VECTOR



VectorCAST/Ada

User's Guide

VectorCAST 2026

New editions of this guide incorporate all material added or changed since the previous edition. Update packages may be used between editions. The manual printing date changes when a new edition is printed. The contents and format of this manual are subject to change without notice.

Generated: 10/19/2026, 8:02 PM

Rev: 6be4125

Part Number: VectorCAST/Ada User's Guide for VectorCAST 2026 SP5

U.S. Government Restricted Rights

This computer software and related documentation are provided with Restricted Rights. Use, duplication or disclosure by the Government is subject to restrictions as set forth in the governing Rights in Technical Data and Computer Software clause of

DFARS 252.227-7015 (June 1995) and DFARS 227.7202-3(b).

Manufacturer is Vector North America, Inc. East Greenwich RI 02818, USA.

Imprint

Vector reserves the right to modify any information and/or data in this user documentation without notice. Vector disclaims all liabilities for the completeness or correctness of the contents and for damages which may result from the use of this documentation. This documentation nor any of its parts may be reproduced in any form or by any means without the prior written consent of Vector. To the maximum extent permitted under law, all technical data, texts, graphics, images and their design are protected by copyright law, various international treaties and other applicable laws. Any unauthorized use may violate copyright and other applicable laws or regulations.

© Copyright 2026, Vector Informatik, GmbH. All rights reserved.

Protected Trademarks

All brand names in this documentation are either registered or non-registered trademarks of their respective owners.

Third-Party copyright notices are contained in the folder: 3rdPartyLicenses, located in the VectorCAST installation directory.

TABLE OF CONTENTS

Introduction **14**

VectorCAST Overview **15**
VectorCAST Automation 15
Key Terminology 15
Key Concepts 17
VectorCAST Components 19

Learning Your Way Around **21**

Starting VectorCAST **22**
To Start VectorCAST and Open an Environment 22
To Start VectorCAST, Build, and Open an Environment 22
To Start VectorCAST and Open an Options Dialog 22
To Start VectorCAST in C++ or Ada "Mode" 24
To Set the Product Mode 25
To Exit VectorCAST 25

VectorCAST Interface **25**
To Specify the Language for the VectorCAST GUI 27
Multibyte Characters 27
To Set the Industry Mode for Coverage 27
Changing Industry Mode 30
To Collapse / Expand the Message Window 31
To Hide the Message Window 31
To Clear or Copy Text in Message Window 32
To Hide the Environment View 32
To Move Docking Windows 33
To Float Docking Windows 34
To Return Docking Windows to Default Locations 35
To Save the Window Configuration 35
The Toolbar 36
Customize Icons 38
The Status Bar 40
To Open a File Named in a Report 40
The MDI Window: Groups 40
To Open a Group 41
To Close a Group 41
To Maximize a Window 42
To Close a Tab 42
To Arrange Groups in a Cascade 42
To Tile Groups 43
To Bring a Tab to the Top 45
To Close the Current Window (Tab) 45
To Close All Windows 45
VectorCAST Keyboard Shortcuts 45

Getting Help 48
To Determine Available Licenses 48
To View Online User Guides 49
To Contact Technical Support 50
To Send a Test Environment to Technical Support 50
To Determine the VectorCAST Version and Customer Number 50
Using the Jobs Window and Jobs Monitor 52
The Jobs Window 52
Opening the Job Monitor 53
Job Status Viewer 54
Execution Status Viewer 57
Changing Application Preferences 60
To Set the Display Theme 60
To Set Up an External Text Editor 61
To Edit User Code with the External Editor 64
To Set Up an External Browser 64
To Toggle Gridlines in the Test Case Editor 65
To Alphabetize Test Cases 65
To Change the Main Window's Font 67
To Automatically Save Test Cases Before Execution 67
To Turn on a Reminder Before Closing Multiple Tabs 67
Customizing Reports 68
Customizing Existing Reports 68
Customizing Templates 70
Customizing Report Sections 71
Customizing Report Style 73
Creating New Reports 76
Editing, Searching, and Printing 81
To Create a New Text File 81
To Open a Script or Report File 81
To Save a Script or Text File 82
To Save the Open Window As 82
To Save All Open Windows 82
Print Setup 82
To Print an Open Window 83
To Preview Before Printing 84
To Undo a Recent Change 84
To Redo 84
To Copy, Cut, and Paste 84
To Search for Text Using the Find Banner 85
To Apply a Search Filter to the Test Case Tree 88
To Apply a Search Filter to the Parameter Tree 93
To Repeat a Search 94
To Goto a Line 94
Browsing Configuration Options 96
Using the Configuration Options Viewer 96

Creating a New Environment 98

Using the Wizard to Create an Environment 99
To Set the Working Directory 99
Troubleshooting the Working Directory 100
To Start the Wizard 101
To Save the Settings in the Wizard 102
Step 1: Choose Compiler 102
Step 2: Name the Environment 103
Step 3: Build Options 104
Whitebox Restrictions and Limitations 106
To Not Allow Recursive Calls in the Harness 106
Step 4: Locate Source Files 107
Step 5: Choose UUTs & Stubs 108
To Specify a Stubbing Strategy for Individual Units 110
To Specify Custom Stubbing Interactively 112
Step 6: User Code (Optional) 113
Step 7: Summary 114
To Build the Environment 114
To View the Environment Overview Report 115
Files Created by the Environment 117
Troubleshooting Environment Creation 117
To Create a System Testing Environment 119

Setting Compiler Options 120
Options: Ada Tab 120
Options: Wizard Tab 123

Setting Builder Options 127
Options: Builder Tab 127
Deprecated Builder Options 135

Working with a Test Environment 135
To Open an Environment 135
To Close an Environment 136
To Rename an Environment 136
To Update an Environment 136
To Create Regression Scripts for an Environment 138
To Post-Process the Regression Scripts 142
To Integrate Regression Scripts with ClearCase™ 144
To Save and Load a Post-Processing Script 145
To Incorporate Source Code Changes into Environment 145
To Rebuild the Environment 146
Troubleshooting Environment Rebuild 147
To Delete an Environment 147
To Re-create a Deleted Environment 147

Other Environment Tools 148
To Create an Environment Script 148
To Create an Environment by Running a Script 148
To Recompile the Test Harness 149

To Recompile the Instrumented Test Harness without Re-instrumenting It 149
To Create a Test Harness Recompile Script 149
Recompile the Test Harness by Running a Script 152
To Relink an Environment 153
To Refresh Type Range Data 153

Building Test Cases 154

Using The Test Case Tree 155
The Test Case Tree Hierarchy 155
To Navigate the Test Case Tree 156
To Multi-Select Test Cases 156
Types of VectorCAST Test Cases 156
To Open a Test Case for Editing 157
Test Case Naming 157
To Rename a Test Case 158
To Sort Test Cases Alphabetically 158
To Duplicate a Test Case 158
To Delete a Test Case 158
To Delete All Test Cases from a Subprogram 159
To Delete All Test Cases from a Unit 159
To Delete All Test Cases in the Environment 159
To Restore Deleted Test Cases 160
To View the Source for a Subprogram 161
To View Coverage for a Unit or Subprogram 161

Simple Test Cases 161
To Insert a Test Case for a Subprogram 161
To Insert an <<INIT>> Test Case 162
To Create a Min, Mid, or Max Test Case 162
To Insert Basis Path Test Cases 164
Troubleshooting Min, Mid, Max Test Cases 166

Compound Test Cases 167
Building a Compound Test Case 167
To Specify the Number of Iterations of a Test in a Compound 168
To Specify a Delay Between Tests in a Compound 170
Compound as Automatic Initialization / Finalization 170
To Reorder Test Cases in a Compound 171
To Open a Test Case from a Compound 171
To Delete a Slot from a Compound 172
To View Coverage for a Test Case from a Compound 172
To Locate a Slot in the Test Case Tree 172
To Search for Text in the Compound Test Editor 172
Compound Only Test Cases 173
Nested Compound Test Cases 173
To Enable / Disable Reporting for Compound Slots 176
To Set Expected Values to Actual Values 178

Entering Test Case Data 178
The Parameter Tree 179

To Navigate the Parameter Tree ... 180
To Alphabetize Parameters in the Parameter Tree ... 180
To Enter Input and Expected Values ... 182
To Enter Test Case Requirements or Notes ... 182
To Enter Values for Global Data ... 182
To Enter a Range ... 182
To Enter a List ... 183
The List Values Tab ... 183
To Pass a Null String ... 184
To Raise an Exception from a Stub ... 184
Working with In/Out Parameters in Stubs ... 185

Data Types ... 185
To Enter Values for Enumeration Types ... 185
To Enter a Number for an Enumeration ... 186
To Enter Integer Values in Different Bases ... 186
To Enter Real Numbers using Scientific Notation ... 186
To Enter Values for Record Types ... 187
Variant Record Types ... 187
Character Types ... 187
To Enter a String ... 187
System.Address Types ... 188
Access Types ... 188
Special Kinds of Stubs ... 189
Testing Out of Range Values ... 191
VectorCAST and "Class" based Testing ... 192
Using the Multi-unit Whitebox feature to Manipulate Private Types ... 197

Working with Arrays ... 198
To Enter Values for Constrained Array Types ... 198
To Expand All Elements of an Array ... 198
To Expand the First Element of an Array ... 199
To Collapse Unused Elements of an Array ... 200
To Expand Certain Elements of an Array ... 200
To Apply Values to an Array ... 203
To Clear Values from an Array ... 205
Using the Array Properties Dialog ... 205
Multi-Dimensional Arrays ... 206
Unconstrained Arrays ... 206
Working with Unconstrained Array Parameters ... 207

Entering Data in the Parameter Dialog Box ... 208
To Use a Range Expression in an Expected Value List ... 208
To Use the <<Any>> Tag in a List ... 209
To Access the Parameter Dialog ... 209
The Scalar Values Tab ... 210
To Apply Values to an Array ... 211
To Clear Values from an Array ... 213
The Range Values Tab ... 213
To Set Up an Input Range or List for More Than One Parameter ... 214
To Use Ranges as Expected Values ... 214

The List Values Tab ... 215
Controlling Values in the List ... 215
To Repeat Values in the List ... 216
To Use a Range Expression in an Expected Value List ... 217
To Use the <<Any>> Tag in a List ... 218

Test Case Scripting ... 219
To Export Test Cases to a Test Script ... 219
To Clear Unapplied Test Script Data ... 220
To Import Test Cases from a Test Script ... 220
To Create a Test Case Template ... 221
Troubleshooting Test Script Template ... 222
Automating Test Script Maintenance ... 223

Generating Test Cases Using CSV- or Tab-Delimited Files ... 228
To Create a Template for a CSV- or Tab-Delimited File ... 228
To Create a Map from an Existing CSV or Tab-Delimited File ... 234
To Create Test Cases from a MAP Test Case ... 238
To Edit an Existing Map Test Case ... 239
To Create and Edit a Test Script from a Map File ... 240

VectorCAST Tools ... 242
To Set VectorCAST Options ... 242
Tools Menu ... 243
View Harness Source ... 243
Basis Path ... 244
MC/DC ... 245
Working With Custom Tools ... 245
To Create a Diagnostic Report ... 252
Troubleshooting VectorCAST ... 252

Executing Tests ... 254

Executing Test Cases ... 255
To Execute a Test Case and View Results ... 255
Deprecated Report Options and Commands ... 261
Compound Test Case Execution and Reports ... 261
To Execute Multiple Test Cases ... 263
To Abort Test Case Execution ... 264
To Execute a Test Case in the Debugger ... 264
Working with a Control Flow ... 264
To Save the Control Flow ... 267
To Clear the Control Flow ... 267
To Set Expected Values to Actual Values ... 268

Viewing Test Reports ... 268
View Reports in an External Browser ... 269
View a Test Case Data Report ... 269
To View Test Execution Results ... 270
To Create a CSV Execution Report ... 271
The Full Report ... 272

The Test Case Management Report ... 273
View Test Data Summary ... 274
View the Test Comparison Report ... 278
View a Test Case's Raw Data Set ... 285

Setting Execution Options ... 285

Setting Report Options ... 294
Report Content Options ... 294
Report Format Options ... 306
The Testcase Options Tab ... 312
Deprecated Report Options and Commands ... 318

Incremental Rebuild ... 320

About Change-Based Testing ... 321

Performing an Incremental Rebuild ... 321

Using Code Coverage ... 324

Code Coverage ... 325
To Initialize Statement Coverage ... 326
To Initialize Branch Coverage ... 327
To Initialize MC/DC Coverage ... 328
Suppressing MC/DC Initialization on Compile Error ... 329
To Initialize Statement+MC/DC ... 330
To Initialize Statement+Branch ... 330
To Initialize Coverage for Units Other than UUT ... 331
To Avoid Instrumenting Sections of Source Code ... 332
To Uninstrument an Environment ... 332
To Enable Coverage ... 333
To Disable Coverage ... 333

Viewing Coverage Results ... 333
To Turn on Coverage Results ... 333
The Coverage Viewer ... 334
To Customize the Coverage Viewer ... 340
Map Line Selection to Original Source View ... 340
To Open a Test Case That Covers a Line ... 341
To Remove All Coverage Results ... 343
To View the Aggregate Coverage Report ... 343
To View the Metrics Report ... 344
To View Code Coverage Summary ... 347
Understanding Basis Paths ... 350
To Build Test Cases from Basis Path Analysis ... 351
To View a Basis Paths Report ... 351
MC/DC Coverage ... 352
Understanding MC/DC Analysis ... 353
To View the Equivalence Matrices Report ... 354

Viewing Execution Flow with Animated Coverage ... 356

To Activate Coverage Animation ... 357
To Play the Coverage Animation ... 358
The Animation Toolbar ... 358
To Set a Breakpoint ... 359

Setting Coverage Options ... 360
Options: Coverage Tab ... 360
Coverage Viewer Options ... 368
To Format the Text in the Coverage Viewer ... 369

Importing Coverage Results ... 371
Preparing to Import Results ... 371
Importing the Results ... 372
To Export a Script of Coverage ... 373
To Import a Coverage Script ... 376
To Delete Imported Results ... 376
To View the Coverage Import Log ... 377

Using VectorCAST Covered By Analysis ... 378

Covered By Analysis (CBA) ... 379

To Add Coverage Analysis ... 379
Using the Coverage Analysis Editor ... 379
To Edit an Existing Analysis Result ... 382
Viewing CBA Coverage in the Coverage Viewer ... 383
To Change Covered-By-Analysis Display Color ... 383

Working With Analysis Files ... 384
To Import Analysis Files ... 384
To Export Analysis Files ... 385
To Remove Analysis Files ... 385

Re-Instrumenting With CBA Data ... 385

Viewing Analysis Data in Reports ... 386
Covered By Analysis Report ... 386
Aggregate Coverage Report ... 388
Metrics Report ... 389

User Code ... 390

Understanding User Code ... 391
Types of User Code ... 391
Order of Execution ... 391
Editing User Code ... 392
User Code Tags ... 392
Features Common to All User Code ... 393

Environment User Code ... 396
Types of Environment User Code ... 396
To Edit Environment User Code ... 398
To Test Compile Environment User Code ... 399

User Globals ... 402
To Edit User Globals ... 403
To Save User Globals ... 404
To Recompile Environment User Code ... 405
To Remove Environment User Code ... 405
Example: To Perform Timing Analysis ... 405
Example: To Read Test Data from a Database ... 406
User Parameters ... 407
Example: Chaining Data Values With User Parameters ... 409

Test Case User Code ... 410
To Enter Test Case User Code ... 410
Test Compile Testcase User Code ... 411
To Clear Test Case User Code from Test Harness ... 412
Add Test Case User Code Back Test Harness ... 412

Parameter User Code ... 412
To Enter Parameter User Code ... 413
To Test Compile Parameter User Code ... 413
Input User Code Example ... 413
Expected User Code Example ... 414
Another User Code Example ... 414

Stub User Code ... 417
To Enter Configure Stub User Code ... 418
To Test Compile Stub User Code ... 419
To Save Stub User Code ... 421
To Recompile Stub User Code ... 422
To Remove Stub User Code ... 422
Example of Stub User Code ... 423

Using the Requirements Gateway ... 425

Requirements Gateway (RGW) ... 426

Using Requirements Gateway ... 426
Create a Requirements Repository ... 427
The Authentication Tab ... 427
The Import Tab ... 428
The Export Tab ... 430
The Requirements Tab ... 431
The Script Tab ... 432
Link Requirements to Test Cases ... 433
Tracking Requirement Change Impacts on Test Cases ... 434

Working With Supported Gateways ... 437

Example Work Flow for RGW Command Line ... 438
Step 1 - Set the Repository ... 438
Step 2 - Initialize the Gateway ... 438
Step 3 - Set Configuration Options ... 439
Step 4 - Import Requirements ... 440

Step 5 - Link Requirements to a Test Case 441
Step 6 - Run the Test Case 441
Step 7 - Configure the Export Settings 442
Step 8 - Export the Test Data 443

Appendix A: VectorCAST/Ada Messages 444

Start-up Messages 444
License Messages 444
Environment Building Informational Messages 444
Environment Building Error Messages 445
Test Case Messages 448
General Messages 449
CLICAST Messages 452

Appendix B: Environment Script Language 454

Enviro Command List 454

Appendix C: Test Script Language 465

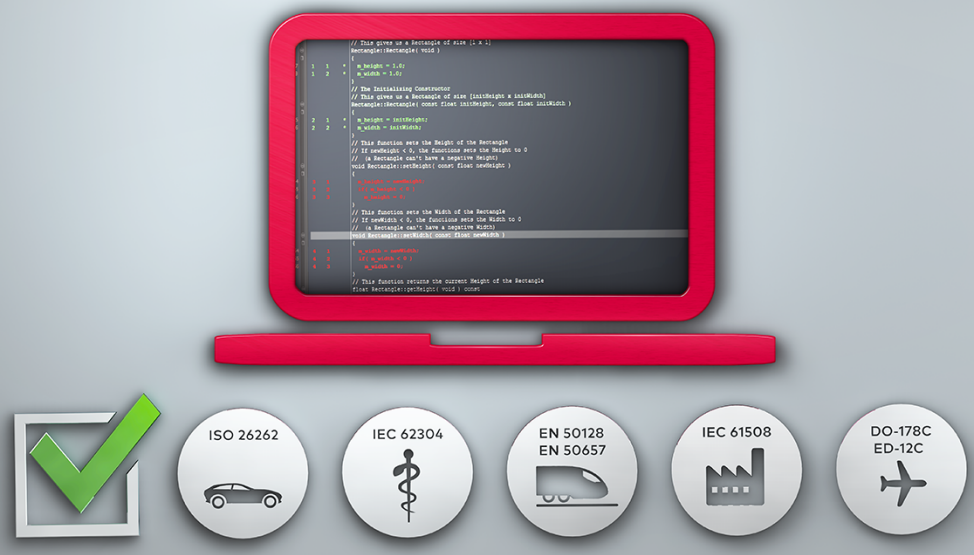
TEST.SCRIPT Command List 465
Script Commands 465

Setting Input and Expected Values 471
Numeric Types 472
Structure Types 472
Enumeration Types 472
Boolean Types 472
Character and String Types 472
Access Types 473
Array Types 473
Unconstrained Array Parameters 473
Record Types 474
Subprogram Return Parameters 474
Global Objects 474
Test Case Options 474
Range of Values for Input Data 475
Range of Values for Expected Results 476
List of Values 477
Test Values Spanning Multiple Lines 477
Working with Overloaded Subprograms 477
Adding User Code to a Test Script 478

Appendix D: Limitations and Restrictions 479

Compiler Limitations 479

Appendix E: Environment Variables 488
Environment Variables 488
Appendix F: CLICAST - Command Line VectorCAST 493
Quick Start 493
Command Format 494
The Options Append Command 495
Console Report Format 495
Appendix G: CLICAST - Tool Options Reference 499
Tool Options 499
Appendix H: RGW - Command Line Reference 547
RGW Commands 547
Configure Set 547
Export 548
Import 548
Initialize 548
Requirement Report 548
Set Gateway 549
Testcase Link 549
Testcase Mark as Reviewed 549
Testcase Migrate_links 549
Testcase Report 550
Index 551



Introduction

VectorCAST Overview

VectorCAST is a suite of tools for automating the entire process associated with conducting unit and integration testing:

- VectorCAST/C++ and VectorCAST/Ada automatically generate executable test harnesses for one or more units of application source code written in C, C++ or Ada. In addition, they generate and execute unit test cases and report on results and metrics.
- VectorCAST/QA is a code-coverage analysis tool that identifies which areas of an application have been exercised by test executions. It supports C, C++ or Ada, and supplements the unit coverage provided by VectorCAST/C++ and VectorCAST/Ada.

VectorCAST Automation

You can use VectorCAST to conduct unit or integration testing on applications written in the C, C++, or Ada programming languages. VectorCAST automates the following unit and integration test functions:

- Generating and compiling test stubs and driver programs
- Generating test cases based on boundary values
- Constructing user-defined test cases by interactive point-and-click, or by script
- Executing modified test cases without re-compiling the test harness (Exception: when running in stdout only mode, the test harness must be recompiled.)
- Conducting regression testing
- Generating standards-compliant test reports
- Conducting comprehensive branch and statement coverage analysis
- Conducting Modified Condition/Decision Coverage (MC/DC) analysis
- Conducting basis-path analysis and determining cyclomatic complexity
- Executing tests on host and embedded-target development systems

Three types of input data can affect the processing of a unit under test (UUT):

- The input parameters to the UUT
- The output parameters of dependent-unit subprograms (that is, the output of subprograms invoked by the unit)
- Global data objects

VectorCAST provides direct control over all three types of data. It does this by automatically creating a unique test harness for each software unit to be tested.

Key Terminology

You should be familiar with the following terms before you begin using the tutorials:

Environment – The term “Environment”, as it is used in this document, refers to a repository of information saved in a VectorCAST-created disk directory. This directory is used to store test harness source files, test-specific data files, etc.

Test Harness – A test harness is the executable program created when the test driver, the units

under test, and the dependent units or stubs of dependent units are linked together.

The test harness has three components:

- **Units Under Test (UUTs)**
  A Unit Under Test is a compilation unit to be tested. In C or C++, a UUT is a single source file. In Ada, it can be a standalone subprogram specification, subprogram body, a package specification, or package body. You can have more than one unit under test in a test environment (for integration testing).
- **Test Driver**
  The driver program is capable of invoking all subprograms of the unit under test with values provided from the test case data and capturing any returned data or exceptions that may be raised during test execution. The components of the test harness are automatically compiled and linked into an executable test harness, which is used by VectorCAST to effect all testing.
- **SmartStubs**
  When testing a software unit, it is often desirable to create alternate subprogram definitions for some or all dependent units. These “placeholders” used for testing purposes are known as *stubs*. Stubbed subprograms allow you to begin testing long before the actual system is built. They also allow you to exercise more exact control over the values passed back to the Unit Under Test than would be available with the actual dependent unit in place. VectorCAST automatically generates the code for those dependent units chosen to be stubbed.

Visible Subprogram – In Ada, a visible subprogram is a subprogram specification contained in the visible part of a compilation unit.

Dependent Unit – A file that contains a function called by a UUT, or data objects used by a UUT. A dependent unit is an Ada package that is 'withed' by a unit. Dependent units can be stubbed or used 'as is'.

Event – An event is a change in the flow of control that VectorCAST can monitor. Each of the following is an event:

- A call from the test harness to a UUT
- A call from a UUT to a stubbed dependent in the test harness
- A return from a UUT to the test harness

Test Case – A collection of input and output data, defined with the purpose of exercising specific software and verifying results. This data is commonly comprised of the formal input and output parameters of the dependent subprograms, and input and output parameters of the subprogram under test. The Test Data controls the execution of the unit under test as well as values for global objects visible in the unit under test or stubbed subprograms.

Test Driver – VectorCAST generates the main program, or *driver*, necessary to exercise all visible subprograms in all units under test. This consists of calls to each visible subprogram in a unit under test.

Parent Library (Ada) – The directory that contains the units under test and dependent units. In Ada environments, the Parent Library is a single directory in the hierarchy of libraries.

Stub by Function Units – Prior to VectorCAST version 5.0, only subprograms in dependent units of the UUT were allowed to be stubbed. In fact, all subprograms in stubbed units had to be stubbed. In

VectorCAST 5.0, you have the choice of making a Unit Under Test "stub by function (SBF)." When a UUT is made "SBF", this means that functions in the unit can be stubbed individually, while others in the unit are not stubbed.

Key Concepts

You should be familiar with the following concepts before you begin using the tutorials:

Execution Closure – An execution closure consists of all the units required for an application to successfully link. Figure 1 shows a closure consisting of a single UUT, three direct dependent units, and several sub-dependents.

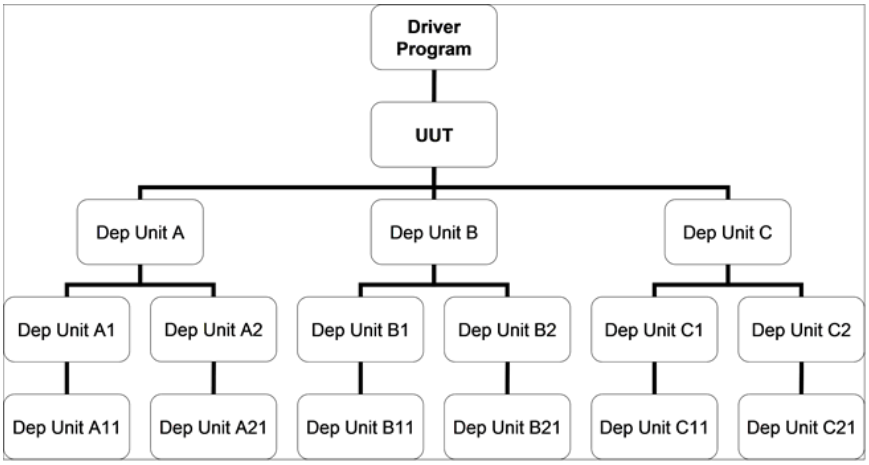


Figure 1. An example execution closure.

Stubbing Units in a Closure – When building a VectorCAST environment, you designate stubbing from the top-most level of dependency downward to the bottom-most level in each branch.

In the closure illustrated in Figure 1, for example, dependents Dep Unit A, Dep Unit B, and Dep Unit C would be the primary candidates for stubbing. If you were to leave one of these dependents unstubbed, the dependents immediately under it would then become the primary candidates for stubbing, and so on to the bottom of each branch.

Figure 2 shows the example closure after stubbing has been designated.

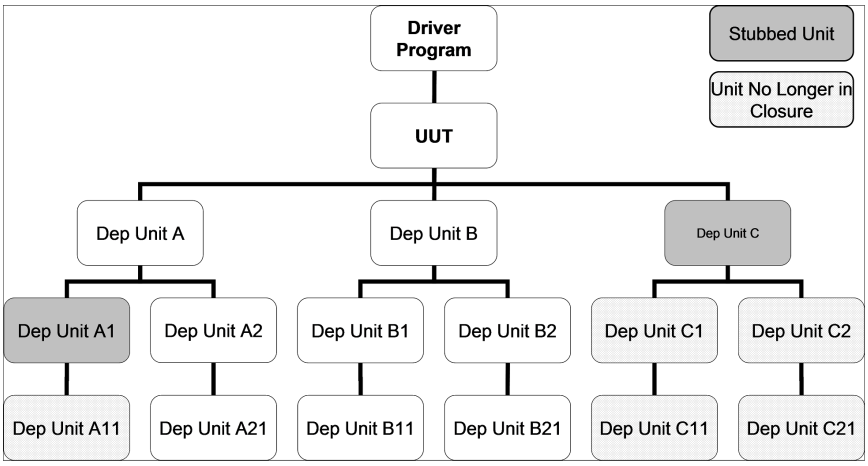


Figure 2. A typical VectorCAST closure.

The scheme allows a flexibility spanning from testing a single unit in a standalone environment in which every dependent is stubbed, to testing groups of units with only the dependents on the periphery (bottom) of the closure stubbed.

Ignoring Units in a Closure (Ada only) – As mentioned above, normally when you choose not to stub a unit, any dependents below this unit remain in the closure. In VectorCAST/Ada, however, you can choose to leave a unit unstubbed but effectively remove all the downstream dependents from the list of stub candidates; all downstream dependents will not be stubbed. You do this by directing VectorCAST/Ada to ignore the unit.

Note: In Ada, ignoring dependents applies only to the dependents of the 'body' of the ignored unit; it does not apply to the dependents of the 'spec'. The reason for this is that VectorCAST needs to parse the dependents of the 'spec' in order to resolve types derived from the dependents.

This option is useful when you want to avoid doing any stubbing beyond a certain point. For example, in Figure 3, suppose Unit C is an interface to a subsystem already tested. There is no need to stub beyond Unit C. By directing VectorCAST to ignore Unit C, it will use this entire branch as if you had chosen not to stub the units.

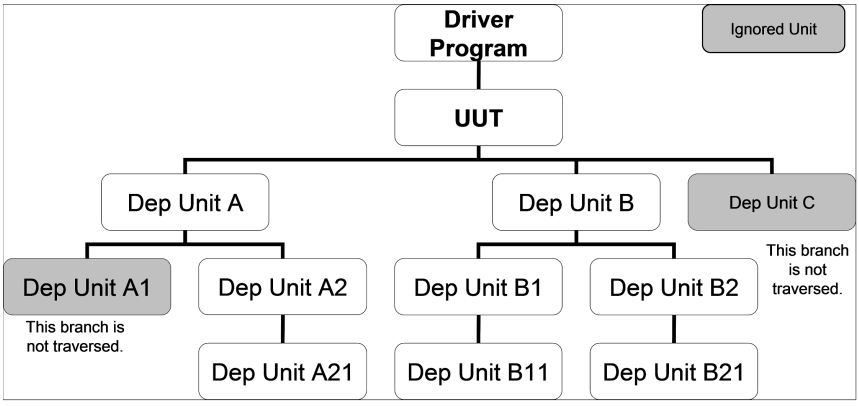


Figure 3. Closure with ignored units (Ada only)

This option can in some cases dramatically reduce the time it takes for VectorCAST to build an environment.



Note: In choosing to ignore a unit, downstream dependents will not be ignored if these units are brought into the closure through an alternate dependency.

VectorCAST Components

VectorCAST consists of four major functional components:

- Test Environment Builder
- Test Case Generator
- Execution Manager
- Report Generator

Test Environment Builder

The Test Environment Builder is central to VectorCAST and serves two related functions:

- It generates a framework (environment) consisting of all the necessary resources to build a test harness
- It generates (or regenerates) the test harness itself

To initiate a test environment, the user:

- Supplies the location of the source files to undergo test
- Designates the individual UUT(s) to be exercised
- Designates any dependent units to be stubbed

The Environment Builder parses the designated files and, from the information gathered, generates a complete test harness.

Unlike a manually generated test harness, a VectorCAST test harness can be automatically regenerated whenever changes are made to a UUT or stub included in the environment (for example, when a subprogram is fully coded).

In addition, test cases can be added to an existing environment without any need for manual coding, recompiling, etc.

Test Case Generator

The Test Case Generator uses static analysis and/or user input to build test cases for exercising the UUTs included in a test environment. These test cases (and their results) are stored within the test environment itself and are thereby available for use in future testing. This is an important feature for regression testing.

Execution Manager

The Execution Manager invokes the test harness created by the Environment Builder, and executes

the test cases built by the Test Case Generator.

Report Generator

The Report Generator produces a variety of reports based on information maintained in every environment database. This information includes:

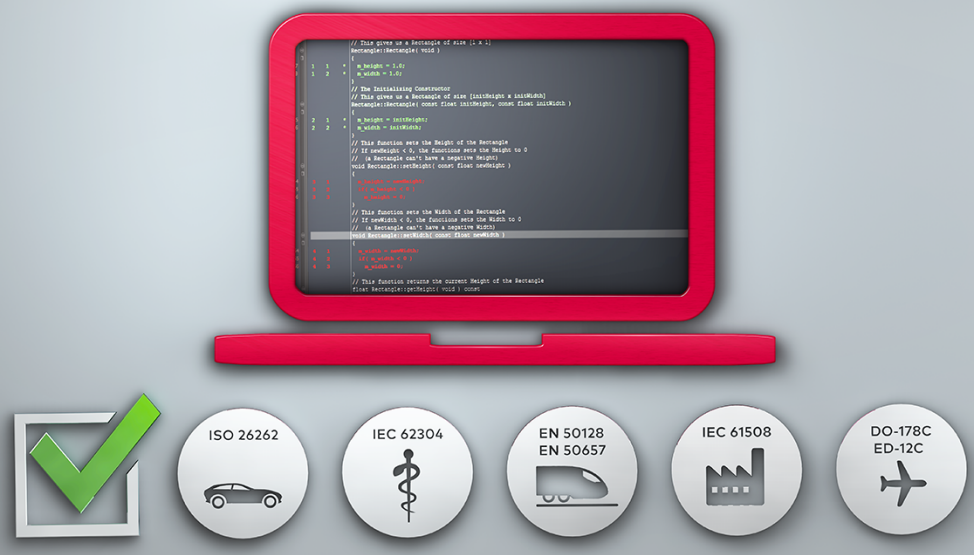
Test configurations – Unit names, test names, dates and times.

Test case data – Listings of inputs and expected results.

Execution histories – Listings of execution results; comparisons of expected and actual results; comparisons of control flow; pass/fail indications.

Coverage – Source listings annotated and color-coded to indicate code coverage.

Results – Tables summarizing pass/fail status and coverage results.



Learning Your Way Around

Starting VectorCAST

Prior to starting VectorCAST, you should ensure that VectorCAST is installed. See the *VectorCAST Installation Guide* for details. Also, the *VectorCAST Quick Start Guide* has additional information for starting VectorCAST.

To Start VectorCAST and Open an Environment

To start VectorCAST and open an environment, first change directories to the directory where the environment exists, then use the **-e** command line option, as in the following examples.

Windows Command Prompt

```
%VECTORCAST_DIR%\vcastqt -e <environment_name>
```

Linux

```
$VECTORCAST_DIR/vcastqt -e <environment_name>
```

To Start VectorCAST, Build, and Open an Environment

To start VectorCAST, build an environment and open it, use the **-b** command line option, as in the following examples.

Windows Command Prompt

```
%VECTORCAST_DIR%\vcastqt -b <env script>
```

Linux

```
$VECTORCAST_DIR/vcastqt -b <env script>
```

The path to **<env script>** can be relative to the directory from which VectorCAST is invoked, or a full path. The filename can be specified with or without the **.env** extension.

To Start VectorCAST and Open an Options Dialog

To start VectorCAST and open an Options dialog from the command line use the **-o** command line option, as in the following examples.

Windows Command Prompt

```
%VECTORCAST_DIR%\vcastqt -lada -o <path to working directory>
```

Linux

```
$VECTORCAST_DIR/vcastqt -lada -o <path to working directory>
```

Users can specify a path to a working directory immediately after the **-o** command line option. The Options Dialog uses this path to set the current working directory from which configuration options are loaded.

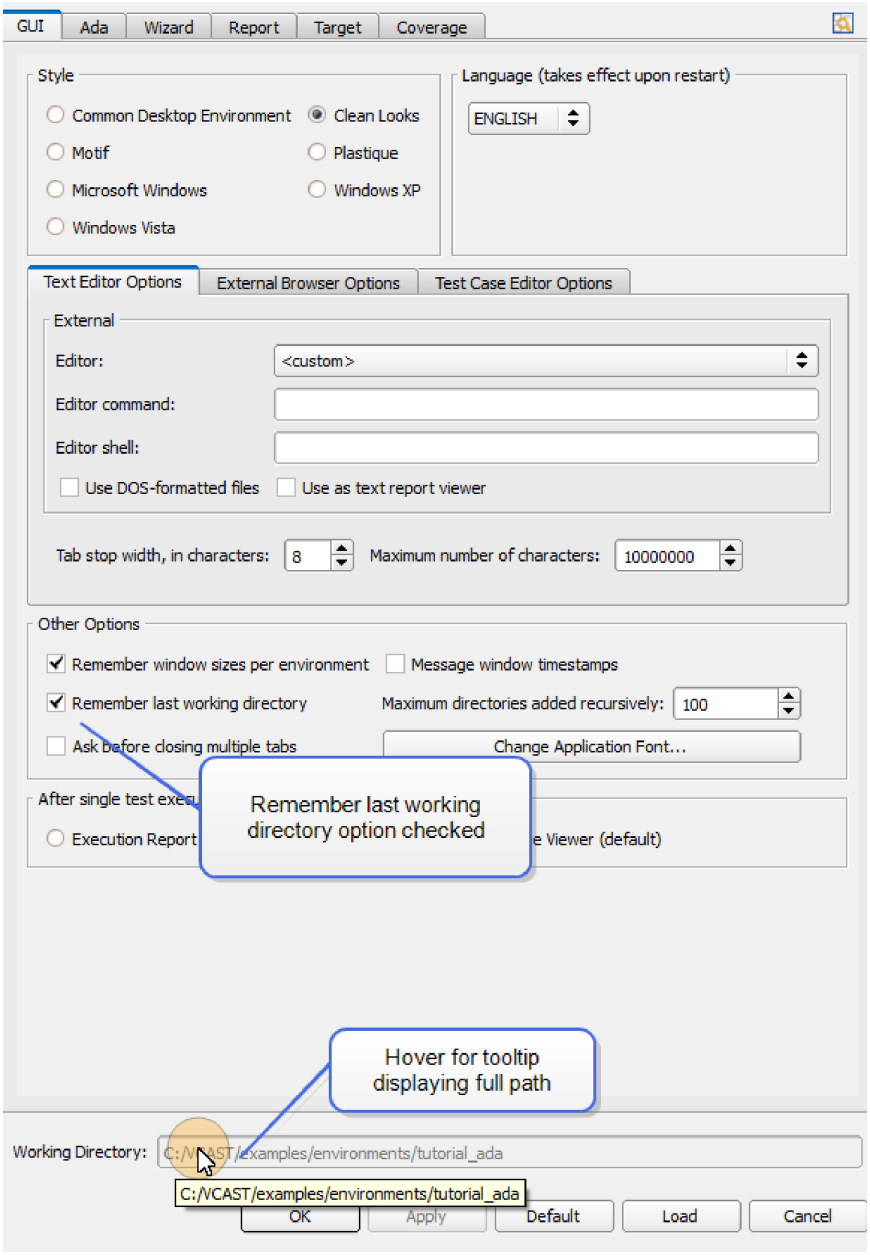
When the **-o** option is used without specifying a path, the behavior of the Options Dialog depends on two factors:

- If the option to remember the last working directory is not checked, or if the last working directory is not set, the Options Dialog defaults to the current directory.
- If the option to remember the last working directory is checked and the last working directory is valid, the Options Dialog defaults to using that directory.

The **-o** option accepts the following path formats:

- Absolute paths
- Relative paths
- Paths with tilde (~) expansion
- Paths containing spaces
- Paths derived from environment variables set to a directory

Note that invalid paths are ignored. In such cases, the Options Dialog reverts to using the current directory.



To Start VectorCAST in C++ or Ada "Mode"

If you have both VectorCAST/C++ and VectorCAST/Ada licensed, use the following command to start VectorCAST in “C/C++ mode” or “Ada mode.” Thereafter, when you click the New button, the wizard for the specified language opens, instead of giving you a choice.

Windows Command Prompt

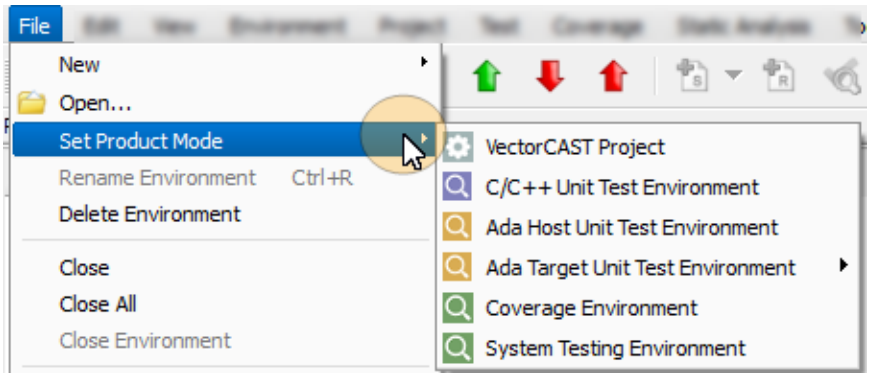
```
C:\> %VECTORCAST_DIR%\vcastqt -lc (to open VectorCAST/C++)
C:\> %VECTORCAST_DIR%\vcastqt -lada (to open VectorCAST/Ada)
```

Linux

```
$> $VECTORCAST_DIR/vcastqt -lc (to open VectorCAST/C++)
$> $VECTORCAST_DIR/vcastqt -lada (to open VectorCAST/Ada)
```

To Set the Product Mode

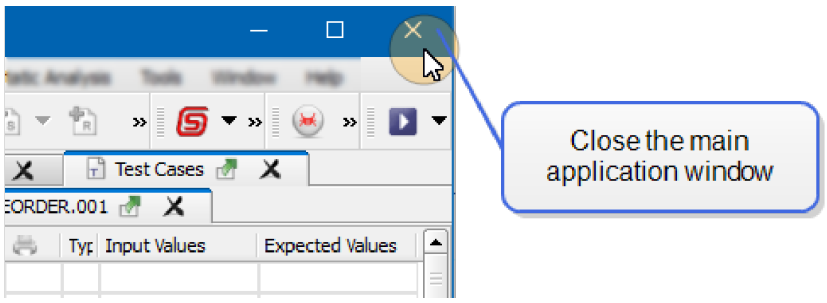
You can set or change the current product mode without having to open an environment. To do this, choose **File => Set Product Mode => <mode>**. The available options for <mode> depend on the VectorCAST products your organization is licensed to use.



If you already have an environment open, the **File => Set Product Mode** menu item is dim.

To Exit VectorCAST

The **File => Exit** command enables you to exit the VectorCAST application. You can also exit by using the shortcut of clicking the **"X"** in the top right-hand corner of the main window.



VectorCAST Interface

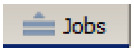
When you start VectorCAST, the VectorCAST interface (main window) appears. This window can vary slightly in detail on the basis of the licenses held. For users with only VectorCAST/Ada licensed, the VectorCAST/Ada window opens. For users with only VectorCAST/C++ licensed, the VectorCAST/C++ window opens. For users with licenses for multiple VectorCAST products, a

product-neutral VectorCAST window opens.

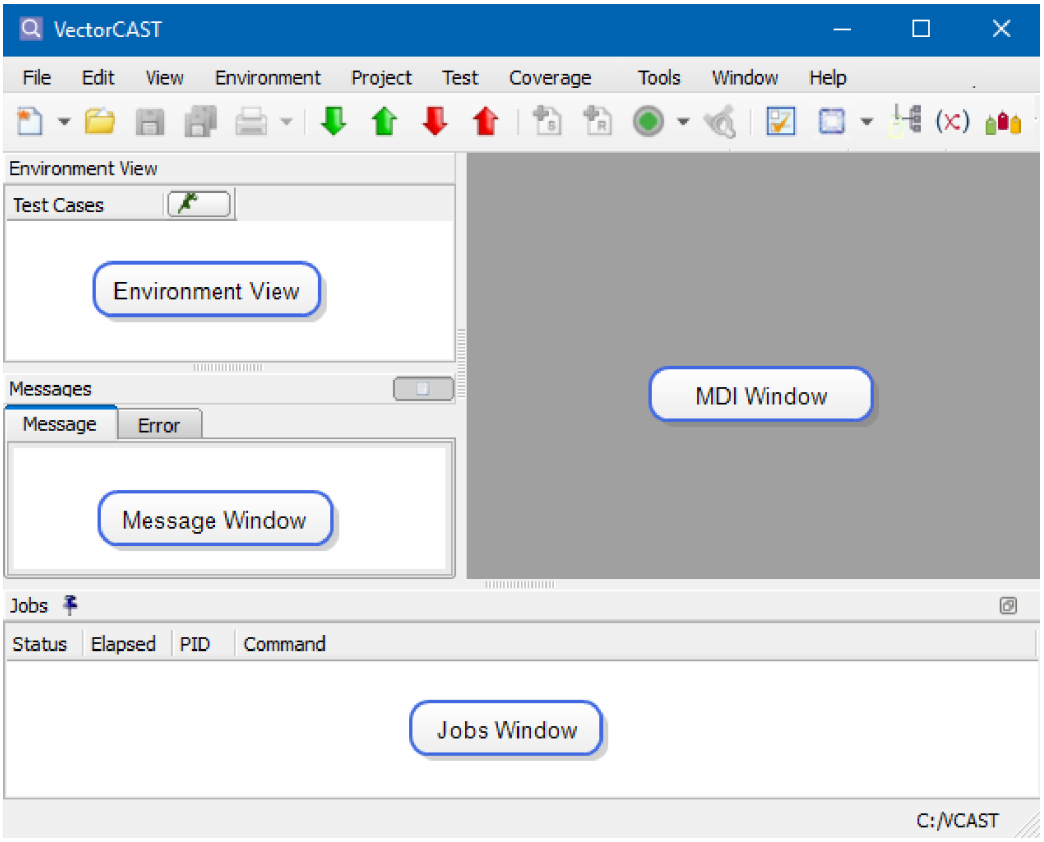
The VectorCAST main window is divided into four panes:

- The **Environment View** is located on the left-hand side of the main window. It provides a high level view of the project structure. The Environment View pane contains the Test Case Tree.
- The **Message Window** is located along the bottom left of the main window. It contains tabs for informational messages and for error messages. The Message window can be expanded and collapsed using the small  button located on the upper right corner of the message window.
- The **Multiple Document Interface (MDI) Window** is located to the right of the Project Tree. It displays a variety of windows, including Test Case editors, Coverage Viewers, Report Viewers and Text Editors. Windows are collected into groups.
- The **Jobs Window** is located on the bottom of the main window. It displays the status of jobs as they execute and exposes the associated back-end commands.

The **Environment View** and the **Message Window** are docking windows: That is, you can move these from the context of the main window to any other location on your desktop.

The **Jobs Window** is an auto-hide window. Hover over the Jobs tab 

to open it and click on the pin icon  to keep it open.



See also "To Move Docking Windows" on page 33.

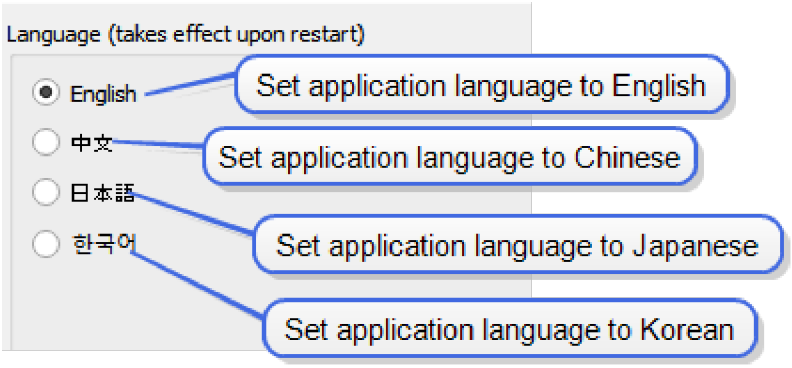
The VectorCAST tools and functions are available from a menu bar and a toolbar located at the top of the main window.

To Specify the Language for the VectorCAST GUI

Choose **Tools => Options**, and click the **GUI tab**.

By default, the VectorCAST application displays text in English, using the Latin1 character encoding. You can set the application to display in Chinese, Japanese or Korean by selecting the radio button for the desired language. Available languages are displayed in their native script for better recognition and usability. Note that you must exit VectorCAST and restart it for the change to take effect.

This information is saved to a file named **`.vcast-qt`** in the user's HOME directory and persists across sessions for a consistent user experience.



Multibyte Characters

If source code files use the character encoding Shift JIS, select the option **Wide-character support**, located in the **Options** dialog. Choose **Tools => Options**, and click the **Builder** tab.

```
clicast option VCAST_WIDE_CHARACTER_SUPPORT True | False

Allow VectorCAST to parse Ada code containing wide characters by using
ADA.WIDE_TEXT_IO when reading code. When this option is off, VectorCAST
reverts to using ADA.TEXT_IO. The default value is True.
```

To Set the Industry Mode for Coverage

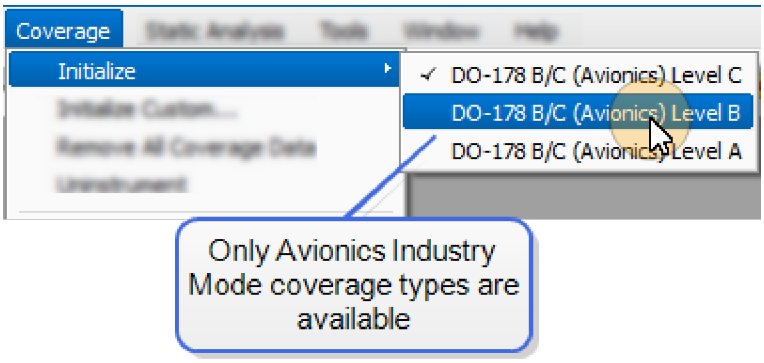
Setting an Industry Mode provides familiar industry specific choices when selecting a coverage type. The supported Industry Modes and their associated coverage types are listed below.

- Avionics (DO-178 B/C)
  - Level A (Statement, Branch, MC/DC),
  - Level B (Statement, Branch)
  - Level C (Statement)

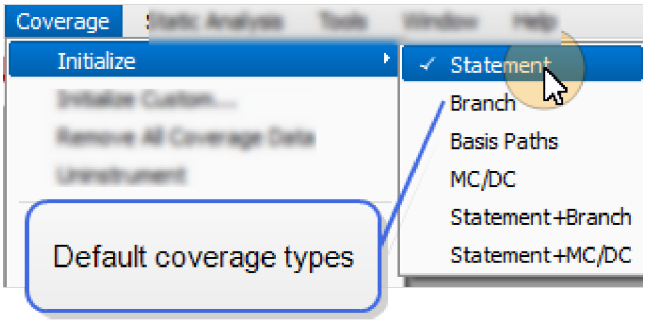
- Automotive (ISO-26262)
  - Unit Level ASIL A (Statement)
  - Unit Level ASIL B/C (Statement, Branch)
  - Unit Level ASIL D (Statement, Branch, MC/DC)
- Industrial (IEC-61508)
  - SIL4 (Statement, Branch, MC/DC)
  - SIL3 (Statement, Branch)
  - SIL 1/2 (Statement)
- Railway (EN-50128)
  - SIL4 (Statement, Branch, MC/DC),
  - SIL3 (Statement, Branch)
  - SIL 1/2 (Statement)
- Medical (IEC-62304)
  - Class C (Statement, Branch, MC/DC)
  - Class B (Statement, Branch)
  - Class A (Statement)
- Default
  - Statement
  - Branch
  - Basis Paths
  - MC/DC
  - Statement + Branch
  - Statement + MC/DC

To select the industry mode, select the **Tools=>Industry Mode** command and then select the specific Industry Mode from the context menu.

When initializing or selecting a coverage type, the industry appropriate coverage types are shown.

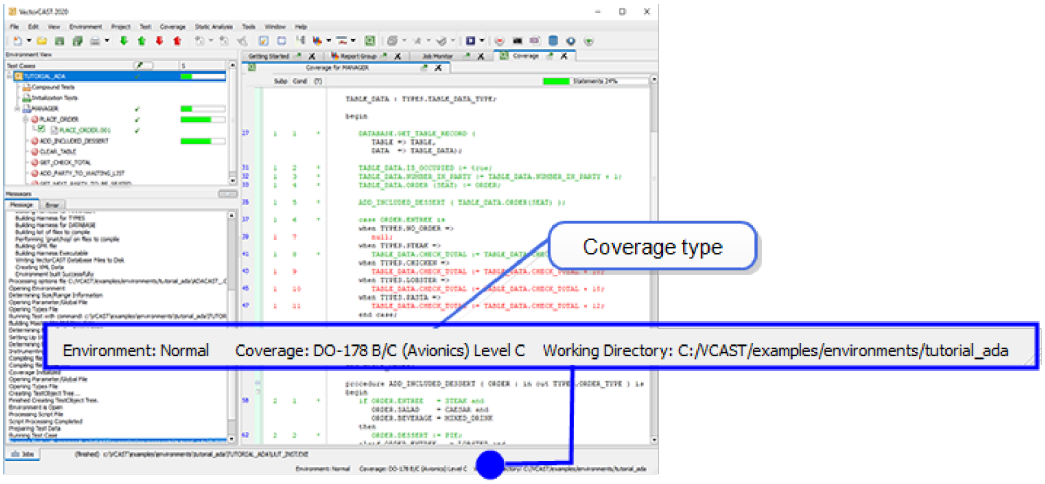


If you do not set an Industry Mode, the Default mode is used and you will be presented with the Default coverage types when choosing or initializing coverage.

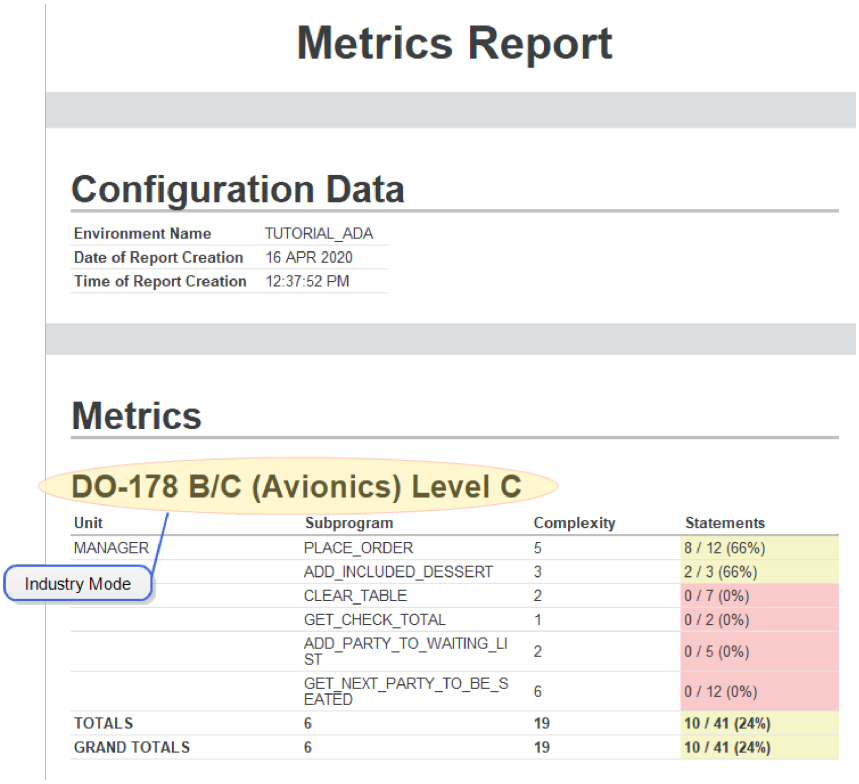


Once an Industry Mode is selected, it is stored in the **`.vcast-qt`** file and the chosen Industry Mode remains in effect until a different Industry is selected.

When coverage is initialized, the industry appropriate coverage type is displayed in the status bar at the bottom.



Coverage reports reflect the associated coverage categories, Statements, Branches, MC/DC Pairs, for the coverage type chosen. In addition, the Industry Mode will be shown on the report.



Metrics Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	16 APR 2020
Time of Report Creation	12:37:52 PM

Metrics

DO-178 B/C (Avionics) Level C

Industry Mode

Unit	Subprogram	Complexity	Statements
MANAGER	PLACE_ORDER	5	8 / 12 (66%)
	ADD_INCLUDED_DESSERT	3	2 / 3 (66%)
	CLEAR_TABLE	2	0 / 7 (0%)
	GET_CHECK_TOTAL	1	0 / 2 (0%)
	ADD_PARTY_TO_WAITING_LIST	2	0 / 5 (0%)
	GET_NEXT_PARTY_TO_BE_SEATED	6	0 / 12 (0%)
TOTALS	6	19	10 / 41 (24%)
GRAND TOTALS	6	19	10 / 41 (24%)

Changing Industry Mode

For existing environments that have a coverage type that is contained in the table below, changing the Industry Mode will result in an immediate translation of coverage type from one Industry Mode to another. This immediate translation is reflected in the status bar of the application and in any test reports generated after the change.

Default	Avionics	Automotive	Industrial	Railway	Medical
Statement	Level C	Unit Level ASIL A	SIL 1/2	SIL 1/2	Class A
Statement + Branch	Level B	Unit Level ASIL B/C	SIL 3	SIL 3	Class B
Statement + MC/DC	Level A	Unit Level ASIL D	SIL 4	SIL 4	Class C

For example, if the Industry Mode currently is set to Automotive and the coverage type is ASIL A, if the Industry Mode is changed to Avionics, using the table, the coverage type will be translated to Level C.

If the current Industry Mode is Default, and the current coverage type is Branch, Basis Path or MC/DC, when the Industry Mode is changed the current coverage type is retained. However, the coverage choices in **Coverage => Initialize** or in the Build Options step of the Update Wizard will reflect the industry specific coverage types.

If the environment is rebuilt (**Environment => Rebuild Environment**) after changing the Industry

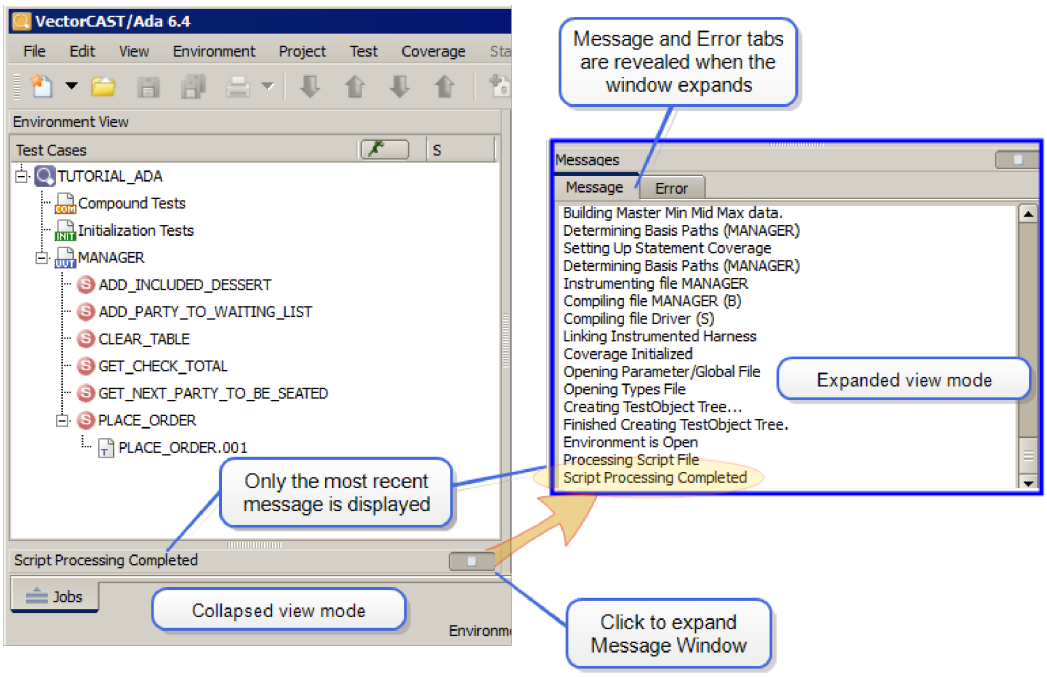
Mode from default, the Branch, Basis Path or MC/DC coverage type is retained.

If the environment is updated (**Environment => Update**) after changing the Industry Mode from default, the Branch, Basis Path or MC/DC coverage type is set to None.

For environments that were created with a version of VectorCAST prior to 6.0, Level A, Level B, and Level C coverage types will be mapped using the table above as if the Industry Mode was set to Avionics prior to the Industry Mode change.

To Collapse / Expand the Message Window

The Message window can operate in a collapsed, single line mode. When the Message window is "collapsed" to show only a single line, it takes less space vertically. Only the most recent message is displayed, and the Message and Error tabs are hidden until the window is expanded.

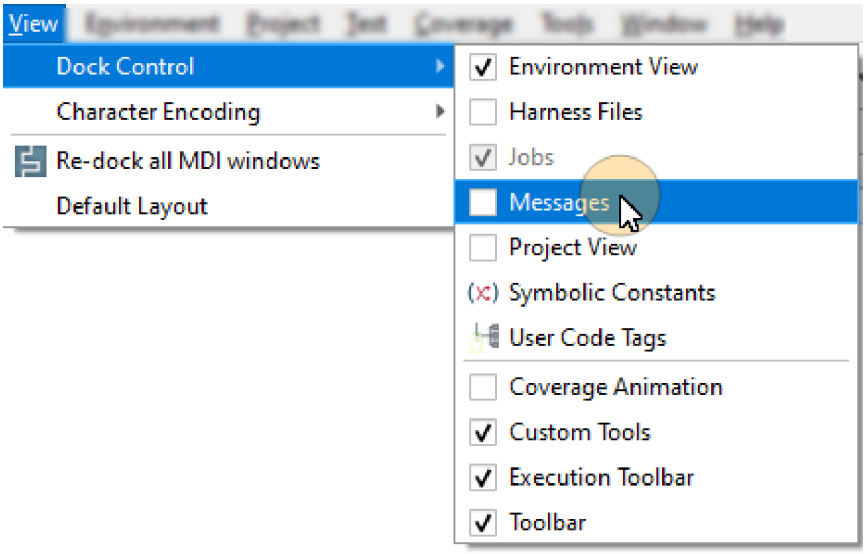


To expand the Message window to see all of the text in the Message window, click the small  button located on the right side of the single line message. This action can be done almost at any time, and is specifically permitted between execution of multiple tests.

The expanded Message window consists of the following tabs: **Message** and **Error**. The **Message** tab displays user status messages. The **Error** tab displays VectorCAST diagnostic messages.

To Hide the Message Window

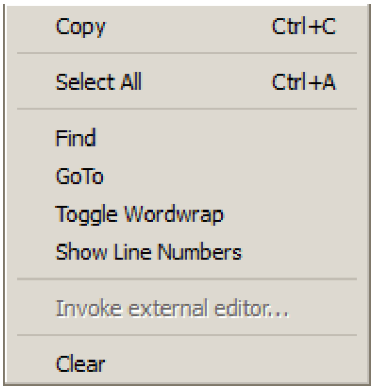
To hide the Message window in the main window, choose **View => Dock Control** and uncheck **Messages**.



To make it visible again, choose **View => Dock Control => Messages**, or **View => Default Layout**.

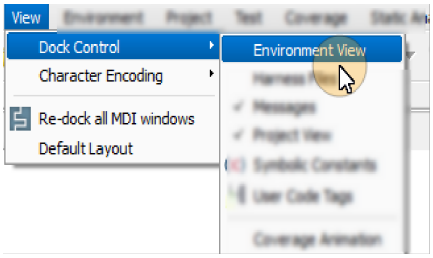
To Clear or Copy Text in Message Window

You can Copy, Select All, or Clear the text in the Message window by right-clicking in the Message window.



To Hide the Environment View

To hide the Environment View in the main window, choose **View => Dock Control** or right-click the handle on the Environment View, and uncheck the **Environment View**.



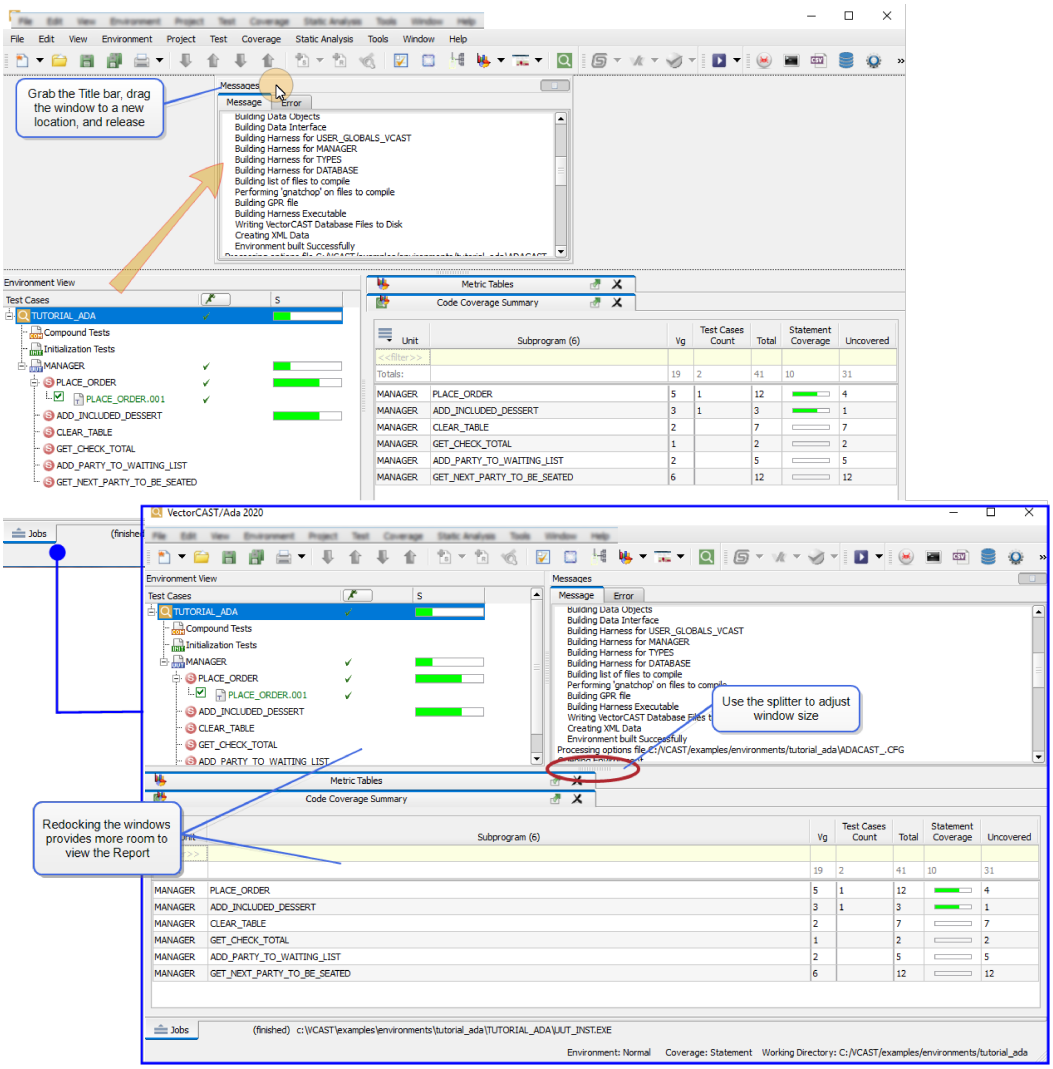
To make it visible again, choose **View => Dock Control => Environment View**, or **View => Default Layout.**

To Move Docking Windows

The Test Case Tree and Message Window are docking windows, meaning they can be moved within the main application window or float outside of the application window.

To move the Message Window or the Test Case Tree to a new location, grab the window by its top Title bar and drop the window within the boundaries of the VectorCAST main window.

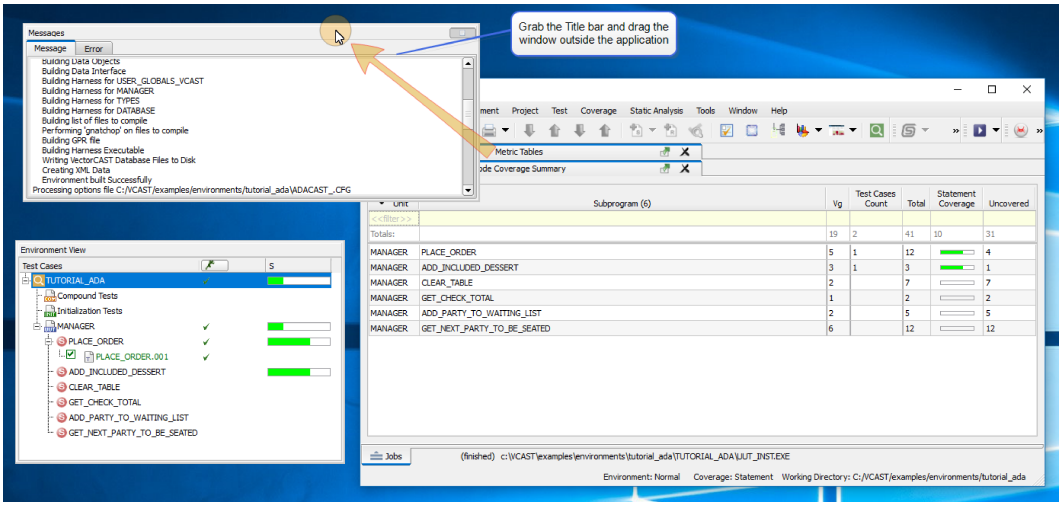
For example, to make more room in the MDI window, you can drag the Message window and Test Case Tree and dock them above the MDI window. To do this, drag the Title bars of the Message window and Test Case Tree and drop them above the MDI window. Use the splitter to adjust the window size.



Use **View => Default Layout** to return the main window to its default layout, if desired.

To Float Docking Windows

To float the Message Window or the Test Case Tree on your desktop, grab the window by its Title bar and move it outside the boundaries of the main window. You can then resize the window, as shown below.



Use **View => Default Layout** to return the window to its default layout, if desired.

To Return Docking Windows to Default Locations

Both the Message Window and the Test Case Tree can be moved from their default locations by dragging them by their Title bars. To return them to their default locations, choose **View => Default Layout**. If the Message Window appears as a column in the middle, just make the main application window a little larger.

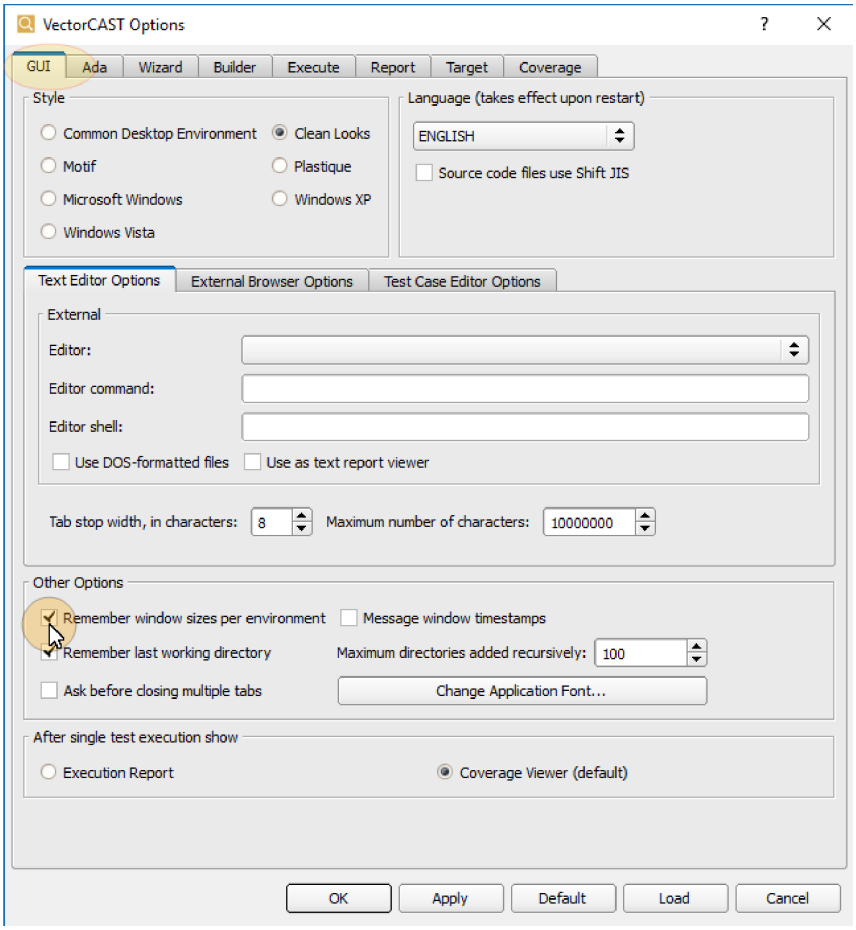
To Save the Window Configuration

When you change the current VectorCAST window configuration, you can save the new configuration for use in all subsequent sessions, or you can save it for use only in sessions involving the same environment.

By default, VectorCAST saves the current configuration for use in all subsequent sessions. In other words, if you change the window configuration, exit VectorCAST, and then return, the window configuration will be the same as it was when you exited the previous session.

To save the current window configuration for use only in subsequent sessions involving the same environment, choose **Tools => Options => GUI tab**, then check **Remember window sizes per environment**.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.



The Toolbar

The Toolbar shows all icons associated with the various VectorCAST commands. The Toolbar includes icons for third-party tools integrated with VectorCAST. The icons are functional only if you are licensed for these features.

By default, the Toolbar is visible. If you move the Toolbar you can return it to its default position by choosing **View => Default Layout**.

Toolbar Icon	Description	Comment
	Create new environment	
	Open new file or environment	
	Save the current file in MDI	Ctrl+S
	Save all	
	Print	
	Next matched result	Ctrl+Down

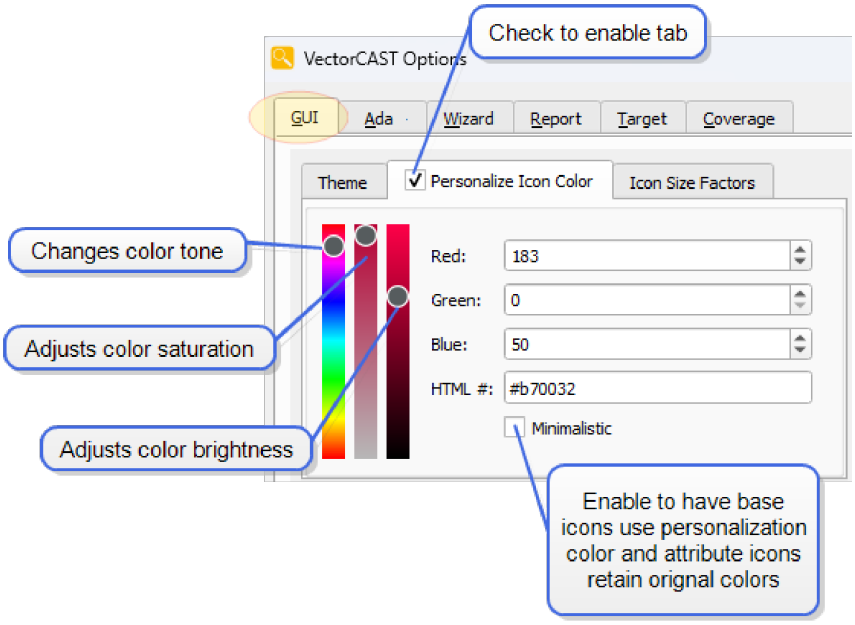
Toolbar Icon	Description	Comment
	Previous matched result	Ctrl+Up
	Next unmatched result	Shift+Down
	Previous unmatched result	Shift+Up
	Add source file	
	Add test result	
	Edit Probe Points	
	Add Coverage Analysis	
	Tool Options	
	Open Requirements Gateway	
	User Code Tags	
	Symbolic Constants	
	Create and assign Variant Logic to test cases	
	Data Summary Report	
	View Coverage	
	RSP documentation	
	Execute test case	
	ATG	
	Bug Tracking	
	Command Prompt	
	Generate CSV Execution Report	
	Settings Database	
	RSP Configurator	
	RSP Installer	

Customize Icons

Users can personalize the color and size of icons within the graphical user interface. This ability to adjust icon colors and to scale subsets of icons relative to the application font size improves accessibility and usability for the user. These settings are saved to the `.vcast-qt.ini` file, located in the user's home directory, and persist across sessions for a consistent user experience.

Personalize Icon Color

To change the color of the icons, choose **Tools => Options**, and select the **GUI** tab. Select the **Personalize Icon Color** tab and insert a checkmark to enable the tab. Once the tab is enabled, customize the colors by adjusting RGB values or using a HEX code. Enter the desired values directly in the fields provided. Move the associated sliders to adjust the color as needed. The slider in the first position changes the color tone. The slider in the second position adjusts color saturation. The slider in the third position adjusts color brightness. Enabling the Minimalistic checkbox will have all base icons use the selected personalization color, while attribute icons, such as exclamation marks or checkmarks, will retain their original colors.

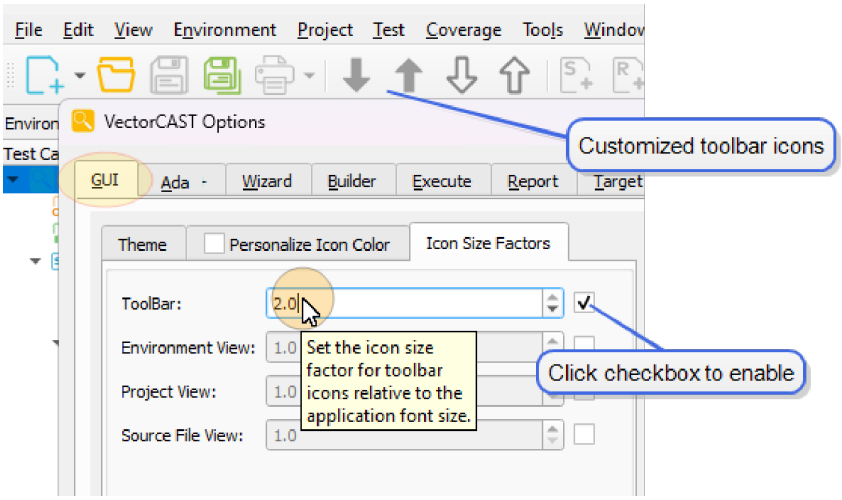


Icon Size Factors

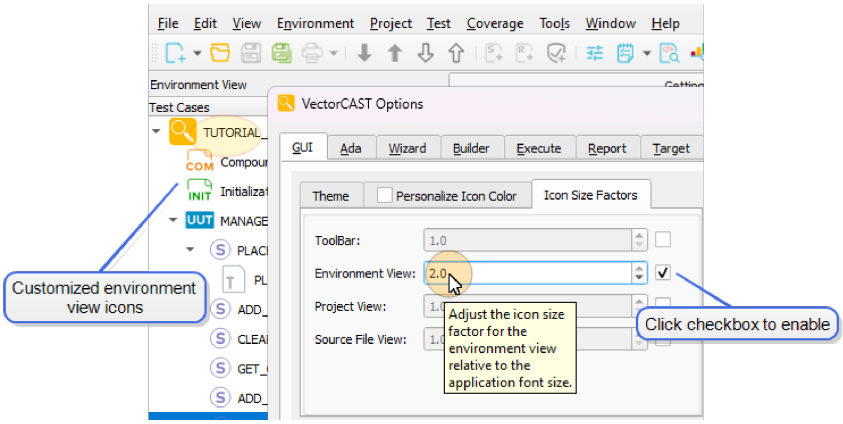
To change the size of the icons, choose **Tools => Options**, and select the **GUI** tab. Then select the **Icon Size Factor** tab. Customized icon sizing is enabled and disabled by selecting the checkboxes. Insert a checkmark to enable editing of the desired view. Enter the desired size factor in the field provided for the selected view. Icon size is determined by multiplying the application font size by the selected factor.

Icon size customization is available for four views:

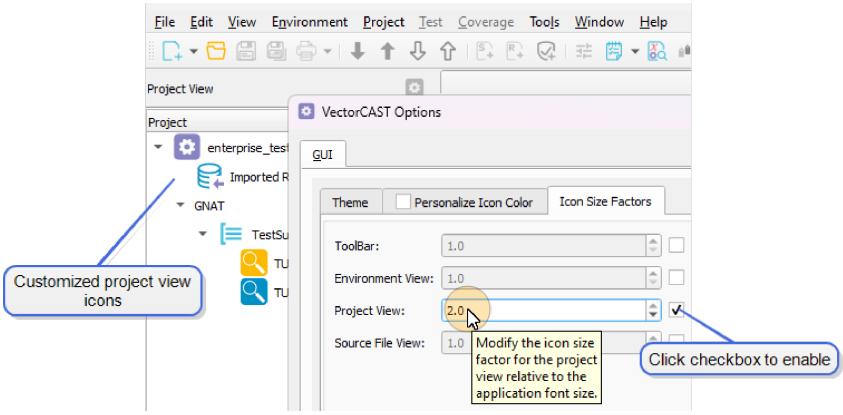
1. Toolbar



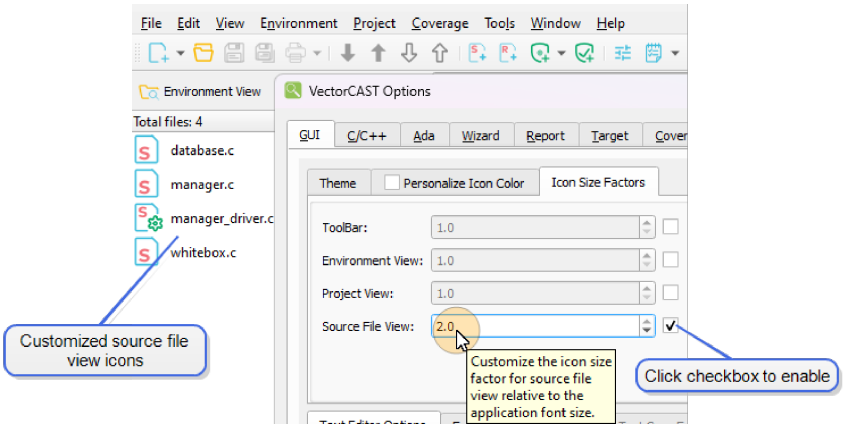
2. Environment View



3. Project View



4. Source File View



The Status Bar

When an environment is open, the Status bar shows the status of the Environment, the type of Coverage initialized, and the current working directory.

The environment's status should be "Normal"; if an error occurs when building or rebuilding an environment, the status changes to one of the following:

- Compile Error – An error occurred while compiling the test harness. You will need to correct the problem and recompile the environment. Choose **Environment => View => Compile Errors** to view the compile errors.
- Link Error – An error occurred while linking the test harness. You will need to correct the problem and relink the environment. Choose **Environment => View => Link Errors** to view the linker errors.
- Data Interface Error – The test harness linked but generated a catastrophic error during program elaboration.

If coverage has not been initialized, then the Coverage status is "None." If you have initialized any of the types of coverage (statement, branch, etc.), then the status is changed to "*Type*," where *Type* is one of the types of coverage. Note that if you disable coverage, the status changes to "*Type* (disabled)". Enabling coverage removes "disabled" from the status.

The right-most section of the status bar shows the full path to the current working directory. It is the default directory in the File Save dialog when saving a report or exporting a script to a file, and the default directory in the File Open dialog when opening a file or importing a script.

To Open a File Named in a Report

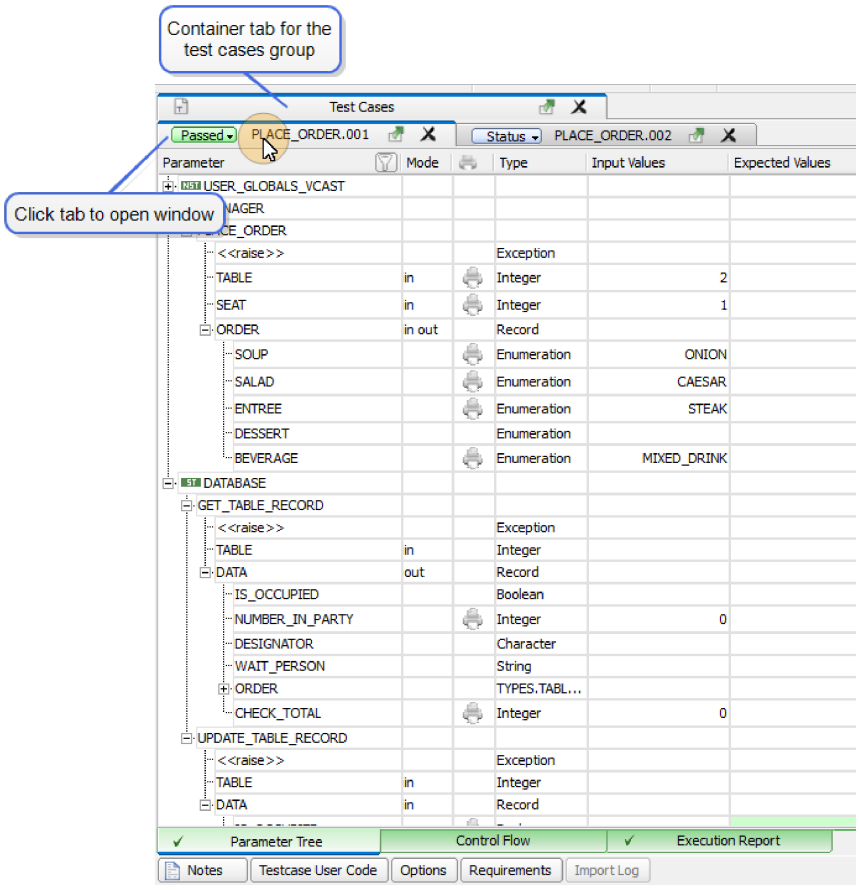
If you highlight a filename displayed in any VectorCAST window and right click on the name, the file that you selected is opened. Additionally, if you select the name of a header file, VectorCAST looks for the header file in the Search Directories as well as Include Directories.

The MDI Window: Groups

The MDI window is always visible. When an environment is open, it is used to display a variety of windows, including Test Case editors, Coverage Viewers, Report Viewers, and Text Editors. The various types of windows are collected into *groups;* as you open a new window it is added as a tab to

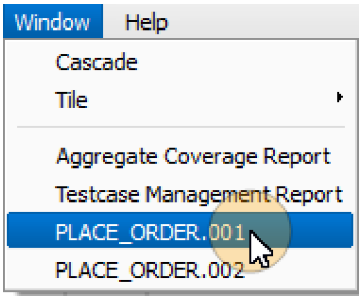
an existing group, or a new group is created.

You can switch between different windows in the group by clicking the named tab at the top.



To Open a Group

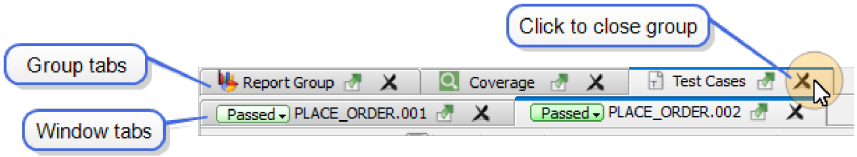
The lower half of the Window menu contains a list of open windows. To switch between windows, choose **Window => <window name>**.



To Close a Group

To close a group, click the X in the upper right corner of the group tab. Closing a group closes all the

tabs in that group. If any tabbed window requires saving before it can be closed, you are asked if you want to save it.

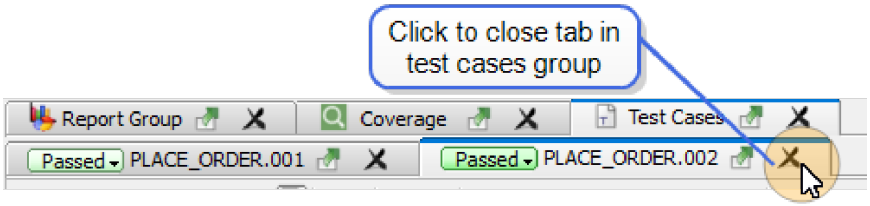


To Maximize a Window

You can maximize a window by un-docking it and clicking the maximize button in the title bar of the window.

