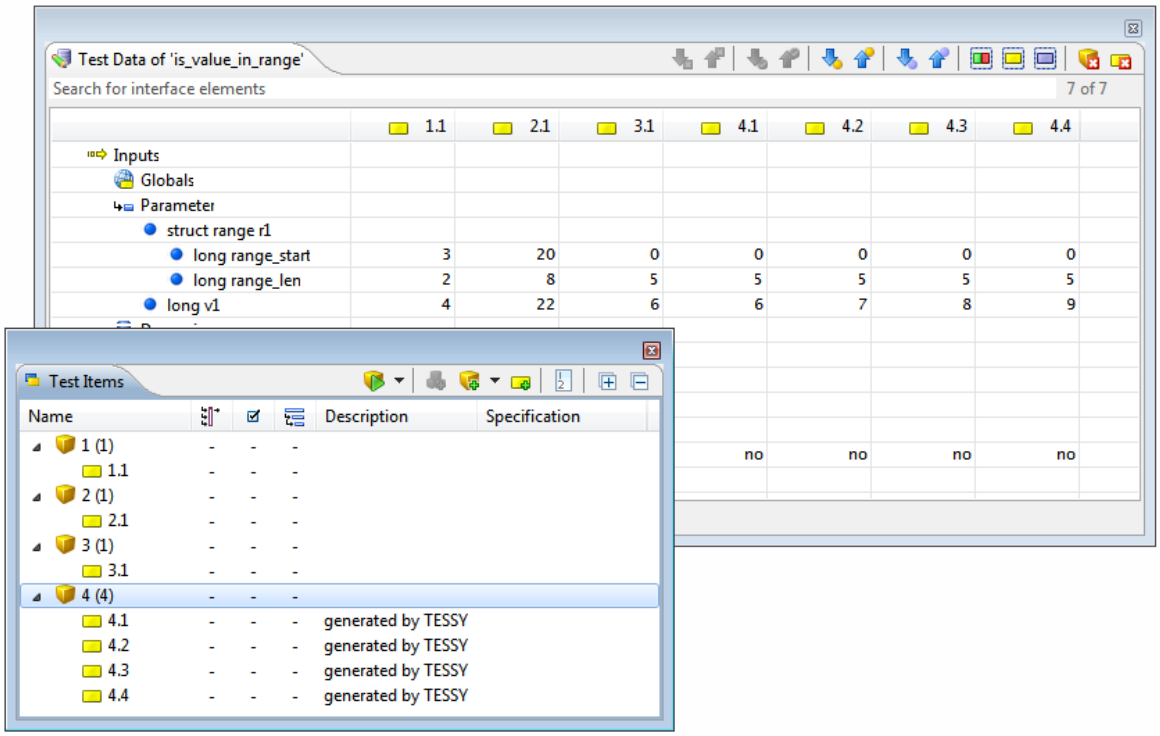


Figure 6.210: The test case and test steps originally being generated.

You can reverse the action with a right click and choose “Change Test Case Type to Generator” from the context menu.

6.9.7.9 Changing inherited values

Inherited modules and their test objects need to be synchronized (see Creating variant modules) to get the inherited test cases and test steps with all inherited values. The Test Data view shows inherited and overwritten values with different colors.

Please notice the following restrictions for editing of inherited values:

- Dynamic objects will be inherited from the parent test object. Additional dynamic objects cannot be created within the inherited test object.
- CTE test cases cannot be edited within the inherited test object. Any changes need to be done within the parent test object.
- Inherited user code (e.g. prolog/epilog) cannot be overwritten with “empty” user code. It is recommended to add a comment stating why the inherited usercode has been overwritten instead.

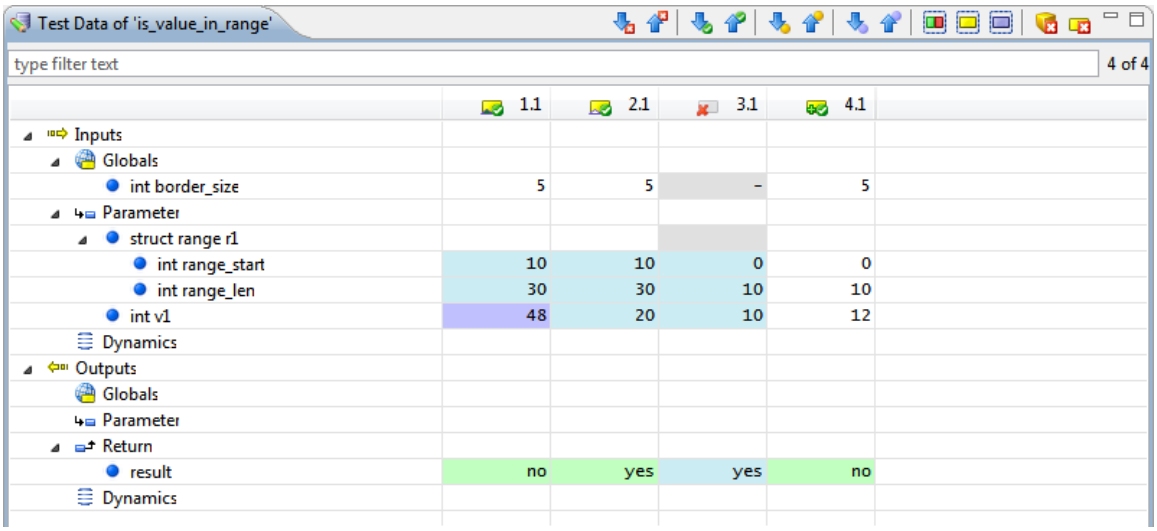


Figure 6.211: Inherited value coloring within Test Data view

See figure 6.211 for the color coding of values displayed within Test Data view:

- Inherited values are displayed in light blue.
- New variables of the variant test object are shown like normal values without special highlighting. The variable “border_size” was introduced within the variant source code, therefore there are no inherited values.
- Overwritten values are displayed in darker blue. The value 48 is overwritten and the tooltip will show the inherited values for such overwritten values.
- The test step 3.1 has been deleted: The inherited values are displayed for information only. The test step will be skipped when executing the test.
- The additional test step 4.1 cannot have any inherited values. All values are displayed as normal.

6.9.7.10 Pointers

→ Right-click the pointer value cell to open the context menu.

The context menu offers the following possibilities to assign a value for a pointer:

Option	Meaning
Set Pointer NULL	The value of the selected pointer will be set to NULL. The text box will be filled with NULL.
Set Pointer Target	You can select another interface element or a component of a structure or union and assign its address to the pointer. The cursor will change, when you move the mouse pointer over a variable: ⌖ The object type fits the pointers target type. You can assign the pointer. ⊘ The object type does not match the pointers target type. You cannot assign the pointer. When you click on an element, the variable name of that element will be entered into the field next to the pointer. During test execution, the address of the variable will be assigned into the input field of the pointer.
Create pointer target value	Allows to create a new object as target object for the pointer. The address of the object will be assigned to the pointer. The type of the created object depends on the target type of the pointer. → Choose “Create Pointer Target Value” from the context menu. → Enter a valid C identifier as name for the new target object. → Click “OK”. A new target object will be listed in the dynamic objects section of the TDE.

continue next page

Option	Meaning
Array as Target Value	It is also possible to create an array as target value using the Dimension option of the Create Pointer Target dialog: → Tick the check box for" As Array of Size" to enter an appropriate size into the input field. → Click "OK". The name of the new object appears in the input field of the pointer value. TDE will create an array of the pointers target type. The pointer will point to the first array element. Within the Dynamics section, you will see the newly created target object. You can enter values, like for every other interface element.

Table 6.73: Value assignments for pointers

6.9.7.11 Static local variables

Variables defined as static local variables within the test object or called functions can also be tested. Since such variables are not accessible from outside the location where they are defined, TESSY instruments the source code and adds some code after the variable definition to get a pointer to the memory location of the variable. All static local variables can only be accessed after the code containing the variable definition has been executed. You need to keep this in mind when providing input values or checking results for such variables. The following restrictions apply for static local variables:

- The first time when the code of a static local variable definition is executed, the variable will get the initialization value assigned from the source code. It is not possible to set the initial value from TESSY. You need at least one test step to initialize the variable by executing the definition code. The next test step can then supply an input value for the variable.
- The same applies for expected values: If the source code of the variable definition has not been executed, the result value of the respective variable is not accessible and will be displayed as *unknown* in this case. This situation may arise when the variable definition is located within a code block which has not been executed, e.g. within an if statement block.

6.9.8 Test Definition view

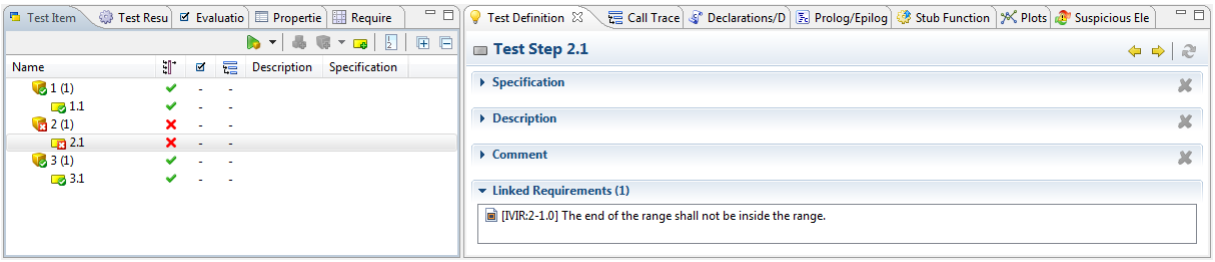


Figure 6.212: Test Definition view within TDE with linked requirement

The Test Definition view displays the test case specification, the optional description and linked requirements of the current test case in individual input fields. The test case specification should enable the tester to provide concrete input values and expected results.

The Test Definition view is context sensitive! To display the specifications, definitions and requirements for a test case:

→ Select a test case within the Test Items view (see figure 6.212).

Important: The contents are not editable if the test cases have been created and exported using the CTE!

6.9.9 Call Trace view

The Call Trace view displays the called functions for each test step of a test object within the Expected Calls area. All functions that may be called from the test object are listed within the Available Functions area.

The Call Trace view allows the evaluation of the called functions.

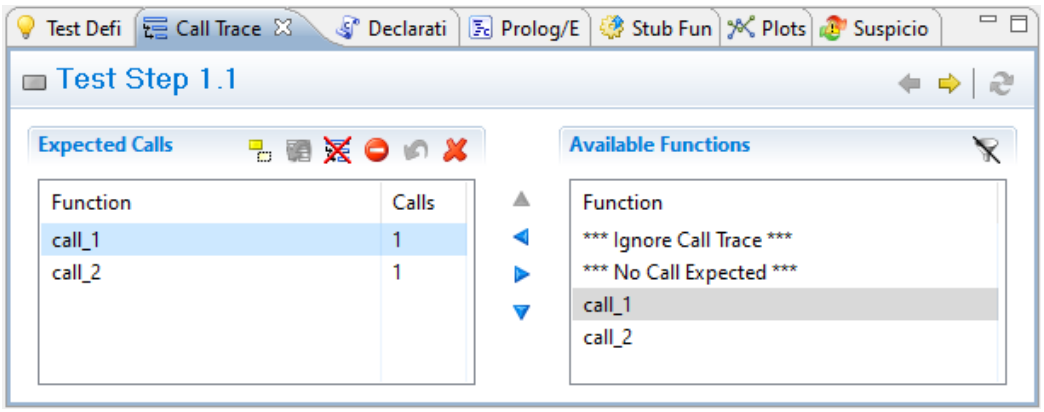


Figure 6.213: Call Trace view

Within the two areas of the Call Trace view it is possible to manage the expected order of function calls for all test steps individually making use of the tool bar. It is also possible to use the blue arrows between the areas pointing left, right, up or down. When the editing is completed the settings need to be saved.

6.9.9.1 Icons of the view tool bar

Icon	Action / Comment
	Applies called functions to all test steps.
	Copies the actual called function into the Expected Calls area.
	Ignores all called functions.
	Deletes all called functions.
	Restores inherited called functions.
	Deletes selected called function.
	Disables the filter and shows all available functions (by default, functions called by stubbed functions are filtered out).

Table 6.74: Tool bar icons of the Call Trace view

6.9.10 Declarations/Definitions view

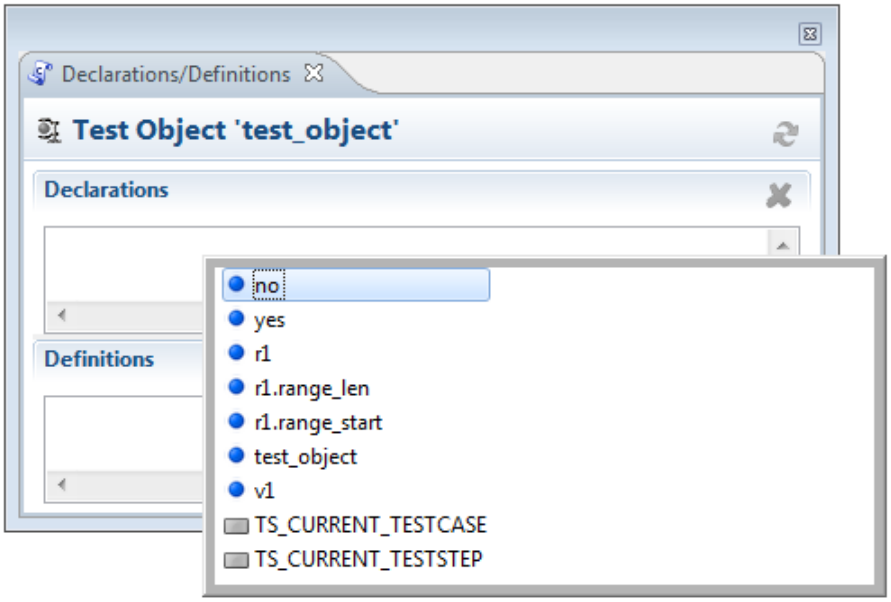


Figure 6.214: Declarations/Definitions view

Within the Declarations/Definitions view you can define your own helper variables that may then be used within the user code. If you just want to declare a variable that is already available within linked object files, you do this within the declarations section. If you want to create a new variable, you need to enter the declaration into the declarations section and the respective definition into the definitions section. The variable can then be used within the prolog/epilog and stub function code.

Important: TESSY provides the means to add new variables within the TIE perspective (see section 6.7 TIE: Preparing the test interface). Such variables can be used like normal interface variables of the test object which is much more convenient than defining them here in the Declarations/Definition view.

6.9.11 Prolog/Epilog view

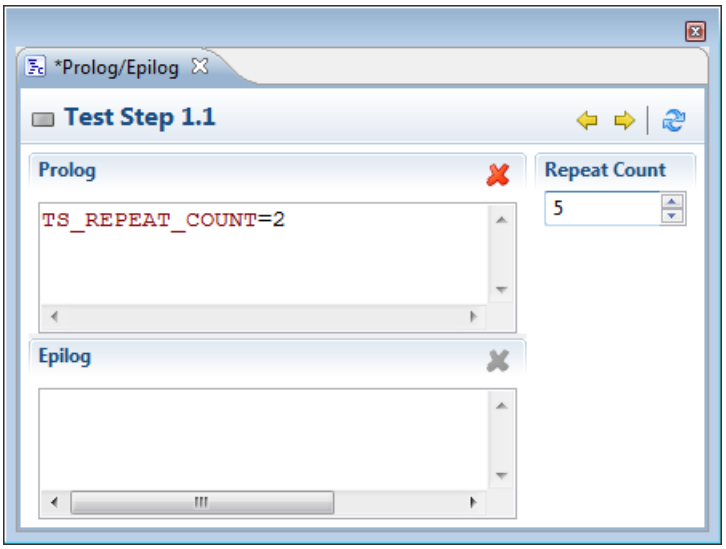


Figure 6.215: Prolog/Epilog view

Within the Prolog/Epilog view you can specify usercode that will be executed at a certain point in time during the test execution. The C part of the usercode will be integrated into the test driver and executed at the places specified. The following figure outlines the call sequence of the usercode parts.

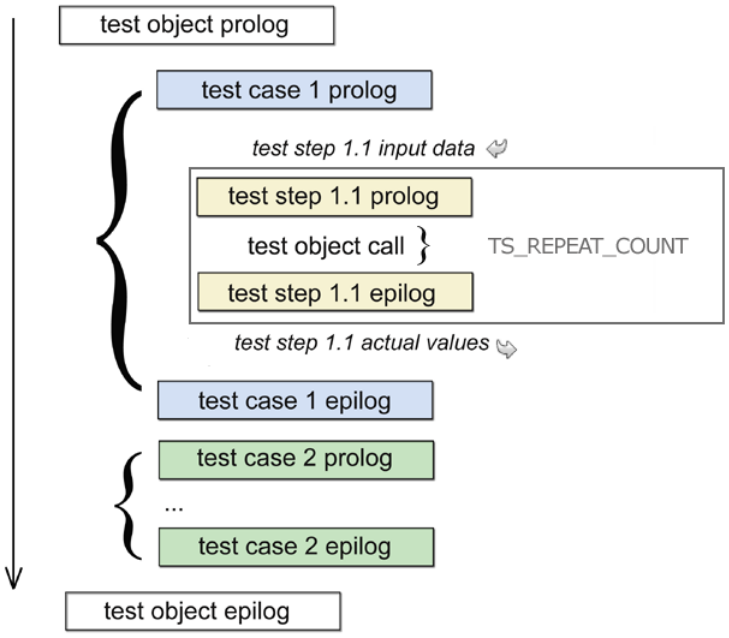


Figure 6.216: Call sequence of the usercode parts

The figure shows the interaction of the usercode sections with the assignment of test data provided within TDE and the result values that are saved into the test database and evaluated against the expected results.
During the test object call, the code specified for the stub functions (if any functions are called from your test object) will be executed depending on the code logic of your test object.

Within the prolog/epilog code you can reference the global variables used by your test object that have one the passing directions IN, OUT, INOUT or EXTERN. The following special macros are available within the prolog:

- TS_REPEAT_COUNT is a variable that controls the number of repeated calls of the test object, the default is 1.
- TS_CURRENT_TESTCASE, TS_CURRENT_TESTSTEP contains the current test case and test step number. This can be used to do different things for certain test cases.
- TS_TESTOBJECT_RETURN contains the return value of the latest test object call (if the test object has a return value). This value can be used within evaluation macros in the test step prolog.
- TS_THIS is available for C++ methods only and allows to access members of the current “this” object (e.g. “`TS_THIS.member = 5;`”).

It is recommended to define specific prolog/epilog code instead of using the macros TS_CURRENT_TESTCASE/TS_CURRENT_TESTSTEP

Example

Have a look at figure 6.215 Prolog/Epilog view in the beginning of this section. The test step 1.1 prolog contains the code TS_REPEAT_COUNT=2, and the Repeat Count for this prolog/epilog section was set to 5.
The whole prolog/test object call/epilog sequence will be repeated five times and the test object will be called twice in every repetition of this loop. Since there are 5 loops, the test object will be called 10 times in total.

6.9.11.1 Defining and overwriting inherited code

In some cases it is useful to specify a common prolog/epilog for all test steps or for the test steps of a certain test case. For this reason, you can enter prolog/epilog on test object or on test case level. Such a default prolog/epilog will be inherited to the respective child test steps. In this way you avoid to copy the same prolog/epilog multiple times to each test step. Default prolog/epilog can be overwritten on test case/step level for individual test cases/steps if desired.

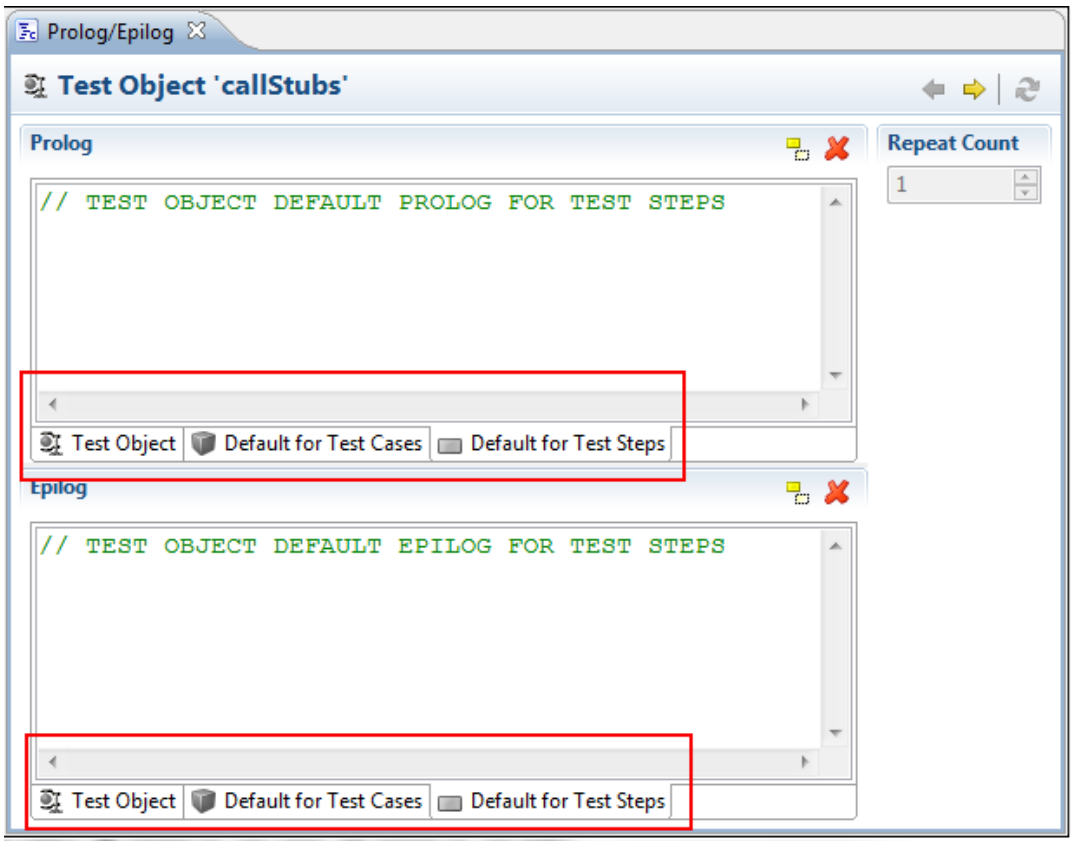


Figure 6.217: TESSY provides default prolog/epilog on test object level to be inherited to test cases and test steps

For both prolog and epilog, there are up to three tabs available, depending on the selected object:

Test objects:

- “Test Object” - The test object’s own prolog/epilog
- “Default for Test Cases” - The default prolog/epilog that will be inherited to all test cases of this test object
- “Default for Test Steps” - The default prolog/epilog that will be inherited to all test steps of this test object

Test cases:

- “Test Case” - The test case’s own or inherited prolog/epilog
- “Default for Test Steps” - The default prolog/epilog for the test steps belonging to that test case

Test steps:

- “Test Step” - The test step’s own or inherited prolog/epilog

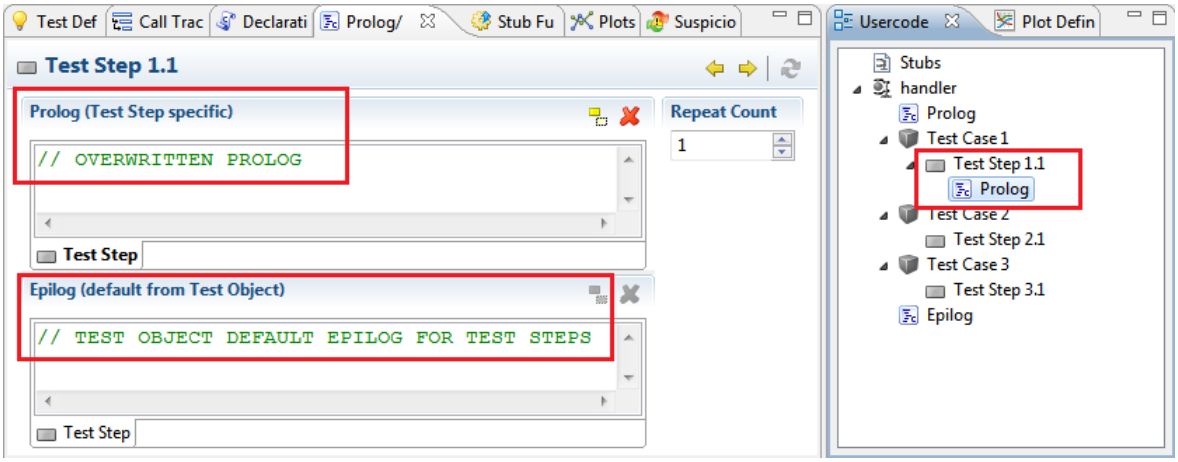


Figure 6.218: TESSY allows Prolog/Epilog being inherited from test case or test object

Important: To edit the default prolog/epilog, select the corresponding test object or test case and edit the code via the “Default for Test Cases” and “Default for Test Steps” tabs.

6.9.11.2 Entering C code

The Prolog/Epilog view provides a popup menu containing variables for convenient editing.

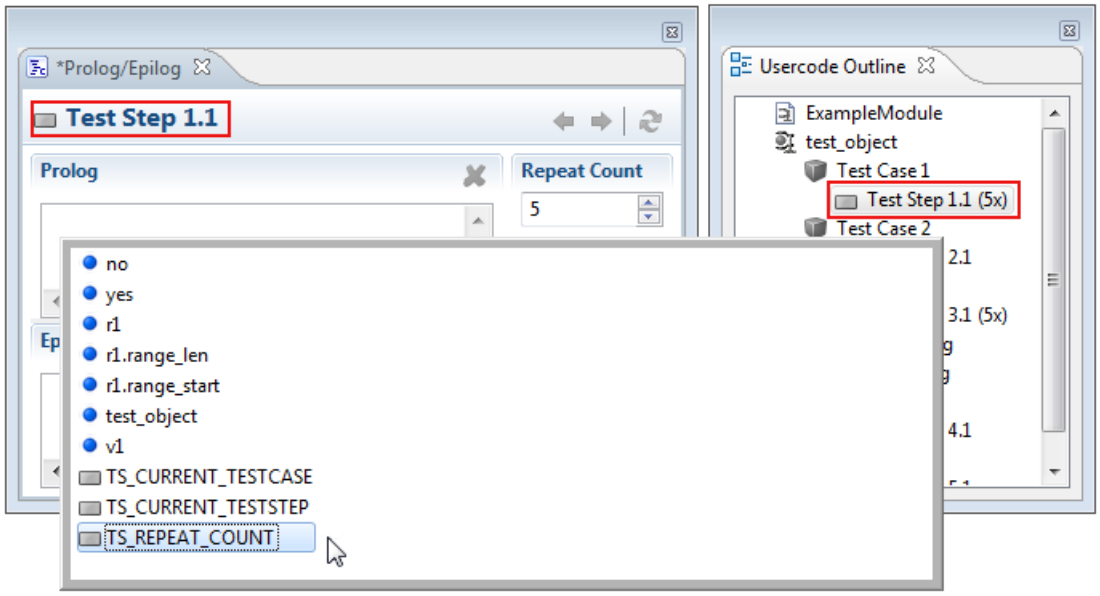


Figure 6.219: Prolog/Epilog functions

To show this menu,

→ use the Usercode Outline view to mark the test case or test step for which you want to set the usercode.

→ Click into the Prolog or Epilog section of the Prolog/Epilog view and enter the usercode.

→ Press CTRL + Space or type the first letters and press CTRL + Space.
The popup menu appears (see figure 6.219), showing all available names respectively the filtered list according to the characters you already typed.

Important: TS_REPEAT_COUNT is only usable within the prolog section!

When generating the test driver, the prolog/epilog code is appended to the source file of the current test object. Therefore, only variables that are declared or defined within the source file of the current test object may be used within prolog/epilog or stub function code.

To edit the prolog/epilog for a test case/test step:

→ Use the Usercode Outline view to navigate and select a test case or test step from the tree.

→ Enter the code within the Prolog/Epilog view.
A new node will automatically appear at the corresponding place in the outline tree (see figure 6.220).

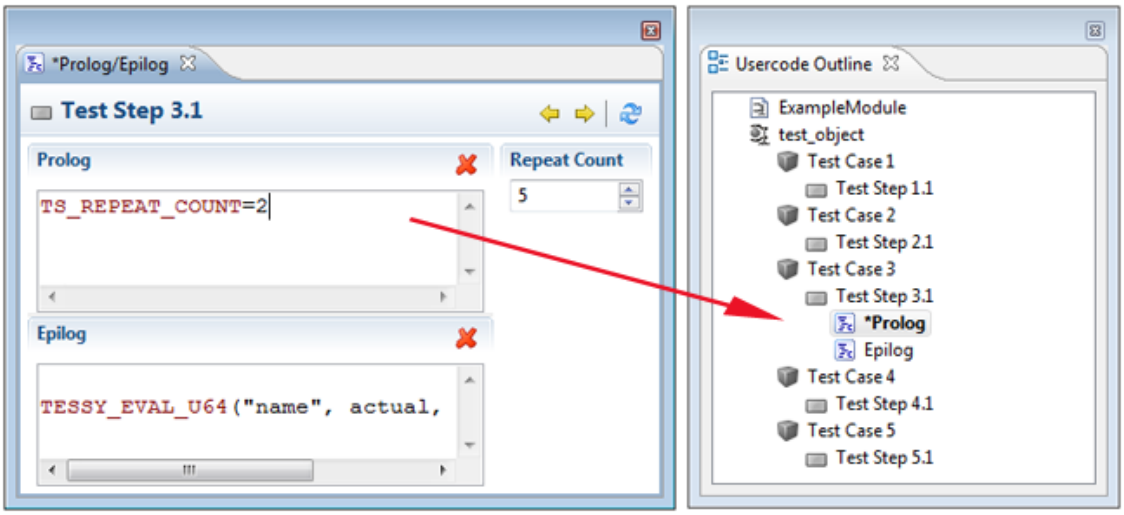


Figure 6.220: Editing C code

6.9.11.3 Using evaluation macros

Within the test step epilog or within stub functions, you can evaluate any variable or expression using the evaluation macros. These predefined macros allow to check an expression against an expected value. The result is stored within the test report like the evaluation of normal output variables of the test object.
Evaluation macros can only be used within the following Usercode sections:

- test step epilog
- stub function code

Evaluation macros

A popup menu contains all available interface variables and symbolic constants for convenient editing as well as the available evaluation macros, e.g. TESSY_EVAL_U8 for unsigned character values:

→ Press CTRL + Space.

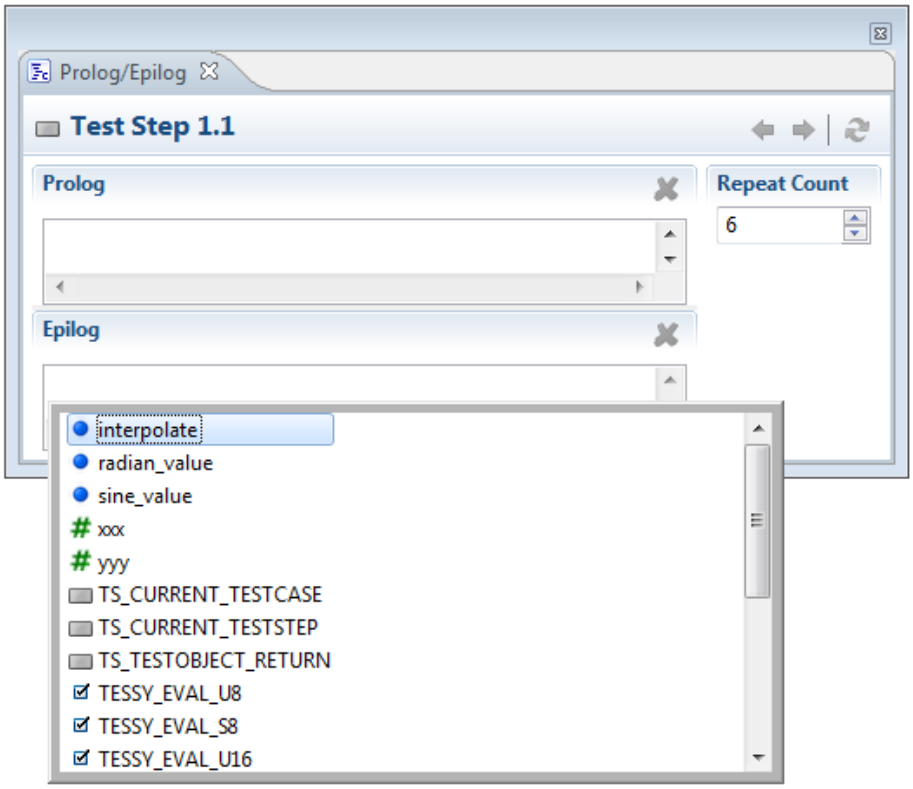


Figure 6.221: Call the popup menu by pressing CTRL + space

→ Select the evaluation macro for the specific data type of the variable which shall be evaluated. The only difference of the evaluation macros is the type of argument for the actual and expected value, see table below for a description of the available types.

→ Now the template can be edited.

Example: Below is an example showing the template in the second row and the edited evaluation macro underneath.

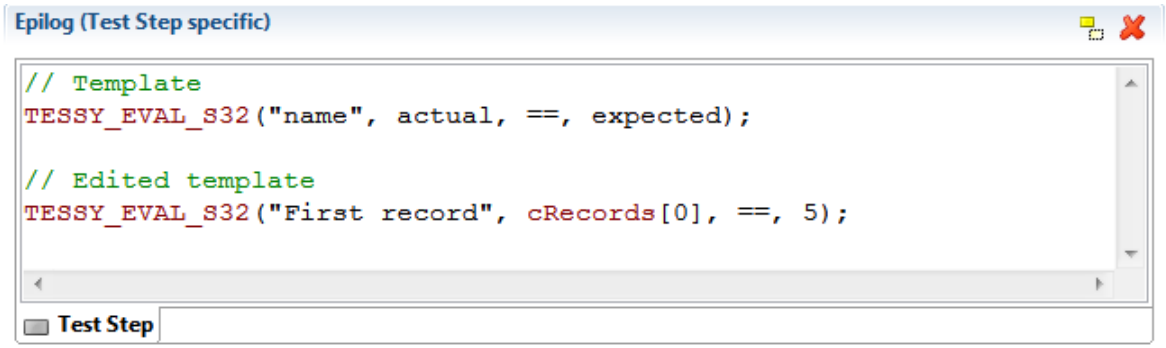


Figure 6.222: Editing the evaluation macro templates

Both value arguments given to the evaluation macro may be of any value that fits the specified eval macro type. By convention, the first (left side) value should be the actual value that shall be checked and the second (right side) value should be the expected result. Like this you will get the same order of values within the test report as for normal output values.

Evaluation macro name	C type
TESSY_EVAL_U8	unsigned, 1 byte
TESSY_EVAL_S8	signed, 1 byte
TESSY_EVAL_U16	unsigned, 2 byte
TESSY_EVAL_S16	signed, 2 byte
TESSY_EVAL_U32	unsigned, 4 byte
TESSY_EVAL_S32	signed, 4 byte
TESSY_EVAL_U64	unsigned, 8 byte
TESSY_EVAL_S64	signed, 8 byte
TESSY_EVAL_FLOAT	float
TESSY_EVAL_DOUBLE	double
TESSY_EVAL_LONGDOUBLE	long double

Table 6.75: Available types of evaluation macros

Operator	Meaning
==	equal
!=	unequal
<	less

continue next page

Operator	Meaning
>	greater
<=	less or equal
>=	greater or equal

Table 6.76: Operators of evaluation macros

Each invocation of an evaluation macro results in an additional entry within the test report. All evaluation macros will be added to the list of actual/expected values of the current test step. The results will be displayed within the Usercode Outline view and the Evaluation Macros view.