To Close a Tab

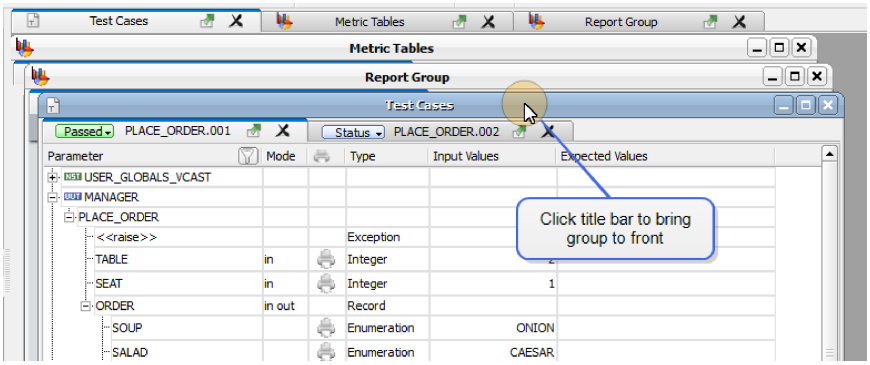
To close a tab in a group, click the **X** in the tab. Alternatively, right-click a tab and choose **Close** from the context menu.



Alternatively, selecting the **File => Close** command from the Menu Bar enables you to close the current active tab in the MDI window. Note that It does not close the environment, only the tab.

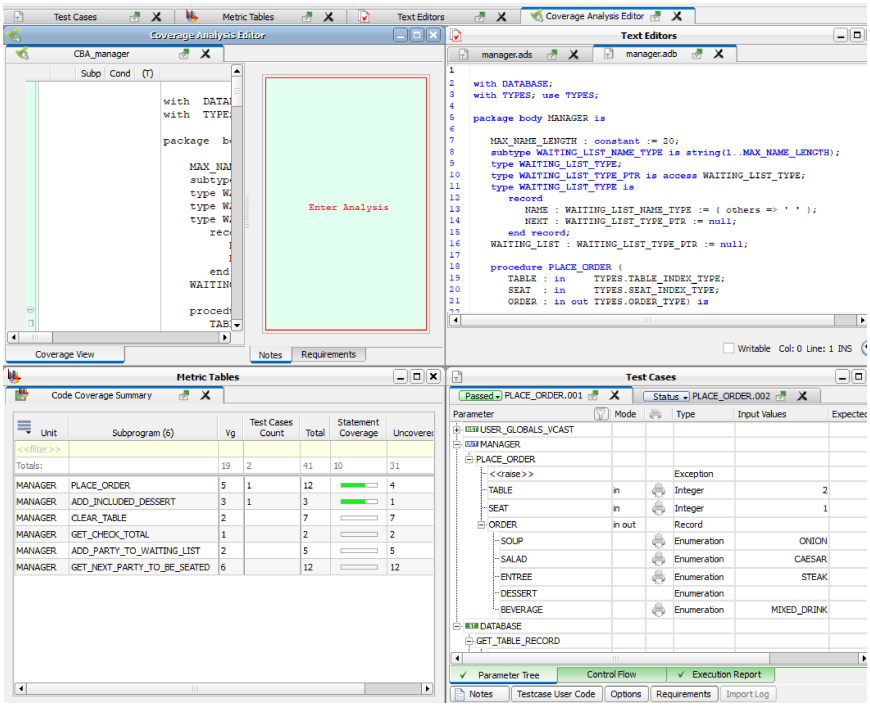
To Arrange Groups in a Cascade

Choose **Window => Cascade** to layer all open groups, showing each title bar. To bring a group to the front, click its title bar.

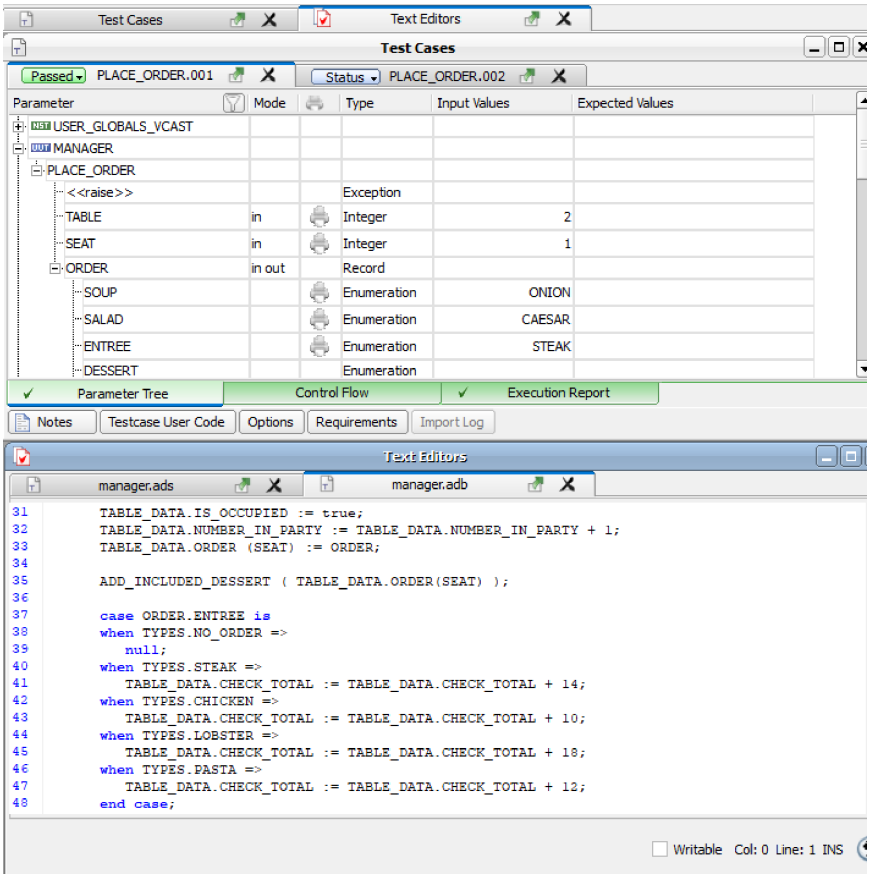


To Tile Groups

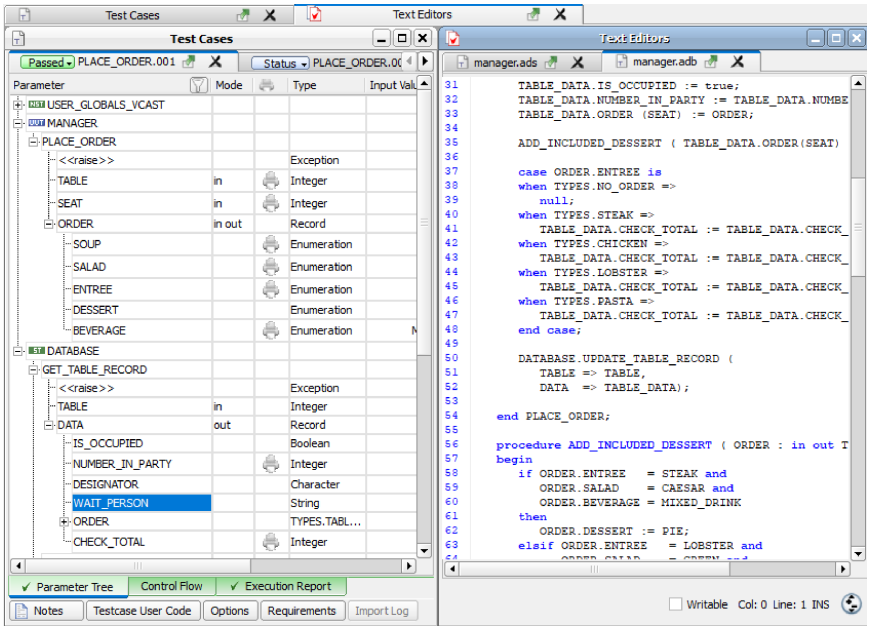
The **Window => Tile => Grid** menu item arranges all open windows in a grid pattern. Each window is reduced to a rectangle, and set next to another window. There is no overlap or layering. The keyboard shortcut is Ctrl+Alt+G.



Choose **Window => Tile => Horizontal** to arrange the groups horizontally. The figure below shows two groups tiled horizontally. The keyboard shortcut is Ctrl+Alt+H.

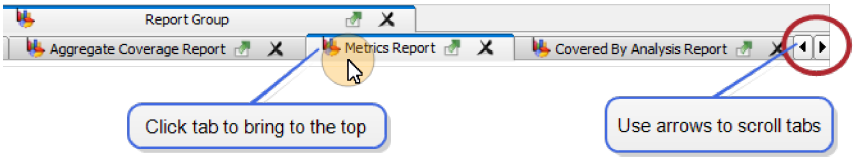


Choose **Window => Tile => Vertical** to arrange the groups vertically. The keyboard shortcut is Ctrl+Alt+V.



To Bring a Tab to the Top

A group may have many tabs open in it. To bring a particular tab to the top, you can dlick its tab. When a window is in focus, its tab is highlighted. If a tab is not visible, use the left and right arrows  to scroll the tabs to the left or right.



To Close the Current Window (Tab)

The **File => Close** command enables you to close the current active tab in the MDI window. It does not close the environment.

To Close All Windows

To close all open tabs and groups in the MDI, choose **File => Close All**. This menu item does not close the environment.

VectorCAST Keyboard Shortcuts

Tiling Window Groups	
To Tile Groups Horizontally	**Ctrl+Alt+H**
To Tile Groups Vertically	**Ctrl+Alt+V**
To Tile as a Grid	**Ctrl+Alt+G**

Editing, Searching and Printing	
Create a New Text File	**Ctrl+T**
Go to a Line	**Ctrl+G**
Select All	**Ctrl+A**
Delete	**Ctrl+Del**
Undo	**Ctrl+Z**
Redo	**Ctrl+Y**
Cut	**Ctrl+X**
Copy	**Ctrl+C**
Paste	**Ctrl+V**
Save	**Ctrl+S**
Find	**Ctrl+F**
Repeat a Search	**F3**
Print	**Ctrl+P**

Expand / Collapse Windows

Available in a Unit Test environment, Cover environment, or VectorCAST project:

Expand / Collapse Message Window	**Ctrl+Alt+m**
Expand / Collapse Jobs Window	**Ctrl+Alt+j**

Available in a Unit Test environment or Cover environment that has been opened from within a VectorCAST project:

Expand / Collapse Project View	**Ctrl+Alt+p**

Building Test Cases

Available from Test Case Tree:

Insert a Test Case	**Ctrl+Insert**
Open a Test Case	**Ctrl+Return**
Open Source	**Ctrl+Return**
Rename a Test Case	**Ctrl+Shift+R**
Rename an Environment	**Ctrl+R**
Expand a collapsed node	**Right Arrow**
Collapse an expanded node	**Left Arrow**

Available from Compound Test Case Tree:

Open an Individual Test Case from a Compound Test Case	**Ctrl+Shift+O**
Delete a Test Case from a Compound	**Del**
Move Slot Up	**Ctrl+Shift+Up**
Move Slot Down	**Ctrl+Shift+Down**
Find Slot in the Test Case Tree	**Ctrl+Shift+F**

Available from the Test Case Editor:

Expand a collapsed node	**Right Arrow**
Collapse an expanded node	**Left Arrow**
Display list of enum items for a selected parameter	**Alt+Down Arrow**
Switch between Input Value and Expected Value	**Tab**

Executing Test Cases

Batch Execute	**F6**
Edit Properties	**Shift+P**

Available from the Test Case Tree:

Execute	**F5**

Working With Control Flow

Available from the Control Flow tab in the Test Case Editor:

Move Up	**Ctrl+Shift+Up**
Move Down	**Ctrl+Shift+Down**
Remove Subprogram	**Del**

Working With Coverage

Available from the Test Case Tree:

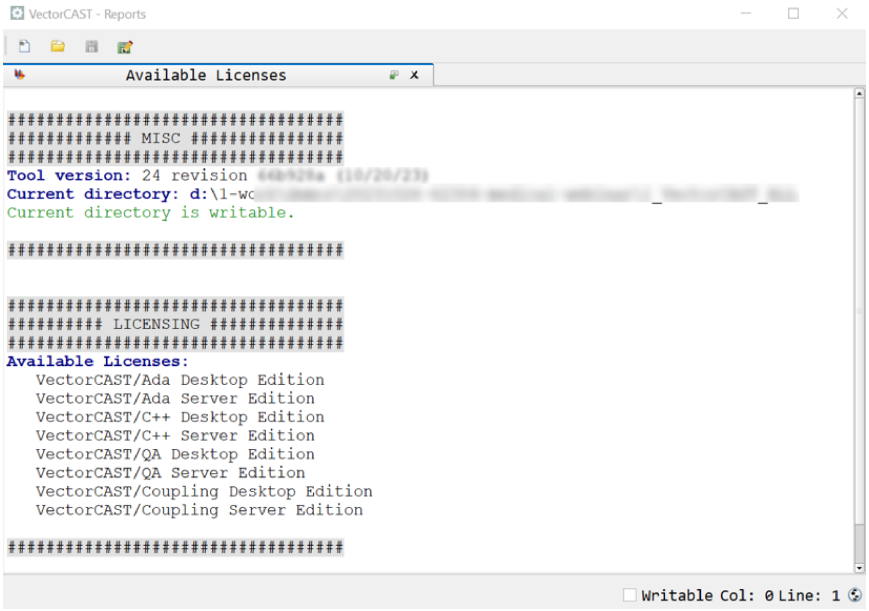
View Coverage	**Shift+Return**
Select Coverage	**Ctrl+Shift+S**
Deselect Coverage	**Ctrl+Shift+D**

Exit VectorCAST	**Ctrl+Q**

Getting Help

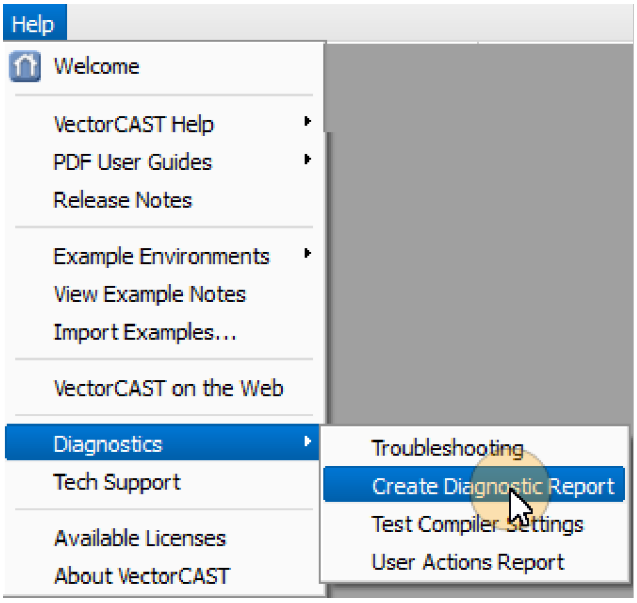
To Determine Available Licenses

To determine which VectorCAST products your organization is licensed to use, choose **Help => Available Licenses**:

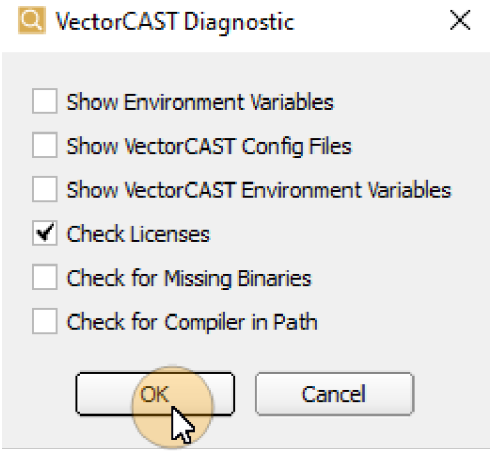


An Available Licenses Report is displayed in the MDI window, listing all available licenses.

Alternatively, to determine which features and targets are licensed, choose **Help => Diagnostics => Create Diagnostic Report**:



Check **Check Licenses** on the Diagnostics popup, click **OK**, and then scroll down the report to the Licensing section.




```
clicast -lada REports DIagnostic [<outputfile>]

Generates diagnostic report to standard output or the specified output
file.
```

To View Online User Guides

The Help menu gives you access to the various VectorCAST User Guides in PDF and HTML format. Adobe Acrobat Reader must be available on your system in order to view the PDF files. The following documents are available:

- Release Notes, which include the changes and improvements to VectorCAST since the last released version.
- User Guides for all VectorCAST products.

The User Guide .PDF files are located in the VectorCAST installation directory, **`/docs/PDF`** sub-directory, and can be accessed by selecting **Help =>PDF User Guides** from the Menu Bar. Note that only User Guides for licensed products are available from the menu.

.PDF filename	User Guide
vcast_quick_start.pdf	*Quick Start Guide*
vcast_enterprise.pdf	*Enterprise Testing With VectorCAST*
vecchelp.pdf	*VectorCAST/C++ User's Guide*
veachelp.pdf	*VectorCAST/Ada User's Guide*
vcqa.pdf	*VectorCAST/QA User's Guide*
vcast_utilities.pdf	*VectorCAST Utilities User's Guide*

.PDF filename	User Guide
vcast_install_guide.pdf	*VectorCAST Installation Guide*
fnp_LicAdmin.pdf	*License Administration Guide*

In addition to the PDF versions of the User Guides, VectorCAST includes HTML Help. Use the **Help =>VectorCAST Help** menu item to access the HTML guides. After selection, the guide opens in a VectorCAST browser window which is detached from the main VectorCAST GUI. If multiple help guides are opened, each will exist in a separate tab in the browser window.

To Contact Technical Support

Choose **Help => Tech Support** to find contact information for VectorCAST Technical Support.

Email: **support@vector.com**
Web: **https://support.vector.com**

To Send a Test Environment to Technical Support

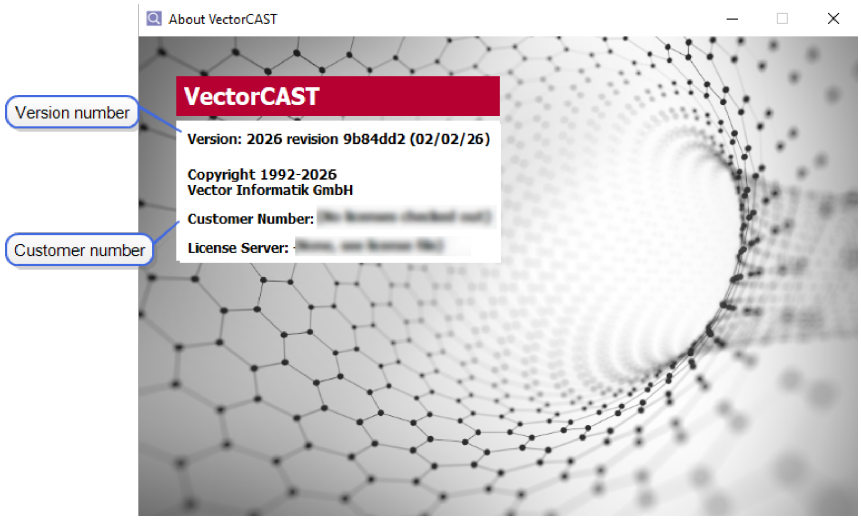
Sometimes, Tech Support may need to review your test environment to answer a question. The easiest way to send your test environment is to compress it (zip, tar, or gzip) and email it to **support@vector.com**.

If your environment is named TEST, for example, you should send the entire contents of the directory named TEST along with the file TEST.vce. Under Windows, you can use PKZip or WinZip; under Linux, you can use tar and compress or gzip to create an archive file that can be emailed to Tech Support.

If the size of the compressed file is greater than 5 MB, please send an email to support@vector.com and they will instruct you on how to send the file.

To Determine the VectorCAST Version and Customer Number

Choose **Help => About VectorCAST** to determine VectorCAST's version number and build date. Your Customer Number is also displayed.

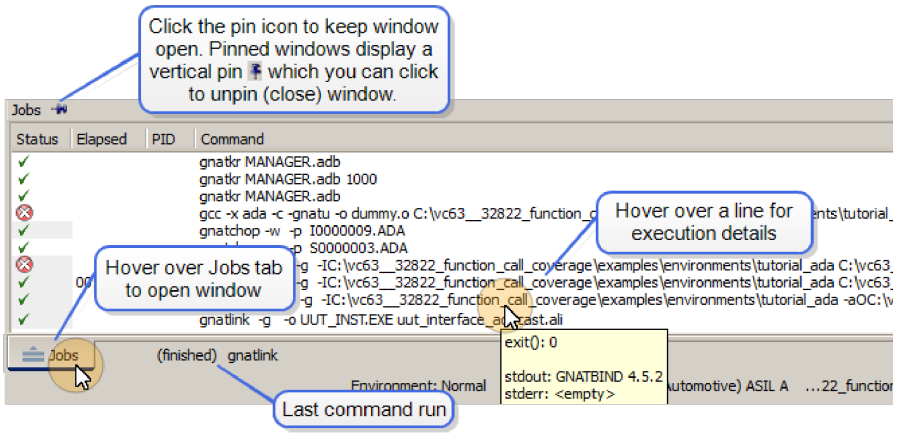

About VectorCAST
VectorCAST
Version: 2026 revision 9b84dd2 (02/02/26)
Copyright 1992-2026
Vector Informatik GmbH
Customer Number:
License Server:
Version number
Customer number

Using the Jobs Window and Jobs Monitor

It is often desirable when running a target test to view the output in real time as the commands are running. The Jobs window displays jobs as they execute and the Jobs Monitor displays the status of each job as it executes and exposes the toolchain processes. The Jobs Monitor also allows the user to diagnose failing commands and test a possible fix.

The Jobs Window

The Jobs window is an auto-hide window located at the bottom of the main window. If the Jobs window is hidden, the status bar displays the current or last command run. Hover over the window to open it, and use the pin icon to keep the window open.



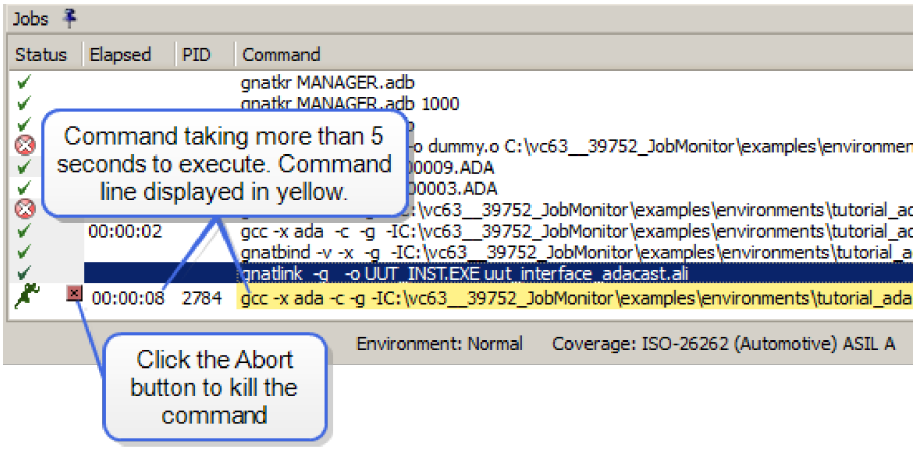
The Jobs window displays the full command line for every invocation of an executable by VectorCAST's back end. Such executables include the compilers, .bat files, python and any other commands called while building the environment.

Single-clicking on a command line highlights the associated line in the Messages window. Hovering over a command line shows the exit code and stdout and stderror for that line.

For each command, the status, execution time elapsed and Process ID (PID) is displayed. Icons in the Status column represent the following:

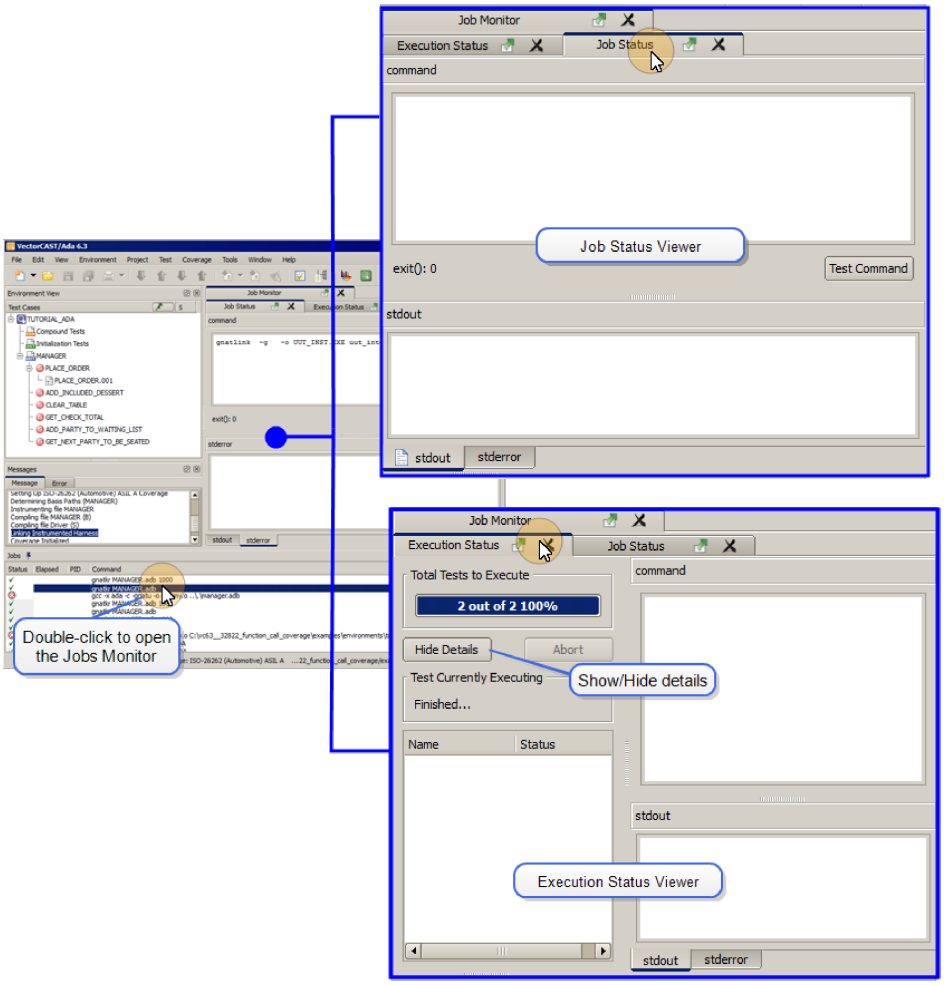
- = exit() code is 0
- = exit() code is non-zero
- = Click to abort job
- = Executing
- = Error
- = Monitored sub-process (used in Verbose mode)

A command taking more than 5 seconds to execute displays a yellow background. Click the Abort button to kill the command if desired.



Opening the Job Monitor

The Job Monitor is accessed via the Jobs window. The Job Monitor contains two panes: the Job Status Viewer and the Execution Status Viewer. The Job Status Viewer displays details about a command. The Execution Status Viewer provides real-time test case execution information.

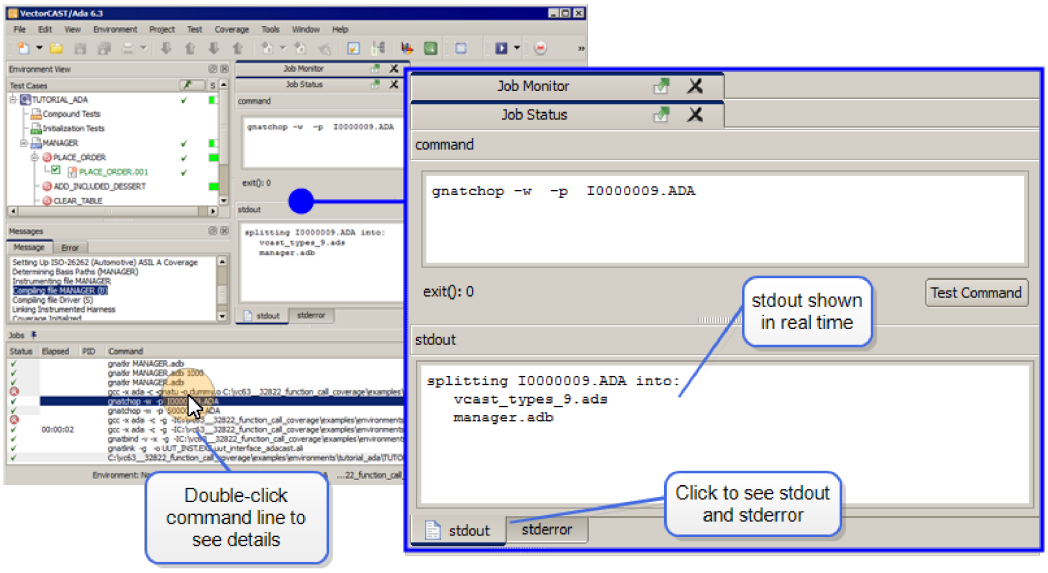


To open the Job Monitor and the Job Status viewer, double-click a command line in the Jobs window or right-click a command line and select **Job Status** from the context menu. The Job Status Viewer opens in the MDI window.

Job Status Viewer

The Job Status viewer displays details about a command and provides a way to diagnose failing commands (or "jobs") and test a possible fix. Each job is an invocation of a program, such as the VectorCAST test harness, the compiler, or linker.

The Job Status viewer is primarily used to diagnose how VectorCAST makes calls to the Target compiler, and provides the user with helpful information about what a command is and where it is run from.



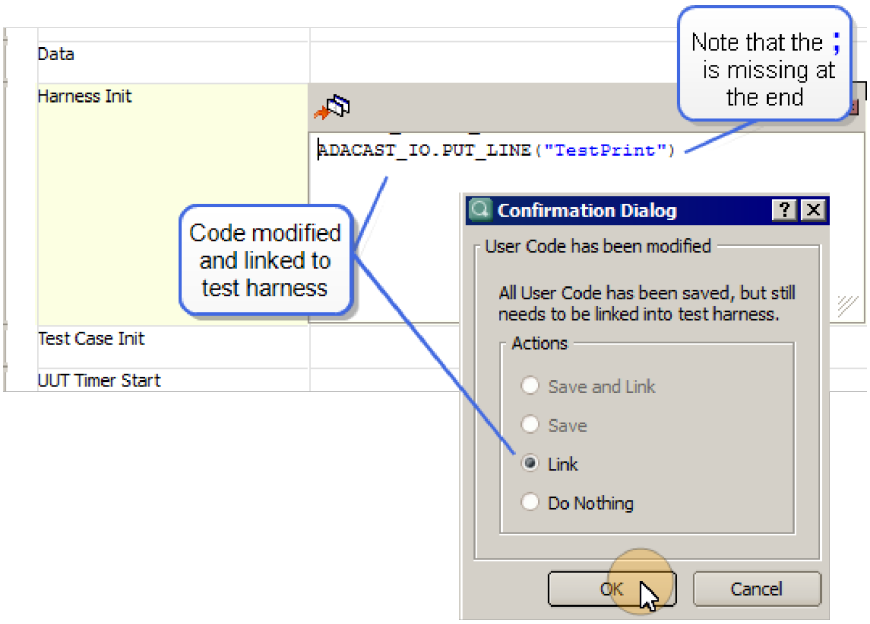
The Job Status viewer displays the executable called, any arguments and full stdout and stderror information. Click a running command to see the stdout and stderror in real time.

Debugging Using the Job Status Viewer

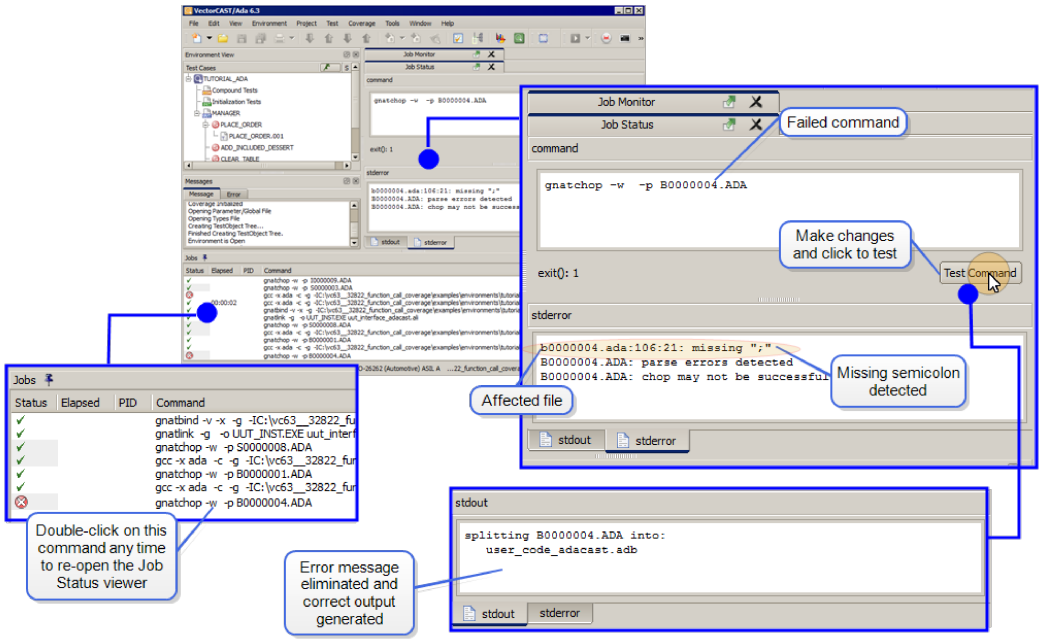
When a compile or link error occurs in user code during test execution, the Job Status viewer opens automatically, showing the command that failed and the standard output and standard error. The command displayed in the Job Status window is editable, and a **Test Command** button is provided to re-run the command.

This debugging feature provides a way to diagnose failing commands. It does not affect the state of the environment or resolve errors caused by the original command failing to run.

For example, say the Environment User Code is modified and a coding error introduced. When the change is saved and linked, the coding error is detected and the Job Status viewer automatically opens.



You can use the information in the standard error tab to determine which test harness file has the problem, make changes to the file, save, and then click the **Test Command** button to try the command again.



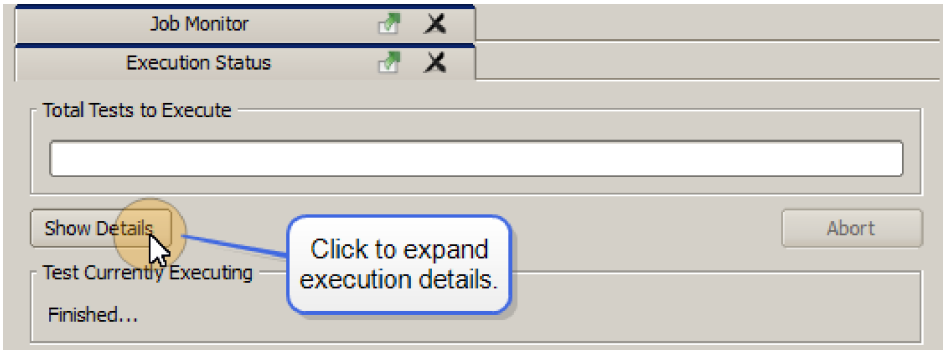
Once satisfied that you have the solution to the problem, you still need to make the change in the user code or Compiler template before attempting the test execution again.

Execution Status Viewer

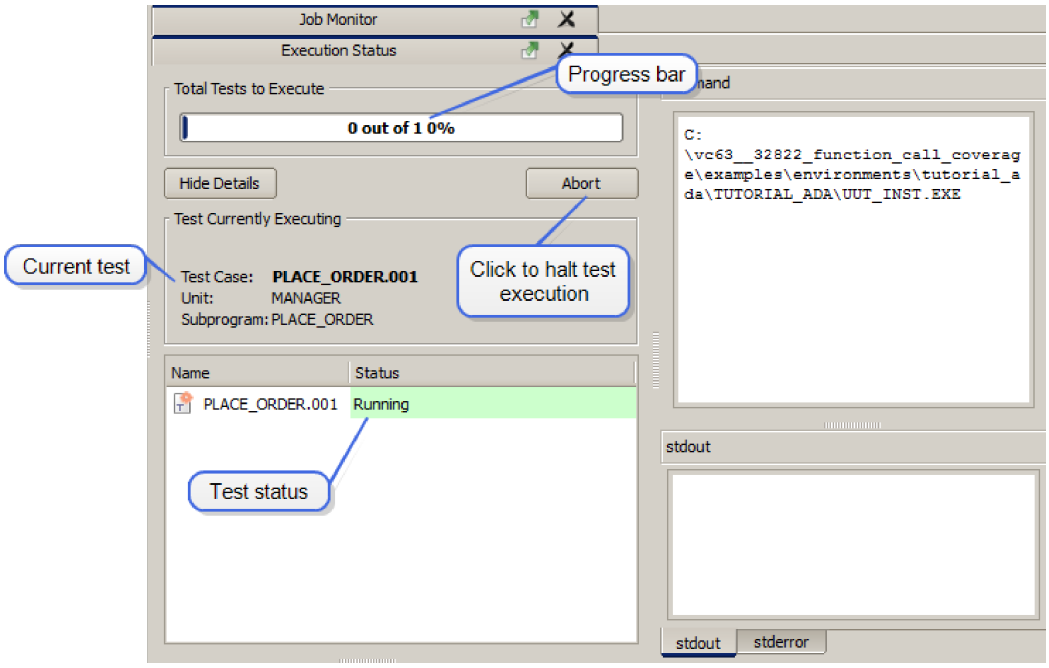
The Execution Status viewer is the counterpart to the Job Status viewer. The Execution Status viewer provides detailed real time test case execution information.

The Execution Status viewer opens automatically when a test case is executed. Click the **Show Details** button to display the full details of the execution in real time.

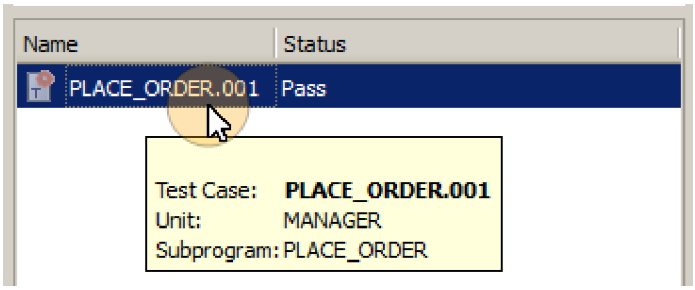
Note: When running hundreds of tests, hiding details can reduce total execution time.



A progress bar is displayed on the top left showing the number of tests remaining to execute. The name of the currently executing test is displayed beneath. An **Abort** button is available to halt the test case execution if desired and return control to VectorCAST.



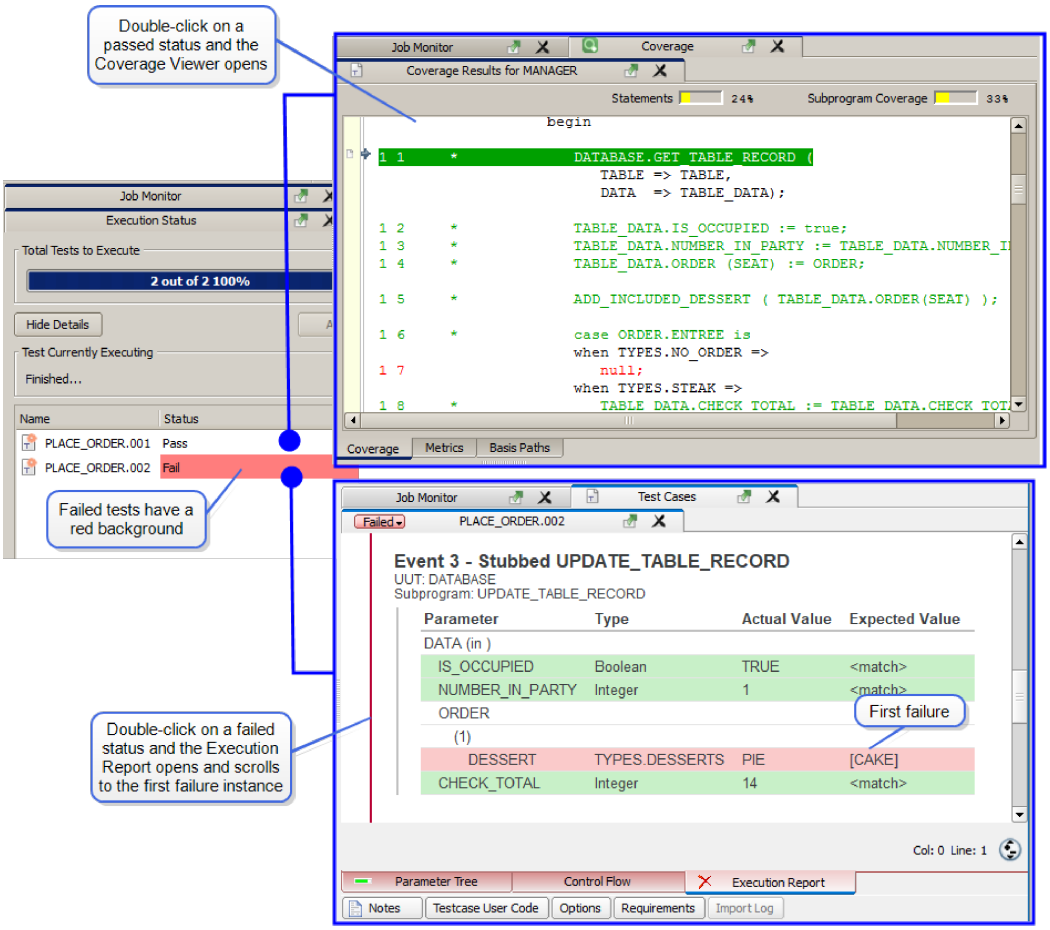
When viewing full execution details, as each test completes it is listed on the bottom left along with its pass/fail status. Hover over the name of a test case to see the name, unit and subprogram.



A deleted test displays an icon to the left of its execution status indicating that it is an invalid result now.

Hover over the status of a test case to see details of the execution. If a test aborts during execution, that information is displayed in the tooltip.

If the environment has coverage and the test passed, double-clicking on a passed status opens the Coverage Viewer. If the environment has no coverage and the test passed, double-clicking on a passed status opens the Execution Report.



Failing test cases are displayed on a red background. Double-clicking on a fail status opens the Execution Report, and the report scrolls to the first failure instance.

The full command line and the stdout and stderror details are displayed in the right column of the Execution Status viewer.

Changing Application Preferences

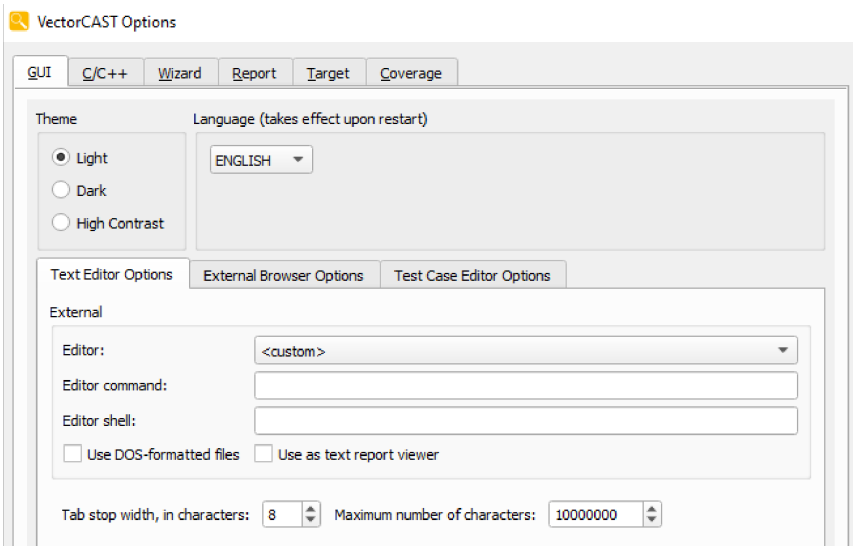
To Set the Display Theme

To select a display theme, choose **Tools => Options**, and click the **GUI** tab. Select the radio button for the desired theme and click **Apply** or click **OK** to close the dialog.

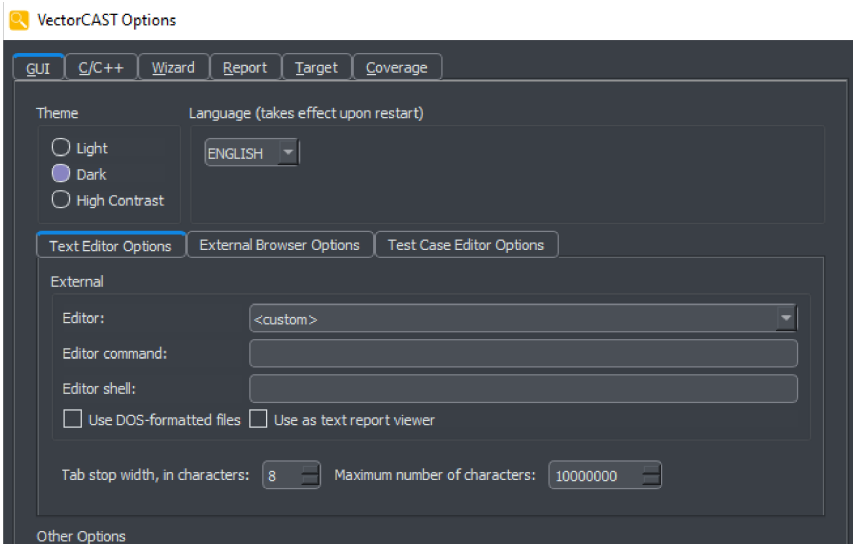
VectorCAST offers three visual display themes for selection:

- Light
- Dark
- High Contrast

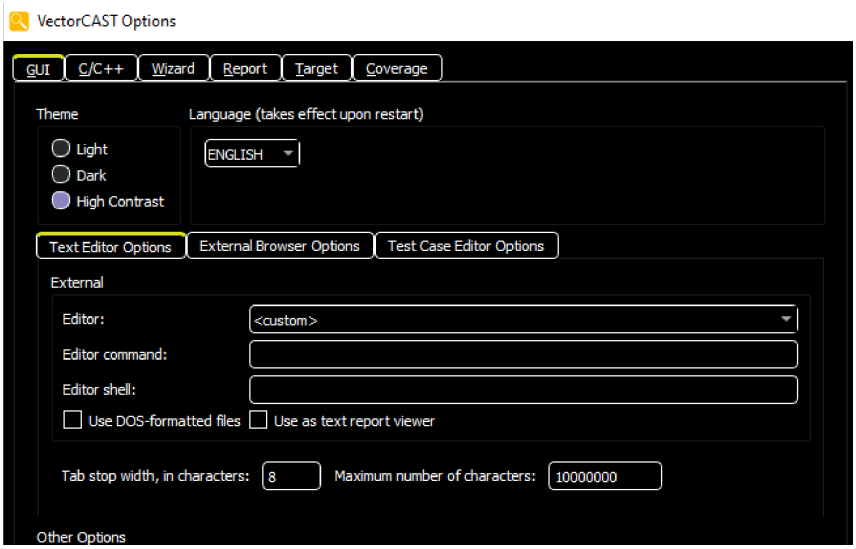
Light mode is the default display theme.



Dark mode reduces eye strain and improves visibility for certain users, particularly those with visual impairments or sensitivity to light.



High Contrast mode improves text legibility and overall visibility for those with low vision or color blindness. High contrast colors are easier to read when paired together.



To Set Up an External Text Editor

By default, all text and HTML files opened in VectorCAST are displayed in the MDI window. If you want to use an alternative editor for text files and text reports, you may configure this by selecting **Tools => Options** from the Help Menu, then selecting the **GUI** tab, **Text Editor Options** sub-tab.

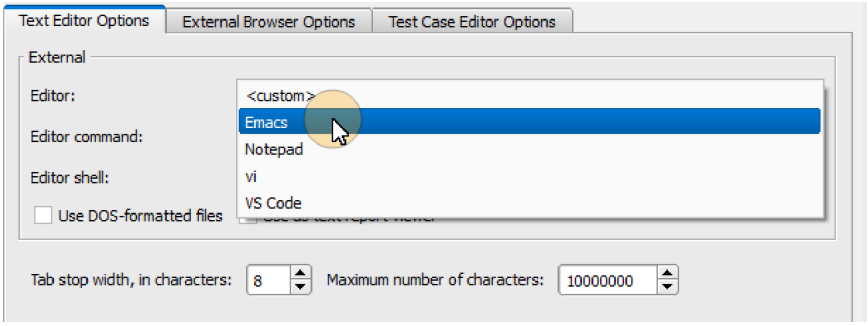
Note: These options are for text files and text reports only. To use an external browser for HTML reports, see "To Set Up an External Browser" on page 64.

The **Editor** drop-down list contains a list of predefined commonly used text editors which can be selected as a VectorCAST external editor for text files. Optionally, based on the setting of the "Use as

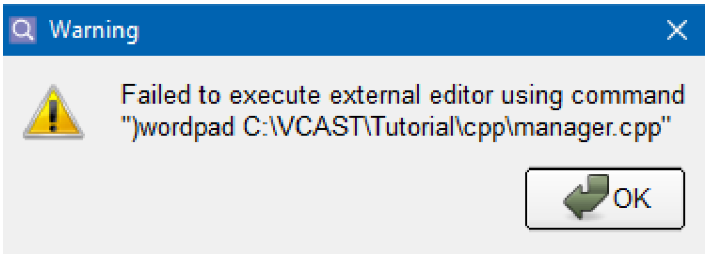
text report viewer" option, the external editor can be used to view text reports.

Text editor options include:

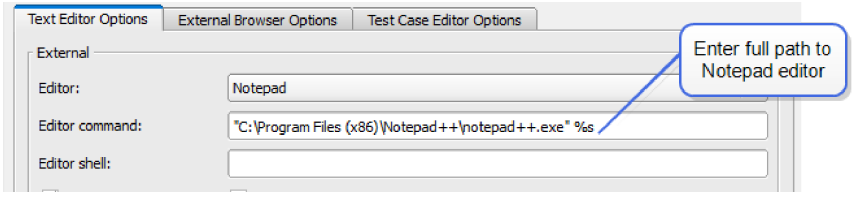
- <custom> (this option allows users to specify their own external editor)
- VS code (Visual Studio Code)
- Emacs
- Notepad
- vi



The executable file for the editor must exist on your PATH environment variable. In cases where the executable of the editor is not on your path, a warning appears:



In this case, you can either modify your PATH environment variable or configure VectorCAST with the full path to the editor executable using the Editor Command option.



Click **Use DOS Formatted Files** if your choice of editor requires that all lines be terminated with a CR + LF (i.e. Notepad).

If you want to use the same editor for viewing text reports, click the checkbox next to **Use as text report viewer**. If this option is set, and you have chosen “Text” as the report format in the Reports tab, then any time you choose a “View” action, the file chosen is opened in the external editor.

When you have made your selection(s), click **Apply** or click **OK** to close the dialog.

Set an External Editor to Scroll to First Line of Function

When an external editor is used to open source files, a script can be used to enable the editor to scroll to the line number of a selected function in a source file. The script uses the environment variable VCV_EDIT_LINE_NUMBER, which is set to the line number of the selected function. The external editor can then be set to a script command that uses the environment variable VCV_EDIT_LINE_NUMBER to go to a specific line number in the external editor.

When opening a file using an external editor in a context where no particular location is requested, such as having selected a unit rather than a subprogram, VCV_EDIT_LINE_NUMBER is set to 0, which opens the file and sets the cursor to the top.

Linux Example

On Linux, a vi user creates a bash script named **`my_start_editor.sh`**, whose contents include:

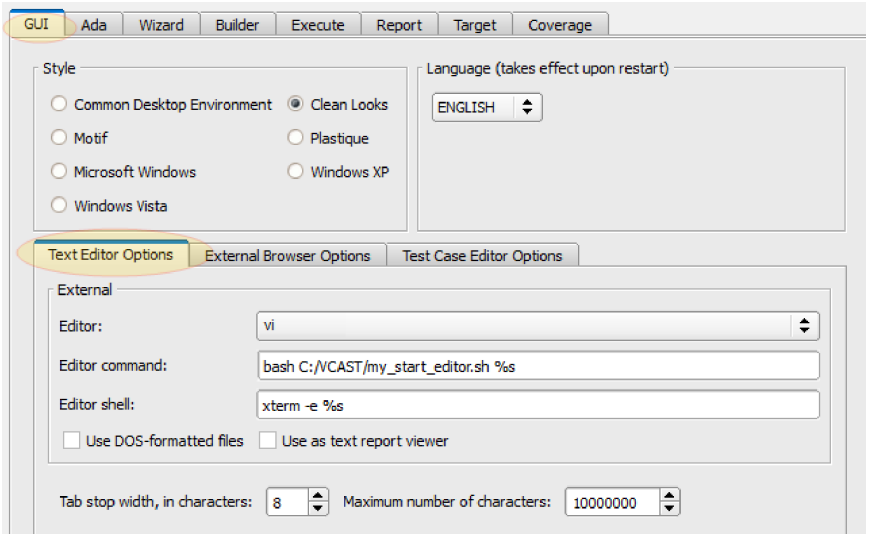
```
#!/bin/bash
vi "$@" +$VCV_EDIT_LINE_NUMBER
```

In VectorCAST's Options dialog, **GUI** tab, **Text Editor Options** sub-tab, the user sets the option **Editor command** to:

```
bash <path to>/my_start_editor.sh %s
```

and sets the option **Editor shell** to:

```
xterm -e %s
```



Then, in a VectorCAST unit test environment, when right-clicking on a UUT's subprogram name and choosing **Open Source**, **`xterm`** opens with the vi editor correctly scrolled to the first line of the selected subprogram.

Windows Example

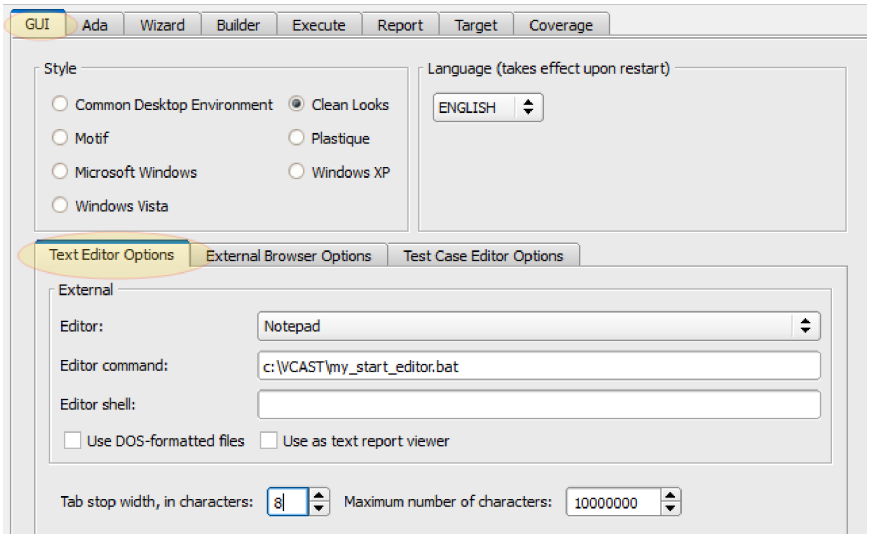
On Windows, a Notepad++ user creates a batch file named **`my _start_editor.bat`**, whose contents include:

```
"<path to>\notepad++.exe" -n%VCV_EDIT_LINE_NUMBER% %1
```

where **`<path to>`** is written similar to **`"c:\Program Files (x86)\Notepad++"`** and enclosed in quotes.

In VectorCAST's Options dialog, **GUI** tab, **Text Editor Options** sub-tab, the user sets the option **Editor command** to:

```
<path to>\my_start_editor.bat
```



Then, in a VectorCAST unit test environment, when right-clicking on a UUT's subprogram name and choosing **Open Source**, Notepad++ opens with source file correctly scrolled to the first line of the selected subprogram.

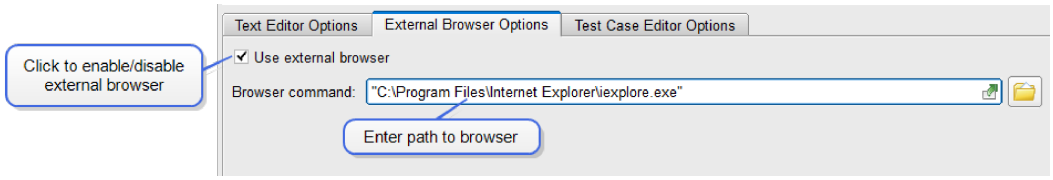
To Edit User Code with the External Editor

When you are editing user code (in the Configure Stubs dialog, the Test Case User Code tab, Parameter User Code dialog, or Environment User Code dialog), right-click in the text area and select **Invoke External Editor** from the popup menu. The external editor opens and you can edit user code there. When you save and exit the editor, the temporary file is imported into the user code text edit box.

To Set Up an External Browser

By default, all HTML reports opened in VectorCAST are displayed in the MDI window. If you want to use an alternative browser for HTML reports, you may configure this by selecting **Tools => Options** from the Help Menu, then selecting the **GUI** tab, **External Browser Options** sub-tab.

First, click the checkbox next to **Use External Browser** to enable the feature. Note that checking this box without specifying a browser causes VectorCAST to display all reports, including the test case Execution Results, in a tab in the Report group.



If you would like all reports to be displayed in a separate browser window, enter the path to your HTML browser in the edit box next to **Browser Command**. If none is specified, VectorCAST's internal browser is used. You can set up a browser but temporarily turn it off by un-checking the **Use External Browser** checkbox.

When you have made your selection, click **Apply** or **OK** to close the dialog.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.

To Toggle Gridlines in the Test Case Editor

Choose **Tools => Options**, and click the **GUI** tab and click the **Test Case Editor Options**sub- tab.

The **Show gridlines in test case** option enables you to view the Test Case Editor either with gridlines (if you prefer a spreadsheet look to the editor) or without gridlines. By default, the Test Case Editor displays with gridlines.

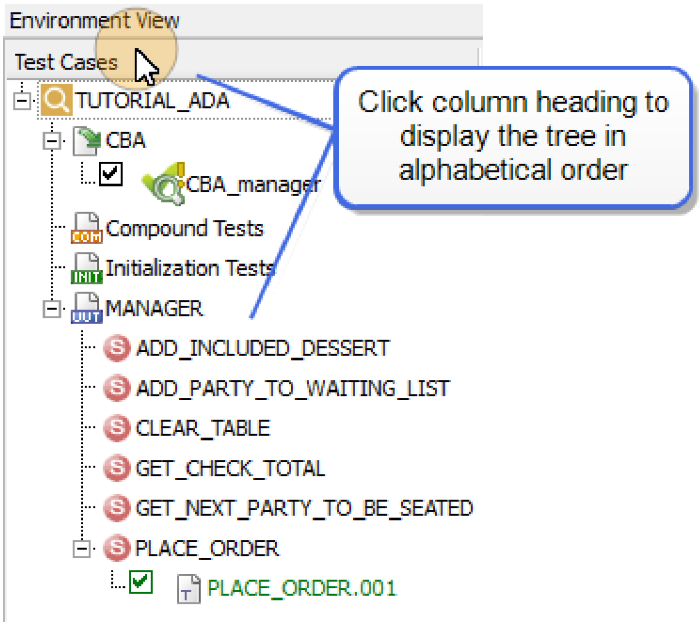
When you have made your selection, click **Apply** or **OK** to close the dialog.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.

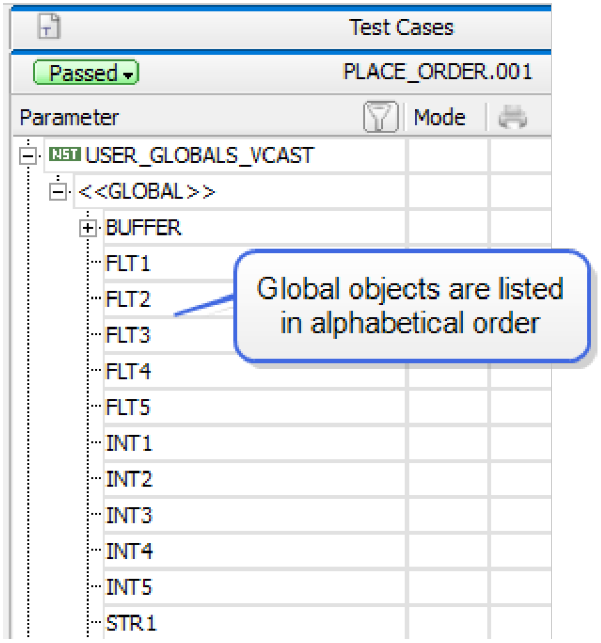
To Alphabetize Test Cases

Users can set the order in which the items in the Test Case Tree and in the Test Case Editor Parameter Tree are displayed.

In the Test Case Tree, clicking on the title bar of the Test Cases column will toggle the display from an alphabetical listing of Unit, Subprogram, and Test Case names to a listing based on the order of definition of those items. "Order of definition" means the order that subprograms are defined in your source file under test, or the order in which a test case is added to the environment.



To alphabetize in the Test Case Editor Parameter Tree, choose **Tools => Options**, and click the **GUI** tab and click the **Test Case Editor Options** sub-tab. Select the **Alphabetize parameters** option. When this option is set, the Unit names, Subprogram names, Global Object names and Parameter names are alphabetized in the Parameter Tree. When this option is cleared, the Parameter Tree items are displayed in the order they are defined in the source code.

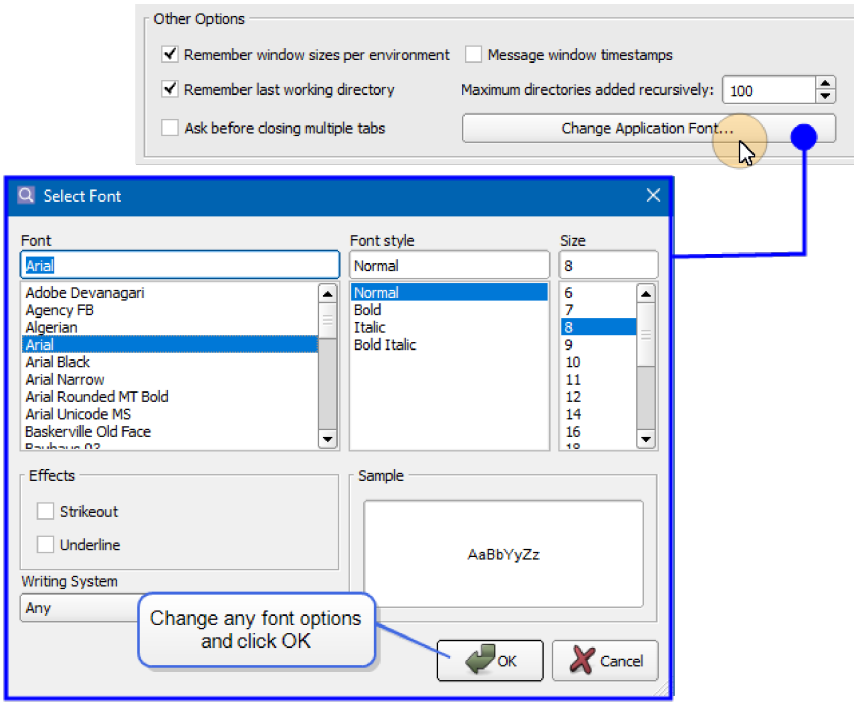


Note: If you turn this option on or off, you must close and re-open any existing Test Case Editor windows for the change to take effect.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.

To Change the Main Window's Font

To change the font used by the VectorCAST application, choose**Tools => Options**, and click the **GUI** tab. Click the **Change Application Font** button in the Other Options panel. The Font dialog opens. Select from any of the provided font options. Click **OK** in the font dialog.



Click either **OK** or **Apply** in the **Tools => Options** dialog, and the new font appears.

To Automatically Save Test Cases Before Execution

Choose **Tools => Options**, and click the GUI tab.

The "Automatically Save Test Cases Before Execution" option saves the current test case prior to execution. If this option is not selected, you are prompted to save a modified test case when execute is chosen. By default, modified test cases are automatically saved before executing.

When you have made your selection, click **Apply** or **OK** to close the dialog.

This information is saved to a file named `.vcast-qt` in the user's `HOME` directory.

To Turn on a Reminder Before Closing Multiple Tabs

Choose **Tools => Options**, and click the GUI tab.

If you would like VectorCAST to remind you that more than one tab is being closed, turn on the "Ask before closing multiple tabs" option. This option is off by default, meaning that when you click the close X of a tag group, all the tabs close without a reminder.

When you have made your selection, click **Apply** or **OK** to close the dialog.

Customizing Reports

VectorCAST's reporting system allows reports to be customized to contain any information with any layout. Reports can be produced as either HTML or text. To generate a PDF of a report, use the HTML version.

Note that the full data API into all statistics is only available with the VectorCAST/Ada Enterprise edition.

Customizing Existing Reports

VectorCAST supports customizing existing reports. The following sections discuss how to customize sections of reports (add, remove, replace, and reorder sections), and also how to configure VectorCAST to use custom templates.

The option VCAST_RPTS_CUSTOM_CONFIG is used to customize the sections used in reports and to replace the contents of a template wihout needing a customization directory. The option can point to a file containing the configuration or the JSON configuration itself.

```
clicast -lada Option VCAST_RPTS_CUSTOM_CONFIG <config text> | <config file>

Custom report configuration. If this option points to an existing file,
then the contents of that file is used as the custom report configuration,
otherwise the contents of the option is used.
```

Adding and Removing Report Sections

To add or remove a section:

```
{
  "reports": {
    "<report_name|*>" : {
       "<HTML|TEXT>": {
        "<sections|extra_sections|remove_sections":[<data>]
      }
    }
  }
}
```

Where:

report_name - the report to customize, or all reports (*****)

HTML|TEXT - report format to apply the sections to

sections - specify a list of sections for the report

extra_sections - allows the addition of sections without having to list out all of the sections. Use the option '**+**' to add the new section after the existing section. Use the option '**-**' to add the new section before the existing section.

remove_sections - remove one or more sections.

The example below shows adding the Overall Results and Metrics tables after the Configuration Data in the Execution Report:

```
{
  "reports": {
    "ExecutionResultsReport" : {
       "HTML": {
        "extra_sections":
          "[["CONFIG_DATA", "+OVERALL_RESULTS"],{"OVERALL_
RESULTS","+METRICS"]]
      }
    }
  }
}
```

The following example shows removing the Metrics Table from the Full Report:

```
{
  "reports": {
    "FullReport" : {
       "HTML": {
        "remove_sections": ["METRICS"]
      }
    }
  }
}
```

Specifying Report Sections to Replace

To replace a section:

```
{
  "sections": {
    "<section_name>" : {
      "<HTML|TEXT_FILE|TEXT>":"<path to file|text>"
    }
  }
}
```

Where:

`section_name` - the name of the section

`HTML | TEXT_FILE | TEXT` - specifies whether the following string is the actual HTML or text template or a file name containing the template.

The following example shows replacing the Metrics Table with a text file:

```
{
  "sections": {
    "METRICS" : {
      "TEXT_FILE": "C:\template.txt"
     }
```

```
   }
}
```

Specifying Report Section Order

To specify the order of the sections:

```
{
  "reports": {
    "<report_name |*>" : {
      "<HTML | TEXT>" : {
        "<sections | extra_sections | remove_sections": [<data>]
      }
    }
  }
}
```

Where:

`report_name` - the report to customise, or all reports (**`*`**)

`HTML | TEXT` - report format to apply to the sections

`sections` - specifies a list of sections for the report

`extra_sections` - allows the addition of sections without having to list out all of the sections.

For example,
`[ 'EXISTING_SECTION', '<option>NEW_SECTION' ]`
– where **`option`** is '**`+`**' or '**`-`**' to add the new section after or before the existing section

`remove_sections` - remove one or more sections

The example below shows adding a "new section" after Configuration Data and "another section" before the Metrics Table to all reports:

```
{
  "reports": {
    "*" : {
      "TEXT" : {
        "extra_sections" : [["CONFIG_DATA", "+NEW_SECTION"], ["METRICS", "-
ANOTHER_SECTION"]]
      }
    }
  }
}
```

Customizing Templates

All report templates reside in the product installation directory located at:
`%VECTORCAST_DIR%\python\vector\apps\ReportBuilder\templates\*`

Templates can be customized. It is important to note the following:

- Any template changes will apply to ALL VectorCAST projects.
- It is possible that you may not have permission to modify the installation directory. This means that each user who wants to run reports must modify the files.

Best practice is to use a Configuration Directory to override the default templates. If a template exists in the Configuration Directory, it will be used over the template located in the installation directory.

The Configuration Directory can also be the location to create new reports.

To create a new Configuration Directory:

```
%VECTORCAST_DIR%\clicast REPORTS EXTRACT <new_dir>

This command copies all the templates and adds a .orig extension.
```

Configuring VectorCAST to Use Custom Templates

To configure VectorCAST to use the Custom Templates in the Configuration Directory, use the following command:

```
clicast -lada Option VCAST_RPTS_USER_REPORTS_DIR <directory>
```

Template files are located in the **templates\<section_name>** directory. All template files have a **.orig** extension, so by default, they are not used. To use the template in the custom report directory, you must remove the **.orig** extension.

Template files use a full-featured template engine for Python called Jinja (http://jinja.pocoo.org/), which allows:

- Easy decoupling of layout from data,
- Easy modification of report structure, and the
- Ability to modify the Jinja code to change appearance.

Customizing Report Sections

VectorCAST supports adding customized headers and footers. By default, every report includes a custom header and a custom footer, but they are empty. You must define a custom header or a custom footer for them to appear.

To accomplish this, users need to configure VectorCAST to use custom templates. The following steps are used to set up the configuration:

1. First, to override the default report templates, it is recommended that you create a configuration directory to store and create new reports. For our example, we are creating a new configuration directory named **CustomDirectory**. This will copy all the default report template directories into our new directory and add a **.orig** extension to each file.

```
clicast REPORTS EXTRACT CustomDirectory
```

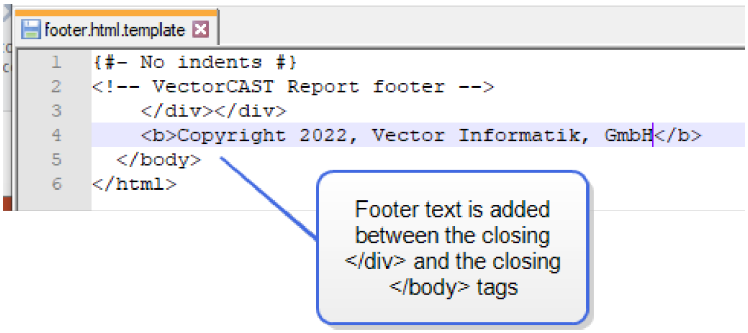
2. Next, configure VectorCAST to use the new configuration directory, **CustomDirectory**.

```
clicast -lada Option VCAST_RPTS_USER_REPORTS_DIR
C:\VCAST\examples\environments\enterprise_testing_demo\CustomDirectory
```

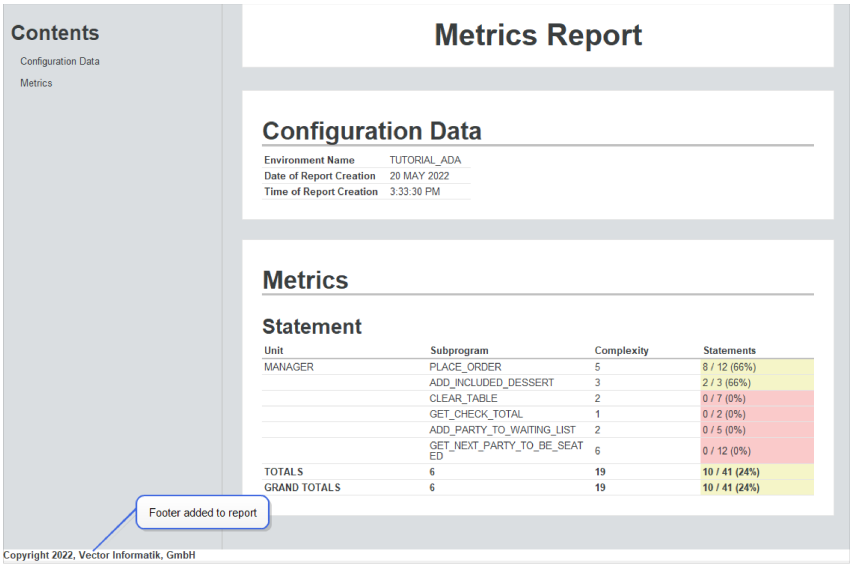
To customize reports, locate the report template in the configuration directory, remove the **.orig** extension and modify the file as needed.

Add a Custom Footer

To add a custom footer to the end of all reports, you will customize the **footer.html.template** file located in your configuration directory under **<configuration directory>\templates**. Open the file in your editor and add the desired footer text after the closing tag, and before the closing </body> tag, as shown below.



Save the changes and generate a report. The footer is displayed at the bottom of the report.



Add Your Logo to Reports

To add your custom logo to reports, you will customize the **main.html.template** file, located in

your configuration directory under **`<configuration directory>\templates\ReportTitle`**.

Open the file in your editor and override the default VectorCAST logo with your own Base64 encoded image.


main.html.template
<!-- ReportTitle -->

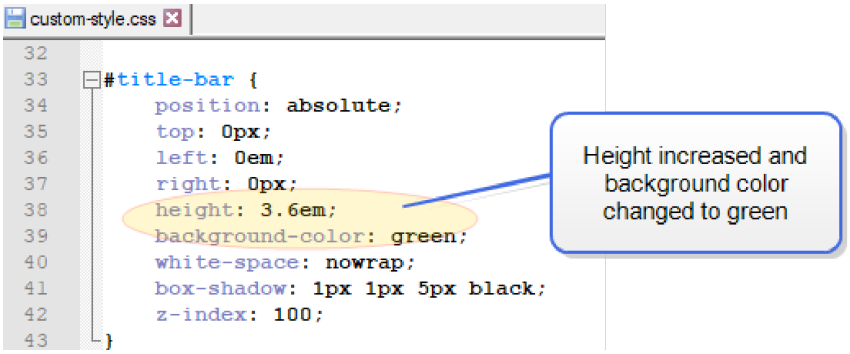
{%- if report.is_self_contained %}
<img alt="Vector"
src=" WxkIDApICAtLT4NCjwhRE9DVFlQRSBzdmcgUFVCTE1DICItLy9XM0MvL0RURCBTVkcgMS4xLy9FTiIgImh0dHA6Ly93d3cudzMub3JnL0dyYXBoaWNzL1NWRy8xLjEvRFREL3N2ZzExLmR0ZCI +DQo8c3ZnIHZlcnNpb249IjEuMSIgaWQ9IldfU1QtX3gyR19CSUxETUFSS0VfclJNQiIgeG1sbnM9Imh0dHA6Ly93d3cudzMub3JnLzIwMDAvc3ZnI iB4bWxuczp4bGluaz0iaHR0cDovL3d3dy53My5vcmcvMTk5OS94bGluayINCgkgeD0iMB4Ii B5PSIwcHgiIHdpZHRoPSIOMTUuNTc5cHgiIGhlaWd odD0iMTk3LjU3MnB4IiB2aWV3Qm94PSIwIDAgNDE1LjE3OSAxOTcuNTcyIiBlbmFibGU tYmFja2dyb3VuZD0ibmV3IDAgMCA0MTUuNTc5IDE5Ny4iN zIiDQoJIHhtbDpzcGFjZT0icHJlc2VydmUiPg0KPHBvbHlnb24gaWQ9IkpbGRtYXJrZV8xODNfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzBfeDJGXzUwIiBmaWxsPSIjQjcwMDMyIiB wb2ludHM9IjI5Mi4yODgsMTElLjI1IDI5Mi4yODgsMTMxLjcxNSAzNDkuMzIyLDk4Ljc4NiAyOTIuMjg4LDgyLjMyM SAzMjAuODAlLDk4Ljc4NiAiLz4NCjxwb2x5Z29uIHBvaW50cz0iOTAuNTU0LDQ0LjUyOSA5OC43OcsODQuNTI5IDgyLjMyMiwxMTMuMDQ3IDY1LjgvLjMyM lNyw4NC4iMjkgNzQuMDg5LDg0LjUyOSA4Mi4zMjIsOTguNzg4ICIvPg0KPHBhdGggZD0iTTIxNy43NTQsODMuODE5Yy04Ljk2NC0wLjE0LjI2OSwwLjA3IDY1LjgvLjMyM jcwMy0xNC45NzIsMTQuOTcyYzAsOC4yNjksNi43MDMsMTQuOTcxLDE0Ljk3MiwxNC45NzEsMTQuOTcyLTE 0Ljk3MUMyMzIuNzI2LDkwLjQwMzUuLDMuNTI3LTcuODc3LDcuODc4LTcuODc3LDcuOTgzYy04Ljc1QtMzgzYy04LjI2OSwwLTE0Ljk3MiwzNjQuNDQzL 2LTUuMDE2QzE1OS4yNjgsMTAlLjc4NiwxNTcuMjk3LDEwNi42NjgsMTU1LjEyMSwxMDYuNjY4ei IvPg0KPHBvbHlnb24gcG9pbnRzPSIxNzEuNDQzL DkwLjk0NiAxODEuMDcxLDkwLjk0NiAxODEuMDcxLDExMy4wNDcgMTg4LjIwMSwxMTMuMDQ3IDE4OC4yMDEsOTAuOTQ2IDE5Ny44MjUsOTAuOTQ2IDE 5Ny44MjUsODQuNTI5IAOKCTE3MS40NDYsODQuNTI5ICIvPg0KPHBvbHlnb24gcG9pbnRzPSIxMDguMDY5LDExMy4wNDcgMTMwLjE3LDExMy4wNDcgM TMwLjE3LDE4My4wMTUuMTk4LDEwNi42MyAxMTUuMTk4LDEwMS42MyAxMTUuMTk4LDEwMS42MzAgMTA2LjA2OSw4NC4wMTUuMTk4LDk2 1LjM2N iAxMTUuMTk4LDkwLjk0NiAxMzAuMTcsOTAuOTQ2IDEzMC4xNyw4NC4iMjkgMTA4LjA2OSw4NC4iMjkgMTA4LjA2OSw4NC4iMjkgMTA4LjA2OSw4 ywsNC4wNDdjMC01LjI1LTQuMjU1LTkuNTE4LTkuNTE4LTkuNTE4aC0xNy41ODd2MjguNTE4aDcuMTI5di0zLjQ4NmwtNS44OTktMTAuMjMxQzE3 4LjI1MWwtNS44OTktMTAuMjMxQC0xQzI3Ny43NDgsMTAxLjM5NSwyNzAuMTg3LDk4LjAyLDI3MC4xODcsOTQuMDQ3eiBNMjYwLjAyMywyNi4yMzg 1LjM2N iAxMTUuMTk4LDkwLjk0NiAxMzAuMTcsOTAuOTQ2IDEzMC4xNyw4NC4iMjkgMTA4LjA2OSw4NC4iMjkgMTA4LjA2OSw4NC4iMjkgMTA4LjA2OSw4

Your custom style sheet is appended when reports are generated and the contents of the custom style sheet are embedded in the **<style>** section following the default styles, as shown in the example below:

```
<!DOCTYPE html>
<!-- VectorCAST Report header -->
<html lang="en">
  <head>
    <title>Report</title>
    <meta charset="utf-8"/>
    <style>
       default styles...
       user's custom styles...
    </style>
  </head>
...
```

Any user defined styles will follow standard cascading style sheets rules and will correctly override the defaults, meaning that only the specific attributes of a style must be specified.

In the example below, we are modifying our custom style sheet to change the report header background color to green and make it twice as wide.

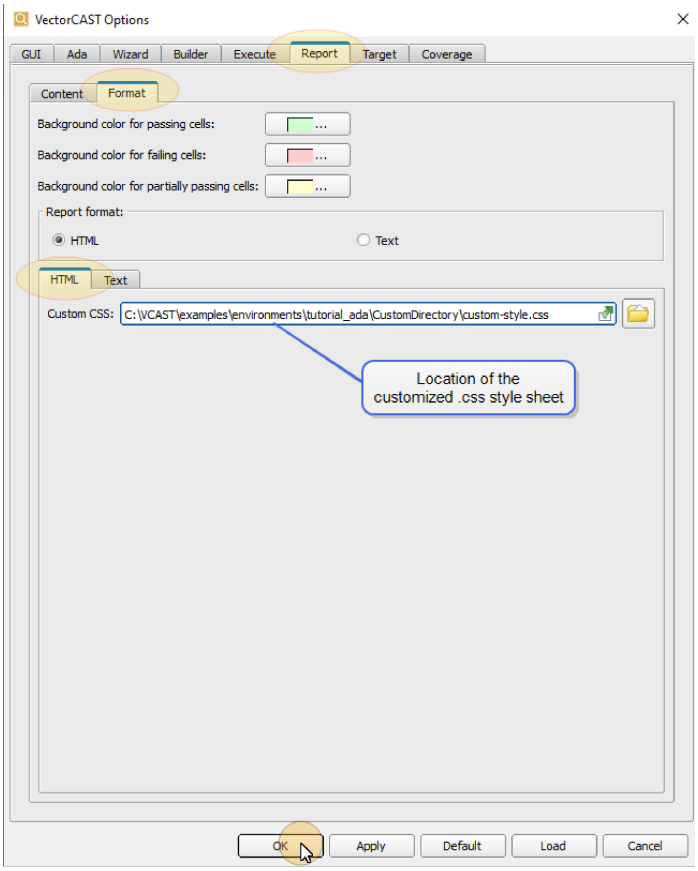


Next we add our custom style sheet to VectorCAST.

Adding a Custom Style Sheet

From the GUI:

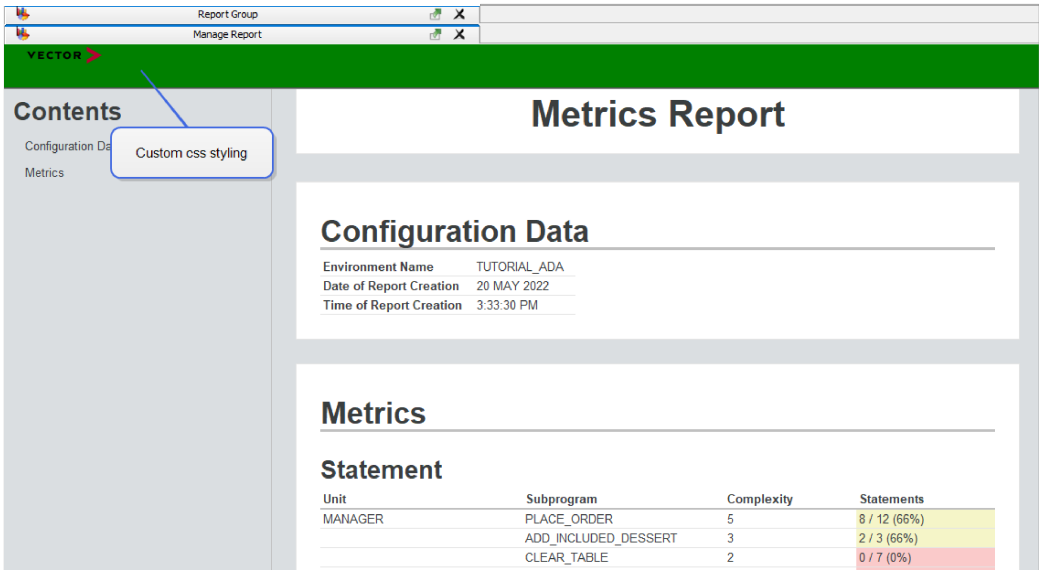
To add a custom style sheet, Choose **Tools => Options** and click the **Report** tab => **Format** sub-tab => **HTML** sub-tab. Use the browser button to select the location of your custom style sheet. Environment variables in the path to the custom style sheet are supported, using the syntax **$(ENV_VAR)**. Click **OK** to apply the custom style sheet.



From the command line:

```
clicast -lada option VCAST_RPTS_CUSTOM_CSS <path to custom.css file>

Reports use the default cascading style sheets (*.css) located in
$VECTORCAST_DIR/python/vector/apps/ReportBuilder/css/, unless a custom
style sheet is specified in this option. Environment variables in the path
to the custom style sheet are supported, using the syntax $(ENV_VAR). When
used, the contents of the custom style sheet are embedded in each report,
between the <style>...</style> tags in the header section after the default
styles, allowing for CSS style overriding.
```

Once the custom style sheet is added, it is applied when generating a report. In the example below, we see our custom-style.css is applied with its modifications to the title bar.



Embedding Style Sheets

VectorCAST provides the clicast-only option VCAST_RPTS_SELF_CONTAINED, which controls how reports are generated. By default, the option is set to True. This means that each HTML report is created as a single, self-contained file with an embedded style sheet and embedded images (for all built-in, non user-modified reports). This is useful when you are planning to email reports.

Embedding can cause problems with some web servers. To address this, VectorCAST provides the ability to set the option to False. When the option is set to False, the styles contained in the default style sheet (and any custom style sheet) are generated into a local style file referenced by the report. Style sheets and images are stored in the same directory as the report, which is useful when serving reports from a web server.

```
clicast -lada option VCAST_RPTS_SELF_CONTAINED True | False

When True, HTML reports are created as a single, self-contained file with
an embedded stylesheet and embedded images (for all built-in, non user-
modified reports). Set the option to True (default value) when planning to
email reports. When False, HTML reports are created as multiple files,
storing stylesheets and images in the same directory as the report. Set the
option to False when serving reports from a webserver.
```

Creating New Reports

VectorCAST uses a cascading style sheet (CSS) to control the appearance of reports. The default CSS is located at:
%VECTORCASTDIR%\python\vector\apps\ReportBuilder\css\default-style.css

Report Components

There are three components to writing a new report: a main report file, a section file, and a template

layout file. The table below lists the files used to create an example of a new report, named My Report, along with the location of each file and its purpose. Each component is discussed in more detail in the following sections.

File Name	Location	Purpose
my_report.py	custom_rep\reports	The 'main' report file. This defines the report name and overall sections. In this report there are some built in sections and a new section called EXAMPLE_SECTION. The file and class name should match up, so the file is my_ report.py with a class called MyReport. The report name is 'my_report'.
example_ sections.py	custom_rep\sections	Uses the data API to extract data from VectorCAST and put it into a data structure so it can be formatted into the report file. The file/class/section names are all related: example_sections.py contains an ExampleSections class to implement the EXAMPLE_SECTIONS section. The prepare_ data function is called and this adds data to the self.section_context object. In this case we use all the test cases and iterate over them, getting the result.
main.html.template	custom_ rep\templates\ExampleSection	Contains the presentation format for the section. Note that this file has a fixed name, but is in a directory corresponding with the section name. This is a Jinja template which allows you to control the HTML formatting of the Python data. This example just adds the report name and then a simple table of the test case results, coloured using built in styles.

Report File

The report file, **`my_report.py`**, is the main entry point for the report. This file sets up any initial data, defines the report name, and describes the sections of the report and the order in which they appear.

```
import os
from vector.apps.ReportBuilder.custom_report import CustomReport

class MyReport (CustomReport):
    default_sections = ['CUSTOM_HEADER', 'REPORT_TITLE', 'TABLE_OF_
CONTENTS', 'EXAMPLE_SECTION', 'METRICS', 'CUSTOM_FOOTER']

    def initialize(self, **kwargs);
       pass
```

Section File

The section file, **example_sections.py**, contains the data that goes into the report. Report data is typically extracted using the Manage Data API (although it is possible to retrieve the data from elsewhere).

```
from __future__ import absolute_import
from vector.apps.ReportBuilder.report_section import ReportSection
from vector.apps.DataAPI.api import Api

class ExampleSection(ReportSection):
# Define the title (this shows in the config data and the Table of
Contents)
    title = 'Results Section'
# Sections can be conditional on type of environment - in this case, this
will be used for both
supported_environments = ('COVER', 'UNIT')

# prepare_data is called to setup any data required for generating the
report
       def prepare_data(self):
           self.section_context['rep']="Execution Results Summary"
           self.section_context["results"]=[]
           api=Api('.')

           a=api.TestCase.all()
           for test in a:
                result={}
                result["name"]=test.name
                result["verdict"]="PASS" if test.passed else "FAIL"
                self.section_context["results"].append(result)
```

Layout Template File

The layout template file, **main.html.template**, is a Jinja template that describes the layout of the data. Note that:

- Elements in double curly braces **{{}}** are Python data objects from the section file
- Can also contain regular HTML
  - But do not include styles. Presentation should be placed in the CSS file.

```
{# This is a comment! My Report template #}
<h1>{{rep}}</h1>

<table style="width:100%">
    {%- for result in results %}
       {%- if result.verdict == "PASS" %}
            <tr class="success">
       {%- else %}
            <tr class="danger">
       {%- endif %}
       <td>{{result.name}}</td>
       <td>{{result.verdict}}</td>
```

```
        </tr>
    {%- endfor %}
</table>
```

Running New Reports

From the command line:

```
clicast -e <environment> report user <report_name>
```

For example:

```
clicast -e my_env report user my_report
```

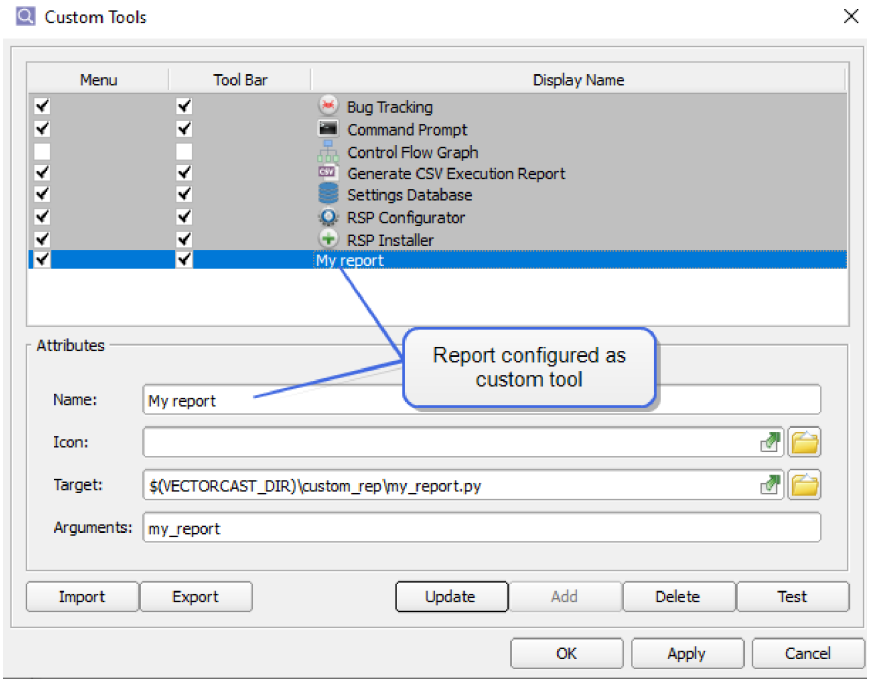
From the GUI:

To generate new reports from the GUI, the report can be configured as a custom tool which will run the clicast command.

Create a helper script which will do the following:

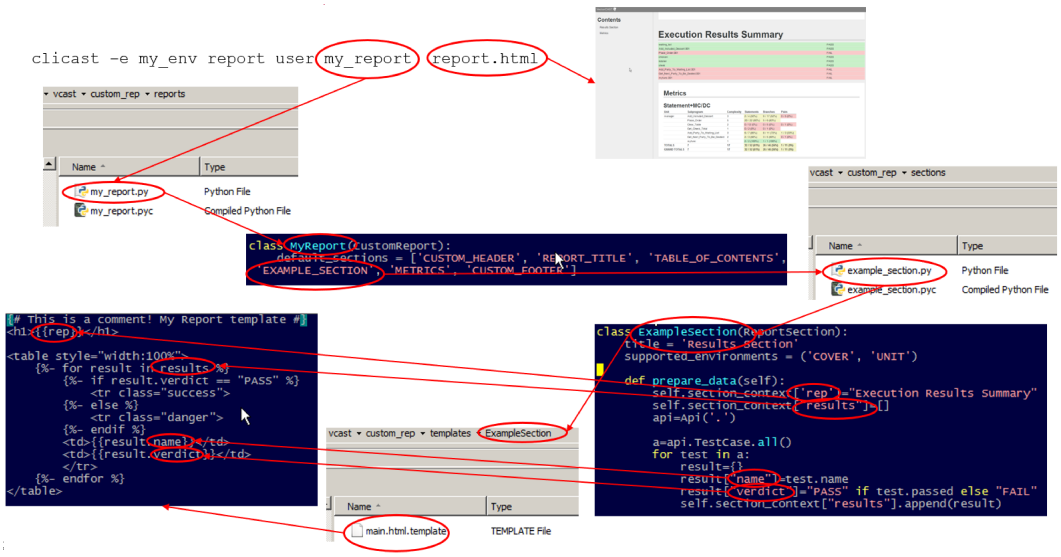
- Change up a directory out of the environment directory
- Remove any previous stale report file
- Run the report
- Open the report in the GUI

The script can accept any report name, so it can be used for any customised report.



The following figure illustrates how the components used to create a custom report work together to

produce the final custom HTML report.



Editing, Searching, and Printing

To Create a New Text File

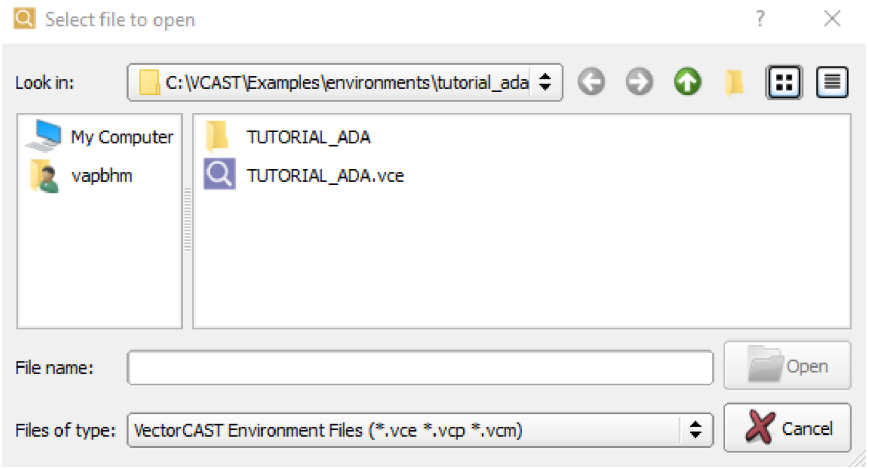
Use **File => New => New Document** to create a new text file. The keyboard shortcut is Ctrl+T. When selected, a new, untitled text window opens in a Text Editor in the MDI window. The file is set as writeable by default.

If you have specified an external editor on the **Tools => Options** dialog, GUI tab, then the text file opens in that editor.

To Open a Script or Report File

File => Open... enables you to open an existing environment. A standard open file dialog box is used. By default, the file extension "***.vce *.vcp *.vcm**" is selected as the file type.

The command may also be launched by clicking on the Open button  in the toolbar.



By changing the "Files of Type" selection, you can also use **File => Open...** to open other kinds of files such as source files. Files are opened in a Text Editor or the external editor, if specified.

To open a(n)	Choose Files of Type
Unit test environment	Environment Files (***.vce** *.vcp *.vcm)
Coverage environment	Environment Files (*.vce ***.vcp** *.vcm)
Manage project	Environment Files (*.vce *.vcp ***.vcm**)
Test case script	Script Files (*.scr, *.env, *.**tst**)
Environment script	Script Files (*.scr, *.**env**, *.tst)
Windshell script	Script Files (***.scr**, *.env, *.tst)
C or C++ source code file	C/C++ Source Files (*.cpp, *.cc, *.cxx, *.c,

To open a(n)	Choose Files of Type
	*.c++)
C or C++ header file	C/C++ Header Files (*.h, *.hxx, *.h++)
Ada Source code files	Ada Source Files (*.adb, *.ada, *.ads)
Configuration file	VectorCAST Config Files (*.CFG)
HTML or Text file	Report Files (*.html, *.txt)
CSV file	CSV Map files (*.csv, *.tab)
a file of any type	All Files (*)

To Save a Script or Text File

File => Save enables you to save the contents of the current active text document in the Text Editor.

You can save a changed test case if it is in the MDI window. If a source code file, test script, or configuration file is currently open in the MDI window, then it is saved. If a results file report, data file, or anything listed in the **Environment => View** or **Test => View** menu is saved, then the Save As dialog opens, allowing you to give the file a name.

The command may also be invoked by clicking on the **Save** button on the Toolbar  .

To Save the Open Window As

File => Save As... enables you to save the text or HTML data in the current open window to a filename of your choice. Choose **File => Save As...** from the Menu Bar. You are prompted for the name of the file where the extracted data should be stored.

The command may also be invoked by clicking on the **Save All** button  in the toolbar.

To Save All Open Windows

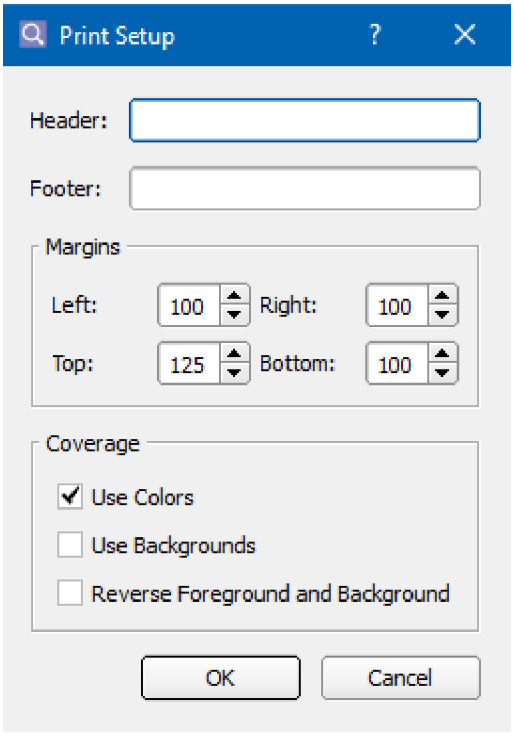
File => Save All applies the Save command to all unsaved, visible windows. For windows in the Test Case Editor, it saves unsaved parameter data. For reports and scripts in the Text Editor, it saves the information to a file. If more than one unsaved text file is open, you are prompted to save each one, starting with the top-most file.

The command may also be invoked by clicking on the **Save All** button  in the toolbar. When you use the **Save All** button, each tabbed window is saved. If you have several different windows open and tiled or cascaded, then each tab in each visible window is saved. In cases where a filename is required, you are prompted once for each tabbed window that needs a filename.

Print Setup

File => Print Setup... enables you to customize your printing options, such as headers, footers and

margins.



Header and Footer: Add a header and/or footer text string to your printed report.

Margins: Customize the size of the margins in the report output. The margins options are: Left, Right, Top and Bottom. You can enter the desired margin value for each option. You can also use the spin arrows to select the margin values.

Use Colors: Print coverage reports with the colors chosen in **Tools => Options, Coverage** tab, **Coverage Viewer** sub-tab. Colors can be chosen for covered lines, partially covered lines, uncovered lines, and uncoverable lines.

Use Backgrounds: Print the background color specified in **Tools => Options dialog, Coverage tab, Coverage Viewer sub-tab**. The default is to print using the foreground color only. Printing with background colors enabled can use a lot of ink or toner.

Reverse Foreground and Background: Reverse foreground and background colors when printing coverage reports. It is useful when the selected coverage foreground colors are light and the background colors are dark, saving on ink or toner.

To Print an Open Window

This command enables you to print out the contents of any text file in a Text Editor or Coverage Viewer. The **File => Print...** command applies to reports, results, scripts, and coverage information.

Select **File => Print...** or click on the **Print** button on the toolbar  .

To Preview Before Printing

The **File => Print Preview…** command enables you to preview any print job before sending it to the printer. Selecting **Print Preview...** will open a separate Print Preview window.

This command can also be accessed by clicking the arrow on the right of the Print button  on the Toolbar and then selecting **Print Preview…** from the drop-down menu.

To Undo a Recent Change

Edit => Undo applies to entered text and enables you to undo the last text stored in the text buffer.

This command can also be accessed by pressing **Ctrl+Z**.

To Redo

Edit => Redo reverses the action of the **Undo** button. If you select **Undo** by mistake, **Redo** will correct the error.

This command can also be accessed by pressing **Ctrl+Y**.

To Copy, Cut, and Paste

You can copy sections of text by copying and pasting. This means you can select text from one location and insert it into another place. To copy and paste do the following:

1. Select the text you want to copy.
2. Perform one of the following actions to save a copy of the text to the clipboard:
   - Choose **Edit => Copy** from the Menu Bar,
   - Press **Ctrl+C**,
   - Or right-click and choose **Copy** from the context menu.
3. Position the cursor where you wish to insert the text.
4. Perform one of the following actions to insert the text :
   - Choose **Edit => Paste** from the Menu Bar,
   - Press **Ctrl+V**,
   - Or right-click and choose **Paste** from the context menu.
5. To undo the paste, choose **Edit => Undo** from the Menu Bar.

You can also move text using cutting and pasting. This means you can remove text from one place and move it to another. To cut and paste do the following:

1. Select the text you want to move.
2. Choose **Edit => Cut** from the Menu Bar.
3. Position the cursor where you wish to insert the text.
4. Choose **Edit => Paste** from the Menu Bar to insert the text.

To Search for Text Using the Find Banner

The Find Banner is used to search for text within the following areas:

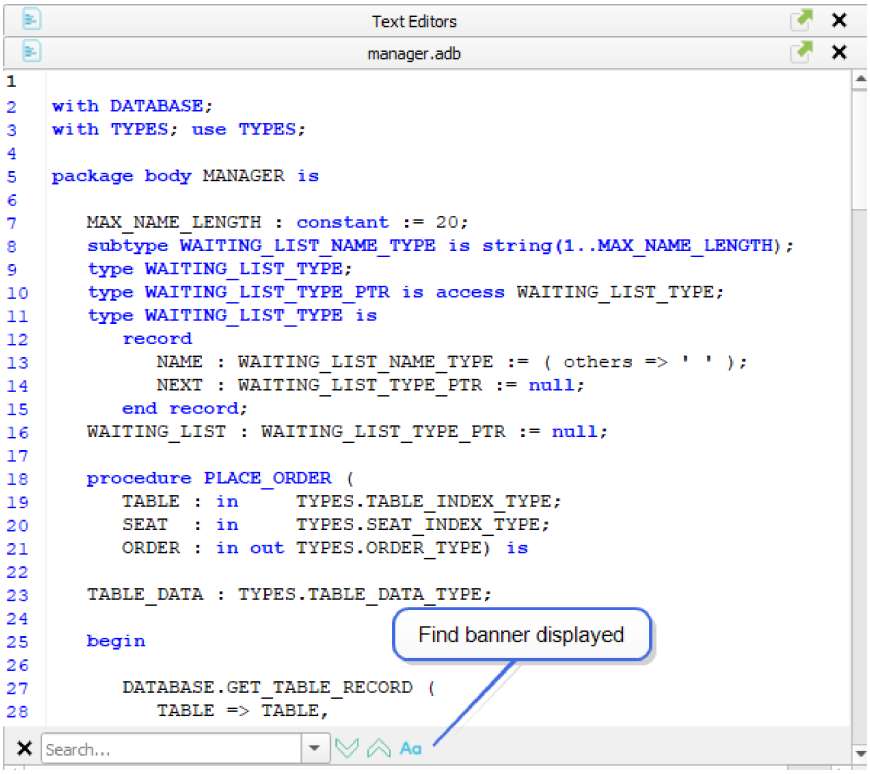
- Text Editor
- User Code Editor
- Coverage Viewer
- Parameter Tree
- Test Case Tree
- Report Viewer

The Find Banner appears at the bottom of the window, and is off by default.

Open The Find Banner

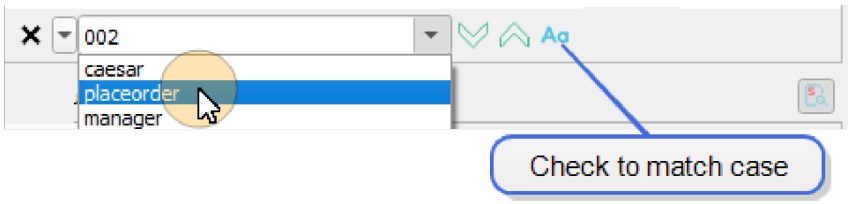
To open the Find Banner and search for text, follow these steps:

1. Open a file in an Editor, a report in a Viewer, or open a Parameter or Test Case Tree.
2. Click the mouse within the window where you wish to perform the search.
3. Choose **Edit => Find...** from the Menu Bar (or press **Ctrl+F**) and a Find Banner appears at the bottom of the Text Editor.

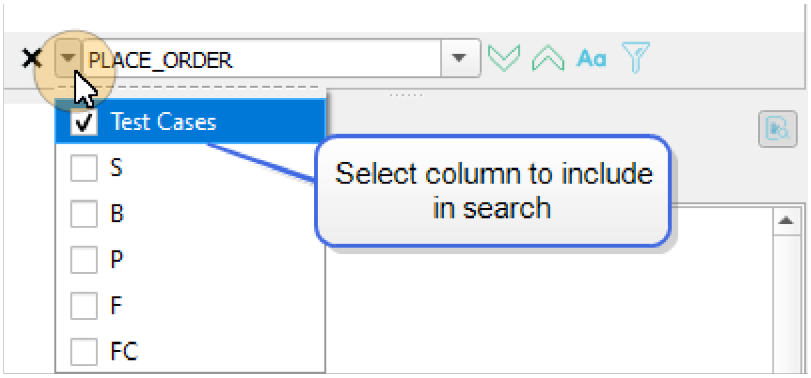


Execute a Search

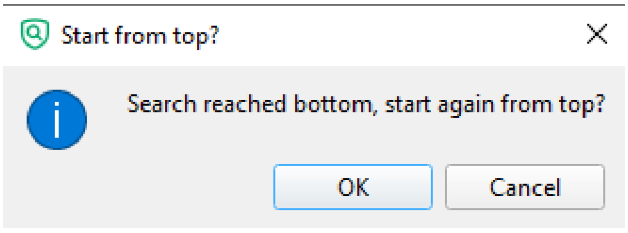
1. Type the text you are looking for in the **Find** text edit box. To choose text you have typed before, click the drop-down arrow, and select text from the list.



2. To specify that the case is matched exactly as you typed it, click the **Match Case** button.
3. When executing a search in the Test Case Tree, you must first select the columns to be included in the search. Click the drop-down arrow to the left of the search entry field, and from the menu, select the column names.

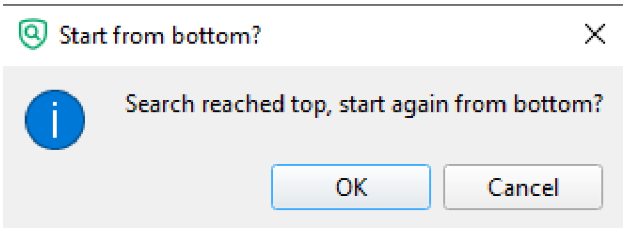


4. Press **Enter** to execute the search. All instances of the search string are highlighted, making it easy to locate search results.
5. To search for the next instance of the text, click the **Down** arrow.
   > In the Coverage Viewer, Report Viewer and User Code Editor, if the search text between the current line and the end of the file is not found, you are prompted to search from the top. Click **OK** to continue searching, or **Cancel** to quit the search.



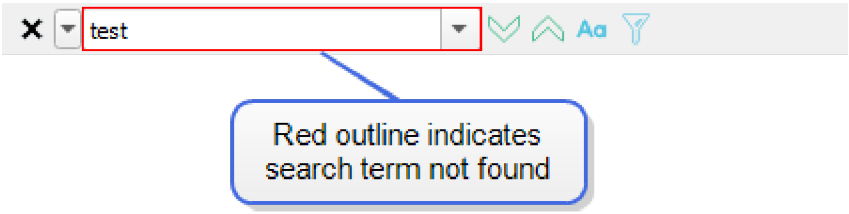
   > In the Text Editor, Parameter Tree and Test Case Tree, the search will continue to cycle through from the beginning.
6. To search for the previous instance of the text, click the **Up** arrow.

> In the Coverage Viewer, Report Viewer and User Code Editor, if the search text between the current line and the top of the file is not found, you are prompted to search from the bottom. Click **OK** to continue searching, or **Cancel** to quit the search.

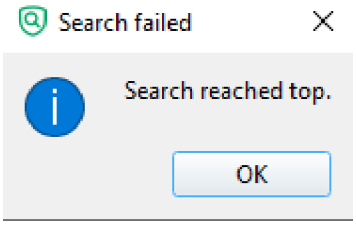


> In the Text Editor, Parameter Tree and Test Case Tree, the search will continue to cycle through from the bottom.

7. In the Parameter Tree, Text Editor, and Test Case Tree, if the text is not found, the text entry field displays a red outline.



8. In the Coverage Viewer, if the text is not found you are notified that the search failed.

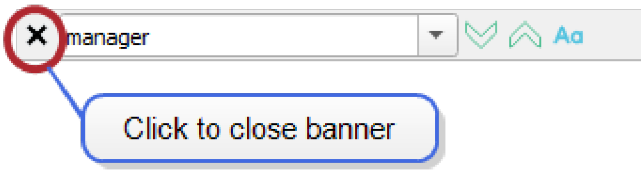


Repeat a Search

To search again without using the Find dialog, choose **Edit => Find Again** from the Menu Bar, or press the **F3** key.

Close the Find Banner

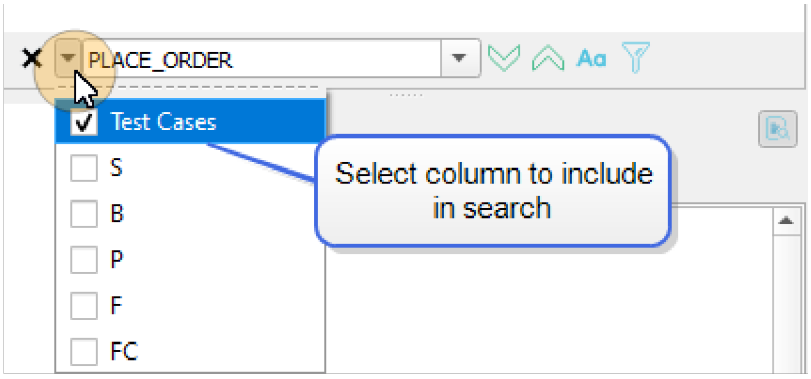
To dismiss the Find Banner, click the **Close** button located on the banner.



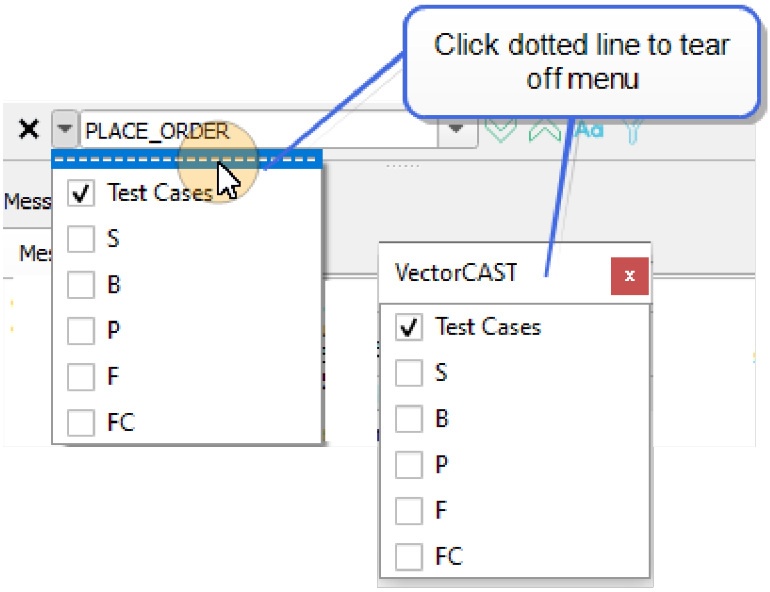
To Apply a Search Filter to the Test Case Tree

A search filter can be applied to the Test Case Tree. Filters are useful for pruning the tree and showing only those items of current interest. A filter is applied to all nodes in the tree, regardless of whether they are in an expanded part of the tree or not.

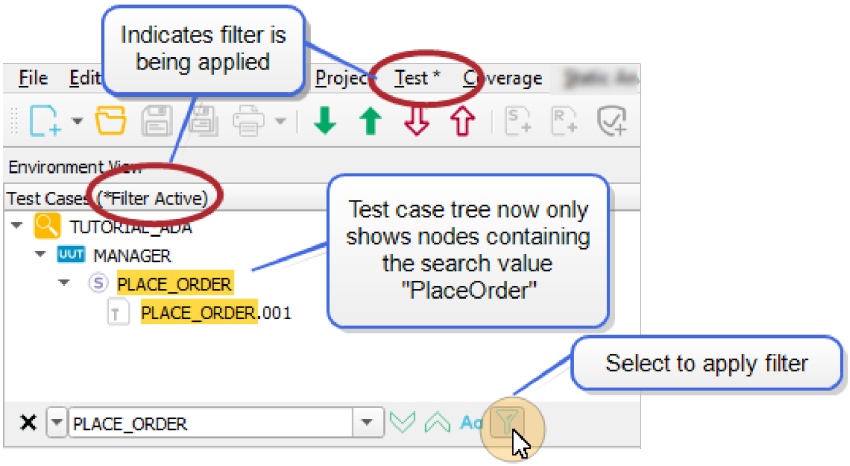
To apply a filter to the Test Case Tree, first select the columns to be included in the search. Click the **Find** button drop-down menu, and from the menu, select the column names.



The Find drop down menu is a tear off menu. Click once on the dotted line, and a floating menu stays on the screen.



Next, select the **Filter** box on the Find Banner. Notice that after selecting **Filter,** two filter indicators are displayed. The menu bar Test item is now displayed as **Test ***, and the Test Cases header in the Test Case Tree is now displayed as **Test Cases (*Filter Active)**. These indicators make it easy to detect if a filter has been applied to the tree.



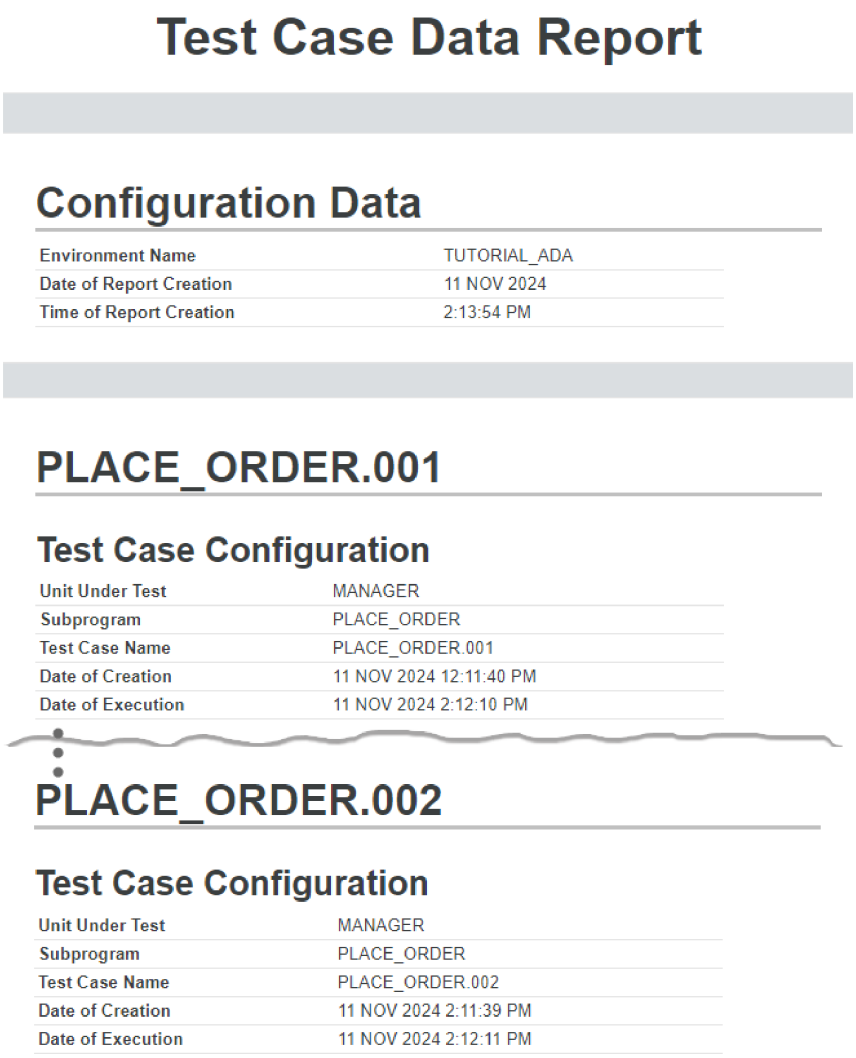
Enter the search value and press the **Enter** key. The Test Case Tree will now filter to only show nodes containing the search value.

To change the search term and reapply the filter, enter the new term in the search text box, click **Filter** to uncheck, and click **Filter** again to check it. The filter for the new term is applied and the results will be displayed.

To Apply a Search Filter to Reports

Test Case Tree filters can be applied to Test Case Data Reports, Execution Results Reports and Full Reports.

In this example, the TUTORIAL_ADA environment is executed and a Test Case Data Report is generated (**Test => View => Test Case Data**). A portion of the Test Case Data Report is shown below. The report contains information for all test cases.



Test Case Data Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	11 NOV 2024
Time of Report Creation	2:13:54 PM

PLACE_ORDER.001

Test Case Configuration

Unit Under Test	MANAGER
Subprogram	PLACE_ORDER
Test Case Name	PLACE_ORDER.001
Date of Creation	11 NOV 2024 12:11:40 PM
Date of Execution	11 NOV 2024 2:12:10 PM

PLACE_ORDER.002

Test Case Configuration

Unit Under Test	MANAGER
Subprogram	PLACE_ORDER
Test Case Name	PLACE_ORDER.002
Date of Creation	11 NOV 2024 2:11:39 PM
Date of Execution	11 NOV 2024 2:12:11 PM

A filter is applied for PLACE_ORDER.001 in the Test Case Tree, and the report is re-executed. (**Test => View => Test Case Data**). The report now shows information for only the test case that is displayed for the filter.

Test Case Data Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	11 NOV 2024
Time of Report Creation	2:23:56 PM

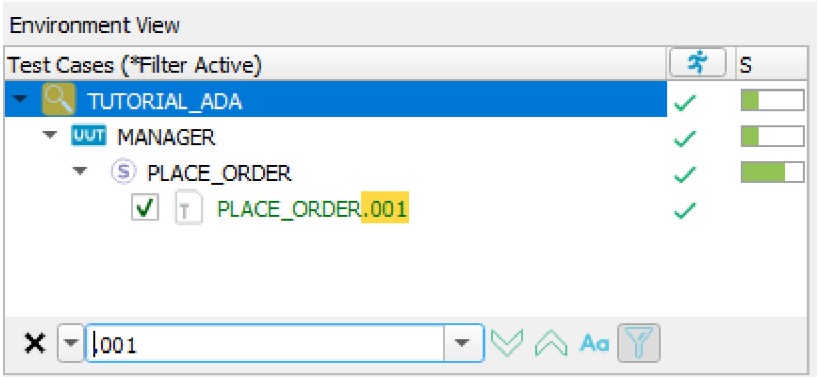
PLACE_ORDER.001

Test Case Configuration

Unit Under Test	MANAGER
Subprogram	PLACE_ORDER
Test Case Name	PLACE_ORDER.001
Date of Creation	11 NOV 2024 12:11:40 PM
Date of Execution	11 NOV 2024 2:12:10 PM

To Apply a Search Filter to Test Scripts

The Test Case Tree filters can also be applied in exporting test scripts. The tree below has been filtered to search the Test Cases for **.001**.



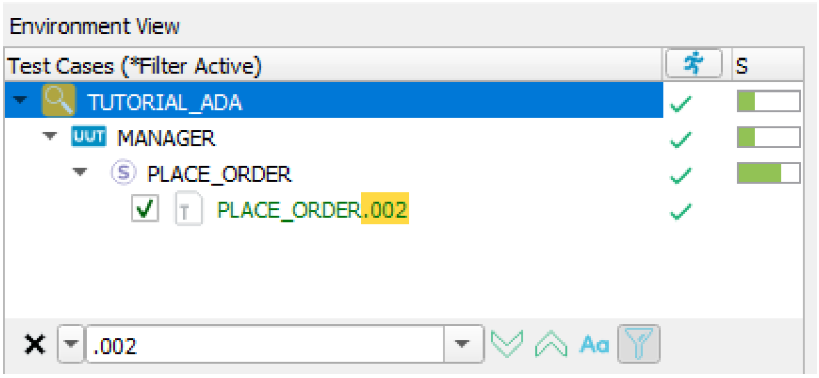
A test script is then exported from the TUTORIAL_ADA environment level of the tree. TUTORIAL_ADA is selected, then, **Test => Scripting => Export Script** is selected. The following test script is generated. Note that the only test case in the test script is for the test case returned by the filter. In an unfiltered tree, the test script would contain all test cases for the environment.

```
-- VectorCAST 25 (11/04/24))
-- Test Case Script
--
```

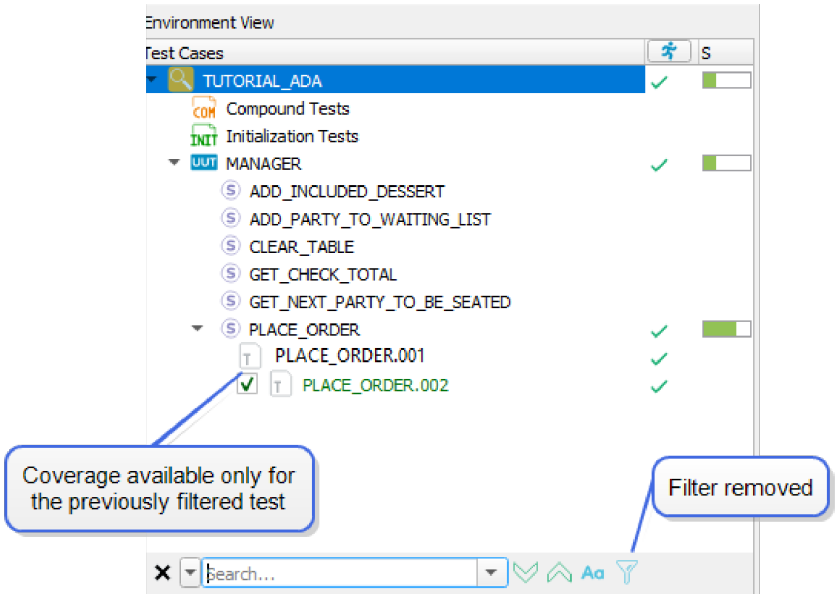
```
-- Environment : TUTORIAL_ADA
-- Unit(s) Under Test: MANAGER
--
-- Script Features
TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT
TEST.SCRIPT_FEATURE:MIXED_CASE_NAMES
TEST.SCRIPT_FEATURE:ADA_DIRECT_ARRAY_INDEXING
--

--Unit:MANAGER

--Subprogram: PLACE_ORDER

-- Test Case: PLACE_ORDER.001
TEST.UNIT:MANAGER
TEST.SUBPROGRAM:PLACE_ORDER
TEST.NEW
TEST.NAME:PLACE_ORDER.001
TEST.NOTES:
This test is the same as the one created by following all of
the steps in the "Ada Tutorials -> Basic Tutorial" from the
VectorCAST Getting Started manual.
It shows the basic concepts associated with setting input and
expected values for both the Unit Under Test and Stub Functions.
TEST.END_NOTES:
TEST.VALUE:MANAGER.PLACE_ORDER.TABLE:2
TEST.VALUE:MANAGER.PLACE_ORDER.SEAT:1
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SOUP:ONION
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SALAD:CAESAR
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.ENTREE:STEAK
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.BEVERAGE:MIXED_DRINK
TEST.VALUE:DATABASE.GET_TABLE_RECORD.DATA.NUMBER_IN_PARTY:0
TEST.VALUE:DATABASE.GET_TABLE_RECORD.DATA.CHECK_TOTAL:0
TEST.EXPECTED:DATABASE.UPDATE_TABLE_RECORD.DATA.IS_OCCUPIED:TRUE
TEST.EXPECTED:DATABASE.UPDATE_TABLE_RECORD.DATA.NUMBER_IN_PARTY:1
TEST.EXPECTED:DATABASE.UPDATE_TABLE_RECORD.DATA.ORDER(1).DESSERT:PIE
TEST.EXPECTED:DATABASE.UPDATE_TABLE_RECORD.DATA.CHECK_TOTAL:12..16
TEST.END
```

To Apply a Search Filter to Enable Coverage

A Test Case Tree filter can also be used to selectively enable coverage. In this example, the environment is initially built and executed without coverage. The tree below has been filtered to search the Test Cases for **.002** and Statement coverage is initialized (**Coverage => Initialize => Statement**) at the environment level.



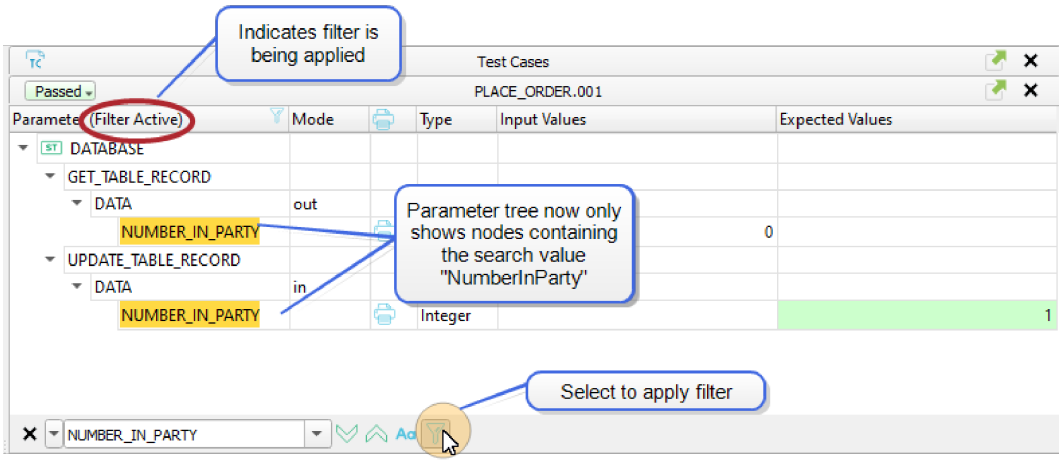
The environment level is executed and the filter is then removed to display the entire Test Case Tree. Notice that coverage is available only for the test that was selected by filtering. This is indicated by a Check Box next to the previously filtered test case name.



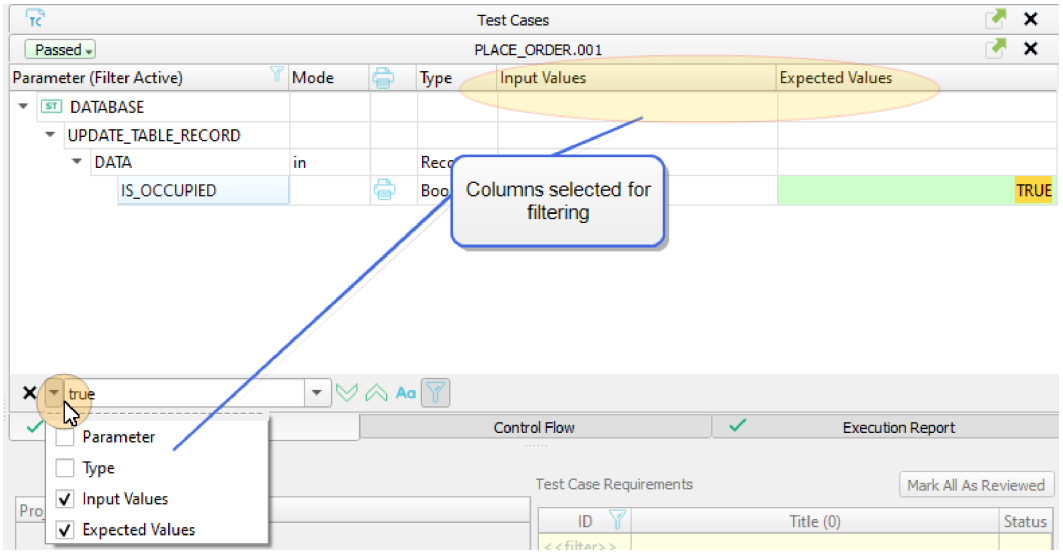
To Apply a Search Filter to the Parameter Tree

A search filter can be applied to the Parameter Tree. Filters are useful for pruning the tree and showing only those items of current interest. A filter is applied to all parameters in the tree, regardless of whether they are in an expanded part of the tree or not.

To apply a filter to the Parameter Tree, select the **Filter** button on the Find Banner. Enter the search value and press the **Enter** key. The Parameter Tree will display the unit node(s) associated with the filtered results. Note that **(Filter Active)** is displayed next to the Parameter column heading, making it easy to identify when a filter is being applied.



A filter can be applied to any column or combination of columns in the Parameter Tree. To select column filters, click the drop-down arrow to the left of the search entry field, and from the menu, select the column names.



To Repeat a Search

The **Edit => Find Again** command enables you to repeat a search for previously specified text.

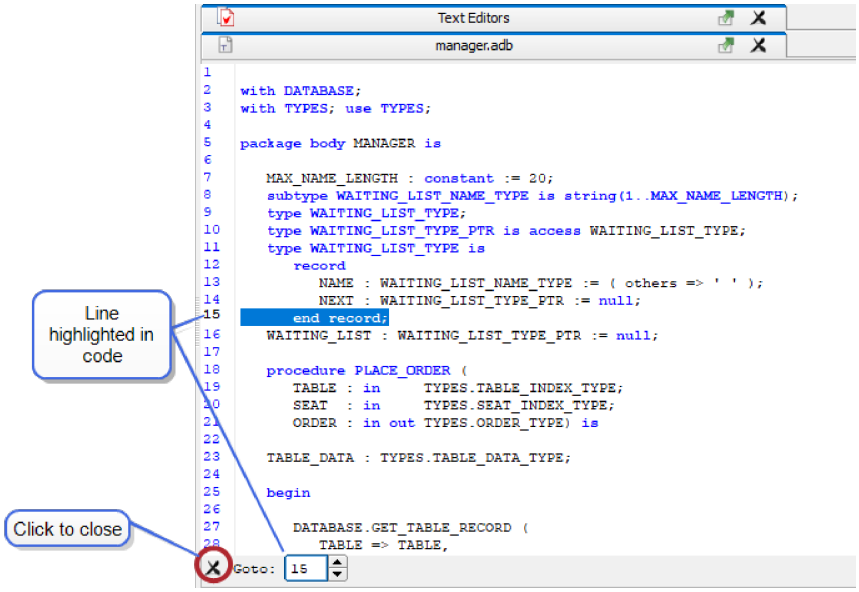
The shortcut for the **Edit => Find Again** command is **F3**.

To Goto a Line

The **Edit => Goto Line** command enables you to go to a specific line number in User Code and source code in Text Editors. The file line numbers start at 1. This command is available in any text file: script files, source files, text reports, and results files.

The shortcut for the **Edit => Goto Line** command is **Ctrl+G**.

At the bottom of the Text Editor window a Goto dialog opens. Insert the line number and hit Enter. The specific line is highlighted in the code.

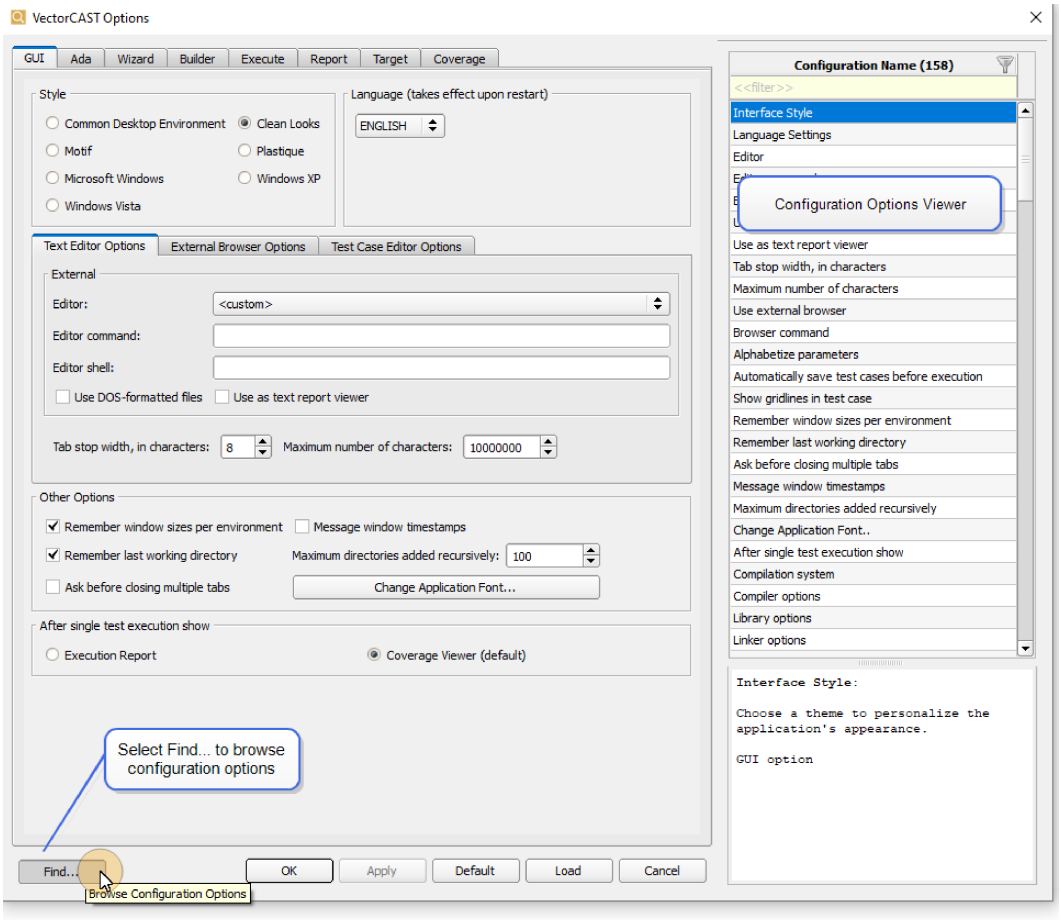


Note: The file line number is the source file line number, not the executable source code line number.

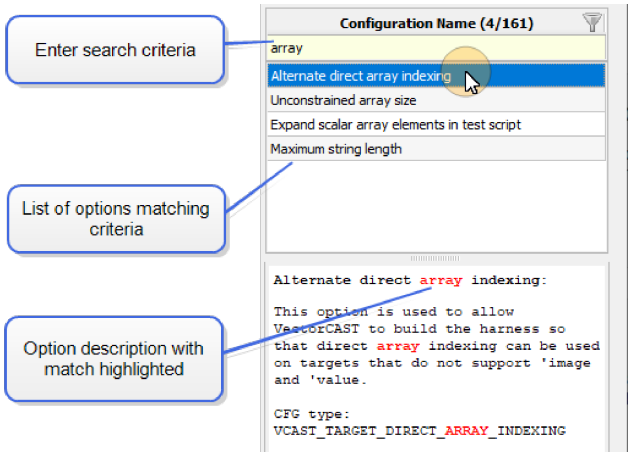
Browsing Configuration Options

Using the Configuration Options Viewer

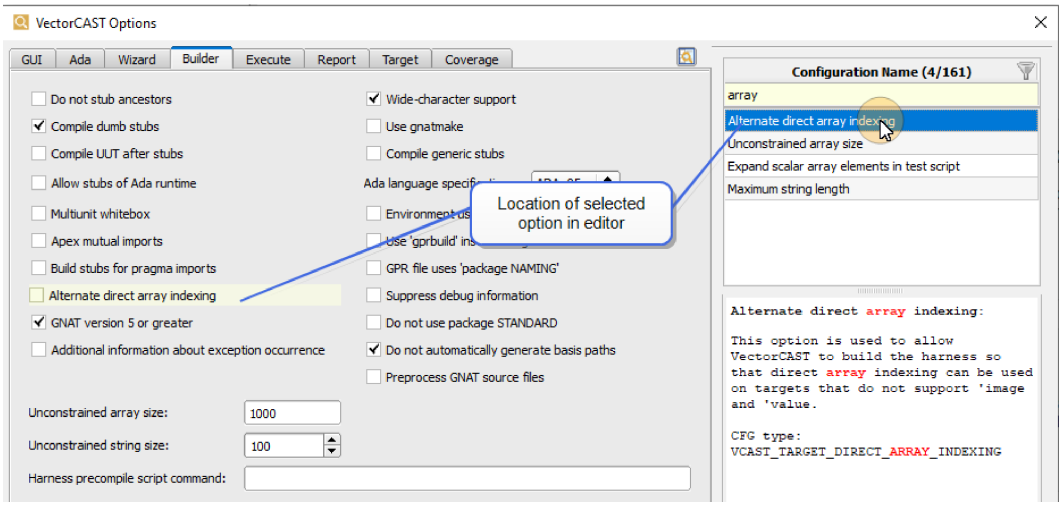
The Configuration Options Viewer provides users with the ability to easily locate and manage configuration settings located in the Options dialog. The viewer is accessed by toggling the **Find...** button located in the lower left corner of the Options dialog.

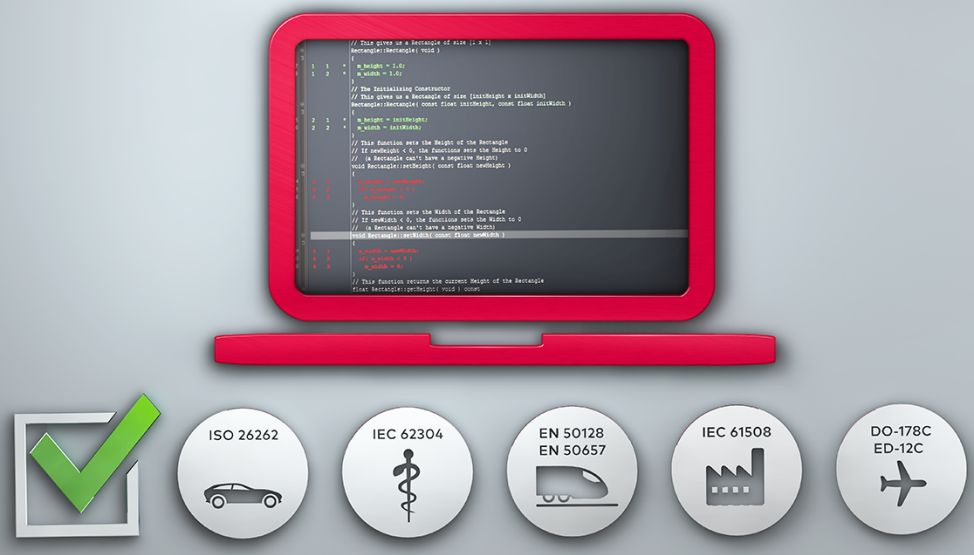


The Configuration Options viewer provides a filter to search for relevant options. As the user types, the viewer dynamically filters the list of settings, presenting only those configuration options that match the input criteria. Highlighting an option in the list displays the description of the selected option in the lower pane, where the matches to the search criteria are highlighted in red.



Upon selection of an option in the list, the corresponding Options editor interface is immediately brought into focus and the option field is highlighted in yellow. This allows the user to edit or review the current value of an option without the need to navigate through multiple layers of nested tabs.





Creating a New Environment

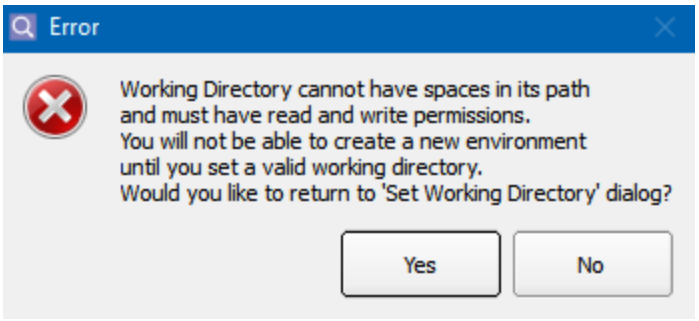
Using the Wizard to Create an Environment

To Set the Working Directory

The working directory is the default location VectorCAST uses for building environments and when loading an environment script that has been saved with relative paths. The local configuration file is also saved there. This file contains all tool options that are modified in the **Tools => Options** dialog. Also, it is the default directory in the File Save dialog when saving a report or exporting a script to a file, and the default directory in the File Open dialog when opening an environment or importing a script.

The **File => Set Working Directory** command is used to change the current working directory before creating a new environment. This command is not available when an environment is open. When you select this command, a dialog appears enabling you to navigate to the directory of choice or create a new directory.

If you select a directory which contains spaces or does not have read / write permissions, you will receive the following error and be prompted to return to the Set Working Directory dialog to set a valid directory.



The “Remember last working directory” option is located on the **Tools => Options dialog, GUI tab**. Whenever this option is enabled, VectorCAST opens in the last-used working directory, regardless of how the VectorCAST application is invoked.

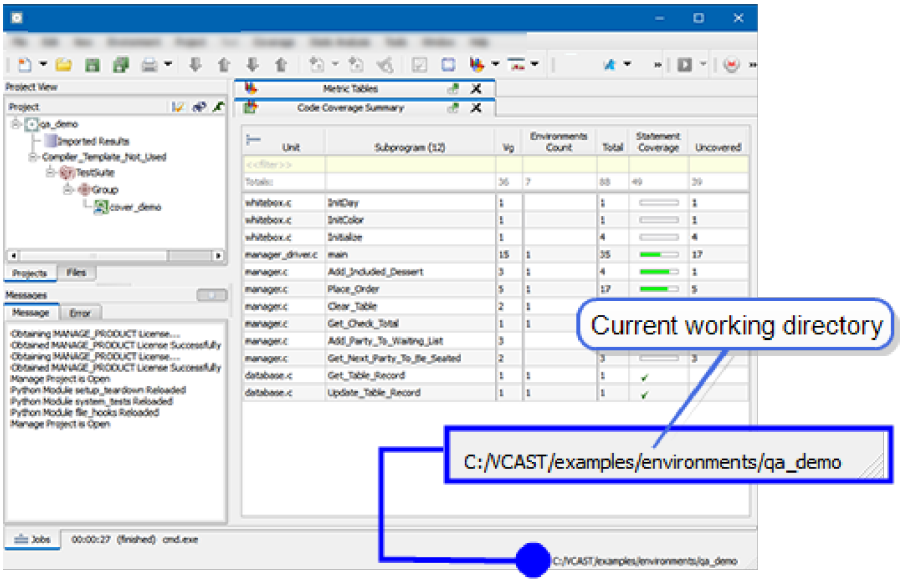
When you start the VectorCAST application from the command line, the working directory is the directory from which you invoked the application, unless the “Remember last working directory” option is set.

When you start VectorCAST from the Windows Start menu, the working directory is the Environments directory in the VectorCAST installation directory, unless the “Remember last working directory” option is set. On Windows, you can change the default working directory by modifying the Windows properties of the VectorCAST shortcut.

Note: The working directory must not contain spaces and must have Read and Write permissions. If you start VectorCAST from such a directory, the status bar has a red outline. The Set Working Directory error dialog appears, prompting you to enter a valid directory.

If you install VectorCAST in the C:\Program Files\ directory, then the Environments directory will have spaces in its path and therefore will be invalid. If this happens, set the working directory to a different location on your drive.

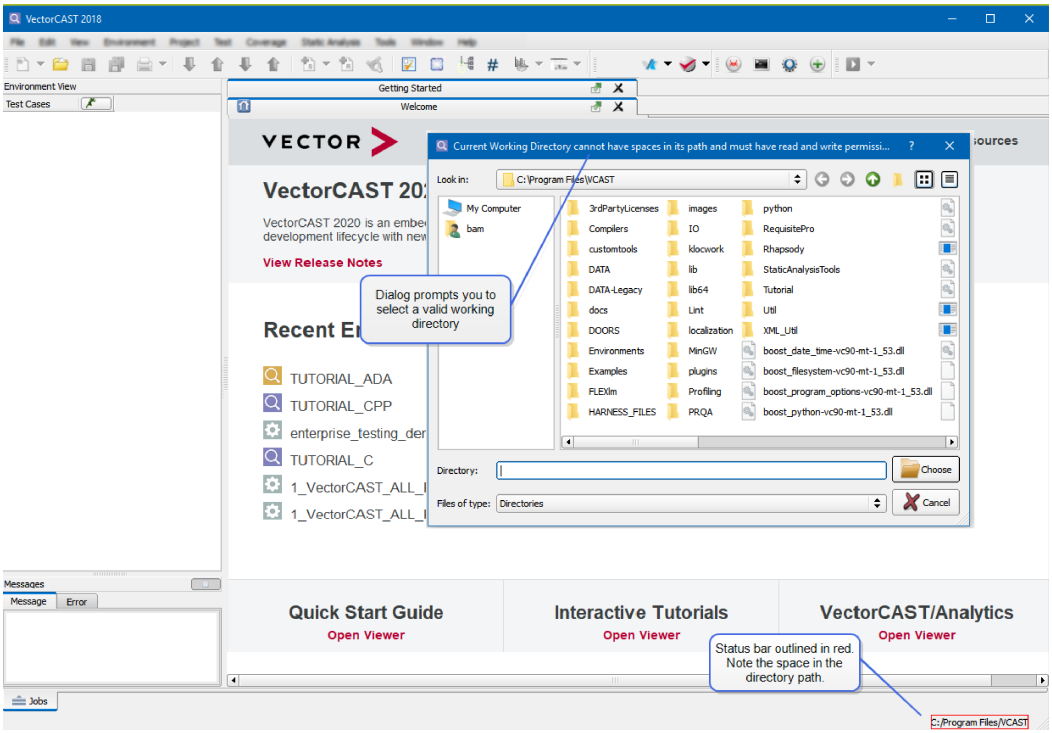
The current working directory is indicated in the status bar in the main application window.



Troubleshooting the Working Directory

Space in Working Directory Path, or Current Working Directory is Not Writeable

For Windows users, VectorCAST does not support spaces in the default directory path, and you must have write permission. (For some Windows users, the C:\ drive is not writeable.)



If you start VectorCAST from such a directory, the status bar has a red outline. The Set Working Directory dialog appears, prompting you to enter a valid directory.

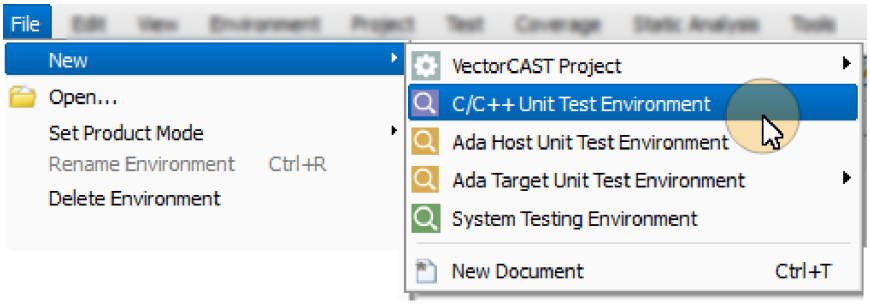
See also "To Set Defaults for the Wizard" on page 123

To Start the Wizard

The **File => New** menu item is used to begin a new VectorCAST Project, C/C++ Unit Test Environment, Ada Host Unit Test Environment, Ada Target Unit Test Environment or System Testing environment. The menu changes depending on what products you have licensed from Vector.

Alternatively, the **New** button on the toolbar  has a small arrow to the right. Clicking this arrow causes a popup menu to appear, which lists the types of environments that can be built.

In the figure below, the user has licensed: VectorCAST/Project, VectorCAST/C++, VectorCAST/Ada, one or more Ada targets and VectorCAST/System Testing. Your menu may differ, depending on the products you have licensed from Vector.



If you have only one product licensed from Vector, then clicking the **New** button on the toolbar brings up the wizard to create a new environment of that product type.



Tip: The Create New Environment wizard reflects the current settings on the **Tools => Options** dialog, **Builder** tab. If you consistently want to create environments with Whitebox checked or Coverage on, for example, choose **Tools => Options**, and go to the **Builder tab** to set these options. Then, the next time you choose **File => New => Environment**, those settings are already specified, saving you time.

To Save the Settings in the Wizard

In the Create New Environment wizard, the **Save** button is used to create an environment script that reflects the data entered in the environment wizard. The name of the environment script is the environment name entered in the dialog with an ".env" extension. Once an environment is built, a script is automatically created. The **Save** button is used to create a script while working through the Create New Environment wizard.

If saving the environment script would modify the existing environment script of the same name, VectorCAST asks if you want to overwrite the original environment script with the changed version.

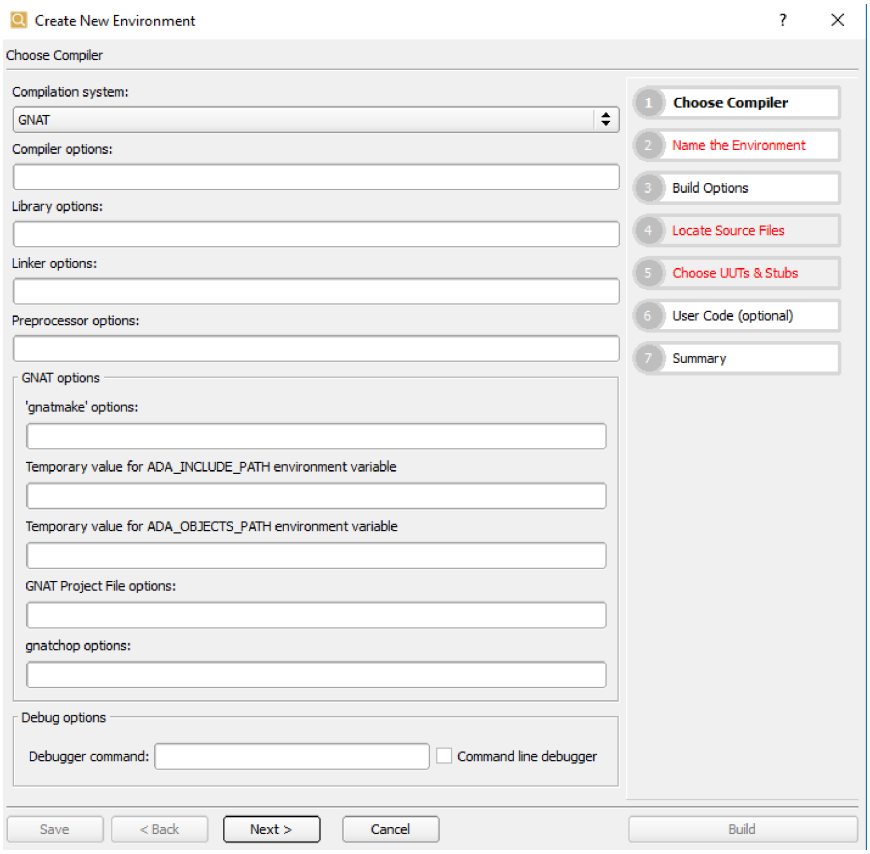
To create an environment script from a currently open environment, use **Environment => Scripting => Create**.

```
clicast -e <env> ENvironment SCript Create <scriptfile>

Create an environment script file from an existing environment. If no
extension is provided, .env is used.
```

Step 1: Choose Compiler

If you have already set your compiler, then the wizard skips over Step 1 and opens on **Step 2: Name the Environment.** You can click **Step 1: Choose Compiler** to go directly to Step 1.

On this page, you select the compiler. The preprocessor and compiler commands must be on your default system PATH.



Select your compiler from the dropdownmenu next to Compilation system.

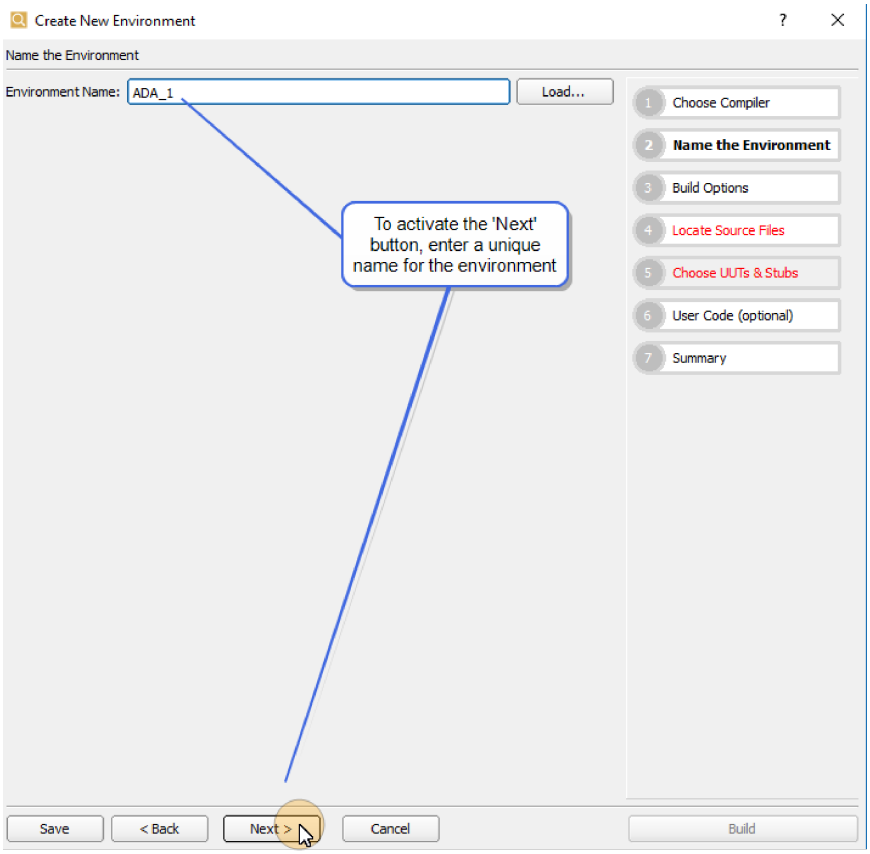
GNAT options are enabled only when the compilation system is set to GNAT.

See also "Options: Ada Tab" on page 120

Step 2: Name the Environment

On this page you perform the following actions:

- Give the environment a name, which becomes a directory
- Load in an existing environment script (.env), which fills in the settings in the whole wizard (optional)



Environment Name: Enter a unique name you wish to give to the new environment. Any space characters you type are converted to "_" characters. A directory is created with this name, and a file is created with this name and a ".vce" extension. If a directory with that name already exists in the working directory, the name entry is outlined in red. The tool-tip asks you to choose a unique name for the environment.

To Load an Environment Script

Load... Button: Loads an existing environment script into the Create New Environment dialog to create an environment similar to an existing one. Click the **Load...** button, select an environment script (.env) and click the **Open** button. The wizard's pages are filled in according to the specifications in the environment script. At this point you can make any changes and then build the environment.

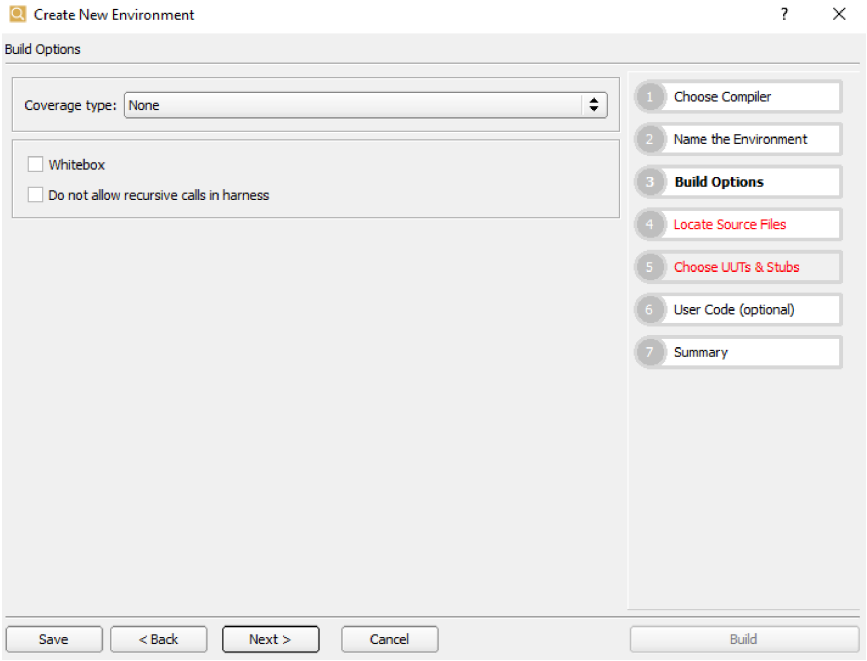
Tip: When loading an environment script that was saved with Relative Paths, ensure that your current working directory is the same as it was when the script was saved.

If you need to modify an *open* environment, say, to turn on Whitebox, add a Search directory/Parent Library, or change the stubbing strategy, use **Environment => Update Environment**. See also, "To Update an Environment" on page 136.

Step 3: Build Options

On this page, you select:

- **Coverage type** (optional) - Although coverage can be added at any time after the environment has been built, it is recommended that you select coverage here in the wizard for increased speed and efficiency.
- **Whitebox** (optional) - A Whitebox environment provides visibility to data defined in the package body, as well as the package spec. Unchecking this box allows visibility only to data defined in the package spec.
- **Do not allow recursive calls in the test harness** (optional)



To Turn on Coverage

Coverage Type: Choose a type of code coverage for the environment from the dropdown menu, if desired. To build an environment without code coverage, set the Coverage Type to NONE.

To turn off coverage in an *open* environment, use the **Environment => Update Environment** command.

To set a Coverage Type as the default setting when you open the Create New Environment wizard, use the **Tools => Options** dialog, **Wizard** tab, **Coverage Type** option.

You can initialize code coverage at any time after the environment is built.

To Turn on Whitebox

Whitebox: If you wish to perform Whitebox testing, check the box marked **Whitebox**. The unchecked state is the default, unless you have checked "Whitebox" on the **Tools => Options** dialog, **Wizard** tab.

Whitebox gives you access to data visible in the UUT package body (objects and subprograms) and includes access to private types in the UUT. Whitebox therefore allows access to everything at the

package level, both spec and body, and everything in packages nested one level down.

Whitebox makes the following changes to copies of the source files :

- Inserts a nested package definition to a UUT specification to allow manipulation of private types and access to subprograms defined in the body of a UUT.
- Inserts a package body separate definition to a UUT body to correspond to a UUT specification change.

Whitebox Restrictions and Limitations

Because the utility makes modifications to the source files for the unit under test, there will be cases when using the Whitebox Test Utility may result in link errors for the test harness, as described below:

Out-of-date units VectorCAST makes every effort to re-compute the compilation order and recompile any out-of-date units that result from re-compiling the UUT specification and body. In rare cases, link errors from out-of-date units may occur. If they do, you must recompile the out-of-date units into the VectorCAST Ada library. For specific details regarding your compiler, please contact VectorCAST Technical Support: support@vector.com.

Circular References In some cases, a whitebox conversion may result in a circularity during link. This is generally caused by a unit in the dependency list of the UUT "with"ing the UUT. Because VectorCAST-created stubs depend on harness files, and these harness files depend on the Unit Under Test, a circular dependency can result. This problem can be avoided by creating stubs for dependent units.

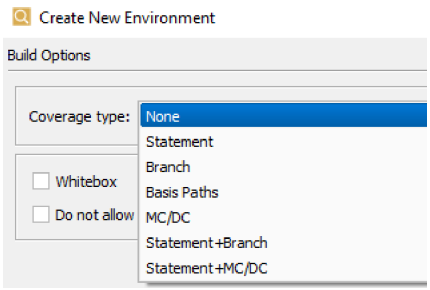
Due to the inherent invasiveness of the Whitebox technique, the following limitations should be noted:

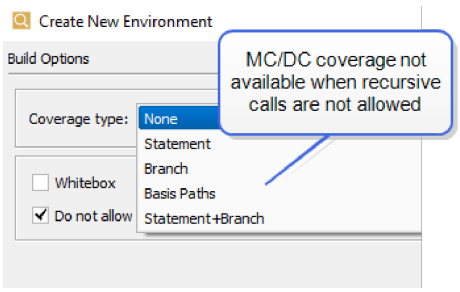
- Nested packages must use named notation for the end statement
- A source file can only contain a single package body
- Private types defined in nested package specification are not supported

As an alternative to Whitebox and its modifications to source files, try using the Builder option “Multiunit whitebox” instead.

To Not Allow Recursive Calls in the Harness

Do not allow recursive calls in harness: If you wish prevent recursive calls in building the harness, check this box. The result of enabling this feature is that it limits the type of coverage available for selection. VectorCAST uses recursion when instrumenting MC/DC and Level_A coverage.

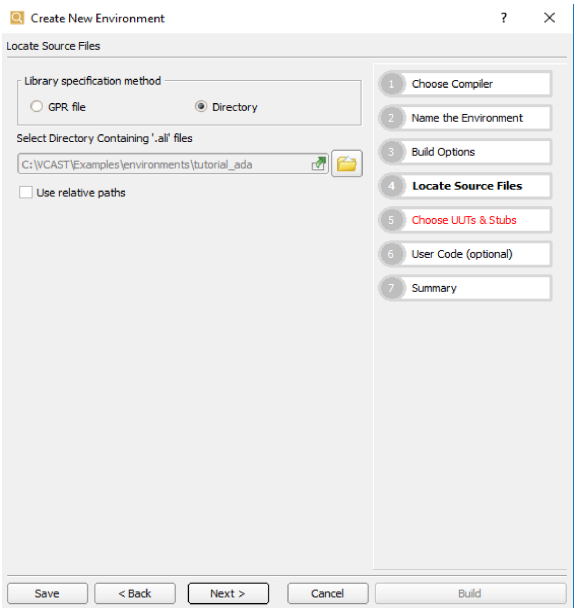




Step 4: Locate Source Files

Select Directory Containing Parent Library: Specify the directory where the unit you wish to test is stored. The Ada source files must already be compiled. The text on this page depends on which compiler you are using:

- Green Hills 95 or ObjectAda compiler – the text on this page reads “Select Directory Containing ADA.LIB.” Navigate to the location of the ADA.LIB file.
- ActivAda compiler – the text reads “Select Parent Library Directory.” Navigate to the directory which is the parent library directory. This Parent Library can be over-ridden with the environment variable VCAST_PARENT_LIBRARY. See Appendix E.
- GNAT compilers – select the Library specification method by selecting either the GPR file or Directory radio button. Selecting GPR file, the text reads “Select GPR file”. Selecting Directory, the text reads “Select Directory Containing .ali files”.
- DACS compiler – the text reads “Select Ada Library file.” Navigate to the directory that contains the .alb file.
- SCORE compiler – the text on this page reads "Select Ada Libraries." Click the **Add Library** button, navigate to the location of a SCORE library file (.alb) and click **Open**.
- Library specification method – select either GPR file or directory



To use relative paths as the default setting when you open the Create New Environment wizard, use the **Tools => Options** dialog, Wizard tab, “Use relative paths for parent library” option.The target I/O directory will also use relative paths is this option is selected.

Note: The Library/Directory path may be entered with a relative or absolute path consistent with the default directory. See Appendix D for details specific to a particular compiler.

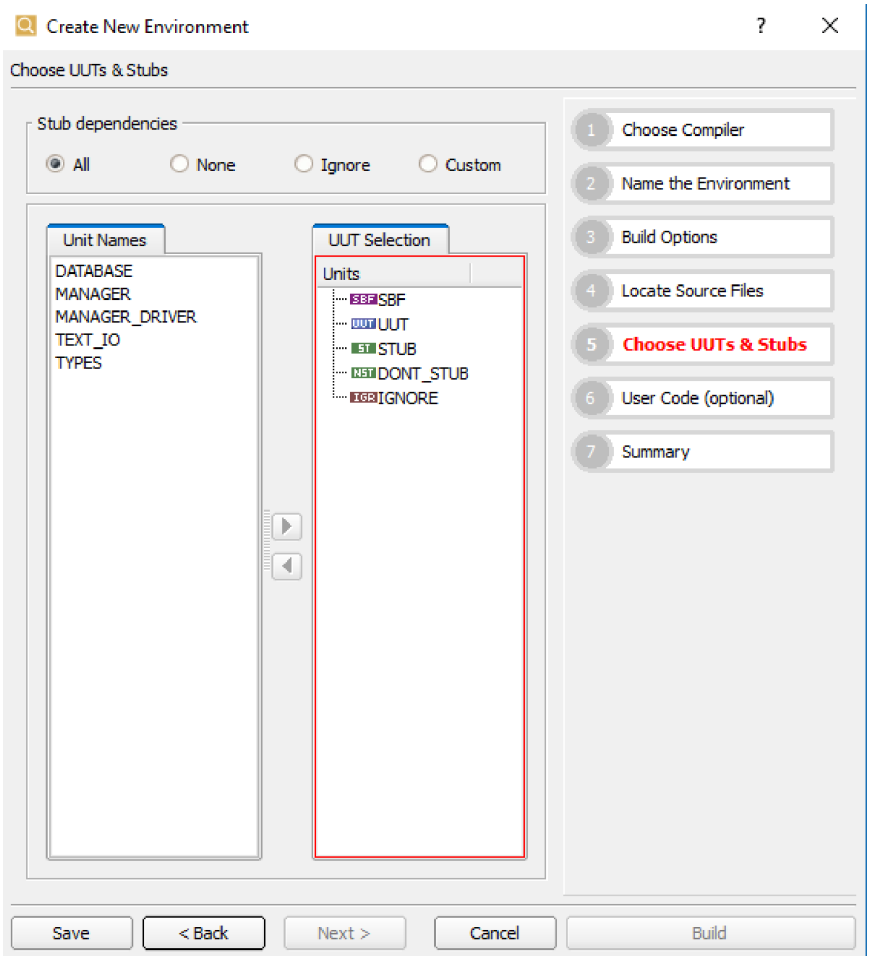
Note: :On platforms where the operating system is case sensitive (i.e. Linux) the exact case matched path must be entered.

Step 5: Choose UUTs & Stubs

This page of the wizard, called **Choose UUTs & Stubs**, is used to specify the Units Under Test (UUTs) and their optional compilation arguments, stubs, and Classes of Interest.

On this page you perform the following:

- Specify the Unit(s) Under Test (UUT)
- Choose stubbing type for all dependents or customize the stubbing for individual units. (The default is to stub all dependents, unless a different default setting is specified on the **Tools => Options, Wizard tab**.)



Specifying Unit(s) Under Test: The units found in the directories specified in the Source directories list are displayed on the left, under the tab labeled “Unit Names.” Removing or adding a directory to the Source directories list changes the items in this list.

To specify a selected unit as a Unit Under Test, move it from the left to the right. It appears in the UUT selection list under the UUT icon (UUT). You may specify more than one unit as a Unit Under

Test.

You may specify more than one Unit Under Test.

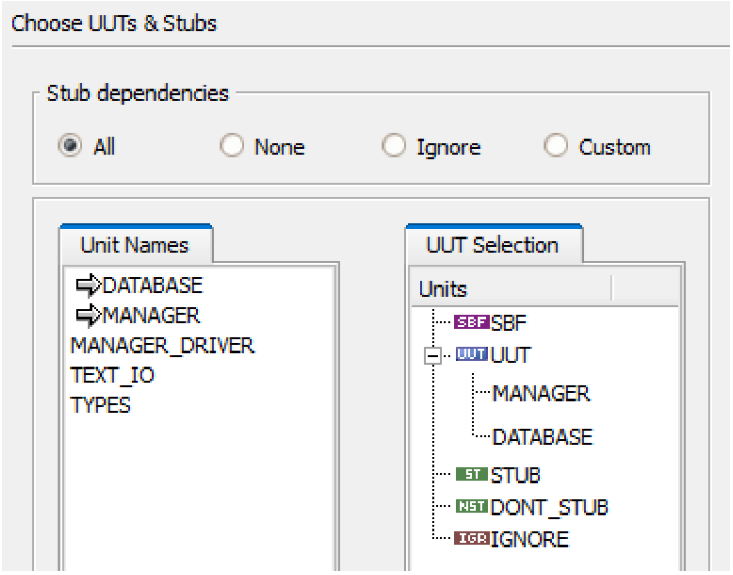
To move selected units into the UUT selection list:

- select a unit name and click the move-to-right button
- drag and drop a unit name onto the "UUT" node of the hierarchy (UUT)
- double-click a unit name.

To move selected units out of the UUT selection list:

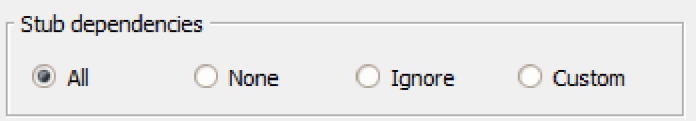
- select the name and click the move-to-left button
- drag and unit out of hierarchy and drop it in the Unit Names list
- double-click a UUT name.

In the following example, the units MANAGER and DATABASE are the Units Under Test. They are positioned under the UUT node (UUT) of the UUT Selection list.



Stubbing Type: When unit testing, it is often desirable to stub functions called by the unit you are testing. Stubbing allows you to test without waiting for dependent units to be completed and tested. VectorCAST stubs enable you to capture parameters passed in, control parameters passed out, and raise exceptions.

When you first open the Create New Environment wizard, the default stubbing strategy is to stub All of the dependent units. The radio button selected in the **Stub Dependencies** group reflects the stubbing strategy chosen for all of the dependents of the Units Under Test. (You can change the default by going to the **Tools => Options dialog, Wizard tab**, and changing it to None or Custom. Thereafter, when you open this dialog, it will default to that setting.) Choose Ignore if you want to use the real source code for each dependent, but save time by not parsing the dependents or any of their dependents. You can get code coverage data on Ignored units.



There are several choices for the stubbing strategy for all dependant units of the UUTs:

>

> **All** - Stub all dependent units for all UUTs. Every subprogram in the file is stubbed; the default

> **None** – Do not stub any dependent units of the UUTs.

> **Ignore** - Do not stub the dependent units or any of their dependent subprograms. This option is useful when, for example, a specific unit is the interface to a set of dependent units that have already been tested. Using Ignore can sometimes dramatically reduce the time it takes VectorCAST to build an environment because it will not parse dependents of an ignored unit.

> **Custom** - Stub only those dependent units indicated interactively; ignore or don't stub others. see below for more information.

To Specify a Stubbing Strategy for Individual Units

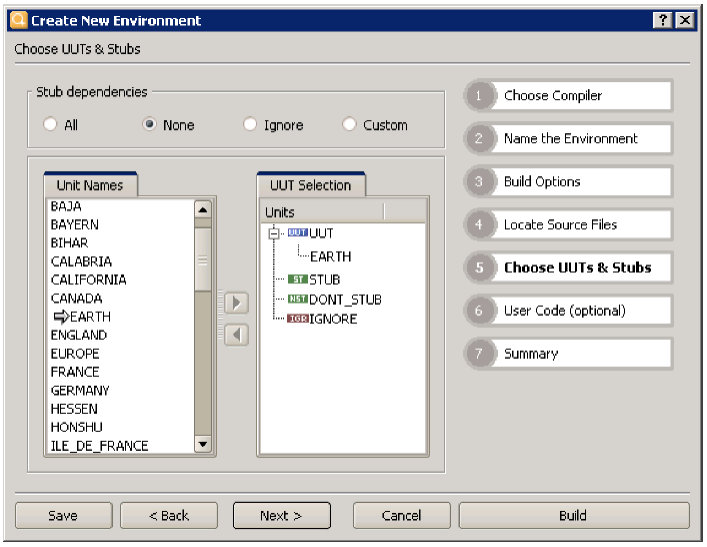
The STUB (STUB) and IGNORE (IGNORE) nodes in the UUT Selection hierarchy enable you to specify a stubbing strategy for individual units by name, in cooperation with the Stubbing strategy you indicated for all units. In effect, it is a way to “override” your overall stubbing strategy for particular units.

> If you click the radio button to stub “None” or “Custom,” all the dependents of the UUTs are not stubbed, except for the ones you place under the STUB or IGNORE nodes.

> If you click the radio button for “Ignore,” all the dependents of the UUTs are ignored, except for the ones you place under the STUB node.

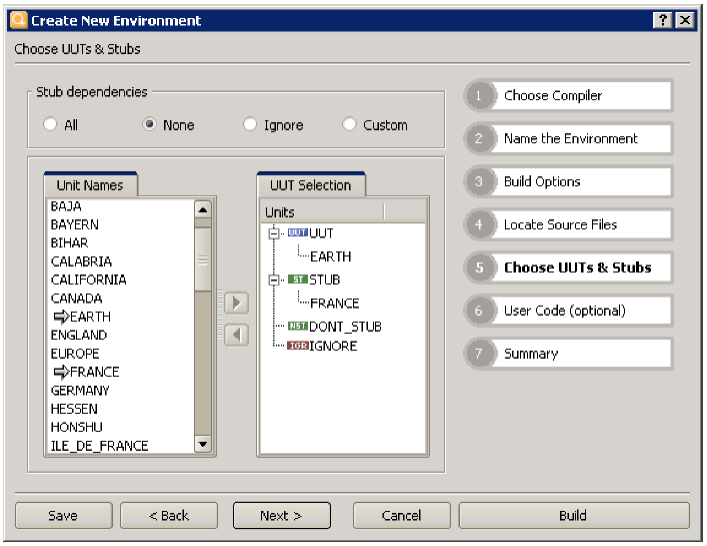
> If you click the radio button for “Stub All,” all the dependents of the UUTs are stubbed, except for the ones you place under the IGNORE node.

In the example below, EARTH is a UUT. We have indicated to Stub **None**, but we plan to “override” that setting to stub one unit, FRANCE, and ignore one unit, GERMANY.



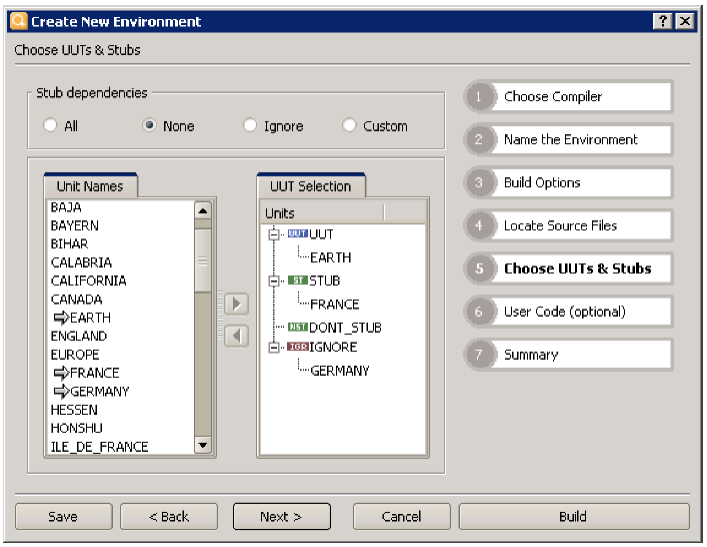
In our example, we drag over the unit FRANCE and drop it onto the STUB node.