It is possible to format the output of the evaluation macros as binary value, decimal or hexadecimal (default setting) by appending one of the following format specifiers at the end of the evaluation macro name:

binary	`%bin`, e.g. "`Value printed as bin%bin`"
decimal	`%dec`, e.g. "`Value printed as dec%dec`"
hexadecimal	`%hex`, default setting.

Table 6.77: Evaluation macro specifiers

The report shown below contains all possible evaluation macro name formats. The format specifier itself will be omitted within the final evaluation macro name.

Test Case 1 ✔

Test Step 1.1 (Repeat Count = 1) ✔

Epilog

```
TESSY_EVAL_U8("Value (default is hex)", cRecords[0], ==, 5);
TESSY_EVAL_U8("Value printed as dec%dec", cRecords[0], ==, 5);
TESSY_EVAL_U8("Value printed as bin%bin", cRecords[0], ==, 5);
```

Name	Actual Value	Expected Value	Result
Value (default is hex)	0x05	0x05	✔
Value printed as dec	5	5	✔
Value printed as bin	0b00000101	0b00000101	✔

Figure 6.223: Formatting of evaluation macro values

6.9.12 Stub Functions view

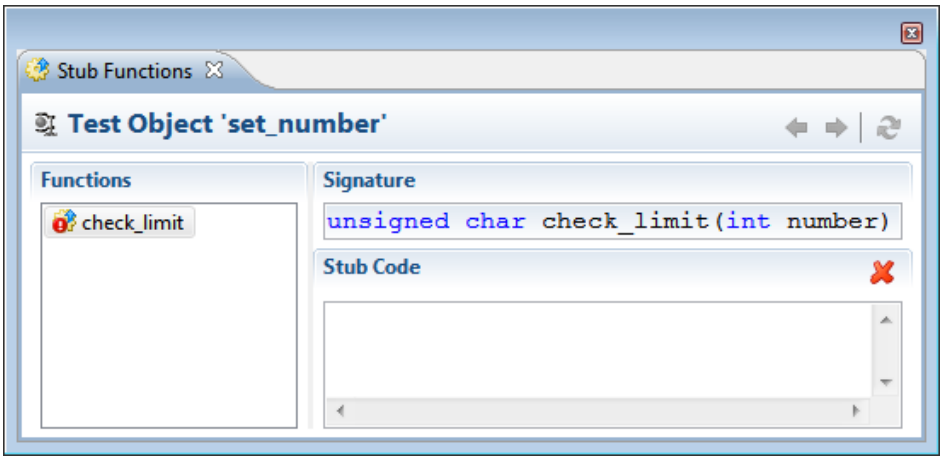


Figure 6.224: Stub Functions view without contents

The Stub Functions view displays the code for all stub functions. Normally all stub code is defined on test object level.

In the Stub Functions view you can insert stub code for test steps, test cases, and test objects. The code fragments will be be combined into a single stub function implementation and will be called in the order as shown in figure 6.225. If you don't want to execute the parent fragments for specific test cases or test steps, you need to add a return statement within the respective stub code fragment.

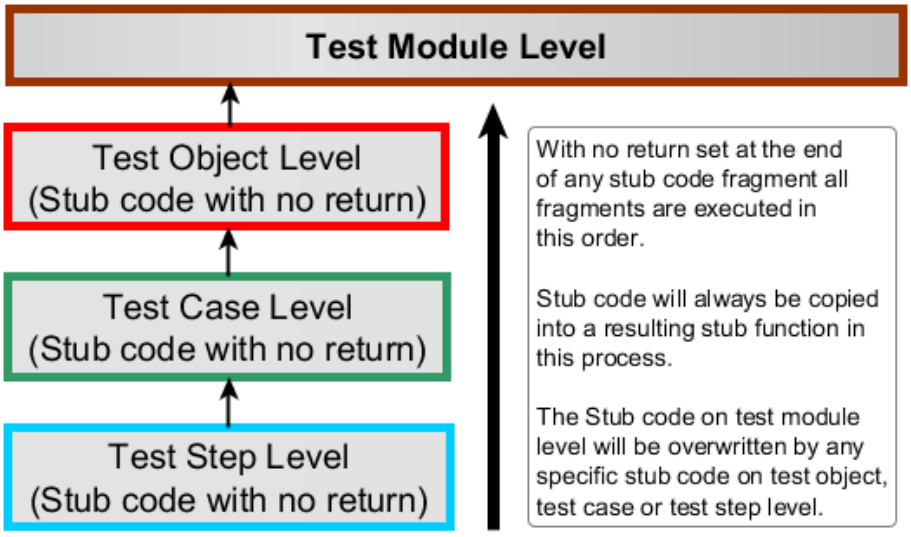


Figure 6.225: Test execution direction using stub code

Important: Stub code must be provided for all non-void stub functions in order to return a valid value as result of the stub function call. If there are stub functions without stub code, the test execution will be aborted with an error. If the return value of a stub function is not used by the test object, you should add at least a comment here.

Please note the error icon at the stub function name in figure 6.224 indicating that stub code is missing. You can switch off this check by unchecking the respective test execution preference. This preference setting will be stored within the preferences backup file as described within Window > Preferences menu.

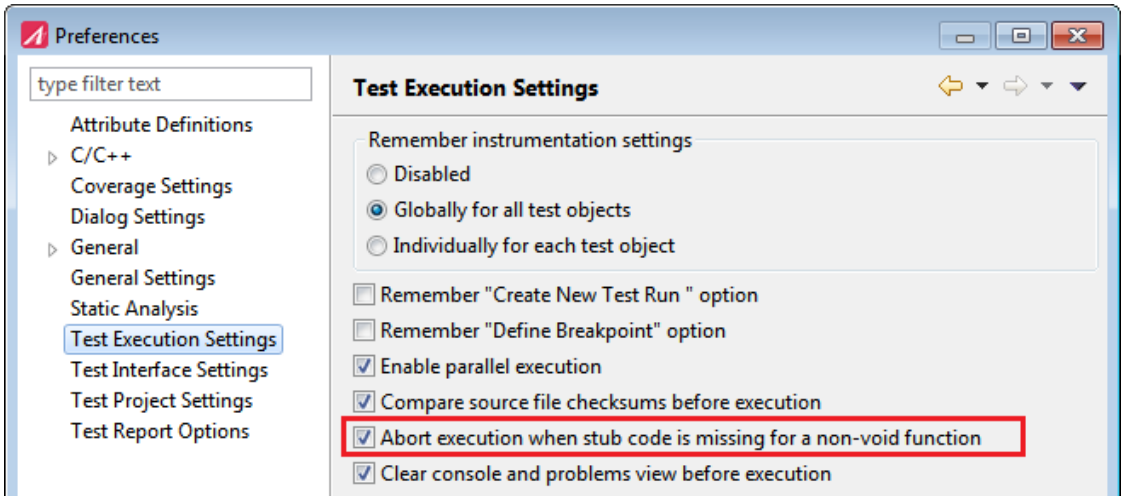


Figure 6.226: TESSY Preferences: Abort on missing stub code

6.9.12.1 Using Stub Functions

Within the stub function code you can reference the parameters passed to the stub function and also global variables used by your test object. The following special macros are available within the stub body:

- TS_CALL_COUNT contains the number of calls to this stub function for the current test step. This count will restart from one for each test step.

The size of the call count is limited to 8 bit (maximum call count of 255). Fore more information about defining the call count size please refer to the application note “Environment Settings (TEE)” in TESSY (“Help” > “Documentation”).

- TS_CURRENT_TESTCASE, TS_CURRENT_TESTSTEP contains the current test case and test step number. This can be used to provide different values for certain test cases.

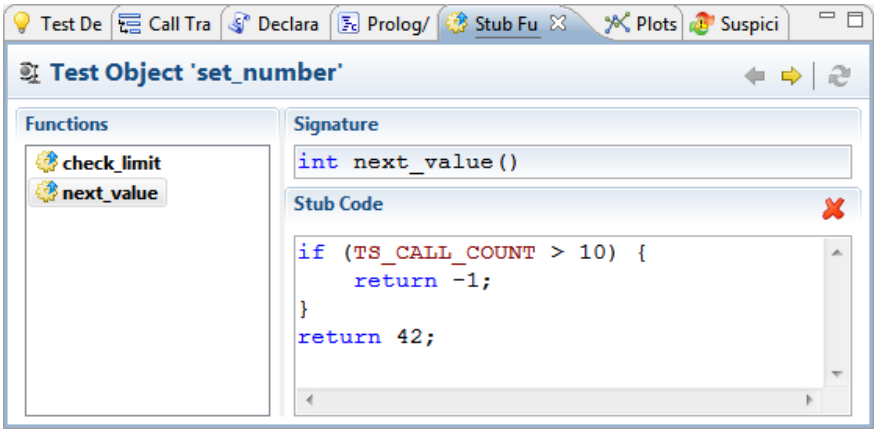


Figure 6.227: Stub Functions view with code using TS_CALL_COUNT macro

It is recommended to define test case/step specific stub code instead of using the macros TS_CURRENT_TESTCASE/TS_CURRENT_TESTSTEP.

Example for the use of test object, test case and test step specific stub code:

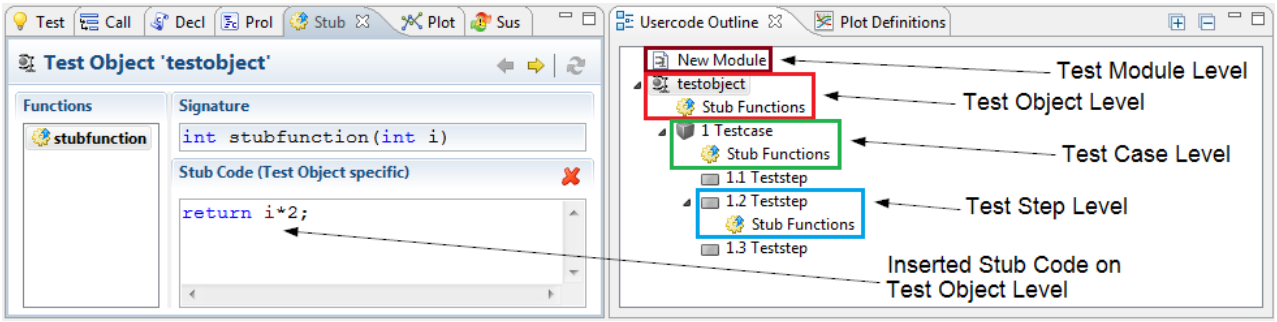


Figure 6.228: Stub Code Levels in the Usercode Outline view

If only stub code of e.g. the test step should be executed, you need to set a return at the end of the inserted code on test step level. If only stub code of the test step and the test case should be executed, you need to set a return at the end of the inserted code on test case level. Stub code on test module level will be overwritten.

It is recommended to read the sections Usercode Outline view and Prolog/Epilog view to fully understand the handling of stub code.

TESSY will automatically generate the code to execute a test including all the stub code you inserted in the Stub Functions view. Below you can see brief examples of inserted stub code on test object level, test case level and test step level on the left along with the automatically generated code resulting from that on the right.

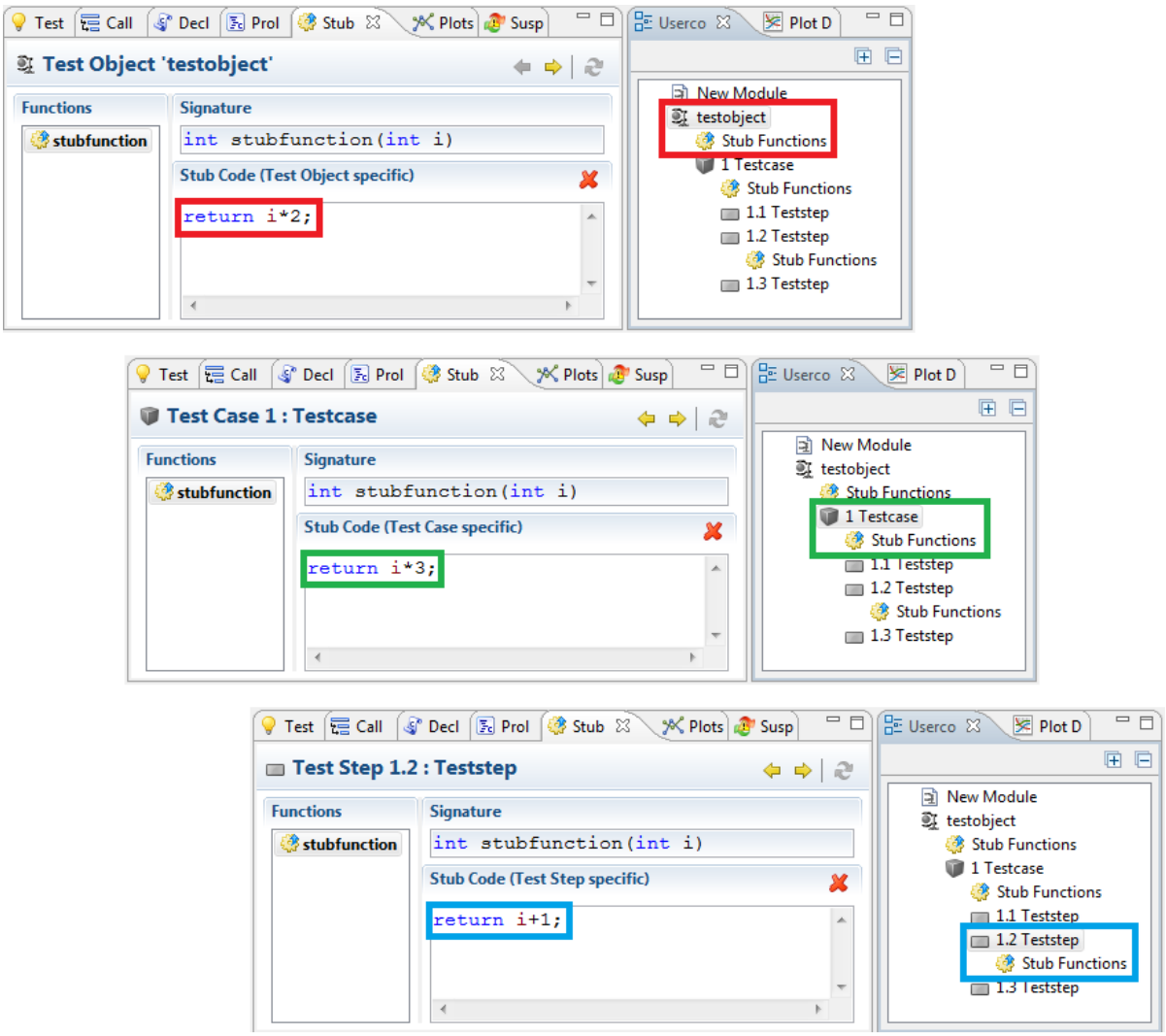


Figure 6.229: Stub code examples on test object, test case and test step level

In the automatically generated test code (see figure 6.230) you can recognize the test execution direction as shown in figure 6.225.

```
{
if (1 == TS_CURRENT_TESTCASE && 2 == TS_CURRENT_TESTSTEP) {
return i+1;
}
if (1 == TS_CURRENT_TESTCASE) {
return i*3;
}
{
return i*2;
}
```

Figure 6.230: Automatically Generated Test Code

For more information about the Usercode Outline view and navigating within the test items see section Usercode Outline view.

6.9.13 Usercode Outline view

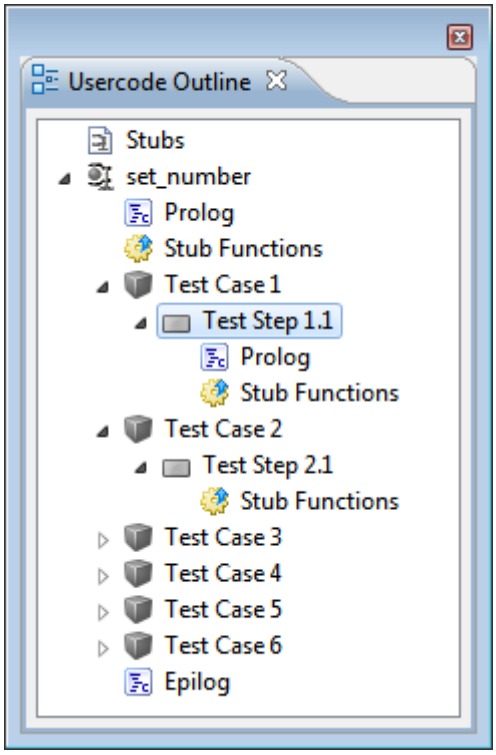


Figure 6.231: Usercode Outline view

The Usercode Outline view displays the usercode and stub function code that will be executed at a certain point in time during the test execution and that you just defined in the Prolog/Epilog view or Stub Functions view. Use this view to navigate within the test items when editing prolog/epilog or stub function code.

The view shows entries for each location where usercode is defined. Click on a test case or test step to see the inherited stub function code for the selected test item.

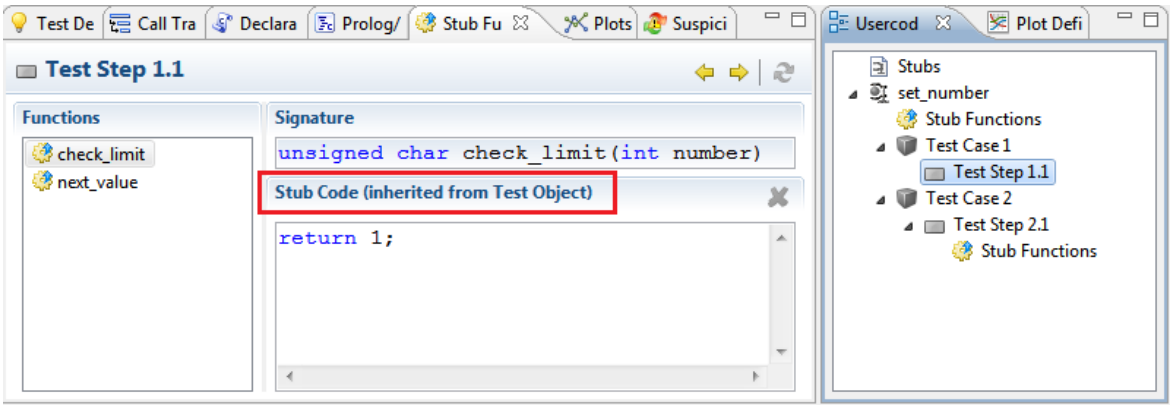


Figure 6.232: Usercode Outline view showing inherited stub code

The Stub Functions view shows the stub code to be executed for the test step 1.1 that is currently selected within the Usercode Outline view. Please note the hint within the text field title indicating that the stub code is inherited from the test object level.

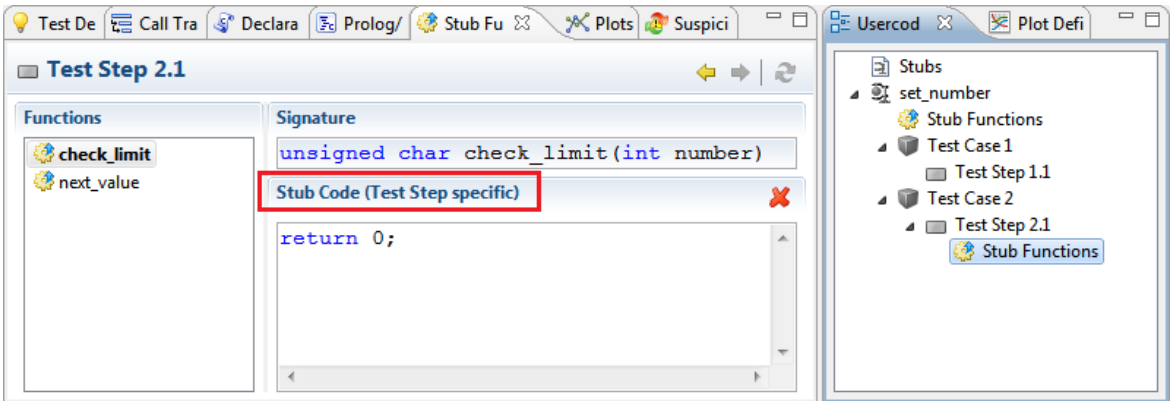


Figure 6.233: Usercode Outline view showing inserted stub code

Now there is an inserted stub function code entry selected within the Usercode Outline view. The entry indicates that the stub code is inserted for test step 2.1 which is also indicated within the text field title.

6.9.14 Plots view

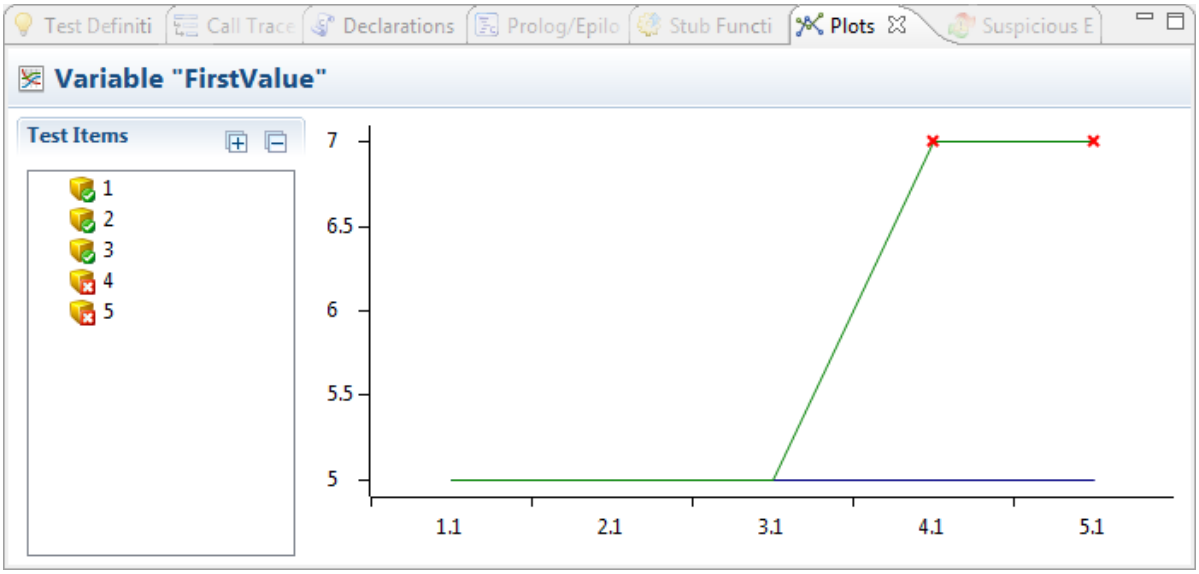


Figure 6.234: Plots view

The Plot view displays the included test items and chart(s) for a plot selected in the Plot Definitions view. The number of different charts per plot depends on the plot mode:

- test case plot: one chart for all included test items
- test step plot: one chart per included test case
- array plot: one chart per included test step

For test step and array plots the chart can be selected by navigating the test item tree.

Important: If you use other evaluation modes than equal (e.g. <, <=, >, >=, !=, [Range]), it is **not** possible to display the expected values within the plot chart. Displaying the expected values is only possible when using the evaluation mode == (equal). See section 6.9.7.6 Entering evaluation modes.

6.9.14.1 Test Item Tree

The test item tree on the left-hand side of the Plot view shows all test items included in the selected plot, as defined in the Plot Definitions view via the “Set Included Test Items” command.
This tree is for navigating the different charts of a plot, if there is more than one chart available.

6.9.14.2 Chart

The chart displays the values of the variables included in the selected plot. The values are color-coded:

- Yellow line: Input values
- Blue line: Expected values
- Green line: Actual values
- Red cross: Failed values

For expected values, dotted blue lines represent the upper and lower bound of expected values such as 10 ± 5.

Only variables that have “Use in Report” checked in the Plot Definitions view are shown in the chart. Selecting a variable in the Plot Definitions view will highlight the corresponding value series in the Plot view.

6.9.15 Plot Definitions view

The Plot Definitions view allows creating and configuring plots from within the TIE and TDE perspective. For details refer to section 6.7.5 Plot Definitions view within chapter TIE: Preparing the test interface.

6.10 Script Editor: Textual editing of test cases

The script perspective provides textual editors supporting a test scripting language for editing test cases, test data and usercode. The contents of the script editors show all information of the internal TESSY data model for test objects, test cases and test steps. Changes within the test scripts can be saved to the internal data model and vice versa. It is also possible to merge concurrent changes made within the internal data model and within the Script Editor.

Editing and saving test data and usercode

When working with the Script perspective, a new internally managed script file will be created for each test object. All test data and usercode can now be edited and saved within the editor. It is necessary to commit script changes to the internal data model when tests shall be executed. Until this point in time, the editor contents can just be saved to the underlying file. After committing the changes, the script editor contents and the internal model are in sync.

6.10.1 Editing test objects, test cases and test steps

Within the Script perspective, any selection of a test object will open or reveal the respective Script Editor. The Script Editor provides syntax highlighting, auto completion, formatting, validation and an outline view.

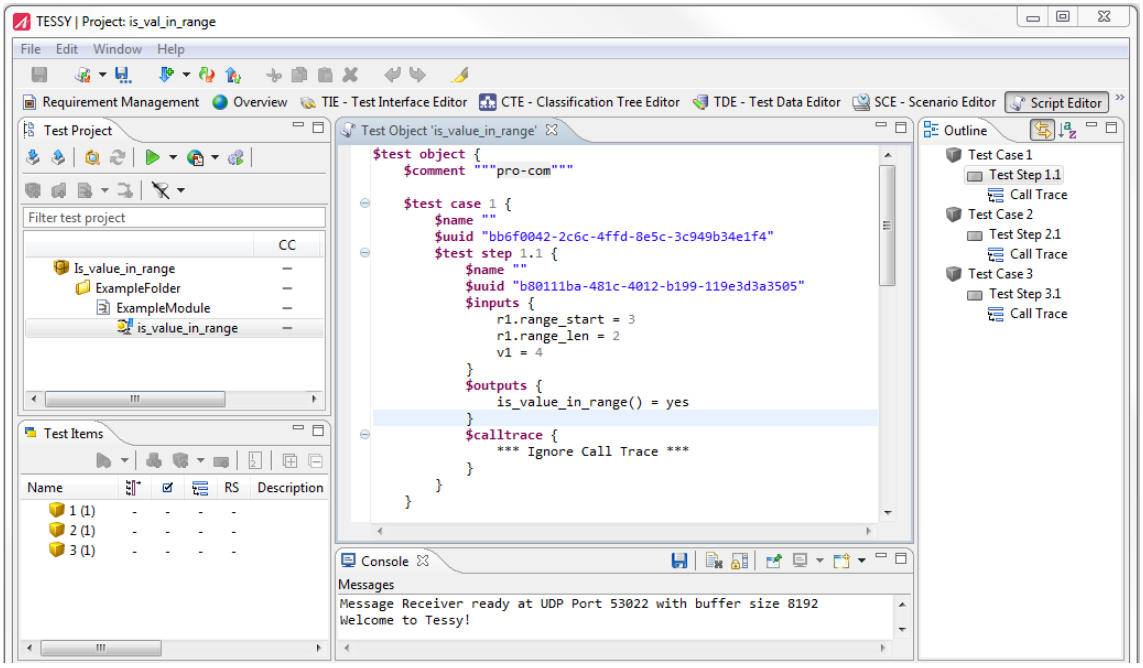


Figure 6.235: The Script Editor view.

The Script Editor also provides templates for test cases or other parts of the script using the CTRL+SPACE shortcut. This will show a menu containing context sensitive auto completions.

Test cases and test steps can be added by duplicating existing test items within the editor and adjusting the test item numbers. Duplicate UUIDs can be removed, they will be created when committing the changes to the model.

Alternatively, test items can be added or deleted using the Test Item view. The script will then be out of date or in conflict and will need to be merged. This can easily be done using the merge dialog if there are no really conflicting changes at the same locations.

6.10.2 Structure of the Script Editor perspective

Pane	**Location** (default)	**Function**
Test Project view	upper left	Same view as within the Overview perspective.
Test Items view	lower left	Same view as within the Overview perspective.
Script Editor area	upper middle	Displays the internally managed script file and allows editing and saving it.
Console view	lower middle	Same view as within the Overview perspective.
Outline view	right pane	Displays the script structure and allows locating script when linked with the Script Editor.

Table 6.78: Structure of the Script Editor

6.10.3 Script Editor related Icons of the main tool bar

Icon	Action / Shortcut / Comment
	Commits changes and saves the current editor contents to the internal TESSY data model. The script contents must be valid in order to do this synchronization.
	Merges changes and opens a merge dialog showing the current script and data model contents. If changes have been made both within the script and the internal model, it is necessary to merge both contents.
	Reloads from model and discards the whole contents of the editor. Builds a new script from the current internal model.
	Replaces the editor contents with the contents of the selected file.
	Saves the editor contents into the selected (new) file.

Table 6.79: Tool bar icons of the Script Editor perspective

6.10.4 Script states

Script changes are tracked

Changes to the script and the internal model are always tracked. If there are any deviations, the script editor window title will contain a status indicator prefix indicating possible modification states (see figure 6.236).

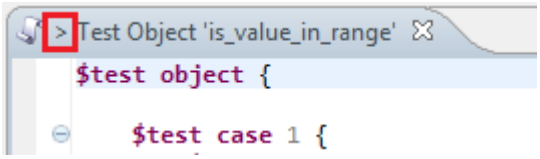


Figure 6.236: Status indicator within the editor title

The following script states are possible (a tooltip will be displayed when hovering over the editor title):

Indicator	Status / Meaning
>	MODIFIED / The script has been changed but the model is unchanged.

continue next page

Indicator	Status / Meaning
<	OUT OF DATE / The script is unchanged but the model has been changed.
!	CONFLICT / Both the script and the internal model have been changed concurrently.

Table 6.80: Status indicators of possible script states

Any conflicts can be resolved when committing the changes or by using the merge operation as displayed and described in section 6.10.7 Merging script contents.

6.10.5 The Script Editor Outline view

The Outline view in the Script Editor perspective displays the script structure and allows locating script items when linked with the Script Editor.

Icon	Action / Shortcut / Comment
	Links with the Script Editor.
	Sorts in alphabetical order.

Table 6.81: Tool bar icons of the Outline view in the Script Editor perspective

6.10.6 Synchronization with the internal model

The editor contents can be saved into the underlying script file while editing or when closing TESSY. In order to execute a test the editor contents need to be synchronized with the internal TESSY data model first. This can be done using the buttons in the global TESSY tool bar described in section 6.10.3 Script Editor related Icons of the main tool bar.

Synchronizing script

When committing script changes in conflict state, the merge dialog will automatically appear.

6.10.7 Merging script contents

If conflicting changes have been made within the internal model and the script file, the Merge Changes button can be used to merge both contents. This will show a merge dialog with both the model contents and the current script contents.Icons of the view tool bar

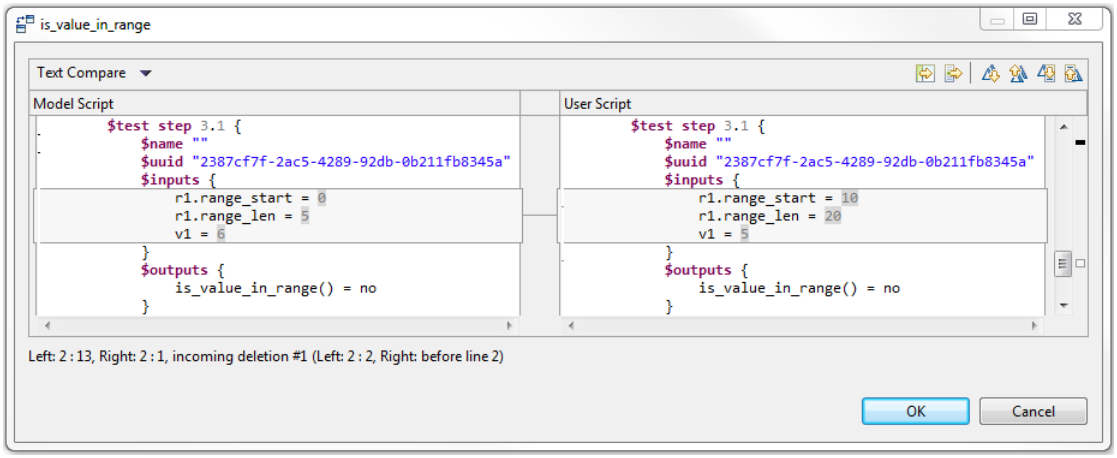


Figure 6.237: Merge dialog in the Compare view

The merge dialog above shows that the test data of the third test case has been changed within the script. It is easy to resolve the conflict using the buttons of the Compare view tool bar.

The merge dialog

After closing the dialog, the script editor will contain the results of the merge operation (i.e. the contents of the “User Script” text pane of the merge dialog). The script needs to be committed in order to save the contents to the internal model.

If the merge dialog was opened automatically during a commit operation, the content of the “User Script” pane will be used as content being saved to the internal model.

6.10.8 Importing and exporting script contents

Script content can be edited outside of TESSY using the save button and the replace button in the global TESSY tool bar (see section 6.10.3 Script Editor related Icons of the main tool bar).

Use to save the editor contents into the selected (new) file and to replace the editor contents with the contents of the selected file.

6.11 CV: Analyzing the coverage

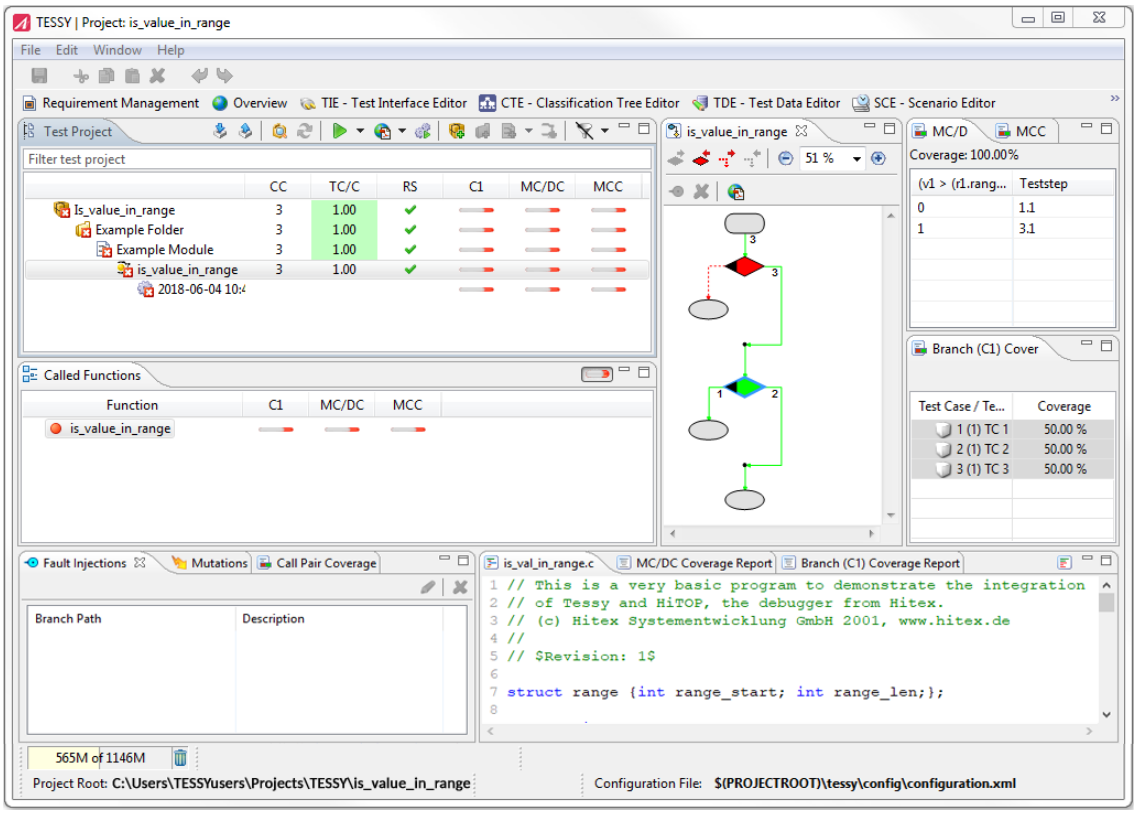


Figure 6.238: Perspective CV - Coverage Viewer

Coverage measurement

The Coverage Viewer (CV) displays the results of the coverage measurement of a previously executed test, which is either

- an overall instrumentation for your whole project, which you selected in the preferences menu of TESSY (see section 6.1.2.1 Window > Preferences menu), **or**
- an instrumentation which you selected within the Properties view for your module or test object under test (see section 6.2.3.4 Coverage tab), **or**
- an instrumentation which you selected for your test run (see section 6.2.2.11 Executing tests).

The available information displayed and the sub windows shown within the CV depends on the coverage options selected during the test run. The CV will be updated with the coverage information of the currently selected test object whenever you switch to the CV or when you select another test object within the Test Project view.

For more information about static analysis and quality metrics (Cyclomatic Complexity (CC)), Test Case To Complexity Ratio (TC/C) and Result Significance (RS)) please refer to subsection 6.2.2.7 Static code analysis and quality metrics.

6.11.1 Structure of the CV perspective

Pane	**Location** (default)	**Function**
Test Project view	upper left	Same view as within the Overview perspective.
Called Functions view	middle left	Contains the test object itself and the functions called from the test object.
Flow Chart view	upper middle	Displays the graphical representation of the control structure of the currently selected function.
Coverage views	upper/ middle right	Displays the results for the selected coverage instrumentation.
Fault Injections view	lower left	Displays the fault injections. (For more information see chapter 6.14 Fault injection.)
Call Pair Coverage view	lower left	Displays the call pair measurements (CPC).
Code view	lower right	Displays the source code of the currently selected function (and highlights selected decisions/branches).
Report views	lower right	Displays the ASCII based coverage summary reports for the selected instrumentation.

Table 6.82: Structure of CV

6.11.2 Instrumentation for coverage measurements

TESSY supports the following coverage measurements:

- C0 (Statement Coverage)
- C1 (Branch Coverage)
- DC (Decision Coverage)
- MC/DC (Modified Condition / Decision Coverage)
- MCC (Multiple Condition Coverage)
- EPC (Entry Point Coverage) - only for unit tests
- FC (Function Coverage) - only for component tests
- CPC (Call Pair Coverage)

For more information about coverage measurements and usage of coverage analysis refer to the application note “Coverage Measurement” in TESSY (“Help” > “Documentation”).

There are no views for the Entry Point Coverage (EPC) and the Function Coverage (FC)! The results are displayed only within the Test Overview Report (see section 6.2.2.16 Creating reports) or the Test Project view (see figure 6.239).

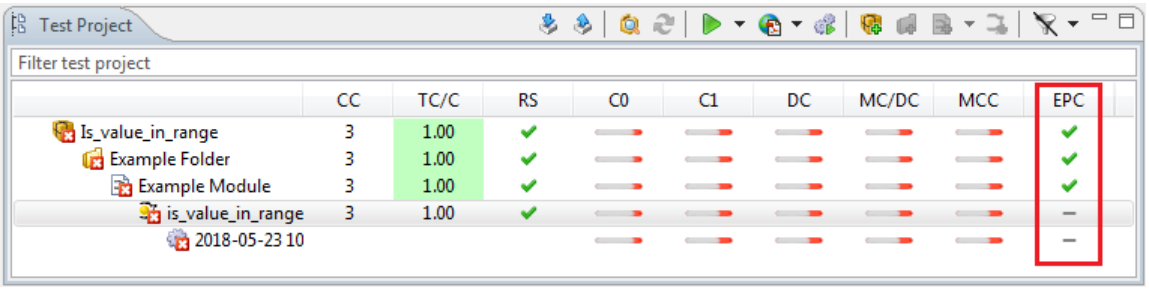


Figure 6.239: Results of the EPC are displayed within the Test Project view

The following figure 6.240 displays a component test with a Function Coverage instrumentation result (amongst others).

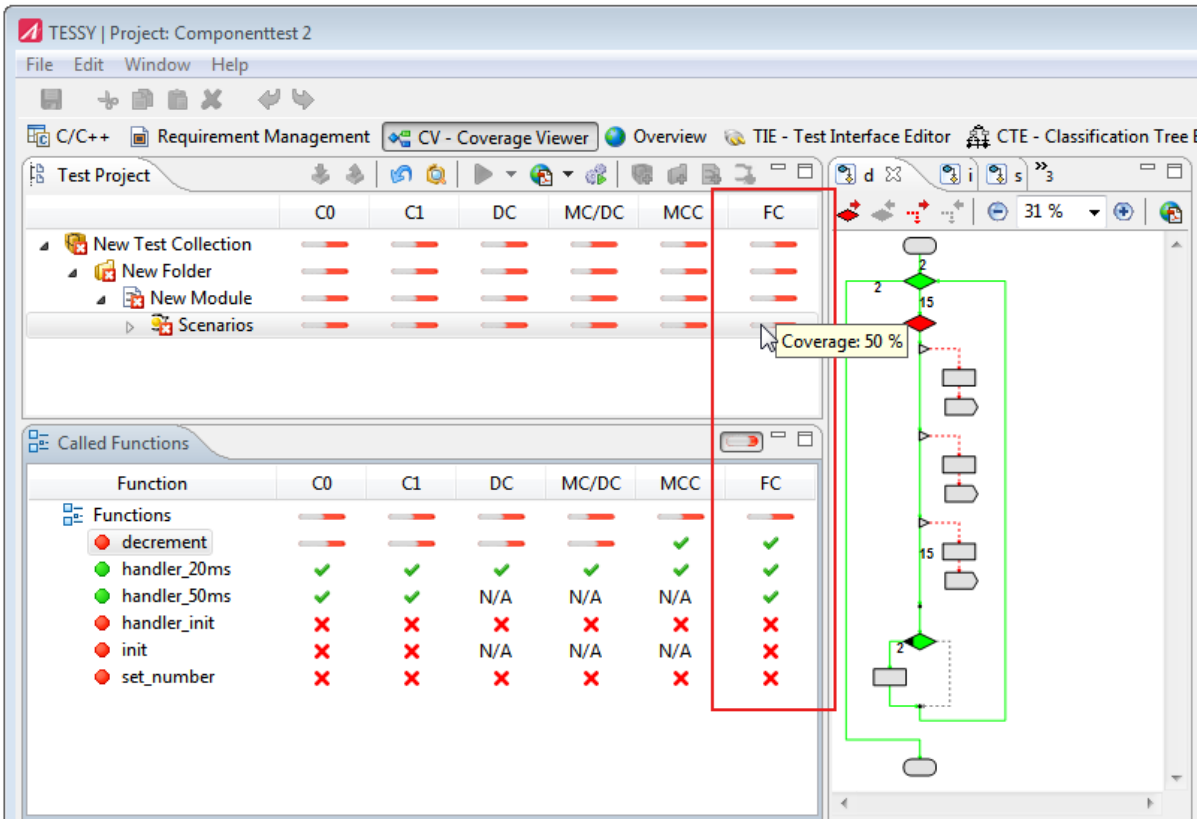


Figure 6.240: Coverage results within the CV perspective (component testing)

Please notice:

- If you move the mouse over the result within the Test Project view, the percentage of the coverage for the respective item will be displayed.
- The Called Functions view displays the coverage result for every function.

6.11.3 Test Project view

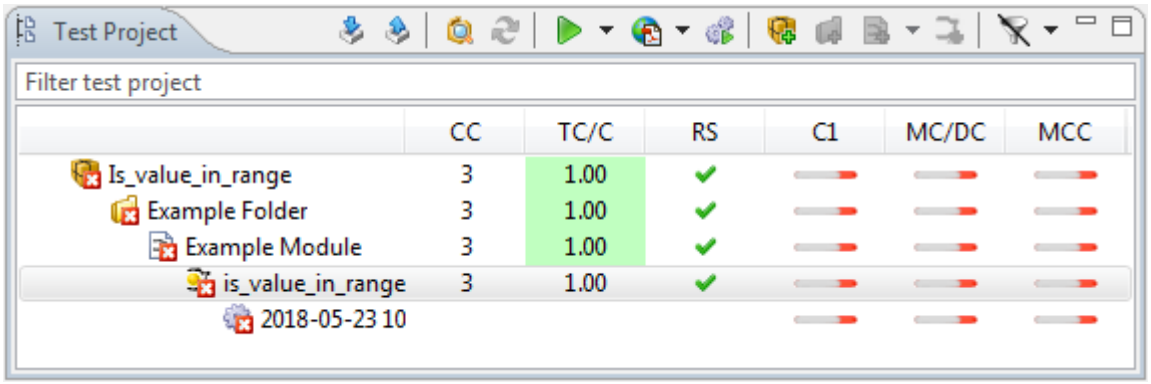


Figure 6.241: Test Project view within the CV perspective

The Test Project view displays your test project which you organized within the Overview perspective.

6.2.2 Test Project view

Important: We recommend to do any changes of the test project structure within the Test Project view of the Overview perspective. The view layout of this perspective is optimized for this purpose!

After a test run you will see columns being added to the Test Project view for each applied coverage measurement. The coverage icons provide an overview about the reached coverage for each test object as well as cumulated for modules, folders and test collections.

6.11.4 Called Functions view/Code view

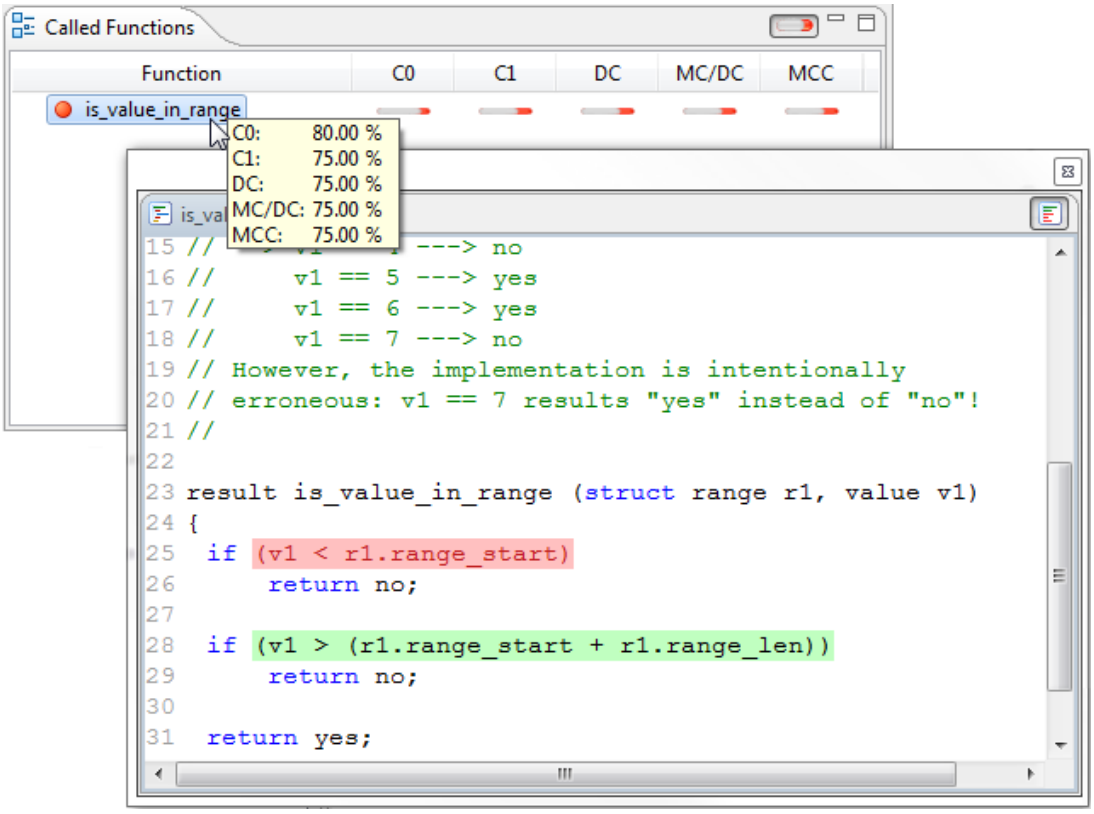


Figure 6.242: Called Functions view

The Called Functions view contains the test object itself and all called functions of the test object. It displays the achieved coverage of the current test run. By clicking on a function, you can review the source code within the Code view and see the code structure within the Flow Chart view.

To highlight the source code within the code view:

→ Click on (Toggle Code Coverage Highlighting) in the tool bar of the Code view. The statements, branches or conditions of the source code will be marked within the Code view according to the selected coverage measurement. If the respective code location have been covered successfully, i.e. 100% coverage has been reached for this code part, it will be marked in green. Otherwise the code location will be marked in red indicating that it has not been fully covered.

To **highlight a specific code location**:

→ Select an element within the Flow Chart view.
The respective source code lines will be marked within the Code view.

To have a quick **overview of the coverage**:

→ Within the Called functions view move the mouse over the function.
All the coverages will be displayed (see figure 6.242).

6.11.5 Flow Chart view

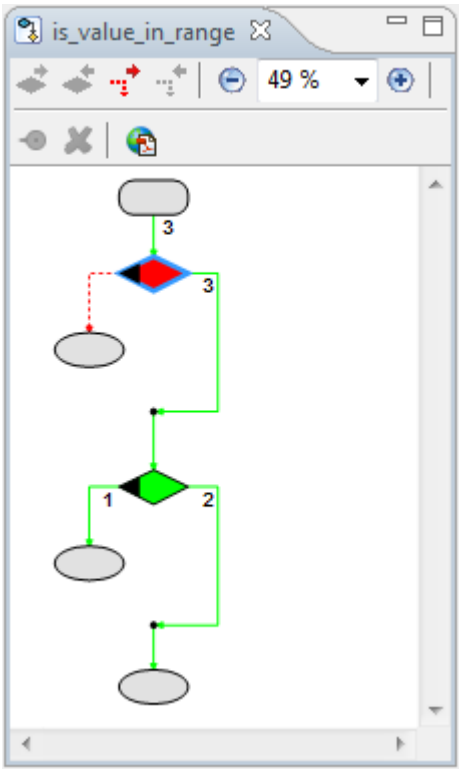


Figure 6.243: Flow Chart view

Coverage graphically displayed

The Flow Chart view displays the code structure and the respective coverage in graphical form.

You might want to learn the functions of the Flow Chart view with an easy example: Consult section 5.1.10 Analyzing the coverage of the Tutorial: Practical exercises.

6.11.5.1 Icons of the view tool bar

Icon	Action / Shortcut / Comment
	Searches the next uncovered decision.
	Searches the previous uncovered decision.
	Searches the next unreached code branch.
	Searches the previous unreached connection.
	Zooms out.
	Zooms in.
	Creates and edits fault injections.
	Deletes fault injections.
	Generates chart report.

Table 6.83: Tool bar icons of the Test Items view

More information about fault injections is provided in chapter 6.14 Fault injection.

6.11.5.2 Viewing functions

To display a flow chart of a function:

→ Click on a function within the Called Functions view.
The code structure of the function will be displayed in the Flow Chart view.

Zoom in or out using the tool bar icons or the entries from the chart menu.

Coloring within the Flow Chart view

Within each flow chart, you will see the branches and decisions of the function being displayed in green and red colors, which indicates whether the respective decision has been fully covered or the respective branch has been reached:

- green: 100 % coverage
- red: less than 100 % coverage; a branch or at least one condition combination was not executed
- gray: None of DC, MC/DC or MCC coverage has been selected for the last test execution. (Decision elements are still selectable in order to find the respective line of code in the source code view (see figure 6.244)).

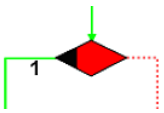
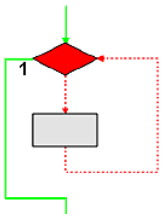
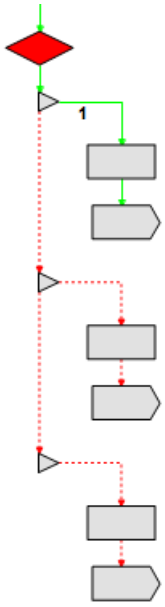
Important: Please not: The coloring of branches according to statement and branch coverage depends on the selected test cases and/or test steps within the C0/C1 Coverage views (see subsections 6.11.7 Statement (C0) Coverage view and 6.11.8 Branch (C1) Coverage view). The coloring of decision symbols is always based on the results of all test cases.
If both C0/C1 and one of DC, MC/DC, MCC coverage have been selected and not all test items are selected within the C0/C1 Coverage views, this may lead to inconsistent coverage coloring.

is_val_in_range.c | DC Coverage Report | MCC Coverage Report

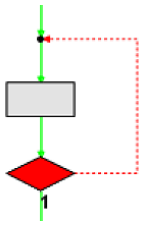
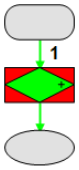
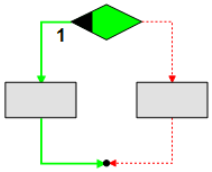
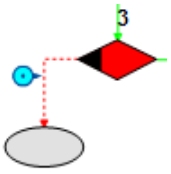
```
1  // This is a very basic program to demonstrate the integration
2  // of Tessy and HiTOP, the debugger from Hitex.
3  // (c) Hitex Systementwicklung GmbH 2001, www.hitex.de
4  //
5  // $Revision: 1$
6
7  struct range {int range_start; int range_len;};
8
9  typedef int value;
10
11 typedef enum {no, yes} result;
12
13 // Checks if r1.start <= v1 < (r1.start + r1.len),
14 // e.g. r1.start = 5, r1.len = 2
15 // ==> v1 == 4 ---> no
16 //     v1 == 5 ---> yes
17 //     v1 == 6 ---> yes
18 //     v1 == 7 ---> no
19 // However, the implementation is intentionally
20 // erroneous: v1 == 7 results "yes" instead of "no"!
21 //
22
23 result is_value_in_range (struct range r1, value v1)
24 {
25  if (v1 < r1.range_start)
26      return no;
27
28  if (v1 > (r1.range_start + r1.range_len))
29      return no;
30
31  return yes;
32
33 }
34
```

Figure 6.244: Source code view on the bottom right of the Coverage View perspective

The following elements are displayed within the flow chart of the CV:

Element	Meaning
	**If-else decision** The if branch on the left side was reached once, the else branch on the right side was not reached. The decision was not fully covered.
	**For or while loop** The loop body was not reached, instead the exit out of the loop was executed once. The loop decision was not fully covered.
	**Switch statement** The first case branch was reached once, the second case branch was not reached. The default branch was also not reached.

continue next page

Element	Meaning
	**Do while loop** The loop body was only reached once (without repeated execution of the loop body branch) and the exit branch was reached. The loop decision was not fully covered.
Main flow  Condition flow 	**Sub flows** Main flow Statements and decisions containing sub flows (e.g. the “?” operator) are displayed with a special symbol in which a square and a rhombus are merged together. The coloring of this symbol displays the coverage of both the decision as well as the sub flow. In this example the square is red as the sub flow was not fully covered. (Only one test step was selected within the C1 Coverage view in this case.) The rhombus however is green as the condition was fully covered. (Based on the result of all executed test cases). Condition flow The detailed structure of a sub flow will be shown within a separate flow chart. Such a Condition view opens after double-clicking the respective element (see figure 6.245).
	**Fault injection** In this branch a fault injection was created.

continue next page

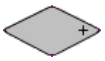
Element	Meaning
	**Plus symbol** Indicates a sub flow, also appears in combination with green or red coloring.
	Black triangle Indicates an if-else decision, also appears in combination with green or red coloring.

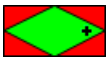
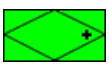
Table 6.84: Elements of the Flow Chart view

Viewing sub flows

Sub flow elements (e.g. the “?” ternary operator) are displayed with a special symbol which can appear in different colors according to the general coloring rules within the Flow Chart view, green for 100% coverage and red for less than 100% coverage.

It needs to be kept in mind that this flow chart symbol may appear in one color or in two different colors in different positions according to the coverage of the sub flow. The various coloring options display the coverage status of the individual test. It has to be interpreted on the basis of the special symbolizing of sub flows.

Possible coloring of sub flow elements within the flow chart:

Element	Meaning
	Showing that the decision is fully covered (inner part of the symbol, colored in green, 100%) while the sub flow is not fully covered (outer part of the symbol, colored in red, less than 100%). Coverage details can be seen in the Condition view after double-clicking the respective element in the Flow Chart view (see figure 6.245).
	Showing that the decision of the condition (inner part of the symbol) as well as the coverage of the sub flow (outer part of the symbol) is fully covered (100%).

continue next page

Element	Meaning
	Showing that neither the decision (inner part of the symbol) nor the sub flow (outer part of the symbol) are fully covered (less than 100%).

Table 6.85: Sub flow coloring in the Flow Chart view

Sub flows can appear in decisions as well as in statements.

Condition view

Double-clicking or right-clicking a sub flow element opens the Condition view to show the coverage details of the respective element (see figure 6.245). Selecting test steps in the Branch (C1) Coverage view displays the coverage of the selected test steps in the Condition view.

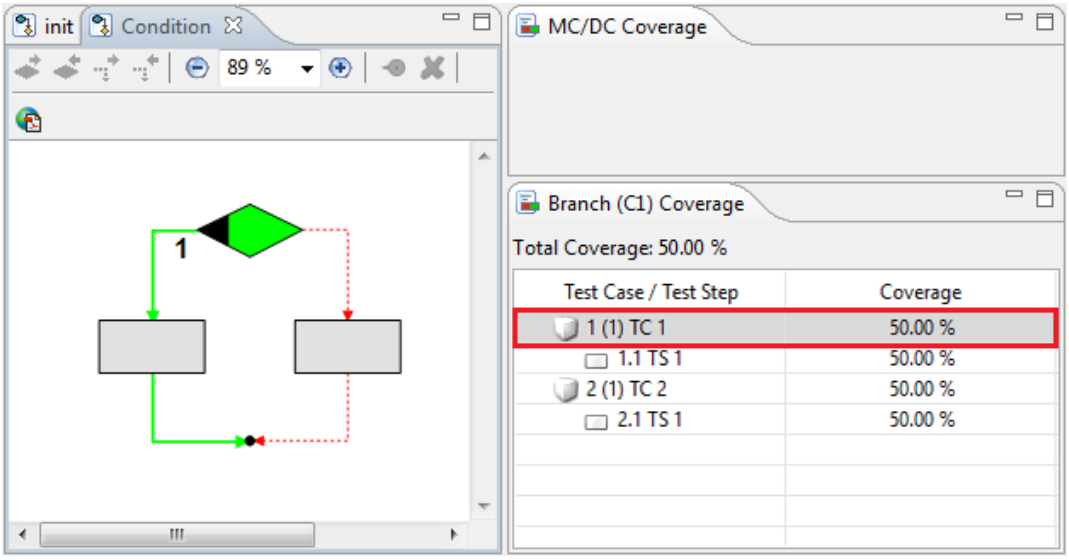


Figure 6.245: Condition view showing the sub flow coverage for one test case

6.11.5.3 Selecting elements

You can select decisions, branches and code statement elements within the flow chart. The respective code section will then be highlighted within the source code view. Since not all

connection lines within the flow chart are branches in terms of the C1 branch definition, some of the connection lines may not be selectable.

If a condition or a code element contains sub flows (e.g. the “?” ternary operator or statements containing boolean expressions) you can visualize the sub flow with a double click on the respective element. CV will open a new flow chart showing the sub flow.

You may also want to select elements to create fault injections (see chapter 6.14 Fault injection).

6.11.5.4 Searching for uncovered decisions and unreached branches

The CV provides search functionality for decisions and branches that are not fully covered respectively reached through all the executed test cases. The decisions and branches are already marked in red, but the search function can assist in finding all uncovered decisions or unreached branches.

→ Select the respective icon from the tool bar (i.e.).
The chart will change into the search result mode, marking the found element in blue.

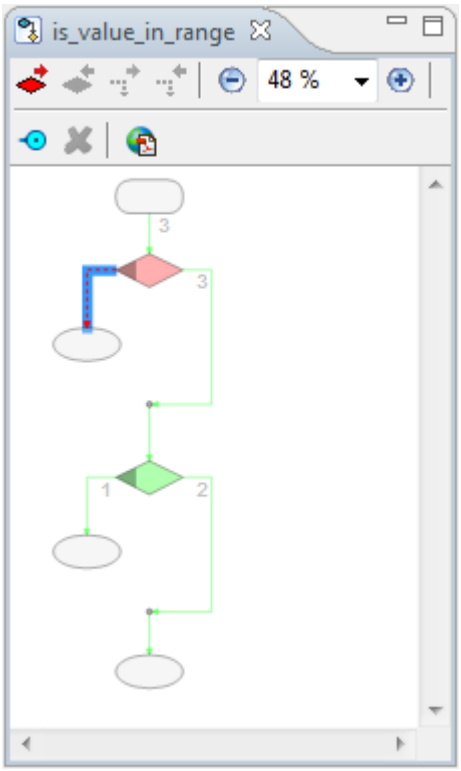


Figure 6.246: Unreached code branch is marked blue

6.11.6 Fault injection

The fault injection feature provides means to test code parts that are not testable using normal testing inputs e.g. endless loops, read-after-write functionality or error cases in defensive programming. Dedicated testing code can be injected at selected branch locations of the test object so that decision outcomes can be manipulated.

6.14 Fault injection

Fault injections are edited within the coverage viewer (CV) based on the flow chart of the test object. They are displayed as blue circles at the respective branch.

Fault injections that cannot be mapped to the current source code control flow are marked with an error symbol within the “Fault Injections” view on the lower pane on the left.

For more information about the handling of fault injections please refer to chapter 6.14 Fault injection.

6.11.7 Statement (C0) Coverage view

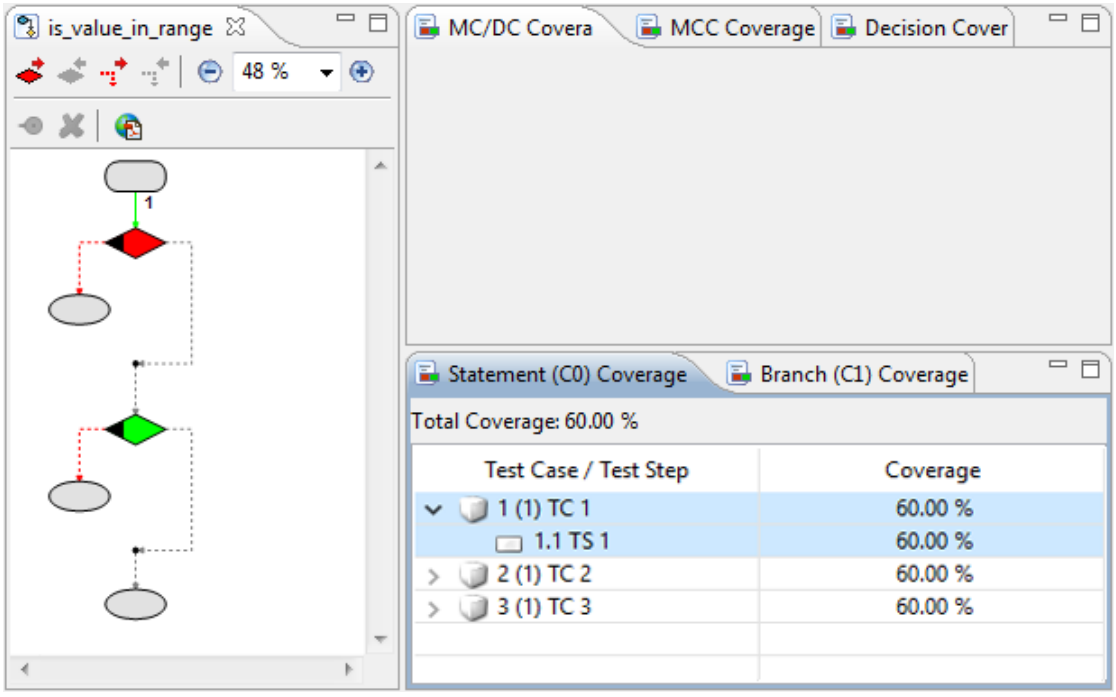


Figure 6.247: Statement coverage

The Statement (C0) Coverage view displays the statement coverage for each individual test case and test step as well as the total coverage for all test cases (and test steps). The

coverage information in this view is calculated for the selected function within the Called Functions view.

If you only selected the C0 coverage instrumentation for test execution, you will see the code branches marked in red and green within the flow chart; “else” branches, that do not exist within the code, will be displayed in the Flow Chart view in gray.

Also the loop branches of while, for and do statements that are irrelevant for C0 coverage will be displayed in gray.

The flow chart shows code branches and not individual statements and also blocks of statements will be shown as one block instead of individual items for each statement.

If you select individual test cases or test steps within the test case list, the respective statements covered by those test steps will be marked within the flow chart, i.e. the code branch containing these statements will be marked. This allows finding out the execution path of the selected test step. By selecting multiple test steps, review the resulting cumulated statement coverage within the flow chart. The total coverage number will also be updated with the C0 statement coverage for the selected test cases/test steps.

6.11.7.1 Coverage percentage

The coverage percentage is the relation between the total numbers of statements of the currently selected function compared to the number of reached statements. This coverage calculation includes the currently selected test cases and test steps within the test case/test step list (see figure 6.247). By default, all test cases are selected when opening the CV.

6.11.8 Branch (C1) Coverage view

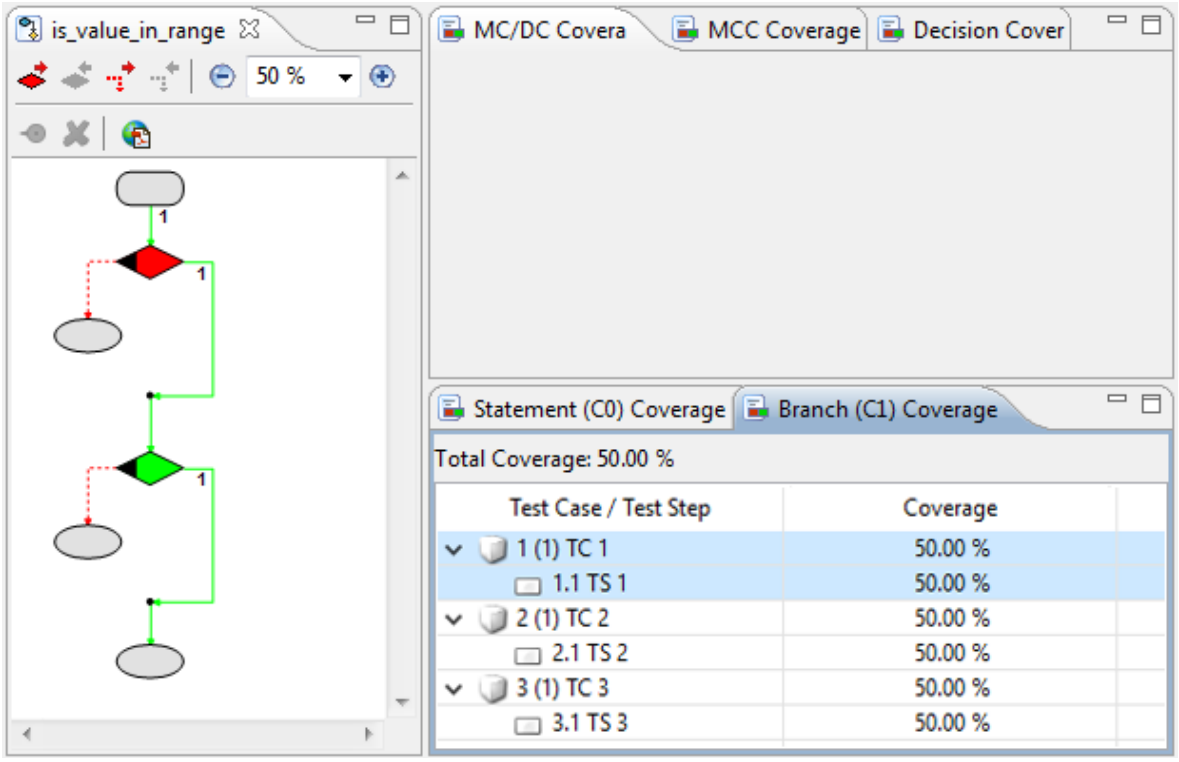


Figure 6.248: Branch coverage

The Branch (C1) Coverage view displays the branch coverage for each individual test case and test step as well as the total coverage for all test cases (and test steps). The coverage information in this view is calculated for the selected function within the Called Functions view.

If you only selected the C1 coverage instrumentation for test execution, you will see only the C1 branches marked in red and green within the flow chart.

If you select individual test cases or test steps within the test case list, the respective branches covered by those test steps will be marked within the flow chart. This allows finding out the execution path of the selected test step. By selecting multiple test steps, review the resulting cumulated branch coverage within the flow chart. The total coverage number will also be updated with the C1 branch coverage for the selected test cases/test steps.

6.11.9 Decision Coverage view

To understand the Decision Coverage view please refer to the description of the MC/DC Coverage view below. The only difference is the calculation according to the definition of the decision coverage.

6.11.10 MC/DC Coverage view

The MC/DC Coverage view displays the coverage of the currently selected decision within the Flow Chart view (see figure 6.249). If no decision is selected (as initially when starting the CV), the MC/DC Coverage view is empty.

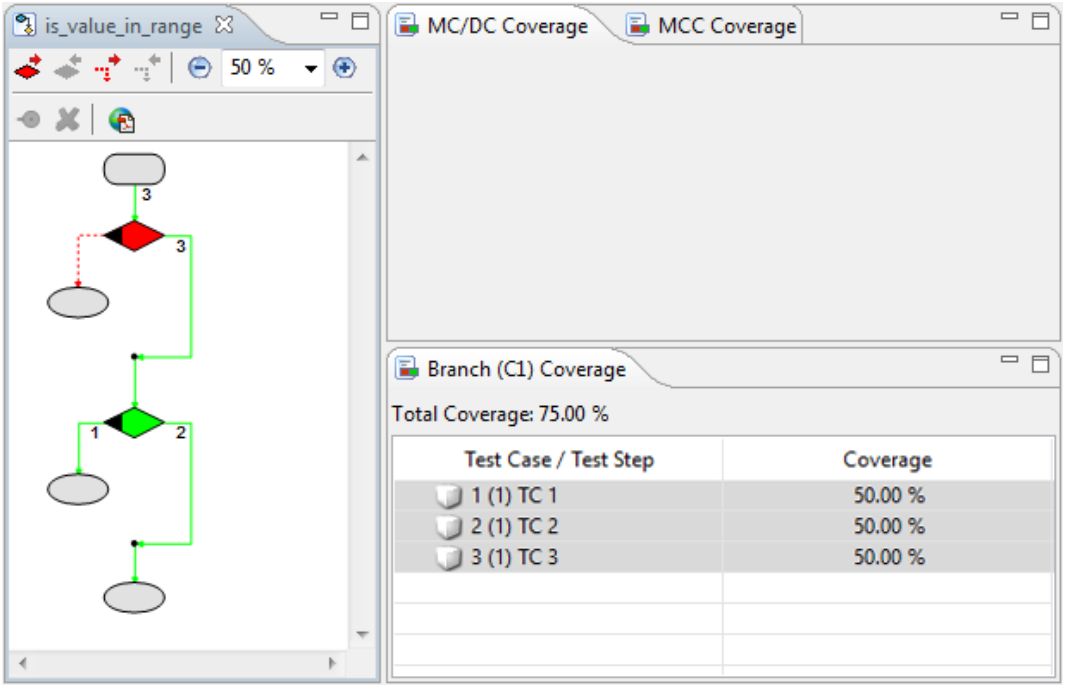


Figure 6.249: MC/DC Coverage view

When selecting a decision, the respective combination table according to the MC/DC coverage definition will be displayed within the MC/DC-Coverage view (see figure 6.250).