We drag over GERMANY and drop it onto the IGNORE node.





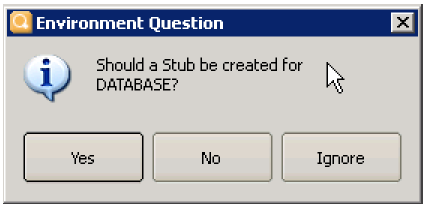
Once this environment is built, VectorCAST writes the following to the environment script (.env).

```
ENVIRO.UUT: EARTH
ENVIRO.STUB:FRANCE
ENVIRO.IGNORE:GERMANY
```

Note: If “Ignore” is selected in the Stub Dependencies group, any units under the IGNORE node are superfluous. Similarly, if “All” is selected in the Stub Dependencies group, any units under the STUB node are superfluous.

To Specify Custom Stubbing Interactively

If you want to indicate that some dependent units be stubbed and some not interactively, specify only the UUT(s) and click the radio button next to **Custom**. After you click the Build button, a dialog box presents you with the choice to stub, not stub, or ignore any dependent units that contain subprograms. This process is interactive, taking place after you click the **Build** button.



Select one of the following buttons for each dependent:

- **Yes**, stub – Stub this unit; any dependents are lost.
- **No**, don't stub – Do not stub this unit; any dependents are parsed and can have their own setting.
- **Ignore** – Do not stub this unit or parse any of its dependents. The remainder of the closure for this dependent is ignored.



Note: An environment variable called "VCAST_NEVER_STUB_LIST" can be set to direct VectorCAST not to stub certain units. The units should be entered in a comma-separated text string by filename without the file extension. Any unit in this "never stub list" will be ignored when building an environment. This feature enables you to choose radio button "stub all" or "ignore" and still specify that some units should not be stubbed.

Also, the environment variable can be set to the full path to a filename. If this file exists, units will be read from this file, with one unit per line. This is especially useful when the list of units is long, or when sharing lists among users.

Step 6: User Code (Optional)

Step 6 is used to optionally specify environment user code, user globals, user parameters, driver prefix user code, unit prefix user code and unit appendix user code. If you have loaded in an environment script having any of these types of user code defined, then you can view and modify the user code here.

On this page you can perform the following, if desired:

- Add environment user code
- Create your own global variables
- Create an object to represent a parameter

This section briefly describes the types of environment user code. For a complete description, see "User Code" on page 390.

User Globals - User Globals provide a mechanism for user-defined types and objects to be included in the test harness. By default, there are five integer objects, five floating point objects, five string objects, and an array of 200 integer elements. All data objects that are defined in User Globals when the environment is built can be manipulated as test data when building a test case.

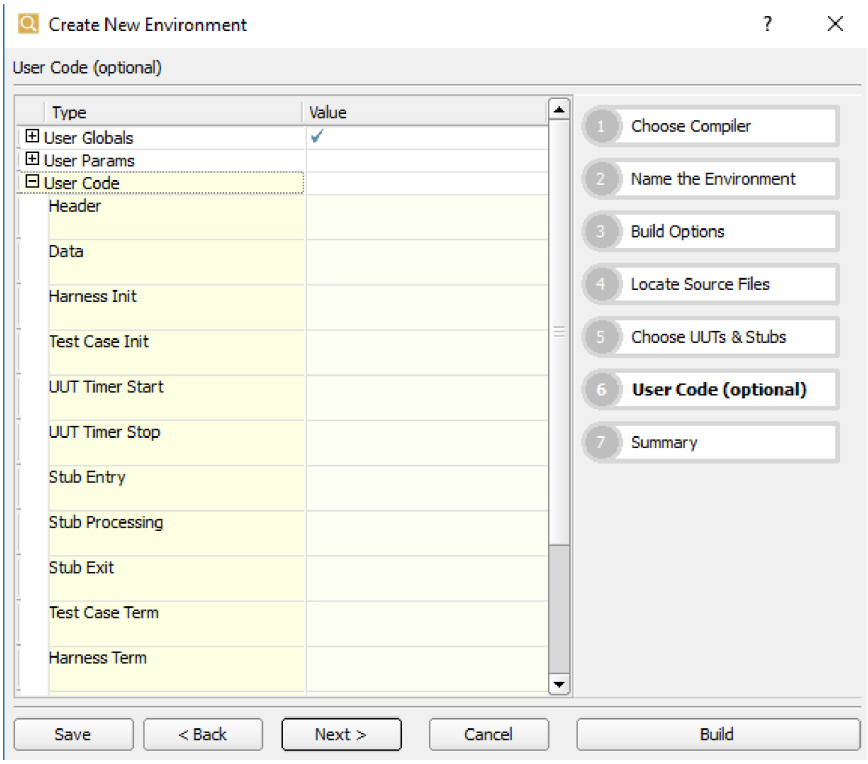
Preface any variables defined here with VCAST_USER_GLOBALS_EXTERN to ensure that only one definition of the variable is created in the test harness. They will be listed in the Parameter Tree under USER_GLOBALS_VCAST.

User Params - User Params are used to instruct VectorCAST to use a user-defined object for a parameter, rather than a VectorCAST-generated object. User Params are used to provide the association between the user-defined parameter and the subprogram's parameter. A syntax example is:

```
<<manager.Place_Order.Table>> VECTORCAST_INIT1
```

User Code - User Code is best suited for operations that relate to the harness as a whole; loading data from a file or initializing a database unit are two examples. It enables you to write source code to handle some of the harness tasks that are not easily accomplished with static data. With user code, you can write code to read data from a file, call initialization routines, or assign and verify data objects

based on dynamic criteria.



Unit Appendix User Code - Unit Appendix User Code is user code that is appended to the end of the specified UUT (and therefore added to the test harness). Because Unit Appendix User Code is considered part of the specified UUT, it is useful for `#including` a concrete subclass for an abstract class, as well as many other purposes.

You have access to all types of environment user code after the environment is built by choosing **Environment => User Code => Edit.**

See also "Environment Script Language" on page 454.

If you are not familiar with Environment User Code, see "Types of Environment User Code".

Step 7: Summary

This page of the wizard, called Summary, displays a summary of your settings. If there is any missing information, the problem is displayed in red text. In particular, it checks that the compiler and preprocessor are on the system PATH and that there are no empty fields. Missing information must be supplied before the **Build** button is enabled.

To Build the Environment

Once you have completed data entry in the Create New Environment wizard, click the **Build** button. The environment builder begins creating the test harness. The Message Window displays status messages.

If you chose “Custom” stubbing, a dialog box presents you with the choice to stub or not stub or ignore

any dependent units that contain subprograms.

See also “To Specify Custom Stubbing Interactively” on page 94 for information about Custom stubbing.

```
clicast -lada ENvironment Build <scriptfile>

Build an environment from <scriptfile>. The environment script
<scriptfile> must be in the current working directory; it cannot be a full
path or a relative path to a file.

    You can also use *.env for <scriptfile>. CLICAST is passed a command
    line with each <filename>.env found in the directory. On Linux, to
    avoid having the command line get too long after expansion, you can
    quote the argument to CLICAST, using "*.env" for <scriptfile>. This
    form passes CLICAST the actual wildcard string for it to expand.
```

If the compiling or linking of the test harness component fails, the diagnostic messages for the compiler are displayed, and the Message Window displays “Environment Built but not compiled”, or “Environment Built but not linked” or “Environment Built but run-time errors exist.” After correcting the errors, choose **Environment => Rebuild Environment**.

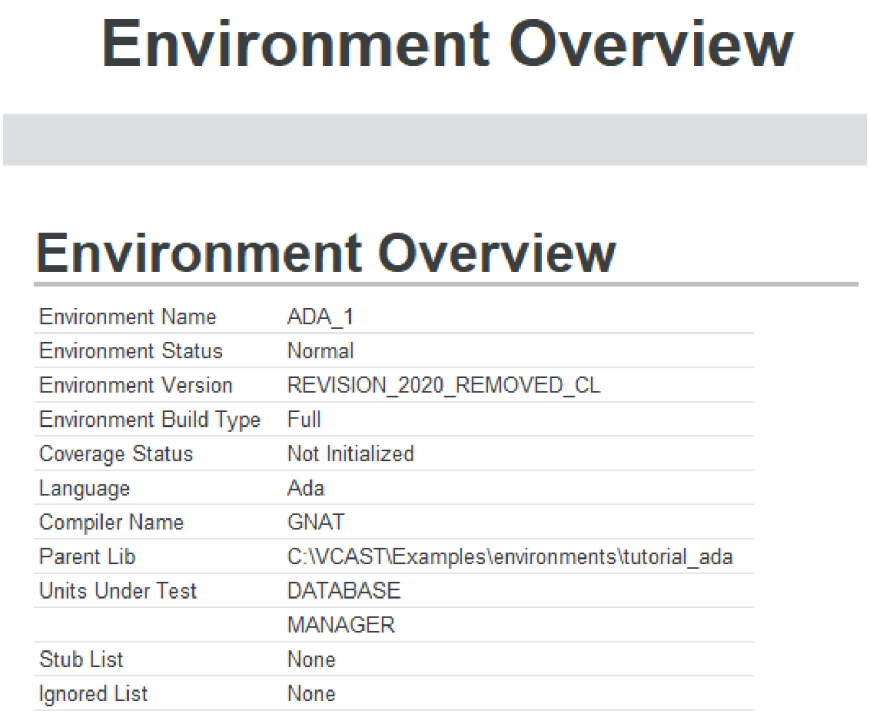
To View the Environment Overview Report

The Overview report is a configuration report on the environment. To open the Environment Overview Report, select **Environment => View => Overview** from the Menu Bar. The report contains the following items:

- **Environment Name** – the name of the environment, the VectorCAST internal version number, and the environment status.
- **Environment Status** – Normal, Compile Error, or Link Error.
- **Environment Version** – the version of the VectorCAST environment builder. For internal use.
- **Environment Build Type** – Full, Incremental, Incremental Rebuild Failed
- **Build History**: –The original build entry is listed at the top of the list, followed by later builds.
- **Coverage Status** –The type of coverage initialized, and whether it is enabled or disabled.
- **Language** – the source code language: Ada, C, C++.
- **Compiler Name** – the name of the compiler used to create the test harness.
- **White Box**: Yes – this line is present in the report only if the environment was created with Whitebox checked.
- **Parent Library** – the Ada library into which the source code has been compiled.
- **Units Under Test** – the name(s) of the unit(s) under test.
- **Stub List** – the names of the units that were stubbed, if any. If units were stubbed by prototype, then the unit name is listed as “uut_prototype_stubs”.
- **Ignored List** – the names of the units that were ignored, if any.
- **Additional Environment Information** – This line is not present unless the Builder option 'multiunit whitebox' is turned on.

```
clicast -e <env> ENvironment Extract Overview [<outputfile>]

Extract environment overview report to standard output or to a file if
specified. If VCAST_CUSTOM_REPORT_FORMAT is HTML, the standard output is
saved to <env>_overview_report.html.
```

A sample Environment Overview report is shown below.



Environment Overview

Environment Overview

Environment Name	ADA_1
Environment Status	Normal
Environment Version	REVISION_2020_REMOVED_CL
Environment Build Type	Full
Coverage Status	Not Initialized
Language	Ada
Compiler Name	GNAT
Parent Lib	C:\VCAST\Examples\environments\tutorial_ada
Units Under Test	DATABASE
	MANAGER
Stub List	None
Ignored List	None

A sample Environment Overview report in Text format is shown below.

```
---------------------------------------------------------------
Environment Overview
---------------------------------------------------------------
Environment Name:   ADA_1
Environment Status: Normal
Environment Version: REVISION_2020
Environment Build Type: Full
Coverage Status:    Not Initialized
Language:           Ada
Compiler Name:      GNAT
  Parent Lib:         C:\VCAST\Examples\environments\tutorial_ada
Units Under Test:   MANAGER
                      DATABASE
Stub List:          None
Ignored List:       None
```

Files Created by the Environment

The following table lists some of the files that are contained in the working directory. Under normal conditions, you will not need to open or edit these files.



Note: These files do not need to be maintained for configuration management or version management. The **Environment => Create Regression Scripts** command creates the three files that need to be archived for each environment (unless you have imported coverage results, which adds one more).

File name	Description
Files in the working directory	
Env.vce	The VectorCAST environment file. In Windows, double-clicking this file or its icon launches VectorCAST and opens the environment.
Env.env	The environment script file.
ADACAST_.CFG	Configuration file created and read by the Tools => Options dialog box.
A directory named *ENV*	The environment directory, which contains the test harness source files, the harness executable, user code files, and environment data files. None of the files in the environment directory should be modified or opened by the user! If they are modified, unpredictable tool performance and/or data loss could result.

Troubleshooting Environment Creation

Tools to Aid Troubleshooting

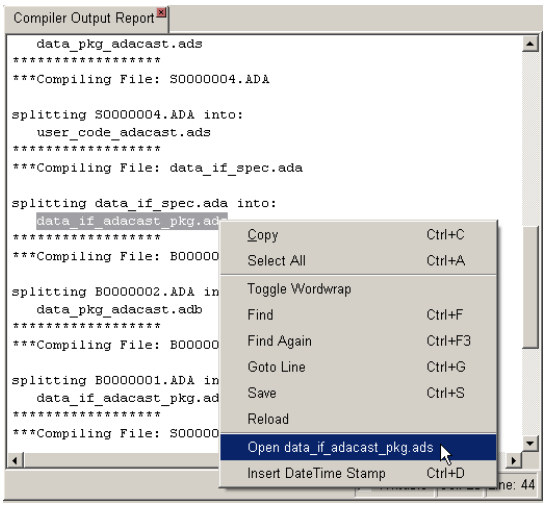
Compile Log File

All compiler diagnostic messages that are generated during the initial environment construction or during the recompile of an existing environment are saved in the Compile Log File (ACOMPILE.LIS) located in the environment directory, and can be viewed using the **Environment => View => Compile Errors** command.

```
clicast -e <env> ENvironment Extract Compile_errors [<outputfile>]

Extract compile errors to standard output or to a file if specified. If
VCAST_CUSTOM_REPORT_FORMAT is HTML, the file is saved to <env>_compiler_
output.html.
```

For example, in the TEXT Compiler Output listing shown below, we have highlighted filename data_

if_adacast_pkg.ads and right-clicked on it. Selecting **Open data_if_adacast_pkg.ads** from the popup menu opens that file. (This feature does not work for HTML reports on Windows, because VectorCAST has an embedded IE browser.)



Binder/Linker Log File

All linker diagnostic messages that are generated during environment linking or during the re-linking of an existing environment are saved in the Binder/Linker Log File (AALINKER.LIS) located in the environment directorty, and can be viewed using the **Environment => View => Link Errors** command.

```
clicast -e <env> ENvironment Extract Link_errors [<outputfile>]

Extract link errors to standard output or to a file if specified. If VCAST_
CUSTOM_REPORT_FORMAT is HTML, the file is saved to <env>_linker_
output.html.
```

Environment Build Log File

All messages generated by the environment builder during environment creation or rebuilding are captured in the Environment Build Log file.

This file can be viewed from within VectorCAST using the **Environment => View => Environment Build Log** command. The log data includes information written to the Message window.

```
clicast -e <env> ENvironment Extract Build_log [<outputfile>]

Extract the environment build log to standard output or to a file if
specified. If VCAST_CUSTOM_REPORT_FORMAT is HTML, the file is saved to
<env>_env_build_report.html.
```

To Create a System Testing Environment

The **File => New => System Testing Environment** command enables you to build a System Testing environment, if you have VectorCAST/QA licensed from Vector. The command may also be launched by clicking the arrow ( ) to the right of the New button  on the toolbar.

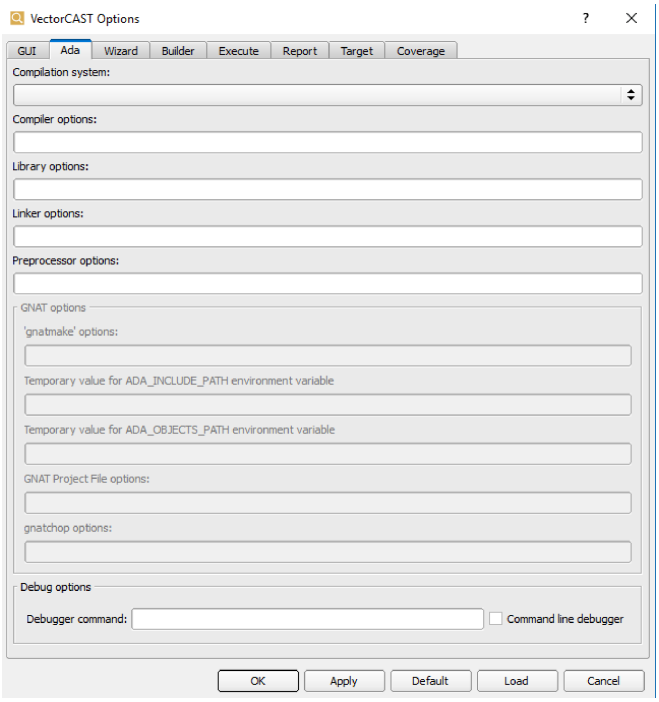
Refer to the *VectorCAST/QA User's Guide* for information on using VectorCAST/QA.

Setting Compiler Options

Options: Ada Tab

Choose **Tools => Options** and click the **Ada** tab.

The Ada tab consists of configurable items applicable to your compiler. The options defined here should be analogous to the options you use for source code compilation, when using the command line interface to the compiler. Pass your cursor over any of the edit boxes to see an explanation of that option in a tool-tip.



"To Set VectorCAST Options" on page 242

Compilation System

```
clicast option COMPILATION_SYSTEM <compiler-name>

The Ada compiler to be used to compile and link the test harness.
```

Compiler Options

```
clicast option COMPILER_OPTIONS <options>

Command line options to be used when invoking the compiler.
```

Library Options

clicast option LIBRARY_OPTIONS <options>

Command line options to be used when creating the test Ada library.

Binder Options

clicast option BINDER_OPTIONS <binder-options>

GNAT only. Specify command line options to be used when invoking the binder.

Linker Options

clicast option LINKER_OPTIONS <options>

Command line options to be used when invoking the linker.

Preprocessor Options

clicast option PREPROCESSOR_OPTIONS <options>

Command line options to be used when invoking the Ada preprocessor (only available for some compilers).

Temporary Value for ADA_INCLUDE_PATH

clicast option VCAST_GNAT_INCLUDE_PATH <directory path>

GNAT only. This value overrides the ADA_INCLUDE_PATH environment variable used by GNAT to find source files.

Temporary Value for ADA_OBJECTS_PATH

clicast option VCAST_GNAT_OBJECTS_PATH <directory path>

GNAT only. This value overrides the ADA_OBJECTS_PATH environment variable used by GNAT to find source files.

GNAT Project File Options

clicast option GPR_OPTIONS <GNAT Project File options>

GNAT only. Switches used to control behavior of GPR file (such as '-Xname=value' for GPR external variables).

Gnatchop Options

clicast option GNATCHOP_OPTIONS <gnatchop options>

GNAT only. Use this field to enter options to be used by VectorCAST whenever "gnatchop" is called, either during environment build/compile or coverage instrumentation.

Debugger Command

clicast option ADA_DEBUG_CMD <command>

This option overrides the default debug command that VectorCAST uses to invoke the debugger.

Other Options

These options are not available in the VectorCAST Options dialog.

clicast option VCAST_RHI_CWD <directory>

Remote directory in which to execute commands for remote host interface.

clicast option VCAST_RHI_HOSTNAME <hostname or IP address>

Remote hostname or IP address to use for remote host interface.

clicast option VCAST_GH_PP_COMMAND <full path to Ada preprocessor command with arguments>

Assign a preprocessor for VectorCAST to use on Ada files when using the Green Hills compiler.

clicast option VCAST_USE_GH_BLD_FILE <.BLD filename>

Set this option to an appropriate .BLD file so that VectorCAST will compile and link using the "Build" command instead of compiling each file separately.

clicast option EXECUTABLE_EXTENSION <extension>

File extension of executables created by VectorCAST.

clicast option VCAST_ADA_FILE_EXTENSIONS <ada file extensions>

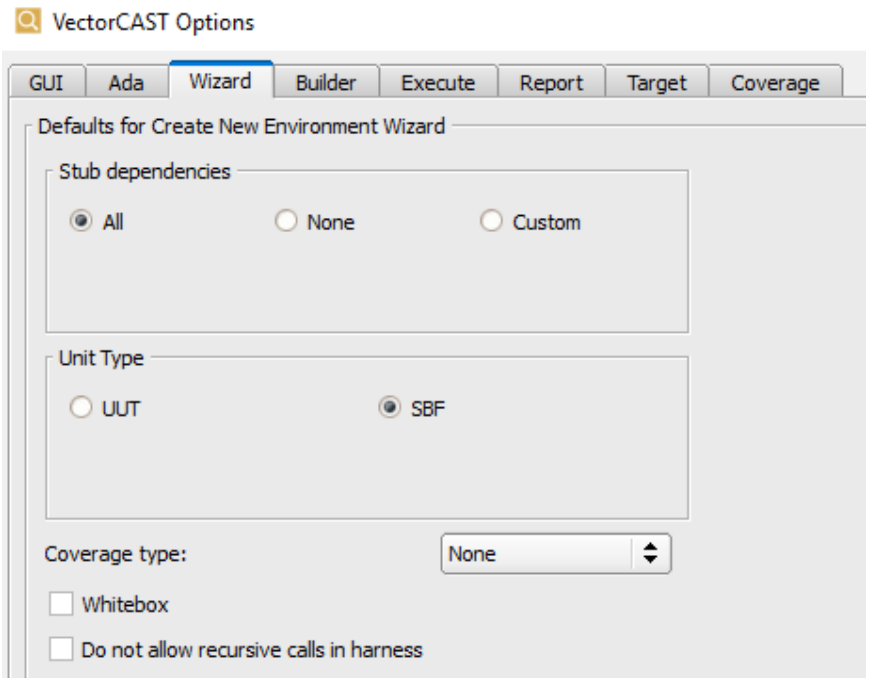
Use this option to specify the ada file extensions.

```
clicast option VCAST_SPEC_FILE_EXTENSIONS <spec file extensions>

Use this option to specify the ada spec file extensions.
```

Options: Wizard Tab

To Set Defaults for the Wizard

This section of the Tools Options enables you to choose a default value for some of the Create New Environment wizard settings. In all cases, values you choose here can be overridden in the wizard. A full description of each of these options and how it affects the building of an environment is discussed in the section Using the Wizard to Create an Environment.



Stub Dependencies

Choose **Tools => Options**, and click the **Builder** tab.

The Stub Dependencies feature provides the following choices for the default stubbing type displayed in the Create New Environment wizard:

- All
- None
- Custom

```
clicast -lada option STUB_DEPENDENCIES Yes | No | Custom

Default setting for Stub Dependencies option in the environment builder.
```

```
ALL means stub all dependencies, NONE means do not stub any dependencies.
Its default value is YES (All).
```

Unit Type

Choose **Tools => Options**, and click the **Wizard** tab.

The default setting for Unit type in the Create New Environment wizard. Choose one of the following:

- UUT – allows stubbing only for functions external to the unit being tested
- SBF (default) – allows stubbing of functions within the unit being tested

```
clicast -lada option VCAST_UNIT_TYPE UUT | SBF

The default setting for Unit type in the Create New Environment wizard. UUT
allows stubbing only for functions external to the unit being tested. Stub
By Function (SBF) allows stubbing of functions within the unit being
tested. The default value is SBF.
```

Coverage Type

Choose **Tools => Options**, and click the **Wizard** tab.

This option sets the type of coverage instrumentation to perform immediately after building the environment. By setting this option to a value other than None, coverage will be initialized each time you build a new environment.

The choices provided in the context menu are dependent upon the current Industry Mode (see "To Set the Industry Mode for Coverage" on page 27 of this manual). If no Industry Mode is set, the Default set of coverage types are used.

Here is a list of coverage types listed by Industry Mode and the clicast commands to set them:

Avionics (DO-178 B/C)

- Level A (Statement, Branch, MC/DC),
- Level B (Statement, Branch)
- Level C (Statement)

```
Clicast -lc option COVERAGE_TYPE None | LEVEL_A | LEVEL_B | LEVEL_C

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | LEVEL_A | LEVEL_B | LEVEL_C

Automotive (ISO-26262)

- Unit Level ASIL D (Statement, Branch, MC/DC)
- Unit Level ASIL B/C (Statement, Branch)
- Unit Level ASIL A (Statement)

```
clicast -lc option COVERAGE_TYPE None | ASIL_D | ASIL_B/C | ASIL_A

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | ASIL_D | ASIL_B/C | ASIL_A

Industrial (IEC-61508)

- SIL4 (Statement, Branch, MC/DC),
- SIL3 (Statement, Branch)
- SIL 1/2 (Statement)

```
clicast -lc option COVERAGE_TYPE NONE | SIL_4 | SIL_3 | SIL_1/2

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | SIL_4 | SIL_3 | SIL_1/2

Railway (EN-50128)

- SIL4 (Statement, Branch, MC/DC),
- SIL3 (Statement, Branch)
- SIL 1/2 (Statement)

```
clicast -lc option COVERAGE_TYPE None | SIL_4 | SIL_3 | SIL_1/2

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | SIL_4 | SIL_3 | SIL_1/2

Medical (IEC-62304)

- Class C (Statement, Branch, MC/DC)
- Class B (Statement, Branch)
- Class A (Statement)

```
clicast -lc option COVERAGE_TYPE None | CLASS_C | CLASS_B | CLASS_A

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | CLASS_C | CLASS_B | CLASS_A

Default

- Statement
- Branch
- Basis Paths
- MC/DC
- Statement+Branch
- Statement+MC/DC

```
clicast -lc option COVERAGE_TYPE
  None | STATEMENT+MC/DC | STATEMENT+BRANCH | MC/DC | BASIS_PATHS | BRANCH
| STATEMENT

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | STATEMENT+MC/DC | STATEMENT+BRANCH | MC/DC | BASIS_PATHS | BRANCH | STATEMENT

Whitebox

Choose **Tools => Options**, and click the **Wizard** tab.

Default settings for the **Whitebox** checkbox in the Create New Environment wizard.

```
clicast -lada option WHITEBOX Yes | No

Default settings for the "Whitebox" checkbox in the environment builder.
Its default value is YES (whitebox) for C environments, NO (blackbox) for
other environments.
```

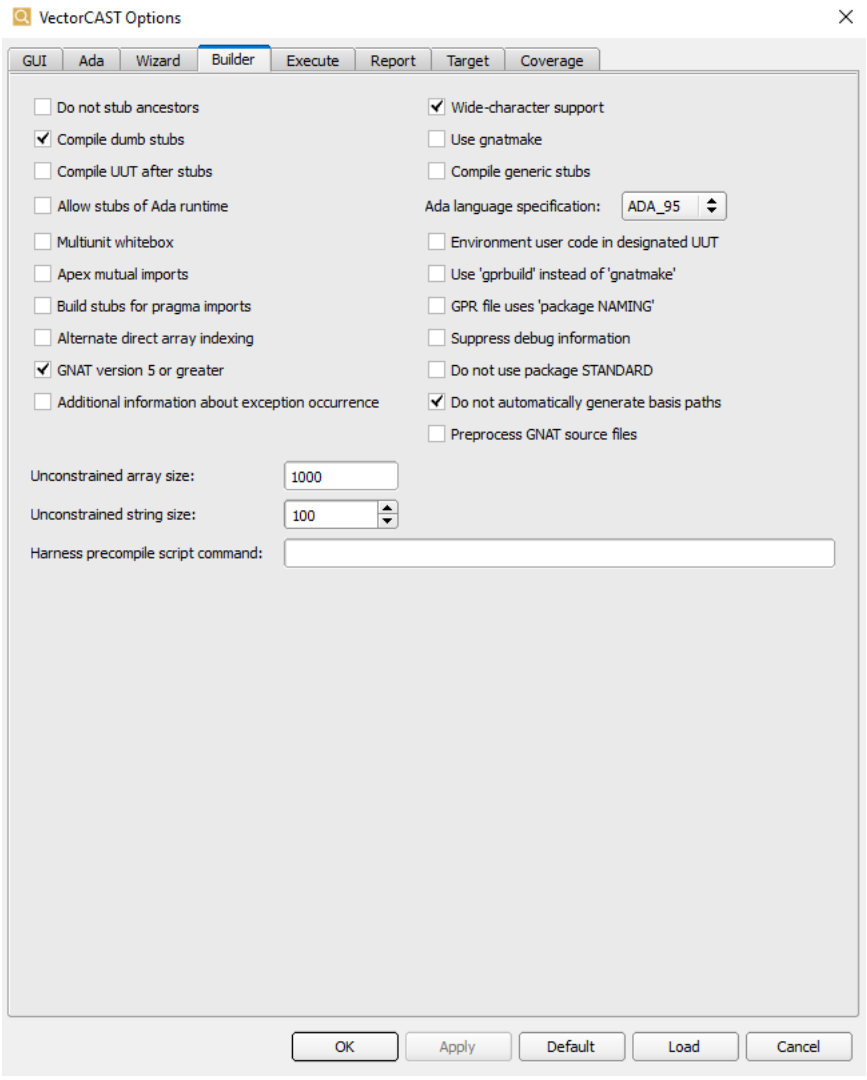
Once the environment is built, the environment script has the line: ENVIRO.WHITE_BOX: YES | NO.

Setting Builder Options

Options: Builder Tab

Choose **Tools => Options** and click the **Builder tab**.

The Builder options relate to building a new environment or rebuilding an environment.



RGW Database Location

```
clicast Option VCAST_REPOSITORY <directory path>

<directory path> is the path to a directory that contains the Requirements
Gateway Database. It has no default value.
```

Do Not Stub Ancestors

If this option is set, VectorCAST will not prompt for stubbing ancestor packages of the Unit Under Test. Also, VectorCAST will add any sub-programs defined in the ancestors to the sub-program list of the unit under test.

```
clicast option VCAST_DONT_STUB_ANCESTORS True | False

Do not prompt for stubbing ancestors of the UUT. Its default value is
false.
```

Compile Dumb Stubs

Choose **Tools => Options**, and click the **Builder tab**.

When a dependent unit is stubbed, VectorCAST creates a “dumb” stub for any unit “withed” by the stubbed units’ specification. By setting this option, these “dumb” bodies will be compiled and linked into the harness, allowing the execution closure to be completed with less effort.

```
clicast option VCAST_COMPILE_DUMB_STUBS True | False

Default value is true.
```

Compile UUT After Stubs

Choose **Tools => Options**, and click the **Builder tab**.

This option tells VectorCAST that you want to recompile the body of the Unit Under Test before linking, but after all stubs are compiled. This is useful when the Unit Under Test calls in-lined subprograms from stubbed units.

```
clicast option VCAST_RECOMPILE_SEPARATES True | False

Recompile the body of the UUT before linking, but after all stubs are
compiled. Default value is false.
```

Allow Stubs of Ada Runtime

Set this option if you want to enable stubbing of the standard Ada packages (TEXT_IO, SYSTEM, etc).

```
clicast option VCAST_ALLOW_STUB_OF_STD_LIB True | False

Default value is false.
```

Multiunit Whitebox

Choose **Tools => Options**, and click the **Builder tab**.

Set this option to allow manipulation of private types from all units in the environment, not just the Unit Under Test.

If Multiunit Whitebox is not set, VectorCAST builds dumb stubs for private ancestors of the UUT. Therefore, you will not be able to set any values for the parameters in the stub. If Multiunit White box is set, VectorCAST build smart stubs for all private units, including ancestors of the UUT.

```
clicast option VCAST_NEW_ARCHITECTURE True | False

Set this option to TRUE to enable manipulation of private types from all
units in the environment, not just the UUT. The default value is false.
```

Apex Mutual Imports

Choose **Tools => Options**, and click the **Builder tab**.

When this option is set, VectorCAST will process source files found in the mutual imports of the parent view.

```
clicast option VCAST_APEX_MUTUAL_IMPORTS True | False

Default value is false.
```

Build Stubs for Pragma Imports

Choose **Tools => Options**, and click the **Builder tab**.

This option instructs VectorCAST to build stubs for subprograms that are defined using pragma import. The pragma tells the compiler and linker that the source code for this subprogram is provided by an external source; you cannot supply an Ada body for this subprogram. VectorCAST builds the stub for the imported routine and then exports it as appropriate. This option enables you to manipulate the parameters of an imported subprogram as though the subprogram were a native Ada subprogram.

```
clicast option VCAST_PRAGMA_IMPORT True | False

This option tells VectorCAST to build stubs for subprograms that are
defined using 'pragma import'. The default value is false.
```

Alternate Direct Array Indexing

Choose **Tools => Options**, and click the **Builder tab**.

When this option is True, VectorCAST builds the harness using a different mechanism which allows VectorCAST to get the appropriate index for almost every array, regardless of target limitations.

```
clicast option VCAST_TARGET_DIRECT_ARRAY_INDEXING True | False

This option is used to build the harness so that direct array indexing can
be used on targets that do not support 'image' and 'value'. The default
value is false.
```

GNAT version 5 or Greater

Choose **Tools => Options**, and click the **Builder tab**.

Newer versions of GNAT use different syntax for the ‘gnatchop’ command. Disable this option if ‘gcc --version’ returns 3.4.3 or earlier.

```
clicast -lada option VCAST_USING_GNAT_5 True | False

Indicates if your version of gcc is later than 3.4.4 Default value is TRUE.
```

Additional Information about Exception Occurrence

Choose **Tools => Options**, and click the **Builder tab**.

Set this option to cause VectorCAST to cause ADA.EXCEPTIONS.EXCEPTION_MESSAGE to be called when an exception is propagated out of a UUT call. Text generate by the Ada run-time is included in the Execute Results report such as:

```
Exception name: CONSTRAINT_ERROR
Message: B0000002.ADA:1824.
```

Be sure to have the Execute option “Redirect standard output” on.

```
clicast -lada option VCAST_EXTENDED_EXCEPTION_INFORMATION True | False

 When an exception is propagated out of a UUT call, setting this flag to
TRUE will cause ADA.EXCEPTIONS.EXCEPTION_MESSAGE to be called. Default
value is false.
```

Unconstrained Array Size

Chose **Tools => Options**, and click the **Builder tab**.

This option sets the default size of "extern" array objects in which the index range is not specified. For example, if VectorCAST encounters an extern (e.g. **`extern int A[];`**) and does not find the actual definition of A, it generates a definition of the size specified in this option.

```
clicast option UNCON_ARRAY_SIZE <integer number> | <integer_
number>:<integer number>

Default size for unconstrained arrays where the index range is unknown or
very large. May be specified as [lower bound]:[upper bound] or just [upper
bound]. The default value is 10 for upper bound.
```

Unconstrained String Size

Choose **Tools => Options**, and click the Builder tab.

This option provides an upper limit for the length of an unconstrained string parameter.

```
clicast option VCAST_UNCONSTRAINED_STRING_SIZE <length>

Maximum length for an unconstrained string parameter. The default value is
100.
```

Harness Precompile Script Command

Choose **Tools => Options**, and click the Builder tab.

This command is executed prior to full compilations of the test harness. It can be used to make modifications to harness files before the initial compilation. Note that this command may also be called for subsequent harness compilations, but it is not called before every compilation.

```
clicast -lada option VCAST_PRECOMPILE_SCRIPT <script>

This command is executed prior to full compilations of the test harness. It
can be used to make modifications to harness files before the initial
compilation. Note that this command may also be called for subsequent
harness compilations, but it is not called before every compilation.
```

Wide-character Support

Choose **Tools => Options**, and click the Builder tab.

This option disable support for wide character when parsing Ada source code for environment building and coverage instrumentation. When set, this option causes VectorCAST to parse Ada source code using ADA.WIDE_TEXT_IO. When this option is off, VectorCAST reverts to using ADA.TEXT_IO.

If your source code contains the form feed character, turn this option off to properly build and instrument an Ada unit test environment.

```
clicast option VCAST_WIDE_CHARACTER_SUPPORT True | False

Allow VectorCAST to parse Ada code containing wide characters by using
ADA.WIDE_TEXT_IO when reading code. When this option is off, VectorCAST
reverts to using ADA.TEXT_IO. The default value is True.
```

To disable wide-character support in order to instrument Ada source code with a formfeed character in a Cover environment, use the clicast-only version of the command:

```
clicast -lada option VCAST_WIDE_CHARACTER_SUPPORT FALSE
```

Use Gnatmake

Choose **Tools => Options**, and click the **Builder tab**.

This option forces VectorCAST to use gnatmake to compile, bind, and link the test harness. If this option is not set, VectorCAST will use “gcc” to compile the test harness files, “gnatbind” to bind, and “gnatlink” to link.

```
clicast option VCAST_USE_GNATMAKE True | False

Default value is false, use “gcc”.
```

Compile Generic Stubs

Choose **Tools => Options**, and click the **Builder tab**.

If you set this option, VectorCAST will compile stubs that are created for generic packages.

```
clicast option VCAST_COMPILE_GENERIC_STUBS True | False

Default value is false.
```

Ada Language Specification

Choose **Tools => Options**, and click the **Builder tab**.

Set this option to tell VectorCAST to generate either Ada83 or Ada95 compliant test harness.

```
clicast -lada VCAST_ADA_LANGUAGE_SPECIFICATION ADA_95 | ADA_83

Default value is ADA_95.
```

Environment User Code in Designated UUT

Choose **Tools => Options**, and click the **Builder tab**.

This option allows environment user code to be placed in any UUT.

When this option is False, VectorCAST inserts the Environment User Code using the old methodology (first UUT). When the option is True and the environment is whitebox, environment user code is parsed / stored as follows:

- Any code at the beginning of the section is written to the USER_CODE_ADACAST package
- When the token **--VCAST_UNIT: <unitname>** is found in the Environment User Code, code

after that line is written to the appropriate unit body user code section.

> If **<unitname>** is not found, code is written to the last valid package

An example of Environment User Code (with line numbers):

```
1 ENVIRO.USER_CODE_ONE_SHOT_INIT:
2 USER_GLOBALS_VCAST.INT1 := 1234;
3 --VCAST_UNIT: UUT1
4 UUT1_SPEC := 100;
5 UUT1_BODY := 1000;
6 --VCAST_UNIT: UUT2
6 --VCAST_UNIT: UUT2
8 UUT2_BODY := 2000;
9 ENVIRO.END_USER_CODE_ONE_SHOT_INIT:
```

Line 2 is inserted into the "Harness Initialization" subprogram in the common user code package.

Lines 4 and 5 are inserted into the "Harness Initialization" subprogram in the first UUT.

Lines 7 and 8 are inserted into the "Harness Initialization" subprogram in the second UUT.

clicast -lada option VCAST_UNIT_SPECIFIC_USER_CODE_INSERTION True|False

In a whitebox environment, when this option is True, environment user code is inserted into the specified unit based on finding the flag '--VCAST_UNIT: <unitname>'. Lines with no designation preceding them are inserted into the common user code package. If the option is False, all environment user code is inserted into the first UUT. Default value is False.

Use 'gprbuild' Instead of 'gnatmake'

Choose **Tools => Options**, and click the **Builder tab**.

This option causes the Ada environment builder to use gprbuild instead of gnatmake to compile and link the test harness for Ada environments built with GPR files. Using this option may slow the build process, as this command compiles objects in other languages that are part of your source code build process.

clicast -lada option VCAST__USE_GPRBUILD True|False

This option forces VectorCAST to use 'gprbuild' to compile/link environments built from GPR files. This may slow the build process, as this command will compile objects in other languages that are part of the build process. Default value is False.

GPR File Uses 'package NAMING'

Choose **Tools => Options**, and click the **Builder tab**.

This option allows the GNAT Project file to use 'package NAMING' to override GNAT's default file naming convention.

```
clicast -lada option VCAST_GPR_USES_NAMING True | False

The GNAT Project file uses 'package NAMING' to override GNAT's default file
naming conventions. Default value is false.
```

Suppress Debug Information

Choose **Tools => Options**, and click the **Builder tab**.

Set this option to cause VectorCAST to compile source code files without debug information.

```
clicast option VCAST_SUPRESS_DEBUG_FLAG True | False

Default value is false.
```

Do Not Use Package STANDARD

Choose **Tools => Options**, and click the **Builder tab**.

By default, the Multiunit Whitebox test harness redefines TRUE and FALSE by using STANDARD.BOOLEAN. Setting this option disables that override, necessary when user source files use ‘STANDARD’ in a different context.

```
clicast -lada option VCAST_NO_STANDARD_PKG_USAGE True | False

Disable the test harness’ use of STANDARD.BOOLEAN when using multiunit
whitebox. Default value is false.
```

Do Not Automatically Generate Basis Paths

Choose **Tools => Options**, and click the **Builder tab**.

When this option is set, the environment builder does not generate the basis paths information automatically, saving some time during environment build. The complexity column in the Metrics report indicates "(not computed)" and all basis path report items are gray. Once the environment is built, the basis paths can be computed at any time by choosing **Tools => Basis Path => Generate**.

```
clicast option VCAST_DISABLE_AUTO_BASIS_PATH_GEN True | False

Specifies whether basis paths should be computed at build time. Its
```

Preprocess GNAT Source Files

Choose **Tools => Options**, and click the **Builder tab**.

This option enables preprocessing of the Unit Under Test (and any non-stubbed unit instrumented for code coverage) when the preprocessor is configured in the GNAT Project file (.gpr) specified when building the environment.

```
clicast option VCAST_PREPROCESS_ADA_FILES True | False

This option allows VectorCAST to use the GPR file to preprocess the Unit(s)
under test before parsing or instrumenting. Its default value is False.
```

Deprecated Builder Options

RANGE_FILE_MAX_SIZE was deprecated in VectorCAST version 4.1. However, it is still available as a CLICAST option.

VCAST_EXPAND_THRESHOLD was deprecated in VectorCAST version 4.0.

VCAST_USE_APEX_LINK was deprecated in VectorCAST version 4.0.

VCAST_FORCE_ADA83 was deprecated in VectorCAST 5.3. If VectorCAST detects VCAST_FORCE_ADA83: TRUE in a regression script or ADACAST_.CFG file, it is removed and VCAST_ADA_LANGUAGE_SPECIFICATION is set to ADA_83.

Force Ada83 Conformance

Choose **Tools => Options**, and click the **Builder** tab.

Set this option to tell VectorCAST that Ada83-compliant source code should be generated even though the compiler supports Ada 95.

```
clicast option VCAST_FORCE_ADA83 True | False

The default value is False.
```

Working with a Test Environment

To Open an Environment

There are several ways to open an environment.

- Click the **Open** button  in the toolbar or use **File => Open** if you need to navigate to find the environment you want.
- Choose **File => Recent Environments** to open a recently-opened environment from a list. The most recently-opened environment is at the top of the list. Coverage environments are shown with a green icon  ; Ada environments with an orange icon  ; and C environments with a blue icon  .
- Double-click the environment's icon (Windows)
  To use this method, the **VECTORCAST_DIR** environment variable must be set at the system level. The Windows installation program does this for you.
- Use the command line:

```
$VECTORCAST_DIR/vcastqt -e <env> (Linux)
%VECTORCAST_DIR%\vcastqt -e <env> (Windows)
```

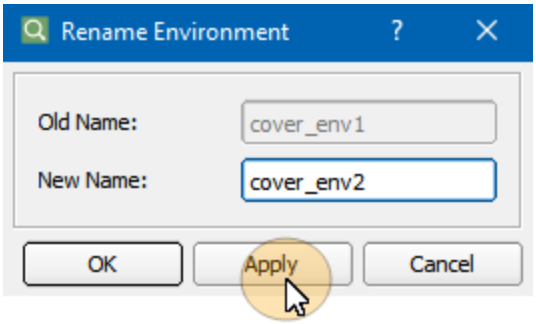
Once you open an environment, the working directory is set to the location of the newly-opened environment.

To Close an Environment

The **File => Close Environment** command closes the environment and any open files in the MDI window. If any file is unsaved, you are prompted to save before it closes.

To Rename an Environment

Choose **File => Rename Environment** to give the current open environment a new name. A dialog appears with the current name at the top and the new name is at the bottom. If the new name is not unique, then it is outlined in red. Type a new name and click **OK** or **Apply**.



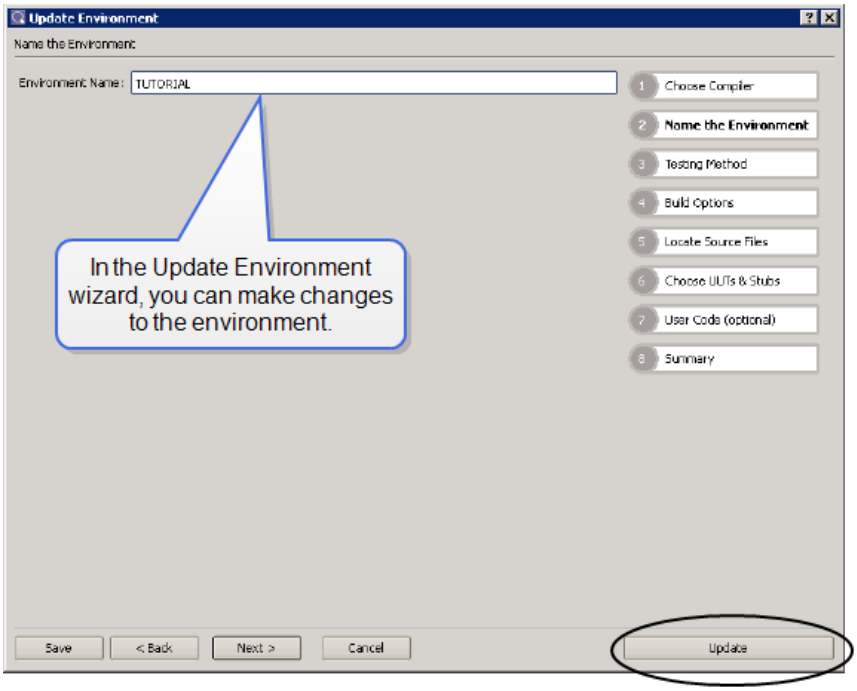
Once you click **OK** or **Apply**, VectorCAST renames the environment directory and the .vce file.

Note: The environment script is not changed when an environment is renamed.

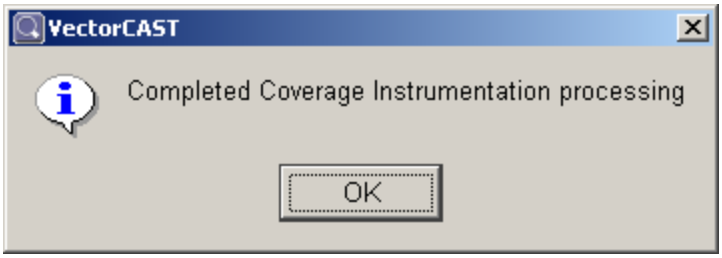
To Update an Environment

The Update Environment command provides the opportunity to easily make changes to an open environment, such as adding another UUT, adding a source directory, making the environment Whitebox, or changing the stubbing strategy. Updating the environment rebuilds the environment, and re-initializes coverage, if applicable.

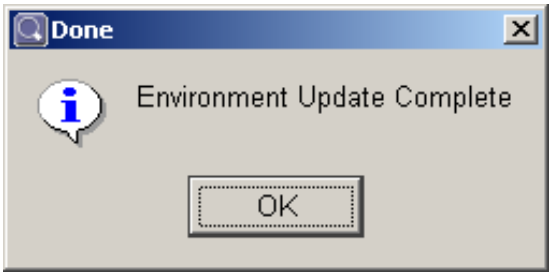
When you choose **Environment => Update Environment**, VectorCAST first creates environment and test case scripts for the existing environment. It then moves the existing environment to a backup directory. The Update Environment dialog appears, with all the information filled in for the current environment. Here you can enter new information or click to a different step. Note that the Load button is not available.



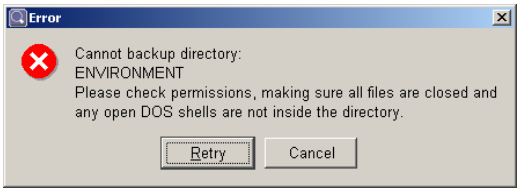
Click the **Update** button to begin building the environment with the changed settings. If coverage was initialized in the former environment, it is initialized in the updated environment, and the following message appears:



The following message indicates that the process finished successfully:



If there is some reason that VectorCAST cannot create the backup directory, then the user is alerted and offered the choice to **Retry** or **Cancel**.



Reasons that VectorCAST cannot copy the environment directory include:

- there is a DOS shell in the environment directory
- there is a file open in the environment directory
- the directory does not have write permissions
- there already is a file or directory(for rename) with the name *environment_name*.bak in the working directory
- the environment is on a mapped drive, and the copy process isn't quite finished when VectorCAST begins to rebuild

Once you have corrected the problem, click the **Retry** button and the rebuild process continues. Click the **Cancel** button to stop the rebuild process.

See also "Using the Wizard to Create an Environment" on page 99 for more information on the various steps in the wizard.

To Create Regression Scripts for an Environment

The **Environment => Create Regression Scripts** command creates a set of regression scripts in the directory you choose (excluding the environment directory).

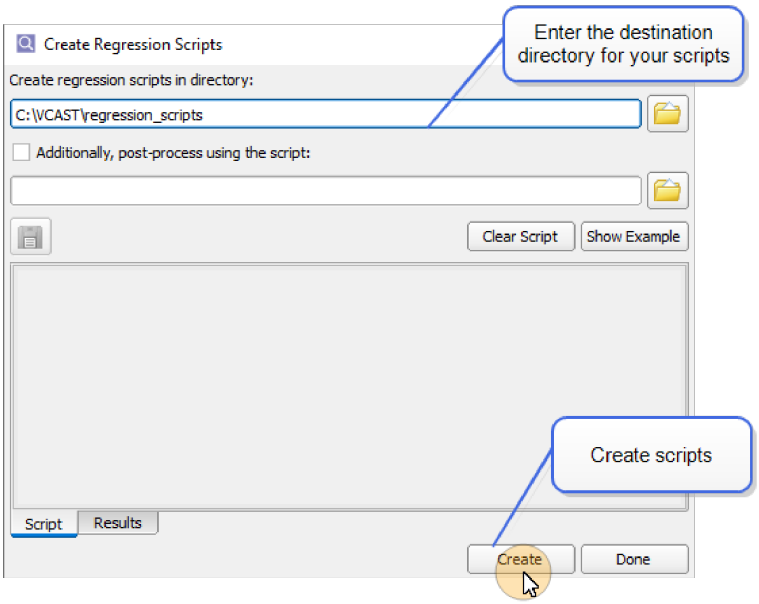
The files created are:

- a batch file *environment_name*.bat or shell script *environment_name*.sh (depending on your platform)
- an environment script with instructions on how to build the environment, *environment_name*.env
- the test script file *environment_name*.tst

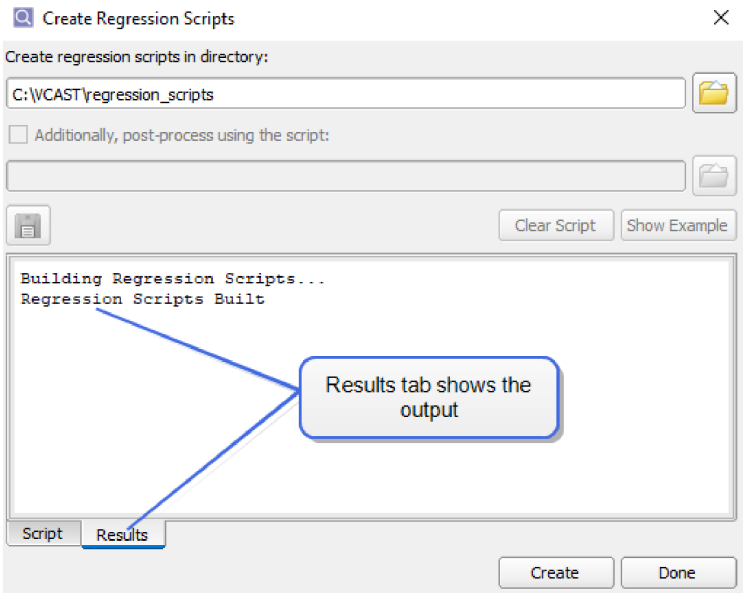
  If you have imported coverage results present in the environment, then an additional file is created:
- *environment_name*.cvr, which includes the imported coverage results themselves

This command enables you to generate the minimum set of files that allow for the regeneration of the tool options, test environment, and all test cases. The test environment directory does not need to be maintained when you are not actively testing. These regression files contain everything you need to recreate the test environment. You simply run the batch file or shell script and VectorCAST rebuilds the environment, loads and runs the test cases, and creates a test case management report (by default).

When you choose **Environment => Create Regression Scripts**, the following dialog appears.



The destination directory is relative to the current working directory. You can use relative or full paths. For example, entering "." causes the scripts to be created in the current working directory. Once a valid directory is entered, the Create button enables. Click **Create** to write the regression script files to the directory specified. The Create button is dimmed if you choose the environment directory. The Results tab shows that the scripts have been built successfully. Click **Done** to close the dialog.




```
clicast -e <env> TOols Regression_test [<post-processing script>]

Build the environment script, test script, and shell script (Linux) or
batch file (Windows) used to perform a complete regression test. The
```

```
optional post-processing script is invoked after the regression scripts
have been created.
```

When the regression script is run to recreate the environment, a temporary file is created, named **commands.tmp** This file contains the list of commands to build the environment, run all tests, and to create a management report. The last line of the regression script uses **commands.tmp** as an argument for the clicast command shown below. This clicast command runs all of the commands in a single invocation, using a single license checkout, and single "open" of the underlying project, resulting in enhanced performance.

```
clicast -lc -e <env> TOols EXEcute_commands <command file> [True|False]

Run multiple commands in one invocation. If the last argument is 'false',
then the commands will run to completion. Default behavior is to halt on
error.
```

The following are excerpts from the regression scripts for a Windows environment called TUTORIAL.

tutorial.bat (Windows)

```
del commands.tmp

echo clear_default_source_dirs >> commands.tmp

echo options TESTABLE_SOURCE_DIR

C:\VCAST\Environments\tutorial\ >> commands.tmp

echo environment build TUTORIAL.env >> commands.tmp

echo /E:TUTORIAL tools script run TUTORIAL.tst >> commands.tmp

echo /E:TUTORIAL execute batch >> commands.tmp

echo /E:TUTORIAL reports custom management

TUTORIAL_management_report.html >> commands.tmp

"%VECTORCAST_DIR%"\CLICAST" /L:CPLUSPLUS tools execute commands.tmp

false
```

commands.tmp

```
 clear_default_source_dirs

options TESTABLE_SOURCE_DIR C:\VCAST\Environments\tutorial\environment
build TUTORIAL.env

/E:TUTORIAL tools script run TUTORIAL.tst

/E:TUTORIAL execute batch
```

```
/E:TUTORIAL reports custom management TUTORIAL_management_report.html
```

tutorial.env

```
ENVIRO.NEW

ENVIRO.NAME:TUTORIAL

ENVIRO.COVERAGE_TYPE: NONE

ENVIRO.STUB_BY_FUNCTION:database

ENVIRO.STUB_BY_FUNCTION:manager

ENVIRO.STUB_BY_FUNCTION:manager_driver

ENVIRO.STUB_BY_FUNCTION:whitebox

ENVIRO.WHITE_BOX:NO

ENVIRO.MAX_VARY_RANGE: 20

ENVIRO.STUB: ALL_BY_PROTOTYPE

ENVIRO.SEARCH_LIST: C:\VCAST\Environments\

ENVIRO.TYPE_HANDLED_DIRS_ALLOWED:

ENVIRO.LIBRARY_STUBS:

ENVIRO.LINK_OPTIONS:

ENVIRO.END
```

tutorial.tst

```
-- VectorCAST 5.3 (07/18/11)

-- Test Case Script

--

-- Environment : TUTORIAL

-- Unit(s) Under Test: database manager manager_driver whitebox

--

-- Script Features

TEST.SCRIPT_FEATURE:C_DIRECT_ARRAY_INDEXING

TEST.SCRIPT_FEATURE:CPP_CLASS_OBJECT_REVISION

TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT
```

```
TEST.SCRIPT_FEATURE:STANDARD_SPACING_R2

TEST.SCRIPT_FEATURE:OVERLOADED_CONST_SUPPORT

--

-- Test Case: (CL)MANAGER::PLACEORDER.001

TEST.UNIT:manager

TEST.SUBPROGRAM:(cl)Manager::PlaceOrder

TEST.NEW

TEST.NAME:(CL)MANAGER::PLACEORDER.001

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Table:1

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Seat:4

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Soup:Onion

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Salad:Caesar

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Entree:Steak

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Dessert:Pie

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Beverage:Wine

TEST.VALUE:whitebox.<<GLOBAL>>.
(cl).WhiteBox.WhiteBox.<<constructor>>.WhiteBox().<<call>>:0

TEST.END

--

... (many other test cases follow)
```

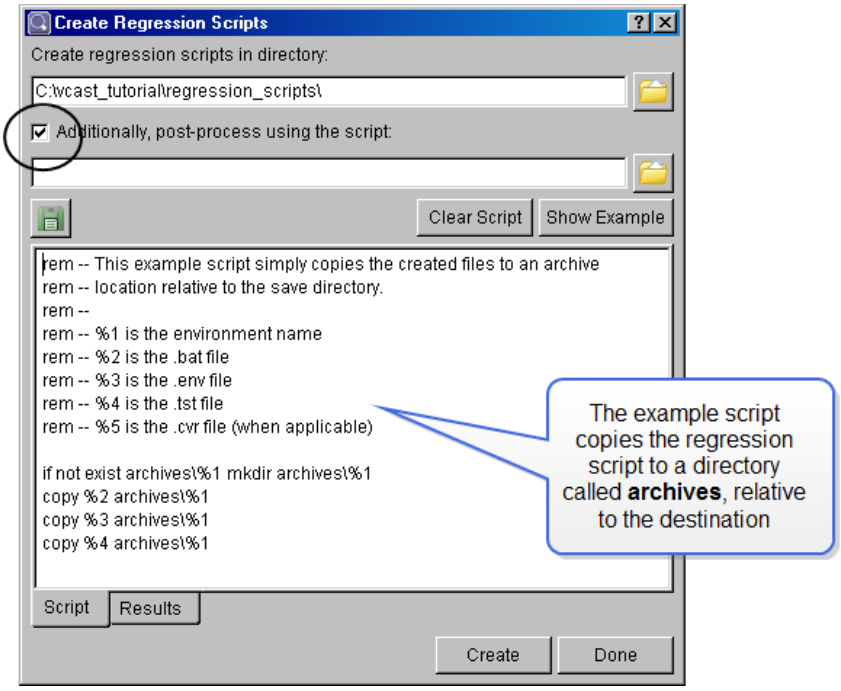
To Post-Process the Regression Scripts

The Create Regression Scripts dialog offers the ability to copy, archive, or otherwise post-process the regression scripts. This feature is useful in cases where you want to check the regression script files into your configuration management (CM) system, or do some other sort of post-processing.

You can then save these post-processing steps to a script or just perform them once. The following instructions explain how to perform post-processing on your regression scripts.

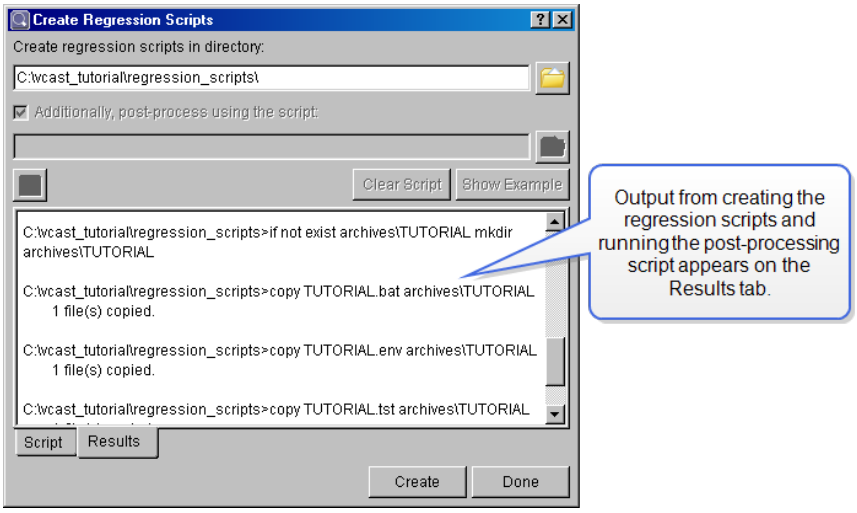
This functionality is controlled by the checkbox "**Additionally, post-process using the script**." After checking this option, you can then type or load a script into the Script tab. When you click the Create button, the regression scripts are created and then manipulated according to your instructions in the script.

To see an example script, click the **Show Example** button. An example batch file (Windows) or shell script (Linux) is displayed in the Script tab.



You can use this script, modify it, or clear it (click **Clear Script**) and write your own.

When you provide a post-processing script and click the **Create** button, the Results tab displays the standard output generated by post-processing the regression scripts. Note that the script itself is not saved by the Create action.




```
clicast -e <env> TOols Regression_test [<post-processing script>]

Build the environment script, test script, and shell script (Linux) or
batch file (Windows) used to perform a complete regression test. The
optional post-processing script is invoked after the regression scripts
have been created.
```

See also "To Save and Load a Post-Processing Script" on page 145 for information on saving to a script and loading in a script.

To Integrate Regression Scripts with ClearCase™

To integrate VectorCAST's regression scripts with IBM ClearCase, you use the Create Regression Scripts feature to create a script. Within the script, copy the regression script files to the directory from which you want to commit them, and then run the ClearCase commit call on these files.

1. With an environment open, choose **Environment => Create Regression Scripts**.
2. In the dialog box, choose the destination directory for the regression scripts.
3. Click the checkbox next to **Additionally, post-process using the script.**
4. Click the **Show Example** button to get started.
   The example script currently reads:

```
rem -- This example script simply copies the created files to an
rem -- archive location relative to the save directory.
rem --
rem -- %1 is the environment name
rem -- %2 is the .bat file (or .sh file on Linux systems)
rem -- %3 is the .env file
rem -- %4 is the .tst file
rem -- %5 is the .cvr file (when the environment has imported coverage
results)

if not exist archives\%1 mkdir archives\%1
copy %2 archives\%1
copy %3 archives\%1
copy %4 archives\%1
```

5. Change the script so that it runs the ClearCase commit call on the regression script files:

```
rem -- This example script simply copies the created files to an
rem -- archive location relative to the save directory.
rem --
rem -- %1 is the environment name
rem -- %2 is the .bat file
rem -- %3 is the .env file
rem -- %4 is the .tst file
rem -- %5 is the .cvr file

DEST_DIR = \home\user_name\project\regression\%1
if not exist $DEST_DIR mkdir -p $DEST_DIR
copy %2 $DEST_DIR
copy %3 $DEST_DIR
copy %4 $DEST_DIR
cd $DEST_DIR
cleartool commit *
```

6. To create the regression scripts and commit them using ClearCase, click the **Create** button.
   VectorCAST creates the three (or four) regression script files for the environment and post-processes them using the script you entered. You can save this script by clicking the **Save** button.
7. Close the dialog by clicking **Done**.

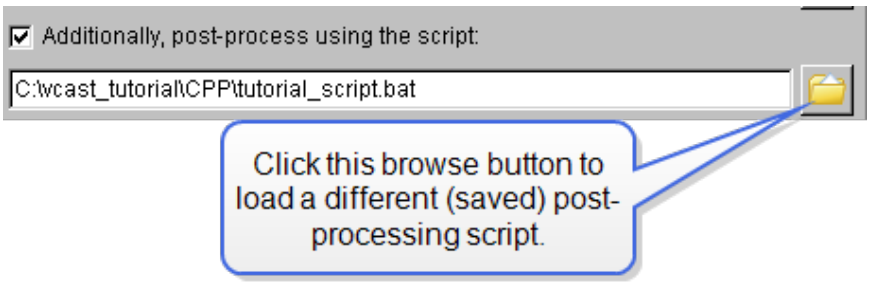
To Save and Load a Post-Processing Script

Once you have written a script in the Create Regression Scripts dialog, you can save it to use it again.

To do this, click the Save icon  . In the Save Script dialog, give the file a name and extension appropriate for your platform. Once saved, the name appears in the edit box below **Additionally, post-process using the script**.

The script name and its path is preserved in the registry (Windows) or in the .vcast-qt file (Linux). This file name is loaded in the Create Regression Scripts dialog as the default post-processing script. Once a post-processing script is saved, you can create regression scripts and post-process them with one click.

You can load an existing script file (such as a project-level file created by someone else) by using the browse button  to the right of the edit box for the script. The post-processing script you load in becomes the new default script.



See "To Post-Process the Regression Scripts " on page 142 to learn how to create a script that post-processes your regression script files.

To Incorporate Source Code Changes into Environment

The **Environment => Relink** command enables you to relink the object files of a previously-created test environment. This command should be used any time modifications are made to the source code of a unit under test. You would then recompile your source files, and choose **Environment => Relink** to link the test harness with the new object files. This command is useful when you are operating in a find-and-fix environment and debugging problems with a UUT.

To save you time, VectorCAST can relink both blackbox and whitebox testing environments, without requiring you to rebuild the whole environment after making changes to the source code. Keep in mind that if you make changes to the source code other than logic, such as changing subprogram names, or removing parameters, objects, or types, then compile errors may occur and you will need

to rebuild. If subprograms or objects are added, the environment will compile correctly, but these new items will not be visible. In this case also you will need to rebuild the environment (using **Environment => Rebuild Environment**).

Note: Before using the Relink command, the changed code must be compiled into the parent library manually.

Note: If the program specification for the unit under test or a dependent unit is modified, a new environment must be created.

```
clicast -e <env> ENvironment RELink

Re-link a previously created test environment.
```

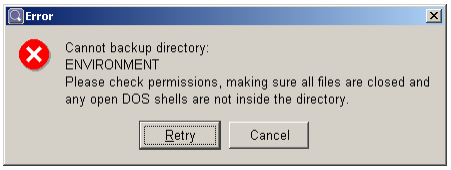
See also "To Rebuild the Environment" on page 146

See also "To Recompile the Test Harness" on page 149

To Rebuild the Environment

The **Environment => Rebuild** command completely rebuilds the test harness and associated data files (i.e. the environment). If you have changed any Builder options, VectorCAST uses the new settings. VectorCAST first creates environment and test case scripts for the existing environment. It then moves the existing environment to a backup directory, and then rebuilds the environment from the script and reloads the test cases. The backup environment directory name is *environment_name*.bak, where *environment_name* is the original environment name.

If there is some reason that VectorCAST cannot create the backup directory, then the user is alerted and offered the choice to Retry or Cancel.



Reasons that VectorCAST cannot copy the environment directory include:

- there is a DOS shell in the environment directory
- there is a file open in the environment directory
- the directory does not have write permissions
- there already is a file or directory(for rename) with the name *environment_name*.bak in the working directory
- the environment is on a mapped drive, and the copy process isn't quite finished when VectorCAST begins to rebuild

Once you have corrected the problem, click the **Retry** button and the rebuild process continues. Click the **Cancel** button to stop the rebuild process.

```
clicast -lada -e <env> ENvironment RE_Build

Rebuilds a previously built environment.
```

Troubleshooting Environment Rebuild

Some Other Application is Accessing the Environment Directory

On the Windows platform, if you have a Command Prompt (DOS) window open and in the environment directory, or if any other application is accessing files in the environment directory, then the error message “Cannot backup directory” appears. Change directories in the Command Prompt, or quit the application, and then try rebuilding the environment.

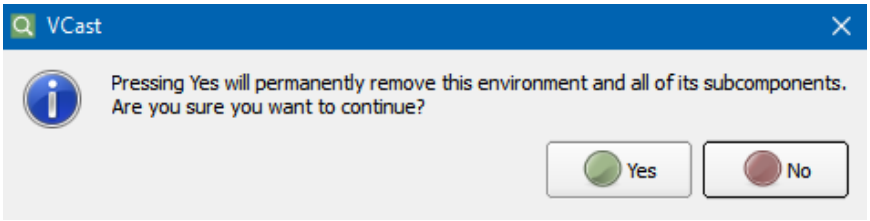
To Delete an Environment

The **File => Delete Environment** command deletes the current open environment or enables you to select an environment to delete. Deleting an environment deletes the sub-directory that was created for that environment and all files contained in the sub-directory. It leaves the .env file that can be used to re-create the environment later. Note that the test cases are not saved for you. To avoid losing any work, you should create Regression Scripts before deleting.



Note: The directory created by VectorCAST when building a new environment should only contain VectorCAST-created files. Do not store any files that are not tool-specific in these directories. When the **File => Delete Environment** command is invoked, all files contained in the specified directory are deleted.

If you wish to delete the current open environment, choose **Environment =>Delete Environment**. You are asked to confirm deleting the open environment.



If no environment is open when you choose **File => Delete Environment**, the standard Choose File dialog appears, providing the opportunity to navigate to find any environment to delete. Locate the .vce or .vcp file for the environment you want to delete. Click the **Open** button.

```
clicast -e <env> ENvironment Delete

Delete the specified environment.
```

To Re-create a Deleted Environment

To rebuild a deleted environment, choose **Environment => Scripting => Run**. Find the *environment_name*.env file of the environment you deleted. Click **Open**. VectorCAST rebuilds the

environment. The test case(s) are no longer present, unless you previously exported a test case script or created Regression Scripts.

```
clicast ENvironment SCript Run <scriptfile>

Build a new environment from the specified script file. You can also use
*.env for <scriptfile>. CLICAST is passed a command line with each
<filename>.env found in the directory. On Linux, to avoid having the
command line get too long after expansion, you can quote the argument to
clicast, using "*.env" for <scriptfile>. This form passes CLICAST the
actual wildcard string for it to expand.
```

Other Environment Tools

To Create an Environment Script

When an environment is built, an environment script is automatically generated. It is named *environment_name*.env, and stored in the working directory.

If you delete the environment script, it can be re-created using **Environment => Scripting => Create**. The menu item is dimmed unless you have an environment open. This command enables you to create a script file that has all the necessary information to rebuild the current environment, including user globals, stubbed units, and whitebox indication. To create a script, choose **Environment => Scripting => Create**. A Save File dialog appears, prompting you for a file name. Regardless of the file name, the ENVIRO.NAME line in the script reflects the name of the currently open environment.

```
clicast -e <env> ENvironment SCript Create <scriptfile>

Create an environment script file from an existing environment. If no
extension is provided, .env is used.
```

```
clicast -lada ENvironment SCript Quick <Ada_compiler> <Ada_library>

Build multiple environment script files based on an Ada library.
```

To Create an Environment by Running a Script

Running an environment script enables you to specify all environment build information in a text file, and then build the environment from that file.

If an environment is currently open when you choose this command, it is closed and the new one is built and opened.

When you choose **Environment => Scripting => Run**, the Open File dialog appears, prompting you to open an environment script file (.env). These files are created by VectorCAST, but sometimes you may want to copy one and slightly modify it as a text file. This command enables you to build an environment by running an environment script.

If you attempt to build an environment from a script and a directory already exists with the same name as the ENVIRO.NAME line in the script, the environment will not be overwritten and the build process

will terminate.

```
clicast ENvironment SCript Run <scriptfile>

Build a new environment from the specified script file. You can also use
*.env for <scriptfile>. CLICAST is passed a command line with each
<filename>.env found in the directory. On Linux, to avoid having the
command line get too long after expansion, you can quote the argument to
clicast, using "*.env" for <scriptfile>. This form passes CLICAST the
actual wildcard string for it to expand.
```

To Recompile the Test Harness

The **Environment => Recompile => Automatic** command is used to recompile and link all of the elements of an existing VectorCAST test harness (and instrumented test harness if you have code coverage initialized). This command is useful when a compiler macro definition has changed, causing different preprocessor behavior. The test harness needs to be recompiled to reflect this new behavior.

```
clicast -e <env> ENvironment RECompile Auto

Recompile and link all of the elements of an existing environment.
```

To Recompile the Instrumented Test Harness without Re-instrumenting It

If you have initialized code coverage for the environment, VectorCAST creates a modified version of the test harness with coverage instrumentation, and does so with every rebuild. If VectorCAST Technical Support instructs you to modify this instrumented file, the only way to recompile this file is with the Recompile Instrumented Code option (Recompile Automatic and Relink will cause the instrumentation to be performed again, thus removing your changes).

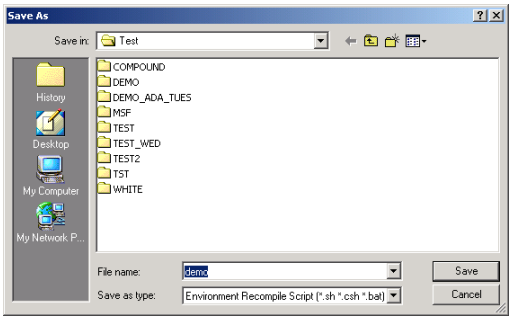
```
clicast -e <env> ENvironment RECompile Instrumented

Re-compile instrumented unit(s) and relink instrumented harness without re-
initializing coverage. This command is useful if you have made changes to
the instrumented harness that you do not want deleted.
```

To Create a Test Harness Recompile Script

The **Environment => Recompile => Create Script** command is used to create a compile script for all of the elements of an existing VectorCAST environment. These compile scripts are operating system dependent and can always be run outside of VectorCAST if desired. You may also add any additional commands or conditional logic to the scripts to accommodate your specific requirements. In addition, this feature enables you to view the exact compile and link commands that VectorCAST uses to build the harness executable; this is especially useful to diagnose when specific compile or link options are incorrect.

Note: No actual compiling is performed by this command. The commands that would normally be executed are dumped to a file. Execution of the script is performed with the **Environment => Recompile => Run Script** command.

Select **Environment => Recompile => Create Script** from the Main menu. Enter a filename to store the compile script. If you are running under Windows, give the filename an extension **`.bat`**. If you are running under Linux, give the filename the extension **`.sh`**. Click **Save**.



You will see this dialog box when the script has finished building.




```
clicast -e <env> ENvironment RECompile Create <scriptfile>

Create a script of compile and link commands used to build harness
executable. <scriptfile> is a required argument, specifying the filename.
```

Sample Re-Compile Script

```
cd \\\\dataserver\vcast_tutorial\ADA\TUTORIAL

rem # (S)

gnatchop -r -w vcast_ada_options.ads

rem # (B)

gnatchop -r -w vcast_ada_options.adb

rem # vectorcast_io (S)

gnatchop -r -w S0000007.ADA
```

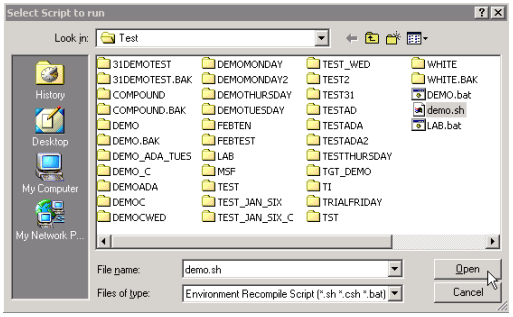
```
rem # vectorcast_io (B)
gnatchop -r -w B0000007.ADA
rem # user_globals_vcast (S)
gnatchop -r -w S0000008.ADA
rem # data_pkg_adacast (S)
gnatchop -r -w S0000002.ADA
rem # user_code_adacast (S)
gnatchop -r -w S0000004.ADA
rem # data_if_adacast_pkg (S)
gnatchop -r -w data_if_spec.ada
rem # data_pkg_adacast (B)
gnatchop -r -w B0000002.ADA
rem # data_if_adacast_pkg (B)
gnatchop -r -w B0000001.ADA
rem # uut_interface_adacast (S)
gnatchop -r -w S0000003.ADA
rem # user_code_adacast (B)
gnatchop -r -w B0000004.ADA
gcc -x ada -c -g -I\\\\dataserver/vcast_tutorial/ADA/ *.ads
gcc -x ada -c -g -I\\\\dataserver/vcast_tutorial/ADA/ *.adb
gnatbind -v -x -g -I\\\\dataserver/vcast_tutorial/ADA/   \
-aO\\\\dataserver/vcast_tutorial/ADA/uut_interface_adacast.ali
gnatlink -g -o UUT_INTE.EXE uut_interface_adacast.ali
rem # database (B)
gnatchop -r -w I0000009.ADA
rem # manager (B)
gnatchop -r -w I0000010.ADA
rem # uut_interface_adacast (S)
gnatchop -r -w S0000003.ADA
gcc -x ada -c -g -I\\\\dataserver/vcast_tutorial/ADA/ *.ads
```

```
gcc -x ada -c -g -I\\\\dataserver/vcast_tutorial/ADA/ *.adb

gnatbind -v -x -g -I\\\\dataserver/vcast_tutorial/ADA/  \

-aO\\\\dataserver/vcast_tutorial/ADA/  \

uut_interface_adacast.ali

gnatlink -g -aO\\\\dataserver/vcast_tutorial/ADA/  \

-o UUT_INST.EXE uut_interface_adacast.ali

\\\\dataserver

cd \vcast_tutorial\ADA\TUTORIAL
```

Recompile the Test Harness by Running a Script

The **Environment => Recompile => Run Script** command is used to execute a recompile script to update a VectorCAST environment.

If an environment is currently open, select **Environment => Recompile => Run Script**. The following dialog window opens, in which you may select the script you wish to run.



Once you have selected the script you wish to run, click Open. If you are on the Windows platform, VectorCAST opens a Command Prompt (DOS box) and runs the batch file. If you are on the Linux platform, VectorCAST runs a shell script. The Recompile is complete when you see the following text in the Message window:

```
Executing script <path to script file>
Script processing complete
```

The Scripting Log is displayed in the MDI window, and it shows the output from the recompile operation.

```
clicast -e <env> ENvironment RECompile Run <scriptfile>

Run script of compile and link commands to build harness executable.
```

See "To Create a Test Harness Recompile Script" on page 149 for information on how to create a recompile script.

To Relink an Environment

The **Environment => Relink** command enables you to relink the object files of a previously created test environment. This command should be used any time modifications are made to the source code of the unit under test. You would then recompile your source files, and choose **Environment => Relink** to link the test harness with the new object files.

```
clicast -e <env> ENvironment RELink

Relink a previously created test environment.
```

By making modifications to the original source code file, compiling the file, and re-linking the VectorCAST environment, you will be able to test your changes with your existing test cases. This command is useful when you are operating in a find-and-fix environment and debugging problems with a UUT.

If you make changes to the source code other than logic, such as changing subprogram names, or removing parameters, objects, or types, then compile errors may occur and you will need to rebuild. If subprograms or objects are added, the environment will compile correctly, but these new items will not be visible. In this case also you will need to rebuild the environment (using **Environment => Rebuild Environment**).



Note: Before using the Relink command, the changed code must be compiled into the parent library manually.



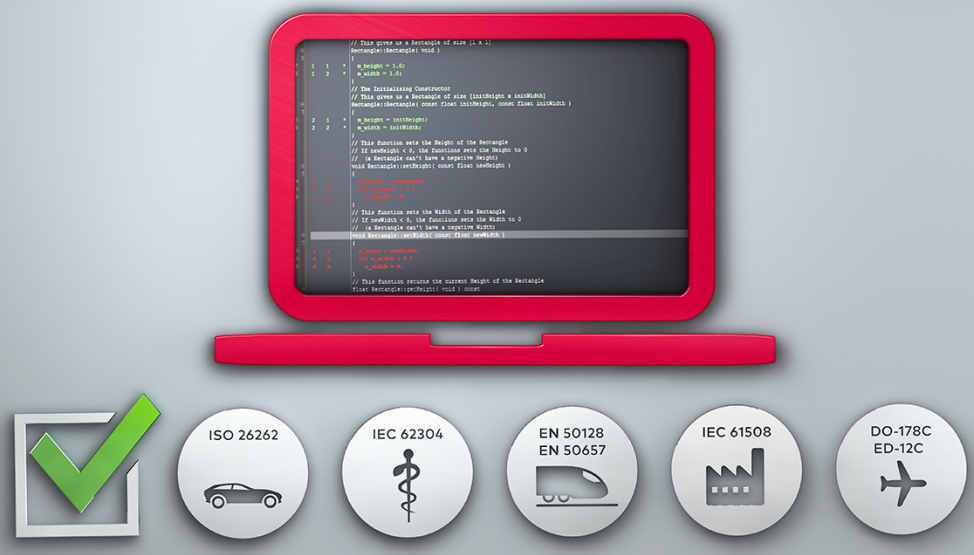
Note: If the program specification for a unit under test or a dependent unit is modified, a new environment must be created.

```
clicast -e <env> ENvironment Compile <unitname> spec | body

clicast -e <env> ENvironment Compile <source-file> file

Compile an individual unit or additional source file in environment <env>.
<unitname> is the name of a unit in the environment followed by the word
spec or body. If you want to compile a new source file into the
environment, then use its filename for <source-file>, followed by the word
file.
```

To Refresh Type Range Data

The **Environment => Rebuild Range Data** command is used to re-calculate the ranges for all types in the code under test. This command is used when changes have been made to the original unit under test, or any type it depends on.

```
clicast -lada -e <env> ENvironment REBuild_range_data

Regenerate typemark range data.
```



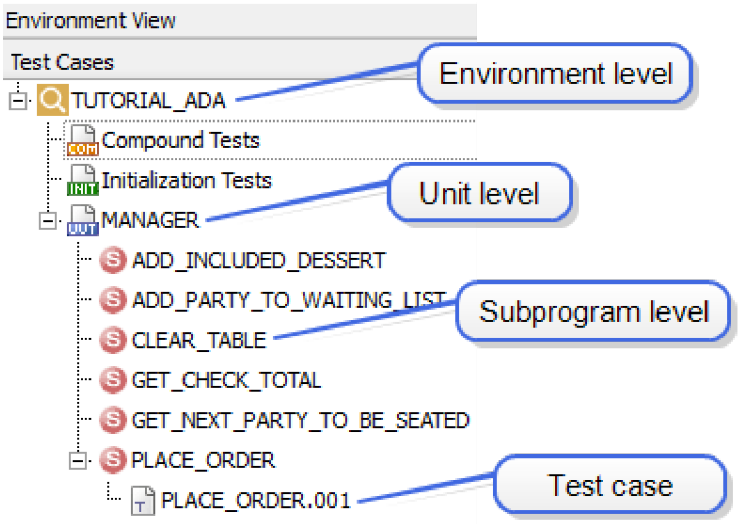
Building Test Cases

Using The Test Case Tree

The Test Case Tree Hierarchy

When an environment is opened in VectorCAST, the panel on the left (the *Test Case Tree*) displays the units, subprograms, and test cases in the environment in a hierarchy view.

Many of the menu items in the Test menu, Tools menu, and of course the right-click menu, operate on the item or items that are selected in the Test Case Tree.



At the top of the hierarchy is the environment name. Click this node if you want your next action to affect the whole environment.

Next are the <<COMPOUND>> and <<INIT>> items. Click one of these special items if you want your next action to affect only the COMPOUND test cases or INIT test cases, respectively.

The Units Under Test are listed at the next level of the hierarchy. They are listed alphabetically. Click this a unit node if you want to your next action to affect all subprograms in the unit.

The subprograms for the unit are listed below the unit in the hierarchy, in the order they are specified in the source code. Click a subprogram node if you want your next action to apply to that subprogram only.

The lowest level of the hierarchy is the test case level. Click a test case if you want your next action to apply only to that test case.

When selecting items in the Test Case Tree, you can combine items at different levels using Shift+click or Ctrl+click actions.

Naming of Overloaded Subprograms

For overloaded subprogram names, the list of parameter types will be appended to the subprogram

name to differentiate between overloaded subprograms. Example: store (integer), store (float), store (boolean).

To Navigate the Test Case Tree

In the Test Case Tree, the units, subprograms, and test cases are displayed hierarchically.

To expand a node:

- Mouse: click the  icon to the left of the node
- Keyboard: after placing the keyboard focus on that node, press the right-arrow key
- Menu: right-click the node and choose **Expand All Children**

To collapse a node:

- Mouse: click the  icon to the left of the node
- Keyboard: press the left-arrow key
- Menu: right-click the node and choose **Collapse All Children**

Additionally, use the keyboard's up- and down-arrow keys traverse the expanded nodes. You can also type the first few letters of an expanded node name to jump to that node.

To Multi-Select Test Cases

To select one test case in the Test Case Tree, simply click it. To select more than one, select one, then, holding down the Ctrl key, select the other test cases you want to execute, duplicate, delete, or modify. You can select a range of test cases by holding down the Shift key while clicking the second test case. Or, you can drag your cursor over a span.

The same process is used to multi-select subprograms, units, or a mixture of test cases, subprograms, and units.

Types of VectorCAST Test Cases

There are two types of VectorCAST test cases: simple test cases and compound test cases. A *simple test case* corresponds to a single invocation of a UUT. Simple test cases are primarily used to test the processing of a single subprogram within a UUT. The simple test case data is loaded, the subprogram being tested is invoked and results are captured. <<INIT>> test cases are a type of simple test case. They invoke the test harness, but do not call any subprograms. They are used primarily to perform initialization or to check final values of global data.

A *compound test case* is a collection of simple test cases that perform a series of calls. A compound test case provides the ability to invoke a UUT multiple times with a variety of data within a single execution. Each test case in a compound test case can be executed one or more times. Compound test cases can be used to test processing which is dependent on multiple calls to different procedures of a UUT. For example, you may have processing that implements a database with one subprogram to write to the database, and another subprogram to read from the database. In this case a compound test could be developed to invoke a call to the "write" subprogram, followed immediately by a call to the "read" subprogram. In this way you can verify the integrity of the stored data.

When executing a compound test case the Event header contains information about the compound test case that is executing, the slot number executing, its name, and its iteration. For example, the following Event header is displayed for event 1 when <<COMPOUND>>.001 is executed:

```
Event 1
<<COMPOUND>>.001 Slot 1 ((CL)MANAGER::PLACEORDER.001) Iteration 1
```

All simple test cases are associated with a subprogram of a Unit Under Test. Prior to building simple test cases, it is necessary to select a subprogram to test. The test cases built for this subprogram are applicable to this subprogram only.

When building a new test case, you may assign values to parameters of the subprogram under test, any parameters of the stubbed dependent subprograms, and any global data objects.

Values are entered for individual scalar data items. VectorCAST takes you through a simple-to-use tree expansion of the type mark for each data item until a scalar data type is encountered. You are then able to enter a value for the item directly, or double-click to bring up a dialog box that is specific to that scalar type.

VectorCAST can generate the following types of test cases:

- **Min Mid Max** - Min Mid Max tests stress a function at the bounds of the input data types. Min, Mid, and Max test cases are useful when unit testing because they often uncover hidden assumptions present in the code under test.
- **Partitioned** - Partitioned tests create "partitions" for each data type and test the min and max value from each partition. The assumption is that values from the same partition will stimulate the application in a similar way.
- **MC/DC** - MC/DC tests use the MC/DC analysis to examine the unique paths that exist through a procedure. MC/DC tests can automatically create a high level of path coverage.
- **Basis Path** - Basis Path tests use the basis path analysis to examine the unique paths that exist through a procedure. Basis Path tests can automatically create a high level of branch coverage.

To Open a Test Case for Editing

To open a test case in the TestCase Editor, select a test case name, right-click, and choose **Open Test Case**. Or, double-click the test case name.

The keyboard shortcut for Open Test Case is **Ctrl+Return**.

Test Case Naming

When building a new test case, a name is automatically created based on the name of the subprogram and a unique number:

Name of the Subprogram under test.number

For example, PLACE_ORDER.008 is the eighth test case for the subprogram Place_Order.