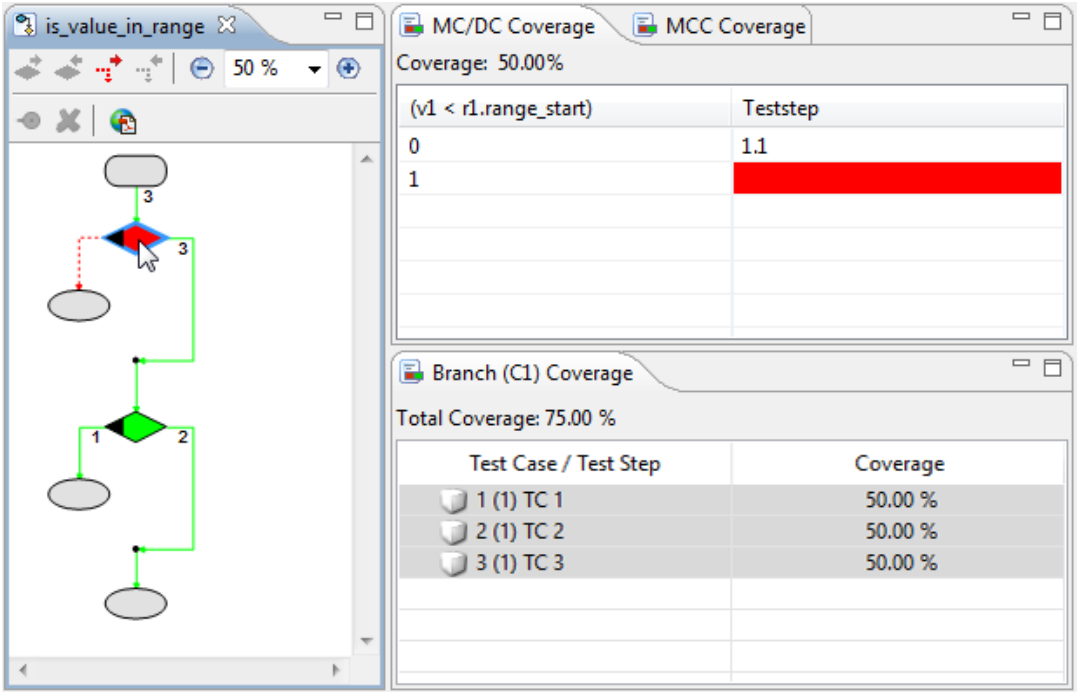


Figure 6.250: MC/DC coverage

The combination table contains all atomic conditions of the decision. The conditions are the basic atoms of the decision which remain after removing the **or**, **and** and **not** operators from the decision. TESSY calculates the MC/DC set of true/false combinations of the condition atoms that fits best to the test steps executed during the test run.

The last table column contains the test step that caused the execution of the decision with the true/false combination of the respective table row. If one or more of the condition combinations were not reached during the test run, the test step column of those rows will be marked in red.

6.11.10.1 Selecting decisions

→ Select a decision by clicking on the respective control flow element within the Flow Chart view.
The code fragment will be marked within the source code view (see figure 6.250).

The decisions are either green or red depending on the degree of coverage. If no coverage information is available, i.e. when you ran the test without any of DC, MC/DC or MCC instrumentation selected, the decisions within the flow chart will appear in gray and the Coverage view will not be available (N/A).

6.11.11 MCC Coverage view

To understand the MCC Coverage view please refer to the description in section 6.11.10 MC/DC Coverage view. The only difference is the calculation according to the definition of the MCC coverage.

6.11.12 Call Pair Coverage view

The call pair coverage measurement (CPC) supports measuring whether all call locations of functions or methods within the test object have been exercised at least once. It fulfills the requirements of ISO 26262 as an alternate coverage method for integration testing instead of applying the function coverage (FC) method.

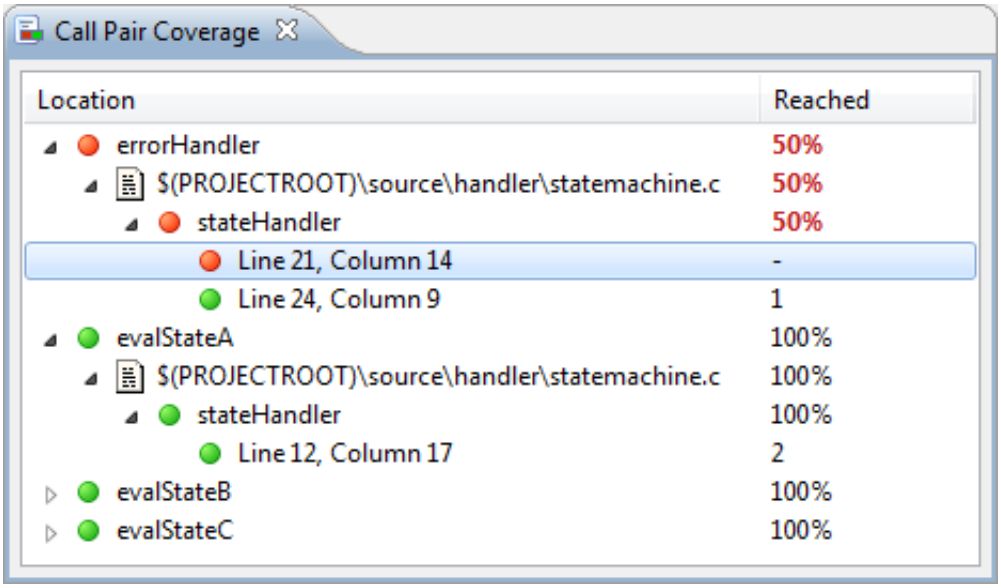


Figure 6.251: Call pair coverage view with coverage results

6.11.13 Coverage Report views

There are up to five coverage reports available depending on the instrumentation mode selected for test execution. They contain the summarized coverage information of the last test execution:

- The statement (C0) coverage report contains some meta information (e.g. number of statements, reached statements, total statement coverage) and the source code of the test object.
- The branch (C1) coverage report contains some meta information (e.g. number of branches, reached branches, total branch coverage) and the source code of the test object.
- The decision coverage (DC) report lists all decisions of the test object code including the coverage tables with the respective decision condition combinations.
- The modified condition/decision (MC/DC) coverage report lists all decisions of the test object code including the coverage tables with the respective MC/DC condition combinations.
- The multiple condition (MCC) coverage report also lists all decisions of the test object code including the coverage tables with the respective MCC condition combinations.

6.12 IDA: Assigning interface data

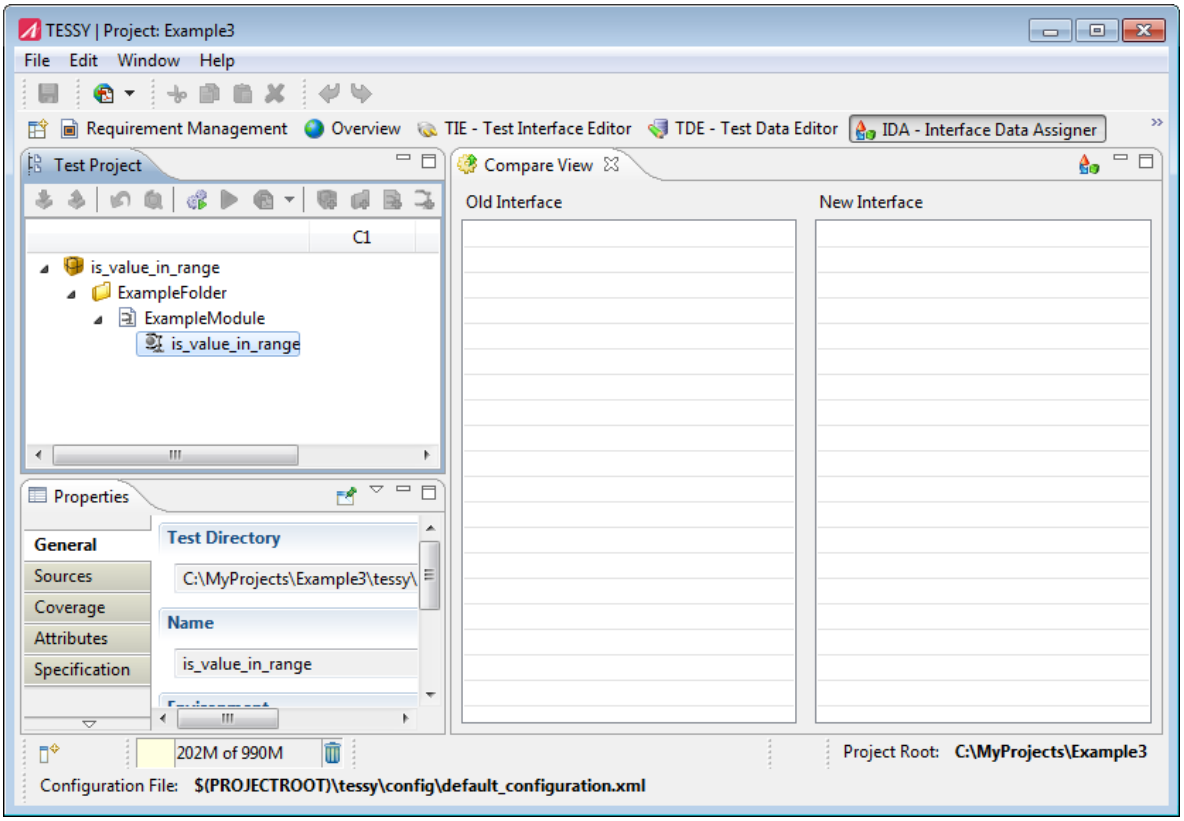


Figure 6.252: IDA perspective

For coherent testing it is essential to realize changes of the interface of test objects and to re-execute previously passed tests to assure that any changes of the source do not cause the previous passed tests to fail. This is often summed up with the keywords “regression testing”.

If the interface of a test object changes, TESSY will indicate the changes with specific status indicators at the test object icon. With the Interface Data Assigner (IDA) you can assign the elements of a changed (new) interface to the respective elements of the old one and start a reuse. The reuse operation will copy all existing test data to the newly assigned interface elements.

To appropriately react to changes, TESSY needs to know the current structure of the interface. Therefore it determines for each module the available functions and their interfaces by analyzing the source code. This information is stored in the interface database so that TESSY knows about any changes and can keep track of test data assignments made for a whole module or just for individual test objects.

6.12.1 Structure of the IDA perspective

Pane	**Location** (default)	**Function**
Test Project view	upper left	Displays your test project. For editing your test project switch to the Overview perspective.
Properties view	lower left	Displays the properties of your test project, e.g. sources to the test object.
Compare view	right	Displays two interfaces, either of the same test object (old and new interface) or of different test objects. You can assign the changes by drag & drop.

Table 6.86: Structure of the IDA perspective

6.12.2 Status indicators

The following test object status indicators are relevant when reusing test data.

Indicator	**Status / Meaning**
	The test object has **changed**. You see these test objects, but there is no operation possible. You have to start a reuse operation.
	The test object is **newly** available since the last interface analysis. You have to add test cases and test steps and enter data for a test.

continue next page

Indicator	Status / Meaning
	The test object has been **removed** or renamed. You still see these test objects, but there is no operation possible. You have to assign this test object to any other and start the reuse operation.

Table 6.87: Status indicators of test objects

6.12.3 Test Project view

The Test Project view displays your test project which you organized within the Overview perspective.

6.2.2 Test Project view

Important: We recommend to do any changes of the test project structure within the Test Project view of the Overview perspective. The view layout of this perspective is optimized for this purpose!

6.12.4 Properties view

The Properties view displays all the properties which you organized within the Overview perspective. Most operations are possible.

6.2.3 Properties view

For changing a source switch to the Properties view within the Overview perspective.

6.12.5 Compare view

The Compare view shows two versions of an interface depending on the TESSY objects selected for comparison:

- For a single module or test object, it shows the old interface on the left side and the new interface on the right side.
- When assigning two different modules or test objects, it shows the interface of the source object on the left side and the interface of the target object on the right side.

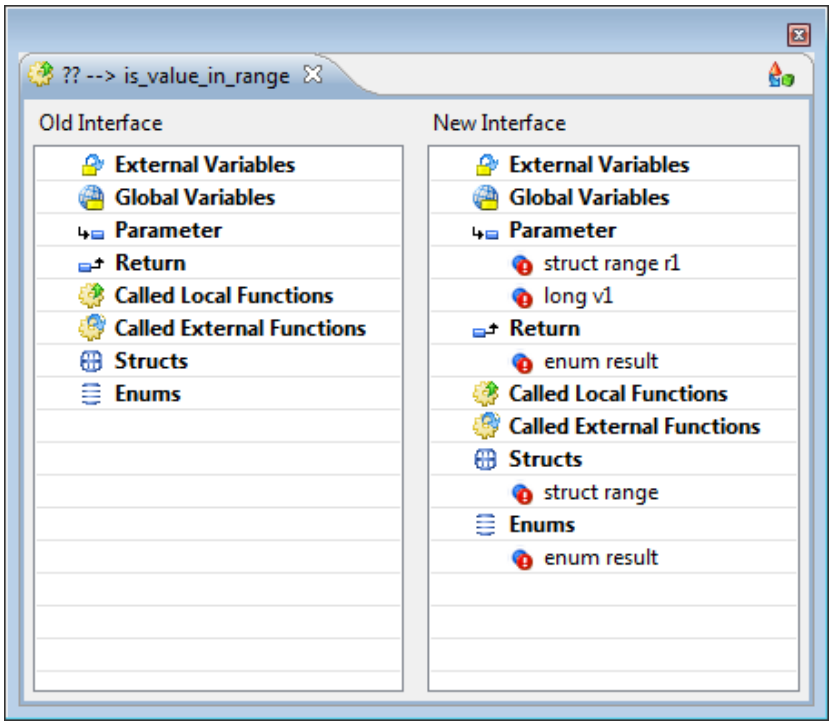


Figure 6.253: Compare view

The Compare view will be used for reuse operations of whole modules or individual test objects as well as when assigning test data from one test object (or module) to another test object of the same or different module.

6.12.5.1 Comparing interfaces and assigning changes

Within the Compare view you can see the old interface of our test object and the new one. The red exclamation mark within the new interface indicates the need to assign this interface object before starting the reuse.
The title of the view shows the old name versus the newly assigned name.

To assign changes:

→ Use the context menu or just drag and drop from the left side (see figure 6.254). The red exclamation mark turns to a green check ✔.

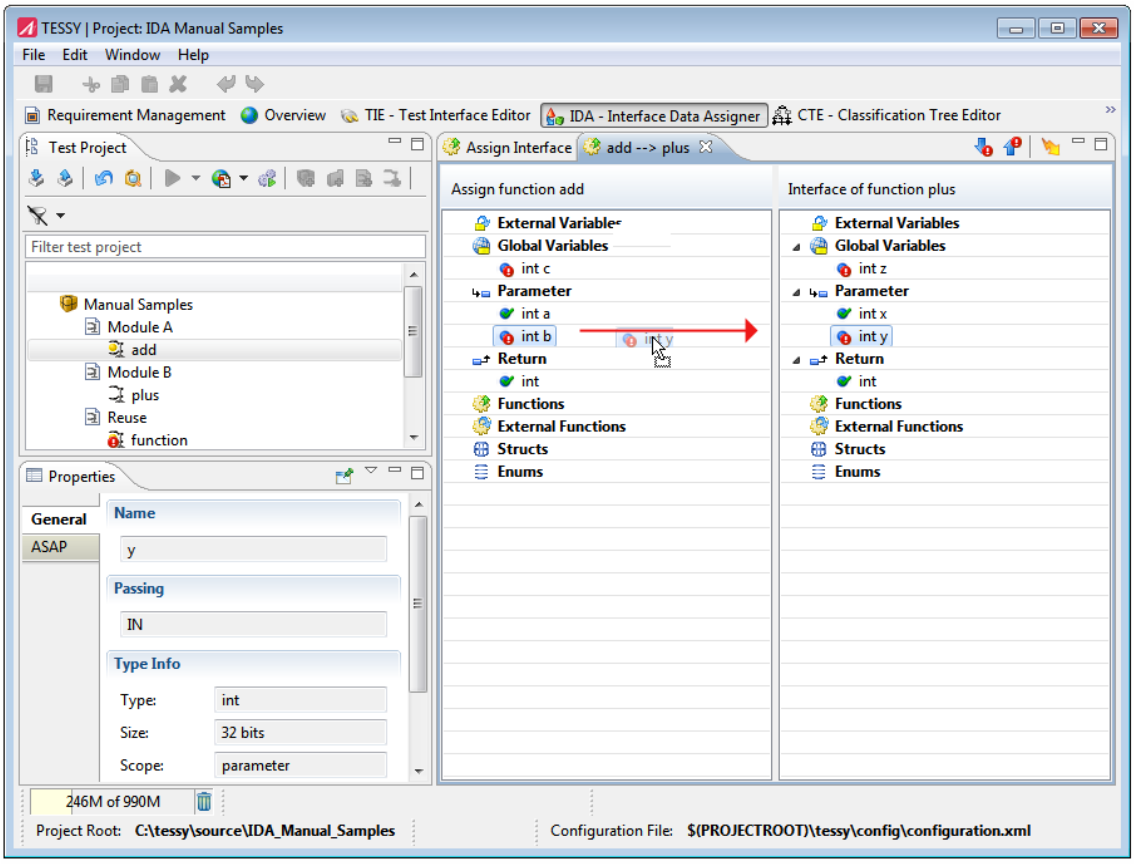


Figure 6.254: Use drag and drop in IDA

You can assign single functions and just commit the assignments for this function (the other functions will stay in state “changed” and can be reused later). Or you can assign and reuse whole modules (which is convenient when there are just little changes within the function interfaces.

To commit assignments:

→ Click on (Commit) in the menu bar of the Compare view.

The data of all test cases and test steps will be copied from the old interface to the current test object interface. The test object changes to yellow to indicate that all test cases are ready to be executed again. If there are missing test data within the new interface (e.g. due to additional variables being used by the test object), the icon will show an intermediate test object state . In this case you need to add any missing test data within the Test Data Editor.

Please notice the following habits:

- Removed and changed test objects require a reuse operation before you can further operate on them.
- Unchanged test objects have been automatically reused when opening a module, i.e. they will be ready to use without further activities required.
- Removed test objects will only be displayed as “removed”, if they did contain any test cases and test steps.

6.12.5.2 Assigning test cases to other test objects

You can use the IDA to assign test cases from one test object to another test object within the current project. Both test objects can be either from the same or from different modules. It is also possible to assign the contents of whole modules to other modules.

Important: When assigning test cases to another test object, the target test object contents will be overwritten completely!

To assign test cases to another test object:

→ Change to the IDA perspective.
→ At first drag the target test object into the right side of the Compare view (or placeholder view).
→ Secondly drag the source test object into the left side of the Compare view.
→ Assign the interfaces to your needs. Variables that cannot be assigned can be left out of scope (they will just not be used). Additional variables of the target test object that cannot be assigned from the source test object will be left empty after the assignment.

To commit assignments:

→ Click on (Commit) in the menu bar of the Compare view.

The data of all test cases and test steps will be copied from the source test object to the target test object. The target test object changes to yellow if every variable of the interface could be assigned from the source test object. Otherwise it will display an intermediate test object state indicating that only parts of the test data are available.

6.13 SCE: Component testing

The component test feature is only used for integration testing. You do not need this feature for unit testing.

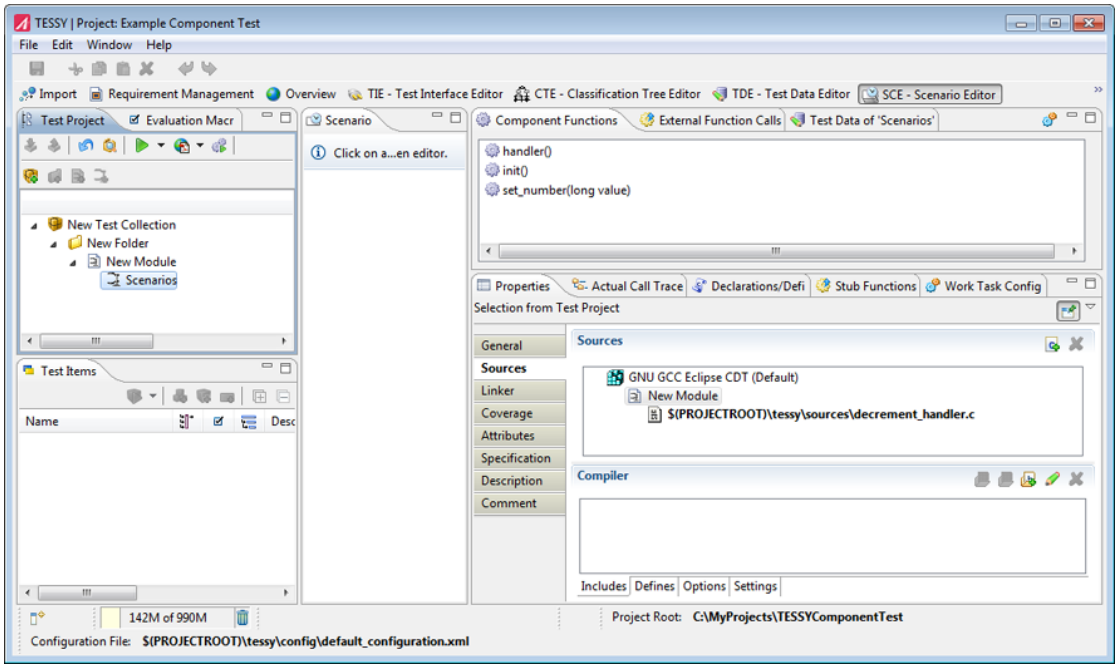


Figure 6.255: Perspective SCE - Scenario Editor

The component test feature within TESSY supports testing of several functions (representing the software component) that interact with themselves as well as with underlying called functions (of other components). The main difference to unit testing of individual functions is the focus of testing on the external interface of the component instead of internal variables or control flow. You should be familiar with the overall usage of TESSY for the normal unit testing. Some features like usercode and stub functions are still available for component testing, but the respective editors will be found at different locations.

The component test feature allows creating calling scenarios of functions provided by a software component. Within these scenarios, the internal values of component variables and any calls to underlying software functions can be checked. TESSY provides the Scenario Editor (SCE) for this purpose. All scenario-related inputs are available through the SCE. Instead of having individual test objects and test cases for the component functions, the component test itself provides a special node called “scenarios” seen as one test object. The test cases belonging to the scenarios node are the different scenarios for the component.

Within one scenario, you can set global input variables, call component functions, check the calling sequence of underlying software functions and check global output variables.

The content of each scenario may be divided into the following parts:

- setting the input variables
- calling component functions
- checking calls to underlying functions
- setting/checking variables during scenario execution
- executing usercode and eval macros
- checking the output variables

The Usercode Editor (UCE) is not available for component testing, because the prolog/epilog code and definitions/declarations sections can be edited directly within the SCE. You will find C-code fragments that can be added into the scenario control flow. Also the code for stub functions can be edited directly within SCE.

6.13.1 Creating component tests

The component test management is based on TESSY modules alike a unit test. In contrary to unit testing you will probably use multiple source files instead of only one file. Other parts of the testing process stay basically the same:

→ Create a new module as described in section 6.2.2.4 Creating tests and reviews.
→ Include all the source files, include paths and defines necessary to analyze the source code of the component.
→ Activate “Component” as kind of test (see figure 6.256).

As environment the default GNU GCC compiler is used. This means the component tests will be executed on the Windows PC, using the microprocess of the PC as execution environment. If you use a cross compiler for an embedded microcontroller, you run the tests either on the actual microcontroller hardware or on a simulation of the microcontroller in question.

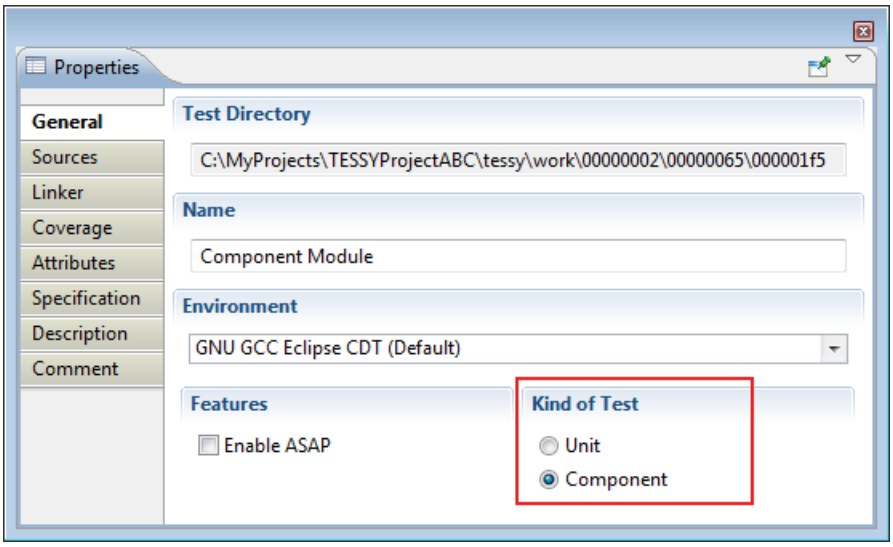


Figure 6.256: Component test

In contrast to normal unit tests, you will only see one special test object called “Scenarios” (see figure 6.257).

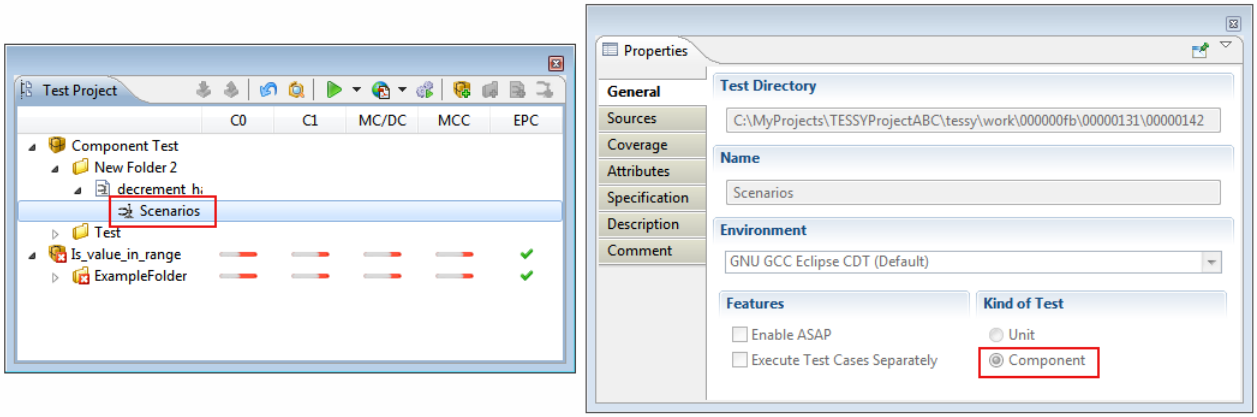


Figure 6.257: Scenarios of a component test

6.13.2 Preparing the test interface

The interface of the component is a summarized interface of all the non-static component functions:

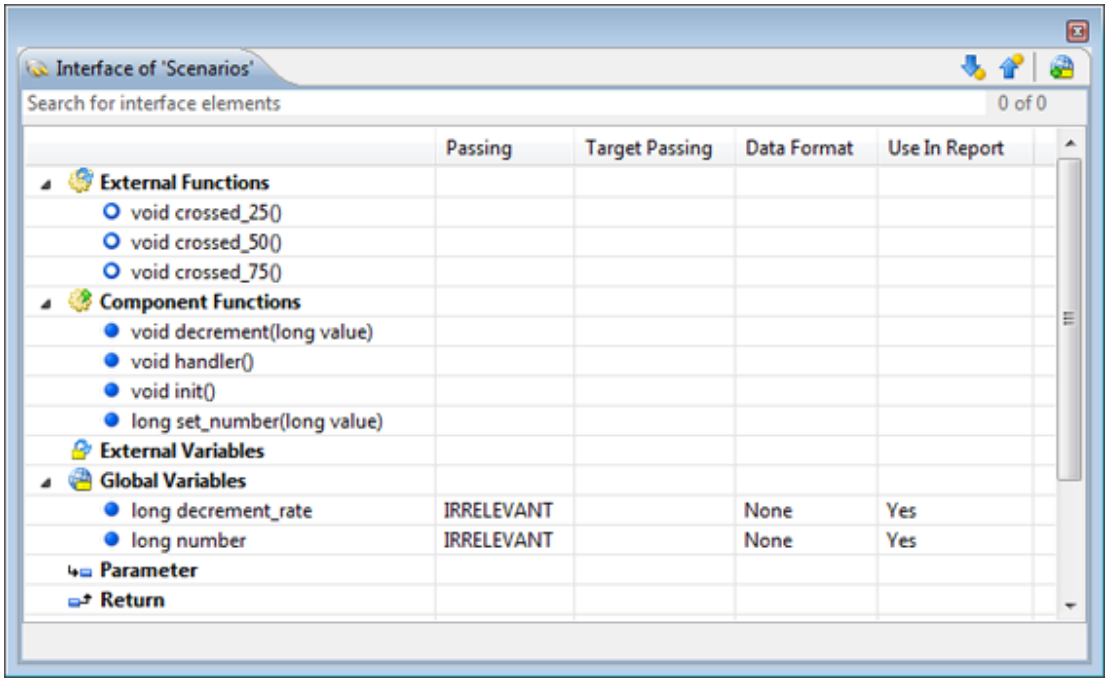


Figure 6.258: Interface of the scenarios

The External Functions section marked with the icon lists the interface to the underlying software functions, if any external function is called from the component. These external functions can be replaced by stub functions like within the normal unit test.
The Component Functions section marked with the icon lists all the component functions, i.e. the functions visible from outside the component. Local static functions will not be listed here.

The meaning of the status indicators for component functions is as follows:

Indicator	Status / Meaning
○	Function is not used for component test.
●	The variables used by this function are not available within the component test interface of the scenario. These variables are set to IRRELEVANT.

continue next page

Indicator	Status / Meaning
	The variables used by this function will be available within the scenario and the passing direction may be adjusted.

Table 6.88: Status indicators of the Interface view of a component test

6.13.3 Configuring the work tasks

The time based scenario description within SCE is based on time steps that represent the cyclic calls to a special handler function of the component. Such a handler function controls the behavior of a time-sliced component implementation.
The handler function needs to be selected as work task prior to executing any scenarios:

→ Select the desired function.
→ Click on (Set as Work Task) in the tool bar.
The icon of the function will change from to (see figure 6.259).

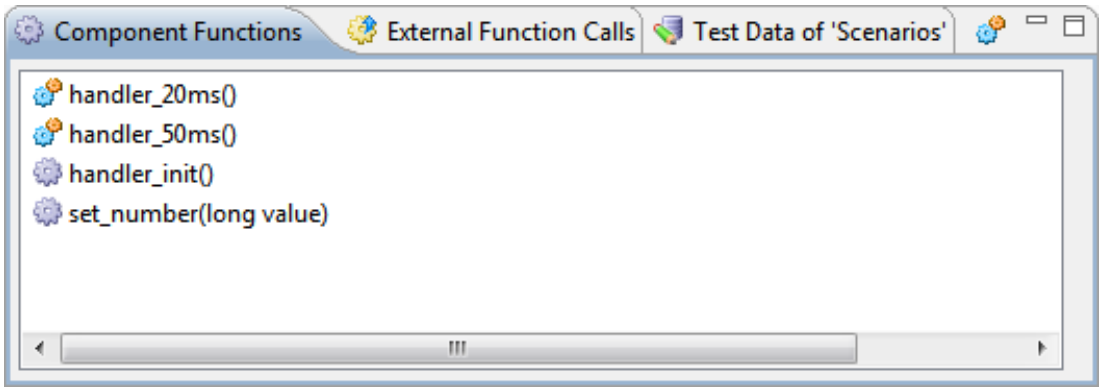


Figure 6.259: Two component functions were set as work task within the Component Functions view

You can select one or more of the component functions.

You can select several component functions as work tasks. This will be useful when testing several components together which all have a handler function.

The Work Task Configuration view allows more detailed settings for the work tasks.

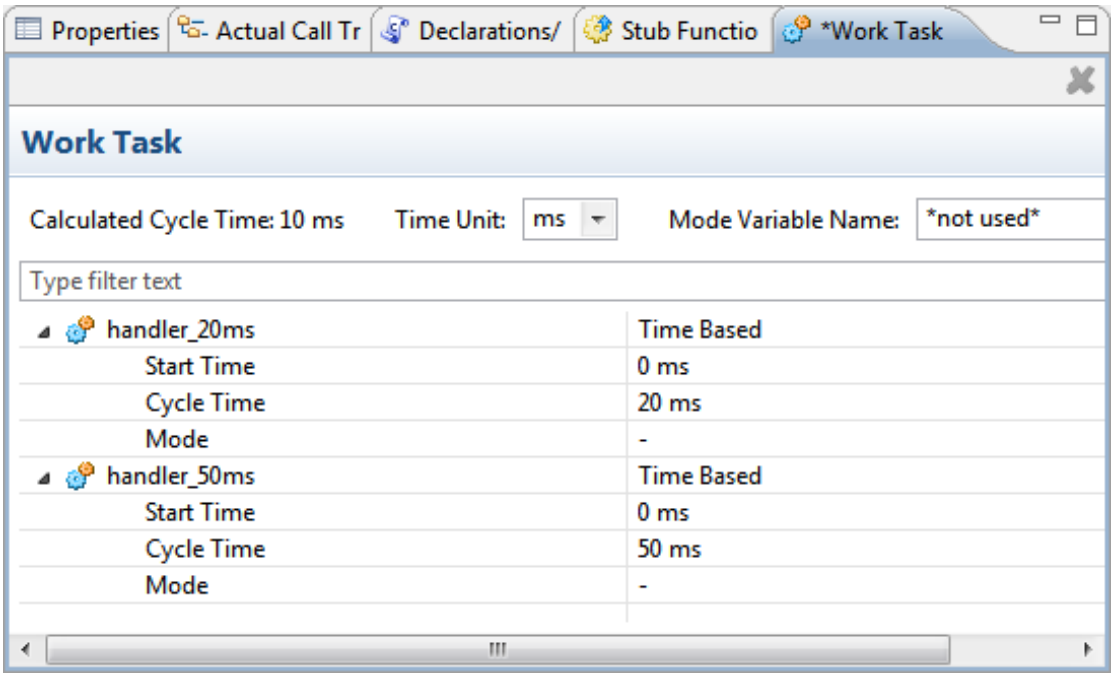


Figure 6.260: Work Task Configuration view

You can drop component functions directly into this view to configure them as work tasks. The view provides the following global settings:

- Time Unit (default is “ms”) which is just the display representation to be used within the GUI and reports.
- Mode Variable Name (not used by default) which optionally provides calling the work tasks depending on the value of the selected variable. All scalar variables can be selected here.

For each work task, you can specify the following settings:

- Start Time: Determines the point in time where this work task shall be called the first time for each scenario. The default is 0 ms which causes the work task being called immediately starting with the first time step.
- Cycle Time: Determines the elapsed amount of time after which the work task shall be called again. The default is 10 ms which causes the work task being called for every 10 ms time step.
- Mode: If a global Mode Variable Name is selected, you can specify for which value of this variable the respective work task shall be called. During test execution, this work task will only be called within its specified start and cycle time, if the mode variable has the specified value.

The order of appearance within the Work Task Configuration view reflects the actual calling sequence of the work tasks for each time step of the scenario. You can reorder the work tasks via drag and drop.

Another global setting is the calculated cycle time which depends on the cycle times of the given work tasks. It will be calculated automatically from the cycle times of the given work tasks.

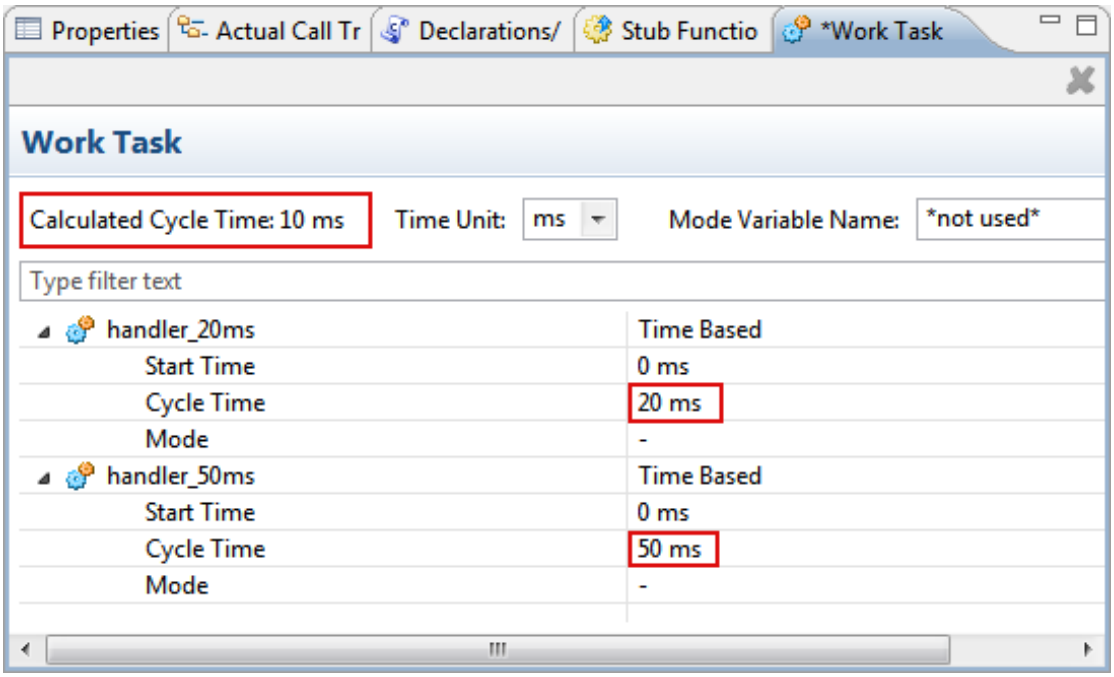


Figure 6.261: Calculated cycle time

Within the example in figure 6.261, the resulting global cycle time (i.e. the step width of the time steps of the scenarios) will be 10 ms, because this is the greatest common divisor of all the given work task cycle times (i.e. 20 and 50 ms in this example).

6.13.4 Designing the test cases

Testing a component requires a set of scenarios that stimulate the component and check the behavior of the component. Such a scenario contains calls to component functions and other possible actions and expected reactions of the component. A scenario can be seen as a test case for the component. Therefore, TESSY displays the list of scenarios within the Test Item view like normal test cases but with a different icon.

There are two possibilities for creating scenarios: Either by creation them ad hoc or by developing them systematically using the classification tree method supported by CTE within TESSY.

After synchronizing the CTE test cases there will be the respective number of scenarios within TESSY. You can add additional scenarios using the context menu within the scenario list. To edit the scenario, start the scenario editor SCE. The (empty) scenarios will be displayed within SCE providing the specification and description of the designed scenario test cases.

6.13.5 Editing scenarios

Any scenario has two different tasks to fulfill:

- The stimulation of the component like any external application would do it. This includes normal behavior as well as abnormal behavior which should check the error handling of the component.
- Checking the reaction of the component caused by the scenario stimulation.

We will examine the different possibilities to check expected behavior of the component under test. There are at least the following methods available:

- Checking return values of component functions called while stimulating the component.
- Checking the values of global variables (of the component).
- Checking the calling sequence of underlying external functions of the component. This would check the interface to any underlying components used by the component under test.
- Checking parameters of calls to underlying external functions (implemented as stub functions).
- Providing return values from calls to underlying external functions (implemented as stub functions) to the component.

The following sections describe the required settings for the above mentioned check methods.

6.13.5.1 Adding Function Calls

Stimulating calls to component functions or checking of calls to underlying external functions can be added to time steps:

→ Drag and drop the functions from the component functions onto the desired time step (see figure 6.262).

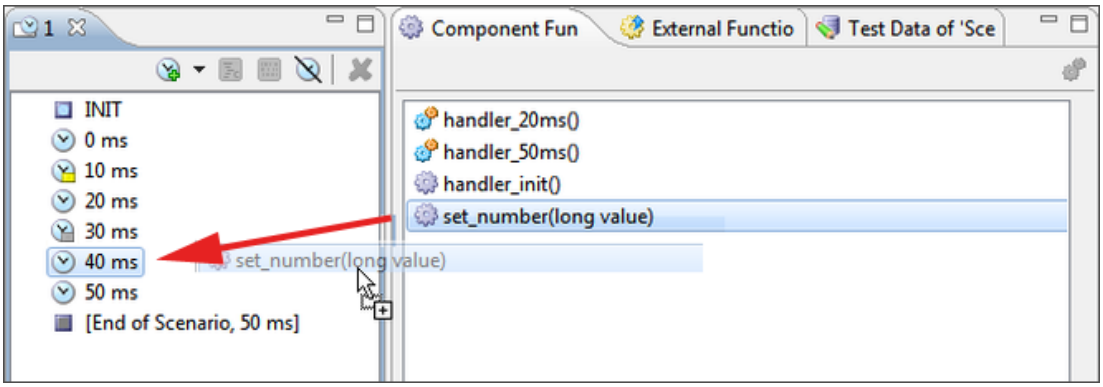


Figure 6.262: Adding Function Calls

There are some settings required for the function calls depending on the kind of function:

- component function: Parameter values need to be provided.
- component function: The return value has to be checked directly (for scalar types) or assigned to a variable for later evaluation.
- external called function: The expected time frame of the call to these functions needs to be specified. This defines the time range starting from the current time step, where a call to this function is rated as successful with respect to the calling sequence.

6.13.5.2 Entering test data for time steps

You can set input values or check output values of any variable at every time step of the scenario. According to your settings within TIE you have access to all variables available within the component interface. The test data can be entered within the Test Data view of the scenario perspective. When you select a time step, the Test Data view provides a column named like the time step for entering either new test data values or editing existing ones (see figure 6.263)

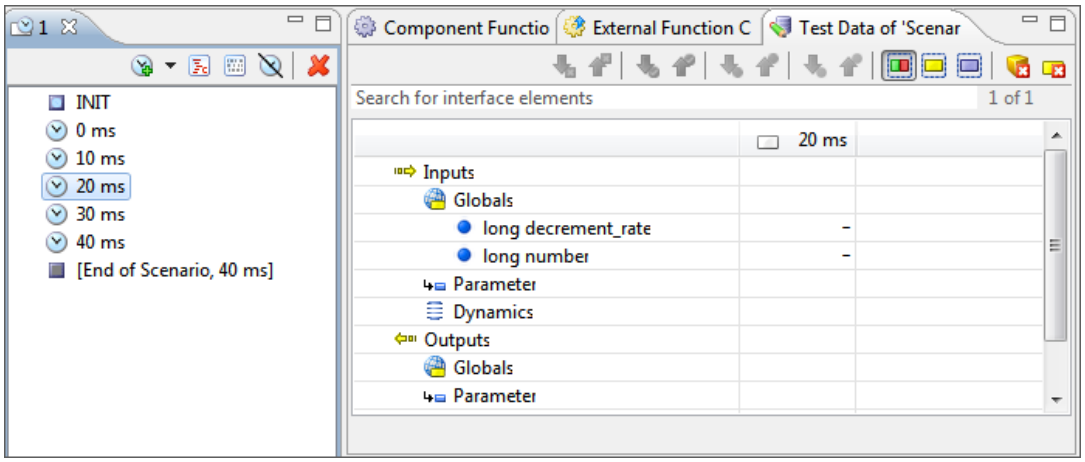


Figure 6.263: The Test Data view of ’Scenarios’

The Test Data view provides most of the editing features like for the normal unit testing. After entering any values, the icon of the respective time step will change indicating the test data status. The Test Data view shows columns for all time steps that contain test data plus one column for the currently selected time step.

Time step indicator icons for test data (see also figure 6.264):

- Gray indicator: Some input values are assigned but some are still missing and need to be provided. Select “*none*” for input values of time steps that you do not want to assign.
- Yellow indicator: At least all input values are assigned for this time step. The output values do not need to be assigned to execute a scenario.

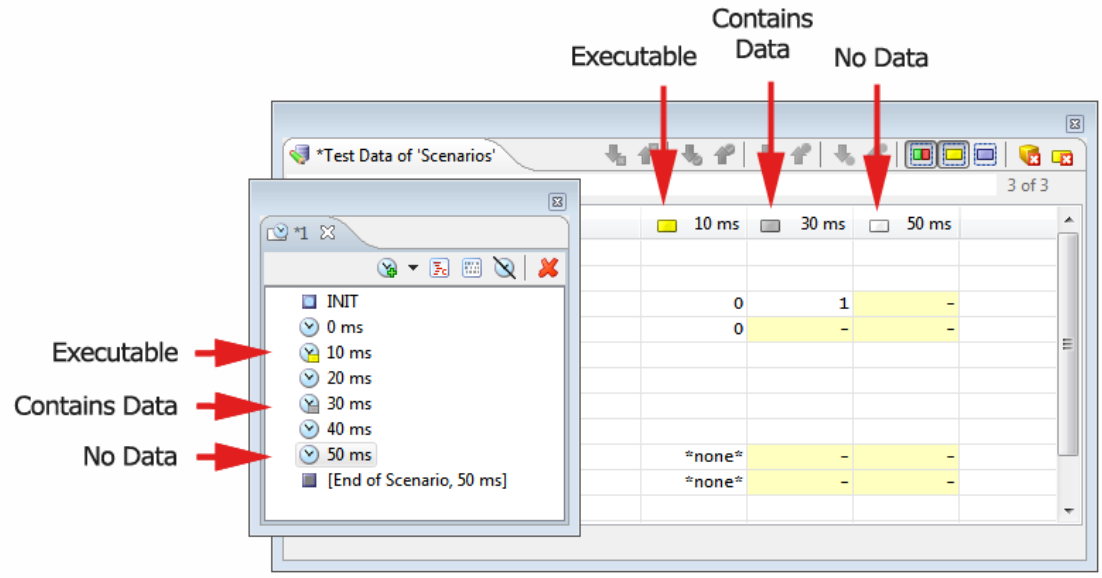


Figure 6.264: Indicator icons for the test data

Important: All time steps with test data need to have a yellow indicator before the scenario can be executed!

The icon of the scenario will change to yellow if there are no more time steps with a gray indicator.

6.13.5.3 Checking return values of component functions

When dragging component functions into the scenario, you need to provide the parameter values. For scalar values, you can simply add decimal or floating point numbers depending on the type of variable.

You can also provide a symbolic name of a variable with the corresponding type. This name will be added into the test application without any checking. If the symbolic name does not exist, there will be error messages when compiling the test application.

Either provide a value (for scalar return value types) or specify the symbolic name of a variable which the return value shall be assigned to (in this case, the variable provided should be of the same type like the return value type).

6.13.5.4 Checking the calling sequence

The calling sequence of calls to underlying external functions may be checked on an abstract level within the scenario. Not the absolute calling sequence will be evaluated, but the existence of function calls within a given period of time within the scenario. This provides a robust mechanism for call trace checking that ignores the internal implementation of the component.

How does it work? You specify the following information within the scenario for each expected function call:

- The time step where you expect the call at the earliest.
- The number of expected consecutive calls to the function (default is 1).
- Optionally a period of time (the time frame) from that starting point where the invocation of the function call is still successful with respect to the expected behavior of the component.

Both these settings are available for each expected call to an external function. The time frame is zero by default indicating that the expected function call shall take place within the same time step. If you specify the time frame as 60 like within the example above, this indicates that the expected call could take place within time step 20ms, 30ms or up to 80ms to be successful.

The exact sequence of the calls to those functions will not be examined, any of them may be called within the given time frame interval. The report shows the result of the evaluation of the call trace for the example above. The actual call trace entry contains the time step where this call occurred, the expected call trace entry shows the expected time frame period.

The following table shows the possible evaluation results for the call trace of the example calls to function crossed_50() and crossed_75().

Time step	**Result for call to crossed_50() [40 ms]**	**Result for call to crossed_75() [40-80 ms]**
40ms	ok	ok
50ms	failed	ok
60ms	failed	ok
70ms	failed	ok

continue next page

Time step	Result for call to crossed_50() [40 ms]	Result for call to crossed_75() [40-80 ms]
80md	failed	ok
90ms	failed	failed

Table 6.89: Example: possible evaluation results

If you need to check the exact calling sequence, you should set the time frame to zero. Other functions called in between the expected function calls are ignored. On the other hand, the time frame provides you with a powerful way to describe expected behavior of the component without knowing details about the exact implementation.

6.13.5.5 Checking if a function is not called

You may check that a function is not called within a given time interval. The example below checks that the function crossed_75()is not called within 100ms after the stimulation of the component by setting the expected call count to zero.

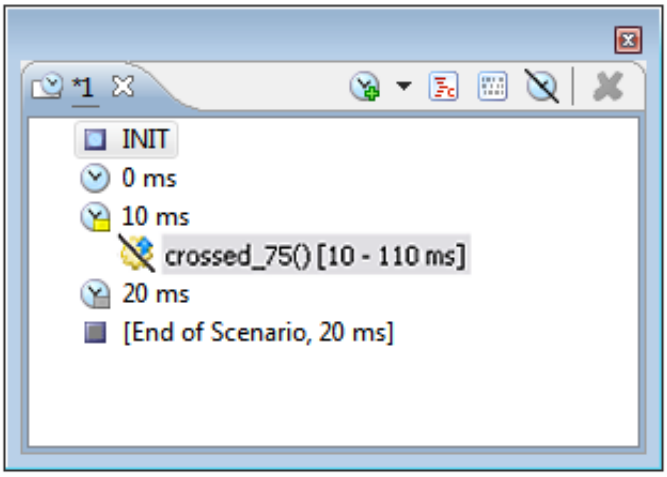


Figure 6.265: A function is not called

The crossed icon shows the special value of the expected call count, indicating a check that the function shall not be called.

6.13.5.6 Checking stub function parameters

Because called external functions need to be replaced by stub functions, you can check the parameter values like during unit testing, depending on the type of stub function you choose.

For more information refer to section 6.7.4.9 Defining stubs for functions.

6.13.6 Executing the scenarios

After implementing and editing the scenarios within SCE, execute the scenarios:

→ Select the desired scenario test cases and execute the test using the Execute Test button within the tool bar.

6.14 Fault injection

The fault injection feature provides means to test code parts that are not testable using normal testing inputs e.g. endless loops, read-after-write functionality or error cases in defensive programming. Dedicated testing code can be injected at selected branch locations of the test object so that decision outcomes can be manipulated. Such fault injections are valid for specially marked fault injection test cases only. This ensures proper operation of the normal test cases without any side effects that may be caused by fault injections.

6.14.1 Managing fault injections in the Coverage Viewer

Fault injections are edited within the Coverage Viewer (CV) (see chapter 6.11 CV: Analyzing the coverage for more information about the coverage viewer) based on the flow chart of the test object. They are displayed as blue circles at the respective branch. The injected code will either be injected into the respective branch or directly before the decision of this branch. This is useful because normally the decision outcome needs to be manipulated in order to reach a formerly unreachable branch.

6.11 CV: Analyzing the coverage

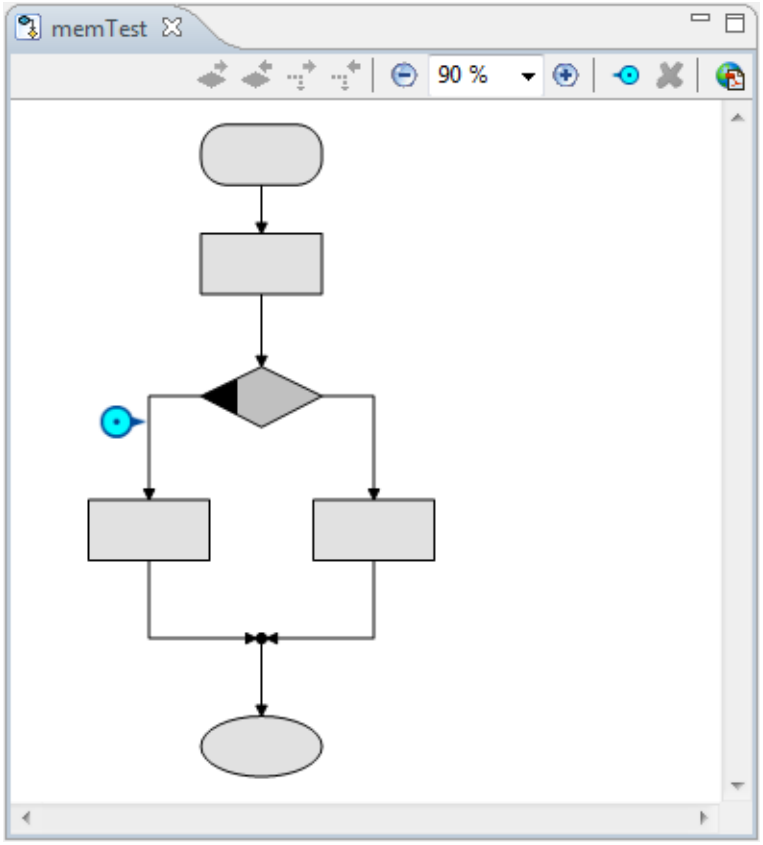


Figure 6.266: Fault injection in the Flow Chart view of the CV

Each fault injection is identified by its branch path which represents the decisions that need to be taken to reach the desired branch. This allows finding the right location of the fault injection even after source code changes. Fault injections that cannot be mapped to the current source code control flow are marked with an error symbol within the Fault Injections view of the CV.

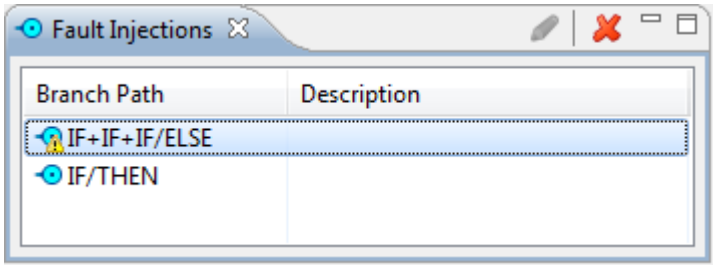


Figure 6.267: Fault Injection not found

Such unmapped fault injections can be assigned to branches via drag and drop onto the desired branch.

6.14.1.1 Fault injection related tool bar icons

Icon	Action / Shortcut / Comment
	Edits fault injections.
	Deletes fault injections.

Table 6.90: Fault injection related tool bar icons in the Flow Chart view of the CV

6.14.2 Creating fault injection test cases

Fault injection test cases are specially marked in order to distinguish between normal functional test cases and those added for specific testing challenges like unreached branches. The same circle icon (see subsection 6.14.2.1 Status indicator) decorates fault injection test cases within the Test Item view in the Overview perspective.

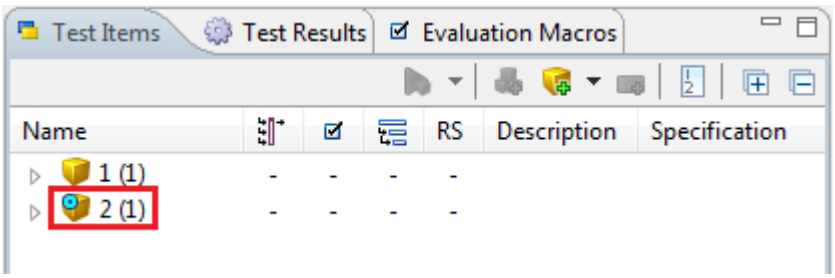


Figure 6.268: Fault injection test case

Normal test cases can be changed to fault injection test cases and vice versa using the context menu of the Test Items view in the Overview perspective.

Test cases for fault injections are only effective if they reach the decision of the branch where the injected code has been added. Otherwise the injected code would be useless because it would not be executed.

It is recommended to first run all normal test cases in order to reveal any unreached branches within the control flow of the test object shown within the CV. Because the CV knows which test cases have reached certain decisions with unreached branches, it is very easy to select appropriate fault injection candidate test cases. For this purpose, the Edit Fault Injection dialog (see figure 6.269) provides a list of test cases reaching the decision of a selected branch. One or several of these test cases can be copied as fault injection test cases for such unreached branches.

6.14.2.1 Status indicator

Indicator	Status / Meaning
	Indicates a fault injection.

Table 6.91: Indicated fault injection in the Test items view

6.14.3 Creating and editing fault injections in the Coverage Viewer

Fault injections are created and edited within the Coverage Viewer (CV). For more information about the CV see chapter 6.11 CV: Analyzing the coverage

→ Select a branch of the flow graph within the CV and click on the “Edit Fault Injection” button within the Flow Chart view tool bar.
This will open the Edit Fault Injection dialog (see figure 6.269).

Fault injection dialog

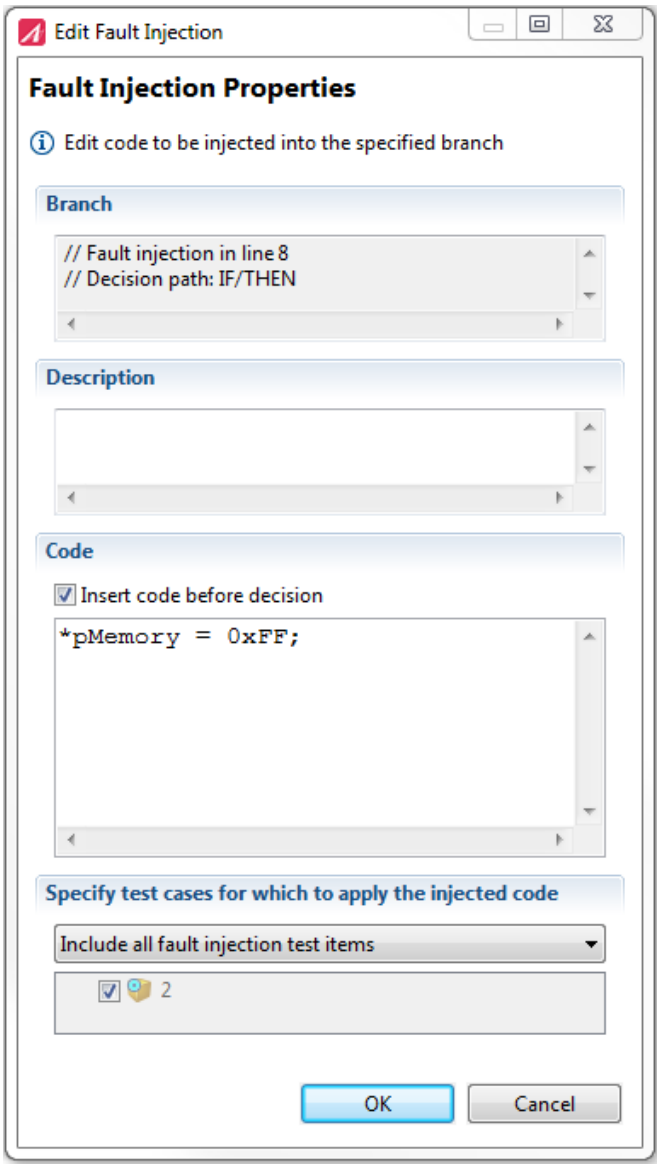


Figure 6.269: The Edit Fault Injection dialog

Important: Fault injections will only be applied if at least one fault injection test case is available! Also make sure that your fault injection test cases reach the respective code location where the fault injection code gets injected.

To edit your fault injection within the dialog you need to know:

- The “Branch” text field is read-only and displays the current line number of the decision of the selected branch. The branch path identifies the location of the selected branch within the control flow of the test object.
- As an option you can enter a description for the cause of the fault injection. This text will appear within the test report.
- Select whether the code shall be injected into the branch or directly before the decision of the selected branch (by selecting the “Insert code before decision” toggle button).
- Enter the code to be injected.
- Select the applicable fault injection test cases. Be aware that this is only possible if the existing test cases have been executed with coverage measurement enabled. Otherwise you can only select to include all fault injection test items.

Within the include list of the dialog you will find all normal functional test cases that reach the branch decision if there is no other existing fault injection test case that reaches this decision. Such fault injection candidates will be listed with a “Copy of” prefix in front of their name. They will be actually copied when you save all changes done to fault injections.

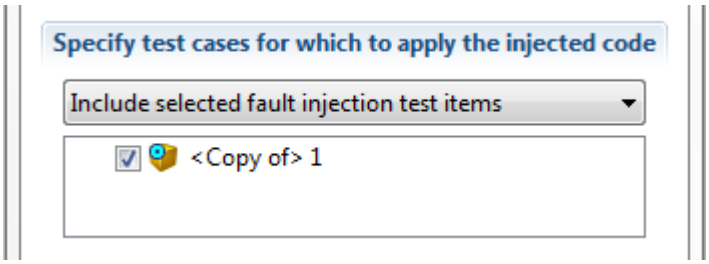


Figure 6.270: Include list at the bottom of the Fault Injection dialog

The following possible situations can arise:

Precondition	Items being displayed within the include list of the fault injection dialog
Test not executed or executed without coverage measurement.	None.
No fault injection test cases available.	“Copy of” candidates reaching the branch decision (or an empty list if none reaches the decision).
Any fault injection test case reaches the branch decision.	Only fault injection test cases reaching the branch decision.
No fault injection test case reaches the branch decision.	“Copy of” candidates reaching the branch decision (or an empty list if none reaches the decision).

Table 6.92: Possible situations in the include list of the Edit Fault Injection dialog

After creating or editing fault injections the CV becomes dirty and displays an asterisk at the respective Flow Chart view title. You will be asked to save the changes made when switching to another test object or another perspective. Any copying of fault injection test case candidates will be delayed until saving. This preserves the information about reached branch decisions until the end of the editing operation.

6.14.4 Fault injections within the report

Fault injection report

The test details report contains a table with all fault injections for the respective test object and a list of test cases marked as fault injection test cases.

Fault Injections	
Fault Injection Test Cases	
Test case 3	
Branch Path: IF/THEN	
Code (inserted before decision)	`*pMemory = 0xff;`
Applicable for	Fault injection test case 3

Figure 6.271: Example of the Fault Injections report

6.15 Backup, restore, version control

With TESSY you can easily backup modules and tasks into a directory and check in into a version control system. Modules and tasks can also be restored from that directory which facilitates checking out modules and tasks from the version control system onto another computer and restoring the test database.

You can backup individual tasks, modules, folders or whole test collections. The backups will be stored as TMB files. Restoring the files is either possible within the original folder or as well from another location.

Warning: Using the restore function you have to keep in mind that script files are only transferred into the location where internal script files are stored. The TESSY model itself will not be updated. (See chapter 6.10 Script Editor: Textual editing of test cases for more information.) Therefore external changes to the generated backup scripts are not recommended.

If uncommitted user script changes are found during the backup process, a warning dialog will appear.

Use the import function instead of the database restore in order to apply script changes made outside of TESSY.

6.15.1 Backup

→ In the menu bar select “File” > “Database Backup” > “Save…” .
The Save Database dialog will be opened with your module already selected (see figure 6.272).

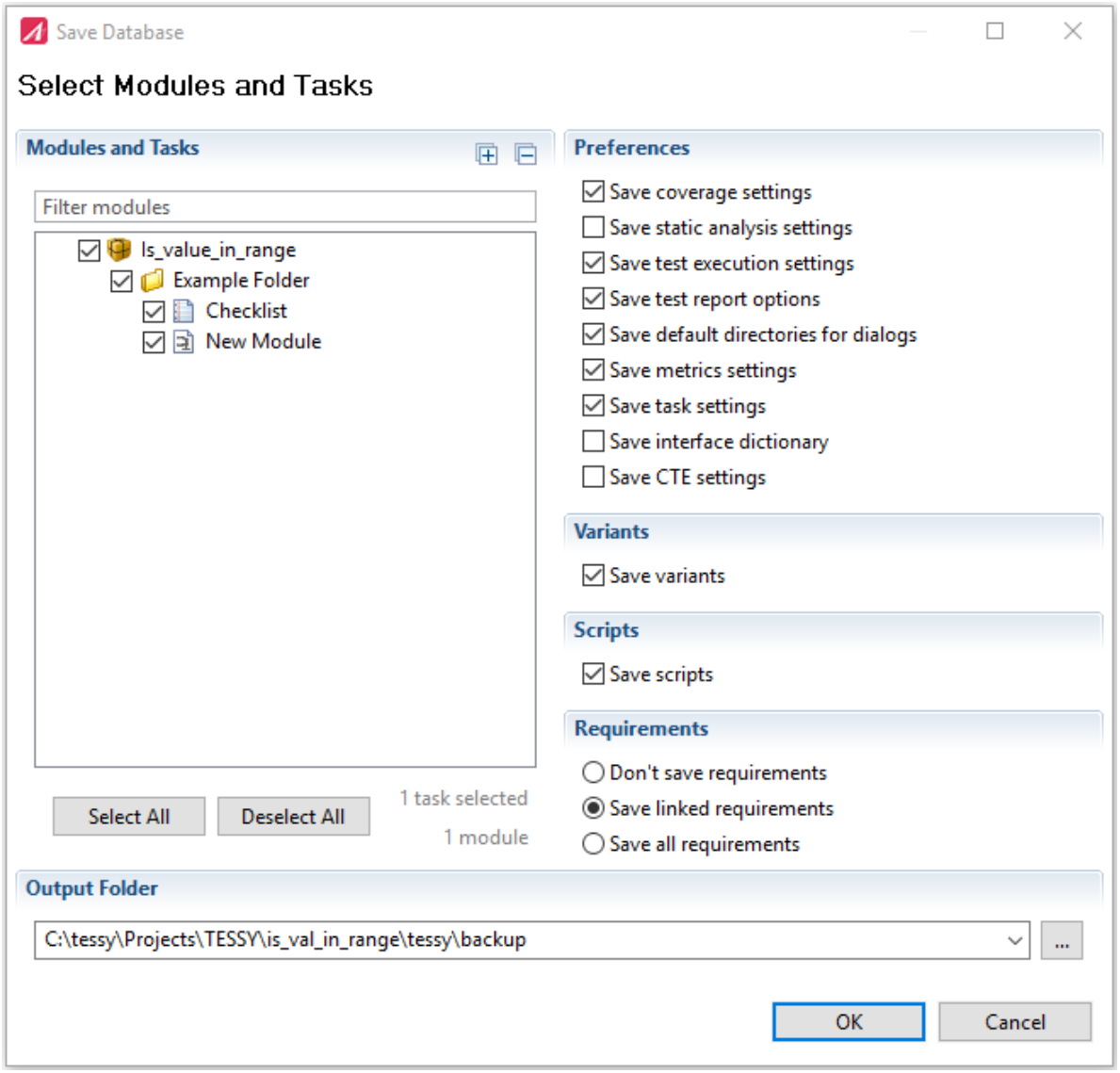


Figure 6.272: Save Database dialog

→ Decide, which modules you want to save by either selecting them separately or pressing the button “Select All”.

→ Decide, if you want to save the coverage settings, test report options or dialog settings from the Window > Preferences menu.

→ If you have linked your test cases with any requirement documents, you can choose to save the referenced requirement documents as well. In this case the requirements will be saved within the TMB file.

→ If you want to safe variants and scripts, please remember to select it (see figure 6.272).

Information about scripts is provided in chapter 6.10 Script Editor: Textual editing of test cases.
Information about variants can be found in chapter 6.2.2.9 Creating variant modules and in section 6.2.5.7 Test cases and steps inherited from a variant module.

→ Click “OK”.

The “Backup Folder” displays the backup directory of the current project. We recommend to use this directory for any backup and restore operations.

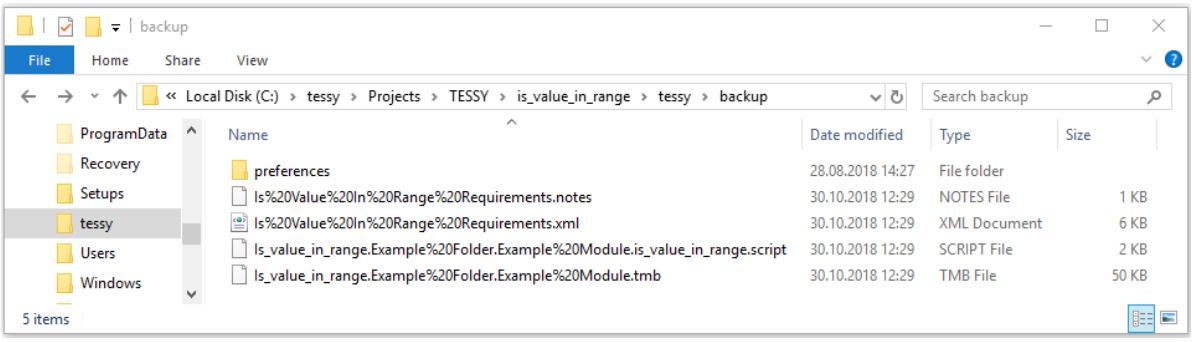


Figure 6.273: Files of the backup

For each module there will be a file named like the path to the module starting from the test collection. The file name will be escaped using URL encoding which replaces for instance the space character with a “%20”. The preferences are stored within separate files within the “preferences” subdirectory.

6.15.2 Restore

6.15.2.1 Restore into the original location

→ Select “File” > “Database Backup” > “Restore…” .
The Restore Database dialog will be opened.

→ Select the directory with the backup.
The box “Modules” now shows the hierarchy of modules that can be restored from the given TMB files within the backup directory (see figure 6.274).

→ If there are any requirement document backups, the respective requirement documents will appear within the box “Requirements”.

Make sure you ticked the requirement boxes to import them!

→ Click “OK”.