The two portions of the automatically generated name are separated by a period ('.'). Note that test case names are limited to 80 characters in length.

The automatically generated name may be edited or completely replaced by right-clicking on the test case name and choosing **Rename**, as described in "To Rename a Test Case" on page 158.

To Rename a Test Case

To rename a test case, select a test case, right-click and choose **Rename**. An edit box surrounds the old name. You can type or paste a new name in. Note that test case names are limited to 80 characters in length

Test case names must contain only alphanumeric characters or punctuation (i.e., hyphens, underscores, or periods). Percent, backslash and single or double quotes are not supported and cause a message to pop up and the new name to be discarded.

Press **Enter** to finalize the name change.

If you rename a test case that is part of a compound test case, VectorCAST renames it in the compound test automatically.

To Sort Test Cases Alphabetically

When a new test case is inserted, by default it is positioned as the last test case for that subprogram. When importing a test script, the test cases are positioned according to the order in the test script.

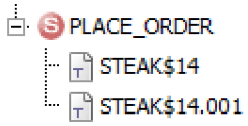
To sort the test cases alphabetically (for each subprogram), choose the **Tools => Options dialog, GUI**, and set the option “Alphabetize test cases” on. As a result, as each test case is created or imported, it is positioned in the list alphabetically, for each subprogram.

To Duplicate a Test Case

Once a test case is built and saved, the data that is contained in it can be copied to another test case. This allows you to clone an existing test case as the basis for a new test.

Select a test case to duplicate, right-click, and choose **Duplicate Test Case**. The new test case is automatically named based on the name of the original test case.

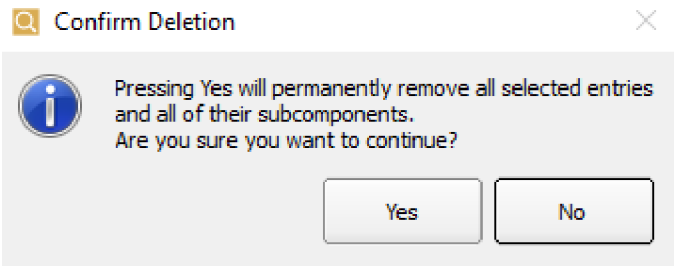
For example, if you duplicate a test case for the subprogram Place_Order named STEAK$14, then the new test case takes the name STEAK$14.001.



To Delete a Test Case

To delete a test case, select a test case in the Test Case Tree. Right-click and choose **Delete**.

The following confirmation dialog box appears:



Note that if a test case is used in a compound test case, then deleting the test case also affects the compound test case. Click **Yes** to delete the test case and remove it from any compound test cases. Click **No** to cancel the delete operation.

```
clicast -lada -e <env> -u <unit> -s <sub> -t <test> TESt Delete [ALL]

Delete the test case(s) specified via the unit, subprogram, and testcase
command line parameters. If no parameters are specified, the user must
specify ALL to confirm deletion of all testcases in the environment. If a
test being deleted is contained in a compound test, the user must specify
ALL to confirm deletion of the test in all locations.
```

To Delete All Test Cases from a Subprogram

To delete all test cases in a subprogram, select a subprogram in the Test Case Tree. Right-click and choose **Delete**.

```
clicast -lada -e <env> -u <unit> -s <sub> -t TESt Delete [ALL]

Delete the test case(s) specified via the unit and subprogram command line
parameters. If a test being deleted is contained in a compound test, the
user must specify ALL to confirm deletion of the test in all locations.
```

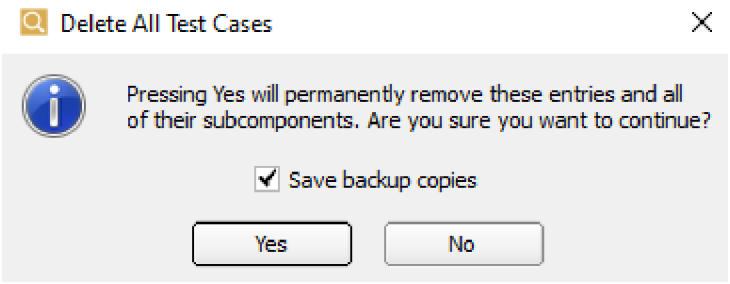
To Delete All Test Cases from a Unit

To delete all test cases from a unit, select a unit in the Test Case Tree. Right-click and choose **Delete**.

```
clicast -lada -e <env> -u <unit> TESt Delete [ALL]

Delete the test case(s) specified via the unit command line parameters. If
a test being deleted is contained in a compound test, the user must specify
ALL to confirm deletion of the test in all locations.
```

To Delete All Test Cases in the Environment

If you wish to delete all test cases from all subprograms, choose **Test => Delete All**. The following confirming dialog appears.



The “Save backup copies” option is selected by default as an added failsafe to prevent accidental deletion of test cases. This “backup script” is saved in the environment directory and named TEST_DELETE.tst.

Alternatively, right-click the top node of the Test Case Tree hierarchy (the environment name), and choose **Delete**. Doing so deletes the test cases from the environment level, which includes all test cases. You are asked to confirm the delete operation, but a backup file is not created.

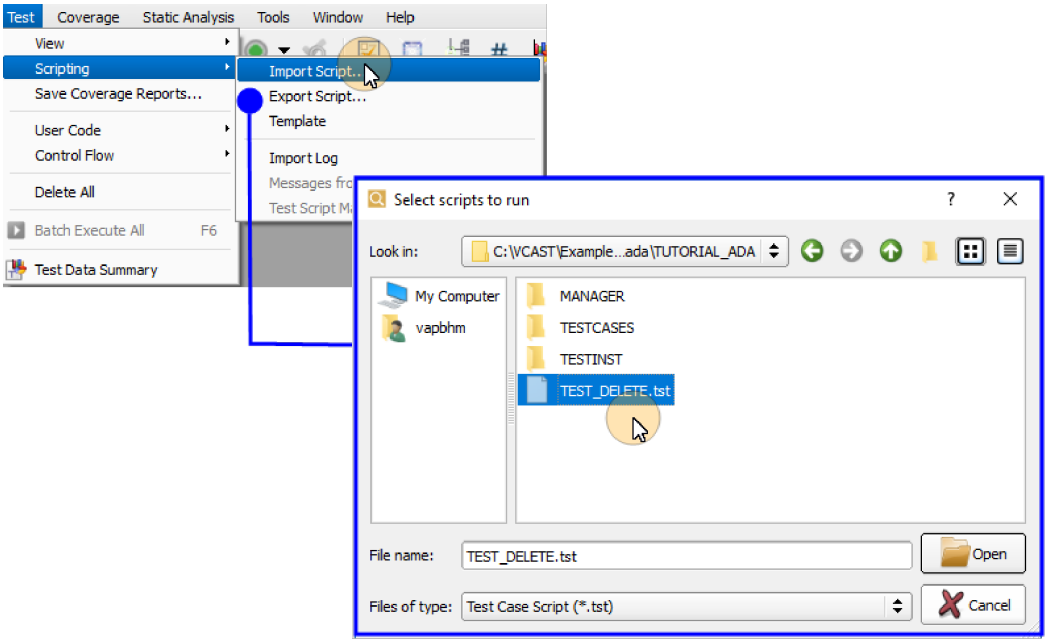
```
clicast -lada -e <env> TESt Delete [ALL]

Delete all test cases in the environment. The user must specify ALL to
confirm deletion of all testcases in the environment.
```

See "To Restore Deleted Test Cases" on page 160 for information on restoring test cases.

To Restore Deleted Test Cases

Only test cases that were deleted with the **Test => Delete All** command can be restored.

The Delete All Test Cases dialog has a checkbox labeled “Save backup copies.” By default, this checkbox is selected. When selected, a backup test case script is automatically generated to preserve the deleted test cases. This test script file is called **`TEST_DELETE.tst`** and is created inside the environment directory. If you need to retrieve the deleted test cases, simply import the test script (**Test => Scripting => Import Script => env_dir/TEST_DELETE.tst**) and all deleted test cases will be restored.



To View the Source for a Subprogram

To open the source file for a unit or subprogram, right-click the unit or subprogram node, and choose **Open Source**. The source file opens in a tab in the Text Editor group. If this action is performed on a subprogram node, then the cursor is placed at the beginning of the subprogram.

The keyboard shortcut for Open Source is **Ctrl+Return.**

To View Coverage for a Unit or Subprogram

To open the Coverage Viewer for a unit or subprogram in the Test Case Tree, right-click the unit or subprogram node, and choose **View Coverage**. The Coverage Viewer opens. If this action is performed on a subprogram node, then the Coverage Viewer is scrolled to the top of the subprogram.

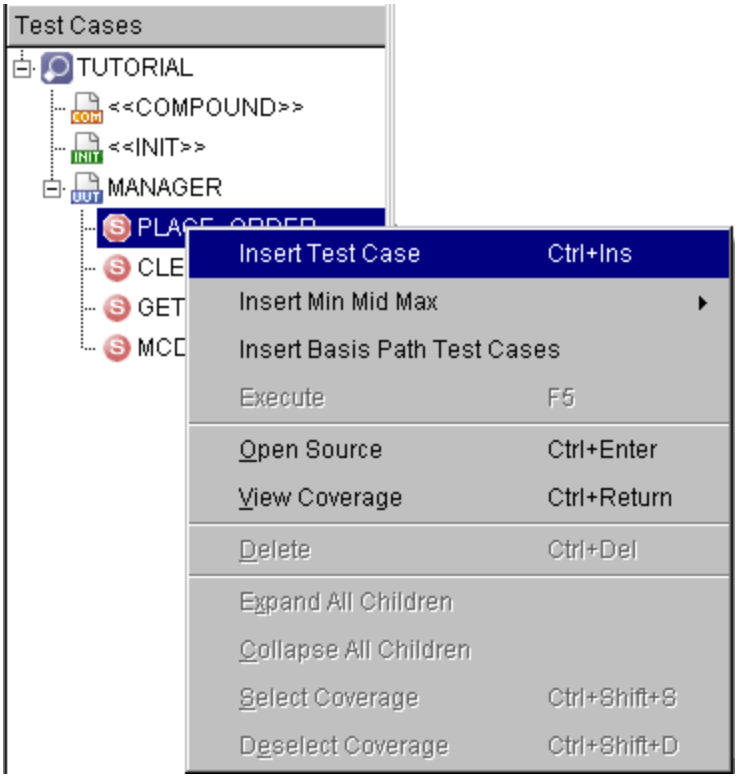
The keyboard shortcut for View Coverage is **Shift+Return**.

Simple Test Cases

To Insert a Test Case for a Subprogram

Select a subprogram, or multi-select several subprograms. Right-click, and choose **Insert Test Case**. A test case is inserted for each subprogram selected. By default, its name is "*subprogram*.001", where *subprogram* is the name of the subprogram.

The keyboard shortcut for Insert Test Case is **Ctrl+Insert**.



See"To Rename a Test Case" on page 158 on page for information on renaming a test case.

To Insert an <<INIT>> Test Case

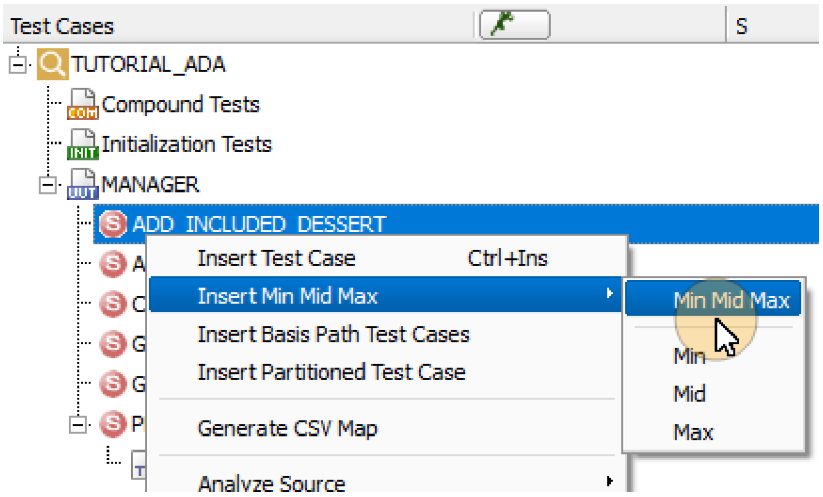
The <<INIT>> test case enables you to initialize global data within the Unit(s) Under Test. This test case invokes the test harness without calling any subprograms. Any initialization (global data initialization, class object construction) will take place before the test driver “main” is called.

Another useful feature for the <<INIT>> test case is as a setup step for compound test cases. You can set the <<INIT>> test case to initialize global data, and then include that test case as the first slot in a compound test. Doing so enables large amounts of initialization data to be entered one time for multiple compound tests.

To Create a Min, Mid, or Max Test Case

The Min-Mid-Max command provides an automated test case generation capability for creating test cases that set all parameter and global data values (excluding User Globals) to their minimum, maximum, or mid-point values. Min, Mid, and Max test cases are useful when unit testing because they often uncover hidden assumptions present in the code under test. For example, using an integer-typed data item to index into an array with only 10 elements without checking that the index value is within range.

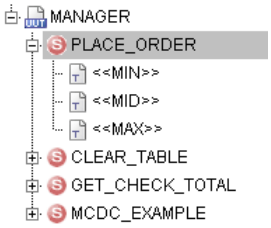
To create all three types, click a node in the Test Case Tree (subprogram, unit, or environment). Right-click and choose **Insert Min Mid Max** from the drop-down menu.



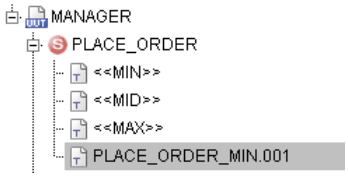

```
clicast -e <env> [-u <unit> [-s <sub>]] TESt Minmidmax

Build the <<MIN>>, <<MID>>, and <<MAX>> special test cases for the
environment, a specified unit, or a specified subprogram.
```

To create only one type (min, mid, or max), right-click and choose **Insert Min Mid Max** from the popup menu, and then choose **Min** (or **Mid** or **Max**) from the submenu.

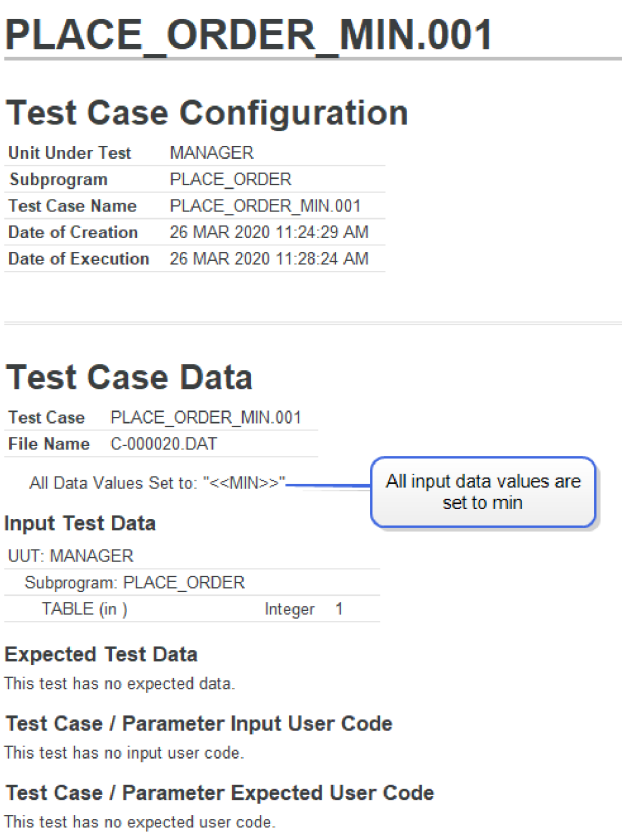
Min-Mid-Max test cases are given the special names <<MIN>>, <<MID>>, and <<MAX>> in the Test Case Tree. These special test cases cannot be edited, but you can derive new test cases from them.



To derive a test case from one of the special <<MIN>>, <<MID>>, or <<MAX>> test cases, double-click on it. A new test case is created. For example, double-clicking on <<MIN>> that appears below the subprogram Place_Order creates the test case PLACE_ORDER_MIN.001. In this test case, all input values for the subprogram and stubs are already set to the minimum value, but you can override specific values by entering data.



When you view the test case data listing, it indicates that all *input* data values are set to min, mid, or max, as shown in the following example.



PLACE_ORDER_MIN.001

Test Case Configuration

Unit Under Test	MANAGER
Subprogram	PLACE_ORDER
Test Case Name	PLACE_ORDER_MIN.001
Date of Creation	26 MAR 2020 11:24:29 AM
Date of Execution	26 MAR 2020 11:28:24 AM

Test Case Data

Test Case	PLACE_ORDER_MIN.001
File Name	C-000020.DAT

All Data Values Set to: "<<MIN>>"

All input data values are set to min

Input Test Data

UUT: MANAGER

Subprogram: PLACE_ORDER

TABLE (in) Integer 1

Expected Test Data

This test has no expected data.

Test Case / Parameter Input User Code

This test has no input user code.

Test Case / Parameter Expected User Code

This test has no expected user code.

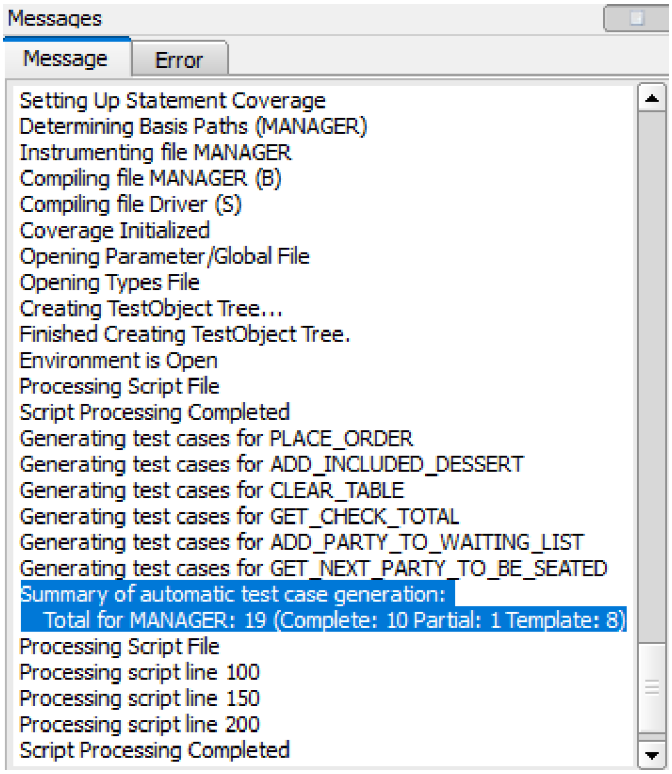
To Insert Basis Path Test Cases

To automatically generate the test cases needed to fulfill all the basis paths, click a node in the Test Case Tree (subprogram, unit, or environment). Right-click and choose **Insert Basis Path Test Cases**.

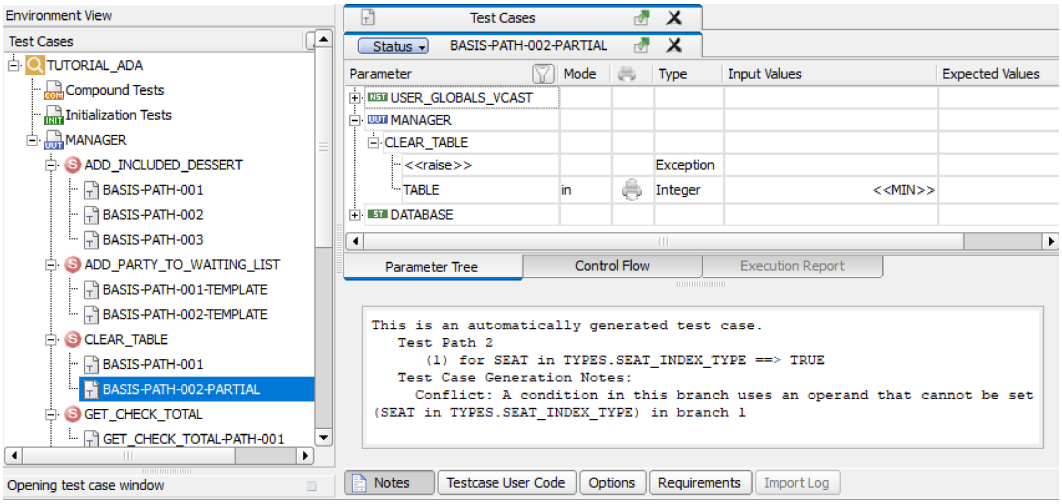
Test cases are added to the environment, named *BASIS-PATH-001,* -002, -003, etc., which correspond to the basis paths for the subprogram. Because of unreachable paths and because some conditions may be based on intermediate computations, the test cases that VectorCAST creates may not be complete. After the basis path test cases are created, a Test Script Log is displayed, showing the test case processing.

The Message window displays a summary of the automatic test case generation. The generated test cases are complete, partial, or template tests.

- Complete – the test case was completely generated, with an input value for each data item necessary to traverse the test path
- Partial – the test case was partially generated. The test case's name has "-PARTIAL" appended to it. The test case's Notes tab contains a full description of which portions of the test case are not complete and why. Example: a dependency on an intermediate computation (`c=foo(); if(c)`).
- Template – VectorCAST was unable to determine any information necessary to traverse the test path. The test case's name has "-TEMPLATE" appended to it. As with partial test cases, the test case Notes tab contains a full description of the reasons.



There are a number of reasons why VectorCAST may not be able to determine input values: computations embedded in the conditional, local variables, etc. In cases where complete test cases cannot be generated, the test case Notes tab describes any missing data that you must **`set`** up to implement the test.



Basis paths are determined by parsing the code and applying the basis path algorithm to the list of branch point nodes that result. As a result, it may be impossible to satisfy a particular branch point when building test cases. In cases where a conditional is based on an intermediate computation, VectorCAST is not able to control the decision point. The following is an example of this case:

Example 1:

```
ROAD_SPEED_LIMIT: constant := 55;
TRUCK_SPEED_LIMIT: constant := 50;

If TRUCK_SPEED_LIMIT <= ROAD_SPEED_LIMIT then
FULL_THROTTLE;
Else
HALF_THROTTLE
End if;
```

Because TRUCK_SPEED_LIMIT is a constant, and always less than ROAD_SPEED_LIMIT, the "else" condition can never be reached.

Example 2:

```
for INDEX in INDEX_TYPE' first..INDEX_TYPE' last
loop
RESET (INDEX);
end loop;
```

Because INDEX_TYPE' first and INDEX_TYPE' last are constants, the loop will always get executed at least once. The basis path calling for the loop to not be entered cannot be satisfied.

```
clicast -e <env> [-u <unit> [-s <sub>]] TOols Auto_Test_Generation
[<outputfile>]

Generate a test script containing one testcase for each basis path in the
environment for the specified unit, subprogram in that unit, or whole
environment.
```

Troubleshooting Min, Mid, Max Test Cases

Constraint Error Caused by Min, Mid, or Max Input Values

When creating Min, Mid, or Max test cases, be aware that *all input* values are automatically set by VectorCAST. In the Tutorial code, for example, the test case PLACE_ORDER_MAX.001 causes a constraint error because the maximum value for TABLE, SEAT, CHECK_TOTAL, and NUMBER_IN_PARTY are too large when they are used as array indices.

Event 3 - Unhandled Exception
UUT: MANAGER
Subprogram: PLACE_ORDER

Exception	Actual Value	Expected Value
<<raise>>	CONSTRAINT_ERROR	[Unexpected Exception]

To solve this problem, click the Parameter Tree tab to return to the Test Case Editor. If you know that

the subprogram under test would never get a value that is too large or too small, then enter your own input values for any parameters that may be getting values that are too large or too small. On the other hand, you may want to modify your source code to handle values that are too large or too small.

Compound Test Cases

Building a Compound Test Case

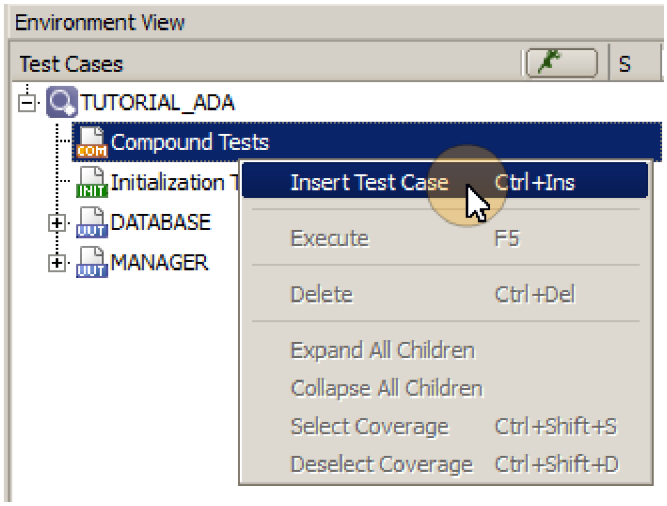
The following sections describe how to build compound test cases. Because compound test cases are a sequence of simple test cases linked together, at least one simple test case must be defined before a compound test case can be built.

A compound test case consists of a series of "slots." Each slot contains the following information:

- A slot number
- The name of the unit
- The name of a subprogram
- The name of a simple test case
- The number of times to invoke the subprogram with the test case data
- A delay (in seconds) to use after the slot is done.

To create a new compound test case:

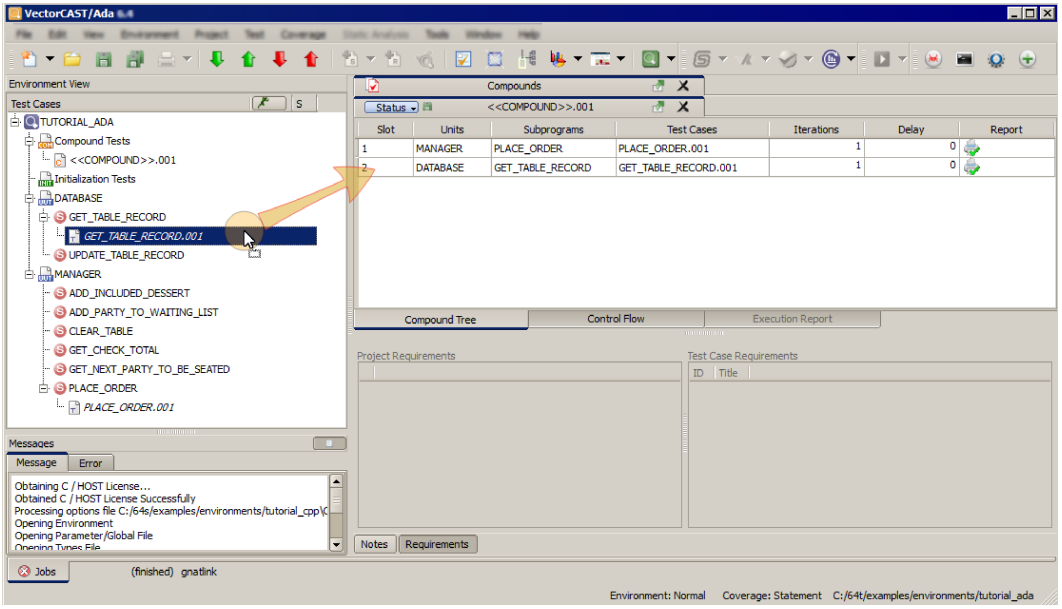
1. Right-click on the Compound Tests node to insert a new test case



   The name defaults to <<COMPOUND>>.001. You can change this by right-clicking on the test case name at any time and selecting **Rename** from the context menu.

2. With your mouse, click-and-drag one or more test cases to the Compound Test Editor located in the MDI area. You can drag and drop a test case to a new position within the Compound Test

Editor.



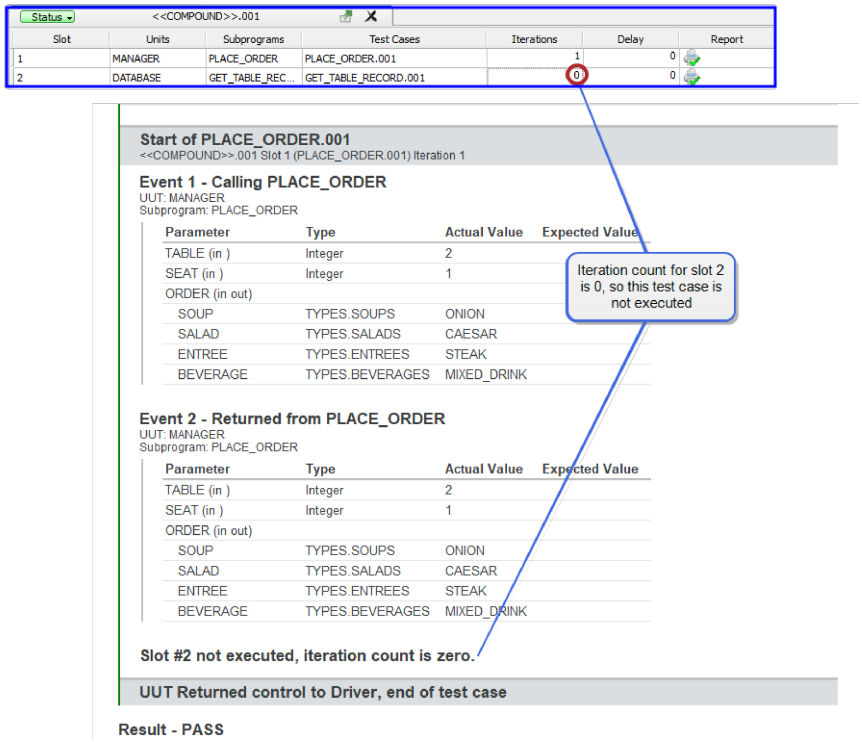
Each test case in a compound is called a *slot*. The unit name, subprogram name, test case name, number of iterations, delay, and report status is displayed for each slot. The slots are executed in order when the compound test case is executed. See " To Reorder Test Cases in a Compound" on page 171 to change the execution order.

To Specify the Number of Iterations of a Test in a Compound

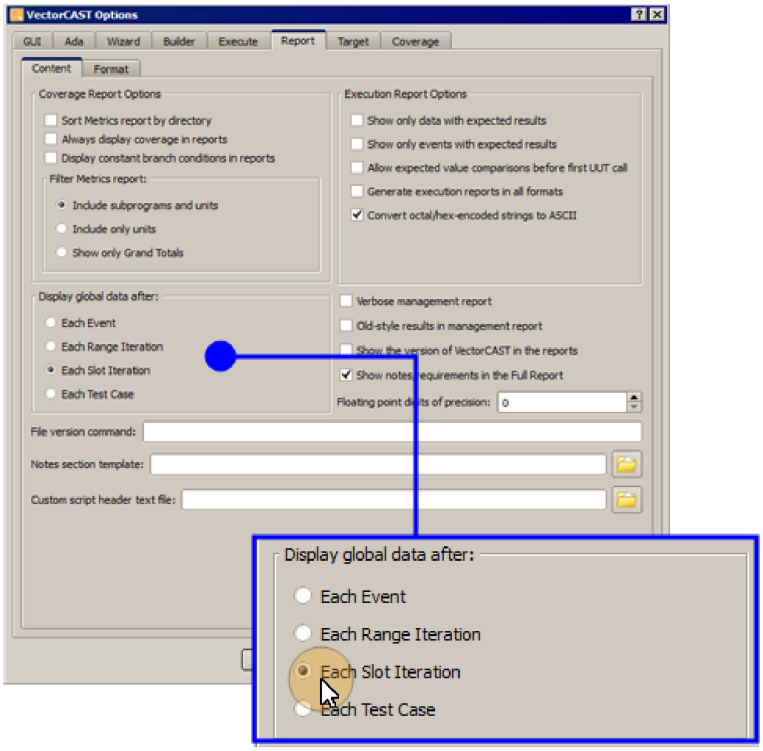
By default, each test case in a compound is executed once. Increasing the number of iterations causes the test case to execute repeatedly before going to the next test case in the compound.

The maximum number of iterations in a compound slot is 999,999.

A setting of 0 Iterations causes the test case to be skipped during execution of the compound. A notation, "Slot <#n> not executed, iteration count is zero", is added to the Execution Report.



If you want global data to be displayed in the Execution Report after each slot iteration, select **Tools => Options** from the Menu Bar to open the Options dialog. Select the Report tab, Content sub-tab and go to the **Display global data after** group. Click the radio button next to **Each slot iteration** and select the **Apply** button.



See also "Compound Test Cases" on page 167.

To Specify a Delay Between Tests in a Compound

A delay capability is available for compound test cases. This enables a time delay to be specified between test slots. The value entered is in seconds, with a precision to the tenth of a second. The delay takes place after the test case slot has been executed the specified number of iterations.

Example: If you want to delay 2.5 seconds after the first test case slot is finished, enter 2.5 in the Delay column for the first slot.

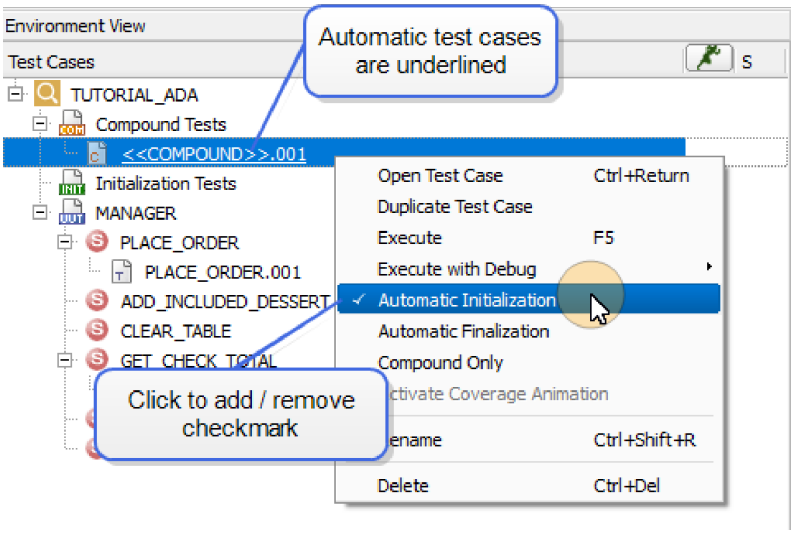
The default delay is zero. The delay feature is useful when testing processing related to synchronizing calls.

Compound as Automatic Initialization / Finalization

A compound test can be set as an Automatic Initialization or Finalization test which runs either before or after other test cases. The Automatic Initialization test is invoked at the beginning of each test execution and is useful for setting up data for the test case that follows it. The Automatic Finalization test is invoked at the end of test case execution and is useful for checking data that the preceding test may have changed.

See the sections "TEST.AUTOMATIC_FINALIZATION" on page 465 and "TEST.AUTOMATIC_INITIALIZATION" on page 466 for further information.

In this example, <<COMPOUND>>.001 has been marked as an Automatic Initialization test. Tests set as automatic tests have their name underlined in the Test Case Tree.



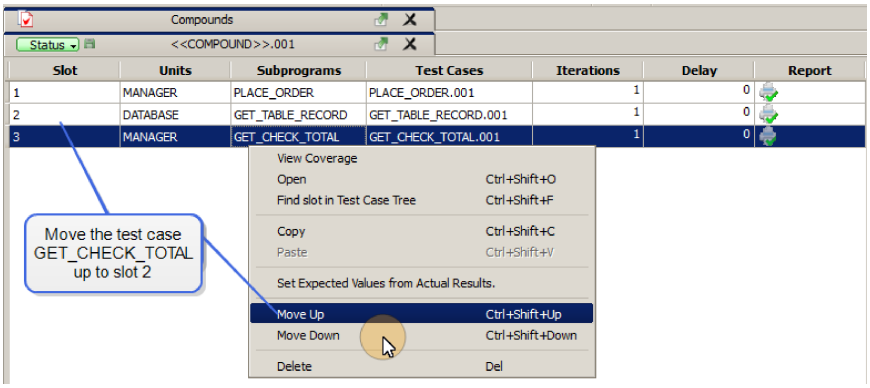
To Reorder Test Cases in a Compound

In a compound test case, the individual test cases, or *slots*, are executed in order from top to bottom. To change the order in which the tests are executed, you change the order that they are listed in the editor. To do this, right-click a test case, and choose **Move Up** or **Move Down**.

The keyboard shortcut for Move Up is **Ctrl+Shift+Up Arrow**.

The keyboard shortcut for Move Down is **Ctrl+Shift+Down Arrow**.

Alternatively, you can drag and drop a test case to a new position within the Compound Test Editor. Multi-selection drag and drop is supported. When you select multiple slots and drop them in a new position, the slots are dropped in the order in which they were selected. In this way you can either duplicate slots or duplicate and re-order slots at the same time.



To Open a Test Case from a Compound

To open an individual test case from a Compound test case, double-click the test case. Alternatively, right-click the test case and choose **Open**.

The keyboard shortcut for Open is **Ctrl+Shift+O**.

The test case opens in the Test Case Editor, located below the Compound Test Editor. Once changes are made to the individual test case and saved, they are part of the Compound too.

To Delete a Slot from a Compound

To delete a test case from a Compound, right-click the test case, and choose **Delete**. Selecting Delete removes the test case entirely from the compound test case; it does not delete the test case from the environment.

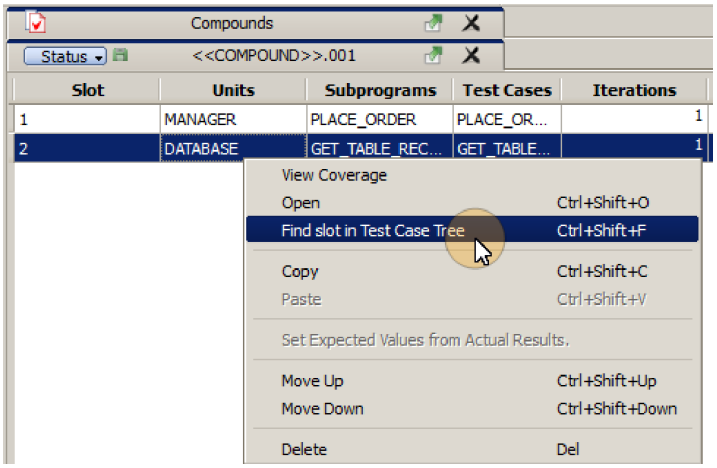
The keyboard shortcut for Delete is **Del**.

To View Coverage for a Test Case from a Compound

To open the Coverage Viewer for a particular subprogram from a Compound, right-click the test case and choose **View Coverage**. The Coverage Viewer opens, scrolled to the top of the particular subprogram referenced by that test case.

To Locate a Slot in the Test Case Tree

To find the test case in the Test Case Tree that corresponds to the slot selected in the Compound Test Case Editor, right-click on the slot in the Compound Test Editor and then select **Find slot in Test Case Tree** from the context menu.



The keyboard shortcut to locate a slot is **Ctrl+Shift+F**.

The slot is highlighted in the Test Case Tree.

To Search for Text in the Compound Test Editor

The Compound Test Editor supports Search to find text within a slot. To search within the Compound Test Editor, select **Edit => Find** or press **Ctrl+F**. A Find Banner appears at the bottom of the Editor.

Type the text you are looking for in the **Find** text edit box and press **Enter**.

The search looks in all columns (slot number, unit name, subprogram name, test case name, iterations, and delay). So searching for the number 3 will find both slot 3 and any slot with Iterations set to 3, for example.

For more details, see "To Search for Text Using the Find Banner" on page 85.

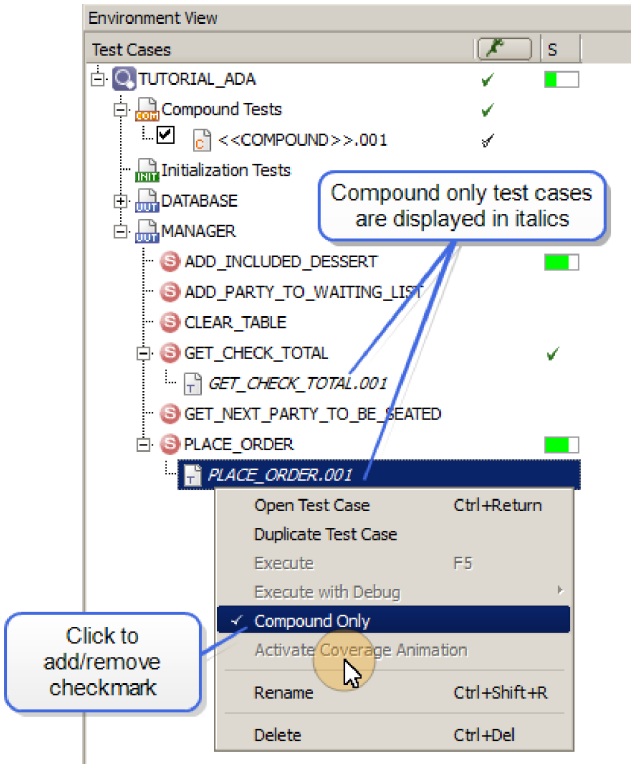
Compound Only Test Cases

Compound Only test cases are only executed from within Compound tests, and are not executed individually in normal or batch mode. Both individual test cases as well as compound test cases can be set to "Compound Only". To specify that a test case is "Compound Only," right-click the test case and choose **Compound Only** from the context menu.

```
clicast -lada -e <env> -u <unit> -s <sub> -t <testcase> TESt Compound_only
true | false

Change the specified test case to a "Compound Only" test case or back
again. "Compound Only" test cases are not executed in normal or in batch
mode, but only as part of a compound test case.
```

When a test case is "Compound Only" its name is italicized, and there is a check mark next to Compound Only in the right-click context menu.

To specify that a test case is no longer Compound Only, right-click the test case and choose **Compound Only** again, such that the check mark is removed.



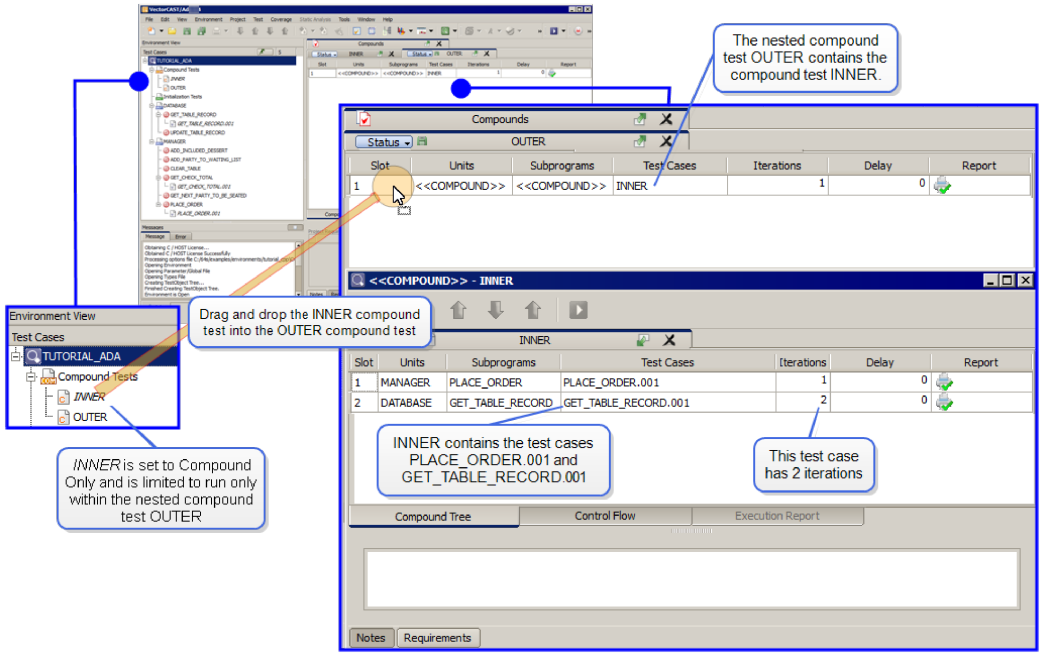
Nested Compound Test Cases

A compound test can be added as a slot to an existing compound test creating a *nested compound*

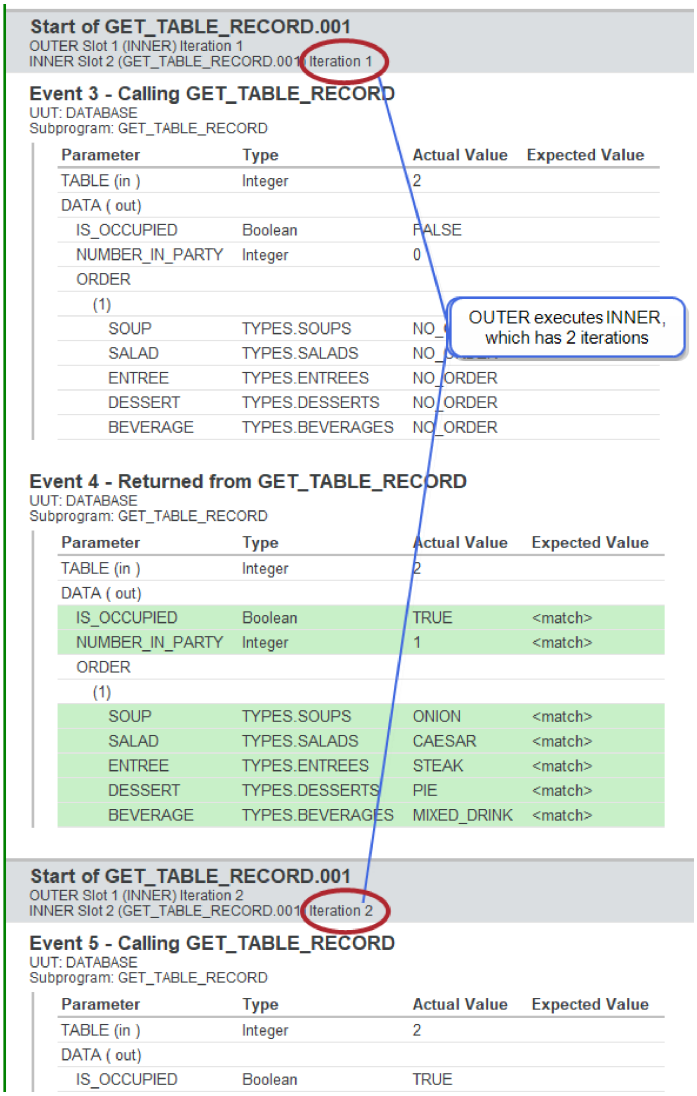
test. To do so, simply drag an existing compound test case into a new compound test case. A compound test can be limited to run only within a nested compound test case by setting it to "Compound Only." See "Compound Only Test Cases" above.

In this example, PLACE_ORDER.001 and GET_TABLE_RECORD.001 are the only slots in the compound test case named INNER. The number of iterations for GET_TABLE_RECORD.001 is set to 2. Another compound test case is inserted in the Test Case Tree, named OUTER, and then INNER is dragged into OUTER.

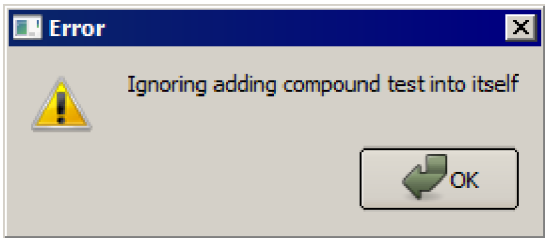
Double-clicking on INNER in the OUTER Compound Tree displays the Compound Tree for INNER.



Execution report event headers for a nested compound event reflect the nesting relationship. Below, Event 3 of the execution report for OUTER shows that OUTER executes INNER and INNER executes the first iteration of GET_TABLE_RECORD.001. Scrolling down the report shows that in Event 5, the second iteration of GET_TABLE_RECORD.001 in INNER is executed.



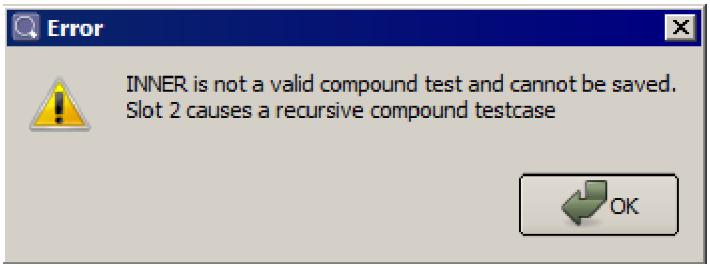
A compound test cannot be a nested slot of itself. If you drag a compound onto itself, the following error dialog appears:



Nested compound tests do not support test recursion. For example, if OUTER nests INNER, and INNER nests OUTER, the relationship is a recursive one.

The recursive relationship is detected when the change to the compound test is saved, not when the nested compound is created. When a recursive relationship is detected, an error dialog appears

when you save the compound test case:



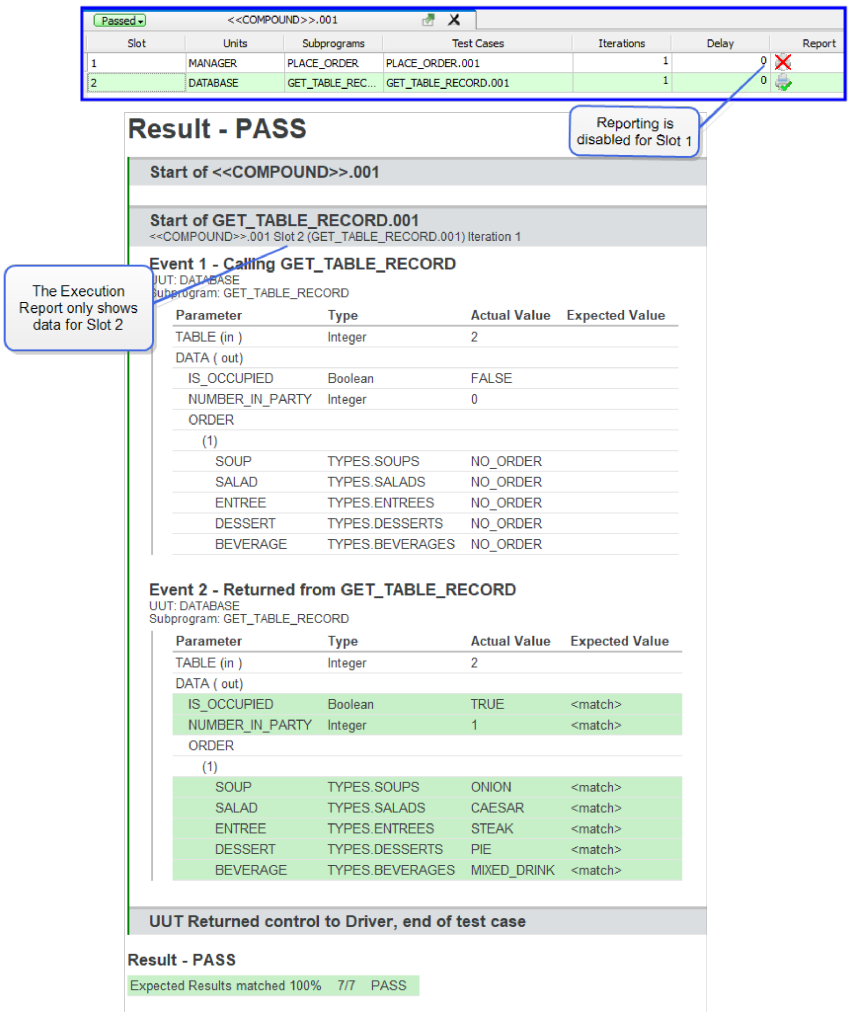
To specify a compound test case as a slot in a compound test script, use syntax similar to the following examples:

```
-- COMPOUND TESTS

TEST.SUBPROGRAM:<<COMPOUND>>
TEST.NEW
TEST.NAME:INNER
TEST.COMPOUND_ONLY
TEST.SLOT: "1", "MANAGER", "PLACE_ORDER", "1", "PLACE_ORDER.001"
TEST.SLOT: "2", "DATABASE", "GET_TABLE_RECORD", "2", "GET_TABLE_RECORD.001"
TEST.END
--

-- COMPOUND TESTS

TEST.SUBPROGRAM:<<COMPOUND>>
TEST.NEW
TEST.NAME:OUTER
TEST.SLOT: "1", "<<COMPOUND>>", "<<COMPOUND>>", "1", "INNER"
TEST.END
--
```

To Enable / Disable Reporting for Compound Slots

VectorCAST provides the ability to prevent the harness from generating Execution Report data for selected slots in a compound test case (coverage data will still be gathered). This is useful in instances of large compound test cases with numerous iterations where, for example, only the first and last iterations of a test are of interest.

By default, reporting is enabled and the **Enable Reporting** icon is displayed in the far-right column of the Compound Editor. To prevent the harness from reporting on executing data in the Execution Report, click on the icon to toggle to the **Disable Reporting** icon .



From a test script, disable reporting by adding "PRINT=FALSE" to the end of the TEST.SLOT line. For example:

```
TEST.SLOT: "1", "MANAGER", "ADD_PARTY_TO_WAITING_LIST", "1", "ADD_PARTY_TO_
WAITING_LIST.001", "PRINT=FALSE"
```

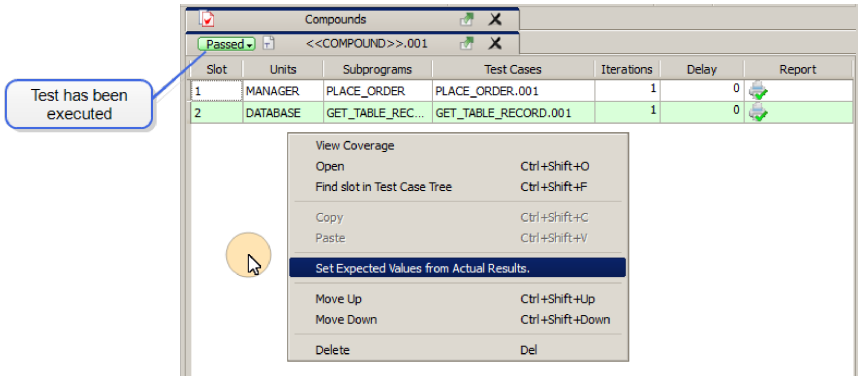
Note: If the slot data contains expected results (either for the particular test case, or, in the case of a nested compound, within the execution of the slot), then the compound test will fail to run and will notify the user of the reason.

Control Flow and Slots With Disabled Reporting

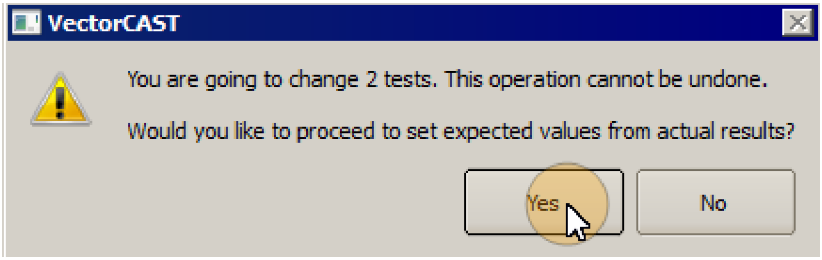
When the Print flag is set to False for one or more slots in a compound test, the slots execute but the test harness is not aware of them. Therefore, the Control Flow for the compound excludes the events for the "hidden" slots.

To Set Expected Values to Actual Values

After executing a test case, Expected Values can be set to the actual execution result values. Right-click anywhere in the Compound Test Editor and select **Set Expected Values from Actual Results** from the context menu.



A warning dialog appears indicating the number of test cases that will be changed by the action. You have the opportunity to abort the action by selecting the **No** button. Selecting the **Yes** button will allow the operation to proceed and modify the test cases. Note that this action cannot be undone.



All Expected Values listed in the Execution Report are set to their actual values.

See Controlling Report Information for more information on capturing output for variables.

```
clicast -l<lang> -e <env> [-u <unit> [-s <sub> [-t <testcase>]]] TESt
Actuals_to_expected

This will set the actual values for a test case as the expected values.
Thus 100% of the expected results will match for that test case.
```

Entering Test Case Data

There are two kinds of test case data: Input Values and Expected Values. The *input values* are values to be provided as stimulus for the test. The *expected values* are values to be verified during the test.

There are two ways to enter test case data. Enter test data directly into the Parameter Tree by selecting the appropriate spot in the “Input Value” or “Expected Value” column next to the parameter name. Alternatively, double-click on any parameter or global object from within the Parameter Tree to bring up the Parameter dialog box for the data object. The two methods are described below.

The Parameter Tree

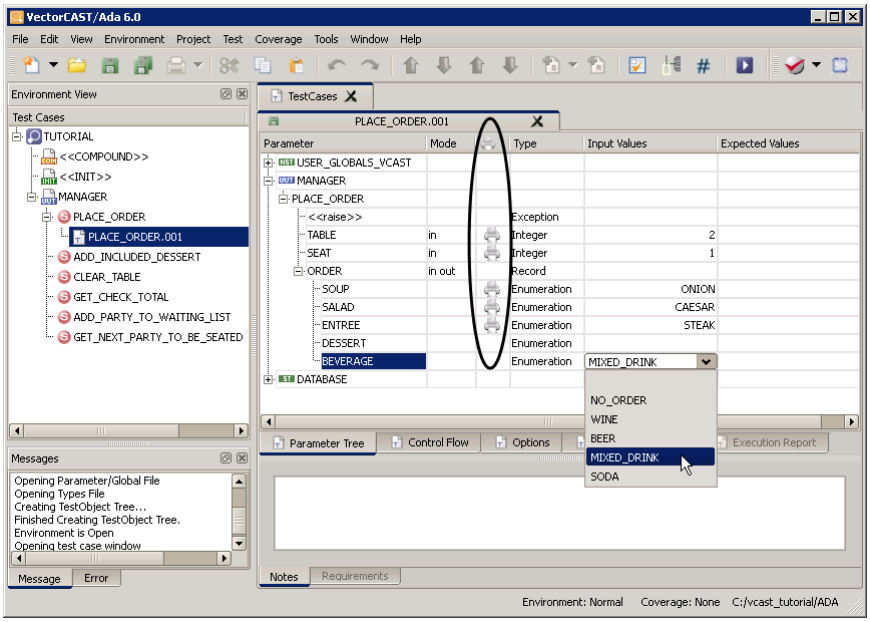
Once you have created a test case, it opens in the Test Case Editor, in the MDI window. One of the tabs of Test Case Editor, the Parameter Tree tab, displays the Unit(s) Under Test and their subprograms in a tree-like hierarchy. Using this hierarchy, you can edit the Input and Expected values for the test case.

When entering parameter values for the subprogram being tested, all parameter values that will be accessed during subprogram execution must be set. If no value is provided for a parameter or object, its value is indeterminate. Some compilers zero the stack heap and therefore parameter values before allocating data but this should not be relied upon.

Note: If you hover the mouse cursor over a cell in the Parameter Tree, VectorCAST displays the legal range of values for that parameter in a tool-tip.

The Parameter Tree also contains a column to control which parameters are to be captured in the Test Execution Report. This column is indicated by the  icon. For more information on using this control, see "To Enter Input and Expected Values" on page 182and "Controlling Report Information" on page 259

Using the Tutorial environment as an example, the UUT is **`manager`**, and one of its subprograms is **`Place_Order`**. **`Place_Order`**'s parameters consist of an integer type for the Table, an integer type for the Seat, and a structure type for the Order. The dependent unit **`database`** is also a UUT. In this case, the Parameter Tree looks similar to the following screen shot:



See also "Data Types" on page 185 for information on entering Input and Expected data for specific data types.

To Navigate the Parameter Tree

In the Parameter Tree, the unit names, <<GLOBALS>>, subprograms, and parameters are displayed hierarchically.

To expand a node

Mouse: click the icon to the left of the node

Keyboard: after placing the keyboard focus on that node, press the right-arrow key

To collapse a node

Mouse: click the icon to the left of the node

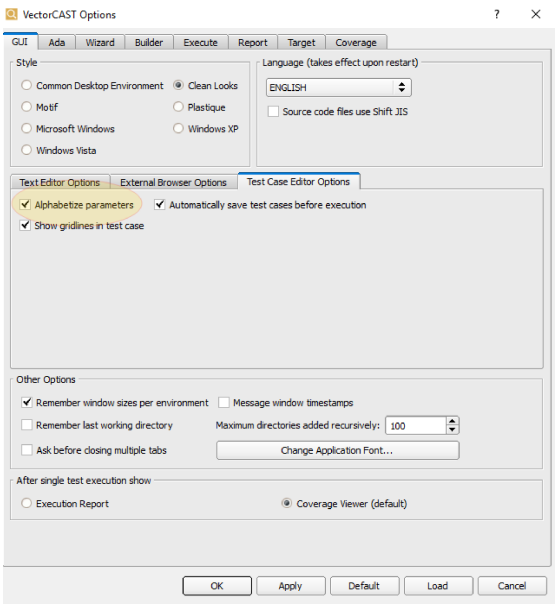
Keyboard: press the left-arrow key

Additionally, use the keyboard's up- and down-arrow keys traverse the expanded nodes. You can also type the first letter of an expanded node name to jump to that node.

When you arrive at the node you would like to edit, simply press Enter. This will edit either the Input or Expected value (depending on which can be edited at the time).While editing a value, press the Tab key to continue to the next value, or use the up- and down-arrow keys to continue to the next editable node in the same column.

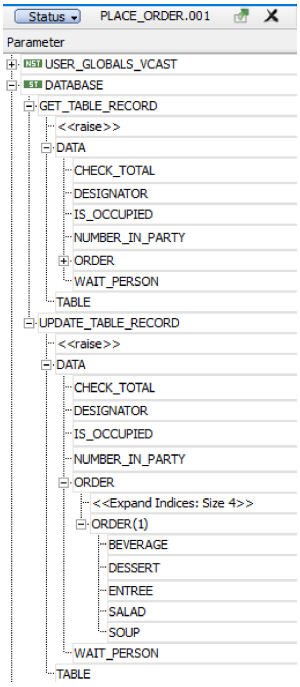
To Alphabetize Parameters in the Parameter Tree

To alphabetize the items for all nodes in the Parameter Tree, select **Tools => Options => GUI**. Then click the Test Case Editor Options tab and click **Alphabetize parameters** in the *Other Options* panel. Click **Apply** or **OK.**

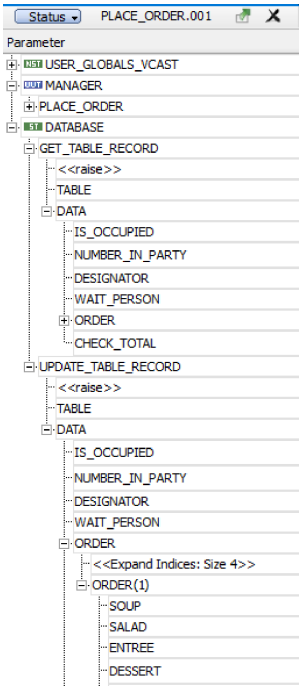


Close any open test cases. Open a test case and all items for each node of the Parameter Tree are

displayed alphabetically, with “return” always appearing last.



To restore the order as originally processed by VectorCAST, uncheck Alphabetize Parameters in **Tools => Options => GUI**. Close any open test cases. Open a test case and all items for each node of the Parameter Tree are displayed as originally processed by VectorCAST.



To Enter Input and Expected Values

To enter values for particular parameters, simply type the value in the appropriate column (Input Values column or Expected Values column) next to the parameter name in the Parameter Tree.

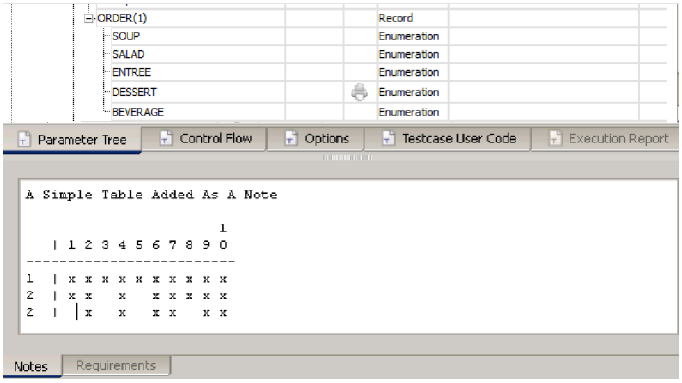
Input Values to a Unit Under Test are “in” parameters; Expected Values are returns or pointers.

Parameters passed to a subprogram in a stubbed unit are Expected Values; values returned from a stub are Input Values.

Entering an Input Value or Expected value into the Parameter Tree automatically enables the data to be displayed in the Test Execution Report. This is indicated by the  in the parameter tree. For Input Values, the data may be optionally hidden (See Controlling Report Information). Entered expected values are always shown on the Test Execution Report and may not be hidden.

To Enter Test Case Requirements or Notes

The Notes tab enables you to specify text that will be associated with a specific test case. This information can be an actual test requirement or test rationale. The information from this window is retained in the test case data. In the figure below, some notes are entered for the test case. The Notes tab uses a monospaced font allowing the creation of a properly spaced “graphical” table.



To Enter Values for Global Data

In the Parameter Tree, if any unit has global data objects, the first entry in the tree below that unit’s name is <<GLOBAL>>. By expanding <<GLOBAL>> you see a list of global data objects which are defined in that unit. All processing associated with setting values for global objects is exactly the same as setting data for a parameter.

To Enter a Range

For any data type, you can enter a range for an Input or Expected Value in the Parameter Tree. In the Input Values column, the range syntax is ***low..high/delta***. The range causes the test case to iterate over the input values starting at *low*, going to *high*, by *delta*.

In the Expected Values column, the range syntax is ***low..high*** and defines a range of values that will match an input. The acceptable range includes ***low*** and ***high***.

See also "The Range Values Tab" on page 213.

To Enter a List

For any data type, you can enter a list for an Input or Expected Value in the Parameter Tree. In the Input Values column, the list syntax is ***value-1,value-2,value-n***. The list causes the test case to iterate over the input values starting with ***value-1***, and continuing to ***value-n***.

The special input value **<<KEEP>>** indicates that the input object should keep its current value from the previous iteration, rather than being modified. This special input value is useful when an item needs an initial value but an iteration is occurring and you don't want a new value set on each iteration. For example, the input might look like:

```
123,<<KEEP>>,<<KEEP>>
```

In the Expected Values column, the list syntax is ***value-1,value-2,value-n*** and defines a list of values that will match an input, one at a time. There is no space between the items. To specify that an item be repeated, use parentheses around the number of times to repeat the value: (***repeat***)***value-1,(repeat)value-2,(repeat)value-n***

Repeating and non-repeating list items can be mixed in the list. A range can be an item in an Expected Values list.

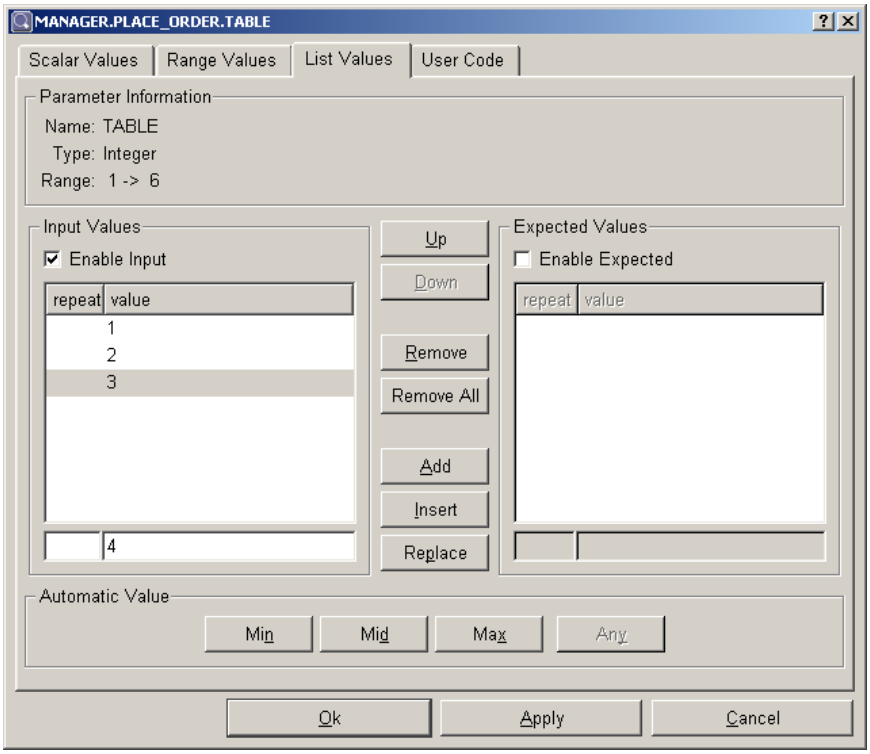
The List Values Tab

The List Values tab enables you to provide a series of values to be used when calling the unit under test or to provide a sequence of specific scalar values to be returned from stubs when a stub is called multiple times during a test case execution. A list is similar to a range, except the items in a list can have different deltas between them, so you specify each item in the list. Items in a range are always separated by the same delta, so you only need specify the beginning and end items, and the delta.

When you add items to the Input Values list for a parameter, you are assigning a series of input values to that parameter. Adding more than one item to the list causes an iteration, that is, the test case is executed once for each item. Repeating a value in the list causes additional iterations.

Using an Input Values list for a stub parameter means that the stubbed subprogram returns one item from the list each time it is called. If a stub is called more time than you have provided values for in your list, then the last value in the list is used repeatedly, once the list items are exhausted. Each time a test case iterates, the Input Values list for a stub is reset to the beginning.

When you add items to the Expected Values list for the same or a different parameter, you are assigning a series of expected values for that parameter. One expected value is compared to the parameter's value per test iteration. The expected value can be a range of acceptable values for that iteration, as described below.



To Pass a Null String

use source code testsuite/LANGUAGES/ADA_95/SOURCE/wide.adb and wide.ads to find a string in out param.To pass a null string as an input value or set an expected value to a null string, you must use user code. The VectorCAST object that contains the length of an unconstrained string is the name of the parameter with a "_L" suffix.

For example:

```
package UUT is
procedure CONVERT (S : in out string);
function GET return string;
```

To pass a null string to CONVERT, use the following user code:

```
<<UUT.CONVERT.S>>_L := 0;
```

To check that GET returns a null string, use the following user code:

```
{{ <<UUT.GET.return>>_L =0 }}
```

To Raise an Exception from a Stub

Test cases can be designed to cause stubs to explicitly raise an exception when they are called. When building test cases for a stub, you will see an item <<raise>> in the list of parameters.

By coordinating the list of data values with the list of exceptions you can achieve a variety of

interesting results. For instance, if you want the stub to return normal data on the first two calls and then to raise an exception on the third and subsequent calls, you would set up test case data similar to the following script example:

```
TEST.VALUE:DATABASE.GET_VALUE.<<raise>>:NONE,NONE,STORAGE_ERROR
TEST.VALUE:DATABASE.GET_VALUE.DATA:12,17
```

This test case data will result in the values 12 and 17 being returned on the first two calls to GET_VALUE and then an exception STORAGE_ERROR being raised on the third and subsequent calls. If you only want the exception to be raised on the third call and then data to again be returned on the fourth call, add NONE after STORAGE ERROR in the list of exceptions.

Working with In/Out Parameters in Stubs

The value of an "in-out" stub parameter is stored when the stub is called and then re-loaded with the test case value (if any) that has been set in the current test case. In the case where there is no test case value the "out" value will equal the "in" value.

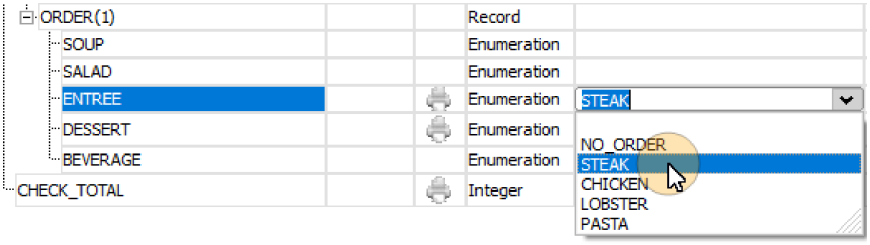
Note: Only the "in" value will appear in the execution results.

Data Types

The parameter types referred to in this user's guide are defined as "Scalar" and “Composite", with Scalar types being broken down into Integer and Floating Point types. Integer types can be further subdivided into Enumerated and Numeric types. Composite types can be subdivided into array, structure (including class), and pointer types.

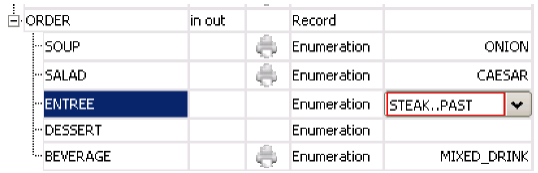
To Enter Values for Enumeration Types

For enumeration types, values are set by clicking in the Input Value space and selecting from the drop-down list of enumerals. You may also type the enumeral value. VectorCAST auto-completes the name as soon as the entered characters uniquely identify an enumeral.



If you enter an illegal value, VectorCAST outlines the cell in red and the tool-tip provides information.

When entering a range for an enumeration type, the syntax is *value1..value2*, where *value1* and *value2* are enumerals defined in the type. VectorCAST auto-completes the first value as you type it. After you type “..” a red outline appears, until you finish typing the second value. VectorCAST does not auto-complete the second value in the range.



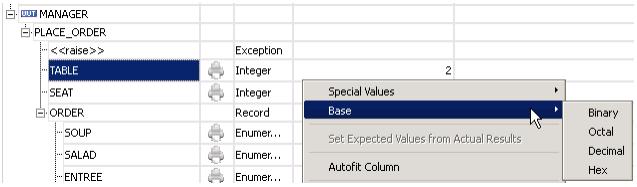
To Enter a Number for an Enumeration

In some cases you may want to enter a number in place of the enumeral. VectorCAST permits integer values to be assigned to enumerated types. In this way, you can force an out-of-range value for an enumeration type parameter.

To Enter Integer Values in Different Bases

The assumed "base" for entered values is base 10. For integer types you may also select alternate bases.

To change the base, right-click an input or expected value in the parameter tree and select the desired base from the context menu.



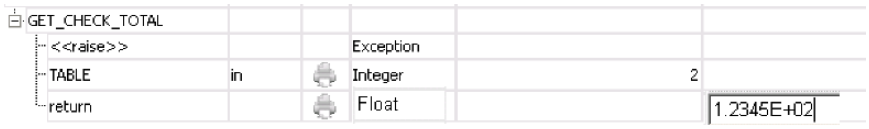
A base may also be specified as part of the value, using the standard C syntax of a “0” prefix for octal or a “0x” prefix for hexadecimal, or Ada syntax of “8#” prefix for octal, “16#” prefix for hexadecimal, or “2#” prefix for binary. When using the Ada syntax, any value between 2 through 16 is valid.

For example:

- Hex: 16#3A0# or 0x3A0
- Octal: 8#737# or 0737
- Binary: 2#1010#
- Base 5: 5#12# (equivalent to the value of 7 for the decimal base).

To Enter Real Numbers using Scientific Notation

Real numbers may be entered using scientific notation (i.e. 1.2345E+02).



To Enter Values for Record Types

To enter values for a Record type, use the ⊞ button next to the parameter name to see the fields defined for the record type.

As a record type is a composite data type, you do not enter values for the record directly. You choose a member of the record by first expanding our the list of members and then setting a value for any scalar members. If a member is a record, you would expand that member using the ⊞ button again.

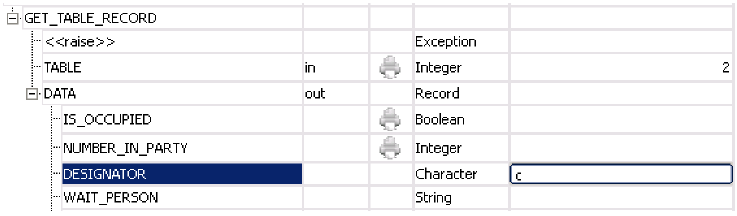
Variant Record Types

VectorCAST supports dynamic modification of the discriminant value for variant structure type with default and non-defaulted discriminants.

When working with variant structures, you will need to change the value of the discriminant before working with fields whose availability is controlled by the discriminant.

Character Types

If the parameter or object chosen is of "character" type, you can enter the character in the "Input Value" or "Expected Value" area.



Note: Enter character without quotation marks.

Note: Enter non-printable characters (i.e. carriage returns, line feed.) by using the hex value for the character in standard Ada syntax (i.e. 16#0d#)

To Enter a String

To enter a value for parameter of type string, type the string without quotes into the Input Values or Expected Values column. To include a comma character in the string, use the escape character before it. For example, the string **`bob\,tom`** is a string of one item.

To type a list of strings, use the comma character to separate the items of the list. For example, entering **`bob,tom`** in the Input Values column for a string parameter enters a list of two string items, causing the test case to iterate with “bob” as the first input and “tom” as the second input.

You can enter the following special characters without escaping them:

```
~ ! ` @ # $ ^ & * ( ) - _ + = { } [ ] | : ; " ' < > ? / <space>
```

To enter the character \ (backslash), escape it with another \ first. To enter the comma, escape it with

a backslash first.

To enter the % character, use the syntax \045, as the % character itself is used internally by VectorCAST as a string delimiter.

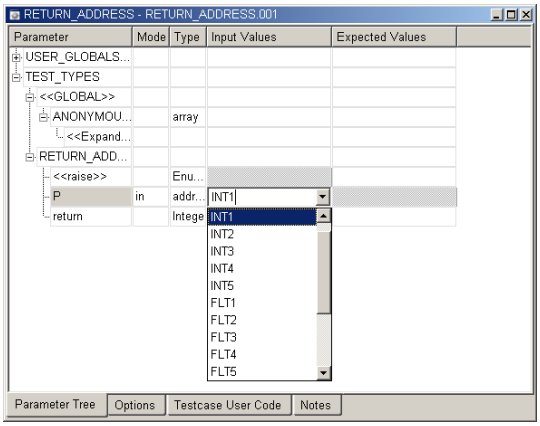
Note: When setting values of any constrained string type or sub-type, VectorCAST will blank pad the entered string value to match the size of the parameter type, or truncate the entered value to fit the appropriate size.

System.Address Types

System.Address types are most commonly used in Ada to pass the address of some data object to a procedure. Once inside the procedure the address parameter is cast to an Ada type and then used. The following code fragment illustrates this point:

```
function ADDRESS_EXAMPLE (P : SYSTEM.ADDRESS) return integer is
LOCAL_VAL : integer;
for LOCAL_VAL use at P;
begin
return LOCAL_VAL;
end ADDRESS_EXAMPLE;
```

If the parameter or object chosen is of "address" type, you can choose from a list of object names that are defined in the USER_GLOBALS_VCAST package. Setting the data value involves 2 steps. First, you need to make an association between the parameter and the address of a particular global as shown here:

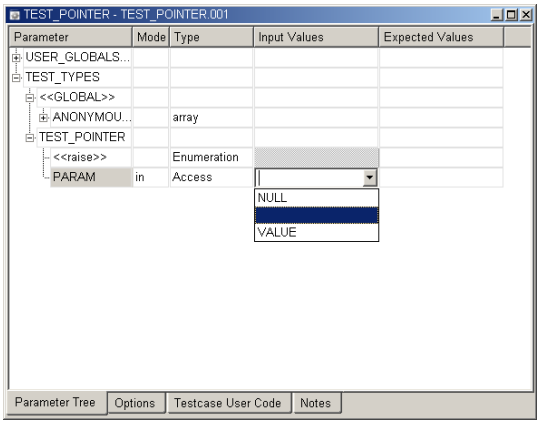


Then, you must set the value of INT1 in the unit USER_GLOBALS_VCAST as you would for any other integer value. By doing these two steps, VectorCAST will pass the “’address” of INT1 to the procedure and when the procedure does a type cast of the address it will be pointing at integer-formatted data.

Access Types

If the parameter or object chosen is an “access” type, you can set the value of the pointer to be null or to point to an allocated object. You choose null by selecting NULL from the drop-down menu. You

choose to allocate an object by choosing VALUE. If you choose to allocate an object you will be able to then set values for the allocated object in the normal way.



Special Kinds of Stubs

VectorCAST supports the stubbing of generic packages and packages that contain: nested generic packages or subprograms, task types, protected types, and task specifications. The following paragraphs describe features specific to the stub type:

Protected type definitions - The stub body for the protected type will allow the user to control the values returned by the entries defined in the protected type in the normal way that this is allowed for other stubbed subprograms.

Generic packages - The stubbed bodies are not automatically compiled into the harness because the compilation of the generic body may put instantiations of that generic out of date. To compile the stubs into the test harness, turn on the option "Compile generic stubs". The only parameters that can be set or expected are for types defined external to the generic package.

Task Types - The stub body for the task type will allow the user to control the values returned by the task entries in the normal way that this is allowed for other stubbed subprograms.

Protected Types and Task Types

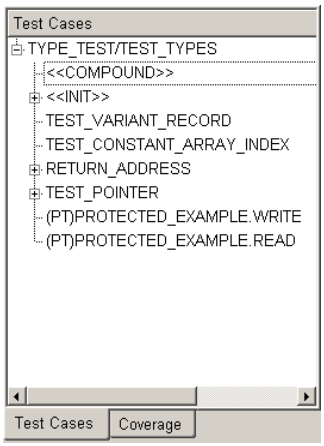
VectorCAST supports the testing of task types and protected types. When testing a package that contains a task type or a protected type, the list of subprograms will contain choices for the task type entries and the protected type subprograms. These choices will be preceded with (pt) for protected types or (tt) for task types, and the name of the entry point or subprogram.

For example, your package contains a protected type called PROTECTED_EXAMPLE with subprograms WRITE and READ.

```
-- Protected Type Example
protected type Protected_Example is
   procedure Write (Value : integer);
   function Read return integer;
private
```

```
    Data : integer := 0;
 end;
```

You will see the following two entries in the list of subprograms:
(PT) PROTECTED_EXAMPLE.WRITE, and
(PT) PROTECTED_EXAMPLE.READ. Data can be set for the parameters associated with the entry or subprogram in the same way as for any other subprogram parameter.



Generic Packages

To test a generic package you must manually create a stand-alone generic instantiation compilation unit for the generic package and then make this instantiation the Unit Under Test.

The following example will illustrate this concept. Given the following generic specification to be tested:

```
generic
type FIRST_GENERIC_TYPE is private;
type SECOND_GENERIC_TYPE is private;
package GENERIC_PKG is
type TYPE_IN_GENERIC_PKG is new integer;
type DERIVED_FROM_GENERIC is new FIRST_GENERIC_TYPE;
type RECORD_IN_GENERIC_PKG is
 record
   field : FIRST_GENERIC_TYPE;
 end record;
type PTR_IN_GENERIC_PKG is access RECORD_IN_GENERIC_PKG;
type PTR_TO_GENERIC_TYPE is access FIRST_GENERIC_TYPE;
procedure DO_GENERIC_PROCESSING (
 PARAM1 : FIRST_GENERIC_TYPE;
 PARAM2 : SECOND_GENERIC_TYPE);
end GENERIC_PKG;
```

We would choose data types for FIRST_GENERIC_TYPE and SECOND_GENERIC_TYPE and

create an instantiation package. For example:

```
package GENERIC_TYPES_PKG is
type DERIVED_FROM is new integer range 20..50;
type RECORD_TYPE is
  record
   FIELD : float;
  end record;
end GENERIC_TYPES_PKG;
with GENERIC_TYPES_PKG;
with GENERIC_PKG;
package GENERIC_INSTANTIATION is new GENERIC_PKG (
GENERIC_TYPES_PKG.DERIVED_FROM,
SECOND_GENERIC_TYPE => GENERIC_TYPES_PKG.RECORD_TYPE);
```

The parameter profile for the generic package may or may not use named association for the generic parameters as in the above example. A VectorCAST imposed restriction is that the actual parameters must be in the same order as the generic formal parameters whether or not they use named notation.

After creating and compiling the GENERIC_INSTANTIATION package we are ready to begin testing. Simply use VectorCAST to build a test environment with GENERIC_INSTANTIATION as the Unit Under Test. This will allow testing of the generic subprograms including the manipulation of objects or parameters defined in the generic package.

Testing Out of Range Values

In some cases it is desirable to test data values that violate the associated Ada type. For example, if Ada exception handling is to be disabled when the application is deployed you will want to test the application with out-of-range data values.

There are two options that are relevant to testing out-of-range values. These are the "Compiler options" and the "Range check" options. To ensure that the VectorCAST-generated test harness components are compiled with run-time checks disabled you will need to modify the VectorCAST "compile options" to use the compiler-specific command line option to force checks off. The syntax for this varies from compiler to compiler. Please consult your compiler documentation for the relevant command. (As an example Green Hills Ada uses "-no_check" on the command line to suppress run-time checks).

Once the harness is built and compiled with "checks off", you can turn on and turn off the VectorCAST imposed range checking on data values by using the "Range Check Option" in the Options Dialog box. To access the Options Dialog use the **TOOLS => OPTIONS** choice in the main menu. The Range Check Option is available in the "Execute" tab and has three choices: None, Disable, All. The default value for this option is All. Each of the choices results in slightly different processing by VectorCAST.

The **All** choice is the default. When range checking is ON, VectorCAST will range check every scalar value that is entered and disallow the entry of any out of range values. This range checking applies to data that is entered interactively through the GUI or via a test case script.

When range checking is in the **Disable** position, VectorCAST will not enforce any scalar ranges. The scalar input dialogs will still show the applicable Ada type and its associated range, but you will be allowed to enter data values that are outside of this range. The only caveat to this is that you will, for

obvious reasons not be able to enter data values that are larger than the physical memory the compiler allocates to objects of this type (either the default or the size provided by a 'size attribute). If a value is entered that is larger than the physical memory size, the value will be truncated at run-time and the smaller, truncated value will show up in the VectorCAST test results. You should consult your compiler documentation for details on the default memory allocation for various Ada types.

The **None** position is similar to Disable with the added restriction that VectorCAST will not show the type range in any of the scalar data entry dialog boxes. This choice will not be used in the normal course of testing, it should be used only if directed by VectorCAST Technical Support.

VectorCAST and "Class" based Testing

VectorCAST Harness Architecture

The VectorCAST architecture is centered on the executable test harness that is constructed using the VectorCAST environment constructor. The source code for this harness is custom built each time a new version of your application is built, and consists of a main program (Test Driver), the units being tested, and one or more stub units that take the place of actual units. The test harness code is compiled and linked into an executable program using the same compiler and linker that you are using for your application code.

The VectorCAST test harness is completely data driven. This means that the test data is de-coupled from the harness source code. The importance of this is that it enables you to build a suite of test data that is persistent across multiple versions of the application source code. Because the test data is persistent and the test harness code can be automatically regenerated for any version of the application code as necessary, regression testing can be accomplished for each and every "build" of your application totally automatically!

Testing and Object Oriented Programming (OOP)

When it comes time to unit test, you will normally want to test each package and child package independently to verify that the types and operations introduced in the base class AND in any derived classes are working properly. Testing will normally proceed downward from the root type of the class toward the children. Once the operations associated with the base type are tested, they do not need to be re-tested for each descendent type since the same logic will be used by all calls to these operations, regardless of the type of the actual parameter.

This discussion will focus on the following example Ada 95 code which uses the concepts of type extension, and child packages to implement one base class and two sub-classes. The following is similar to an example in Chapter 13 of *Programming in Ada 95* by John Barnes.