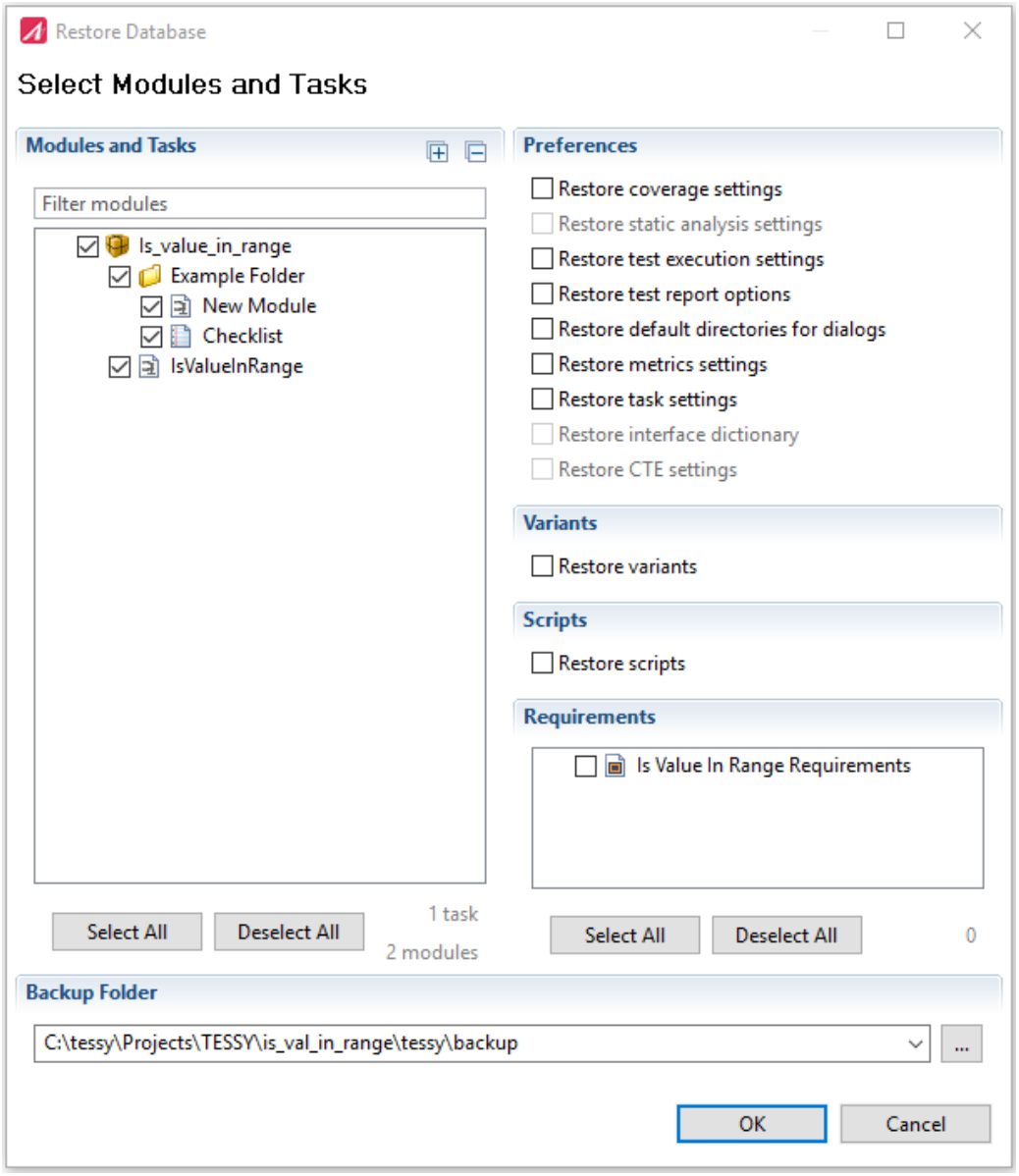


Figure 6.274: Restore Database dialog

If you want to include external user script changes into your restoration, you need to select “Restore scripts” before starting the restoring process. But please keep in mind, that only the internally managed user scripts will be updated and not the TESSY model itself.

More information about the handling of scripts is provided in chapter 6.10 Script Editor: Textual editing of test cases.

6.15.2.2 Restore into another location

You can also restore TMB backup files into another than the original location:

If you select any folder for which there are no corresponding TMB backup files, restore any of the available TMB files as children of this folder. The original test collections and folders of the TMB files will be restored as sub folders of the current folder instead.

6.15.3 Version control

We recommend to save backups of all test relevant files into a version control system on a regular basis. At least when the test development is completed, the whole test setup should be saved as backup.

→ Follow the steps described above to create the necessary files.
→ Save the following files and contents of directories into your version control system:
- *tessy* – PDBX file
- *preferences* – XML document
- Contents of the *config* directory
- Contents of the *backup* directory

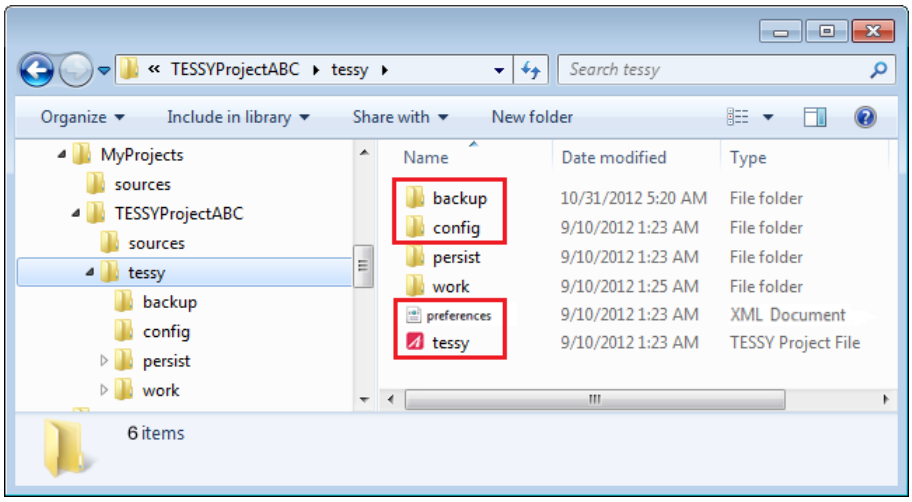


Figure 6.275: Directories and files within the database directory of the TESSY project

The directory *work* contains only temporary files created during development and execution of the tests. You can delete this complete directory to save disk space after the testing work is completed.

The directory *persist* contains the current databases of the test project. This directory and the sub directories will be restored when restoring TMB backup files. The valuable contents of this directory will be saved into the TMB files created during the backup process.
When you restore the whole project onto another computer, the directory *persist* will be restored from the TMB backup files.

In the *preferences.xml* workspace specific options like “Dialog Settings”, “Script Editor”, “Static Analysis”, “Tasks”, “Test Execution Settings”, “Test Interface Settings”and “Test Project Settings” are stored.

Also project specific options like “Metrics” and “Test Report Options” are stored in the *preferences.xml*.

Workspace specific options can be overwritten with project specific values:

By choosing which preferences to save in the Save/Restore dialog (see figure 6.272 and figure 6.274) you can define the workspace preferences which are supposed to be overwritten whenever restoring the project.

6.16 Command line interface

TESSY provides a command line interface which allows writing batch script files that control various operations within a TESSY project. The command line operations are available by invoking an executable program called “tessycmd.exe”.
The program can be called either from a DOS command line shell or from a DOS batch script or other script formats that support calling DOS executables.

Before invoking any “tessycmd.exe” commands you need to start TESSY. The “tessycmd.exe” will connect to a running instance of TESSY in order to execute any commands. You can run TESSY either in GUI mode with a graphical user interface (when started normally using “TESSY.exe”) or in headless mode without a GUI (when started using “tessyd.exe”).

For information about the usage of TESSY together with continuous integration servers like Jenkins refer to the application note “Continuous Integration with Jenkins” (“Help” > “Documentation”).

As a precondition for command line execution you need to have a readily configured project and some TMB backup files within this project containing your tests. This project should then be restored from your version control system into any location on a computer controlled by your continuous integration system. Via “tessycmd” commands you can now restore your test project from the TMB files and execute the tests.

Transformation of the TESSY result XML files into XUNIT format is described within section 6.16.5 Execution and result evaluation.

6.16.1 Starting TESSY in headless mode

For test automation on continuous integration servers or nightly builds it is required to start TESSY in headless mode (i.e. without displaying a GUI). TESSY provides a special starter application for this purpose called “tessyd.exe”. When invoking “tessyd.exe” within your batch script, it will start TESSY in headless mode and wait until the TESSY application is ready to receive commands via “tessycmd.exe”.

At the end of your script you should shutdown TESSY using the same “tessyd.exe” application with the parameter “shutdown”. The calling sequence for running batch tests would be like follows:

`tessyd.exe --file <tessy.pdbx file>`
`tessycmd <commands>`
`tessyd shutdown --copy-log <directory>`

Table 6.93: Calling sequence for running batch tests

When running TESSY in headless mode, the console output will be written into a file “console.log” within the directory:
%USERPROFILE%\.tessy_41_workspace\.metadata

For details on how to archive any problems and console outputs for further analysis refer to subsection 7.2.3.2 Headless operation problems log.

6.16.2 Invoking ``tessycmd.exe''

The executable that provides all command line operations is available within the TESSY installation directory:

`C:\Program Files\Razorcat\TESSY_4.x\bin\tessycmd.exe`

- → Start a DOS shell.
- → Change to the directory “bin”.
- → Call the executable.

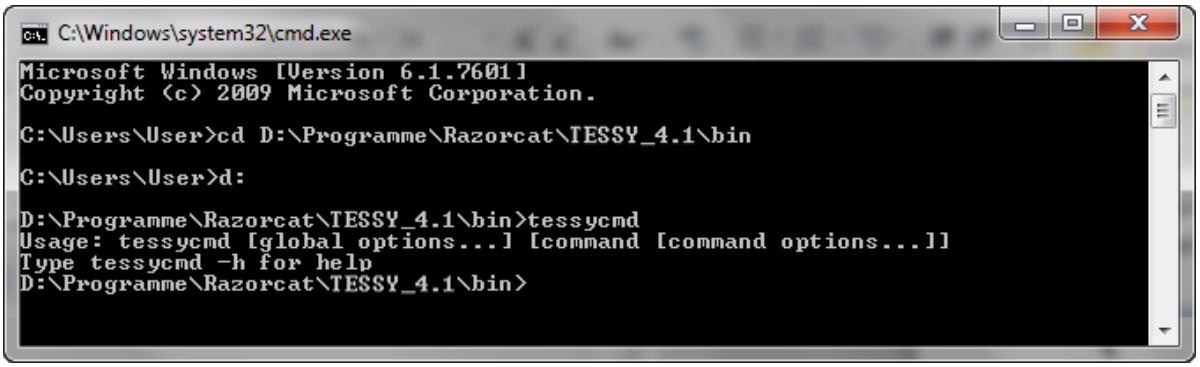


Figure 6.276: DOS command line shell

6.16.3 Usage of ``tessycmd.exe''

The available commands provide means to create, select and list TESSY objects, i.e. a project, test collection, folder, module, test object. After invoking any create commands, the respective new TESSY object will be selected. You can invoke further commands to manipulate any previously created or selected TESSY objects. You need to call all commands according to the following sequence:

- Connect to TESSY.
- Select or create TESSY objects.
- Invoke commands to start operations on the selected TESSY objects.
- Disconnect from TESSY.

Important: If you are not connected, invoking any commands will fail.

The current state (connection and selection of TESSY objects) of the “tessycmd.exe” executable is managed by the currently running TESSY application. If you restart TESSY, the state of “tessycmd.exe” will be reset to the initial state, i.e. disconnected.

6.16.4 Commands

command	operation
tessycmd -h	displays a complete list of the available commands
tessycmd -h <name fragment>	displays a list of the commands matching the given name fragment
tessyd -f <name of pdbx file>	imports and opens the project referred by the given .pdbx file
tessyd -p <name of project>	opens the given project (Must be already available within the project list.)
tessycmd connect	connects to the running TESSY instance (If a project was specified when running tessyd, this project is automatically selected after connecting.)
tessycmd list-projects	lists the available projects
tessycmd disconnect	disconnects from TESSY

Table 6.94: Excerpt of the possible commands of the command line interface

To execute “tessycmd.exe” within any directory, add the directory “bin” of the TESSY installation to the Windows path environment variable.

6.16.5 Execution and result evaluation

The most common batch operation would be to start TESSY with a readily configured project, run a TESSY batch operation and evaluate the results.

Please note the following options for “tessycmd” commands that are useful for this operation:

```
tessycmd exec-test -f <batch file> [-o <output directory>]
```

The optional output directory overwrites the report output directory given within the batch file (*.tbs). If provided, the generated reports are copied to that output directory after execution of the batch file.
In this way you can easily specify the output path for result XML files on the command line:

```
tessycmd xslt [-xsl <XSL file>] [-o <output file>] <XML file>
```

With the optional XSL file you can specify the XSL transformation being applied to the TESSY XML result file created with the previous step. A template for an XUNIT compatible transformation can be found within the installation directory of TESSY:

```
C:\Program Files\Razorcat\TESSY_4.x\bin\plugins\com.razorcat.tessy.
reporting.templates\4.x\ci\TESSY_TestDetails_Report_JUnit.xsl
```

Important: There is no command to create a new project from scratch using the command line because the necessary options would be too extensive to be handled usefully on command line. Please follow the steps described within chapter 4.1 Creating databases and working with the file system and in chapter 6.15 Backup, restore, version control on how to create an empty project with the required configuration and save this project to disk. Such an (empty) project can be copied to any location on disk and populated with your TMB files using “tessycmd”.

6.16.6 Headless operation

The CLI command line execution mode of TESSY is designed for usage on continuous integration platforms like e.g. Jenkins. Therefore it is desired that TESSY does an auto-reuse of existing tests on interface changes and tries to execute as many tests as possible with newer versions of the source code being tested when running in CLI mode.

The following relevant source code changes will be auto-reused as far as possible in CLI mode:

- Additional or removed called functions
- New variables or struct/union components being used
- Old variables or struct/union components being no longer used
- Additional or removed test objects
- Additional or removed defines/enums

As a result, the tests executed in CLI mode may be run with uninitialized new variables which could hide existing or newly introduced errors within the software being tested. Also endless loops may occur due to such uninitialized variables.

Warning: Such auto-reused tests may have lost significant test data! Therefore it is discouraged to apply CLI auto-reuse on the working copy of your project.

For details on how to archive any problems and console outputs for further analysis refer to subsection 7.2.3.2 Headless operation problems log.

6.16.7 Example: DOS script

You will find the following example DOS script within the TESSY installation directory:

`C:\Program Files\Razorcat\TESSY_4.x\Examples\CommandLine\cmd_line_example.bat`

The script is prepared to import TESSY backup files (TMB files) into the currently open TESSY project. It will create a new test collection “Examples” and import the existing TMB files into a newly created folder. After the import it executes the imported modules. To run the script:

→ Start TESSY, create a new project and open this project.
→ Start a DOS command shell.
→ Change to the bin directory of the TESSY installation:
`C:\Program Files\Razorcat\TESSY_4.x\bin`
→ Invoke the DOS batch script “cmd_line_example.bat”. Alternatively, double-click on the file “cmd_line_example.bat” which invokes the DOS script within a new DOS shell. There is a pause command at the end which causes the DOS shell to remain open after execution of the script.

7 Troubleshooting

For compiler/target settings refer to our application notes available in the Help menu of TESSY (“Help” > “Documentation”)!

If you have troubles with errors or do not know how to proceed:

- Check this manual and make sure that you have operated correctly.
- Check section 7.3 Solutions for common problems.
- Check our application notes that are available in the Help menu of TESSY (“Help” > “Documentation”).
- Check our website for commonly asked questions and current issues http://www.razorcat.com.

7.1. Contacting the TESSY support **503**

7.2. Enhanced error handling **507**

7.2.1. Problems Log dialog 507

7.2.2. Problems view 509

7.2.3. Opening external problem logs using the Help menu 510

7.3. Solutions for common problems **512**

7.3.1. TESSY does not start or gives errors when starting 512

7.3.2. TESSY gives errors when quitted 513

7.3.3. License server does not start or gives errors 514

7.3.4. Working with constant variables 515

7.3.5. Dealing with too long project paths 519

7.1 Contacting the TESSY support

If you have further questions or if there is a problem you could not solve with the documentations described above, please contact our **Technical Support** per e-mail: support@razor-cat.com

Include in your support request e-mail:

- your contact data
- a short description of your question or problem
- the TESSY Support File if you get errors
- a screenshot showing the error message and related view contents of the GUI

The **TESSY Support File** (TGZ file) contains information about test object including data, compiler, project settings etc. It helps the support to detect the cause of your problem.

Creating the TESSY Support File

→ In TESSY, select the module or test object that is causing the problem.
→ Click “Help” in the menu bar.
→ Select “Support” > “Create Support File”.
→ Tick the box “Preprocessed sources” if possible.

Important: The box “Preprocessed Sources” is not ticked by default. This avoids that confidential sources might be included accidentally. Whenever you can afford to provide the sources to the support, tick the box “Preprocessed Sources”. In most cases it is necessary for successful problem investigation.

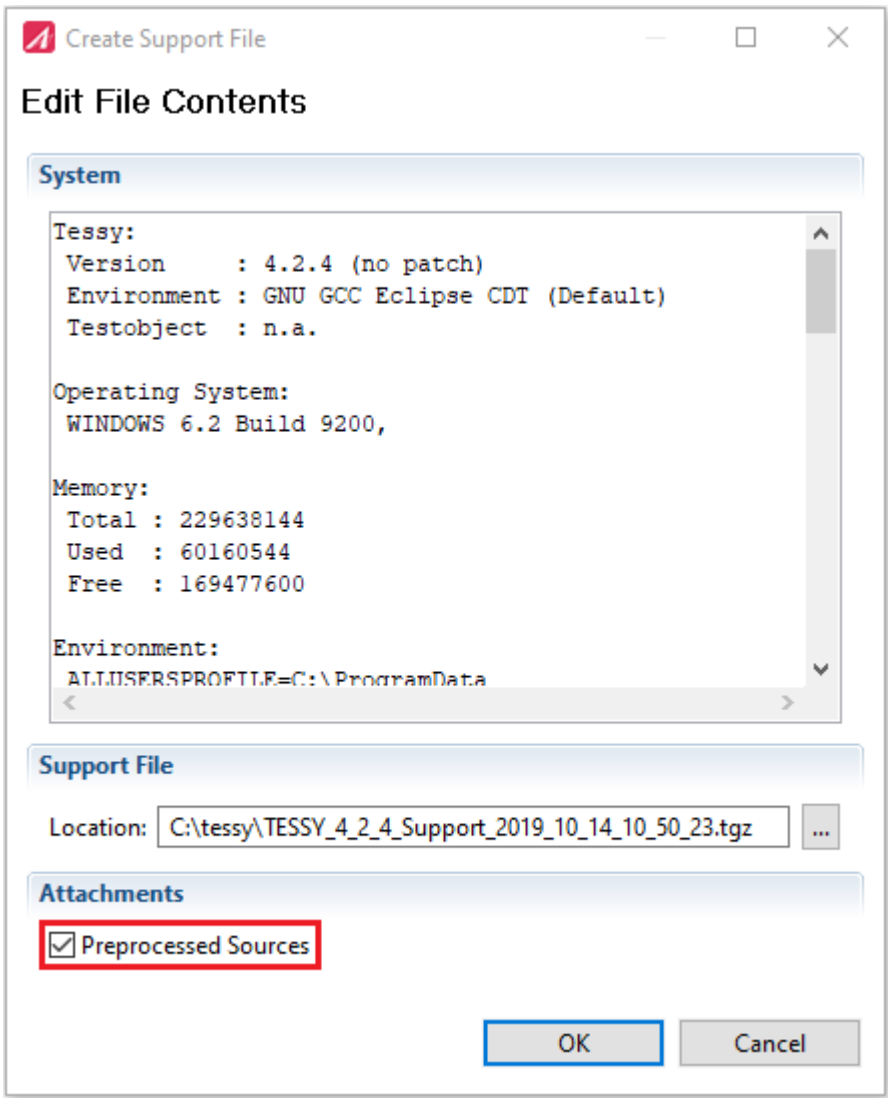


Figure 7.1: Dialog for creating the TESSY Support File

→ You can also change the file name and choose a folder if you wish.

→ Click “OK”.
The TESSY Support File (TGZ file) is created.

Tipps for a better TESSY Support File

1. Reduce unnecessary information:

   → Before reproducing the problem, switch to the Console view of the perspective "Overview".
   → In the tool bar click on the icon (Clear Console).
   All messages will be deleted.

2. Enable TESSY to display additional information:

   → In the main menu bar select "Help" > "Logging".
   → Select a process, the level (if selectable) and "on" as the table beneath "Table Name" explains (see table below).

It is possible to enable logging for all process steps.

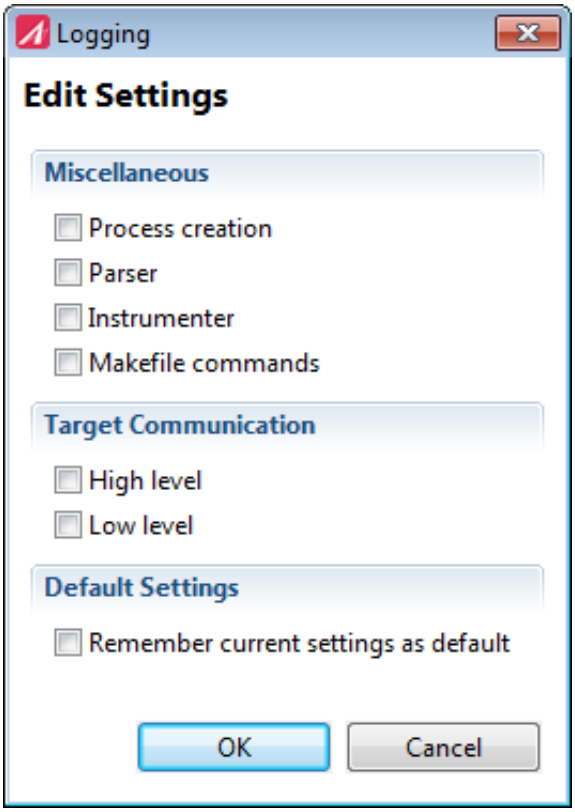


Figure 7.2: Dialog for logging settings

The additional information can relate to different process steps within TESSY. Enable the logging of the information you suspect the problem to stem from:

Process step	**Log if …**
Process Creation	parts of TESSY do not start correctly or TESSY is not able to start the test system (e.g. debugger).
Parser	problems occur during analysis of the source code to determine test objects and their interfaces. Often caused by incorrect syntax.
Instrumenter	problems occur during instrumentation of the source code to determine code coverage.
Makefile Commands	the test application (slave) or the test driver (master) cannot be created or are created incorrectly.
Target Communication	problems occur in the communication between TESSY and the debugger (e.g. the test system).
High level	you want to log the general TESSY activities. Seldom required to find a problem.
Low level	you want to log debugger-specific activities. Often very useful.

Table 7.1: Information to log and add to TESSY Support File

→ You can save the settings for logging with ticking the box “Remember current settings …” .

3. Reproduce the problem:

→ Do the actions again that lead to the problem (e.g. opening the module).
→ Keep the respective element selected that caused the problem (e.g. the test object in case of errors while executing) when creating the support file.

4. Create the support file, now including all log messages.

7.2 Enhanced error handling

In general setting up software tests is a complex task. Problems showing up during the test execution process can have various reasons.

To more easily locate possible errors that can appear while executing tests TESSY offers an enhanced error handling to provide enhanced error messages and logging capabilities for command line execution.

An error dialog shows the full exception chain, the context (e.g. the affected test object) and provides easy access to the error log file and console messages of the affected operation causing the error.

Important: TESSY will collect all problems and related console outputs into problem files within the "`%USERPROFILE%\.tessy_42_workspace\.metadata`" directory.

The newest file will be kept as "problems.zlog" and up to nine older files will appear with time stamps in the file name.

7.2.1 Problems Log dialog

If the test execution fails for some reason, TESSY opens an information dialog with more details about the problems that occurred. This makes it easier to handle such issues.

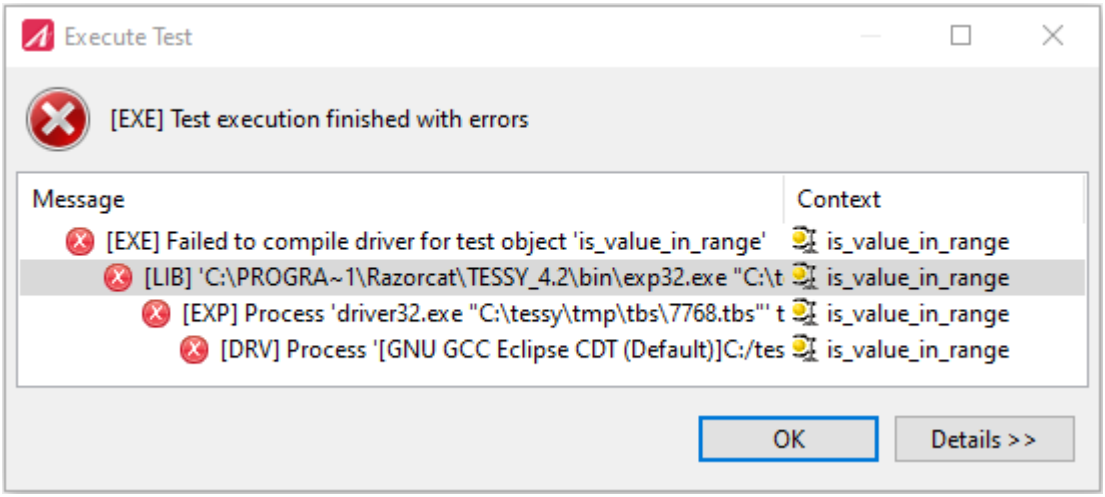


Figure 7.3: Problems Log dialog

→ Click on "Details" to find more information.

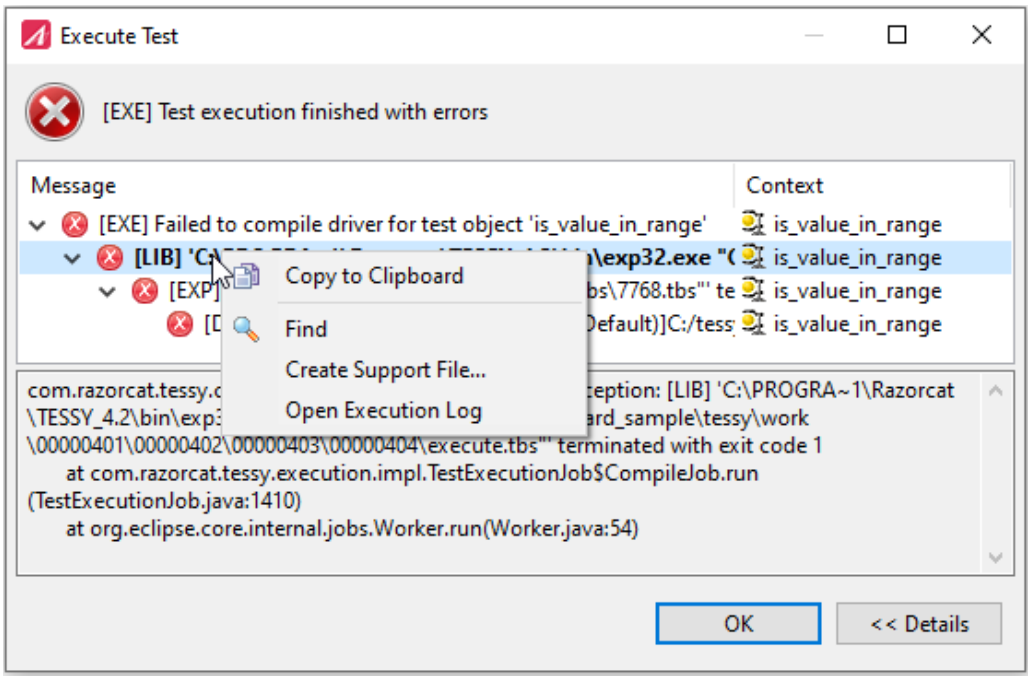


Figure 7.4: Problems Log dialog with details and context menu for individual log entries

"Open Log File" in the Help Menu (see figure 7.6) will open the log file in a text editor.

A right click on an entry within the Problems Log dialog opens a context menu with several options:

Item	Function
Copy to Clipboard	Copies the text of the selected line and additional stack trace information if available.
Find	Locates the context (e.g. module or test object) of error in the Test Project view in the Overview perspective.
Create Support File …	Creates a support file.
Open Execution Log	Shows all console messages of the operation that caused the error.

Table 7.2: The context menu of the Problems Log dialog

7.2.2 Problems view

The Problems view in the Overview perspective also offers access to information about execution problems. It displays all errors occurred during the current TESSY session (unless “Clear problems view before execution” is selected within the execution preferences).

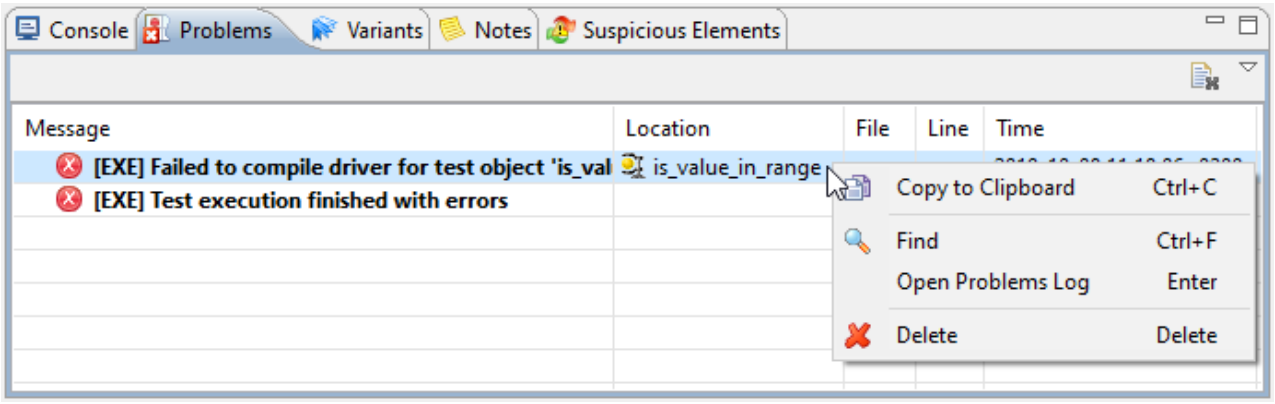


Figure 7.5: Problems view with context menu

A right click in every line opens a context menu with several options:

Item	Function
Copy to Clipboard	Copies the selected line.
Find	Locates the context (e.g. module or test object) of error in the Test Project view in the Overview perspective.
Open Problems Log …	Opens the Problems Log dialog.
Delete	Deletes the selected line.

Table 7.3: The context menu of the Problems view

Also a double click on every line in the Problems view reopens the Problems Log dialog.

7.2.3 Opening external problem logs using the Help menu

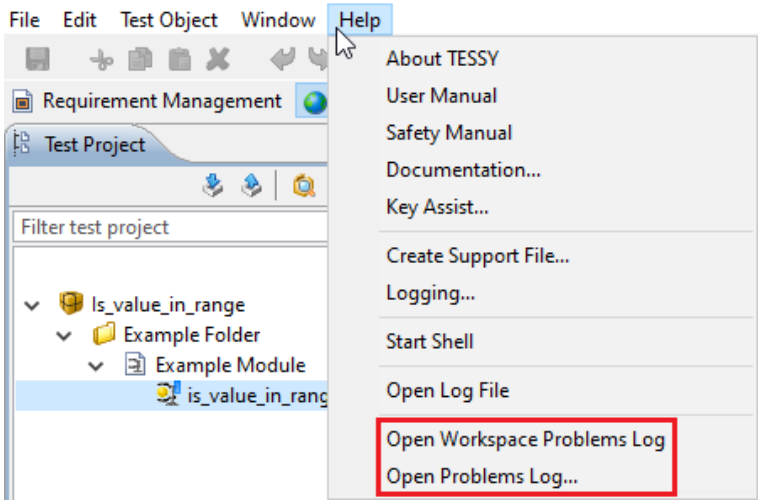


Figure 7.6: Help menu with problem menu items

7.2.3.1 Session problems log

The item “Open Workspace Problems Log” in the Help menu of the menu bar opens an information dialog with a list of problems that have occurred during the current TESSY session.

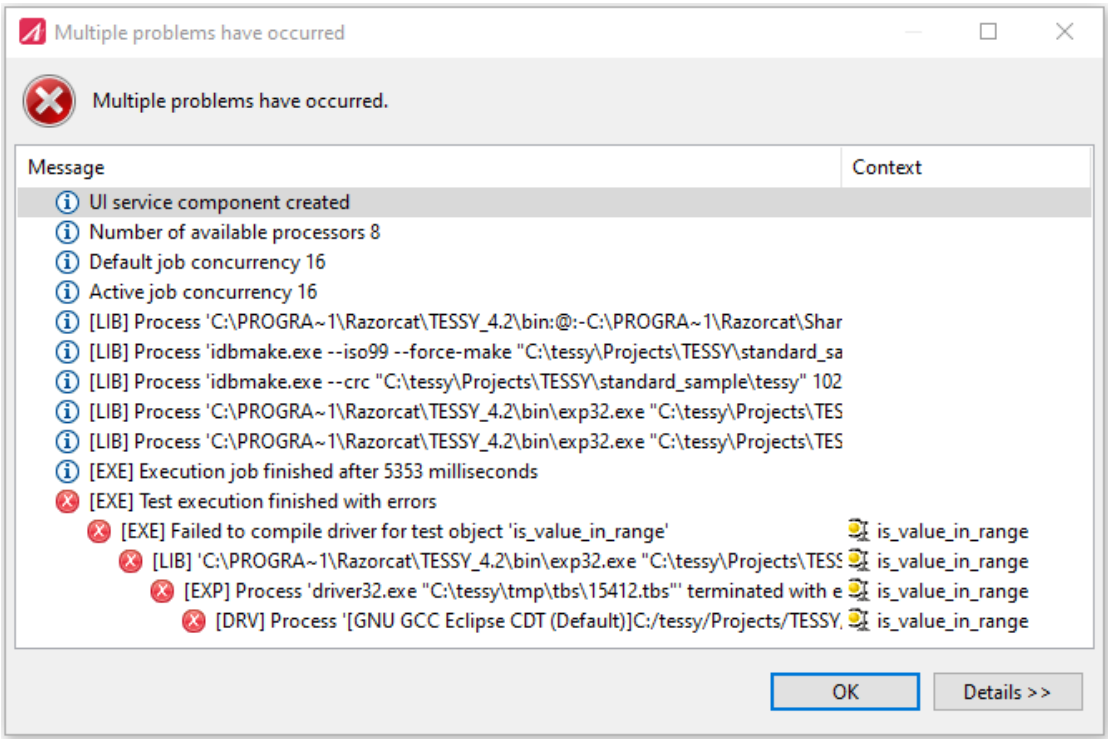


Figure 7.7: The Workspace Problems Log

7.2.3.2 Headless operation problems log

The other menu item “Open Problems Log …” opens the Windows file chooser so you can open a log from a remote continuous integration server (e.g. Jenkins) which was produced while running your current TESSY project. This allows you to analyze any errors or warnings from the Jenkins job and find the related TESSY objects within your project.

Please follow the steps described below to open the log file:

- → Run the command “*tessyd shutdown --copy-log <directory>*” as final part of your Jenkins build step.
  This will shutdown TESSY and copy the “problems.zlog” into the given directory. The directory must exist and should be located within the current Jenkins workspace.
- → Archive the contents of this directory as build artefact.
- → Download the artefact to your local computer.
  (There should be a “problems_<time stamp>.zlog” file.)
- → Select the “problems_<time stamp>.zlog” file within the file chooser dialog.

The current time stamp will be added to the “problems.zlog” file name when using the “–copy-log” option. This allows storing problem log files of multiple subsequent headless TESSY sessions into the same artefact directory.

For more information about the command line interface please refer to section 6.16 Command line interface).

The Problems Log dialog will now display all problems that occurred during the Jenkins job execution and you can find the related TESSY objects and the related console outputs for each individual test object execution error or warning.

7.3 Solutions for common problems

7.3.1 TESSY does not start or gives errors when starting

Error message:	Diverse errors, maybe none
Error description:	TESSY does not start or displays exceptions within all GUI windows (views).
Specific occurrence or requirement:	Error occurs when starting TESSY or while in use.
Possible cause:	There might be a problem due to corrupted files needed for the Eclipse TESSY product startup.

Table 7.4: Startup process problems

Solution:

Delete the following directories in given order. After every deletion try to start TESSY again. If it fails, delete the next directory.

Important: Close TESSY completely before deleting any of those directories! The following example path names reflect a TESSY version 4.0.5. You need to adjust the path names for the TESSY version you are currently using!

1. Delete folder `%userprofile%\.tessy_v4.0.5`.
2. Delete the file `%userprofile%\.tessy_40_workspace\preferences.xml`.

Important: This will reset your preference settings to the defaults!

3. Delete folder `%userprofile%\.tessy_40_workspace\.metadata`.

Important: This will reset your window layout of the GUI to the default settings!

4. Delete folder `%userprofile%\.tessy_v4.0.5`.

5. Delete folder `%userprofile%\.tessy_40_workspace`.

Important: After this you need to re-import all your projects into the project list again! The most simple way to do this is to double-click on the respective “tessy.pdbx” file.

7.3.2 TESSY gives errors when quitted

Error message:	***SQLException*** *SQL State: 08006* *Error Code: 40000* *Insufficient data while reading from the network - expected a minimum of 6 bytes and received only 0 bytes. The connection has been terminated.* ***SQLException*** *SQL State: 08001* *Error Code: 40000* *java.net.ConnectException: Fehler beim Herstellen der Verbindung zu Server localhost auf Port 1527 mit Meldung Connection refused: connect.*
Error description:	When quitting TESSY, one of the error messages above is displayed.
Specific occurrence or requirement:	Error occurs when quitting TESSY.
Possible cause:	Two versions of TESSY were started at the same time.

Table 7.5: Quit process problems

Solution:

If you want to use two different versions of TESSY at the same time, you can change the config file:

Important: The functions of the command line tool are limited if you use both TESSY versions at the same time! The tool works with the TESSY version that was started at last.

→ Open the config file of TESSY under
`%APPDATA%\Razorcat\Tessy\[TESSY Version]\config\tessy.conf`.
→ Remove the hashmark in front of the server port from the database.

```
# Port to use by derbyd
TESSY_DERBYD_PORT=40000
```

→ Choose different port numbers for each of the TESSY config files.

7.3.3 License server does not start or gives errors

Error message:	Diverse, e.g. *A license server is not running.*
Error description:	The license server does not start, or you get an error when starting it.
Specific occurrence or requirement:	-
Possible cause:	Corrupt/incorrect license key.

Table 7.6: License server problems

Solution:

Check your license key:

→ Start the License Manager manually if it has not started yet: “Start” > “All Programs” > “TESSY” > “Floating License Manager”.
→ Click on (Check) to check your license key file.
→ Check the error message (see figure 7.8).

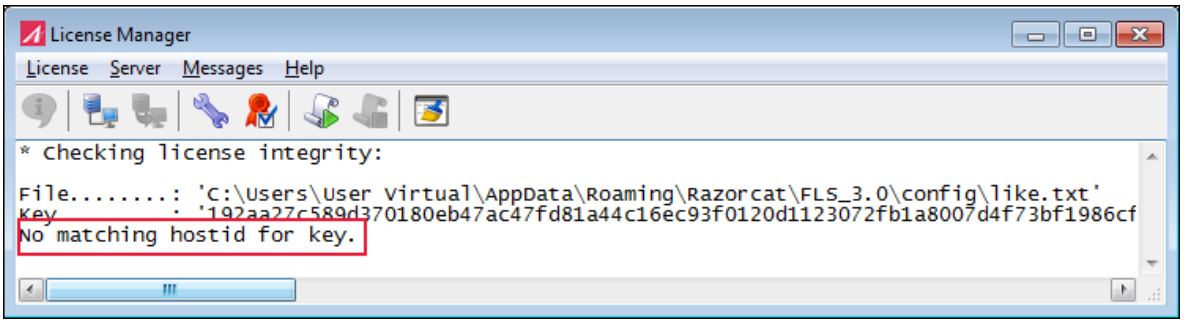


Figure 7.8: License key check unsuccessful: license key is incorrect for the host id

In many cases you can already determine the problem with the help of the error message. In case of the error “No maching hostid for key” the license key does not match to the host id of your computer:

- → Configure the correct license key file in the manager: Click on (Configure) and select the correct license key file. Click “OK”.
- → Click on to check the license key file again.
- → If the error still appears, contact our support (see Contacting the TESSY support) to get a new license key file.

7.3.4 Working with constant variables

Setting a variable declared with the “const” modifier keyword may result in undefined behavior and lead to error messages. In those cases set the variable passing to “IRRELEVANT” in the TIE. After that was done the test shall pass through without any restriction.

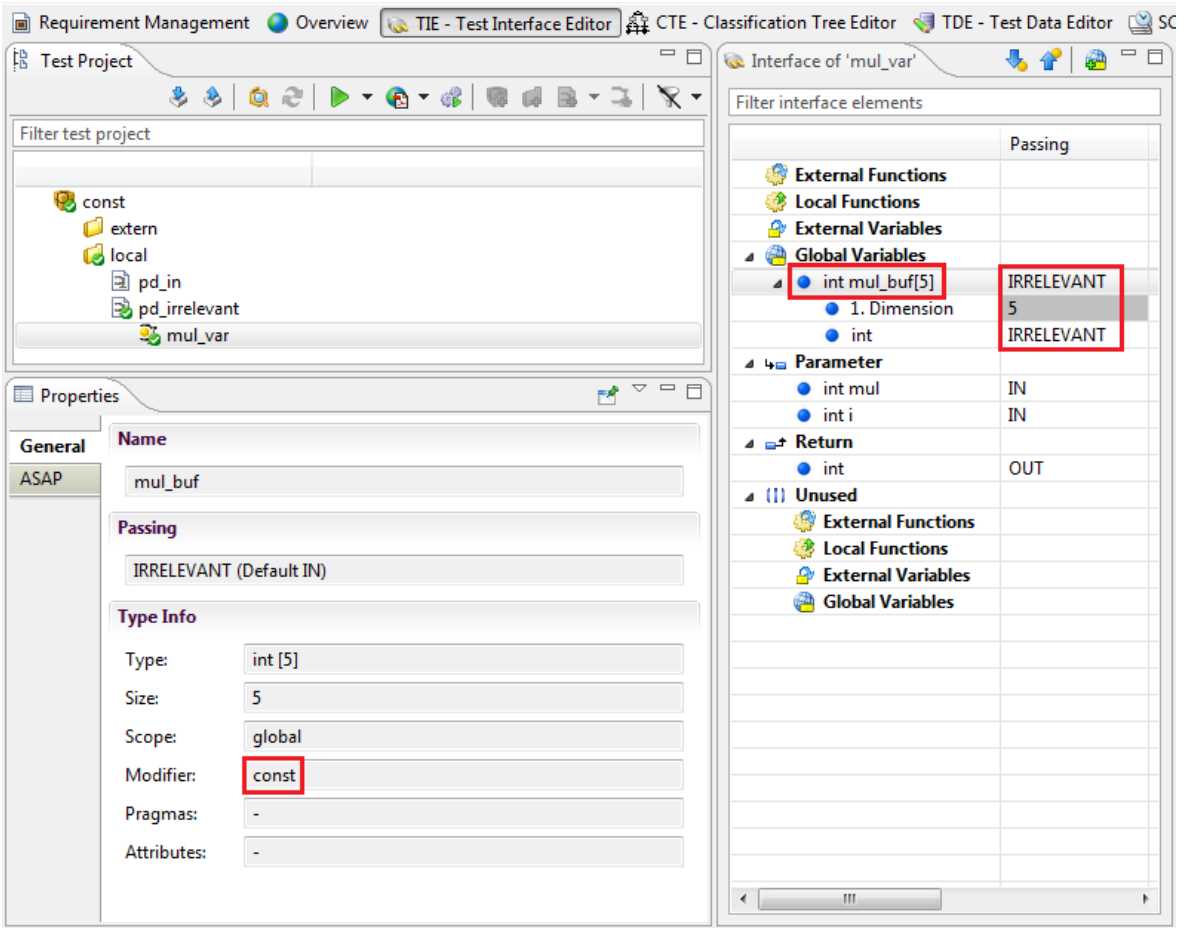


Figure 7.9: Variable passing set to “IRRELEVANT”

Important: Please note that normally constant variables are read-only variables and can not be assigned.

7.3.4.1 Assignment of read-only variables

If read-only variables need to be assigned, choose from the following solutions:

1. Undefine the const modifier in the properties view (for individual modules)

The modifier “const” needs to be removed in order to write to such variables. You can remove this modifier without changing the source file using a special define that replaces the “const” keyword with nothing.

→ In the Test Project view click on the module you want to test.

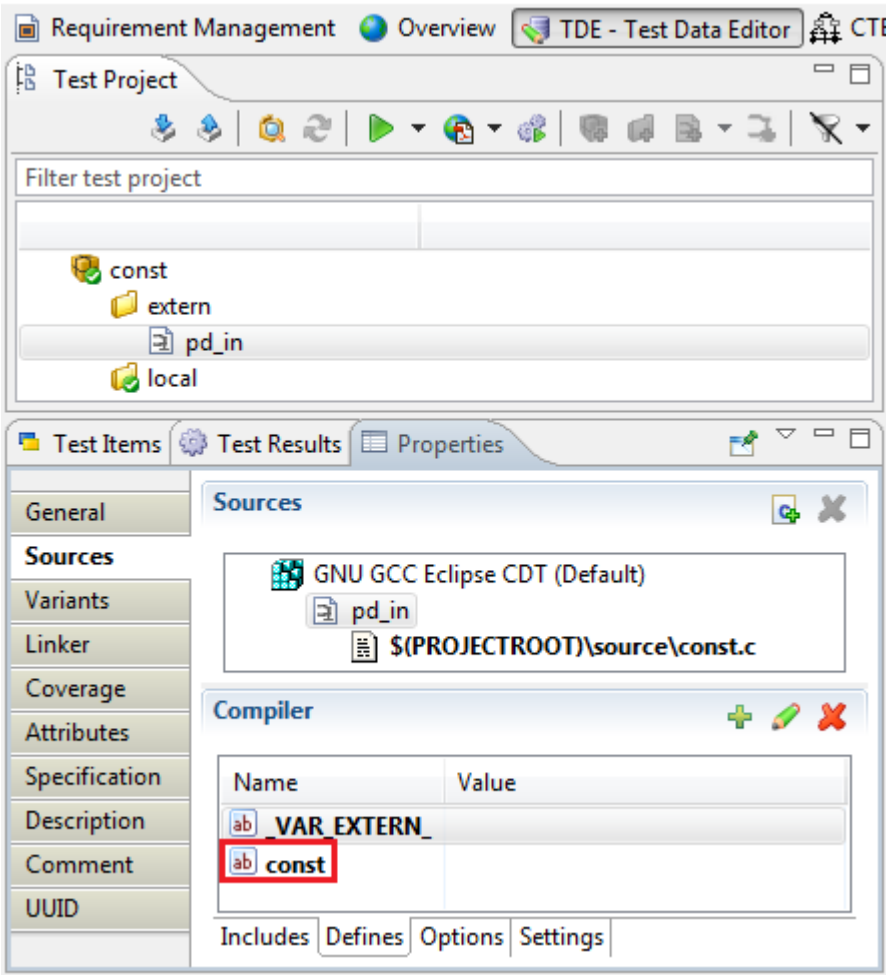


Figure 7.10: Properties view with undefined const modifier

→ To add a define that replaces the “const” keyword with an empty content click on ✚ (see figure 7.10) in the Properties view. The New Define popup window opens.

→ Enter a define with the name “const” and an empty value as shown below (see figure 7.11).

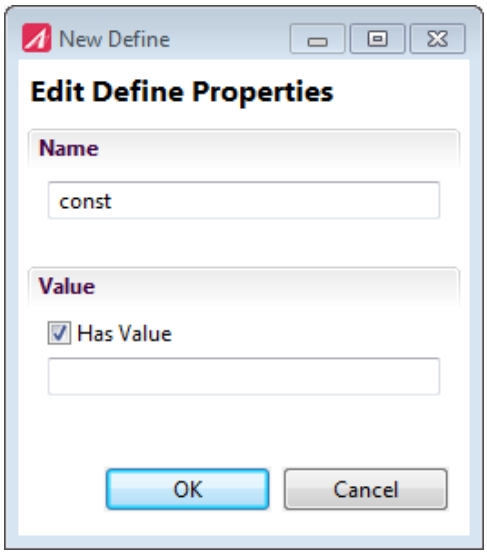


Figure 7.11: New Define popup window

Assignments to read-only variables are now possible in the chosen module. When this define is in place, all variables with the “const” modifier will appear as if the “const” has not been used (i.e. the variables are not “const” any more and can be changed during the test execution).

2. Modify the attribute “Compiler Defines” in the TEE (for global use)

→ In the menu bar click “File” > “Edit Environment…” .

→ In the TEE select your compiler (in this case it is GNU GCC), right-click in “Compiler Defines” > click “Edit Attribute Value”.

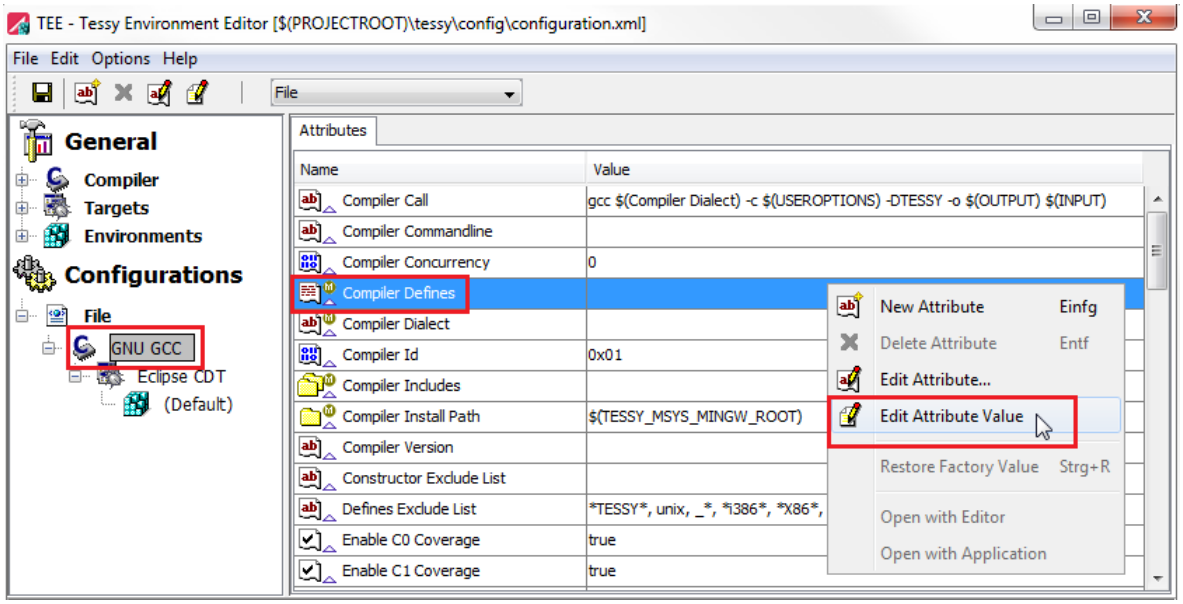


Figure 7.12: Edit the Attribute Value of the “Compiler Defines”

An “Edit Text” window opens.

→ Add “const=” in the “Edit text” window and click “OK”.

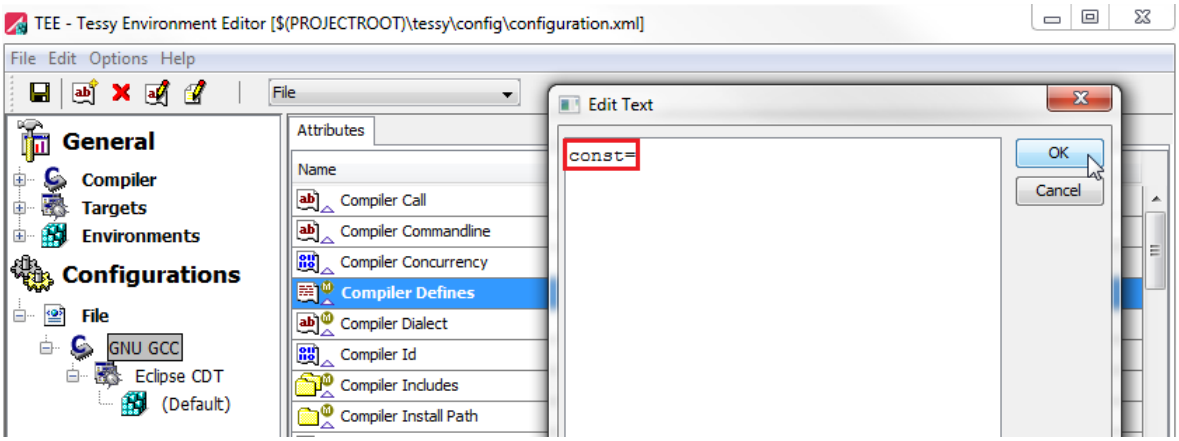


Figure 7.13: Add “const=” in the editor

→ Save the modifications when closing the TEE.

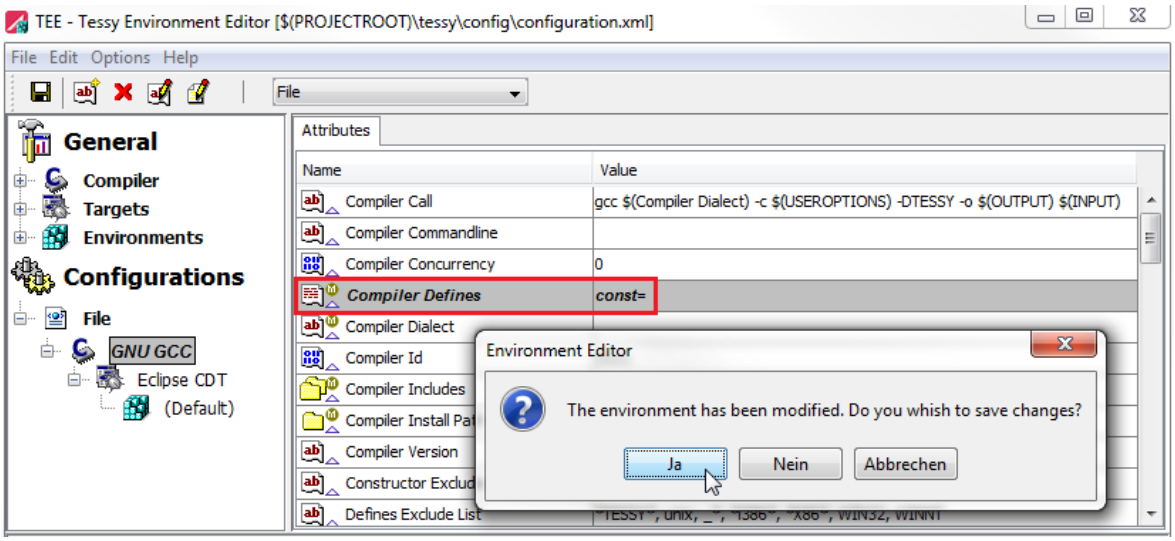


Figure 7.14: Save the “Compiler Defines” changed to “const=”

Constant variables are now generally replaced with an empty content.

7.3.5 Dealing with too long project paths

It is a known problem with TESSY that it leads to error messages if project paths are too long. This occurs due to Windows APIs used by TESSY, also some of the external compilers supported by TESSY show the same behavior. This can not be influenced by Razorcat.

There are two ways to avoid or deal with potential problems in relation to this:

- Store your TESSY project data in a particular project file higher up the path, e. g.: `C:\Projects\ProjectXY` instead of `C:\Projects\ProjectXY\QualityAssurance\Test\TESSY\RootOfProjectXY`.
- Shorten the too long project path with a virtual drive by using the Windows utility "subst".

To create a virtual drive with the "subst" utility:

→ Open the Windows command prompt.

→ Type "`subst T: C:\Projects\ProjectXY\QualityAssurance\Test\TESSY\RootOfProjectXY`".

Fore more information about creating a virtual drive refer to the Windows operating system help.

→ Open your project in TESSY using the new virtual drive, in this case: "`T:\`".

Important: Be aware that making use of a virtual drive will only work for the actual user. Therefore it is necessary that all users working on this project apply the same subst command on their computers.

Appendix

A Abbreviations

API	Application Programming Interface
C0	Statement Coverage
C1	Branch Coverage
CC	Cyclomatic Complexity
CTE	Classification Tree Editor
CV	Coverage Viewer
DC	Decision Coverage
DOS	Disk Operating System
EPC	Entry Point Coverage
ECU	Electronic Control Unit
FAQ	Frequently Asked Questions
FC	Function Coverage
GUI	Graphical User Interface
GUID	Global Unique Identifier
HTML	Hyper Text Markup Language
IDA	Interface Data Assigner
IEC	International Electrotechnical Commission
ISO	International Standards Organization
MC/DC	Modified condition/decision coverage
MCC	Multiple condition coverage
OBT	Original binary test
PDB	TESSY Project Database (TESSY Version < 3)
PDBX	TESSY Project Database (TESSY Version >= 3)
QTS	Qualification Test Suite
RQMT	Requirement
SCE	Scenario Editor
SIL	Safety Integrity Level
TBS	TESSY Batch Script
TC	Test Case
TD	Test Definition
TDB	Test Database

TDE	Test Data Editor
TEE	Test Environment Editor
THAI	TESSY Hardware Adapter Interface
TIE	Test Interface Editor
TMB	TESSY Module Backup
TS	Test Step
UCE	User Code Editor
XML	Extensible Markup Language

B Glossary

Batch Testing A testing procedure in which multiple test objects are executed automatically one after each other without further user interaction.

Branch Coverage Is usually abbreviated "C1". Roughly spoken: Branch Coverage reveals, if all branches were executed, for example, an if-instruction has two branches, the then-branch and the else-branch.

C1 Test During a C1 test, each branch of the test object will be instrumented with a counter to monitor, how often a branch of the program is run through.

Classification Tree The objective of the Classification Tree Method is to determine a sufficient but minimum number of test cases. It is a systematic approach to test planning by test case specifications and priorizations.

Code Coverage A test object is considered to consist of items like branches, conditions, etc. Code coverage measures, how many of the items were exercised during the tests. This number is related to the total number of items and is usually expressed in percent.
TESSY features C1 coverage (branch coverage) and C2 coverage (MC/DC: Modified Condition)

Component Functions are functions that are callable from external.

Component Testing is the test of interacting test objects, i.e. interacting functions in the sense of C. These functions can be a (single) calling hierarchy of functions, but we will consider this mainly as unit testing. We consider as a component mainly a set of functions that interact e.g. on common data and do not necessarily call each other. Component testing then is testing of such a set of functions. The units do not need to be in a calling hierarchy; they may only interact on data, like push() and pop() of the abstract data type "stack". A component according to this specification may also be called a "module", and its testing "module testing" respectively.

Debugger A computer program that is used to test and debug other programs (the “target” program). The code to be examined might alternatively be running on an instruction set simulator (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation - full or partial simulation, to limit this impact.

Enums Type of the C language specification which allows to define a list of aliases which represent integer numbers.

Expected Values Values expected to be calculated by the test object. The result values are checked against the expected values after the test run.

Fault Injection Fault injection is a technique to improve the coverage of tests by injecting faults to test code paths. The TESSY fault injection feature provides means to test code parts that are not testable using normal testing inputs e.g. endless loops, read-after-write functionality or error cases in defensive programming.

Flow Chart A flow chart is a special type of diagram used to outline process, workflow or algorithm. Various boxes represent the steps and their order is illustrated by connecting the boxes with arrows.

Hysteresis Dependence of a system not just on its current environment but also on its past. This dependence arises because the system can be in more than one internal state.

Interface Data Assign editor (IDA) If the interface elements of the test object have changed, you can assign the new interface elements to the old. Your test data will be assigned automatically.

Input Values Function parameters, global and external variables which have effect on the behavior of the function.

Interface Description Information about the passing direction and type of interface elements (parameter, global variables and external variables). The interface description is determined automatically by TESSY and is made visible / changeable in the TIE.

Integration Testing consists of a sequence of calls and can be considered either as unit testing for a calling hierarchy of functions or as a component testing for a set of interacting functions not necessarily calling each other. Component testing is integration testing of the functions in the component.

Modified Condition/Decision Coverage (MC/DC) MC/DC coverage takes the structure of a decision into account. Each decision is made up from conditions, which are combined by logical operators (and, or, not), n conditions require n+1 test cases. Roughly spoken, to get 100 percent MC/DC for a decision, each condition in the decision requires a pair of test cases, that

- differs in the boolean value for that condition, and
- has the same boolean value for all other conditions, and
- produces true and false in the outcome of the whole decision.

Metrics Calculation of the cyclomatic complexity (CC) is a common measure for complexity control. It measures the complexity of source code on the basis of the control flow graph and indicates the number of linearly independent paths through the code.

Module A TESSY module comprises primarily of the test object (in C a function in the sense of C) and source files, compiler settings, interface description and test data. You can pool modules in projects.

Module Testing is a semantic term for testing a collection of cooperating functions (units).

Output Values The same as an expected value in the TESSY context. Both terms are used in equivalence within this manual. The output (repectively expected) values are evaluated against the actual result values after the test run.

Project Root See explanatory box on page 57.

Regression Testing Regression testing is the repetitive running of already successfully completed test cases. The intention of regression testing is to verify that modifications and enhancements to a test object do not break the already successfully completed tests.

Requirement Documented need of what a test should perform and important input for the verification process. Requirements show what elements and functions are necessary for the test.

Requirement Management Management of different types of requirements that need to be covered by at least one test.

Requirement, Functional Describes the features, specific behavior, business rules and general functionality that the proposed system must support.

Requirement, Non-Functional Specifies criteria that can be used to judge the operation of the test.

Script Editor The TESSY script perspective provides textual editors supporting a test scripting language for editing test cases, test data and usercode. A new internally managed script file will be created for each test object and all test data and usercode can be edited and saved.

Search Query Search definitions processed by a search engine.

Stub Function Piece of code used to stand in for some other programming functionality. A stub may simulate the behavior of existing code (such as a procedure on a remote machine) or be a temporary substitute for yet-to-be-developed code.

System Testing Test of the application (software or software and hardware) as a whole.

Test Data Editor (TDE) With the TDE you can enter the input values and expected values for the test run.

TESSY Support File Contains information about test objects including data, compiler, project settings etc. It helps the support to detect the cause of a problem. In section Contacting the TESSY support it is explained how to create a TESSY Support File.

TESSY Hardware Adapter Interface (THAI) TESSY provides an hardware adapter interface to enable stimulation and measurement of hardware signals as well as execution time measurement during the unit test execution.

Test Case Element that encapsulates the abstract test definition, e.g. the specification and description of a test, and the concrete test data managed within test steps.

Test Definition Describes a test to be performed on the test system in textual format. A test definition abstractly describes the inputs and the expected outcome of a test and refers to a list of requirements which shall be validated with this test.

Test Driver C-source files generated by TESSY for the test execution. These files are compiled and linked in order to build an application that prepares the input data, call the test object and store the actual result data.

Test Environment Information about the test object, the compiler used, the target debugger or emulator and more settings.

Test Object The function to be tested.

Test Run One execution of a test object with the given test cases. The result of a test run is stored within an XML result file that may be further processed by external tools.

Test Suite A collection of test objects with test scenarios and/or test cases that were created to fulfill a certain test objective.

Test Interface Editor (TIE) With the TIE you can view all interface elements and review or set the passing direction and/or other information of the interface elements.

Unit A single function, i.e. test object of a C program single function; the smallest reasonable test object of a C program.

Usercode In the usercode you can enter C code, which is executed before or after test cases/test steps during the execution of a test object.

Workspace The space at local disk where the TESSY application reads and writes data. Place for configuration and temporary report data. Project data can be saved separately.

C List of Figures

0.1 Core workflow of TESSY . . . xxi
0.2 UUID property of TESSY objects . . . xxviii
0.3 Exclude test items from the test . . . xxix
0.4 Static analysis metrics in the Test Project view . . . xxx
0.5 Fault injection editing using the Coverage Viewer . . . xxxi
0.6 Script editor with outline view . . . xxxii
0.7 The new Classification Tree editor (CTE) . . . xxxiii
0.8 Comparison of test versions using a diff tool . . . xxxiv
0.9 Automatically generated classification tree and test cases . . . xxxv
0.10 Call pair coverage view with coverage results . . . xxxvi
0.11 Estimated and actual time within the test project view . . . xxxvii
0.12 A review task within the test project view . . . xxxviii
0.13 Task elements can be linked to requirements using the link matrix . . . xxxviii

1.1 InstallAware Wizard . . . 3
1.2 Destination Folder . . . 4
1.3 TESSY Testarea Folder . . . 4
1.4 Program folder and user selection . . . 5
1.5 Installing process of TESSY . . . 5
1.6 Installation is completed . . . 6
1.7 Starting TESSY 4.x . . . 6
1.8 Starting the key request . . . 7
1.9 License key request popup window . . . 8
1.10 Form for the license key request . . . 8
1.11 Configure menu of the license server . . . 11
1.12 Settings for a floating license in the configure menu . . . 13
1.13 Dialog window to select license server in a network . . . 14
1.14 License key check successful: this license key is correct . . . 16
1.15 Uninstalling FLS and Shared installation files . . . 17
1.16 The license info shows the possible number of days for checking out the license 19
1.17 Checking out the TESSY license . . . 20

1.18 Determine the amount of days for the check-out . . . 20
1.19 Transmitting the license file to a computer with no FLS connection . . . 21

2.1 Updating a project . . . 22

3.1 Initial equivalence partitioning for “ice warning” . . . 35
3.2 Repeated equivalence partitioning for “ice warning” . . . 36
3.3 A possible CT for “ice warning” . . . 37
3.4 Result of the CTM: tree (above) with combination table (below) . . . 38
3.5 The problem “is_value_in_range” depicted graphically . . . 39
3.6 The initial CT with three test-relevant aspects . . . 40
3.7 The CT for is_value_in_range, 2nd step . . . 40
3.8 The CT for is_value_in_range, 3rd step . . . 41
3.9 A first specification for the range in the combination table . . . 41
3.10 A second specification for the range in the combination table . . . 42
3.11 The CT for is_value_in_range, 4th step . . . 43
3.12 The third range specification provokes a wrap-around . . . 44
3.13 The completed CT for is_value_in_range . . . 45
3.14 The completed test case specification . . . 47
3.15 An alternative test case specification . . . 49

4.1 Path of the workspace . . . 54
4.2 Creating a new project . . . 55
4.3 Selecting a folder for the project root. . . . 56
4.4 Creating a new project . . . 59
4.5 Context menu of the Select Project dialog . . . 61
4.6 Project identifier handling . . . 62
4.7 Project Example1 is created. . . . 62
4.8 TESSY notifies, that a database update is necessary . . . 65
4.9 A database update is necessary . . . 65
4.10 TESSY interface and its terminology . . . 66
4.11 Open other perspectives . . . 68
4.12 Test Project view within the Overview perspective . . . 68
4.13 Adding views to a perspective . . . 69
4.14 Move the views separately. To switch back, use “Reset”. . . . 70
4.15 To switch back all positions of views and perspectives use “Reset Workbench”. . . . 70
4.16 Minimizing and maximizing views . . . 71
4.17 Maximized view with minimized views on the right and the restore-button on the left . . . 71
4.18 Using the context menu with a right click. . . . 73

5.1 Operational sequences in TESSY . . . 78
5.2 Creating the new project “Example1”. . . . 80
5.3 New project “Example1” is created . . . 80
5.4 Test collection “Is_value_in_range” with an example folder and module . . . 81
5.5 GNU GCC compiler is selected by default. . . . 82
5.6 The source code of the C-Function to be tested . . . 82
5.7 Adding the C-source file. . . . 83
5.8 Analyzing the module, that is the C-source file. . . . 84
5.9 The function of the C-source is displayed as child of the module. . . . 84
5.10 Perspective TIE - Test Interface Editor . . . 85
5.11 The inputs and outputs are already defined . . . 86
5.12 Test Items view . . . 87
5.13 Three test cases were added in the Test Items view . . . 87
5.14 Data is entered, test step turns yellow and test case is ready to run. . . . 89
5.15 Entering data for test object is_value_in_range . . . 90
5.16 The test cases are ready to test . . . 91
5.17 TDE after test run is_value_in_range . . . 92
5.18 Test results of is_value_in_range . . . 92
5.19 Selecting Branch and MC/DC Coverage for test run . . . 93
5.20 Execute Test dialog while running the test . . . 93
5.21 The Coverage Viewer displays the coverage of is_value_in_range . . . 94
5.22 Branch coverage is_value_in_range . . . 95
5.23 Decision coverage is_value_in_range . . . 96
5.24 Code section of the if branch of the first decision . . . 97
5.25 Code section of the second decision . . . 98
5.26 Selecting a folder or creating a new folder for Test Details Reports . . . 99
5.27 Content of the test report is_value_in_range . . . 100
5.28 The Test Report Options in the Preferences . . . 101
5.29 Importing a requirement . . . 103
5.30 Import dialog . . . 104
5.31 The new requirement document . . . 104
5.32 Changing the alias of the new requirement document . . . 105
5.33 Comment for the initial revision of the commit . . . 105
5.34 Linking test cases with requirements . . . 107
5.35 Test Definition view within TDE with linked requirement . . . 108
5.36 Editing the settings of a Planning Coverage Report . . . 108
5.37 Dialog of the settings for the Planning Coverage Report . . . 109
5.38 Planning coverage report of the IVIR requirement document . . . 110
5.39 Generating a Test Details Report . . . 111