Assume you have a package called OBJECTS as follows that declares a type OBJECT with two primitive operations associated with a single dimension object.

```
package objects is
type object is tagged
 record
  xpos : float;
  ypos : float;
 end record;
```

```
function range_in_yards (center : object) return float;
function bearing_in_degrees (center : object) return float;

end objects;
```

Also assume that you have a child package called OBJECTS.DIMENSION2, that extends the type OBJECTS for specific two dimensional geometric shapes and also adds the AREA operations for these extended types.

```
package objects.dimension2 is

type rectangle is new object with
  record
    length : float;
    width : float;
  end record;

type circle is new object with
  record
    radius : float;
  end record;

function area (r : rectangle) return float;
function area (c : circle) return float;

end objects.dimension2;
```

Finally assume that you have a child package called OBJECTS.DIMENSION2.DIMENSION3, that extends the types CIRCLE and RECTANGLE into three dimensional geometric shapes and also adds the VOLUME operation for these extended types.

```
package objects.dimension2.dimension3 is
type box is new rectangle with
  record
   height : float;
  end record;

type sphere is new circle with null record;

function volume (b : box) return float;
function volume (s : sphere) return float;

end objects.dimension2.dimension3;
```

When developing a test strategy for this implementation, you will want to test the operations associated with the base class OBJECT, and then move on to test each of the descendent classes. It is not necessary to test the function BEARING_IN_DEGREES with objects of the type BOX for instance because once these primitive operations are tested they are assumed correct for all descendants. This is a basic foundation of Object Oriented Programming.

VectorCAST and Object Oriented Testing

When you build a VectorCAST environment, one of the data items provided to VectorCAST is the name of the unit under test. This unit under test is any Ada library unit. In the case of Ada 83 the unit under test would normally be a package. With the introduction of child packages in Ada 95, the VectorCAST unit under test will many times be a child package. The reason for this is that the child package is a basic building block for implementing the OOP concept of CLASS in Ada 95.

VectorCAST supports testing classes independently, by allowing test environments (harnesses) to be created for packages (base classes) OR for child packages (sub-classes). To test the implementation used in the above example, you would create three VectorCAST test environments and develop three separate sets of test data to stimulate them. The first test environment would be for the base class OBJECTS, the two other environments would be for the sub-class OBJECTS.DIMENSION2 and the sub-sub-class OBJECTS.DIMENSION2.DIMENSION3. As your code matures and new classes or sub-classes are added, you would add new VectorCAST test environments to exercise them. Existing test environments would not need to be modified in order to test the new classes.

The following sections contain a partial listing of the test artifacts that VectorCAST would generate for each class. For each class we show pseudo code for the driver, along with a single test case and the test results. Normally you would create several test cases for each operation.

CLASS: OBJECTS

```
Driver Pseudo Code

 case SUBPROGRAM is
 when 1 =>
  {result} :=
   OBJECTS.RANGE_IN_YARDS (
    CENTER => {data value});
 when 2 =>
  {result} :=
   OBJECTS.BEARING_IN_DEGREES (
    CENTER => {data value});
Test Case Data

Test Data =>

  UNIT => OBJECTS
   SUBPROGRAM => BEARING_IN_DEGREES
    CENTER (in )
     .XPOS => 199.00000000000000
     .YPOS => 201.00000000000000

Expected Values =>

  UNIT => OBJECTS
   SUBPROGRAM => BEARING_IN_DEGREES
    return => 44..46
```

```
Test Results


-----------------------------------------------------------------------
           Event 1
-----------------------------------------------------------------------

  IN UNIT => OBJECTS
   SUBPROGRAM => BEARING_IN_DEGREES
    CENTER (in )
     .XPOS => 1.99000000000000E+02
     .YPOS => 2.01000000000000E+02
    return => 0.00000000000000E+00


-----------------------------------------------------------------------
            Event 2
-----------------------------------------------------------------------

  IN UNIT => OBJECTS
   SUBPROGRAM => BEARING_IN_DEGREES
    CENTER (in )
     .XPOS => 1.99000000000000E+02
     .YPOS => 2.01000000000000E+02
    return => 4.47135238647461E+01            <match>
```

CLASS: OBJECTS.DIMENSION2

```
Driver Pseudo Code


 case SUBPROGRAM is
 when 1 =>
  {result} :=
   OBJECTS.DIMENSION2.AREA (
    R => {data value});
 when 2 =>
  {result} :=
   OBJECTS.DIMENSION2.AREA (
    C => {data value}1);
Test Case Data

Test Data =>

  UNIT => OBJECTS.DIMENSION2
   SUBPROGRAM => AREA
```

```
    R (in )
     .LENGTH => 10.00000000000000
     .WIDTH => 15.00000000000000

Expected Values =>

  UNIT => OBJECTS.DIMENSION2
   SUBPROGRAM => AREA
    return => 150

Test Results

-----------------------------------------------------------------------
           Event 1
-----------------------------------------------------------------------

  IN UNIT => OBJECTS.DIMENSION2
   SUBPROGRAM => AREA
    R (in )
     .LENGTH => 1.00000000000000E+01
     .WIDTH => 1.50000000000000E+01
     .XPOS => 0.00000000000000E+00
     .YPOS => 0.00000000000000E+00
    return => 0.00000000000000E+00

-----------------------------------------------------------------------
           Event 2
-----------------------------------------------------------------------

  IN UNIT => OBJECTS.DIMENSION2
   SUBPROGRAM => AREA
    R (in )
     .LENGTH => 1.00000000000000E+01
     .WIDTH => 1.50000000000000E+01
     .XPOS => 0.00000000000000E+00
     .YPOS => 0.00000000000000E+00
    return => 1.50000000000000E+02            <match>
```

CLASS: OBJECTS.DIMENSION2.DIMENSION3

```
Driver Pseudo Code

 case SUBPROGRAM is
 when 1 =>
  {result} :=
   OBJECTS.DIMENSION2.DIMENSION3.VOLUME (
    B => {data value});
```

```
 when 2 =>
  {result} :=
   OBJECTS.DIMENSION2.DIMENSION3.VOLUME (
    S => {data value});


Test Case Data
Test Data =>

  UNIT => OBJECTS.DIMENSION2.DIMENSION3
   SUBPROGRAM => VOLUME#2
    S (in )
     .RADIUS => 3.00000000000000

Expected Values =>

  UNIT => OBJECTS.DIMENSION2.DIMENSION3
   SUBPROGRAM => VOLUME#2
    return => 108..115

Test Results

-----------------------------------------------------------------------
           Event 1
-----------------------------------------------------------------------

  IN UNIT => OBJECTS.DIMENSION2.DIMENSION3
   SUBPROGRAM => VOLUME#2
    S (in )
     .RADIUS => 3.00000000000000E+00
     .XPOS => 0.00000000000000E+00
     .YPOS => 0.00000000000000E+00
    return => 0.00000000000000E+00

-----------------------------------------------------------------------
           Event 2
-----------------------------------------------------------------------

  IN UNIT => OBJECTS.DIMENSION2.DIMENSION3
   SUBPROGRAM => VOLUME#2
    S (in )
     .RADIUS => 3.00000000000000E+00
     .XPOS => 0.00000000000000E+00
     .YPOS => 0.00000000000000E+00
    return => 1.13097335815430E+02            <match>
```

Using the Multi-unit Whitebox feature to Manipulate Private Types

One of the advantages of the Ada language is the concept of “private types” – types whose

implementation is hidden from clients of the type. While this capability is useful for data hiding and modularity, it makes testing units that reference these types more difficult, because the test driver and/or stub cannot manipulate objects of these types. VectorCAST has always provided a “whitebox” capability, allowing the tester to manipulate private types that are defined in the Unit Under Test.

The “Multiunit whitebox” option provides the ability to manipulate private types that are defined in the specification of *any* package, not just the unit under test. This enables the user to set up objects using the standard GUI and test scripting available with VectorCAST, rather than having to write User Code to set / examine these types. This capability, called Multiunit Whitebox, is not a replacement for the existing whitebox capability – both methods of building test harnesses are supported.

To access this new capability, the tester must activate this option prior to building the environment via one of the following methods:

From the GUI: Go to **Tools => Options dialog, Builder tab**, select the Multiunit Whitebox checkbox

From CLICAST: issue the command:

```
clicast option vcast_multiunit_whitebox true
```

Once an environment is built with the Multiunit Whitebox capability, data entry for objects of private types (via the GUI or test scripting) is no different than data entry for objects of normally visible types.



Note: The Multiunit Whitebox feature is implemented using Ada 95 child package constructs and, as such, is available only to compilers / targets that support Ada 95. If this option is set and the compiler or target does not support Ada 95, or the option to force Ada 83 compliance is set, the Multiunit Whitebox option will be ignored.

Working with Arrays

To Enter Values for Constrained Array Types

Array Types that contain a fixed number of components are called *constrained arrays*. Consider the following function prototype:

```
type int_array is array (1..3) of integer;
function Test_int_array(array_param : gui_stub.int_array)
                        return gui_stub.int_array;
```

This array has three elements. You can enter an input and expected value for a specific array element, or set multiple array elements to the same value. The display of array parameters and global objects is “collapsed” by default, meaning that only a single entry appears in the Parameter Tree. You can “expand” this display and then enter data for each expanded elements, or you can specify a data value and apply it to specific elements. These methods are discussed below.

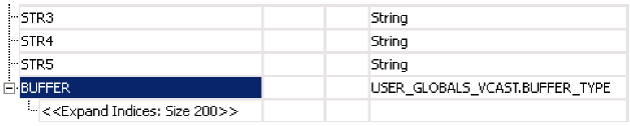
The purpose of expanding or collapsing array data is to enable you to control the size of the Parameter Tree.

To Expand All Elements of an Array

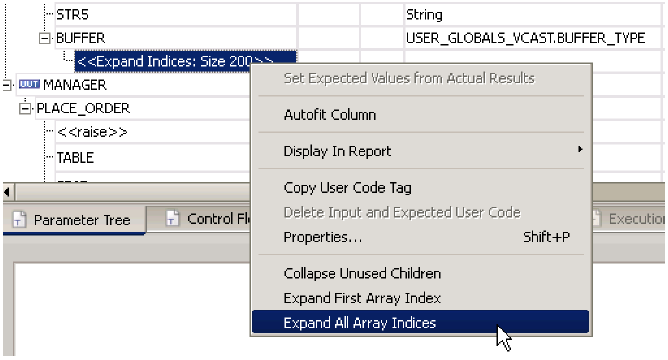
When a parameter is an array type, use the plus  button to expand out the array. For example,

expand <<USER_GLOBALS_VCAST>>, then expand <<GLOBAL>> and scroll down to see the user global array called BUFFER, an array of 200 elements, and click the plus button. The notation for unexpanded elements appears as:

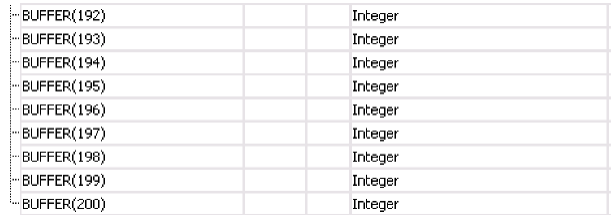
<<Expand Indices: Size n>>, where n is the number of elements in the array.



To expand all elements of the array, right-click on <<Expand Indices>> and choose **Expand All Array Indices**.

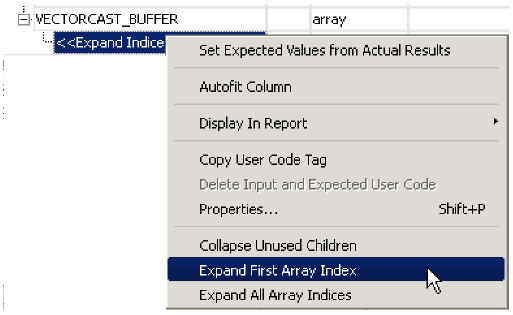


All array elements are expanded in the Parameter Tree.



To Expand the First Element of an Array

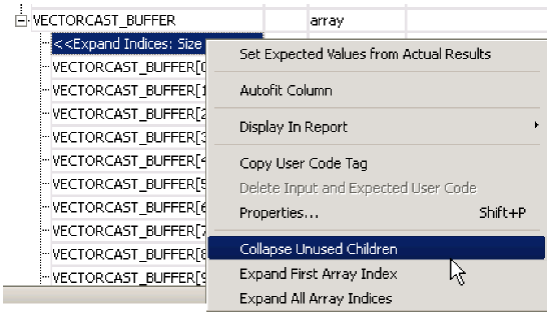
To expand only the first element of the array, right-click on <<Expand Indices>> and choose **Expand First Array Index**. The first array element is expanded in the Parameter Tree.



To Collapse Unused Elements of an Array

If you have expanded some elements of an array and you find that you don't need to enter Input or Expected data for them, you can collapse them in the Parameter Tree.

To collapse unused array elements, right-click on <<Expand Indices>> and choose **Collapse Unused Children**.

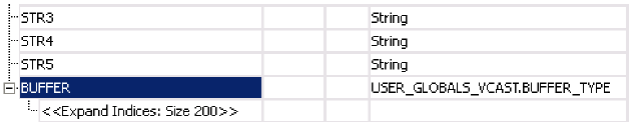


The array elements that do not have Input or Expected data are removed from the Parameter Tree. In the example below, all 200 elements of the VECTORCAST_BUFFER had been expanded, but data was entered for the element VECTORCAST_BUFFER [01]. When the unused array elements were collapsed, all array elements are closed except for index [01], which has data.

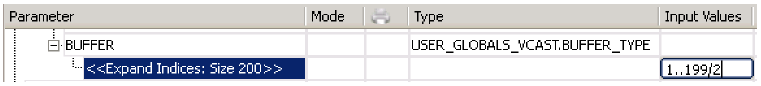


To Expand Certain Elements of an Array

When a parameter is an array type, use the ⊞ button to expand out the array. For example, expand <<USER_GLOBALS_VCAST>>, then expand <<GLOBAL>> and scroll down to see the user global array called BUFFER, an array of 200 elements, and click the ⊞ button. The notation for unexpanded elements appears as <<Expand Indices: Size *n*>>, where *n* is the number of elements in the array.

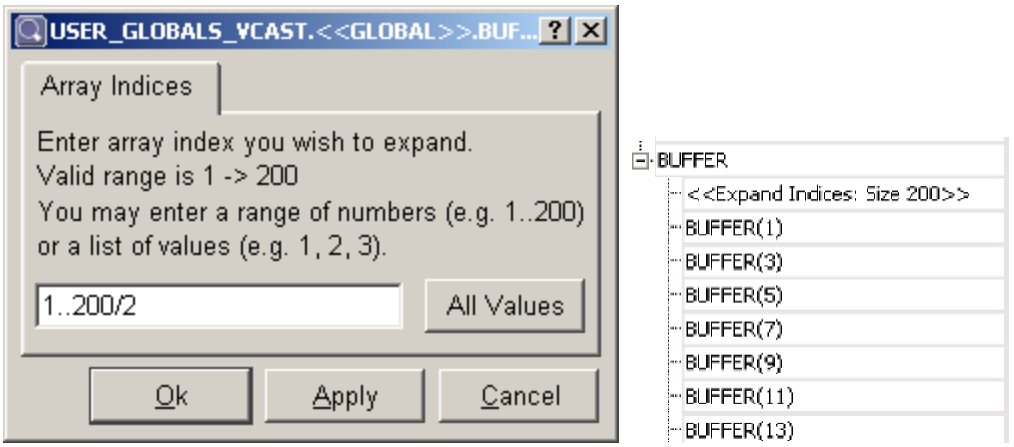


There are several ways to expand (or display) certain elements of this array. In the Input Value or Expected Value edit box in the Parameter Tree tab, you can type a single index, a range, or a list. In the example below, the *odd* array elements from 1 to 199 are specified.



Array Indices Dialog

Another way to expand the array is to double-click <<Expand Indices>> to bring up the Array Indices dialog box. Here you can type either a single element (e.g. 77), a range (e.g. 512..517), or a list of specific elements (e.g. 5,10,15,20). Specifying a range such as 1..200/2 indicates expand the array elements 1 to 200, but skipping one in between. The result is the odd array elements are expanded.



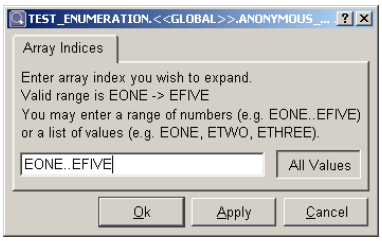
To expand the entire array, click the **All Values** button.

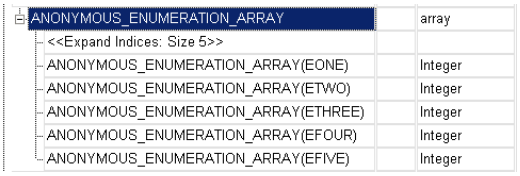
Click **OK** to exit the Array Indices dialog and expand the array.

(example: screen_shot_environments/ADA/test_enumeration.vce)An example of an enumerated array index type is shown below. Here, the array ANONYMOUS_ENUMERATION_ARRAY has 5 elements. When all elements are expanded, VectorCAST shows each element with the proper enumerated array index.

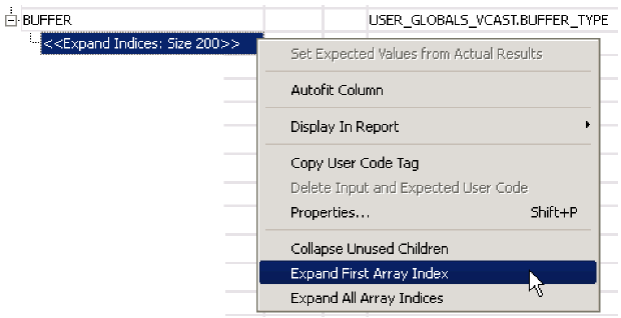
```
type Enumeration_Index_Type is (
   EONE,
   ETWO,
   ETHREE,
   EFOUR,
   EFIVE );
```

```
Anonymous_Enumeration_Array : array (Enumeration_Index_Type) of INTEGER;
```



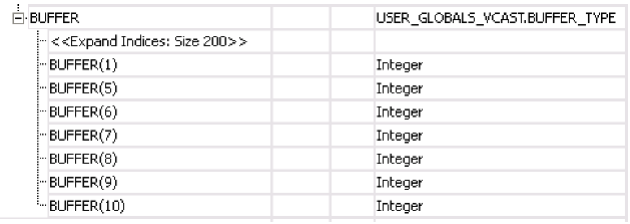


Another way to expand array elements is to right-click on **<<Expand Indices>>** to display the pop-up menu:



- Selecting **Properties** from this pop-up menu takes you to the Array Indices dialog described above.
- Selecting **Collapse Unused Children** causes any elements that have no data to collapse, minimizing the size of the Parameter Tree.
- Selecting **Expand First ArrayIndex** causes the first element of the array to expand.
- Selecting **Expand All ArrayIndices** causes all elements of the array to expand.

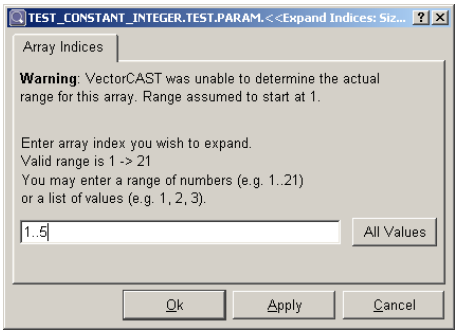
For example, in the figure below, the global array BUFFER has been expanded twice. The first array element was expanded, and then elements 5..10 were expanded. Data can now be entered into these elements.

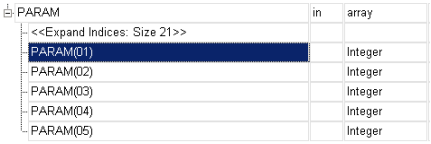


In some special cases, VectorCAST is not able to determine the index type for the array. These cases usually involve constants or expressions in the definition of the array in the package. Therefore, the array indices are presented in the ‘pos notation. This notation is also used in cases where the Advanced option “Disable Direct Array Indexing” is on.

```
Constant_Integer : constant Integer := 100;
type vector_constant_first is array ( Constant_Integer .. 120 ) of integer;
```

In this case the array index values selected would be 1 through 5, as shown below.





To Apply Values to an Array

If you bring up the Parameter dialog on an expanded array element, you can enter input values or expected values to one or more elements of an array. The Scalar Values, Range Values, and List Values tabs have an added section, called “Apply to Array Indices,” located near the bottom of the dialog.



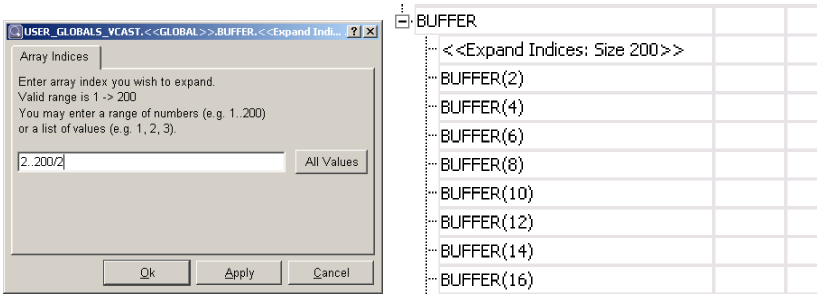
Once you have enabled Input or Expected values and filled in the data, you can easily apply these values to multiple elements of the array. The choices are:

- **This** (default) to set the one element that you expanded and double-clicked
- **All** to set all elements of the array
- **Range** to set a range of elements, (x..y, or 1,3,5,7, or 1..200/2 etc.).

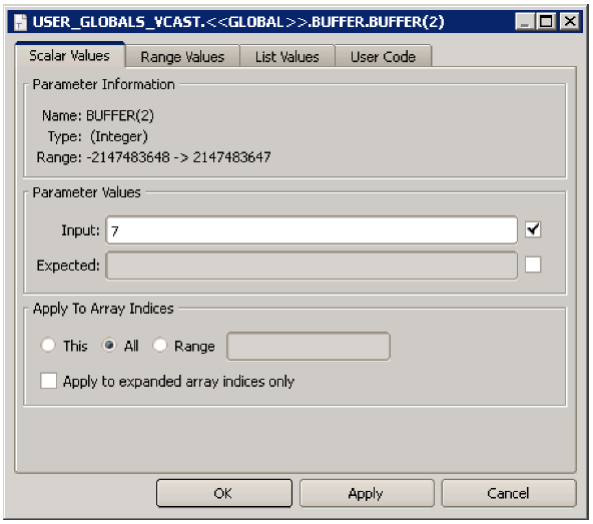
The **Apply to expanded array indices only** checkbox modifies the “All” or “Range” options by limiting their effect to the currently expanded elements in the Parameter Tree.

For example, if you expanded elements 2 to 200, even only, using the Expand Indices dialog, you could then assign a value using the “All” choice in the Parameter dialog. It would only apply to the even elements.

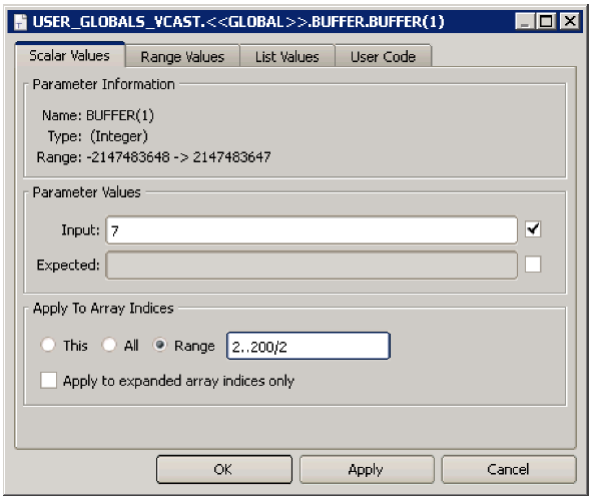
First, expand the even elements as shown below:



Then, double-click on any element to bring up the Parameter dialog. Set an input value, then click the "All" radio button and the checkbox next to "Apply to expanded array elements only."



Alternatively, you could skip the step where you expand the even elements and instead expand the first element, double-click it to go directly to the Parameter dialog, and use the Range radio button with the expression 2..200/2.

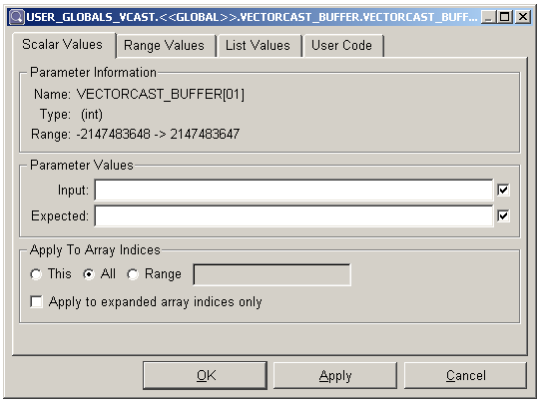


To Clear Values from an Array

To clear scalar, range, or list data from an array, do the following:

1. Check the Enable box(es).
2. Delete the text in the Input and/or Expected boxes.
3. Select the All or Range radio button.
4. Click **Apply**.

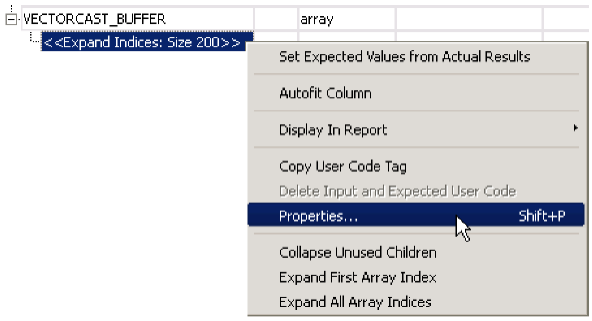
Doing so clears the data from all of the indicated range of array elements.



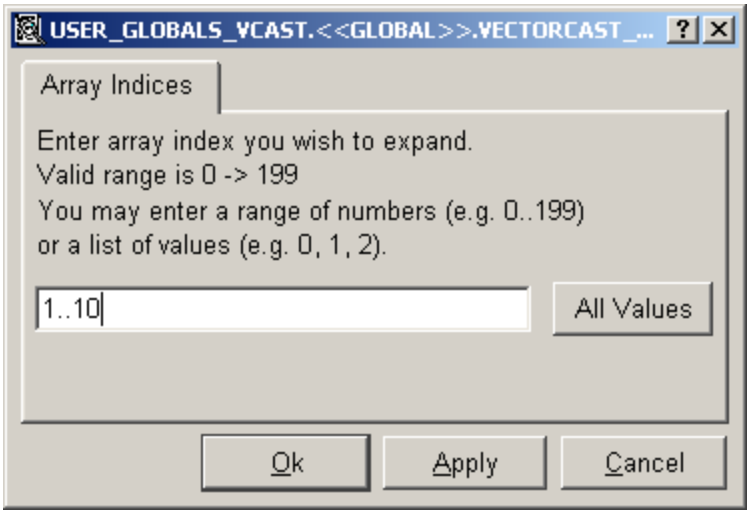
Using the Array Properties Dialog

There are two ways to access the Array Properties dialog:

- in the Parameter Tree, double-click on an array's <<Expand Indices: Size n>>
- in the Parameter Tree, right-click on an array's <<Expand Indices: Size n>> and choose **Properties...**



You can enter an expression to control which array elements are expanded or to expand the entire array, click the **All Values** button. The syntax of expressions that can be entered is described in the previous section.



Click **OK** to exit the Array Indices dialog and expand the array.

Multi-Dimensional Arrays

The index values for multi-dimensional arrays should be selected via the comma-separated expressions: [1, 4]

Unconstrained Arrays

VectorCAST supports dynamic modifications to the size of unconstrained array parameters for single dimension arrays only. This functionality enables you to define, for any subprogram parameter that is a one-dimension unconstrained array, a slice that you want to consider as active for that test case.

For example, given the following:

```
type INDEX_TYPE is integer range 1..10;
type ARRAY_TYPE is array (INDEX_TYPE range <>);

procedure COMPUTE (VALUE : in ARRAY_TYPE);
```

You can construct test cases which pass different sized arrays into COMPUTE. For each test case you will select lower and upper bounds for the VALUE parameter.

In cases where the array size is unknown or very large, you can set the upper-bound or both the upper- and lower-bounds. The setting is called “Unconstrained Array Size,” on the **Tools => Options dialog, Builder tab**.

Modifying Array Constraints

When you select a SINGLE dimension unconstrained array parameter for modification, the parameter name is displayed with an asterisk (*) preceding the name. Array values will be entered as normal. However, there will be two additional choices in the test case hierarchy. The choices will be labeled "first" and "last". By using these choices, you will be able to set the slice of the unconstrained array that you want active for the test case.



Note: To avoid constraint errors, it is important that you define the lower and upper array constraints for "in" AND "out" parameters. For instance, if you know that a subprogram is going to return a parameter which is ARRAY_TYPE (2..6) for a particular test case, you need to set the array constraints for that parameter with VectorCAST before running the test.

Working with Unconstrained Array Parameters

When building environments in which some of the parameters are unconstrained array types you may get compile warnings, link errors, or execution errors if the maximum size of the unconstrained array type is larger than the compilation system enables.

In VectorCAST, when an environment is built, an Ada object is defined for each parameter that exists in the particular closure being tested. In the case of unconstrained array types, that object is always constrained with the maximum size of the index type that was used to define the type. For example given the following declarations:

```
type ARRAY_TYPE is array (integer range <>) of integer;
procedure PROCESS_ARRAY (DATA : in  ARRAY_TYPE);
```

VectorCAST would define an object to map to the parameter for subprogram PROCESS_ARRAY. This object would be declared as the size of the parameter based on the Ada type. For example, in this case the following would be generated:

```
P0012_001_001 : ARRAY_TYPE (integer);
```

This creates a problem though when the compilation system restricts the maximum size of objects. In this example the object declaration requires 4 bytes per array item and the array has an index range of -2**31-1 .. 2**31-1 which is over 4 billion items. This will, with most compilations systems, generate a compiler warning, or a run time exception. To accommodate this limitation, VectorCAST restricts the size of unconstrained array objects of this type to 2000 items. The actual declaration that VectorCAST generates for the above example would be:

```
P0012_001_001 : ARRAY_TYPE (integer range -1000..1000);
```

This automatic range restriction occurs whenever the index type of the array is one of the following package STANDARD types: NATURAL, INTEGER, POSITIVE.

The 1000 used in this example is the VectorCAST default. This value can be modified by the user via the **Tools => Options dialog**.

You will however still get the compiler warning is cases where there are user-defined types that have sizes comparable to integer, because VectorCAST is unable to automatically restrict the index range in these cases.

For example, given the following declarations:

```
type INDEX_TYPE is range -2**31-1..2**31-1;
type ARRAY_TYPE is array (INDEX_TYPE range <>) of integer;
procedure PROCESS_ARRAY (DATA : in  ARRAY_TYPE);
```

VectorCAST will generate:

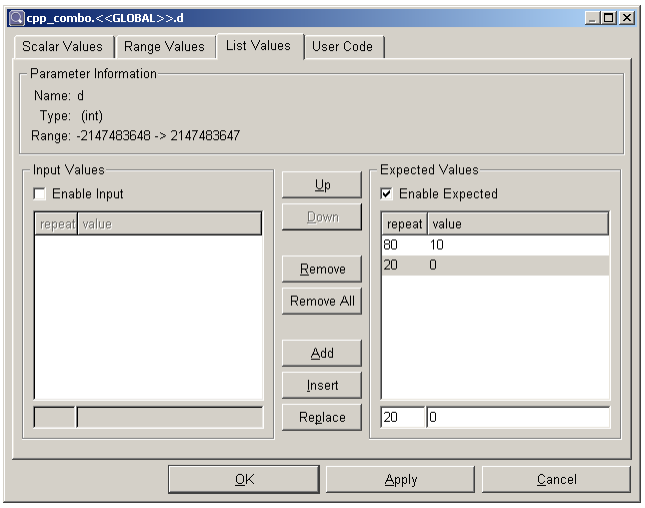
```
P0012_001_001 : ARRAY_TYPE (INDEX_TYPE);
```

To solve this problem, VectorCAST provides a configuration option called VCAST_USER_CONSTRAINTS. This option enables you to provide a comma-separated list of additional types that should be restricted when they are used as unconstrained array index types.

So, for this example, you would set VCAST_USER_CONSTRAINTS to: INDEX_TYPE.

Note: If the array object index uses full dot notation, you must use full dot notation when setting the variable (e.g. MY_TYPES.INDEX_TYPE).

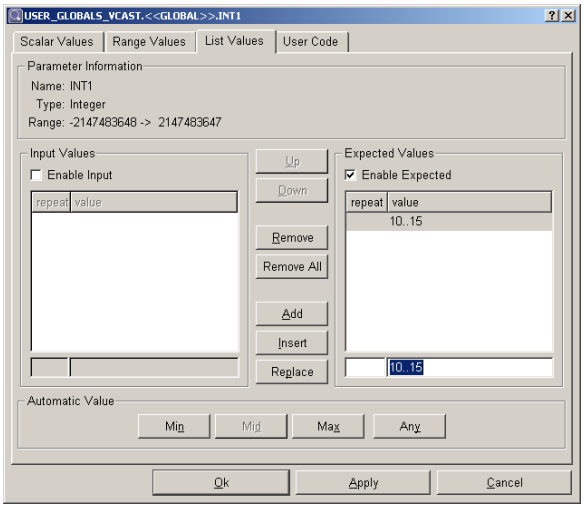
Entering Data in the Parameter Dialog Box



To Use a Range Expression in an Expected Value List

When working with integer or float values, you may also provide a range expression in the list of expected values. Simply type in the data in low..high format and that range is added to the list. When an expected value is compared to the range, it matches if it is in the range.

Ranges are not permitted in the Input Values list. If you need an input range, you can express it as an input list, or just go to the Range Values tab and add the input range there.



To Use the <<Any>> Tag in a List

When you provide a list of expected values, VectorCAST uses each value in the list in the order it is encountered. In some cases you may not care what the value of an object is until a specific point in the test execution. The **Any** tag enables you to indicate an “I don’t care” state for the data value. For example, the following screen shot assumes that the Entree field in parameter Order will be encountered 4 times, and the tester only cares what the value is on the fourth iteration.

A value of <<ANY>> does not get compared; it is merely a placeholder, so that other values are compared on the desired iteration.

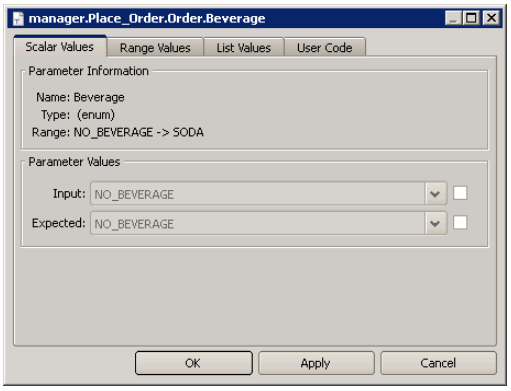
To Access the Parameter Dialog

The Parameter dialog gives you easy access to both Input Value and Expected Value of a parameter, as well as Min, Mid, and Max values, ranges of values, and lists of values.

There are two ways to access the Parameter dialog:

- in the Parameter Tree, double-click on any parameter or global object
- in the Parameter Tree, right-click on a parameter and choose **Properties...**

For example, double-clicking the parameter **`Beverage`**, in the data object Order, in the subprogram Place_Order, in the UUT manager brings up the Parameter dialog as shown below:



The Parameter dialog box contains the following four tabs:

- Scalar Values
- Range Values
- List Values
- User Code

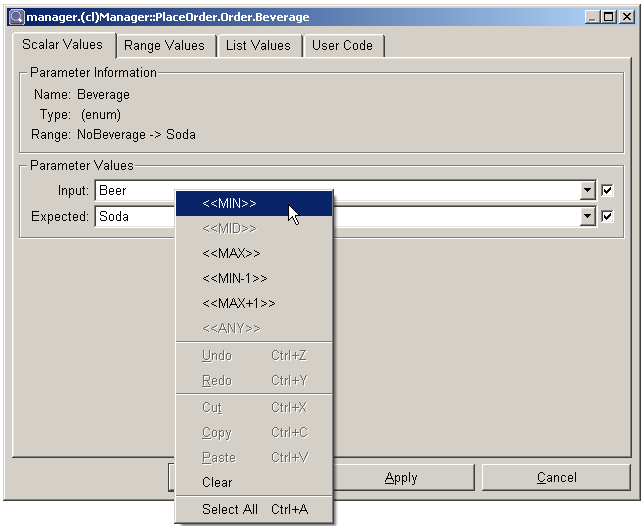
All tabs contain a header which shows the base type of the data object as well as the valid range. Note that the valid range saves you from having to search through your source code files to find the range of a particular data item.

In all of the tabs, the small checkboxes next to the data values are used to "enable" the data value for the test case. Before entering an Input or Expected value, you must click on the checkbox to activate that value. Similarly, if you "unchecked" a value, it is deactivated and the value is removed from the test case.

The Scalar Values Tab

The Scalar Values tab enables you to define an individual value to be assigned to the data object. Right-click to set the **Min-1**, **Min**, **Mid**, **Max**, or **Max+1** values. (<<MID>> values are not allowed for enumerated parameters.)

When you enable the Expected checkbox, the **Any** menu item also becomes available. It is used only with expected values, and indicates that you want to allow any value for that parameter's expected value. Since <<Any>> is a data value, it is recorded in the test execution results. Use <<Any>> if you want to see the value of a non-global parameter in the test execution results without making an actual comparison.



The items in the right-click menu depend on the data type of the parameter. Floats have <<ZERO>>, <<POS_INF>>, <<NEG_INF>>, and <<NAN>>. Strings only have <<ANY>> (for expected values). Enumerals do not have <<MID>> or <<ZERO>>.

To Apply Values to an Array

If you bring up the Parameter dialog on an expanded array element, you can enter input values or expected values to one or more elements of an array. The Scalar Values, Range Values, and List Values tabs have an added section, called “Apply to Array Indices,” located near the bottom of the dialog.



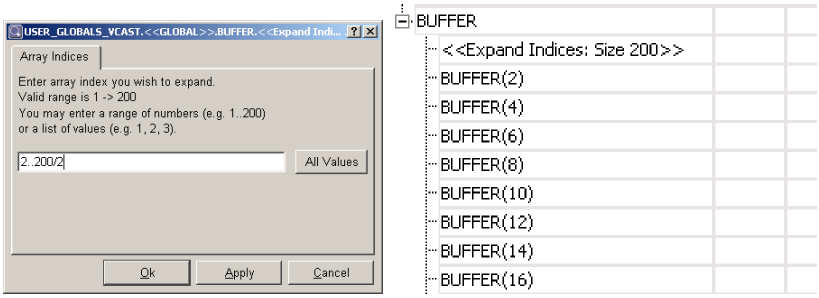
Once you have enabled Input or Expected values and filled in the data, you can easily apply these values to multiple elements of the array. The choices are:

- **This** (default) to set the one element that you expanded and double-clicked
- **All** to set all elements of the array
- **Range** to set a range of elements, (x..y, or 1,3,5,7, or 1..200/2 etc.).

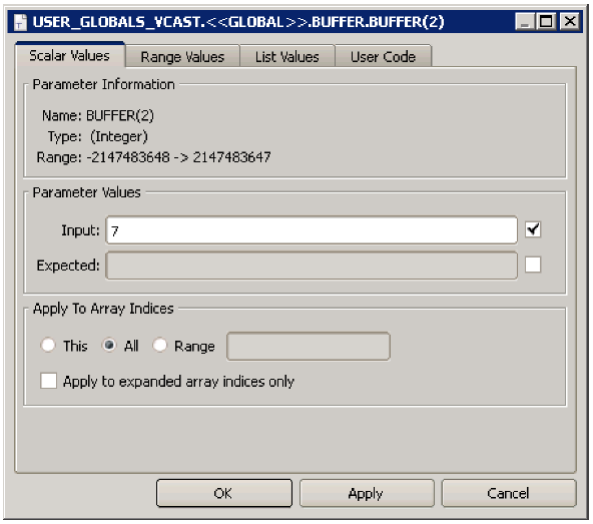
The **Apply to expanded array indices only** checkbox modifies the “All” or “Range” options by limiting their effect to the currently expanded elements in the Parameter Tree.

For example, if you expanded elements 2 to 200, even only, using the Expand Indices dialog, you could then assign a value using the “All” choice in the Parameter dialog. It would only apply to the even elements.

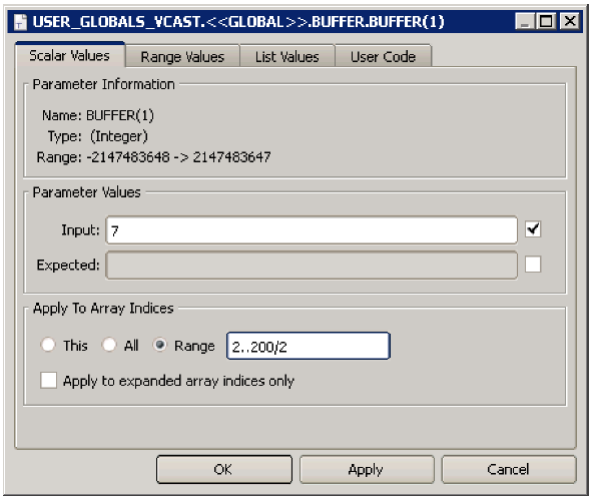
First, expand the even elements as shown below:



Then, double-click on any element to bring up the Parameter dialog. Set an input value, then click the "All" radio button and the checkbox next to "Apply to expanded array elements only."



Alternatively, you could skip the step where you expand the even elements and instead expand the first element, double-click it to go directly to the Parameter dialog, and use the Range radio button with the expression 2..200/2.

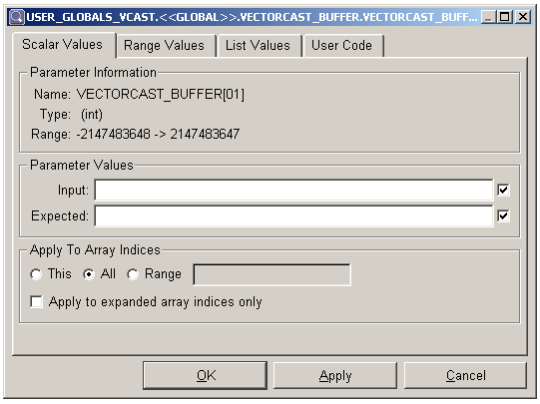


To Clear Values from an Array

To clear scalar, range, or list data from an array, do the following:

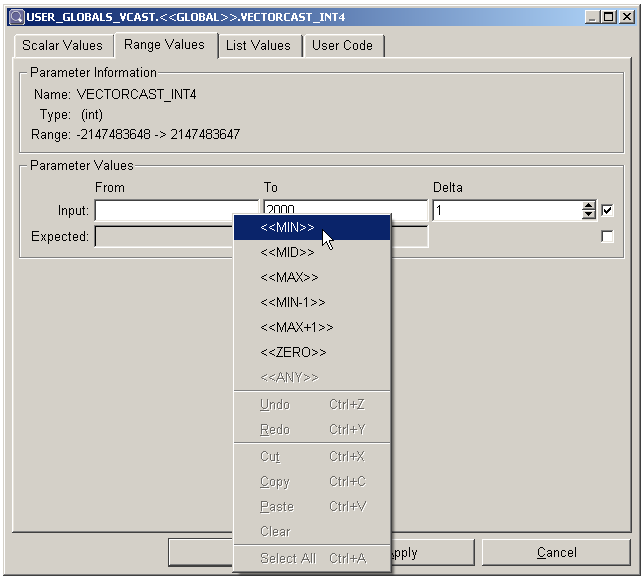
1. Check the Enable box(es).
2. Delete the text in the Input and/or Expected boxes.
3. Select the All or Range radio button.
4. Click **Apply**.

Doing so clears the data from all of the indicated range of array elements.



The Range Values Tab

The Range Values tab enables you to build a test case that will automatically exercise a parameter over a specified range of values. As with scalar values, once you have enabled, or "activated" the input or expected value, then right-click to special values <<Min-1>>, <<Min>>, <<Mid>>, <<Max>>, or <<Max+1>>. If the data type is enum, then the actual minimum, mid, or maximum value is displayed once you click **Apply** or **OK**.



The **From** edit box is the first value used, and the **To** edit box contains the last value used each time the subprogram is called. These two numbers represent the range, and the **Delta** is used to get from the first number to the second. For example, to vary an input from 1 to 6 by 2, use:

From: 1

To: 6

Delta: 2

Input values used for parameter: 1, 3, 5

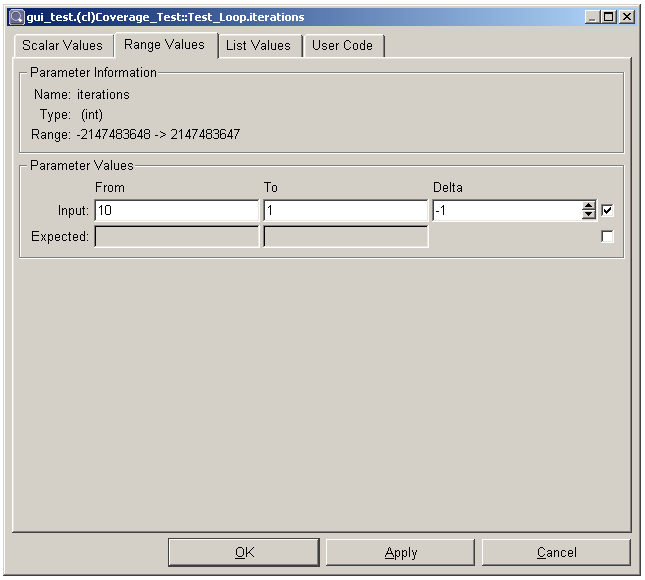
To enter a descending range, use a negative delta:

From: 10

To: 1

Delta: -1

Input values used for parameter: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.



To Set Up an Input Range or List for More Than One Parameter

When setting up a test case for range testing, you may vary different parameters simultaneously, in parallel or in combination. This means that different range expressions may exist within a single test case. A range in the Input Values column causes the test case to iterate. This capability enables you to vary multiple values simultaneously, or vary one value while holding the others constant. The default for the maximum number of parameters to vary is 20. This value can be modified using the “Maximum varied parameters” option, located on the **Target** tab of the **Tools => Options** dialog.

To Use Ranges as Expected Values

The Range tab can also be used to set an expected value. When used this way, an actual value that is between the “From” and “To” values, inclusive, is considered a match when the test case runs.

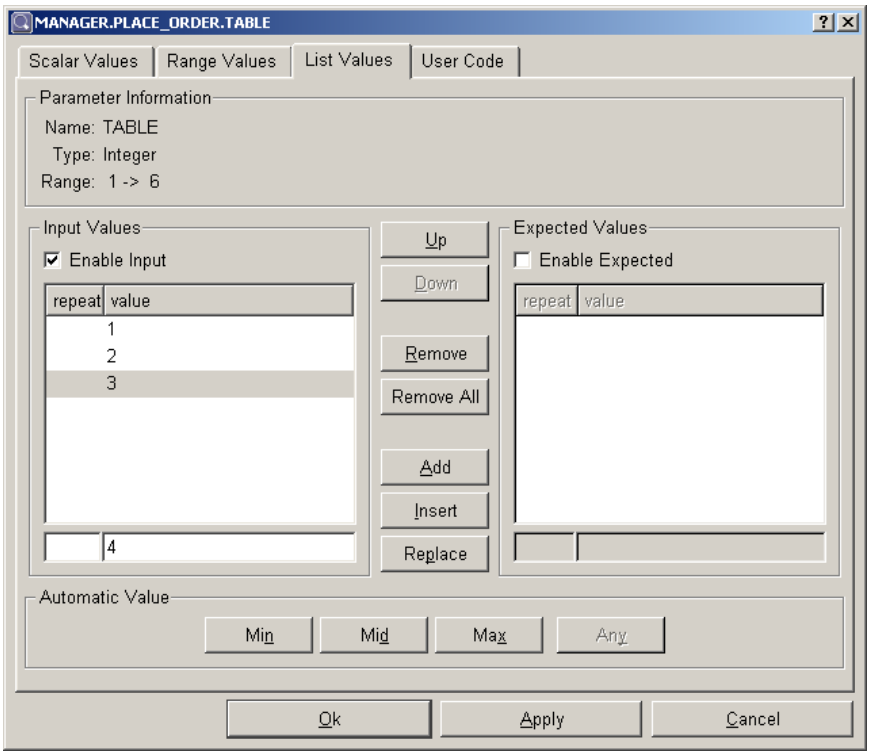
The List Values Tab

The List Values tab enables you to provide a series of values to be used when calling the unit under test or to provide a sequence of specific scalar values to be returned from stubs when a stub is called multiple times during a test case execution. A list is similar to a range, except the items in a list can have different deltas between them, so you specify each item in the list. Items in a range are always separated by the same delta, so you only need specify the beginning and end items, and the delta.

When you add items to the Input Values list for a parameter, you are assigning a series of input values to that parameter. Adding more than one item to the list causes an iteration, that is, the test case is executed once for each item. Repeating a value in the list causes additional iterations.

Using an Input Values list for a stub parameter means that the stubbed subprogram returns one item from the list each time it is called. If a stub is called more time than you have provided values for in your list, then the last value in the list is used repeatedly, once the list items are exhausted. Each time a test case iterates, the Input Values list for a stub is reset to the beginning.

When you add items to the Expected Values list for the same or a different parameter, you are assigning a series of expected values for that parameter. One expected value is compared to the parameter's value per test iteration. The expected value can be a range of acceptable values for that iteration, as described below.



Controlling Values in the List

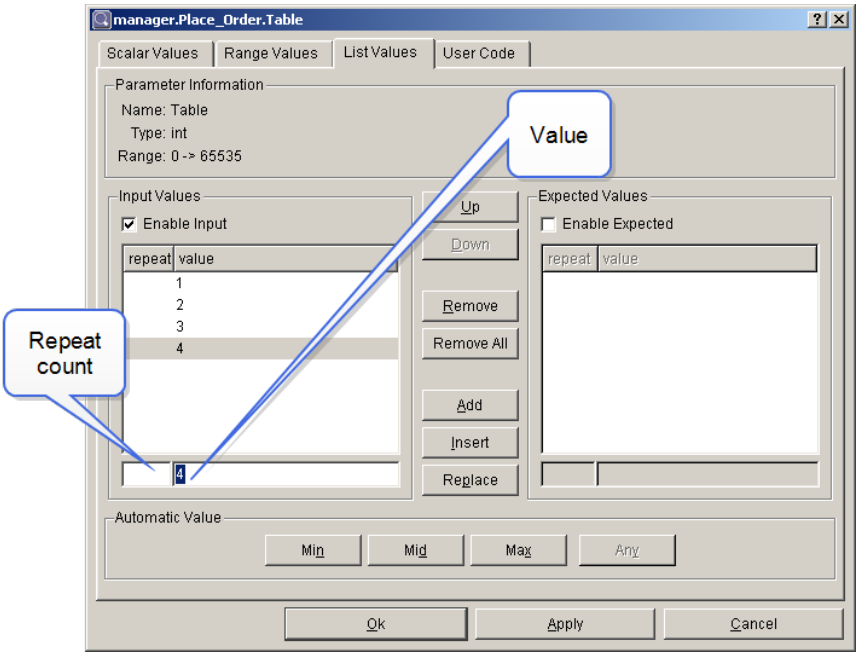
To enter values in a list, click the checkbox next to "Enable Input" or "Enable Expected." The edit boxes below become active, allowing you to type a value. To add a value, enter a value in the edit box below the list area, and click the **Add** button or press the **Enter** key. Repeat to add more entries to the

bottom of the list.

To insert a new entry into an existing list, select an item in the list, enter a new value, and click the **Insert** button. The new entry appears below the selected one. To change an entry already in the list, first select it. Make the change in the text area below (either in the repeat box, the value box, or both), and click the **Replace** button. The entry in the list is replaced with the new values.

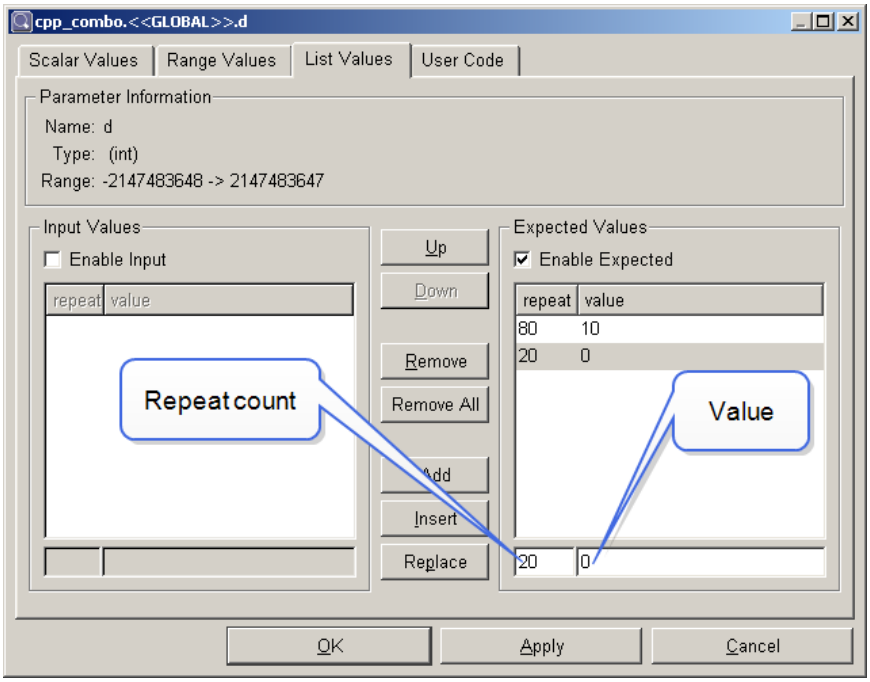
An entry can be moved up or down in the list by selecting the entry and clicking the **Up** or **Down** button.

An entry can be deleted by clicking the **Remove** button. When you click the **Remove All** button, all entries are deleted from the list.



To Repeat Values in the List

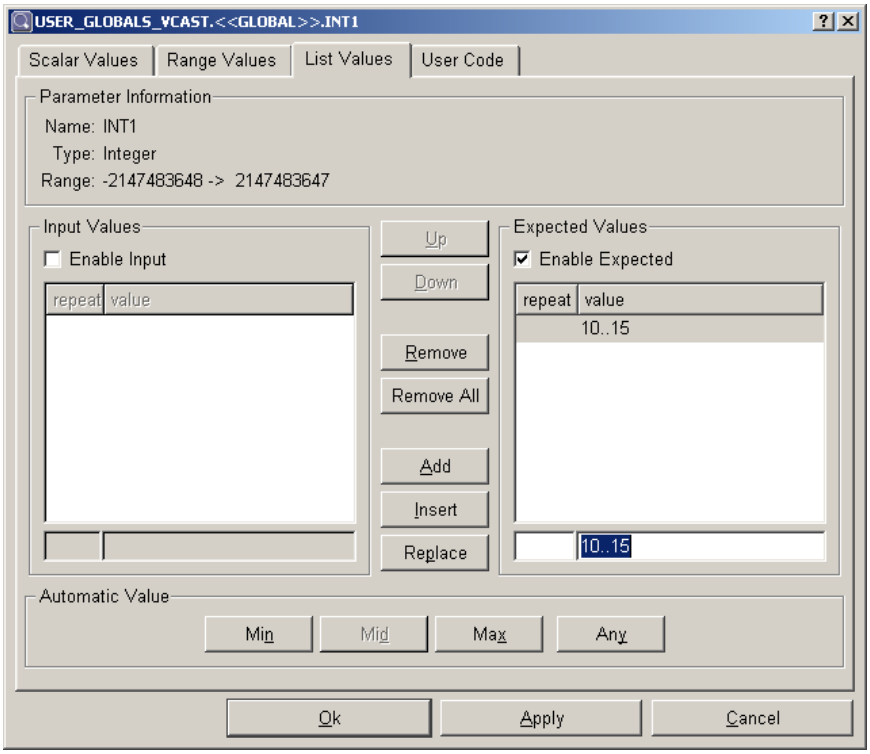
Each value in a list can have an optional repeat count. The repeat count enables you to easily manage lists of values when there are repeating values in the list. For example, if you want to have a list of 100 integers where the first 80 values are 10 and the next 20 values are 0, you enter these values using the repeat count as shown below.



To Use a Range Expression in an Expected Value List

When working with integer or float values, you may also provide a range expression in the list of expected values. Simply type in the data in low..high format and that range is added to the list. When an actual value is compared to the range, it matches if it is in the range.

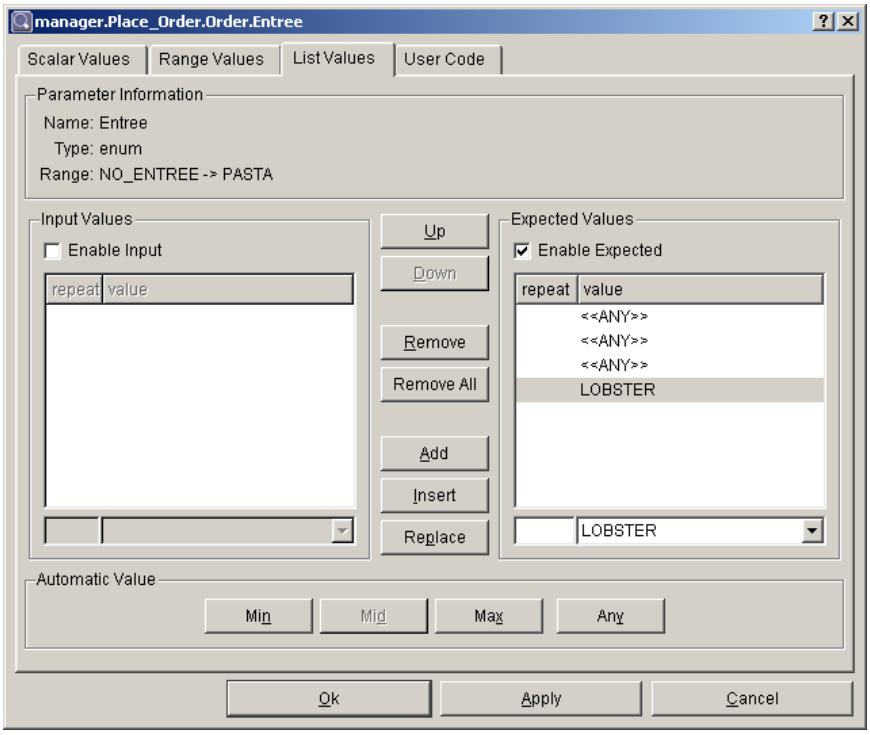
Ranges are not permitted in the Input Values list. If you need an input range, you can express it as an input list, or just go to the Range Values tab and add the input range there.



To Use the <<Any>> Tag in a List

When you provide a list of expected values, VectorCAST uses each value in the list in the order it is encountered. In some cases you may not care what the value of an object is until a specific point in the test execution. The **Any** tag enables you to indicate an “I don’t care” state for the data value. For example, the following screen shot assumes that the Entree field in parameter Order will be encountered 4 times, and the tester only cares what the value is on the fourth iteration.

A value of <<ANY>> does not get compared; it is merely a placeholder, so that other values are compared on the desired iteration.



Test Case Scripting

In addition to being used to archive test cases for regression use, VectorCAST test scripts enable you to build test cases using ASCII text files. Scripting can be an alternative to entering test case data interactively, via the VectorCAST GUI.

Note: Test Case Scripting is an advanced workflow which can have unintended side effects and should be used with caution.

VectorCAST Scripting complements the normal test case creation process. With Scripting you can create a single test case using the VectorCAST GUI, export the script file, edit the file to contain multiple similar test cases, and import the script to quickly generate tens or hundreds of test cases.

Additionally, test case scripting enables you to import data from other tools, such as MATLAB™, that may contain high precision models or data sets.

To Export Test Cases to a Test Script

The **Test => Scripting => Export Script** command provides a means to output test case data to a test script.

The Export Script command is sensitive to what is currently selected in the Test Case Tree. You can click (or multi-select) nodes from the Test Case Tree, then choose **Test menu => Scripting => Export Script**. When you click **Save**, VectorCAST saves the file in the working directory (by default) with the extension .tst. The file contains test data for all selected test cases.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <test>]]] TESt Script CReate
<script file>

Create (export) test script file from existing test case(s). If only the
environment parameter is specified, the script will contain data for all
test cases (including <<init>> and <<compound>>). If the unit parameter is
specified, the script will contain data for all testcases for that unit. If
the unit and subprogram parameters are specified, the script will contain
data for all testcases for the specified subprogram. If unit, subprogram,
and testcase parameters are specified, the script will contain data for the
specified test case.
```

Duplicate Names in Exported Scripts

If any of the test cases in Unapplied Test Script Data have the same name as existing test cases, then the exported Unapplied Test Script Data test cases are assigned new names. This name follows the default naming format described in "Test Case Naming" on page 157.

Any name changes of exported tests from Unapplied Test Script Data are propagated through the compound test slots in Unapplied Test Script Data.

To Clear Unapplied Test Script Data

To remove any unapplied test script data from the environment, right-click the Unapplied Test Data node in the Test Case Tree and select **Delete Unapplied Test Data Script** from the menu.



Note: This permanently removes the unapplied test script from the environment and cannot be undone.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <test>]]] TESt UNapplied Clear

Clear unapplied test script data from environment.
```

To Import Test Cases from a Test Script

The **Test => Scripting => Import Script** command enables you to import a test case script. When you choose **Test => Scripting => Import Script**, the File Open dialog appears. Here, select one or more test scripts (.tst files) and click **Open**. The test scripts are loaded into the environment, creating new test cases.

When a test script is imported, whether by you or as part of another operation (such as Updating or rebuilding an environment), the Test Script Log appears in the MDI window. If any test cases had to be renamed or other errors occurred, they are documented in the Test Script Log.

The Test Script Log is displayed in the MDI window after a test script is imported, using **Test => Scripting => Import Script** or during another operation, such as rebuilding or updating the environment. To view the Test Script Log at any other time, choose **Test => View => Scripting Log**.

Duplicate Names in Imported Scripts

If any of the test cases in the script have the same name as existing test cases, then the imported test

cases are assigned new names. This name follows the default naming format described in “Test Case Naming.”

If an imported compound test case includes an individual test case that must be renamed, the new name is propagated to the compound test case, so that the imported test cases remain consistent.

```
clicast -e <env> TESt Script Run <scriptfile>

Load (import) test cases from a script file.
```

```
clicast -e <env> TESt Script Log

Display to standard output the Test Script Log that was generated by the
last test script import operation.
```

See also "Strict Test Case Importing" on page 289.

See also "Test Script Language" on page 465.

To Create a Test Case Template

The **Test => Scripting => Template** command builds a test case script that contains all possible input and output data for a particular environment. The templates contain all global data, formal parameters of the unit(s) under test, and formal parameters of any stubbed dependent unit. The purpose of the test script template is to show you the correct syntax for every possible TEST.VALUE line for the environment. In addition to showing the command syntax, each line shows the range of allowable values for each parameter.

```
clicast -e <env> TESt Script Template <scriptfile>

Create a test script template containing entries for each parameter and
global object in the environment, along with range information, and save it
in <scriptfile>.
```

The following template excerpt is from the manager.adb tutorial:

```
-- Test Case Script
--
-- Environment : TEST
-- Unit Under Test: MANAGER
--
-- Script Features
TEST.SCRIPT_FEATURE:ADA_DIRECT_ARRAY_INDEXING
--
-- Values are mandatory for SUBPROGRAM and NAME.
-- Delete any TEST.VALUE or TEST.EXPECTED commands not used.
--
TEST.SUBPROGRAM:
TEST.NEW
TEST.NAME:
```

```
TEST.NOTES:
TEST.END_NOTES:
TEST.VALUE:MANAGER.PLACE_ORDER.TABLE: 1.. 6
TEST.VALUE:MANAGER.PLACE_ORDER.SEAT: 1.. 4
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SOUP: NO_ORDER..CHOWDER
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SALAD: NO_ORDER..GREEN
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.ENTREE: NO_ORDER..PASTA
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.DESSERT: NO_ORDER..FRUIT
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.BEVERAGE: NO_ORDER..SODA
TEST.VALUE:MANAGER.CLEAR_TABLE.TABLE: 1.. 6
TEST.VALUE:MANAGER.GET_CHECK_TOTAL.TABLE: 1.. 6
```

Troubleshooting Test Script Template

Range Checking Set to None Before Creating Template

If you have Range Checking set to None in the **Tools => Options** dialog, **Execute tab** when you rebuild the environment , the template generated looks like this:

```
TEST.SUBPROGRAM:
TEST.NEW
TEST.NAME:
TEST.NOTES:
TEST.END_NOTES:
TEST.VALUE:MANAGER.PLACE_ORDER.TABLE: <<unknown>>..<<unknown>>
TEST.VALUE:MANAGER.PLACE_ORDER.SEAT: <<unknown>>..<<unknown>>
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SOUP: <<unknown>>..<<unknown>>
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SALAD: <<unknown>>..<<unknown>>
...
```

The reason there are <<unknown>> entries for the values is because VectorCAST does not determine the range for parameters if **`Range Checking`** is set to None. Change the setting to **All** or **Disable**, rebuild the range data, and generate the template again.

Maximum Number of Scalars Exceeded Before Creating Template

Another case in which the template information will not be generated, is if the total number of scalar data items in the test environment is greater than the setting for **`Maximum lines in EDG range file`** on the **Tools => Options** dialog, **Builder tab**. In this case the generated template file will contain the following text:

```
-- Test Case Script
--
-- Environment   : NAME
-- Unit Under Test: unit
TEST.UNIT:
```

```
TEST.SUBPROGRAM:
TEST.NEW
TEST.NAME:
TEST.NOTES:
TEST.END_NOTES:
TEST.END
--
```

The lack of TEST.VALUE and TEST.EXPECTED lines, or having only a few of them, means that the setting for the maximum number of lines in the range file was too low, so the range file could not be expanded. To overcome this situation, set the value of the option higher, rebuild the range data **(Environment => Rebuild Range Data)**, and then generate the template again.

Automating Test Script Maintenance

VectorCAST provides the VectorCAST Test Script Maintenance Utility which is automatically invoked when rebuilding an environment to maintain existing test scripts across source code changes. At the beginning of environment rebuild, the utility makes a copy of the original test script in the environment directory, naming it **`convertScript-original.tst.`** It then uses the changes in the source code to translate any modified subprogram names, parameters, and global objects to their new names, and writes a converted test script as **`convertScript-translated.tst.`** At the end of environment rebuild, this converted test script is imported.



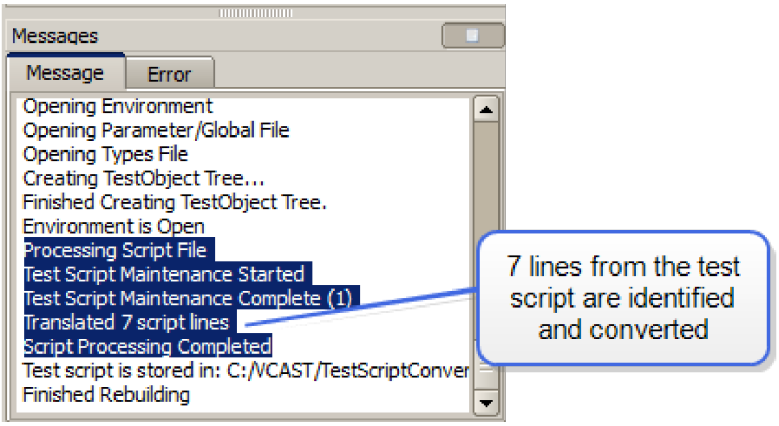
Note: If the option VCAST_DISABLE_TEST_MAINTENANCE is TRUE, then the VectorCAST Test Script Maintenance Utility is disabled.

Using the Test Script Maintenance Utility

The Test Script Maintenance Utility is invoked automatically during environment rebuild. The following example shows the Test Script Maintenance Utility doing an automatic conversion where only one field differs and there is no ambiguity concerning which conversion to choose.

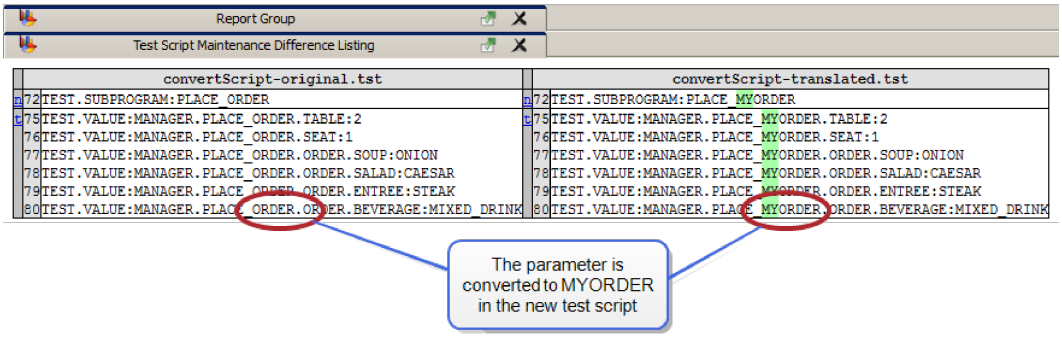
To automatically convert test scripts, you must first have an existing environment with test cases. Next, upload a new version of the source code and select **Environment => Rebuild Environment** from the Menu Bar.

As the environment rebuilds, the Message window shows the progress of the automated test script conversion.



Upon completion of the rebuild, both the Test Script Log and the Test Script Maintenance Difference Listing are displayed. The Test Script Log can be accessed at any time from the Menu Bar by selecting **Test => Scripting => Import Log**.

The Test Script Maintenance Difference Listing shows the changes made in the original script. Note in our example that the seven translated script lines are listed and that the parameter "ORDER" has been converted to "MYORDER". The Test Script Maintenance Difference Listing can be accessed at any time from the Menu Bar by selecting **Test => Scripting => Test Script Maintenance Difference Listing**. This GUI-only report is located in the environment directory and is named **`convertScript-differences.html`**.



The original test script file is saved in the environment directory as **`convertScript-original.tst`**. The new converted test script file is also saved in the environment directory as **`convertScript-translated.tst`**.

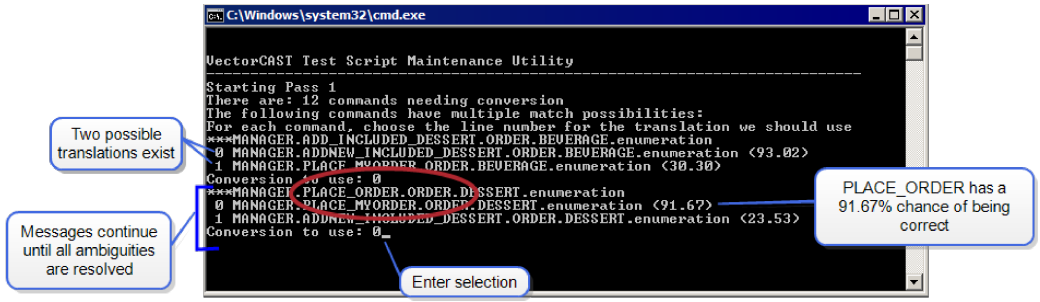
Using the Test Script Maintenance Utility With Multiple Solutions

If the comparison of the old and new files reveals multiple solutions to a conversion, the utility prompts the user to choose the translation. For example, if the comparison reveals the following lines:

```
OLD: MANAGER.ADD_INCLUDED_DESSERT.ORDER.BEVERAGE.enumeration

NEW: MANAGER.ADDNEW_INCLUDED_DESSERT.ORDER.BEVERAGE.enumeration

NEW: MANAGER.PLACE_MYORDER.ORDER.BEVERAGE.enumeration
```

The Test Script Maintenance Utility recognizes that there are two possible translations, **`MANAGER.ADDNEW_INCLUDED_DESSERT.ORDER.BEVERAGE.enumeration`** or **`MANAGER.PLACE_MYORDER.ORDER.BEVERAGE.enumeration`**, but does not know which one to use.

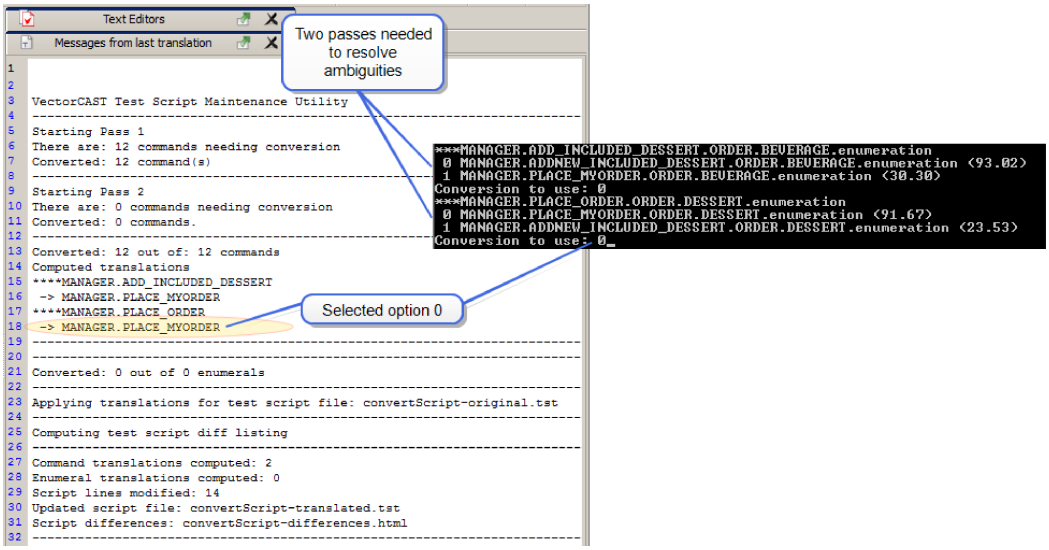
When using the Test Script Maintenance Utility in the GUI, a prompt is displayed allowing the user to resolve the ambiguity and select the translation. Each choice is displayed along with a score indicating the likelihood of the choice being correct. The list is sorted with the highest score first.



Note: The highest score is auto-selected when using the command line tools for regression testing.

The user enters the number of the desired choice, or, if no selection is made and the user hits **Return**, the highest score is automatically selected. The message window remains open and cycles through until all the ambiguities are resolved.

To see the full set of status messages after the message window closes, select **Test => View => Scripting => Messages from last translation** from the Menu Bar.



Importing and Converting Test Scripts on Rebuilt Environments With No Test Cases

The following CLICAST command can be invoked in instances where an environment having no test cases is rebuilt and the user wants to import and convert the test script:

```
clicast -e <env> test script convert <script>

This command imports the test script and converts it to accommodate source
code changes in the environment.
```

Delete Test Script Translation

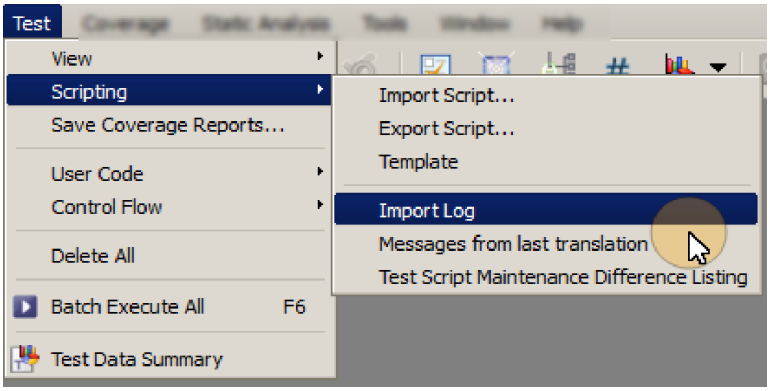
To delete a test script translation and revert to the original test script, perform the following steps:

1. Delete all of the test cases.
2. Select **Test => Scripting => Import Script** from the Menu Bar.
3. Navigate to the environment directory and select the file **`convertScript-original.tst`**.

Test Script Maintenance Reports

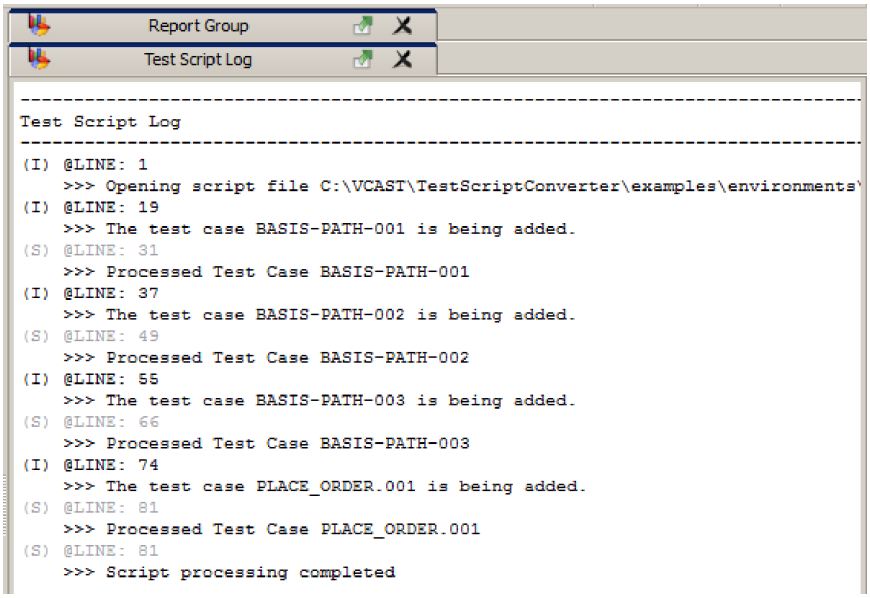
Data on the test script conversion is available from the Menu Bar by selecting **Test => Scripting** and selecting one of the report types:

- **Import Log**
- **Messages from last translation**
- **Test Script Maintenance Difference Listing**

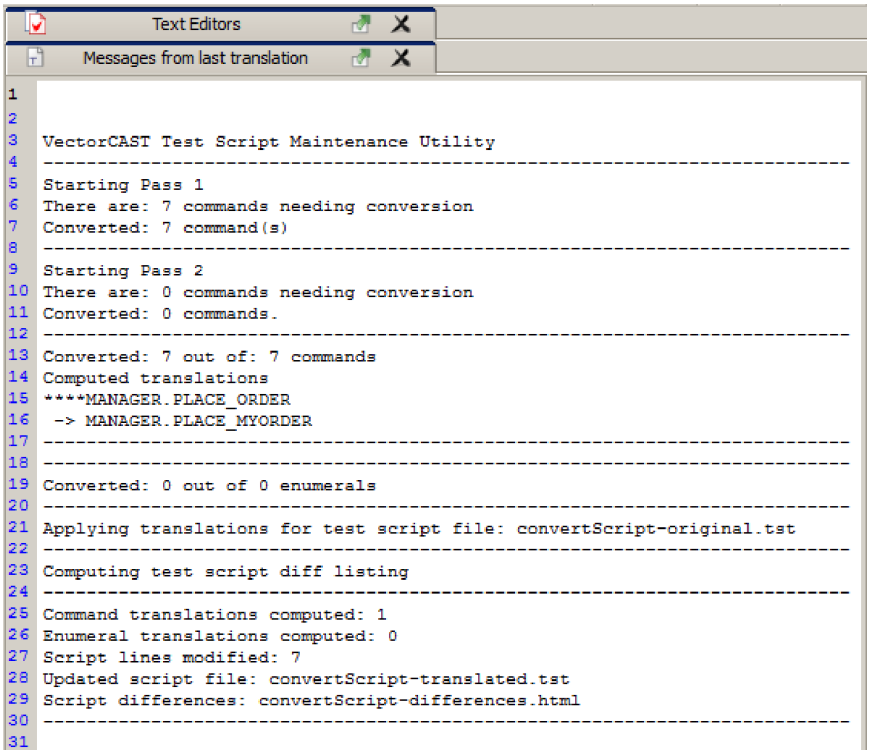


The reports are described below.

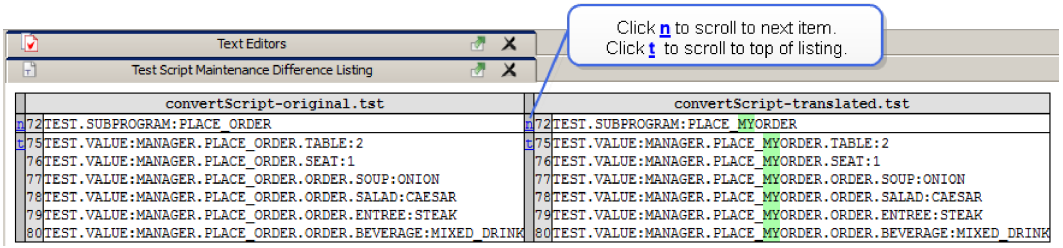
Selecting **Import Log** opens the Test Script Log. The Test Script Log reports on the processing of the test cases and any warnings or errors that occurred.



Selecting **Messages from last translation** opens the Messages From Last Translation report. This report provides status messages for the Test Script Maintenance Utility.



Selecting **Test Script Maintenance Difference Listing** opens the Test Script Maintenance Difference Listing, and shows the changes between the original and translated scripts.



Test Script Conversion Processing

Test script conversion requires that a backup environment directory (.BAK) exists in order to begin processing. The Test Script Maintenance Utility first reads the parse data in the **`param.xml`** and **`types.xml`** files in the .BAK version of the environment, and compares it to the newly-rebuilt environment before the test script has been re-imported. It maps the functions, parameters and global objects from the old version to the new, translating the test script in the process. At the end of the environment rebuild, the translated test script is imported. Output from the Test Script Maintenance Utility can be found at **Test => Scripting => Messages** from last translation, or in the file **`convertScript-log.txt`**.

Generating Test Cases Using CSV- or Tab-Delimited Files

VectorCAST supports importing test data from CSV or tab-delimited data files. Test cases are imported by first creating an association between the columns in the CSV file with scalar data item in the test case parameter tree. After creating this association, the CSV data file can be imported, and each row in the CSV file will become a separate test case in VectorCAST.

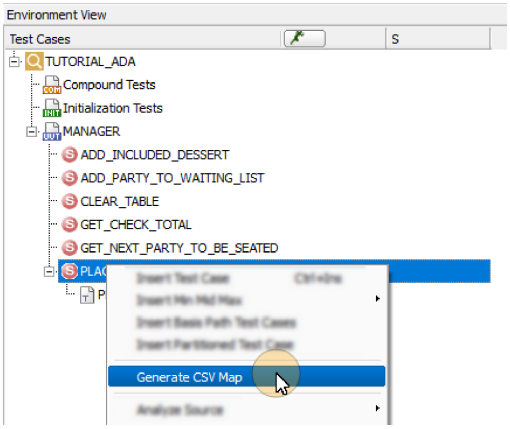
If you do not already have test data in this format, VectorCAST can be used to create the mapping, and then dump a "template". Once created, the template is filled with data, one row for each set of test case data using an external tool. The completed CSV file is then imported into VectorCAST to create the test cases.

The first step in either importing an existing file or in creating a template file is to create a *map test* case in VectorCAST. This special test case maps the columns of the file to any VectorCAST subprogram test case data item. Procedures for both creating a template file and for using a pre-existing file are discussed below. Note that once created, a map test case is imported and exported the same as any other test case.

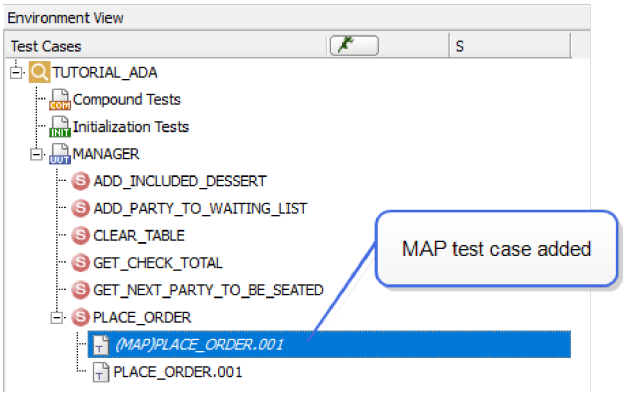
To Create a Template for a CSV- or Tab-Delimited File

VectorCAST can help you generate a CSV or tab-delimited template file that contains columns for test names and test case data items. After the template file is created, you use an external tool to enter test case data. To create a template file for a new **`.csv`** or **`.tab`** file, follow these steps:

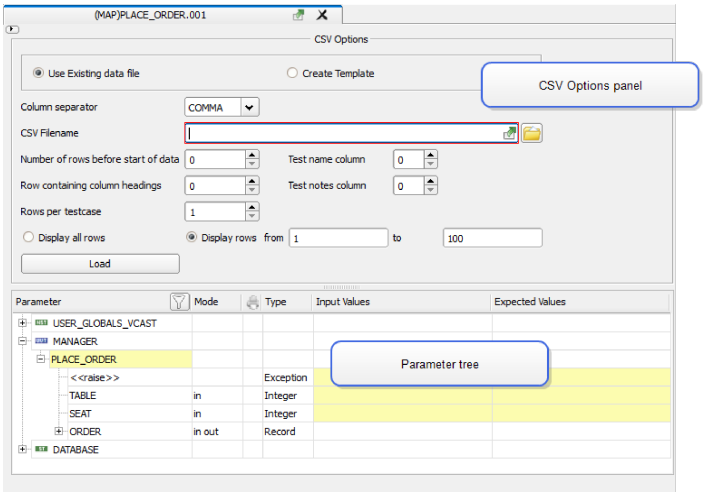
1. Right-click a subprogram in the Test Case tree and select **Generate CSV Map**.



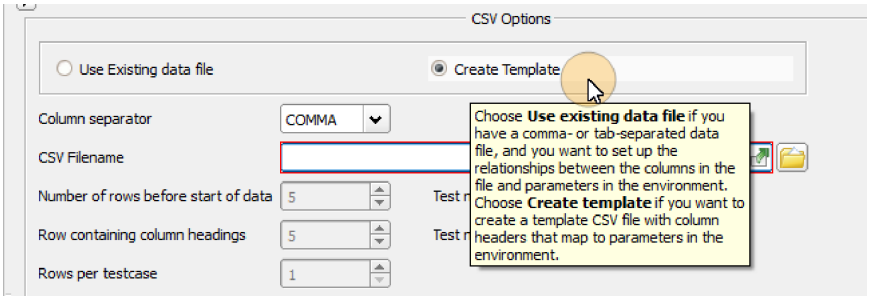
2. A (MAP) test case is placed in the Test Case tree.



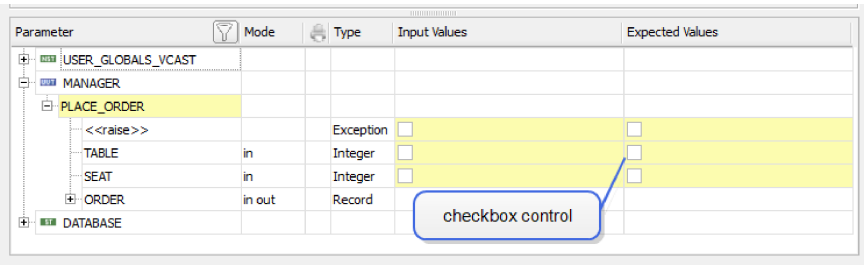
The CSV Options panel opens and below the CSV Options panel a Parameter tree opens. This tree looks similar to the Test Case Parameter tree. When creating a template file, the Parameter tree is used to select test case data items that will become columns in the template file. It is also used to set the order of the columns within the template.