5.40 Part of the generated test report of is_value_in_range . . . 112
5.41 Creating an Execution Coverage Report . . . 112
5.42 Coverage Report of is_value_in_range . . . 113
5.43 Overview perspective after test run (with requirements) . . . 115
5.44 Use the context menu to edit a source . . . 116
5.45 Editing the C-source file is_val_in_range.c . . . 116
5.46 Changed C-source file of is_value_in_range . . . 117
5.47 Adding a “delete” and “new” object . . . 117
5.48 Changed and new test objects of is_value_in_range . . . 118
5.49 Remove the code for test object “deleted”. . . . 119
5.50 Changed and new test objects of is_value_in_range . . . 119
5.51 Changed, deleted and new test object of is_value_in_range . . . 121
5.52 Use drag and drop in IDA . . . 121
5.53 Automatically generated tree with the root “is_value_in_range” in the CTE perspective . . . 123
5.54 Interface elements categorized into ”Inputs“ and ”Outputs“ . . . 124
5.55 Child elements of an atomic type on the inputs side of the subtree . . . 125
5.56 The outputs subtree . . . 125
5.57 CTE tree area and Test Data view . . . 127
5.58 Modify class elements . . . 129
5.59 Deleting elements . . . 130
5.60 Creating test cases in the CTE . . . 131
5.61 Automatically generated tree with 9 Test Cases . . . 132
5.62 Defining test cases in the combination table of CTE . . . 133
5.63 Completed table with all test cases for example “is_value_in_range” . . . 133
5.64 Test data is displayed when selecting a test case in the combination table . . . 134
5.65 Test data displayed within TDE . . . 135
5.67 Example interior_light; ECU = Electronic Control Unit . . . 137
5.68 Test Project view with new project interior_light . . . 138
5.69 Selecting “Component” in the module properties . . . 139
5.70 Scenario of a component test . . . 139
5.71 C-source code interior_light . . . 141
5.73 If a heartbeat function exists, timely behavior can be tested. . . . 142
5.74 The initial interface . . . 143
5.75 Creating the stubs . . . 144
5.76 The final passing directions of variables used by init() . . . 144
5.77 Test Project view with a component test . . . 145
5.78 Adding a description to test cases . . . 145
5.79 Inline editor of the view Test Data of ’Scenarios’ . . . 146

5.80 View Test Data of 'Scenarios' . . . 146
5.81 View External Function Calls . . . 147
5.82 The names of the test cases are displayed in tabs of the view Work Task . . . 147
5.83 Setting the Work Task . . . 148
5.84 Dragging the function onto the scenario . . . 149
5.85 set_sensor_door() is dragged to 30 ms simulated time . . . 149
5.86 The parameter of set_sensor_door() is set . . . 150
5.87 Dragging the function . . . 150
5.88 Extending the time frame . . . 151
5.89 Setting the call "LightOff" and extending the time frame . . . 152
5.90 Setting values to "ignore" . . . 153
5.91 Designing the second scenario . . . 154
5.92 The scenarios of the component "passed" the test . . . 155
5.93 Importing a project . . . 156
5.94 Cloning the project. . . . 157
5.95 Cloning the project. . . . 157
5.96 The project root is displayed within the bottom line of TESSY. . . . 158
5.97 Restoring the database . . . 159
5.98 Test project view with the C++ project . . . 159
5.99 Adding synthetic test objects. . . . 160
5.100 Adding synthetic variables. . . . 161
5.101 Creating test cases for TDD. . . . 161
5.102 Filling test cases for TDD. . . . 162
5.103 Test specification report for TDD. . . . 162
5.104 Implementation source file available for TDD. . . . 163
5.105 Assignment of synthetic test objects to the implemented functions. . . . 163
5.106 Readily assigned TDD test cases. . . . 164

6.1 Menu bar of TESSY . . . 171
6.2 Preferences menu of TESSY . . . 173
6.3 Static Analysis in the Preferences menu . . . 177
6.4 Pre-defined coverage instrumentation settings . . . 178
6.5 Pre-defined coverage metrics settings . . . 179
6.6 Interface dictionary within the Preferences . . . 180
6.7 Editing variables in the interface dictionary . . . 181
6.8 Interface dictionary variable with warning icon . . . 181
6.9 Overview perspective . . . 183
6.10 Test Project view within the Overview perspective . . . 184
6.11 Editing the Task settings . . . 188

6.12 Executed task “Checklist” (Passed) in the Test Project view . . . 189
6.13 Task “Checklist” linked to multiple requirements in Link Matrix . . . 189
6.14 Test Execution Settings in the Preferences . . . 190
6.15 Test Project view with historical test runs . . . 191
6.16 The function of the C-source is displayed as child of the module . . . 192
6.17 Warning when opening older modules with CLANG enabled . . . 192
6.18 Static code analysis in the Test Project view . . . 195
6.19 Select “Estimated Time” and “Actual Time” in the Preferences . . . 197
6.20 Editing the actual time within the AT . . . 199
6.21 Create Variant Modules . . . 200
6.22 Selecting the parent module of the variant . . . 201
6.23 Test Project view with a module and a variant module. . . . 201
6.24 The variant module needs to be synchronized with the parent. . . . 202
6.25 Synchronizing a module with the parent . . . 202
6.26 Synchronize Module dialog . . . 203
6.27 Interface assignment needed . . . 203
6.28 Test cases and test steps that were inherited of a variant module . . . 205
6.29 Add notes via content menu . . . 206
6.30 The Notes view in the Overview perspective . . . 206
6.31 Editing notes in the Notes view . . . 207
6.32 Test Execution Settings . . . 210
6.33 Coverage displayed within the Test Project view . . . 211
6.34 Opening the filter settings . . . 213
6.35 Filter Configuration Dialog . . . 213
6.36 A filter has been set but is currently disabled (filtered test objects appear faded). 214
6.37 The Filter is enabled, the affected test objects are hidden. . . . 214
6.38 Search filter function of the Test Project view . . . 215
6.39 Searching for “global” . . . 215
6.40 Creating a report . . . 218
6.41 Test Details Report Settings dialog with default and optional settings . . . 219
6.42 Test Overview Settings dialog with default and optional settings . . . 220
6.43 Planning Coverage Settings dialog with default and optional settings . . . 221
6.44 Execution Coverage Settings dialog with default and optional settings . . . 222
6.45 Context menu “Define Batch Operation” . . . 223
6.46 Defining the batch operation . . . 224
6.47 Selecting all text objects . . . 224
6.48 Editing the settings of each batch operation . . . 225
6.49 Saving the settings of a batch operation as TBS file . . . 226
6.50 Import settings of data import . . . 227

6.51 Export settings of data export . . . 228
6.52 Properties view . . . 228
6.53 The Compiler pane in the Sources tab of the Properties view . . . 230
6.54 The Setting tab of the Properties view with module selected . . . 232
6.55 The Linker Options tab of the Properties view . . . 233
6.56 The Coverage tab of the Properties view . . . 233
6.57 Selection of the coverage measurement in the pull-down menu . . . 234
6.58 Defining coverage measurements . . . 235
6.59 The Attributes tab of the Properties view . . . 236
6.60 Creating a new attribute . . . 237
6.61 Requirements Coverage view . . . 238
6.62 Test Items view . . . 239
6.63 First test case with one test step . . . 242
6.64 Selecting the test case generator . . . 243
6.65 A new test case generator is created . . . 243
6.66 The test steps were generated and are ready to be executed . . . 244
6.67 Selecting “Change Test Case Type to Normal” . . . 245
6.68 The test case and test steps originally being generated . . . 246
6.69 Test cases created by CTE . . . 246
6.70 Test cases and test steps that were inherited of a variant module . . . 247
6.71 All test cases will be renumbered . . . 248
6.72 Test Results view . . . 248
6.73 Evaluation Macros view . . . 249
6.74 Console view . . . 249
6.75 Preference “Show console on error” . . . 250
6.76 Problems view with error message . . . 251
6.77 Variants View . . . 252
6.78 Assign a variant to a module . . . 252
6.79 Example test collection with base modules . . . 253
6.80 Create variant modules dialog: Filtering and selection . . . 253
6.81 Test Project view with new variant modules . . . 254
6.82 Properties view variants tab for editing the parent module . . . 254
6.83 Perspective C/C++ . . . 256
6.84 Opening the C/C++ perspective . . . 257
6.85 Editor view within the C/C++ perspective . . . 258
6.86 Opening the C/C++ perspective . . . 259
6.87 Project Explorer view within the C/C++ perspective . . . 260
6.88 Outline view within the C/C++ perspective . . . 261
6.89 Requirement Management perspective . . . 264

6.90 RQMT Explorer view . . . 265
6.91 Double-clicking on a requirement opens the requirement editor . . . 266
6.92 Example for the document structure within the RQMT Explorer view . . . 268
6.93 Importing requirements . . . 269
6.94 Import dialog . . . 269
6.95 The new requirement document . . . 271
6.96 The asterix and a mouseover shows the status “new”. . . . 271
6.97 Committing options . . . 272
6.98 Comment for the initial revision of the commit . . . 272
6.99 After the commit . . . 273
6.100 Changing the alias of the new requirement document . . . 273
6.101 The alias of a requirement is used in various views . . . 274
6.102 Requirements List view . . . 274
6.103 Double-clicking on a requirement opens the Requirement Editor . . . 275
6.104 Requirements Editor with test and a figure . . . 276
6.105 The first requirement was modified . . . 277
6.106 The first requirement has the version number 2.0 . . . 278
6.107 VxV Matrix view . . . 278
6.108 Test Means view . . . 279
6.109 Link Matrix view . . . 280
6.110 Adding elements to the Link Matrix view . . . 282
6.111 Adding Test Cases to the Link Matrix view in the Overview perspective . . . 283
6.112 Link Matrix view with suspicious elements . . . 284
6.113 Suspicious Elements view . . . 285
6.114 Suspicious test object and test cases in the Overview perspective . . . 286
6.115 Suspicious test object and linked modified requirements . . . 287
6.116 Selecting the suspicious test case shows the modified requirement(s) . . . 288
6.117 Comparing the versions of the requirement . . . 289
6.118 Attached Files view . . . 290
6.119 Attributes view . . . 291
6.120 Editing the requirement settings within the Attributes view . . . 292
6.121 Changing the “Content Type” attribute to HTML . . . 293
6.122 History view . . . 294
6.123 Differences view . . . 295
6.124 Related Elements view and its interrelations . . . 296
6.125 Related Elements view . . . 297
6.126 Related Elements view with Incoming and Outgoing Links . . . 297
6.127 View Document Preview . . . 298
6.128 Newly opened Document Preview within the TIE perspective . . . 299

6.129 HTML editing within the inline editor (WYSIWYG and plain HTML) 300
6.130 Requirements Coverage view with no linked requirements 301
6.131 Setting or disabling the options of auto refreshing 302
6.132 Test Environment Editor (TEE) 306
6.133 Call of the Test Environment Editor (TEE) 308
6.134 Configuration file 309
6.135 Visualizing all items within TEE 313
6.136 Compiler and targets possible to use in TESSY 313
6.137 Enabled and disabled configurations within TEE 314
6.138 Attribute settings of the Eclipse target 314
6.139 Enable compiler 316
6.140 Adding an environment to a configuration file by using the context menu 318
6.141 Adding an environment to a configuration file by drag and drop 318
6.142 Integration of a hardware adapter (e.g. GAMMA) into the TESSY unit test execution 323
6.143 XML data structure for the configuration THAI 324
6.144 Enable THAI in the Properties view 325
6.145 Required THAI attributes in the Attribute View 326
6.146 Required THAI attributes in the TIE 327
6.147 Required THAI attributes in the TDE 328
6.148 Perspective TIE - Test Interface Editor 329
6.149 Information of passing direction and type 331
6.150 Interface view 331
6.151 White arrow indicating further levels, black arrow when expanded 333
6.152 Resetting passing directions 336
6.153 Setting the data format 337
6.154 Array as pointer 339
6.155 Create a stub function within the context menu 340
6.156 Create a new variable 343
6.157 Example code snippet for alias names 344
6.158 Show alias names preferences 345
6.159 Defined external variables 346
6.160 Undefining an external variable 346
6.161 Undefined external variable 347
6.162 Change external variable/function settings in the TEE 348
6.163 List of unused functions and variables in the TIE interface 350
6.164 Plot Definitions view 351
6.165 Rename a new plot 353
6.166 Plot Definitions menu 353
6.167 Adding variables to a plot in the TDE 354

6.168 Using plots in report . . . 355
6.169 CTE perspective . . . 356
6.170 Classification Tree editor related tool bar . . . 358
6.171 Classification Tree editor . . . 358
6.172 Creating a new classification with the context menu . . . 362
6.173 Creating test cases in the test item list . . . 363
6.174 Setting marks in the Test Table . . . 364
6.175 Classification Tree with test data for class “Zero” . . . 365
6.176 Test cases and test steps created within the CTE in the Test Item view of the Overview perspective . . . 366
6.177 An overview on the automated tree generation based on the function interface . 367
6.178 Automatically generated for the example is_value_in_range . . . 368
6.179 The CTEX file attribute . . . 369
6.180 Remove Test Specification . . . 369
6.181 The CTE Preferences . . . 370
6.182 CTE class node with children associated with test data . . . 371
6.183 Attach the selected interface object . . . 372
6.184 Detach the interface from an CTE note . . . 374
6.185 Showing data of a tree node . . . 375
6.186 Variable assignments in classification trees . . . 378
6.187 Attaching an interface element to a tree node . . . 380
6.188 TIE icon of a CTE node . . . 380
6.189 Dependency defined between “range_length -> negative” and “position -> outside” 382
6.190 Dependencies in the Properties View . . . 383
6.191 Dependency defined between “range_length -> zero” and “position -> inside” . 384
6.192 Composite dependency . . . 385
6.193 Composite dependency in the Properties View . . . 386
6.194 TDE perspective . . . 387
6.195 Type information of the variable long range_start . . . 390
6.196 Test Data view . . . 391
6.197 Test step 1.1 is selected and undefined values are highlighted in yellow. . . . 394
6.198 Test Data view showing selected test steps. . . . 395
6.199 Clicking in the cell shows a combo box with the union components . . . 398
6.200 Clicking in the cell shows a combo box with the available enum constants . . . 399
6.201 Pressing Ctrl + Space opens a list of available defines or enum constants . . . 400
6.202 Arithmetic expression . . . 400
6.203 Entering values as vector for an advanced stub . . . 402
6.204 Choosing shown arrays . . . 402
6.205 Passing direction set to irrelevant. . . . 405

6.206 Entering evaluation mode “unequal” within the inline editor . . . 406
6.207 Generator test case 4 has a range value from 6 to 9 for parameter v1 . . . 408
6.208 Four test steps are generated for every value within the range “6 to 9”. . . . 409
6.209 Selecting “Change Test Case Type to Normal” . . . 410
6.210 The test case and test steps originally being generated. . . . 411
6.211 Inherited value coloring within Test Data view . . . 412
6.212 Test Definition view within TDE with linked requirement . . . 415
6.213 Call Trace view . . . 416
6.214 Declarations/Definitions view . . . 417
6.215 Prolog/Epilog view . . . 418
6.216 Call sequence of the usercode parts . . . 418
6.217 TESSY provides default prolog/epilog on test object level to be inherited to test cases and test steps . . . 420
6.218 TESSY allows Prolog/Epilog being inherited from test case or test object . . . 421
6.219 Prolog/Epilog functions . . . 422
6.220 Editing C code . . . 423
6.221 Call the popup menu by pressing CTRL + space . . . 424
6.222 Editing the evaluation macro templates . . . 424
6.223 Formatting of evaluation macro values . . . 426
6.224 Stub Functions view without contents . . . 427
6.225 Test execution direction using stub code . . . 427
6.226 TESSY Preferences: Abort on missing stub code . . . 428
6.227 Stub Functions view with code using TS_CALL_COUNT macro . . . 429
6.228 Stub Code Levels in the Usercode Outline view . . . 430
6.229 Stub code examples on test object, test case and test step level . . . 431
6.230 Automatically Generated Test Code . . . 431
6.231 Usercode Outline view . . . 432
6.232 Usercode Outline view showing inherited stub code . . . 433
6.233 Usercode Outline view showing inserted stub code . . . 433
6.234 Plots view . . . 434
6.235 The Script Editor view. . . . 436
6.236 Status indicator within the editor title . . . 438
6.237 Merge dialog in the Compare view . . . 440
6.238 Perspective CV - Coverage Viewer . . . 441
6.239 Results of the EPC are displayed within the Test Project view . . . 443
6.240 Coverage results within the CV perspective (component testing) . . . 444
6.241 Test Project view within the CV perspective . . . 445
6.242 Called Functions view . . . 446
6.243 Flow Chart view . . . 447

6.244 Source code view on the bottom right of the Coverage View perspective 450
6.245 Condition view showing the sub flow coverage for one test case 454
6.246 Unreached code branch is marked blue 455
6.247 Statement coverage 456
6.248 Branch coverage 458
6.249 MC/DC Coverage view 459
6.250 MC/DC coverage 460
6.251 Call pair coverage view with coverage results 461
6.252 IDA perspective 463
6.253 Compare view 466
6.254 Use drag and drop in IDA 467
6.255 Perspective SCE - Scenario Editor 469
6.256 Component test 471
6.257 Scenarios of a component test 471
6.258 Interface of the scenarios 472
6.259 Two component functions were set as work task within the Component Functions view 473
6.260 Work Task Configuration view 474
6.261 Calculated cycle time 475
6.262 Adding Function Calls 477
6.263 The Test Data view of 'Scenarios' 478
6.264 Indicator icons for the test data 479
6.265 A function is not called 481
6.266 Fault injection in the Flow Chart view of the CV 483
6.267 Fault Injection not found 484
6.268 Fault injection test case 485
6.269 The Edit Fault Injection dialog 486
6.270 Include list at the bottom of the Fault Injection dialog 487
6.271 Example of the Fault Injections report 488
6.272 Save Database dialog 490
6.273 Files of the backup 491
6.274 Restore Database dialog 492
6.275 Directories and files within the database directory of the TESSY project 494
6.276 DOS command line shell 497

7.1 Dialog for creating the TESSY Support File 504
7.2 Dialog for logging settings 505
7.3 Problems Log dialog 507
7.4 Problems Log dialog with details and context menu for individual log entries . . 508

7.5 Problems view with context menu 509
7.6 Help menu with problem menu items 510
7.7 The Workspace Problems Log 510
7.8 License key check unsuccessful: license key is incorrect for the host id 515
7.9 Variable passing set to “IRRELEVANT” 516
7.10 Properties view with undefined const modifier 517
7.11 New Define popup window 518
7.12 Edit the Attribute Value of the “Compiler Defines” 518
7.13 Add “const=” in the editor 519
7.14 Save the “Compiler Defines” changed to “const=” 519

D List of Tables

0.1 Where to find - matters of the several parts of the TESSY manual xviii
0.2 Font characters xix

1.1 Functions of the FLM 15
1.2 Information about licenses in use in the License Manager 19

4.1 File system and databases of TESSY 54
4.2 Options of the Project Configuration dialog 57
4.3 Tool bar options of the Select Project dialog 60
4.4 Handling of projects with equal names 63
4.5 Shortcuts and key functions 74

5.1 Entering data for test object is_value_in_range. 89
5.2 Meaning of the Test Coverage States 114

6.1 File menu options 172
6.2 Window menu options 172
6.3 Preferences menu options 176
6.4 Help menu options 182
6.5 Structure of the Overview perspective 184
6.6 Tool bar icons of the Test Project view 185
6.7 View icons of the Test Project view 186
6.8 Status indicators of the Test Project view 187
6.9 Status indicators for historical test runs 191
6.10 CLANG parser options and descriptions 193
6.11 Predefined tokens of the available metrics 198
6.12 Status indicators of inherited test cases or test steps 204
6.13 Test Execution Settings - Actions 209
6.14 Test Execution Settings - Options 209
6.15 Reports available within TESSY 217
6.16 Import/export selections 226
6.17 General tab of the Properties view 229

6.18 Optional functions of the Sources tab of the Properties view . . . 232
6.19 Tool bar icons of the Test Items view . . . 239
6.20 Column indicators of the Test Items view . . . 240
6.21 Status indicators of the Test Items view . . . 241
6.22 Tool bar icons of the Console view . . . 250
6.23 Icons of the content menu . . . 251
6.24 Structure of the C/C++ perspective . . . 257
6.25 Tool bar icons of the Outline view . . . 261
6.26 Structure of the Requirement Management perspective . . . 265
6.27 Tool bar icons of the RQMT Explorer view . . . 267
6.28 Status indicators of the RQMT Explorer view . . . 267
6.29 Possible formats of requirement sources . . . 270
6.30 Tool bar icons of the Requirements List view . . . 275
6.31 Tool bar icon of the Requirements List view . . . 275
6.32 Tool bar icons of the VxV Matrix view . . . 279
6.33 Tool bar icons of the Test Means view . . . 280
6.34 Tool bar icons of the Link Matrix view . . . 281
6.35 Status indicators of the Suspicious Elements view . . . 281
6.36 Tool bar icons of the Suspicious Elements view . . . 286
6.37 Tool bar icons of the Attached Files view . . . 290
6.38 Tool bar icons of the Attributes view . . . 291
6.39 Tool bar icons of the History view . . . 294
6.40 Tool bar icons of the Differences View . . . 296
6.41 Tool bar icons of the Document Preview . . . 300
6.42 Tool bar icons of the Requirements Coverage view . . . 302
6.43 Tool bar icons of the Planning tab . . . 303
6.44 Tool bar icons of the Execution tab . . . 303
6.45 Structure of TEE . . . 308
6.46 Tool bar icons of the TEE . . . 310
6.47 Status indicator example . . . 311
6.48 Attribute fonts . . . 312
6.49 Contents, functions and storage location of configuration files . . . 315
6.50 Meanings of flags in the attribute properties . . . 322
6.51 THAI attributes and their descriptions . . . 327
6.52 Structure of TIE . . . 330
6.53 Icons of the Interface view . . . 332
6.54 View icons of the Interface view . . . 332
6.55 Status indicators of the Interface view . . . 333
6.56 Classification sections of interface elements . . . 334

6.57 Possible passing directions of the interface elements . . . 335
6.58 Default passing directions . . . 336
6.59 Icons of the Plot Definitions view . . . 352
6.60 Drag and drop handling with the Plots and Plot Definitions view . . . 354
6.61 Structure of the CTE perspective . . . 357
6.62 Tool bar icons of the Classification Tree view . . . 359
6.63 Structure of Classification Tree view . . . 360
6.64 Icons of the Palette view . . . 361
6.65 Dependencies related icons in the Palette . . . 381
6.66 Structure of TDE . . . 388
6.67 Tool bar icons of the Test Data view . . . 392
6.68 Status indicators of the Test Data view . . . 392
6.69 Interface elements and icons of the Test Data view . . . 393
6.70 Options of initializing values . . . 404
6.71 Initialization values for data types . . . 404
6.72 Evaluation modes . . . 407
6.73 Value assignments for pointers . . . 414
6.74 Tool bar icons of the Call Trace view . . . 416
6.75 Available types of evaluation macros . . . 425
6.76 Operators of evaluation macros . . . 426
6.77 Evaluation macro specifiers . . . 426
6.78 Structure of the Script Editor . . . 437
6.79 Tool bar icons of the Script Editor perspective . . . 438
6.80 Status indicators of possible script states . . . 439
6.81 Tool bar icons of the Outline view in the Script Editor perspective . . . 439
6.82 Structure of CV . . . 442
6.83 Tool bar icons of the Test Items view . . . 448
6.84 Elements of the Flow Chart view . . . 453
6.85 Sub flow coloring in the Flow Chart view . . . 454
6.86 Structure of the IDA perspective . . . 464
6.87 Status indicators of test objects . . . 465
6.88 Status indicators of the Interface view of a component test . . . 473
6.89 Example: possible evaluation results . . . 481
6.90 Fault injection related tool bar icons in the Flow Chart view of the CV . . . 484
6.91 Indicated fault injection in the Test items view . . . 485
6.92 Possible situations in the include list of the Edit Fault Injection dialog . . . 488
6.93 Calling sequence for running batch tests . . . 496
6.94 Excerpt of the possible commands of the command line interface . . . 498

7.1 Information to log and add to TESSY Support File 506
7.2 The context menu of the Problems Log dialog 508
7.3 The context menu of the Problems view 509
7.4 Startup process problems 512
7.5 Quit process problems 513
7.6 License server problems 514

Index

Advanced Stub Functions, 341, 401
Advanced Stubs, 341, 401
Arithmetic expressions, 399
Assigning changed Interface, 120
Attributes
    Adding, 319
    Editing, 319
    View, 291
Automated tree generation, 367

Backup, 489
    Location, 57
Basic knowledge, 24
Batch
    Operation, **223**
    Test, **223**, 524
Batch Test, 524
Branch Coverage, 524
Branch Coverage View, 95, 458

C Code
    Editing, 423
    Entering, 422
C-source file
    Analyzing the, 83
    Editing the, 115
C/C++, 256
C1 Test, 524
Call Trace view, 415
Check-out License, 20
CLANG Parser, 192
Classes
    Creating, 361
Classification
    Tree, 524
    Tree Editor, 32, 123, 356, 358
    Tree Method, 32
    Creating, 361
Classification Tree, 524
Classification Tree Method, 24
Cloning a project, 60
Code Analysis, 194
Code Coverage, 524
Command Line Interface, 495
Committing Requirements, 271
Compare View, 120, 440, 465
Compiler, 312
    GNU GCC, 81, 312
    GNU GCC (C++), 312
Component Functions, 524
Component Functions View, 473
Component Testing, 524
Composite Dependency, 384
Config File, 53
Configuration
    Enabling, 316
Configuration File, 53, 56, 315
    Assigning a, 322
Console View, 249
Constant Variables, 515
Context Menu, 73
Coverage
    Branch Coverage, 210, 233, 441, 524

Code, 524
Decision Coverage, 210, 233, 441
Entry Point Coverage, 210, 233, 441
Function Coverage, 210, 233, 441
Instrumentation, 173, 210, 233, 441
Measurements, 173, 233, 441
Modified Condition/Decision Coverage, 210, 233, 441, 525
Multiple Condition Coverage, 210, 233, 441
Safety Standards, 173
Settings, 173, 210, 233
Statement Coverage, 210, 233, 441
Viewer, 94, **441**
CTE, 32, 123, 356
Classification Tree Editor, 358
Classification Tree Method, 32
Composite Dependency, 384
Creating test cases, 128
Dependencies, 381
Entering test data, 126
Practical exercise, 123
Tree elements, 124
CTM, 24, 524
CV, 94, 441
Cyclomatic Complexity (CC), 194

Data Format
Setting the, 337
Database
Create, 53
Location, 57
Update, 64
Debugger, 524
Decision Coverage View, 459
Declarations/Definitions View, 417
Delete
Compiler, 310
Environment, 310
Folder, 73
Module, 73
Target, 310
Test Case, 73
Dependencies, 381
Dialog Settings, 173
Differences View, 288, 295
Directories Settings, 173
DO-178B, 173

Edit Source, 115
Editing
Fault Injections, 486
HTML of Requirements, 300
Script Content outside TESSY, 440
Script Files, 436
Scripts of Test Cases/Test Data/User-code, 436
Fault Injections, 483
Editor View, 258
EN 50128, xxi
Entering
Test Data (TDE), 387
Test Data for Signals (THAI), 328
Entry Point Coverage (EPC), 443
Enums, 399, 525
Equally Named Projects, 62
Error handling, 507
Eval Macros, 423
Eval Modes, 405
Evaluation
Macros, 423
Macros View, 249
Modes, 405
Examples, 76
Executing a Test, 207
Exercises, 76
Expected Values, 525
Exporting

Coverage Instrumentation Settings, 173
Test Data, 226
Expressions, 399
External Function Calls View, 147
External Variables, 346

Fault Injection, 456, 483, 525
Fault Injection Candidates, 486
Fault Injection Test Case, 484
Fault Injections
Creating, 486
Report, 488
Fault Injections View, 483
File System, 53
Floating License
Check-out, 17
Manager, 17
Floating license, 12
Floating license server (FLS), 9, 14, 15
Flow Chart, 525
Flow Chart View, 95, **447**, 483
FLS, 9, 14, 15
Format
Data (TIE), 337
Formats
Requirements, 270
Function Coverage (FC), 443

General handling, 52
Generating
Test Cases, 243, 407
Test Driver, 208
Generator Test Case, 243, 407
GNU GCC (C++) Compiler, 312
GNU GCC Compiler, 81, 312

Heartbeat function, 140
Help
Menu, 182
with Problems, 512
Historical test runs, 190
Hysteresis, 525

IDA, 114, 463, 525
IEC 61508, xxi, 173
IEC 62304, xxi
Importing
Coverage Instrumentation Settings, 173
Requirements, 102
Test Data, 226
Inheritance of attributes, 309
Inherited Values, 411
Inline Mode, 397
Input Values, 525
Installation, 1, 3
Instrumentation Settings, 210
Integration Testing, 525
Interface
Changes, 114
Data Assign Editor, 114, 463
Description, 525
Hardware Adapter, 323
Interface Description, 525
Interface dictionary, 180
Internal Data Model, 436
ISO 26262, xxi, 173

License
Check-out, 20
No license server, 17
Node-locked, 9, 10, 12
Transmit, 21
License Key
File, 6
Request, 6
License Server, 9, 14, 15
Link Matrix view, 280
Location
Backup, 57
Database, 57

Project, 57

Maximum Criterion, 39
MC/DC, 525
MC/DC Coverage View, 95, 459
MCC Coverage View, 461
Menu Bar, 67
Merge Dialog, 440
Metrics, 194, 526
Migrating TESSY, 22
Minimum Criterion, 38
Module, 81, 526
Module Testing, 526
Multiple Stub Calls, 401

New test runs, 190
Node-locked license, 9, 10
Notes, 205, 218

Outline View, 260
Output Values, 526
Overview Perspective, 81, 183

Parser Options, 192
PDBX-File, 53
Perspective, 67
C/C++, **256**
CTE, 356
CV, 94, 441
IDA, 114, 463
Overview, 81, 183
Requirement Management, 102, 264
SCE, 469
TDE, 88, 387
TEE, 81, 306
TIE, 85, 329
Perspective Bar, 67
Planning Coverage Report, 108
Plot Definitions View, 351
Plots
View, 434
Pointers, 338, 413
Practical exercise
C++, 156
Component test, 136
CTE, 123
Interior light, 136
Is value in range, 78
Test driven development (TDD), 160
Unit test, 78
Practical exercises, 76
Project
Clone a, 60
Creating a, 55
Database, 53
Directory, 62
Equally named, 62
Location, 57
Root, **53**, 83
Settings, 173
Template, 61
Project Explorer View, 260
Prolog/Epilog View, 418
Properties View, **228**, 330, 390

Quality Metrics, 194

Reference book, 166
Registration, 1, 6
Regression Testing, 114, **463**, 526
Renumbering Test Cases, 247
Repeat Count, 418
Report
Creating a, 98, 215
Directories, 173
Execution Coverage, 215
Fault Injections, 488
Names, 173
Planning Coverage, 108, 215
Test Details, 98, 215
Test Overview, 215

Requirement, 526
Alias, 273
Committing, 271
Coverage, 301
Creating, 265
Editing, 277
Engineering, 265
Formats, 270
Functional, 526
HTML Editing, 300
Import, 268
Importing, 102
Linking of, 301
Management, 102, 263, 526
Non-Functional, 526
Renaming Document, 273
Sources, 270
Versioning of, 294
Requirement Editor View, 275
Requirement Management, 526
Requirements List View, 274
Restore, 489
Reuse, 114, 463
Root Project, 83
RQMT Explorer View, 102, 265, 271
Running a Test, 207

Save Data, 489
Saving Changes, 67
SCE, 469
Scenario Editor, 469
Script Content, 436, 440
Script Editor, 436, 527
Script Editor view, 436, 437
Script File, 436, 439
Script Perspective, 436
Script States, 438
Script Status Indicators, 438
Scripting Language, 436
Search Filter Function, 215
Search Query, 527
Setting Passing Directions, 334
Settings
Coverage, 173
Dialog, 173
Directories, 173
Project, 173
Report, 173
Setup of TESSY, 2
Shortcuts, 73
Solutions
Common problems, 502
Source
Analyzing the, 83
Editing the, 115
Root, 56
Source file
Adding a, 230
Statement Coverage View, 456
Status Bar, 72
Stub
Advanced, 341, 401
Creating, 341
Functions, 208, **339**, 527
Stub Functions, 527
Stub Functions View, 427
Sub Flows, 452, 453
Support, 502, 503
Support File, 503
Suspicious Elements, 284, 285
System Testing, 527

Target, 312
Target Passing, 338
Tasks, 188
TDE, 88, **387**, 527
Technical requirements of TESSY, 2
TEE, 81, **306**

Template Project, 61
TESSY
    Interface Hardware Adapter, 323
    Support File, 503
Test
    Definition, 527
    Definition View, 415
    Driver, 527
    Environment, 527
    Environment Editor, 81, **306**
    Execution, 207
    Interface Editor, 85, **329**, 528
    Items View, 86, 238
    Means view, 279
    Project View, 81, 184
    Regression, 526
    Results View, 248
    Run, 207, 528
    Suite, 528
    System, 527
Test Case, 527
    Generator, 243, 407
    Renumbering, 247
Test Cases
    Creating, 361
Test Data
    Editor, 88, **387**, 527
    Exporting, 226
    Importing, 226
    View, 391
Test Data Editor, 527
Test Definition, 527
Test Driver, 527
Test Environment, 527
Test Object, 527
    Changing Interface, 115
    External Functions, 333
    External Variables, 333
    Global Variables, 333
    Local Functions, 333
Test Run, 528
Test Suite, 528
Testing effort estimation, 196
THAI, 323, 527
    Configuration file, 324
    TESSY Interface Hardware Adapter, 323
    Functionality, 325
Theory, 24
TIE, 85, 329, 528
Tool Bar, 67
Tracking, 196
Troubleshooting, 502
Tutorial, 52, 76
Type Information, 330, 390
Type Modifier, 330, 390

Uninstallation, 16
Union, 397
Unit, 528
Unit testing, 24
Unused Functions, 349
Unused Variables, 349
Updating a database, 64
Usercode, **418**, 432, 528
    TS_CALL_COUNT, 429
    TS_CURRENT_TESTCASE, 419, 429
    TS_CURRENT_TESTSTEP, 419, 429
    TS_REPEAT_COUNT, 419
Usercode Outline View, 432

Variant Management, 199
    Inherited Values, 411
Vector Values, 401
Version Control, 271, 276, 489
View, 68
    Attached Files, 290
    Attributes, 291
    Branch (C1) Coverage, 95
    Call Trace, 415

Classification Tree, 358
Compare, 120, 440, 465
Component Functions, 473
Console, 249
Declaration/Definition, 417
Differences, 288, 295
Document Preview, 298
Editor, 258
Evaluation Macros, 249
External Function Calls, 147
Flow Chart, 95
History, 294
Interface, 85
Link Matrix, 280
MC/DC Coverage, 95, 459
Outline, 260
Plot Definitions, 351
Plots, 434
Project Explorer, 260
Prolog/Epilog, 418
Properties, 81, 330, 390
Related Elements, 296
Requirements Coverage, 301
Requirements Editor, 275
Requirements List, 274
Reset Position, 69
RQMT Explorer, 102, 265, 271
Script Editor, 436, 437
Stub Functions, 427
Suspicious Elements, 285
Test Data, 88, 391
Test Definition, 415
Test Items, 86, 238
Test Means, 279
Test Project, 81, 184
Test Results, 248
Usercode Outline, 432
VxV Matrix, 278
Work Task Configuration, 473
Branch (C1) Coverage, **458**
Decision Coverage, **459**
Fault Injections, 483
Flow Chart, 483
MCC Coverage, **461**
Properties, **228**
Statement (C0) Coverage, **456**
Work Task Configuration View, 473
Work Tasks, 473
Work tasks, 148
Working with TESSY, 166
Workspace, 53, 528

Razorcat Development GmbH

Witzlebenplatz 4
Germany, 14057 Berlin
tel: +49 (030) 53 63 57 0
fax: +49 (030) 53 63 57 60
e-mail: support@razorcat.com
internet: http://www.razorcat.com