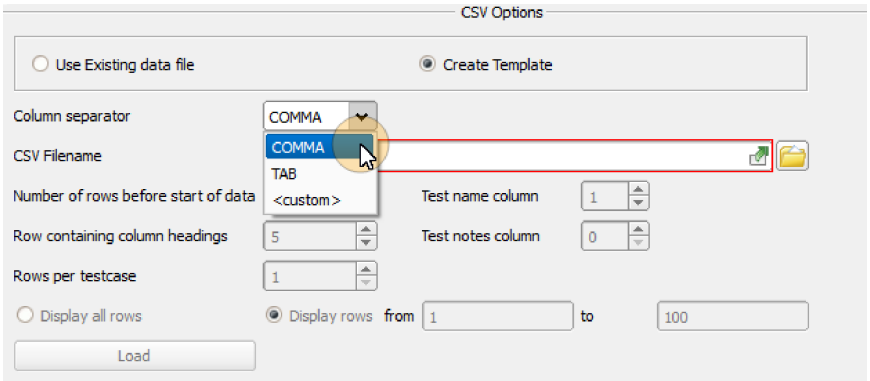
3. Select the **Create Template** radio button in the CSV Options panel.



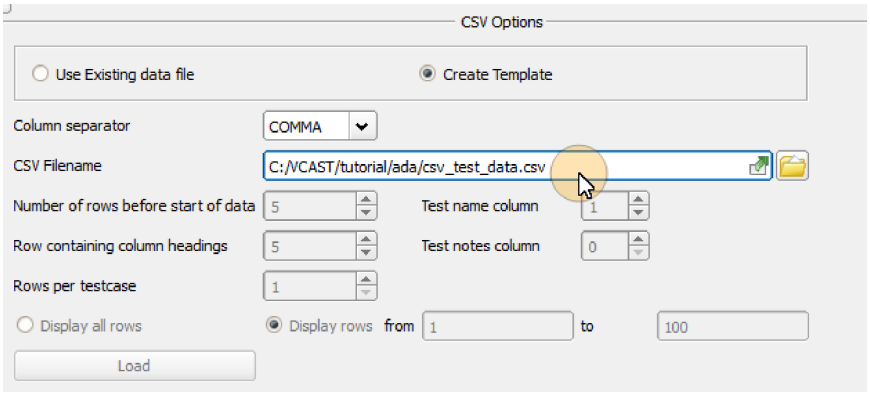
When this option is selected, each input and expected value data item in the Parameter Tree contains an associated *checkbox control*. The check boxes are used to select the data items that will become columns in the template file. Notice that check boxes appear not only for the parameters of the subprogram, but also for global data items and for SBF data items contained in the tree. In addition, if a data item is a pointer type, you can optionally use the Parameter Tree to create an instance of the pointer data item type. You can then select instance members of that type to be included as columns in the template file.



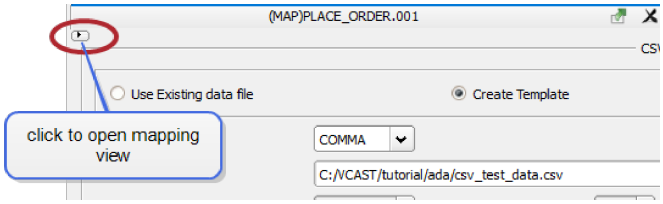
In the CSV Options panel, select a column separator by selecting either **COMMA**, **TAB** or **<custom>** from the drop-down menu for the delimiter you wish to use. Select **COMMA** if your data file or template uses the comma character to separate the field in a row. Select **TAB** if your data file or templates uses the tab character to separate the fields in a row. Select **<custom>** and enter any other character that your data file uses to separate fields in a row.



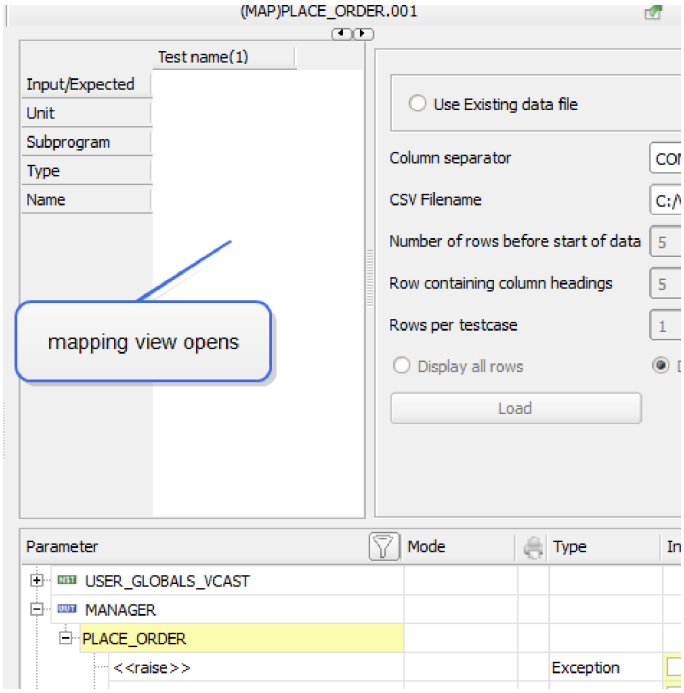
Enter a name for the template file. You may wish to enter an extension for the file name, i.e., **`.csv`**. By doing so, an external program, Microsoft Excel for example, will recognize the file type and open it appropriately so that its contents can easily be edited. You can optionally navigate to a specific directory by using the file browse button.



4. Next, click the arrow button, found in the upper left of the MDI window.



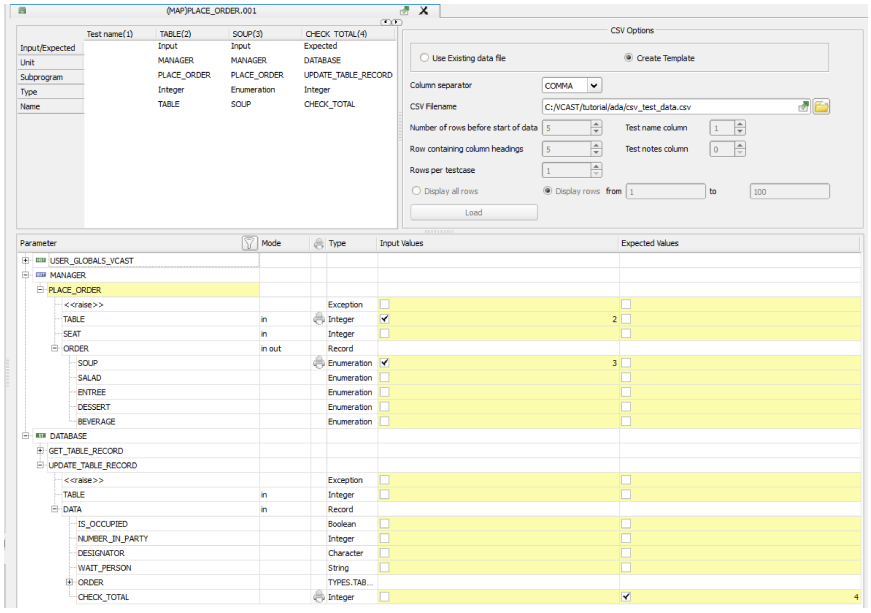
A blank Mapping View appears above the Parameter Tree.



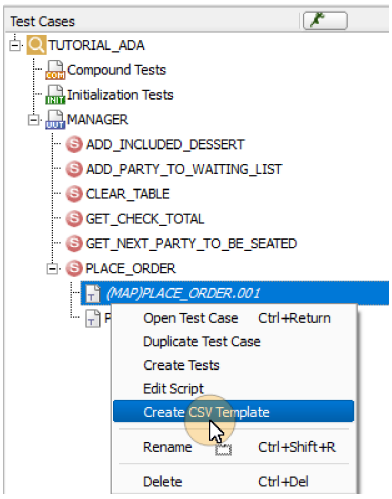
5. Next, select the test case data items to be included in the template file by clicking the check boxes in the Parameter tree. As each data item is selected, a header entry is added to the Mapping View. The header information consists of five rows of information for each data item and will be included in the template file when it is generated. For each column, the template header information consists of:

- **Input** or **Expected**
- **Unit** name
- **Subprogram** name
- data item **Type** (int, float, enum, etc.)
- data item's **Name**

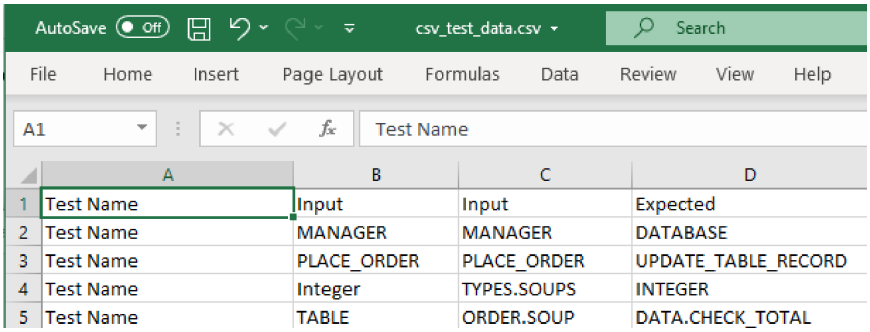
6. As input or expected values are selected for inclusion, a number appears in the associated Parameter tree cell.
7. The same number also appears next to the column header name in the Mapping View. These numbers indicate the column number that the test data item is mapped to. The column order can be easily changed using either of the following methods:
   - The first method is to change the value in the Test Tree Editor. Click the desired cell in the editor and change the column number to the desired number and press **Enter**. The new column order will be immediately reflected in both the Mapping View and the Parameter Tree. All columns are renumbered appropriately based on the change.
   - The second method is to drag and drop columns within the Mapping View column title row. Holding the **CTRL** key, select and drag a column title in the column title row to the desired column position. The column numbers are automatically updated in both the Mapping View and the Parameter Tree using this method as well.



8. When you are finished selecting columns and ordering them, save the map by clicking the **Save** button  on the Toolbar, or selecting **File => Save**. Note that this saves the mapping created above, but a template file has not yet been generated.
9. To create the template file from the map, right-click on the (MAP) test case in the test case tree and choose **Create CSV Template**.

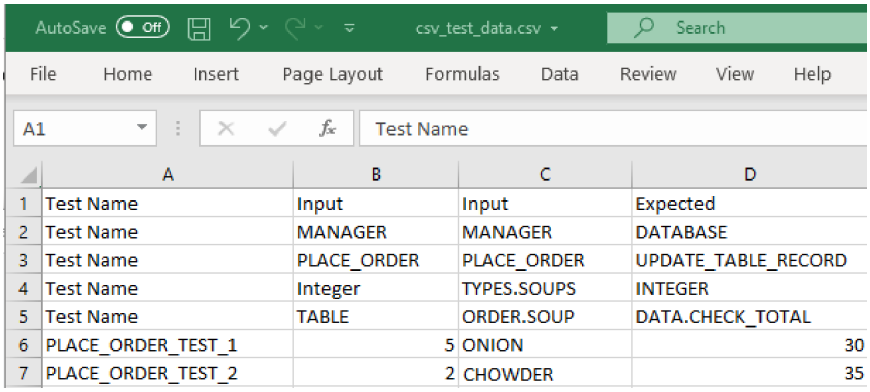


This generates the template file using the name specified in the CSV Options menu and will place the file in the directory you specified. If no path was given, the file is placed in the working directory. The file contains the five rows of header information that were shown in the Mapping View when the map was created.



	A	B	C	D
1	Test Name	Input	Input	Expected
2	Test Name	MANAGER	MANAGER	DATABASE
3	Test Name	PLACE_ORDER	PLACE_ORDER	UPDATE_TABLE_RECORD
4	Test Name	Integer	TYPES.SOUPS	INTEGER
5	Test Name	TABLE	ORDER.SOUP	DATA.CHECK_TOTAL

10. Open the generated file using an external tool. The first column of the file represents the test name, and the following columns represent the test data items that were selected when the map was created. Below, two test cases have been added to the template file. The first test case is named PLACE_ORDER_TEST_1, and the second is PLACE_ORDER_TEST_2. Input parameters for Table and Soup have been entered and an expected return value has also been provided. When you are done adding data, save the file, and the populated test file is now ready to be imported.



	A	B	C	D
1	Test Name	Input	Input	Expected
2	Test Name	MANAGER	MANAGER	DATABASE
3	Test Name	PLACE_ORDER	PLACE_ORDER	UPDATE_TABLE_RECORD
4	Test Name	Integer	TYPES.SOUPS	INTEGER
5	Test Name	TABLE	ORDER.SOUP	DATA.CHECK_TOTAL
6	PLACE_ORDER_TEST_1	5	ONION	30
7	PLACE_ORDER_TEST_2	2	CHOWDER	35

```
clicast -e <env> -u <unit> -s <sub> -t <MAP> TESt CSv2tst TEmplate
[<outputfile>]

Generate a CSV or (tab-separated) template file using template
information identified by the map test case -t <MAP>. The output
filename is specified in the MAP test case; if the file exists it will
not be overwritten.
```

To Create a Map from an Existing CSV or Tab-Delimited File

To demonstrate this feature, an example CSV file, “table.csv”, located in the Examples/c/csv_ example directory in the VectorCAST installation directory will be used. The first few lines of the file contain the following entries:

	A	B	C	D
1	Y	Slope	X	Y-Intercept
2	15	-1	-10	5
3	12.6	-0.9	-9	4.5
4	10.4	-0.8	-8	4
5	8.4	-0.7	-7	3.5
6	6.6	-0.6	-6	3
7	5	-0.5	-5	2.5
8	3.6	-0.4	-4	2
9	2.4	-0.3	-3	1.5
10	1.4	-0.2	-2	1
11	0.6	-0.1	-1	0.5

The first row of the file is a header row containing the names of the input and expected values for each test. Each of the following rows represents the test data for each test case to be created. The function *findY* will be tested using the values in “table.csv”. The source code for findY is as follows:

```
float findY ( float x, float m, float b ) {
 return m * x + b;
}
```

> Column A, labeled “Y,” is the expected value returned from findY.

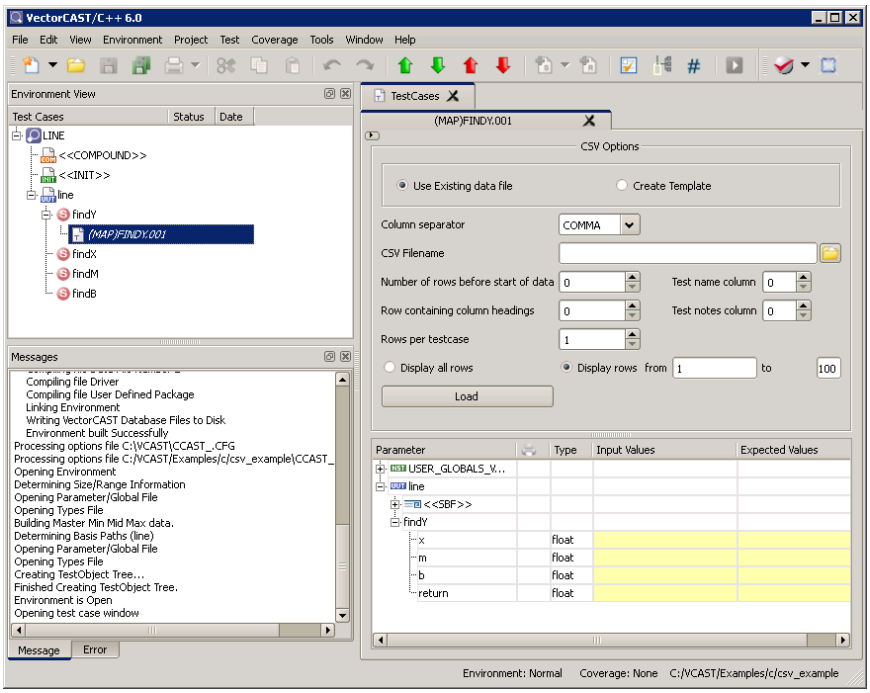
> Column B, labeled “Slope,” is the input parameter “m”.

> Column C, labeled “X,” is the input parameter “x”.

> Column D, labeled “Y-Intercept,” is the input parameter “b”).

To create the mapping, follow these steps:

1. Create a test environment for line.c that is located in the Examples/c/csv_example directory in the VectorCAST installation directory. Right-click on the findY subprogram in the test tree and select Generate CSV Map. A (MAP) test case is created for findY and a CSV options panel appears.



2. In the CSV Options panel select the Use Existing data file radio button

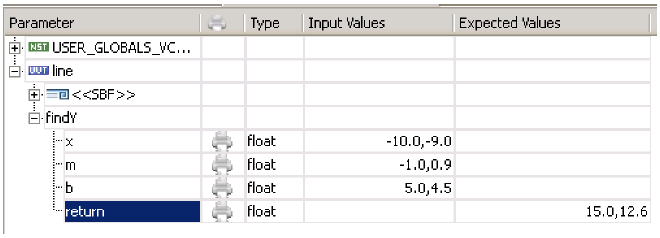


3. Select the type of delimiter used to separate values. The example uses a CSV file, so select Comma-separated values.



4. Select the file "CSV Filename" by browsing for the file "table.csv"
5. The controls below "CSV Filename" in the CSV Options panel provide the facility to load files that are custom tailored.

- **Number of Rows to skip** - enter the number of rows of header information at the top of the file. "Table.csv" has 1 header row, so enter 1. If there were no headings, this value would be set to zero. For template files created with VectorCAST, this value is 5.
- **Row containing column headings** - is the row that contains the data item name. This information is used by the Mapping View as the heading for each column. In the example this is row 1. If there were no heading rows, this value would be set to zero. When using a template file created with VectorCAST, this value is 5.
- **Rows per testcase** - for this example set the value to 1. Typically, each row represents a single testcase, but if you wish to combine multiple rows into one test case (the test parameters are combined as a list), set this value to the number of rows to be combined into one test case. If for example, we selected a value of 2, each parameter would have a list containing two values and two rows would be used per test case. This feature is only used with very large data files.



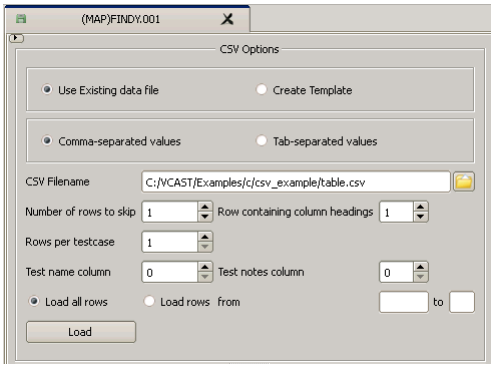
- **Test name column** – in the example, there is no column with test names, so the value is set to 0. The test names for this example will automatically be generated by VectorCAST. When using a template file created by VectorCAST, the name column is 1.

- **Test notes column** – in the example, there is no column containing test notes, so the value is set to 0. When test notes are available, they will be added to the “Notes” tab of the test. Place the column number containing the notes in this field. When using a template file created by VectorCAST, the name column is 0.

- **Load all rows** - selecting this option will load all of the rows from the file into the Mapping View.

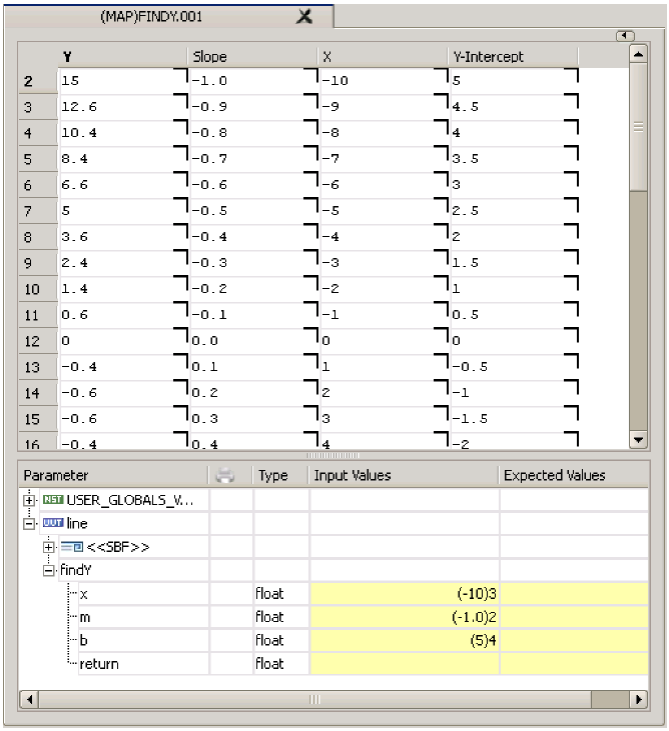
- **Load rows from/to** – this allows a subset of rows to be loaded into the Mapping View, specifying a starting and ending row. If you have a very large CSV or tab-delimited file and only want to a see a small subset in the Mapping View after loading, select this option and specify the

range of rows to load.



6. Click **Load** to open the Mapping View using the current settings. After clicking **Load**, the CSV Mapping View appears. It contains the rows selected for loading and is used to map the file columns to test data items of the subprogram.
7. To map a column in the Mapping View to a test case data item, drag any value in a column in the Mapping View and then drop it to a cell in the Parameter Tree.

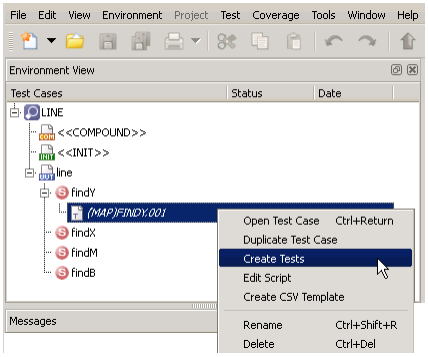
   When you select a cell in the mapping view, all cells with a matching data type in the Parameter Tree are highlighted in yellow. This helps you select an appropriate cell for mapping. You can drag a value from any row. The number in the parentheses in the Parameter Tree is the value contained in the cell selected in the Mapping View that was dragged into the Parameter Tree. The number to the right of that is the column number in the Mapping View that is mapped to the test data item.
8. When you have completed mapping all of the columns to input or expected values, click the Save icon in the toolbar or choose **File => Save**.



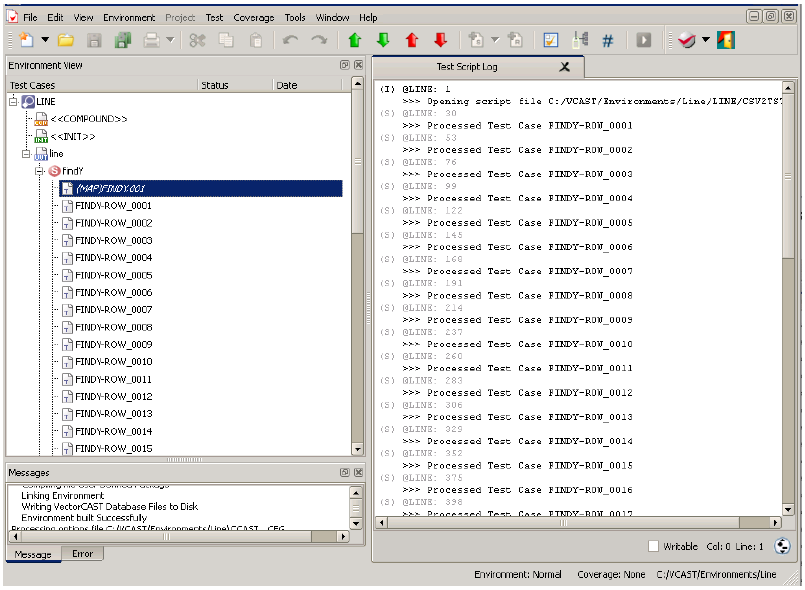
9. Mapping is now complete, and the map may be used to create tests.

To Create Test Cases from a MAP Test Case

Right-click on a (MAP) test case in the Test Case tree and select **Create Tests**. A dialog box indicating the number of tests to be generated is presented. Click **OK** to proceed.



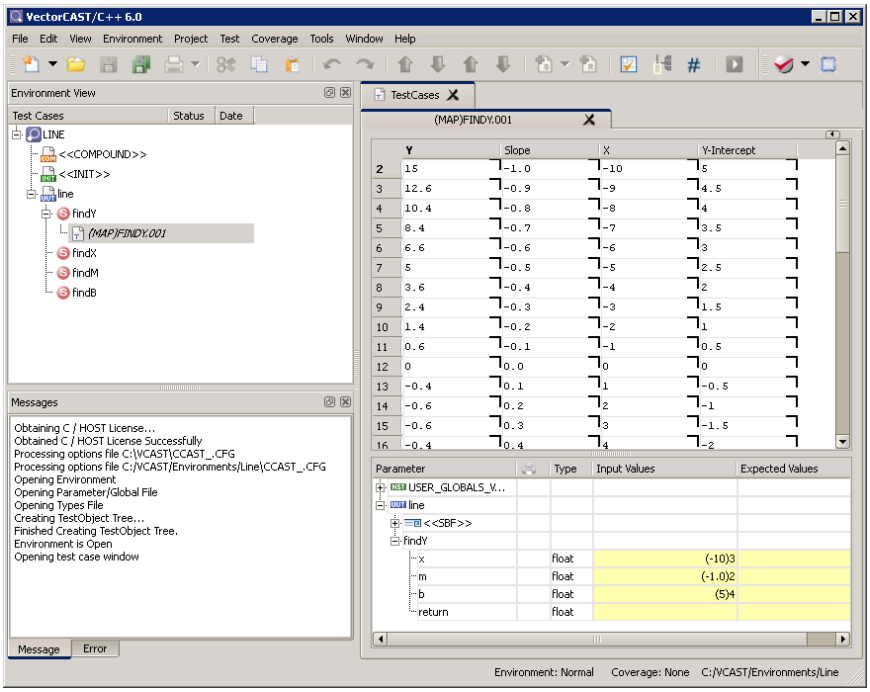
All tests are created and appear in the Test Case tree, and a test script log is displayed. You may now select and execute the tests.



```
clicast -lc -e <env> [-u <unit> [-s <sub> [-t <test>]]] TESt CSv2tst Load

Generate a test case containing a test case for each row in the CSV file
identified by map test case(s) in the environment.
```

To Edit an Existing Map Test Case

An existing map test case may be edited by double clicking the map entry in the Test Case Tree. A Mapping View and Parameter Tree will appear.



The cells that are highlighted in yellow in the Parameter Tree are available for editing. The value in parenthesis in the Parameter Tree cell is the column value for the data item in the map. For example in the picture above, “x” in the Parameter Tree contains (10). This is the same value for “X” in the first row of the Mapping View. The number in the cell outside of the parenthesis is the column number of the Mapping View that the data item is mapped to. In this example, “X” is column 3 in the Mapping View, and cell in the Parameter Tree also contain a 3. This allows you to easily verify the current mapping and to make changes if necessary.

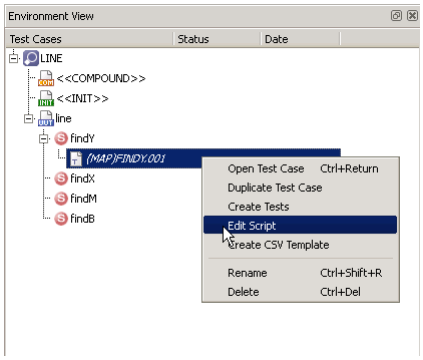
To remove a data item from the map, simply click in the Parameter Tree cell and delete the column number entry and press **Enter**.

To map a column that is currently not mapped to the Parameter Tree or to change a currently mapped column, click a value in the column and drag and drop it into the desired cell of the Parameter Tree. You may also select a Parameter Tree cell and type a column number directly and pressing **Enter**. You will see the value and its column position immediately updated in the Parameter Tree upon releasing the mouse button or pressing Enter.

When you are finished editing the map file, click the **Save** button on the Toolbar  or select **File => Save**.

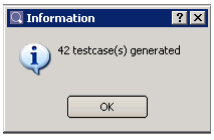
To Create and Edit a Test Script from a Map File

Right-click on a map file in the Test Case tree, and select **Edit Script**

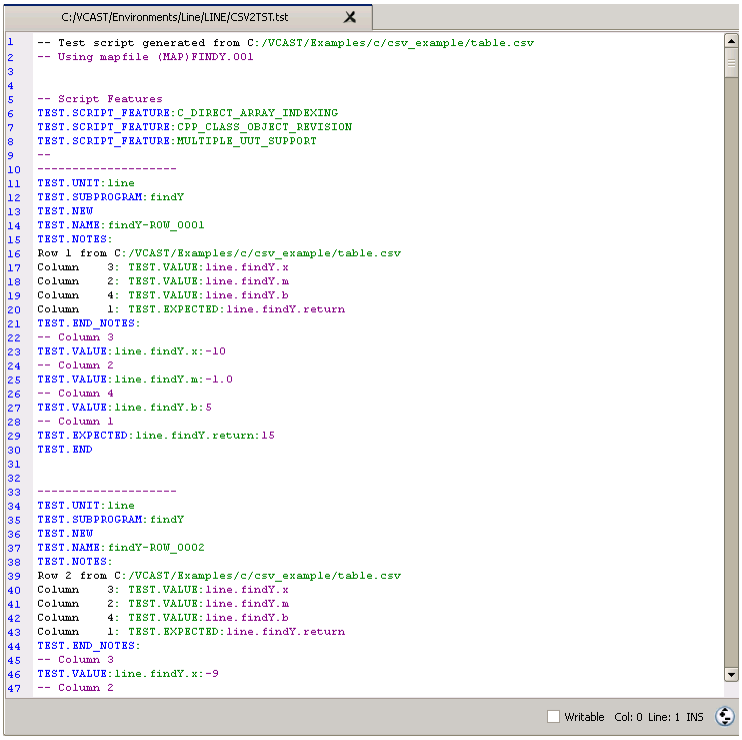


.

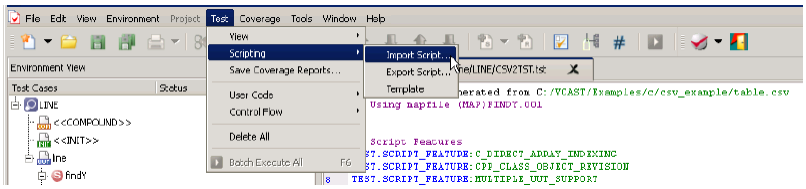
A dialog box appears indicating the number of tests generated from the associated CSV or tab-delimited file. Click **OK**.



A test script, containing all of the generated tests appears in a test script editor.



Review the tests and make any changes you wish to the script file. The test script can be imported by selecting **Test => Scripting => Import Script**




```
clicast -lc -e <env> [-u <unit> [s- <sub> [-t <test>]]] TESt CSv2tst CReate
<test script>

Generate a test script containing a test case for each row in the CSV file
identified by map test case(s) in the environment. Output file <test
script> must be specified.
```

VectorCAST Tools

To Set VectorCAST Options

The **Tools => Options** dialog is used to access the many options and settings available to configure VectorCAST. To access the dialog, choose **Tools => Options...** or click the **Options** button  on the toolbar. The options are grouped by category on the following tabs:

- GUI
- Ada
- Wizard
- Builder
- Execute
- Report
- Target
- Coverage

When you click the **OK** or **Apply** button, VectorCAST writes the options to the ADACAST_.CFG file, located in the current working directory. This directory is noted in the status bar of the main window. Any changes made to the configuration file in the working directory affect other environments in the directory.

If you run VectorCAST in command line mode (clicast), the same file is accessed for Options settings.

When the VectorCAST application is launched, it looks for a configuration file in the following locations:

- VectorCAST installation directory
- User's home directory
- Current working directory

Options are read in the order specified with each file taking precedence over previous files.

If you want some options to apply to all environments (e.g., the compiler you are using), you can set these options in the VectorCAST installation directory.

If you have VectorCAST installed on a network and each user has a client installation, you can create a configuration file that applies to all users of VectorCAST. To do this, set the working directory to network drive, and set any options you want all users to have, such as compiler settings. This configuration file is used unless settings from an environment-specific configuration file overrides them.

If you add a line or make a change to your configuration file external to VectorCAST (while it is running), you can read those values back into VectorCAST. To do this, click the **Load** button. The changes that you made to the file are now reflected in the Options dialog box.

The default values for the current active tab can be reset by clicking the **Default** button.

Clicking **Cancel** closes the dialog box, without saving any changes since the last time you clicked the Apply button.

See "To Set the Working Directory" on page 99 for information on changing the current working directory.

Tools Menu

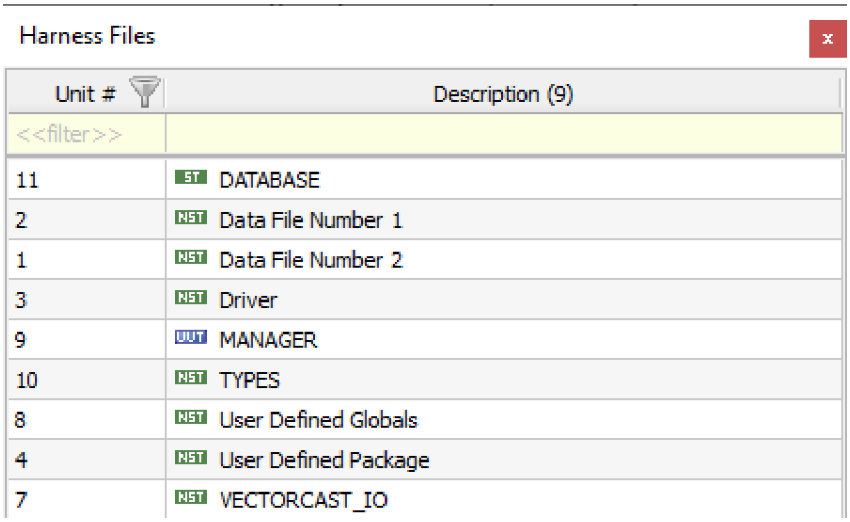
The following sections cover the Tools Menu selections. The selections on the menu that are discussed are:

- View Harness Source
- Basis Path
- MC/DC
- Industry Mode (see "To Set the Industry Mode for Coverage" on page 27)
- Custom Tools (see "Working With Custom Tools" on page 245)
- Requirements Gateway (see "Using the Requirements Gateway" on page 425)
- Options (see "To Set VectorCAST Options" on page 242)

View Harness Source

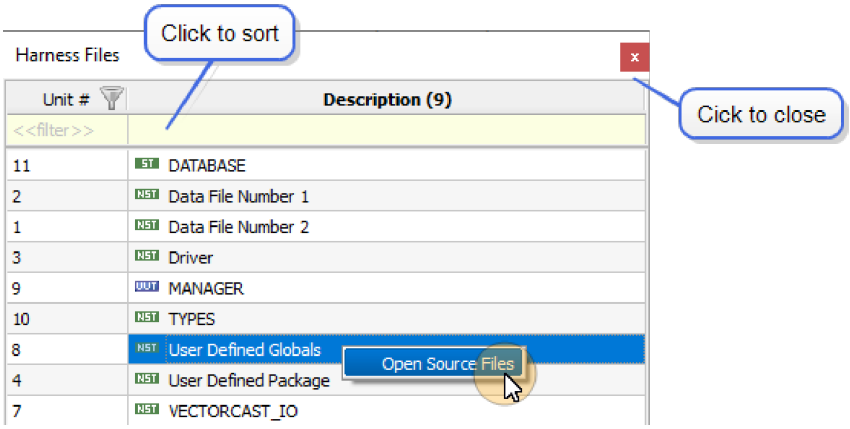
The View Harness Source command enables you to view the test harness source code for all of the units in a test environment. To view the harness files, select **Tools => View Harness Source**.

The pop-up window contains the UUT and any dependent units, along with all the harness source files. Selecting one of them opens up the source code file. This feature provides a convenient way to browse the source code as tests are being developed. The figure below shows the pop-up window for an environment having **`MANAGER`** as the UUT and having **`DATABASE`** as a dependent unit.



To aid in cross-referencing the source filenames with the VectorCAST unit numbers, the VectorCAST unit numbers are displayed in the Harness Files window.

To view one of the files, select the name, right-click, and click **Open Source File**. Clicking on the header bars at the top sorts the files in reverse order by Unit #, alphabetically by description. Clicking again reverses the order. To dismiss the window, click on the button located in the upper right corner.



Once you have chosen a harness source file to view, it appears in the MDI window. You can view the source file or even edit it here. If you change the source code file, and want the changes incorporated into the test harness, choose **Environment => Recompile**. Keep in mind that any changes you make to the test harness source files are not reflected in your original source code files, as these files are separate and distinct from the test harness.

See also "To View the Source for a Subprogram" on page 161.

Basis Path

This menu item has four options:

- Perform Analysis
- View Analysis Report
- Generate Tests
- Generate Test Scripts

To Perform a Basis Path Analysis

The **Tools => Basis Path => Perform Analysis** command enables you to generate a Basis Path Analysis. The contents of this analysis reflect the items selected in the Test Case Tree.

To View a Basis Path Report

The **Tools => Basis Path => View Analysis Report** command enables you to view the results of the computation of test paths for test case building. The contents of this report reflect the items selected in the Test Case Tree.

The number of paths identified equals the code complexity, also referred to as V(g). The code complexity corresponds to the number of test cases that must be developed to exercise the unique paths in a subprogram.

```
clicast -e <env> [-u <unit>] TOols Basis_path [<outputfile>]

Create a Basis Path report and output to HTML file <env>_basis_path_
```

```
report.html, standard output as text, or to the specified output file. If
the Unit parameter is specified, the Basis Path report includes only the
specified unit; otherwise, the Basis Path report includes all units in the
environment.
```

See also "To Insert Basis Path Test Cases" on page 164 for information on having VectorCAST automatically generate a test case for each basis path.

To Generate Basis Path Tests

The **Tools => Basis Path => Generate Tests** command enables you to generate the Basis Path Tests for the selected units in the environment.

To Generate a Basis Path Test Script

When you choose **Tools => Basis Path => Generate Tests**, VectorCAST creates a test script containing all of the automatically generated tests, and then loads this script into the test environment.

To (Re)Generate the Basis Path Tests

After changing your source code, recompiling your units, and relinking the test harness, you may want to regenerate the basis path report. Choose **Tools => Basis Path => Generate Tests** to force VectorCAST to fetch the latest version of the UUTs, and regenerate the basis path report. If you have coverage on, this command will reset the coverage data. Use this command to compute the basis paths if the environment was built without basis paths (the option "**`Do not automatically generate basis paths`**" was set).

```
clicast -e <env> [-u <unit>] TOols Update_basis_path

Update the statement basis paths for the environment.
```

MC/DC

This menu has one option: View Equivalence Matrices.

To View Equivalence Matrices

Selecting **Tools => MC/DC => View Equivalence Matrices** will display the MC/DC Condition Tables Report for the selection in the environment.

Working With Custom Tools

You can add external utilities or applications to the VectorCAST Tools menu and toolbar. The associated tool is added to the Custom Tools dialog as well.

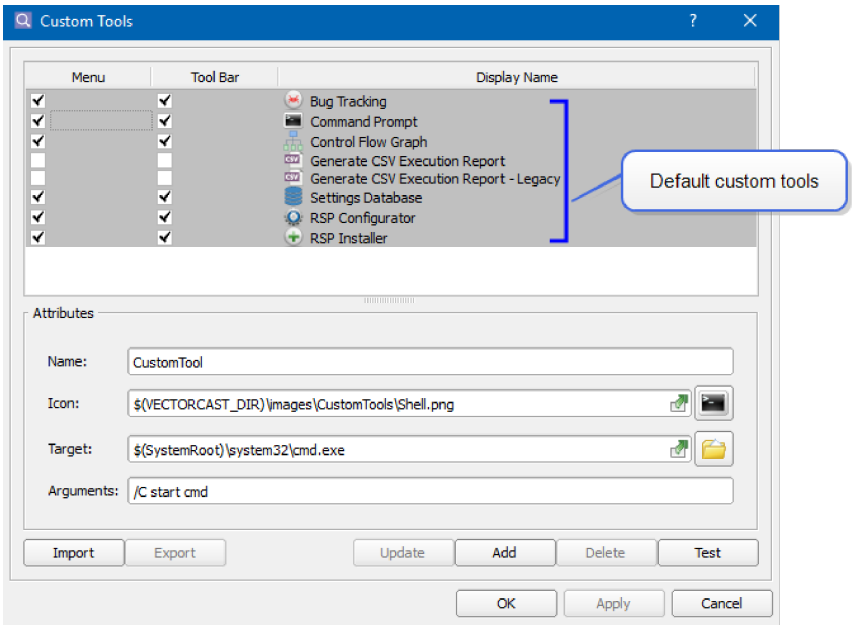
With the Custom Tools dialog, you can:

- Add a new tool.
- Delete a tool.
- Choose an icon for the tool.
- Specify that the icon appear in the VectorCAST Tool Bar and Custom Tools Menu.
- Update the settings for a tool.
- Test a tool.
- Import and Export custom tool configurations.

Default Custom Tools

VectorCAST provides default custom tools for users' convenience. The Custom Toolbar contains the default custom tools and is displayed in the main Toolbar by default. Custom tools are also displayed in the Custom Tools menu. The following custom tools are provided:

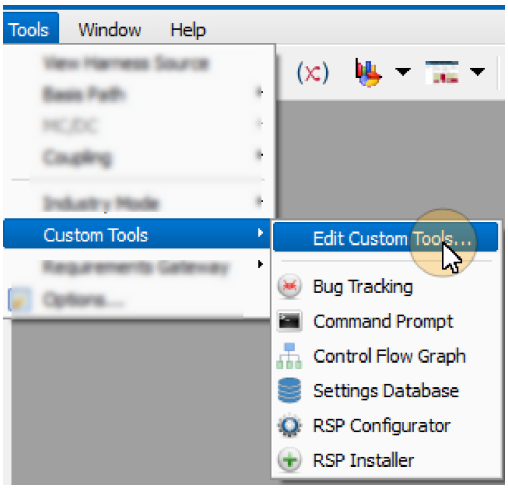
- Bug Tracking - Allows users to create bugs via an integrated bug tracking tool.
- Command Prompt - On Windows, a DOS command prompt is available. On Linux, an XTerm window is available.
- Generate CSV Execution Report - Dumps the Execution Report for the currently selected test case in CSV format, and on Windows, opens the resultant file in Microsoft Excel. By default, this custom tool is not shown in the Toolbar or the Custom Tools menu.
- Settings Database - Provides a visual editor for use with a vcshell database file.
- RSP Configurator - Allows users to select the compiler family and modify and save associated arguments.
- RSP Installer - Allows users to download and install customer-specific VectorCAST/RSPs.



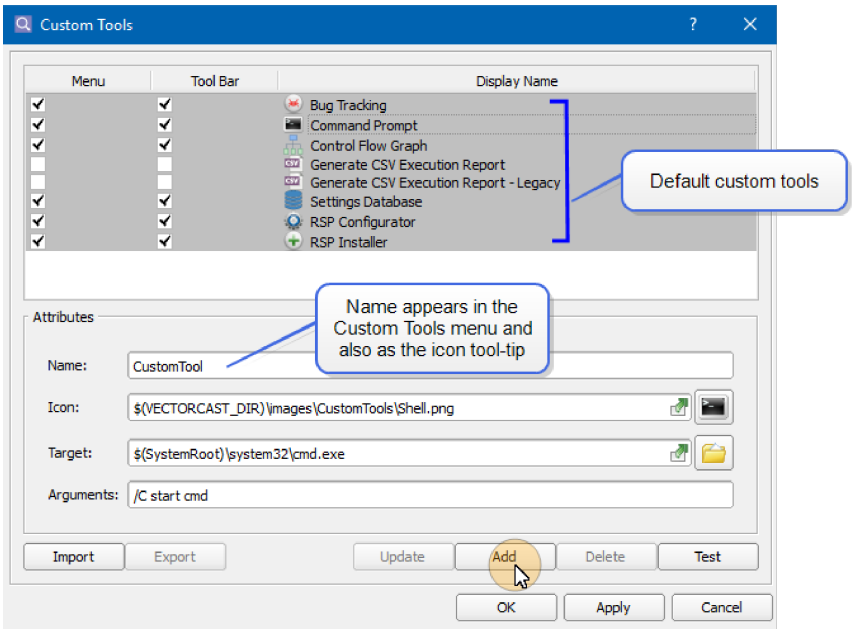
Default custom tools are displayed with a gray background color, and are not editable by the user.

Add a Custom Tool

Add a custom tool by selecting **Tools => Custom Tools => Edit Custom Tools...** from the Main Menu and open the Custom Tools dialog.



The Custom Tools dialog appears:



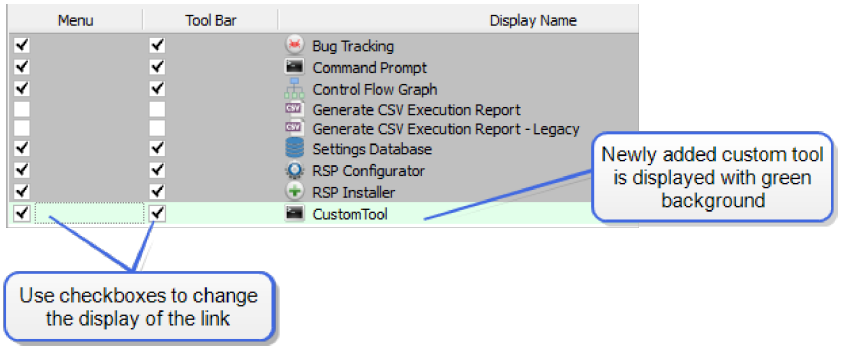
To add a tool, enter the label you want to appear in the Custom Tools menu into the **Name** field. The name is also used as the tool-tip for the icon.

Optionally, enter or browse for the path to the icon image file in the **Icon** field.

Enter or browse for the path to the relevant application file. The path to the executable file, shell script, or batch file is entered into the **Target** field.

Enter any arguments needed for the target in the **Arguments** field.

Select the **Add** button.



The new custom tool is added to the list of Custom Tools. The background for the tool is temporarily set to green, to aid the user in identifying and tracking newly added items. Clicking the **Apply** button adds the tool to the list and the background turns to white.

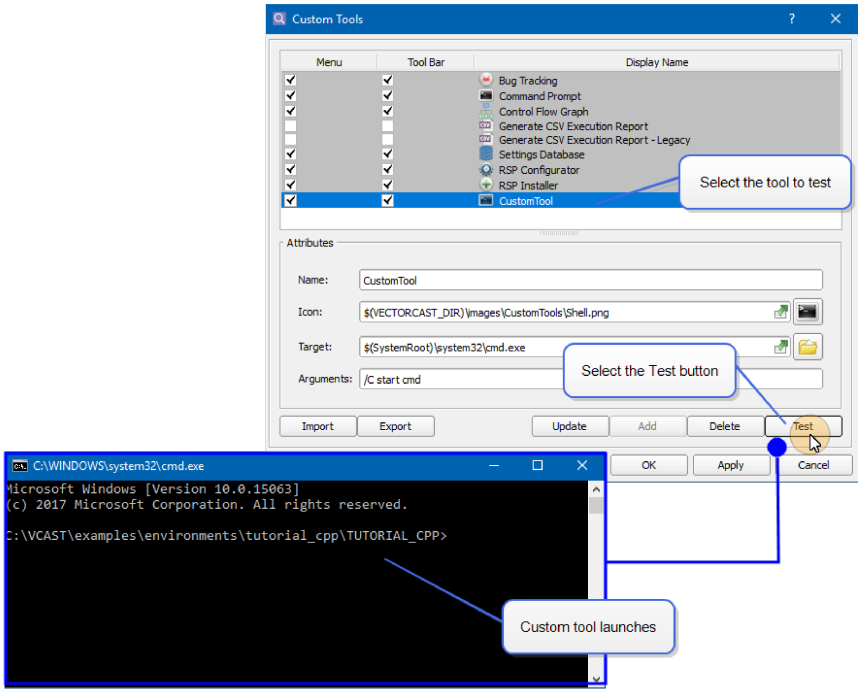
Use the associated check boxes to change the display of the link. By default, the tool's link is displayed in the both the Custom Toolbar and the Custom Tools menu.

Clicking **OK** closes the dialog.

Test the Tool

Test a custom tool by selecting **Tools => Custom Tools => Edit Custom Tools...** from the Main Menu and open the Custom Tools dialog.

To test the tool to make sure it works, press the **Test** button. The tool opens in a separate window.

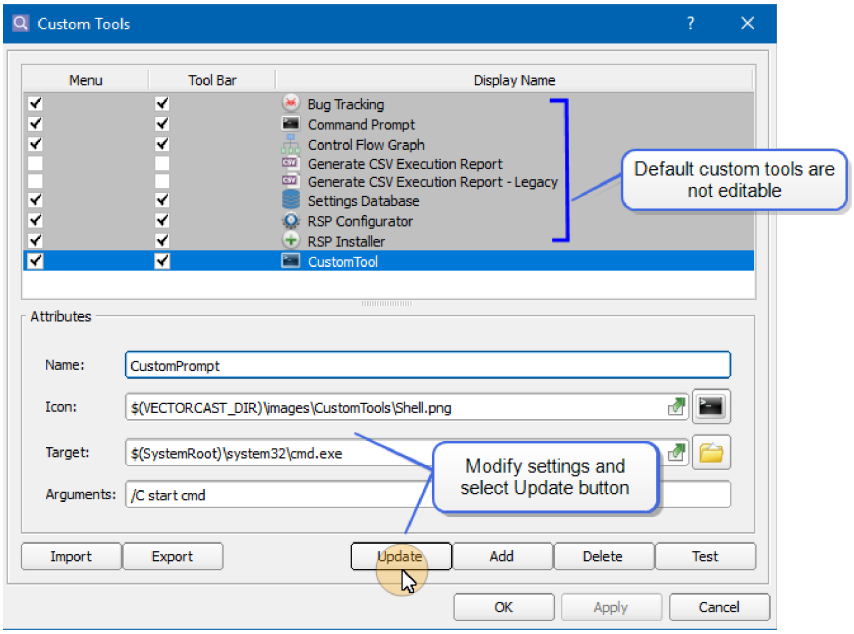


Update the Settings for a Tool

Update the settings for a custom tool by selecting **Tools => Custom Tools => Edit Custom Tools...** from the Main Menu and open the Custom Tools dialog.

Highlight the tool in the Display Name column. The settings for the tool are displayed in the Attributes pane.

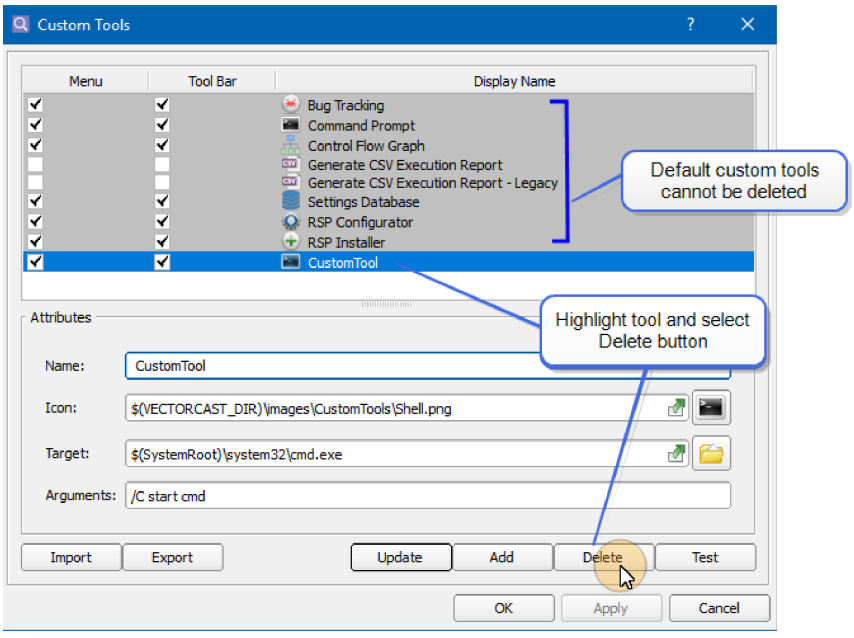
To update the settings for a tool, in the Attributes pane, make changes in the Name, Icon, Target, or Arguments fields. Click the **Update** button. Click **OK** to close the dialog.



Note: Default custom tools are not editable.

Remove a Custom Tool

Remove a custom tool by selecting **Tools => Custom Tools => Edit Custom Tools...** from the Main Menu and open the Custom Tools dialog.

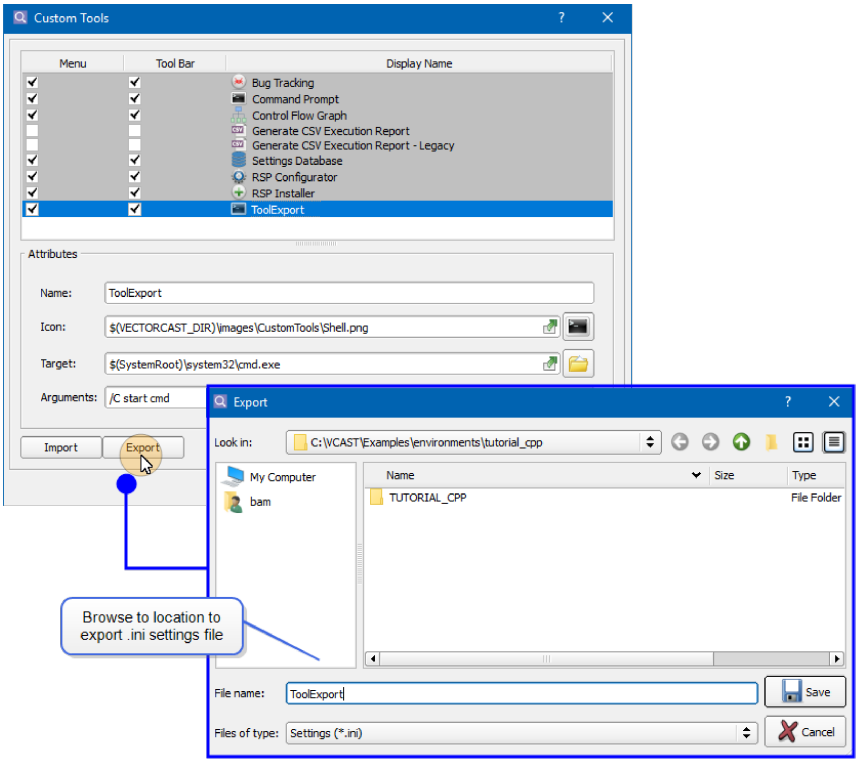


Select the tool to delete and then click the **Delete** button. Click **OK** to close the dialog.

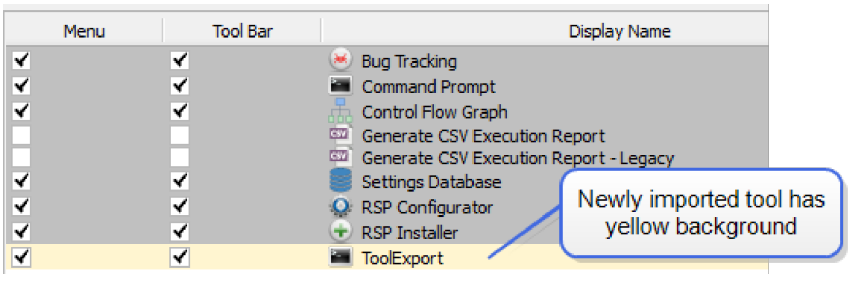
Import/Export Custom Tools

You can share Custom Tools with other VectorCAST users using the Import / Export functionality of the Custom Tools dialog.

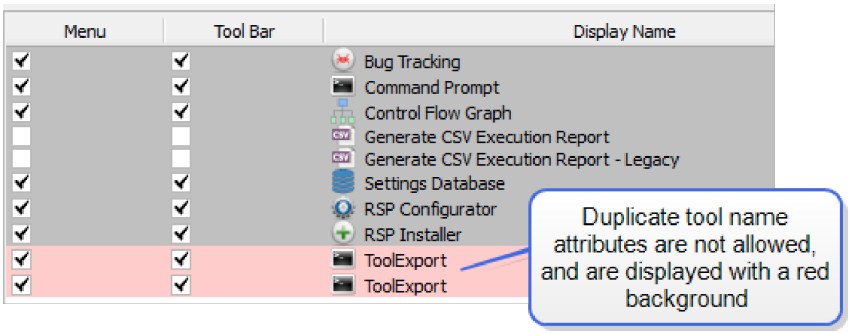
To export an existing custom tool, highlight the tool and select the **Export** button. Note that default custom tools cannot be exported. Browse to the location where you will store the `.ini` settings file and enter the file name. In our example, the file is named **`ToolExport.ini`**.



To import a custom tool, open the Custom Tools dialog and select the **Import** button. Browse to the location of the `.ini` settings file and select **Open**. The tool is added to the Custom Tools list, and the background for the tool is temporarily set to yellow, to aid the user in identifying and tracking newly imported items.

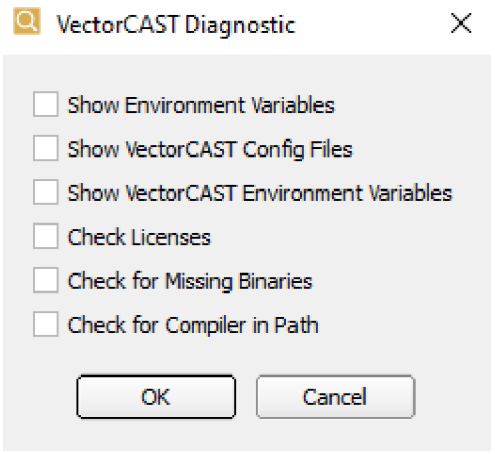


If VectorCAST detects a duplicate name attribute in the list, the background for the tool is set to red to alert the user of the conflict.



To Create a Diagnostic Report

The diagnostic report is useful to diagnose tool configuration problems. To run diagnostic checks, choose **Help => Diagnostics => Create Diagnostic Report**. A dialog box appears, allowing you to check which items to want to see in the report. When you click OK, the report opens in the Report Viewer. You can save the information to a file using the **File => Save As** command.




```
clicast -lada REports DIagnostic [<outputfile>]

Generates diagnostic report to standard output or the specified output
file.
```

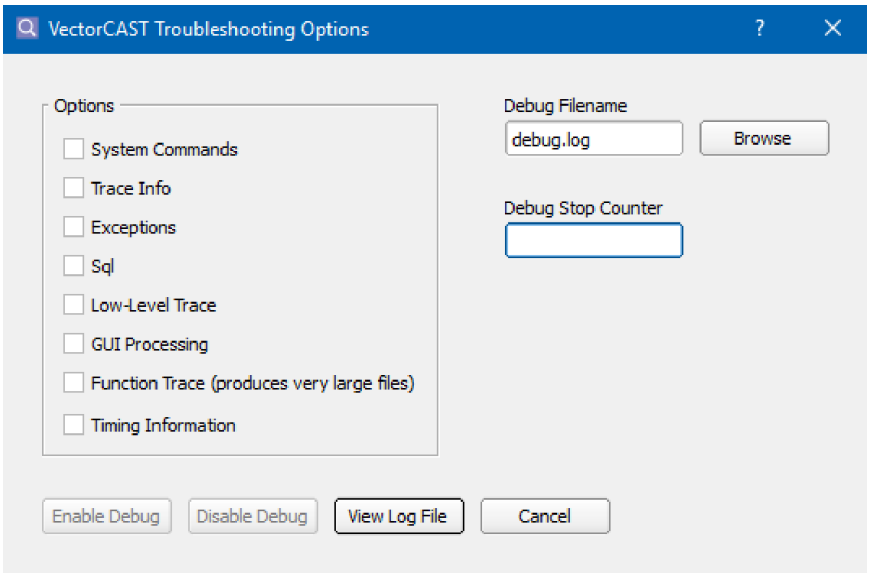
Troubleshooting VectorCAST

Note: This feature should be used only as directed by VectorCAST's Technical Support to resolve issues in the field.

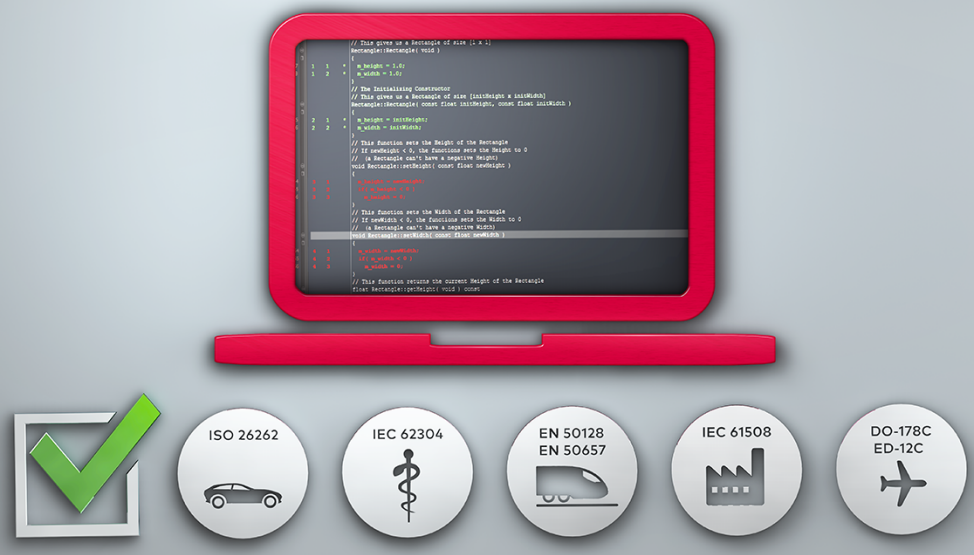
To troubleshoot VectorCAST and create a debug log, choose **Help => Diagnostics => Troubleshooting**. A dialog box appears, allowing you to check which items to want to see in the log. Click the Browse button to specify a location for the file. When you click **Enable Debug**, a debug log is written while you run VectorCAST.

When debug logging is enabled, the default log output location defaults to a file with the name prefix

debug.log. To override this prefix, the environment variable VCAST_DEBUG_FILENAME can be specified. To direct debug log output to **stdout** instead of a file, set VCAST_DEBUG_FILENAME to "-".



To view the log later, choose **Help => Diagnostics => Troubleshooting**. Click the **View Log File** button. In the **Open File** dialog, navigate to the **debug.log.* file**, and click **Open**. The debug log is displayed in the MDI window.



Executing Tests

Executing Test Cases

To Execute a Test Case and View Results

There are several ways to execute a test case:

- Right-click on a test case and select **Execute** from the context menu.
- Select a test case to run, then click this toolbar icon: 
- Select a test case to run, then press the **F5 key** on the keyboard.
- Choose **Test => Batch Execute All**

You can multi-select any combination of test cases, subprograms, and units, before choosing Execute.

When a test case is executing, the Message window gives you feedback on the progress. It informs you as it processes each event and when test execution is complete.

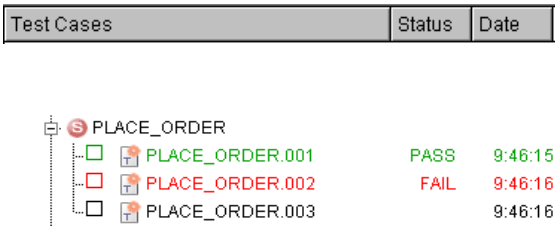
The Execution Status viewer automatically opens when a test case is executed. The Execution Status viewer provides detailed real time test case execution information. See the "Execution Status Viewer" on page 57 for more information.

If a test case contains expected values, those values are compared to the actual values when a test is executed and the Test Case Tree is updated with a PASS or FAIL indication.

```
clicast -e <env> -u <unit> -s <sub> -t <testcase> Execute Run

Execute the specified test case.
```

Test Case Status in the Test Case Tree

The test case name appears in the first column. If all expected values match then the test case passes, and the test case name is displayed in green with the word “PASS” in the Status column. If any expected values do not match, then the test case fails, the test case name is displayed in red, with the word “FAIL” in the Status column.



The time of test execution appears in the Date column. If you open an environment with a test case that you executed a day or more ago, then the date, rather than the time, is displayed in the Date column.

Coverage Checkbox in the Test Case Tree

If the environment has code coverage enabled, then a checkbox appears next to the test case name after it is executed.



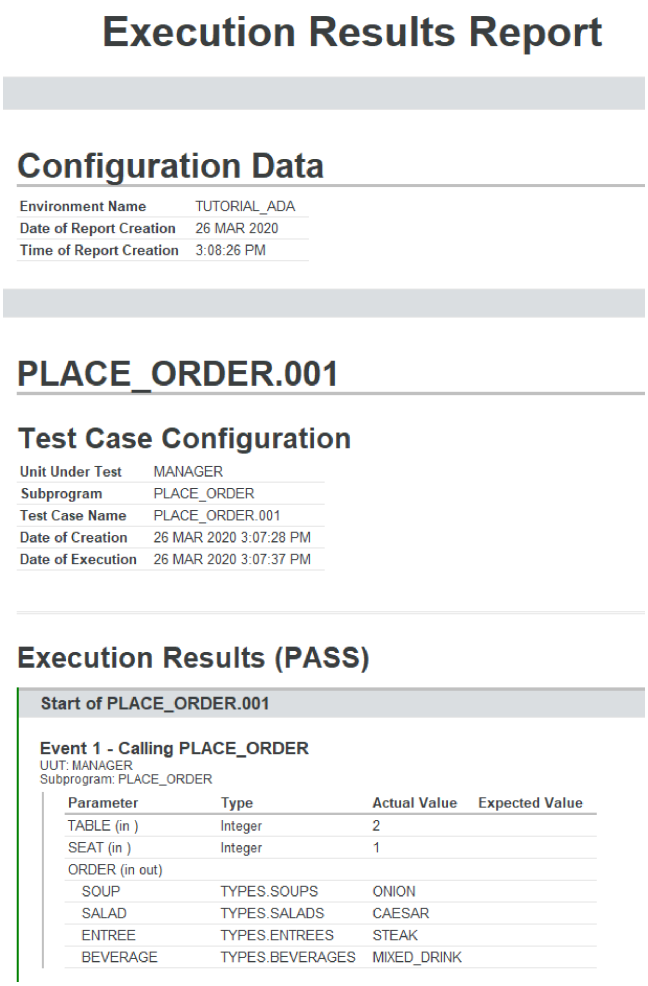
Clicking this checkbox opens the Coverage Viewer and displays the unit's coverage achieved by the selected test cases.

Viewing Execution Results

By default, the test execution results are displayed in a tab on the test case editor window. To view the execution results report, you can either:

- Execute a test case (the Execution Report tab opens automatically)
- Open a test case editor and click the Execution Report tab, or
- Select a test in the Test Case Tree and choose **Test => View => Execution Results**.

The Execution Report tab is automatically opened after a single test case is executed. If you execute multiple test cases, the Test Case Management report is displayed.



Execution Results Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	26 MAR 2020
Time of Report Creation	3:08:26 PM

PLACE_ORDER.001

Test Case Configuration

Unit Under Test	MANAGER
Subprogram	PLACE_ORDER
Test Case Name	PLACE_ORDER.001
Date of Creation	26 MAR 2020 3:07:28 PM
Date of Execution	26 MAR 2020 3:07:37 PM

Execution Results (PASS)

Start of PLACE_ORDER.001

Event 1 - Calling PLACE_ORDER
UUT: MANAGER
Subprogram: PLACE_ORDER

Parameter	Type	Actual Value	Expected Value
TABLE (in)	Integer	2	
SEAT (in)	Integer	1	
ORDER (in out)			
SOUP	TYPES.SOUPS	ONION	
SALAD	TYPES.SALADS	CAESAR	
ENTREE	TYPES.ENTREES	STEAK	
BEVERAGE	TYPES.BEVERAGES	MIXED_DRINK	

If you turn on the “Use external browser” option, located on the GUI tab of the **Tools => Options** dialog, and specify an external browser (IE, FireFox, etc.), then all reports, including the Execution Results, are displayed in a separate browser window. If you would like the Execution Results to be displayed in a report tab, you can turn on the “Use external browser” option but leave the browser unspecified.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <testcase>]]]
Reports Custom ACtual [<outputfile>]

Extract multiple execution results to standard output or the specified
output file. If you have recently changed the VCAST_CUSTOM_REPORT_FORMAT
then you should execute the test cases before using this command.
```

Browsing the Execution Report

When you are viewing an Execution Report, four toolbar buttons are available to easily find the next match or unmatched data item in an Execution Report, Coverage Viewer, Parameter Tree, or Compound Tree.

 Previous matched item (green)

 Next matched item (green)

 Previous unmatched item (red)

 Next unmatched item (red)

Understanding the Execution Report

The Execution Results include a list of calls made by the VectorCAST driver into the UUT and calls made from the UUT into a stubbed dependent subprogram. In both cases, the parameters and data associated with the calls are displayed as well as any global data values that are being monitored. Each transfer of control is an *event*. Event 1 is always the test harness calling the subprogram under test. Events between the first and last events are calls to stubbed functions (if any). The last event is always the return to the test harness.

The maximum number of events is set at 1000 by default, but can be changed for an individual test case or for all test cases.

Expected Values

If a parameter has an input value or expected value, it is included in the Execution Report. If a parameter's value at the time of the call matches the Expected Value or falls within an expected range, then the Expected Values column is colored green for that parameter. If it doesn't match or fall within an expected range, then the column is colored red. If all parameters with Expected Values matched, then the test case status is set to PASS. If any expected values do not match, the test case status is set to FAIL.

A section called "User Code Expected Values" is included at the end of the Execution report when User Code contains an expected value. The expected value for the item is compared to the actual value and if they match, a line is added to the section and colored green. The item is displayed in the left column of the report, and the status, <<match>> or [fail] is displayed in the right column. If the item being compared is lengthy, then the display is truncated to the page width. In HTML, this width is 80 characters. In TEXT format, this width is configurable, using the "Page width" option on the Reports tab, Format tab, Text sub-tab.

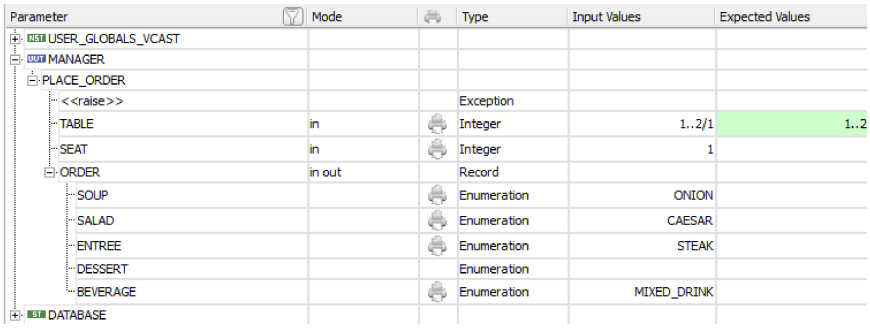
To cause a parameter to be included in the report without having to set an Input Value or a specific Expected Value, enter <<ANY>> as the Expected Value. Doing so causes the parameter to match any value, and also to be included in the report.

If a test case has no Expected Values, then there are no values to match, and so the status is neither PASS nor FAIL.

In the Parameter Tree, the expected values are also colored to indicate PASS or FAIL. If an expected values is provided and it matches at some events in the Execution Report but fails in others, then it is colored yellow in the Test Case Editor.

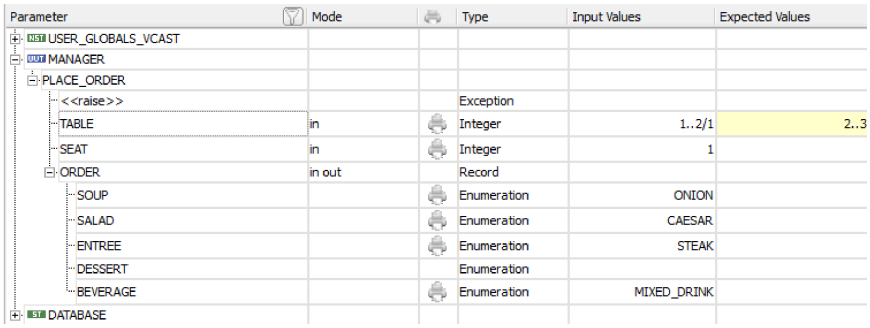
For example, suppose a parameter has a range of input values 1..2, which causes two iterations of the test case. If the expected value for this parameter is set to the range 1..2, then both Input Values

match.



Parameter	Mode	Type	Input Values	Expected Values
USER_GLOBALS_VCAST				
MANAGER				
PLACE_ORDER				
<<raise>>		Exception		
TABLE	in	Integer	1..2/1	1..2
SEAT	in	Integer	1	
ORDER	in out	Record		
SOUP		Enumeration	ONION	
SALAD		Enumeration	CAESAR	
ENTREE		Enumeration	STEAK	
DESSERT		Enumeration		
BEVERAGE		Enumeration	MIXED_DRINK	
DATABASE				

If we change the Expected Value for this parameter to the range 2..3, then the first Input Value (1) does not match, but the second (2) does. This situation results in the expected value cell being colored yellow. (if a test case partially passes and it is part of a compound, then the slot is considered failed. When there are multiple iterations of a slot and some pass in a compound then that slot partially passes.)



Parameter	Mode	Type	Input Values	Expected Values
USER_GLOBALS_VCAST				
MANAGER				
PLACE_ORDER				
<<raise>>		Exception		
TABLE	in	Integer	1..2/1	2..3
SEAT	in	Integer	1	
ORDER	in out	Record		
SOUP		Enumeration	ONION	
SALAD		Enumeration	CAESAR	
ENTREE		Enumeration	STEAK	
DESSERT		Enumeration		
BEVERAGE		Enumeration	MIXED_DRINK	
DATABASE				

Unused Expected Results

If expected results are provided for parameters that are not encountered during test execution, the unused expected results are appended to the Execution Results report and the test case status is set to FAIL. For example, if a list of three expected values is provided for a parameter of a stubbed subprogram, but the subprogram is only called twice (thus providing only two input values to compare), then the third expected value is unused.

For compound test cases, the unused expected results are grouped by slot.

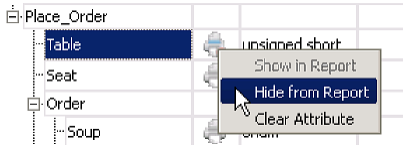
Controlling Report Information

VectorCAST enables you to control the amount of data included in the test result reports. The **Tools => Options** dialog, **Report tab**, **Content sub-tab** contains several options to control the amount of information in the Execution Report.

By default, all parameters, both input and expected, that have values specified in the Parameter Tree, are included on the Test Execution Report.

Optionally, Input Values populated within the Parameter Tree can be hidden on the Test Execution

report. To hide the Input Value, right-click the  icon next to the parameter on the Parameter Tree and select **Hide from Report** on the context menu. This option is not available if the Expected Values column contains data.



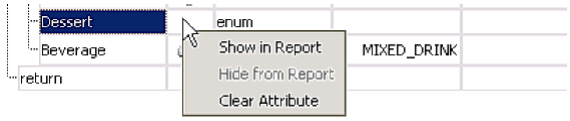
.

The  icon is then displayed indicating that the Input Value is now hidden on the report.

To re-enable the Input Value on the report, right-click the icon and select **Show in Report**. Selecting the **Clear Attribute** allows the value to observe the attribute state for its parent.

For Expected Results with a specified value, the icon appears on the Parameter Tree to indicate that the value will be printed on the Test Execution report, but it is not user controllable and is always included in the report.

For both Input Values and Expected Results parameters that have no values specified, right clicking in the report control column will display the following context menu.



Select **Show in Report** and the force print icon  appears indicating that the parameter is currently included in the report.

To remove the parameter from the report, right-click the **force print** icon. Choose **Hide from Report**, to suppress printing, and the  icon is displayed. Selecting **Clear Attribute** at any time will suppress printing and remove the icon from the Parameter Tree.

In addition, the attribute items are available on composite parameters as well. When the user selects a composite object (e.g. "struct" object) and sets the attribute, all members (recursively) will receive the selected attribute unless directly overridden (with the added limitation that a parameter with an expected value will not inherit the "Hide" attribute).

VectorCAST captures output for variables as follows:

- If you set an input or an expected value for a parameter, VectorCAST will capture the output. This includes the use of the expected value **<<ANY>>**.
- If you set an input value, but you then use the printer option  and select **Hide in Report**, VectorCAST will not capture the output. (Note that VectorCAST needs to capture the output in order to perform the test for the expected value. This scenario sets up a contradiction by hiding the variable and preventing capture.)

> If you do not set an input or an expected value, then you can use the printer option  **Show in Report** and VectorCAST will capture the output.

Output from the Unit Under Test

In some cases, the code under test may generate output to either stdout or stderr. VectorCAST can capture this output and include it in the Execution Report.

See also "Redirect Standard Output" on page 288.

Other Execution and Report Options

There are many other options that control test case execution, report content, and report format.

Deprecated Report Options and Commands

VCAST_COMPACT was deprecated in VectorCAST version 4.2. It is always true.

VCAST_EXPECTED_USER_CODE_EACH_EVENT was deprecated in VectorCAST version 4.0.

`clicast execute extract coverage` was deprecated in VectorCAST version 4.0. Use **`clicast reports custom coverage`**.

`clicast execute extract results` was deprecated in VectorCAST version 4.0. Use **`clicast reports custom actual`**.

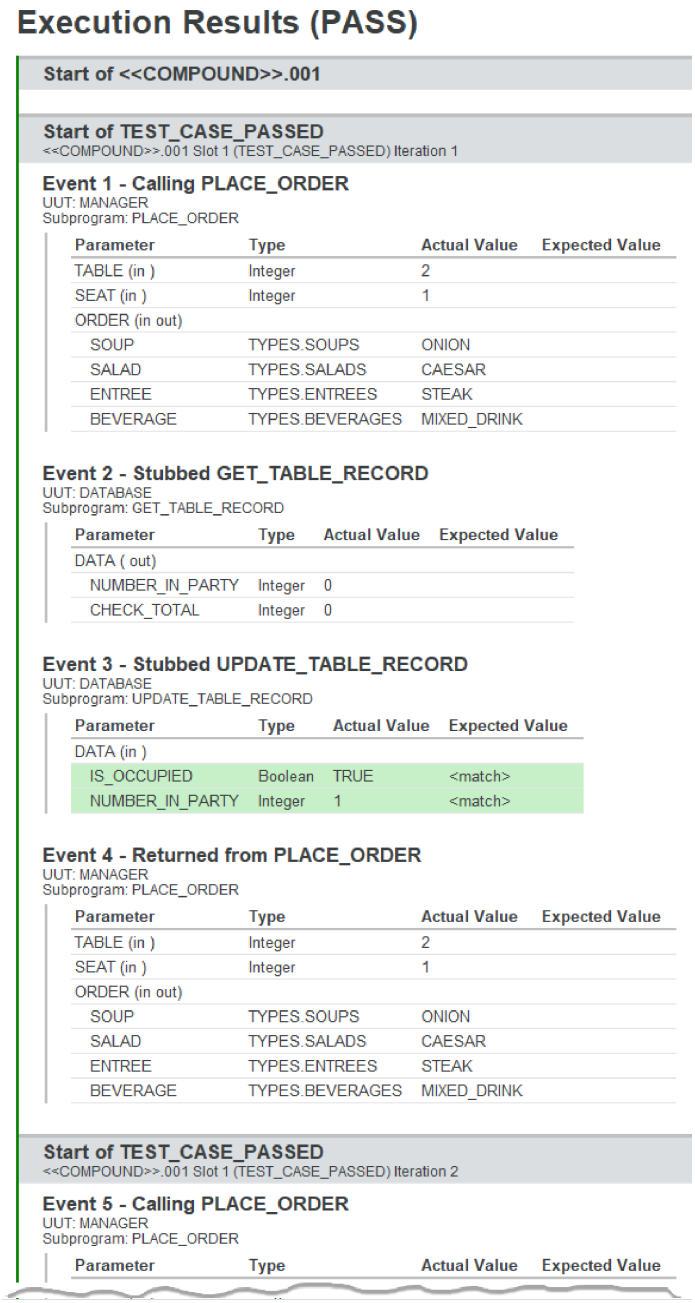
`clicast reports report` was deprecated in VectorCAST version 4.0. Use **`clicast reports custom full`**.

Compound Test Case Execution and Reports

The test cases in a compound test are executed in order, from top to bottom. Data is persistent between test cases in a compound test case, unlike data between simple test cases that are executed together. An <<INIT>> test case in the first slot can be used to initialize variables or instantiate class pointers that subsequent test cases access.

By default, each slot in a compound is executed once, unless the number of iterations is specified. If the cumulative number of events exceeds the environment-wide Event Limit (specified on the **Tools => Options dialog**, **Execute tab**), compound test case execution terminates, with a message in the Execution Report. For test cases that have their own Event Limit specified on their Options tab, then that Event Limit is used when comparing the cumulative number of events. Following that, the environment-wide Event Limit is used.

The Execution Report for a compound test case is very similar to that of a simple test case. The only difference is in the Event header for an Event that marks the start of a new slot. The first slot is executed starting with Event 1, and the second slot's start is indicated as "Event *n*, Slot 2, Iteration 1." If you have specified that a slot iterate, then the Execution Report reflects the slot being called multiple times.



Execution Results (PASS)

Start of <<COMPOUND>>.001

Start of TEST_CASE_PASSED
<<COMPOUND>>.001 Slot 1 (TEST_CASE_PASSED) Iteration 1

Event 1 - Calling PLACE_ORDER
UUT: MANAGER
Subprogram: PLACE_ORDER

Parameter	Type	Actual Value	Expected Value
TABLE (in)	Integer	2	
SEAT (in)	Integer	1	
ORDER (in out)			
SOUP	TYPES.SOUPS	ONION	
SALAD	TYPES.SALADS	CAESAR	
ENTREE	TYPES.ENTREES	STEAK	
BEVERAGE	TYPES.BEVERAGES	MIXED_DRINK	

Event 2 - Stubbed GET_TABLE_RECORD
UUT: DATABASE
Subprogram: GET_TABLE_RECORD

Parameter	Type	Actual Value	Expected Value
DATA (out)			
NUMBER_IN_PARTY	Integer	0	
CHECK_TOTAL	Integer	0	

Event 3 - Stubbed UPDATE_TABLE_RECORD
UUT: DATABASE
Subprogram: UPDATE_TABLE_RECORD

Parameter	Type	Actual Value	Expected Value
DATA (in)			
IS_OCCUPIED	Boolean	TRUE	<match>
NUMBER_IN_PARTY	Integer	1	<match>

Event 4 - Returned from PLACE_ORDER
UUT: MANAGER
Subprogram: PLACE_ORDER

Parameter	Type	Actual Value	Expected Value
TABLE (in)	Integer	2	
SEAT (in)	Integer	1	
ORDER (in out)			
SOUP	TYPES.SOUPS	ONION	
SALAD	TYPES.SALADS	CAESAR	
ENTREE	TYPES.ENTREES	STEAK	
BEVERAGE	TYPES.BEVERAGES	MIXED_DRINK	

Start of TEST_CASE_PASSED
<<COMPOUND>>.001 Slot 1 (TEST_CASE_PASSED) Iteration 2

Event 5 - Calling PLACE_ORDER
UUT: MANAGER
Subprogram: PLACE_ORDER

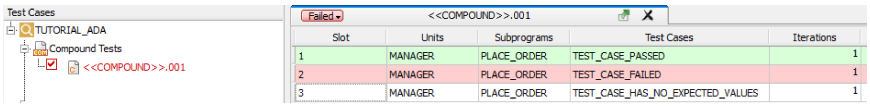
Parameter	Type	Actual Value	Expected Value

See " To Specify the Number of Iterations of a Test in a Compound" on page 168 for information on specifying slot iterations.

The Compound Tree

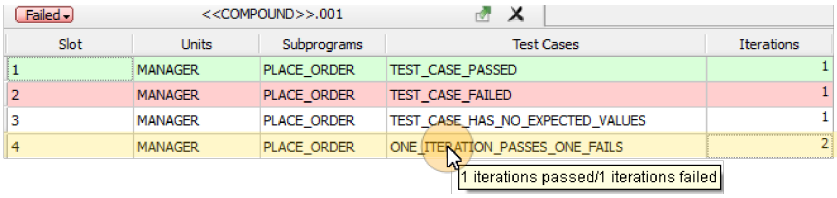
As with the Parameter Tree for simple test cases, the Compound Tree tab reflects the PASS or FAIL status of the compound test case. If all slots in the compound passed or had no expected values, then the compound as a whole passes. If any slot in the compound fails, then the compound as a whole fails.

For example, suppose we have a compound test case with three slots: a test case that passes, a test case that fails, and a test case that has no expected values, so it neither passes nor fails. When this compound is executed, it is given the status of FAIL as a whole, due to slot 2 failing.



The slots in the Compound Test Case are colored to reflect the pass or fail status of each individual slot. A slot that passes is colored green; a slot that fails is colored red, and a slot that has no expected values is left uncolored.

If a slot has more than one iteration specified, then all iterations of that test case must pass in order for that slot to be considered PASSed. In the example below, a test case was added to the compound. One of the parameters in the test case has an Expected List of 2 values, but only 1 input value. The first input will pass, but the second expected value won't match. In the compound, the number of iterations is set to 2. When the compound is executed, this 4th slot is colored yellow, because the first iteration passed, but the second iteration failed, as shown in the tool-tip.



To Execute Multiple Test Cases

There are several ways to execute multiple test cases. The **Test => Batch Execute All** command enables you to execute all test cases. All simple and compound test cases are executed in order from top to bottom of the Test Case Tree. There is no interaction of data between the test cases, and the Event Limit starts over with each test case. When you execute multiple test cases, the Test Case Management report appears.

Alternatively, you can right-click the top node of the Test Case Tree (the environment name) and choose **Execute**, which also executes all test cases. If you click a node other than the top node of the Test Case Tree, then only the test cases under that node are executed.

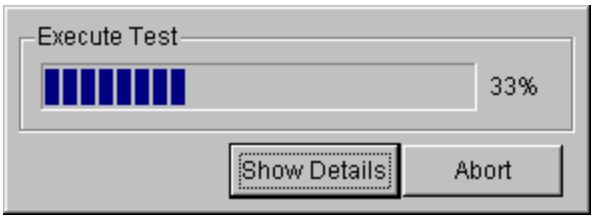
If a test case is marked "Compound Only" it is only executed when its compound test is executed.

The keyboard shortcut for Batch Execute All is the **F6** key.

```
clicast -e <env> [-u <unit> [-s <sub>]] EXecute Batch

Execute multiple test cases. To run all test cases for a subprogram,
specify the subprogram name on the command. To run all tests for an
environment, do not specify the subprogram name.
```

To Abort Test Case Execution

When executing a test case that exceeds two or three seconds in duration or when executing multiple test cases, the following dialog appears. Click the **Abort** button to halt the test case execution and return control to VectorCAST. This is useful for test cases that are running indefinitely, not responding, or when target communication is suspect.



To Execute a Test Case in the Debugger

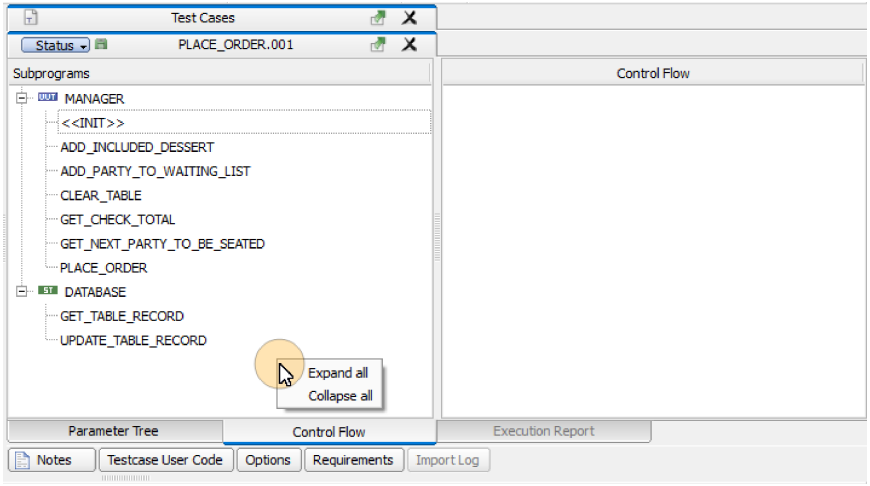
It is often desirable to execute test cases under control of the underlying compiler's debugger. To do this, select the test case name, right-click, and choose **Execute with Debug**. This results in the debugger being launched with the VectorCAST test harness code loaded. (The VectorCAST window is not useable at this time.) From the debug window you can perform all debug commands as you would normally. Upon quitting from the debugger, control is passed back to VectorCAST. The VectorCAST reports generated will reflect the execution history from the debug session.

```
clicast -e <env> -u <unit> -s <sub> -t <testcase> EXecute Debug

Execute the specified test case under control of the debugger.
```

Working with a Control Flow

To view or edit a test case control flow, click the **Control Flow** tab in the Test Case Editor. The left hand side of the Control Flow screen, the *Subprograms* panel, contains a list of UUT subprograms, non-stubbed units as well as stubbed units. <<INIT>> is listed once, under the first unit.

To expand all nodes of the tree, right-click anywhere in the *Subprograms* panel and select Expand All from the context menu. To collapse the tree, select Collapse All from the context menu.

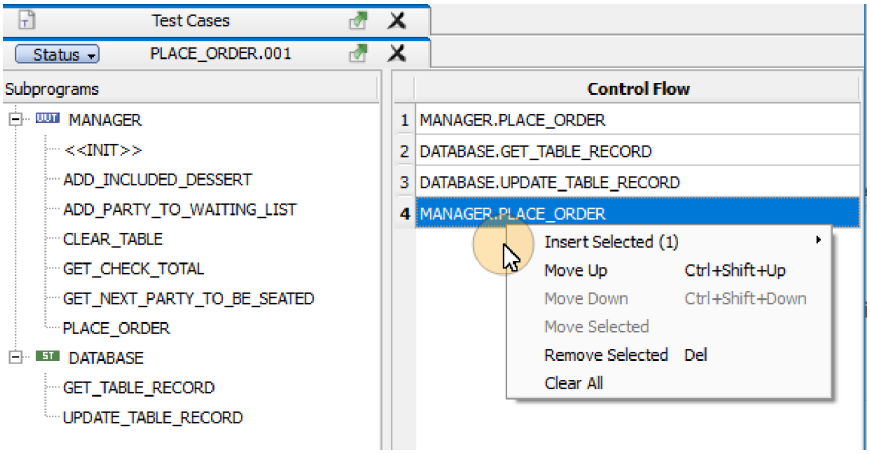


If the *Control Flow* panel is empty, a control flow was not created or saved for the test case. See “To Save the Control Flow” on page 267.

You can manually add a subprogram to the control flow by double-clicking any subprogram listed in the Subprogram panel. The item will be added to the bottom of the list in the Control Flow panel

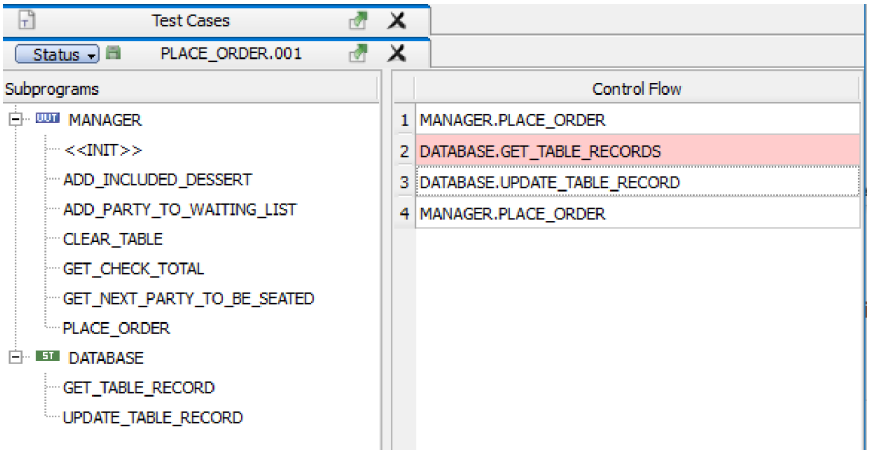
You can edit the control flow by right-clicking on any item in the Control Flow panel. Using the context menu you can:

- **Insert Selected** – this menu item will duplicate the selected subprogram and place a copy directly below the selected subprogram in the Control Flow panel.
- **Move-Up** – the selected subprogram will move up one position in the control flow. This selection choice is inactive if the subprogram selected is at the top of the list.
- **Move-Down** - the selected subprogram will move down one position in the control flow. This selection choice is inactive if the subprogram selected is at the bottom of the list.
- **Move-Selected** – this menu item allows you to select and move one or more subprograms to a different position in the Control Flow. To activate this feature, first, select the subprograms you wish to move by CTRL-clicking each subprogram. Next, select the “move to” position by CTRL+right-clicking the new position. When the context menu appears, select **Move Selected.** The subprograms will be moved just above the CTRL+right-click position.
- **Remove** Selected– delete the selected subprogram or subprograms from the control flow.
- **Clear All** – delete all subprograms in the control flow. See also, “To Clear the Control Flow”.



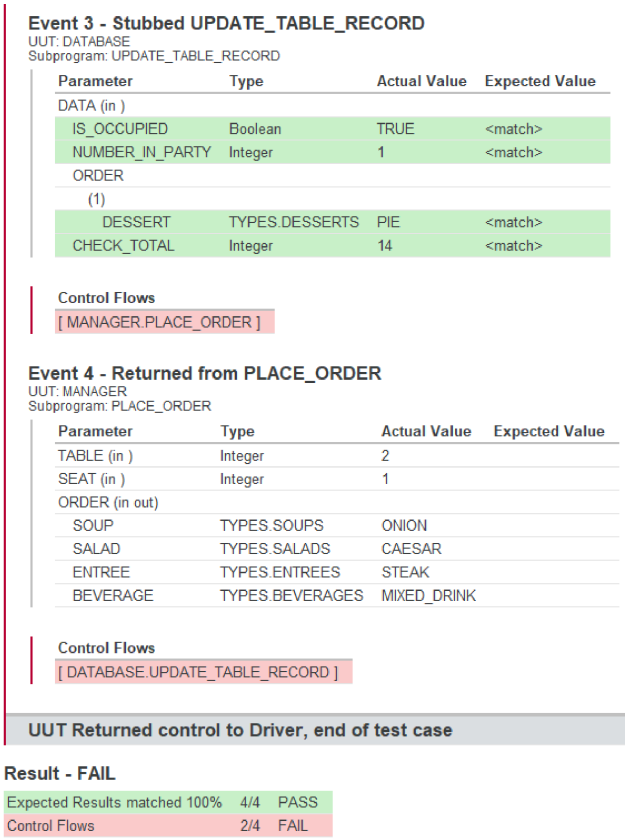
Stubbed subprograms listed in the Control Flow panel are identified with the prefix identifier “uut_ prototype_stubs.” for easy recognition.

By double-clicking a subprogram in the control flow list, you can edit the name of the subprogram. If you accidentally enter a subprogram name that is either misspelled or does not exist, the entry will have a light red background to alert you to the error.



Control flow can change as a result of new source code being introduced into the test environment or as a result of manually editing the control flow.

If a control flow has changed, the Test Execution Report identifies the event where the control flow mismatch occurred as well as providing a list missing control flow events.



Event 3 - Stubbed UPDATE_TABLE_RECORD
UUT: DATABASE
Subprogram: UPDATE_TABLE_RECORD

Parameter	Type	Actual Value	Expected Value
DATA (in)			
IS_OCCUPIED	Boolean	TRUE	<match>
NUMBER_IN_PARTY	Integer	1	<match>
ORDER			
(1)			
DESSERT	TYPES.DESSERTS	PIE	<match>
CHECK_TOTAL	Integer	14	<match>

Control Flows
[MANAGER.PLACE_ORDER]

Event 4 - Returned from PLACE_ORDER
UUT: MANAGER
Subprogram: PLACE_ORDER

Parameter	Type	Actual Value	Expected Value
TABLE (in)	Integer	2	
SEAT (in)	Integer	1	
ORDER (in out)			
SOUP	TYPES.SOUPS	ONION	
SALAD	TYPES.SALADS	CAESAR	
ENTREE	TYPES.ENTREES	STEAK	
BEVERAGE	TYPES.BEVERAGES	MIXED_DRINK	

Control Flows
[DATABASE.UPDATE_TABLE_RECORD]

UUT Returned control to Driver, end of test case

Result - FAIL

Expected Results matched 100%	4/4	PASS
Control Flows	2/4	FAIL

To Save the Control Flow

The Save Control Flow command enables you to save the execution flow of a particular test run for comparison against future runs. To use this command you first create and execute a test case. If, after viewing the results, you wish to save the order of the calls as an expected result, select **Test => Control Flow => Save**. Future runs of the same test case result in a comparison of the call sequence to the saved call sequence. Using this command results in an additional comparison of expected and actual results for pass/fail metrics.

```
clicast -e <env> -u <unit> -s <sub> -t <testcase> EXecute Save Flow

Save the control flow for the specified test case.
```

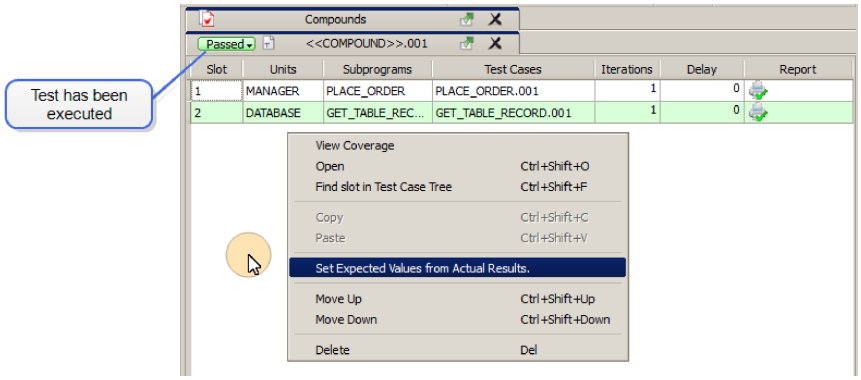
To Clear the Control Flow

If you no longer wish to use the saved control flow for a particular test case, simply select the test case and select **Test => Control Flow => Clear**. Future runs of the same test case will no longer compare the call sequence with the saved call sequence.

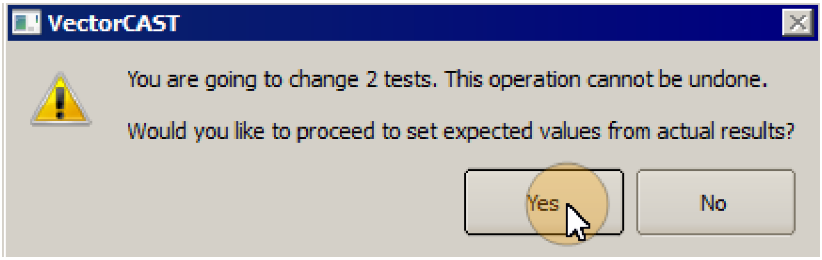
```
clicast -e <env> -u <unit> -s <sub> -t <testcase> EXecute Clear Flow

Clear the control flow for the specified test case.
```

To Set Expected Values to Actual Values

After executing a test case, Expected Values can be set to the actual execution result values. Right-click anywhere in the Compound Test Editor and select **Set Expected Values from Actual Results** from the context menu.



A warning dialog appears indicating the number of test cases that will be changed by the action. You have the opportunity to abort the action by selecting the **No** button. Selecting the **Yes** button will allow the operation to proceed and modify the test cases. Note that this action cannot be undone.



All Expected Values listed in the Execution Report are set to their actual values.

See Controlling Report Information for more information on capturing output for variables.

```
clicast -l<lang> -e <env> [-u <unit> [-s <sub> [-t <testcase>]]] TESt
Actuals_to_expected

This will set the actual values for a test case as the expected values.
Thus 100% of the expected results will match for that test case.
```

Viewing Test Reports

The **Test => View** commands reflect the item selected in the Test Case Tree. For example, if a subprogram is selected in the Test Case Tree, then the report generated using the **Test => View** menu reflects the data from the test cases for that subprogram.

VectorCAST provides the ability to control what coverage data appears in reports containing the Aggregate Coverage Report or Metrics Report. To set the coverage data used, select **Test => View => Coverage data used in reports** from the Menu Bar. Then, select one of the sub-menu options:

- **All results** (default) - Use coverage data from all test results in the environment, even if unchecked (unselected) when generating reports.
- **Only checked results** - Use coverage data only from the checked (selected) test results when generating reports.

The setting is saved in the user's HOME directory, and thus applies to all unit test environments.



Note: This option does not apply to reports generated via **`clicast`**, which uses all results.

View Reports in an External Browser

If you prefer to view HTML reports in an external browser (IE or Mozilla, for example), then you can specify an external browser in the **Tools => Options** dialog, **GUI** tab, under the subtab, **External Browser Options**. There you can add the command to start the browser. To use the external browser, select "**Use external browser**".

If you switch your Report Format from the default HTML to TEXT, then you can view the test reports in an external text editor. If you specified an external editor (notepad or emacs, for example) and indicated that it should be used as the File Viewer in the **Tools => Options dialog, GUI tab**, then the test reports are displayed in that external editor.

View a Test Case Data Report

The **Test => View => Test Case Data** command enables you to view the contents of a test case, once it has been saved. The contents of this report reflects the item(s) selected in the Test Case Tree.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <test>]]]
Reports Custom Test <outputfile>

Create a Test Case Data report and output to HTML file <env>_test_case_
data_report.html, standard output as text, or to the specified output file.
If the Unit parameter is specified, the report includes the Test Case Data
for all test cases for the specified unit. If the Unit and Subprogram
parameters are specified, then the report includes the Test Case Data for
only those test cases for that subprogram of the specified unit. If the
Unit, Subprogram, and Test Case parameters are specified, then the report
includes the Test Case Data for only the specified test case. Otherwise,
the report includes the execution results for all test cases in the
environment, including <<COMPOUND>> and <<INIT>>.
```

The following report is the result of the **Test => View => Test Case Data** command for a simple test case.

Test Case Configuration

Unit Under Test	MANAGER
Subprogram	PLACE_ORDER
Test Case Name	PLACE_ORDER.001
Date of Creation	26 MAR 2020 3:28:42 PM
Date of Execution	No Execution Results Exist

Test Case Data

Test Case	PLACE_ORDER.001
File Name	C-000001.DAT

Requirements/Notes

```
This test is the same as the one created by following all of
the steps in the "Ada Tutorials -> Basic Tutorial" from the
VectorCAST Getting Started manual.

It shows the basic concepts associated with setting input and
expected values for both the Unit Under Test and Stub Functions.
```

Input Test Data

UUT: MANAGER		
Subprogram: PLACE_ORDER		
TABLE (in)	Integer	2
SEAT (in)	Integer	1
ORDER (in out)		
SOUP	TYPES.SOUPS	ONION
SALAD	TYPES.SALADS	CAESAR
ENTREE	TYPES.ENTREES	STEAK
BEVERAGE	TYPES.BEVERAGES	MIXED_DRINK
Stubbed Unit: DATABASE		
Subprogram: GET_TABLE_RECORD		
DATA (out)		
NUMBER_IN_PARTY	Integer	0
CHECK_TOTAL	Integer	0

Expected Test Data

Stubbed Unit: DATABASE		
Subprogram: UPDATE_TABLE_RECORD		
DATA (in)		
IS_OCCUPIED	Boolean	TRUE
NUMBER_IN_PARTY	Integer	1
ORDER		
(1)		
DESSERT	TYPES.DESSERTS	PIE
CHECK_TOTAL	Integer	BETWEEN:12 AND:16

Test Case / Parameter Input User Code

This test has no input user code.

Test Case / Parameter Expected User Code

This test has no expected user code.

To View Test Execution Results

The **Test => View => Execution Results** command enables you to view the execution results for a test case, a subprogram, or environment, depending on where you clicked before choosing the menu item. The report is the same as the one that appears immediately after executing that test case. If the

menu item is dimmed, this means the test case has not been executed.

The contents of this report reflects the items selected in the Test Case Tree. If you click on a subprogram name or a unit name, then the Execution Report contains results for all test cases for that subprogram or unit. If you click on the name of the environment, then the report contains results for all compound test cases, simple test cases, and compound-only test cases.

Similarly, if you click on <<COMPOUND>> or <<INIT>>, then the report contains results for all Compound test cases or Init test cases, respectively.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <test>]]]
Reports Custom ACtual <outputfile>

Create an Execution Results report and output to HTML file <env>_execution_
results_report.html, standard output as text, or to the specified output
file. If the Unit parameter is specified, the report includes the execution
results for all test cases for the specified unit. If the Unit and
Subprogram parameters are specified, then the report includes the execution
results for only those test cases for that subprogram of the specified
unit. If the Unit, Subprogram, and Test Case parameters are specified, then
the report includes the execution results for only the specified test case.
Otherwise, the report includes the execution results for all test cases in
the environment, including <<COMPOUND>> and <<INIT>>.
```

See also "To Execute a Test Case and View Results" on page 255 for more information on Execution Reports.

To Create a CSV Execution Report

To create a CSV Execution Report, follow these steps:

1. Before starting VectorCAST, set the environment variable VCAST_ALTERNATE_REPORT to 1.
2. Start VectorCAST and open desired environment.
3. Execute test cases.
4. Exit VectorCAST.
5. From a shell window in the working directory, execute **`clicast -e <env> reports alternate <outputfile>`**

The default separating character is a comma (,), but this can be changed using the “Delimiter” option on the Tools => Options dialog, Report tab, Format sub-tab, Text sub-sub-tab. To enter a different character as the delimiter, type it in the edit box. To enter a tab, use “\t”.

The following is an example of the output.

```
Test Case Name,Name,Input Value,Expected Value,Actual Value,Pass/Fail
SKY_CONDITION.001
```

```
15 OCT 2008 04:26:36 PM
,sky.sky_condition.cloud_coverage,50
,sky.sky_condition.cloud_coverage,,PartlyCloudy,PartlyCloudy,pass
TEMPERATURE_DESCRIPTION-PATH-1
15 OCT 2008 04:26:39 PM
This is an automatically generated test case.
   Test Path 1
      ( 1) case ( t ) ==> Cold
   Test Case Generation Notes:
,temperature.temperature_description.t,Cold
CELSIUS_TO_FAHRENHEIT-PATH-1
15 OCT 2008 04:26:44 PM
   No branches in subprogram
   Test Case Generation Notes:
,temperature.celsius_to_fahrenheit.celsius,100
,temperature.celsius_to_fahrenheit.celsius,,212,212,pass
VCAST_MAIN.001
15 OCT 2008 04:26:44 PM
Temperature 21 C = 70 F
,weather.VCAST_main.argc,3
,weather.VCAST_main.argv,<<malloc 3>>
,weather.VCAST_main.argv[0],<<malloc 8>>
,weather.VCAST_main.argv[0],"weather"
,weather.VCAST_main.argv[1],<<malloc 3>>
,weather.VCAST_main.argv[1],"21"
,weather.VCAST_main.argv[2],<<malloc 3>>
,weather.VCAST_main.argv[2],"50"
,weather.VCAST_main.argv...,,0,0,pass
```

clicast -e <env> Reports Alternate <outputfile>

Create a character-separated Execution Results report to standard output or the specified output file. The default separator is a comma unless specified with the VCAST_RPTS_DELIMITER option. Test cases must first be executed with environment variable VCAST_ALTERNATE_REPORT set.

The Full Report

The **Test => View => Full Report** command generates a single report that includes the following sections:

- Table of Contents
- Configuration Data
- Overall Results
- Configure Stubs User Code
- Environment User Code
- Test Case Configuration
- Test Case Data Report

- Execution Results Report
- Aggregate Coverage Report
- Metrics Report
- Probe Points

The contents of this report reflects the items selected in the Test Case Tree. Much of the full report is the same for all test cases. The test case data and execution results are specific to the test case(s) included in the report.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <testcase>]]]
Reports Custom Full <outputfile>

Create a Full report and output to HTML file <env>_full_report.html,
standard output as text, or to the specified output file. The Full report
includes the Aggregate Coverage data for all units or the specified unit,
the Metrics table for all units or the specified unit, the Test Case Data
report for the test case(s) specified, and the Execution Results report for
the test case(s) specified. The Subprogram and Test Case parameters impact
only the Test Case Data report and the Execution Results sections.
```

See also "View a Test Case Data Report" on page 269

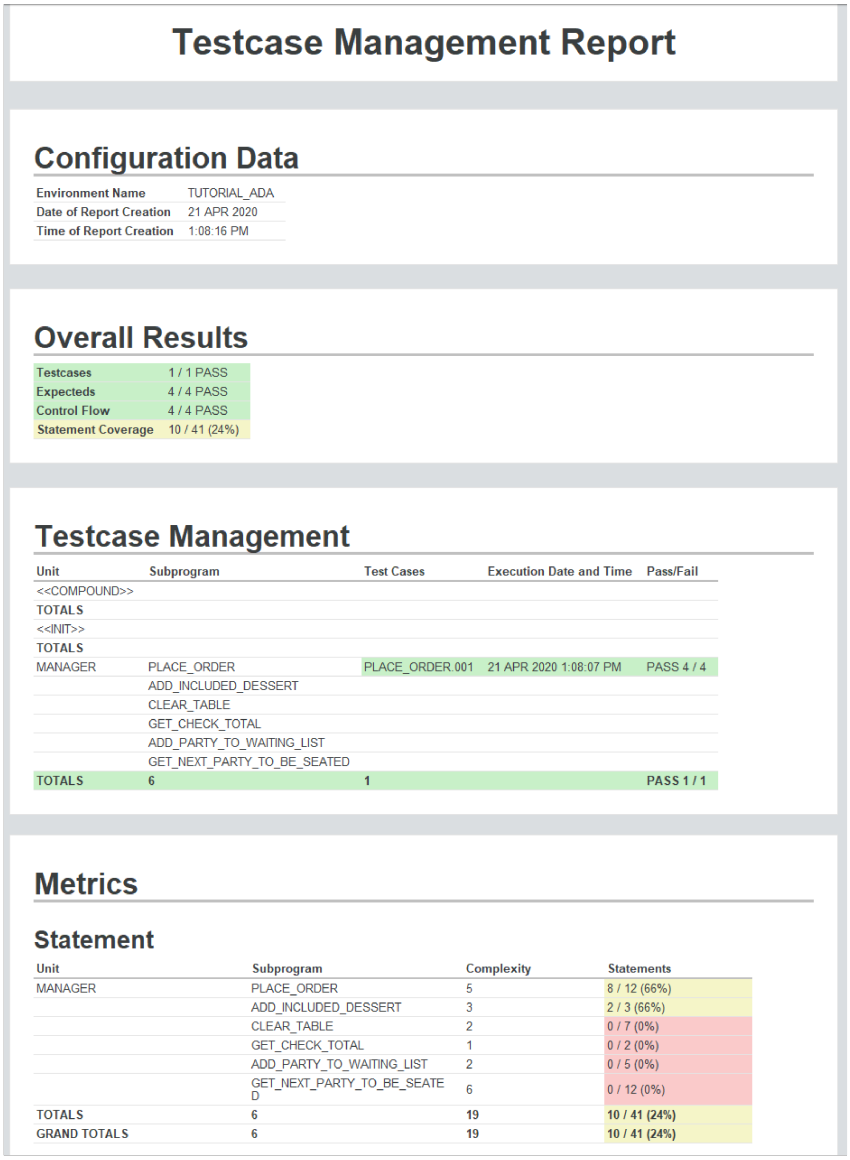
See also "To Execute a Test Case and View Results" on page 255.

The Test Case Management Report

The Test Case Management Report is a summary of the testing status for a particular environment. The report contains information relating to pass/fail status for each executed test case, code coverage summary, code complexity metrics, overall pass/fail status for all test cases, and the number of Expected Value comparisons passing out of the total number of expected values. The contents of this report reflect the items selected in the Test Case Tree.

```
clicast -e <env> [-u <unit>] REports Custom MAnagement <outputfile>

Create a Test Case Management report and output to HTML file <env>_
management_report.html, standard output as text, or to the specified output
file. If the Unit parameter is specified, the report includes the Overall
Execution and Coverage summary for the specified unit, the Pass/Fail status
for all test cases in that unit, and the Metrics report for only the
specified unit. Otherwise, the report includes Overall Execution and
Coverage summary for the environment, the Pass/Fail status for all test
cases, including <<INIT>> and <<COMPOUND>>, and the Metrics report for all
units.
```

The following is an example Test Case Management report.



Testcase Management Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	21 APR 2020
Time of Report Creation	1:08:16 PM

Overall Results

Testcases	1 / 1 PASS
Expecteds	4 / 4 PASS
Control Flow	4 / 4 PASS
Statement Coverage	10 / 41 (24%)

Testcase Management

Unit	Subprogram	Test Cases	Execution Date and Time	Pass/Fail
<<COMPOUND>>				
TOTALS				
<<INIT>>				
TOTALS				
MANAGER	PLACE_ORDER	PLACE_ORDER.001	21 APR 2020 1:08:07 PM	PASS 4 / 4
	ADD_INCLUDED_DESSERT			
	CLEAR_TABLE			
	GET_CHECK_TOTAL			
	ADD_PARTY_TO_WAITING_LIST			
	GET_NEXT_PARTY_TO_BE_SEATED			
TOTALS	6	1		PASS 1 / 1

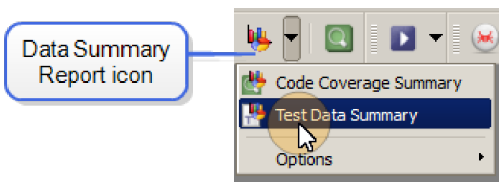
Metrics

Statement

Unit	Subprogram	Complexity	Statements
MANAGER	PLACE_ORDER	5	8 / 12 (66%)
	ADD_INCLUDED_DESSERT	3	2 / 3 (66%)
	CLEAR_TABLE	2	0 / 7 (0%)
	GET_CHECK_TOTAL	1	0 / 2 (0%)
	ADD_PARTY_TO_WAITING_LIST	2	0 / 5 (0%)
	GET_NEXT_PARTY_TO_BE_SEATED	6	0 / 12 (0%)
TOTALS	6	19	10 / 41 (24%)
GRAND TOTALS	6	19	10 / 41 (24%)

View Test Data Summary

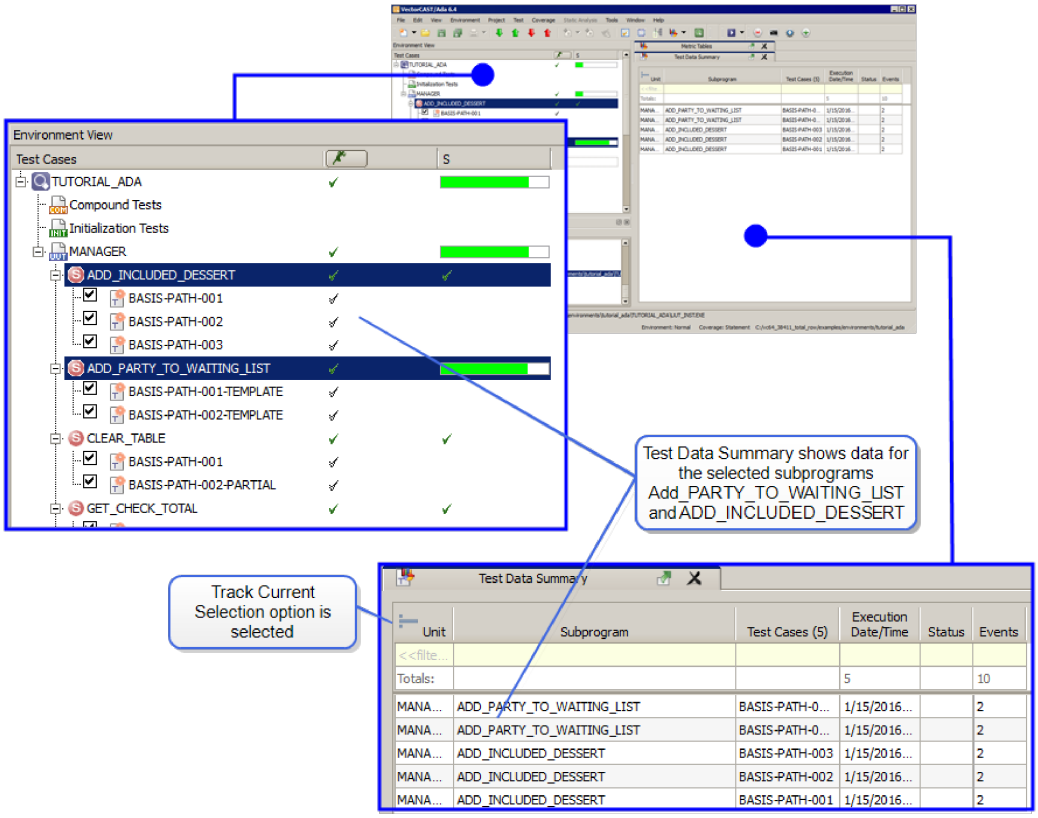
The Test Data Summary allows the user to easily view a summary of test data for UUTs, subprograms and test cases in an environment. To open the summary from the Toolbar, select **Test Data Summary** from the Data Summary Report icon drop-down menu.



Viewing Test Data

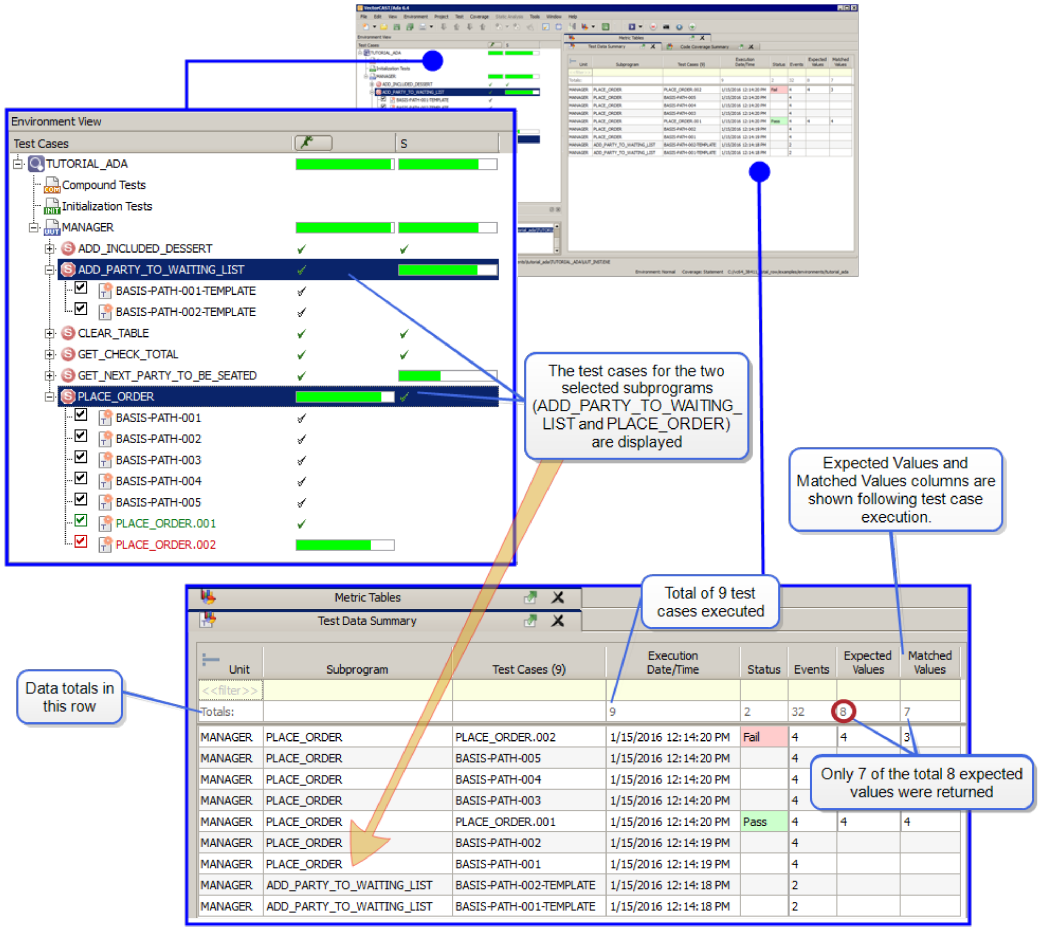
By default, the contents of the Test Data Summary reflect the items selected in the Test Case Tree, and show test case data for test cases, subprograms, UUTs or an environment, depending on where you clicked before selecting the menu item. A tracking icon is displayed at the top of the Unit column indicating that the Test Data Summary is currently tracking selected items from the Test Case Tree.

To override tracking of selected items in the Test Case Tree, open the drop-down menu for the Data Summary Report and select **Options => Track Current Selection**. Remove the check next to the option. The tracking icon on the Summary table will change to gray to indicate that the summary is now tracking all items in the Test Case Tree.



The Test Data Summary table is dynamic. When the **Track Current Selection** option is enabled, as nodes are selected and deselected in the Test Case Tree, the Test Data Summary table updates in real time reflecting the selections.

The data displayed in the Test Data Summary includes the Unit name, Subprogram name, Test Case name, most recent Execution Date/Time, Status and Events for each tracked selection. Additional columns are added to the Test Data Summary when that data is available. These additional columns are: Expected Values, Matched Values, Control Flow, Signals, Exceptions and Abnormal Termination.



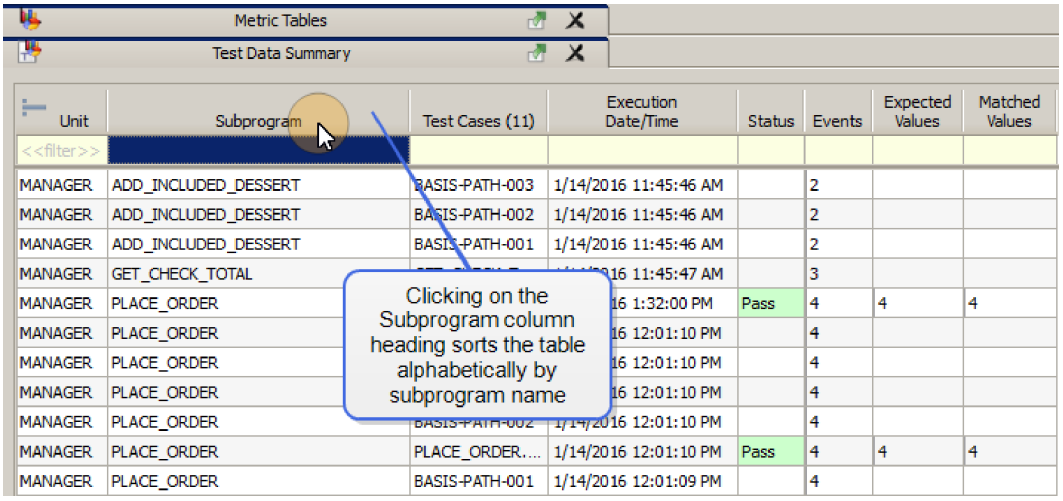
Totals are provided for the data being tracked. The Totals row at the top of the table displays the totals for each data column. In the example above, note that we have a total of 9 test cases executed, and of the 8 total expected values, 7 were matched.

The Summary table updates whenever test data is updated. For example, the table refreshes when test cases are inserted, deleted, or duplicated, or following test execution.

Double-clicking on a line in the Test Data Summary opens the corresponding test case in the Test Case editor.

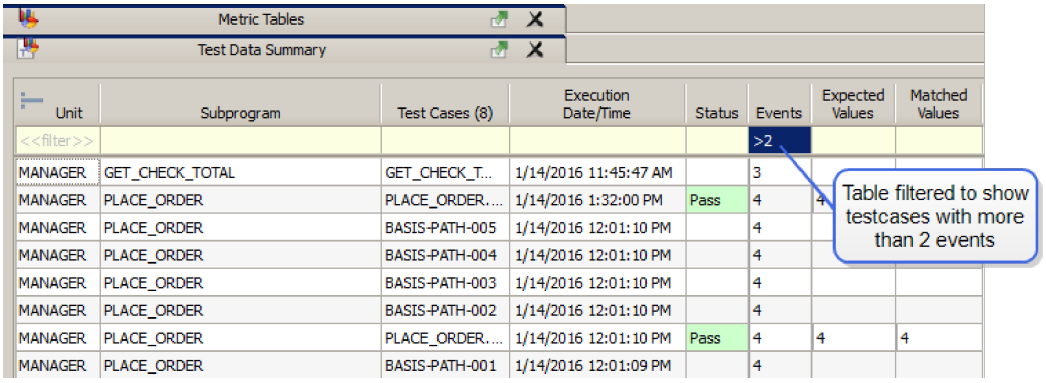
Sorting and Filtering Test Data

Sorting and filtering is available to locate data of interest. Sort by clicking on any column heading. The data will sort in alphabetic or numeric order, as appropriate. Clicking the heading again reverses the order. By default, the table sorts using the Execution Date/Time column, showing the most recently executed test cases first.



Unit	Subprogram	Test Cases (11)	Execution Date/Time	Status	Events	Expected Values	Matched Values
<<filter>>							
MANAGER	ADD_INCLUDED_DESSERT	BASIS-PATH-003	1/14/2016 11:45:46 AM		2		
MANAGER	ADD_INCLUDED_DESSERT	BASIS-PATH-002	1/14/2016 11:45:46 AM		2		
MANAGER	ADD_INCLUDED_DESSERT	BASIS-PATH-001	1/14/2016 11:45:46 AM		2		
MANAGER	GET_CHECK_TOTAL		16 11:45:47 AM		3		
MANAGER	PLACE_ORDER		16 1:32:00 PM	Pass	4	4	4
MANAGER	PLACE_ORDER		16 12:01:10 PM		4		
MANAGER	PLACE_ORDER		16 12:01:10 PM		4		
MANAGER	PLACE_ORDER		16 12:01:10 PM		4		
MANAGER	PLACE_ORDER		16 12:01:10 PM		4		
MANAGER	PLACE_ORDER	PLACE_ORDER....	1/14/2016 12:01:10 PM	Pass	4	4	4
MANAGER	PLACE_ORDER	BASIS-PATH-001	1/14/2016 12:01:09 PM		4		

Test data can be accessed all the way down to the individual test case level by filtering. Access the filter by typing into the top row of any column. Clear the filter by right-clicking in the top row and selecting **Clear Filter** from the context menu. In the example below, the table has been filtered to only show data for test cases with more than 3 events.



Unit	Subprogram	Test Cases (8)	Execution Date/Time	Status	Events	Expected Values	Matched Values
<<filter>>					>2		
MANAGER	GET_CHECK_TOTAL	GET_CHECK_T...	1/14/2016 11:45:47 AM		3		
MANAGER	PLACE_ORDER	PLACE_ORDER....	1/14/2016 1:32:00 PM	Pass	4	4	
MANAGER	PLACE_ORDER	BASIS-PATH-005	1/14/2016 12:01:10 PM		4		
MANAGER	PLACE_ORDER	BASIS-PATH-004	1/14/2016 12:01:10 PM		4		
MANAGER	PLACE_ORDER	BASIS-PATH-003	1/14/2016 12:01:10 PM		4		
MANAGER	PLACE_ORDER	BASIS-PATH-002	1/14/2016 12:01:10 PM		4		
MANAGER	PLACE_ORDER	PLACE_ORDER....	1/14/2016 12:01:10 PM	Pass	4	4	4
MANAGER	PLACE_ORDER	BASIS-PATH-001	1/14/2016 12:01:09 PM		4		

Filtering supports the following symbols: <, >, =. Examples of filtering inputs are:

10 - lists test cases matching the specific value of "10" in the selected column
>50 - lists test cases greater than the value of "50" in the selected column
<90 - lists tests cases less than the value of "90" in the selected column
=100 - lists test cases matching the specific value of "100" in the selected column
< - lists test cases with empty values in the selected column
> - lists test cases with non-empty values in the selected column
= - lists test cases with non-empty values in the selected column

Saving and Printing Test Data Summary

Saving the Test Data Summary in .html or .csv format is supported. See "To Save a Script or Text File " on page 82 for more information.

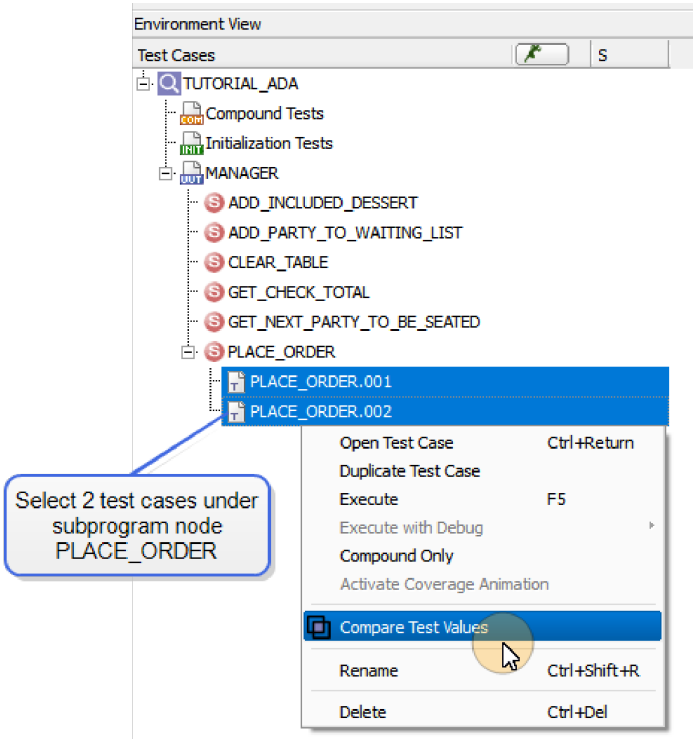
Printing the contents of the Test Data Summary is supported. See "To Print an Open Window" on

page 83 for more information.

View the Test Comparison Report

The Test Comparison Report allows users to compare the test case data of selected test cases within a subprogram. The Test Comparison Report is presented as a filterable table, allowing the user to see similarities and differences in test cases as the data for multiple tests is compared. VectorCAST supports the generation of a CSV export of all test case data.

To open the Test Comparison Report, select two or more test cases under a single subprogram node, and right-click. Select **Compare Test Values** from the right-click context menu.

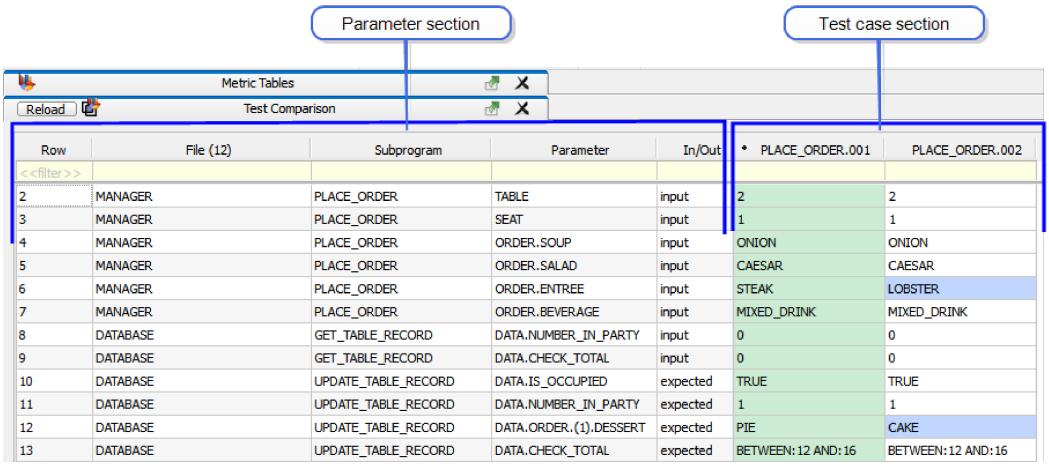


Note: Test cases from different subprograms cannot be compared. You must select at least two test cases from the same subprogram or <<INIT>> node (including specialized test cases and test cases in a TDD environment) to generate the Test Comparison Report.

Selection of (MAP)CSV test cases and <<MIN>>, <<MID>>, and <<MAX>> test cases is not supported.

Viewing the Test Comparison Data

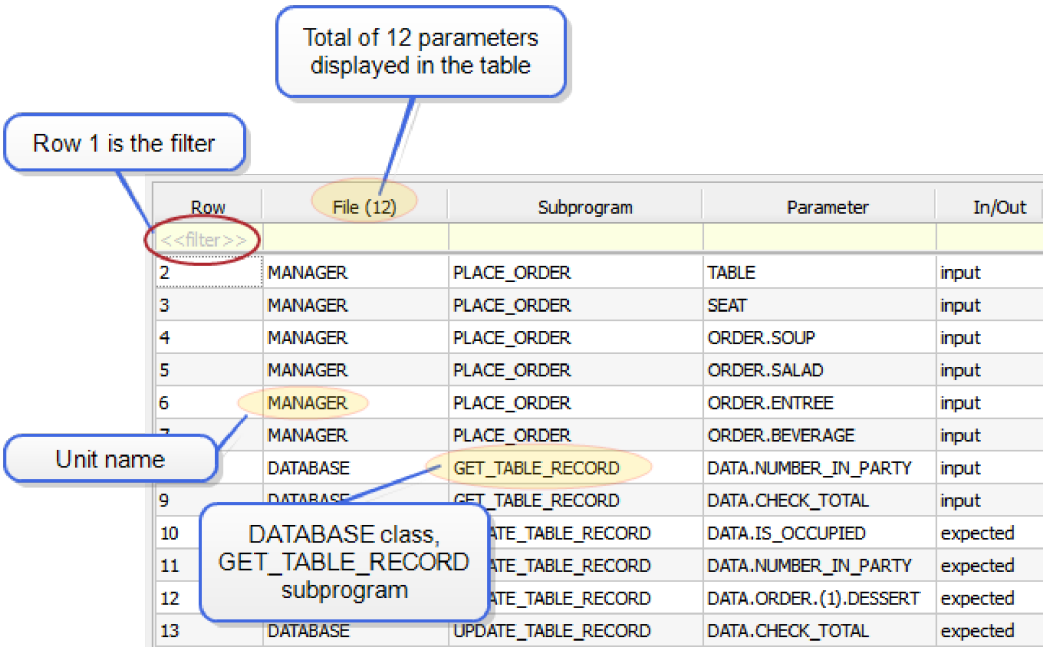
The Test Comparison Report is divided into two sections: the Parameter section and the Test Case section. The report is presented in table format and the user can control the display of the test data through the use of filters and using the Column Controller's show / hide capability.



Row	File (12)	Subprogram	Parameter	In/Out	PLACE_ORDER.001	PLACE_ORDER.002
<<filter>>						
2	MANAGER	PLACE_ORDER	TABLE	input	2	2
3	MANAGER	PLACE_ORDER	SEAT	input	1	1
4	MANAGER	PLACE_ORDER	ORDER.SOUP	input	ONION	ONION
5	MANAGER	PLACE_ORDER	ORDER.SALAD	input	CAESAR	CAESAR
6	MANAGER	PLACE_ORDER	ORDER.ENTREE	input	STEAK	LOBSTER
7	MANAGER	PLACE_ORDER	ORDER.BEVERAGE	input	MIXED_DRINK	MIXED_DRINK
8	DATABASE	GET_TABLE_RECORD	DATA.NUMBER_IN_PARTY	input	0	0
9	DATABASE	GET_TABLE_RECORD	DATA.CHECK_TOTAL	input	0	0
10	DATABASE	UPDATE_TABLE_RECORD	DATA.IS_OCCUPIED	expected	TRUE	TRUE
11	DATABASE	UPDATE_TABLE_RECORD	DATA.NUMBER_IN_PARTY	expected	1	1
12	DATABASE	UPDATE_TABLE_RECORD	DATA.ORDER.(1).DESSERT	expected	PIE	CAKE
13	DATABASE	UPDATE_TABLE_RECORD	DATA.CHECK_TOTAL	expected	BETWEEN:12 AND:16	BETWEEN:12 AND:16

Parameter Section

The Parameter section displays one row for each parameter.



Row	File (12)	Subprogram	Parameter	In/Out
<<filter>>				
2	MANAGER	PLACE_ORDER	TABLE	input
3	MANAGER	PLACE_ORDER	SEAT	input
4	MANAGER	PLACE_ORDER	ORDER.SOUP	input
5	MANAGER	PLACE_ORDER	ORDER.SALAD	input
6	MANAGER	PLACE_ORDER	ORDER.ENTREE	input
7	MANAGER	PLACE_ORDER	ORDER.BEVERAGE	input
8	DATABASE	GET_TABLE_RECORD	DATA.NUMBER_IN_PARTY	input
9	DATABASE	GET_TABLE_RECORD	DATA.CHECK_TOTAL	input
10	DATABASE	UPDATE_TABLE_RECORD	DATA.IS_OCCUPIED	expected
11	DATABASE	UPDATE_TABLE_RECORD	DATA.NUMBER_IN_PARTY	expected
12	DATABASE	UPDATE_TABLE_RECORD	DATA.ORDER.(1).DESSERT	expected
13	DATABASE	UPDATE_TABLE_RECORD	DATA.CHECK_TOTAL	expected

The following data is provided for each parameter:

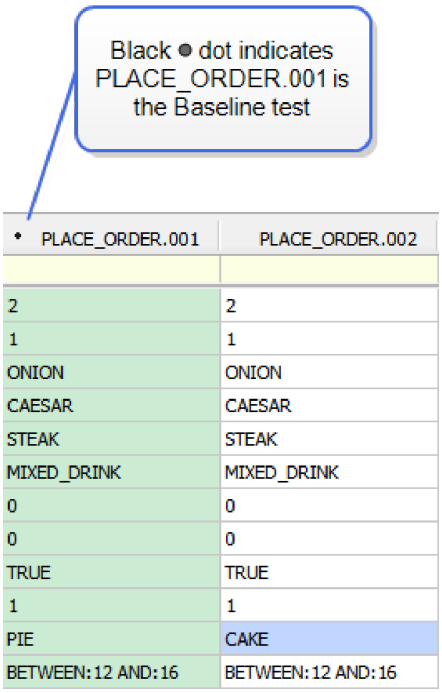
- **Row** - Row number, starting with 2 (the filter occupies row 1 of the table)
- **File** - Unit name or stubbed unit name. For a <<GLOBAL>> parameter, USER_GLOBALS_VCAST is displayed. For <<INIT>> test cases, the unit is displayed as <<INIT>>. Note that in the header for the File column the total number of parameters in the table is displayed in parentheses. For example, File (15).

- **Subprogram** - For constructors and destructors, displayed as <<global>>. For <<SBF>> constructors and subprograms, displayed as class::subprogram. For example, Manager::PlaceOrder.
- **Parameter** - For constructors, the long name is displayed. For an <<SBF>> constructor or subprogram, the parameter name is "stub". For stub-by-implementation (ENVIRO>STUB:database) , the parameter name is "stub".
- **In/Out** - Values are "input" or "expected".

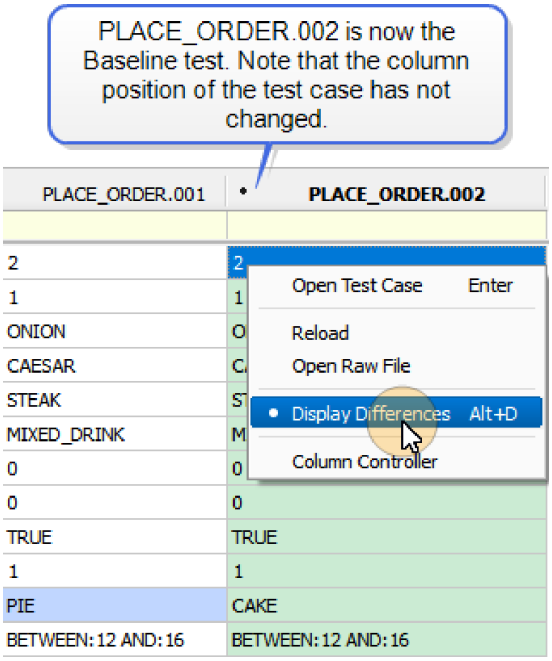
Test Case Section

Setting the Baseline Test

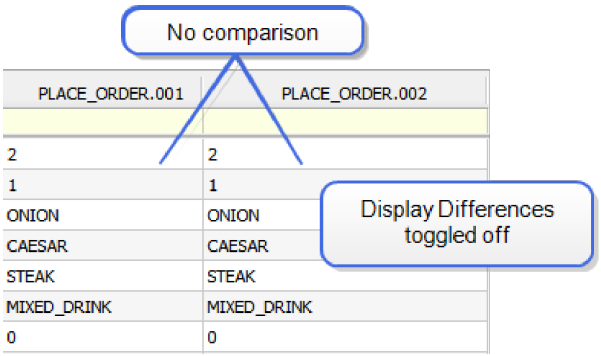
In the Test Case section, one test is designated as the Baseline Test, which the other tests are compared to. By default, the Baseline test is the first test created and is displayed in the first column following the In/Out column in the Parameter section. A black dot in the column header and green cell background indicate the location of the Baseline test in the report.



To change which test case is the Baseline test, right-click in any cell under a different test and select **Display Differences** from the context menu, or press **Alt+D**.

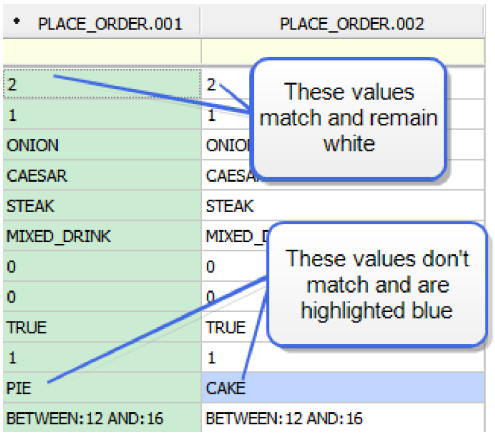


The **Display Differences** setting can be toggled on and off by using the right-click context menu or pressing the **Esc** key. When toggled off all cells have a white background and the table is not doing a comparison, but is just listing the values.



Test Case Comparison

When a test case is compared to the Baseline test, if there are differences in values, the cell background is blue. If there are no differences, the cell background remains white.

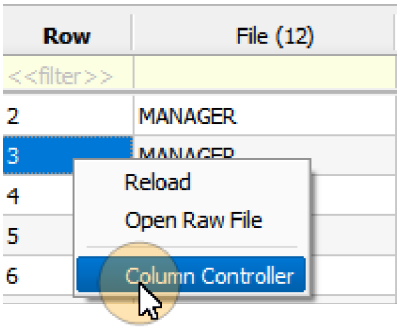


Constructors are displayed as 0. An <<SBF>> constructor or subprogram is displayed as <<STUB>>.

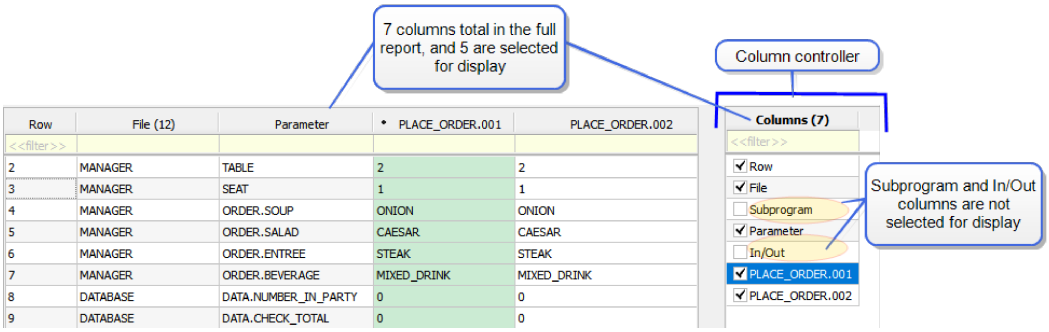
Parameter User Code is displayed as <user>. If both tests have any text in Parameter User Code, they are treated as a match. Note that Test Case User code is skipped.

The Column Controller

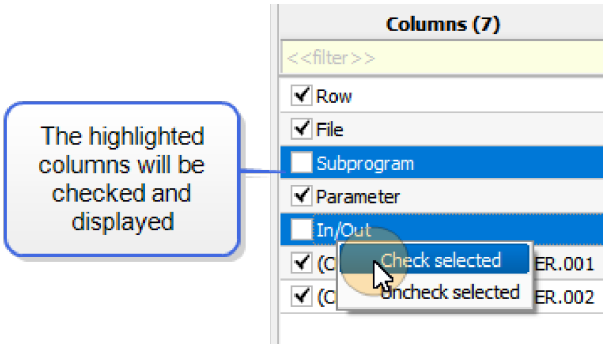
The Column Controller is used to show/hide the columns contained in the Test Comparison Report. To open the Column Controller, right-click on any cell in the Report and select **Column Controller** from the context menu.



The Column Controller panel opens to the right of the Report. Place a checkmark in the boxes of each column to be shown in the Report. Unselected boxes hide the column from view. Hiding columns does not impact the data contained in the Report.



You may also highlight the columns that you want to select or deselect, and right-click to use the context menu to **Check selected** or **Uncheck selected**.



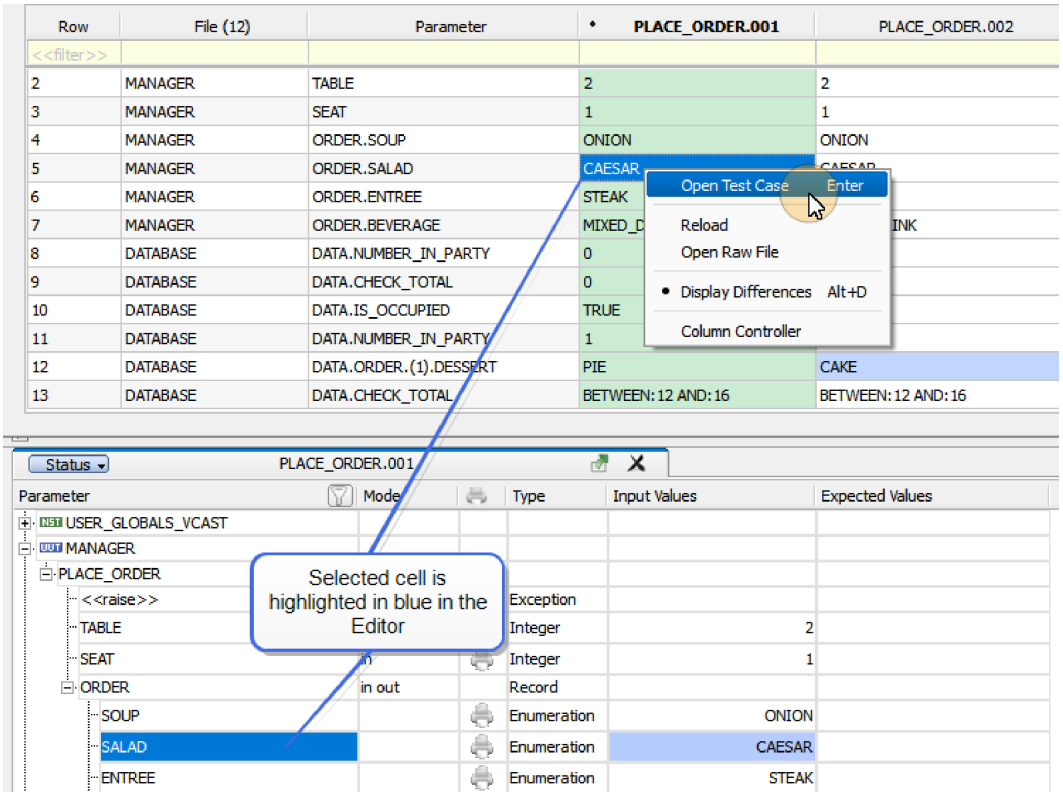
Close the Column Controller panel by right-clicking in any cell in the Report and selecting **Column Controller** from the context menu to toggle the Controller off.

Open a Test Case From the Test Comparison Report

Test cases are accessible from any cell in a test case column. To open the Test Case Editor for a selected test case, perform one of the following:

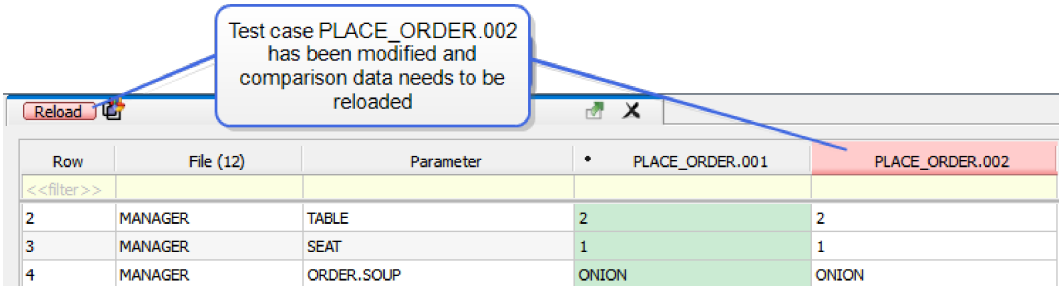
- Right-click on a cell in a test case column and select **Open Test Case** from the context menu. (This will leave keyboard focus in the cell in the Test Case Editor.)
- Highlight a cell in a test case column and select **Enter**. (This will leave keyboard focus in the cell in the Test Case Editor.)
- Highlight a cell in a test case column and select **Alt+Enter**. (This will leave keyboard focus in the cell in the Report, but using the up-arrow and down-arrow keys will move the selection to the associated parameter value in the Test Case Editor.)
- Double-click on a cell in a test case column. (This will leave keyboard focus in the cell in the Test Case Editor.)

The Test Case Editor opens in a horizontal pane under the Report. The cell used to perform the Open action is highlighted in blue in the Editor, making it easy to locate the value to be edited.



Reloading Test Case Data

When data in the Test Case Comparison Report is current, the **Reload** button in the upper left corner of the Report has a gray background. Reload . When test case data is edited and saved, the comparison data becomes obsolete and must be reloaded. The **Reload** button changes to a red background, and the column header for the affected test case also changes to a red background.



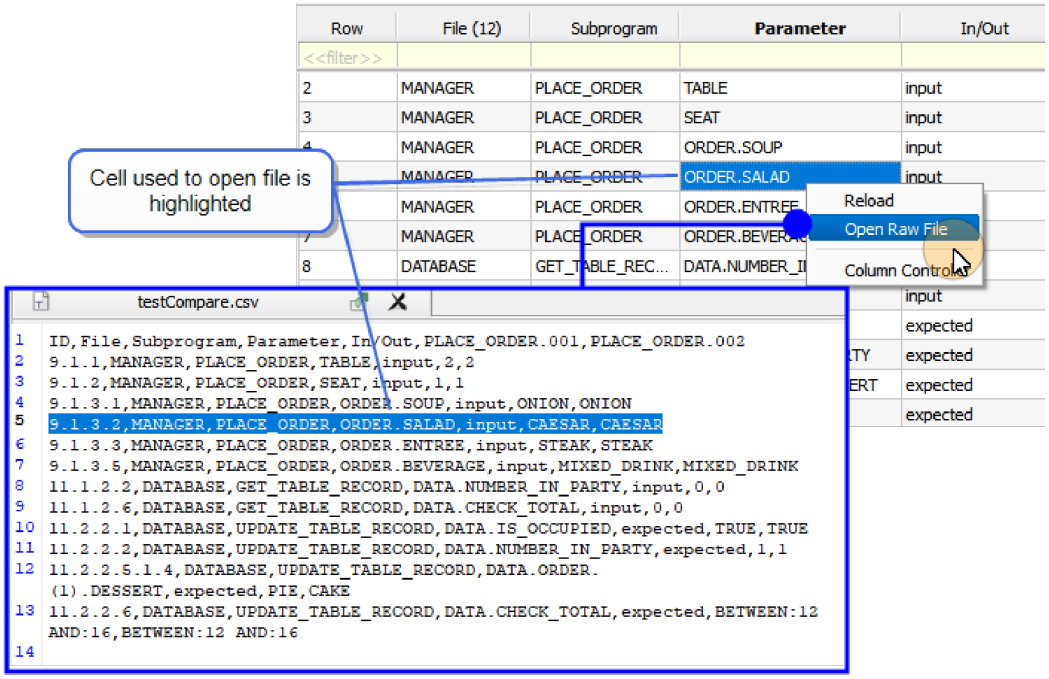
To reload the comparison data, click the **Reload** button. Alternatively, right-click on any cell in the Report and select **Reload** from the context menu. The values for the test case update. Note that reloading may result in a change in comparison, and cell colors (white = matching, blue = difference) may change.

Renaming a test case does not trigger a reload and renamed tests are left out of the reload. Reloading recreates the Report using the original arguments (test case names) and does not

recognize the renamed test case.

Open Raw File

The job execution creates a raw file, named **`testCompare.csv`**, in the ENV directory. To view the raw file, right-click on any cell in the Report and select **Open Raw File** from the context menu. The cell used to perform the Open action is highlighted in blue in the CSV Viewer.



View a Test Case's Raw Data Set

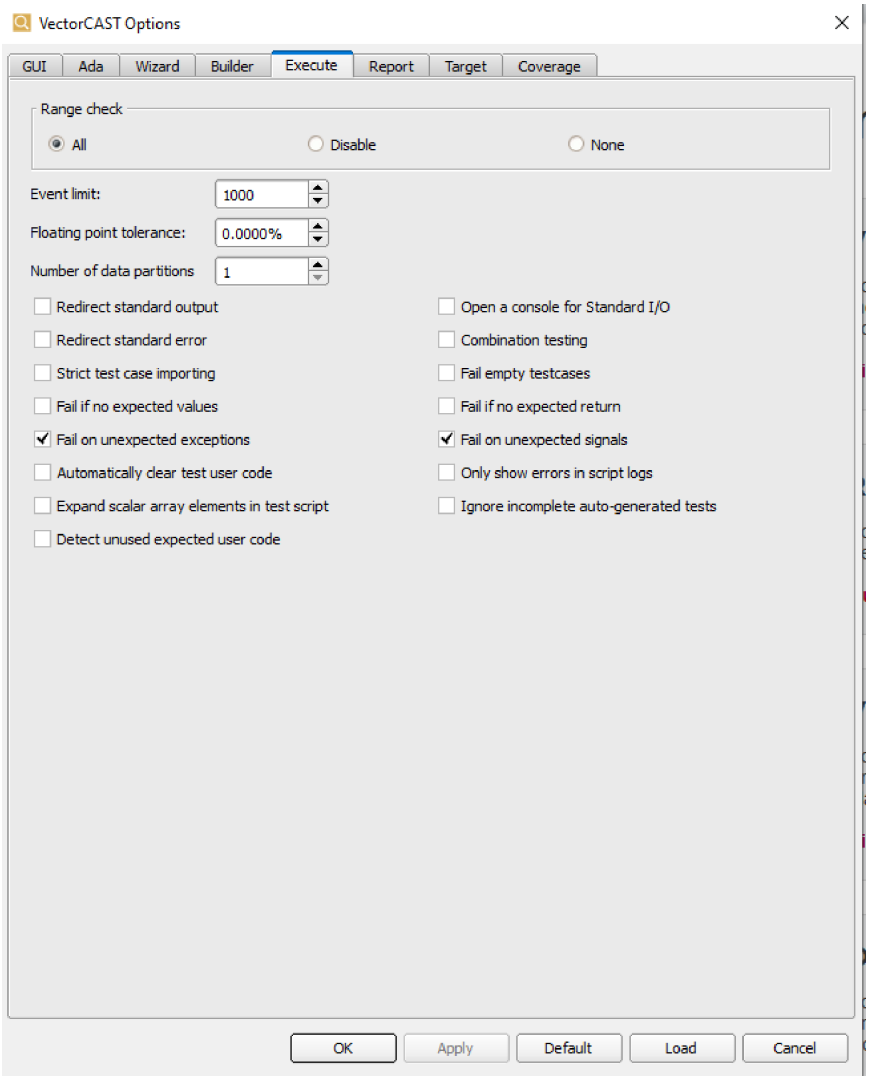
The **Test => View => Raw Data Set** command provides a view of the raw test case data and is used for Technical Support purposes only.

Setting Execution Options

Options: Execute Tab

Choose **Tools => Options** and click the **Execute** tab.

The Execution options tab is used to change the way VectorCAST executes test cases, on an environment-wide scale. To change execution options for an individual test case, use Test Case Options. Pass your cursor over any of the options to see an explanation of that option in a tool-tip.



Range Check

The Range Check option provides the following choices:

- All (default) – determines valid ranges of data types, and enforces input and expected values within valid range
- Disable – determines ranges of data types, but does not enforce
- None – does not determine ranges of data types

VectorCAST provides for full range checking of test data during test cases building. This is reflected in the default range check setting of "All".

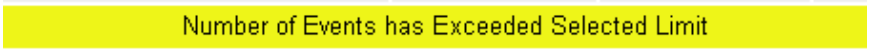
However, when building test cases it is often desirable to be able to specify out-of-range test values for parameters and global data. The Range Check option provides this capability with the “Disable” option. When range checking is disabled, the dialog boxes will still show you the data range that is allowed, but will not enforce that range. This will allow you to enter values that are beyond the data range for the particular parameter or object (including enumerated values).

In order to determine data type ranges, VectorCAST must execute a small program. You can disable this processing by setting the **Range Check option** to **None.** The “None” position is similar to “Disable” with the added restriction that VectorCAST will not attempt to determine scalar ranges. The “None” option is normally used in target testing where execution speed is an issue.

```
clicast -lada option RANGE_CHECK Full | Disable | None

Indicates whether range checking should be performed when accepting input
and expected values. NONE indicates harness should not perform processing
used to verify ranges (useful on slow targets). The default value is Full.
```

Event Limit

The Event Limit option enables you to control the maximum number of events per test harness execution. The event limit defaults to 1000. If this option is set to 0, no results data is captured, although coverage data is captured. If the event limit is exceeded, VectorCAST automatically stops test execution if your compiler supports signals, and the execution report shows:



The Execution Report shows <limit>+1 events.

The event limit applies to each test case. Each time a test executes, it resets the event counter to 1. If you execute several test cases at once using Ctrl+Shift click to select them, or execute them all by selecting the environment name and choosing **Test => Execute**, the counter starts over with each test case (because the test harness is being executed multiple times).

Note: An individual test case can have its own event limit, which overrides the environment event limit when that test case is executed. This option is accessed via the **Options tab** in the **Test Case Editor**.

When executing a compound test case, the event limit is not reset for each slot; the Event Limit setting applies to the Compound Test Case as a whole.

Note: The Event Limit feature is implemented through the raising of signals. If your target does not support signals, the event limit capability will not be available.

```
clicast -lada option EVENT_LIMIT <limit>

Maximum number of events for the harness to process. If this option is set
to 0, no results data is captured, although coverage data is captured. Its
default value is 1000.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.EVENT_LIMIT. Its default value is the environment-wide setting for Event Limit, accessible on the Execute tab of the **Tools => Options** dialog.

Floating Point Tolerance (%)

The Floating Point Tolerance option enables you to set the tolerance of expected results for floating point numbers. This tolerance applies to all test case data in the environment.



Note: There is no tolerance around the value 0.

```
clicast -lada option FLOATING_POINT_TOLERANCE <tolerance>

Percentage that a floating point actual value can vary from its expected
value and still be considered a match. <percentage> is a floating point
number. Its default value is 0.000000.
```

In a test script, the testcase-wide floating point tolerance is specified using:

TEST.FLOATING_POINT_TOLERANCE:<tolerance>



Note: For very small tolerances, the floating point representation of the value may not be exact due to the resolution of floating point numbers; in this case, the floating point tolerance may show more digits than the user expects. For example, a tolerance of 0.000000001 may actually be displayed as 0.00000000099999.

Number of Data Partitions

This option specifies the number of iterations to span an entire range of a scalar parameter. It is used in conjunction with Range Values to allow a specification of iterations instead of using a range delta.

```
clicast -lada Option NUMBER_OF_DATA_PARTITIONS <positive number>

Number of iterations to span an entire range of a scalar parameter. Used
with Range Values to allow specification of iterations instead of range
delta. The default value is 1 partition.
```

Redirect Standard Output

The redirect standard output option enables you to redirect test stdout execution output to a file. Captured text is appended to the end of the Execution Report.

```
clicast -lada option STANDARD_OUTPUT Normal | Redirect

Indicates whether standard output should go to the console or be captured
to a file for inclusion into the execution report. Its default value is
NORMAL (off).
```

Redirect Standard Error

The redirect standard error option enables you to redirect test execution stderr output to a file.

Captured text is appended to the end of the Execution Report.

```
clicast -lada option STANDARD_ERROR Normal | Redirect

Indicates whether standard error should go to the console or be captured to
a file for inclusion into the execution report. Its default value is NORMAL
(off).
```

Strict Test Case Importing

When you import a test script into a VectorCAST environment, it is possible that there will be error in some of the script values. By default, VectorCAST discards the erroneous values and still creates the test case. If you select Strict Test Case Importing, VectorCAST marks the test case with a FAIL status if any error occurred when importing the script.

The status of the test case will be FAIL until you edit and save the test case in the Test Case Editor. If you export the test case to a test script before correcting the problem, “TEST.IMPORT_FAILED” is written to the test script for that test case.

```
clicast -lada option VCAST_STRICT_TEST_CASE_IMPORT true | false

Any errors encountered during test script import cause the test case's
status to be set to Fail. Its default value is FALSE.
```

Fail If No Expected Values

This option identifies test cases that have no Expected Values. When set to True, a test is prohibited from executing if it does not contain at least one Expected Value, Expected Parameter User Code, or Test Case User Code. It is marked as Failed in the Test Case Tree, and appears as "Abnormal Termination - No expected values" in the Testcase Management Report.

```
clicast -lada option VCAST_TESTCASE_FAIL_ON_NO_EXPECTED true | false

If this option is set, then test cases that have no expected results or
expected user code will be marked as failed. Its default value is FALSE.
```

Fail on Unexpected Exceptions

When checked, this option causes a test case to be marked as failed if an exception is thrown or signal is raised that was not expected.

```
clicast -lada option VCAST_UNEXPECTED_EXCEPTIONS_FAIL true | false

Sets test case status to Fail if an exception is raised during test case
execution but no exception was expected. Its default value is TRUE.
```

Automatically Clear Test User Code

If selected, this option tells VectorCAST to clear test user code prior to executing tests. Parameter

user code for the user globals unit is automatically cleared as well. This option is helpful when you need to keep the executable size small.

```
clicast -lada option VCAST_AUTO_CLEAR_TEST_USER_CODE true | false

When set to TRUE, user code is cleared prior to executing tests. The
default value is False.
```

Expand Scalar Array Elements in Test Script

By default, when VectorCAST writes scalar array elements to a test script, it combines script lines when the values are identical. This option turns off this "collapse" capability.

```
clicast -lada option VCAST_SCRIPT_EXPAND_ARRAYS true | false

By default, when an array of scalars is written to a test script, identical
values are condensed to one script line. By setting this option, each array
element will get its own script line. The default value is False.
```

Detect Unused Expected User Code

When enabled, this option detects test-specific expected user code that has not been executed during a test. Such code is most likely to be associated with a stub that has not been called. When True and some testcase or parameter user code has not been executed, the test fails, and the Execution Report displays the unused values, the unit, and whether that unit is stubbed or not-stubbed.

Disable this option if you have added parameter user code to a stub that you expect to never get called. (You might add code to an uncalled stub if you want the test to show a failure if the stub ever gets called.)

```
clicast -lada option VCAST_DETECT_UNUSED_EXPECTED_UC true | false

Set this option to True to detect when any test-specific user code that
checks expected values with the "{{...}}" syntax has not been executed. In
that case, the test will be flagged as having unused expected results. Such
code is most likely to be associated with a stub that has not been called.
Any test case or parameter user code, either Input or Expected, that uses
the conditional syntax "{{...}}" is considered when the option is True. Set
this option to False if you have added parameter user code to a stub that
you expect to never get called. (You might add code to an uncalled stub if
you want the test to show a failure if the stub ever gets called. The
default value is False.
```

Open a Console for Standard I/O

If your program under test reads from stdin during test execution, then the test will hang if this stdin is not provided. To overcome this problem on Windows, you can set the VectorCAST option “Open a console for Standard I/O.” If this option is set, a DOS Command Prompt window (console) is opened before test case execution and you can use this window to type in stdin data.

If the option “Redirect Standard Output” is on as well, then standard output continues to be directed to the execution reports, and standard input is ignored. Therefore, turn off “Redirect Standard Output” when using this option. Under Linux, this option is dimmed, as harness standard input and standard output are by default available in the console window from which VectorCAST was started.

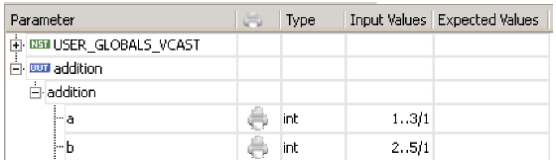
```
clicast -lada option VCAST_SHOW_STDOUT_CONSOLE true | false

Opens a console during test harness execution to facilitate standard input
and output. On Windows, if either of the execute options “Redirect standard
output” or “Redirect standard error” is set, then standard output and error
continue to be redirected to the Execution reports, but standard input is
ignored. The default value is False.
```

Combination Testing

If a test case has a parameter with an input range or list, then the test harness executes the test case once for each iteration. When two or more parameters have a range iteration, then each parameter’s iteration count is incremented at the same time, by default.

Combination testing can be turned on for an individual test case, using Test Case Options, or for the whole environment, using the **Tools => Options** dialog, **Execute tab**. If combination testing is enabled, VectorCAST will determine the combinations of input values and use those values as stimulus values.

For example, if parameter A has a range 1 to 3 with a delta of 1 and parameter B has a range 2 to 5 with a delta of 1, and Combination Testing is off, then the test case executes this way:



Range Iteration #1:

A is 1
B is 2

Range Iteration #2:

A is 2
B is 3

Range Iteration #3:

A is 3
B is 4

Range Iteration #4:

A is 3 again
B is 5

If the combination testing option is enabled, then the test case executes this way:

Range Iteration #1:

A is 1
B is 2
Range Iteration #2:
A is 1
B is 3
Range Iteration #3:
A is 1
B is 4
Range Iteration #4:
A is 1
B is 5
Range Iteration #5:
A is 2
B is 2
Range Iteration #6:
A is 2
B is 3
Range Iteration #7:
A is 2
B is 4
Range Iteration #8:
A is 2
B is 5
Range Iteration #9:
A is 3
B is 2
Range Iteration #10:
A is 3
B is 3
Range Iteration #11:
A is 3
B is 4
Range Iteration #12:
A is 3
B is 5

This feature is not available with an environment built with VectorCAST version 3.2 or earlier. Rebuilding the environment will enable this feature's functionality.

```
clicast -lada option VCAST_DO_COMBINATION true | false

Use all combinations of the values in the range expressions for test
stimulus values. The default value is False.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.DO_COMBINATION. Its default value is the environment-wide setting for Combination testing, accessible on the Execute tab of the **Tools => Options** dialog.

Fail Empty Testcases

If this option is set, then empty testcases will be marked as failed.

```
clicast -lada Option VCAST_EMPTY_TESTCASE_FAIL True | False

If this option is set, then empty testcases will be marked as failed.
```

Fail If No Expected Return

If this option is set, testcases that have no expected return value or expected user code for the function's return value are marked as failed.

Note that while similar to the option VCAST_TESTCASE_FAIL_ON_NO_EXPECTED, which marks any test as failed if it has no Expected Value or Expected User code, this option is specific to the return value of the subprogram under test only.

```
clicast -lada Option VCAST_TESTCASE_FAIL_ON_NO_EXP_RETURN True | False

If this option is set, then then testcases that have no expected return
value or expected user code for the return value will be marked as failed.
```

Fail on Unexpected Signals

When this option is set and a signal is raised during the execution of a test case, then the test case status is set to FAIL.

```
clicast -lada option VCAST_UNEXPECTED_SIGNALS_FAIL true | false

Sets test case status to Fail if a signal is raised during test case
execution. Its default value is TRUE.
```

Only Show Errors In Script Logs

When checked, this option limits the contents of test script logs to error messages. When loading a large test script, setting this option will make it much easier to find errors that occurred during test script importing.

```
clicast -lada option VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS true | false

Limits script logs to show errors only. The default value is False.
```

Ignore Incomplete Auto-Generated Tests

When VectorCAST generates automatic test cases (such as Basis Path tests), it labels the test as "partial" if it cannot specify all of the data for the test, and "template" if it cannot specify any of the data for the test.

This option tells VectorCAST to discard those partial or template tests when importing the generated test script.

```
clicast -lada option VCAST_IGNORE_INCOMPLETE_TESTS true | false

When building Basis Path or MCDC test cases, there are three outcomes for
each test: Complete, Partial, and Template. When this option is on,
VectorCAST will discard the Partial and Template tests, and only load the
Complete tests. The default value is False.
```

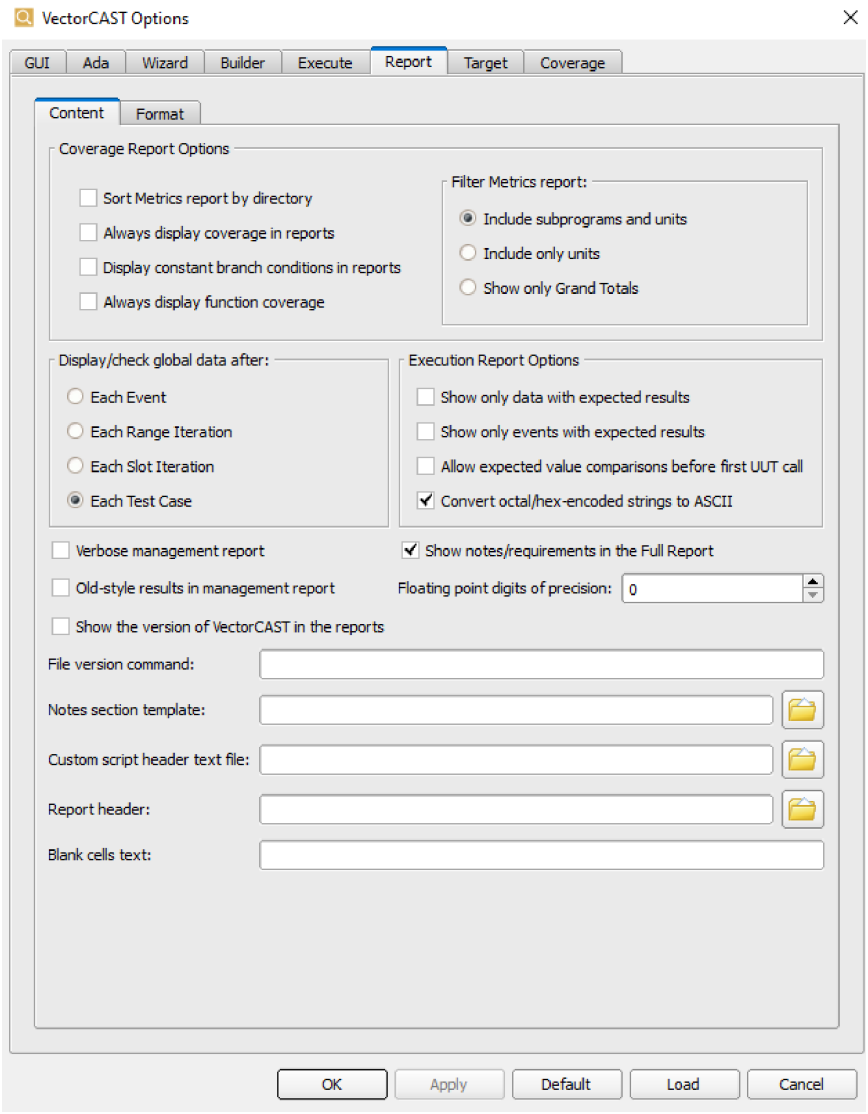
Setting Report Options

As with the Coverage options, the Report options are saved in the CCAST_.CFG file for VectorCAST Cover environments.

Report Content Options

Choose **Tools => Options** and click the **Report** tab, then click the **Content** sub-tab.

The **Content** sub-tab is used to change the way VectorCAST displays test execution reports and coverage reports on an environment-wide scale. Pass your cursor over any of the options to see an explanation of that option in a tool-tip.



See also "To Set VectorCAST Options" on page 242.

Coverage Report Options

Sort Metrics Report by Directory

By default, the Metrics report lists all the units in the environment alphabetically. Setting this option causes the report to have one section per directory, sorted alphabetically. For each section, the full search directory path and the units from that directory are listed.

```
clicast -lada Option VCAST_SORT_METRICS_RPT_BY_DIR <True | False>

Set to true if you want the Metrics report to be sorted by search directory
rather than by unit alphabetically (default). The default value is False.
```

Always Display Coverage in Reports

By default, the Aggregate Coverage Report includes all units in the environment and if they have no coverage data, “No coverage data exists” is written for that unit. Turning on this option causes the annotated source code for all units to be included in the Aggregate Coverage Report, even if a unit has no coverage data.

```
clicast -lada option VCAST_DISPLAY_EMPTY_COVERAGE <True | False>

Display annotated source code in reports even if no run-time coverage data
exists. If this option is disabled, the message "No Coverage Data Exists"
will be displayed instead. The default value is False.
```

Always Display Function Coverage

Extra column that reports if a function was entered.

A subprogram is considered covered in Function coverage if it is entered during test execution. Selecting this option will display a "Function Coverage" column in each report instance that includes the Metrics report (Full Report, Test Case Management Report and Aggregate Coverage Report).



Note: The configuration option VCAST_DISPLAY_FUNCTION_COVERAGE has no effect when used with the Function coverage type.

```
clicast -lada Option VCAST_DISPLAY_FUNCTION_COVERAGE <True | False>

Extra column that reports if a function was entered. This option is only
used if the coverage type is not Function. The default value is False.
```

Filter Metrics Report

This option enables you to reduce the detail that is displayed in the Metrics report. By default the option is set to "Include subprograms and units," (No_Filtering) which generates the familiar Metrics report with all units and subprograms included.

Include Subprograms and units (NO_FILTERING)

Metrics Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	27 MAR 2020
Time of Report Creation	2:18:26 PM

Metrics

Statement

Unit	Subprogram	Complexity	Statements
MANAGER	PLACE_ORDER	5	8 / 12 (66%)
	ADD_INCLUDED_DESSERT	3	2 / 3 (66%)
	CLEAR_TABLE	2	0 / 7 (0%)
	GET_CHECK_TOTAL	1	0 / 2 (0%)
	ADD_PARTY_TO_WAITING_LIST	2	0 / 5 (0%)
	GET_NEXT_PARTY_TO_BE_SEATED	6	0 / 12 (0%)
TOTALS	**6**	**19**	**10 / 41 (24%)**
DATABASE	GET_TABLE_RECORD	1	1 / 1 (100%)
	UPDATE_TABLE_RECORD	1	1 / 1 (100%)
TOTALS	**2**	**2**	**2 / 2 (100%)**
GRAND TOTALS	**8**	**21**	**12 / 43 (27%)**

The other choices for filtering reduce the amount of detail in the report. Choosing "Include only units" (Subprogram_Detail) causes the Metrics report to show only the unit names and their totals for the number of subprograms, complexity, and coverage.

Include only units (SUBPROGRAM_DETAIL)

Metrics Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	27 MAR 2020
Time of Report Creation	2:21:05 PM

Metrics

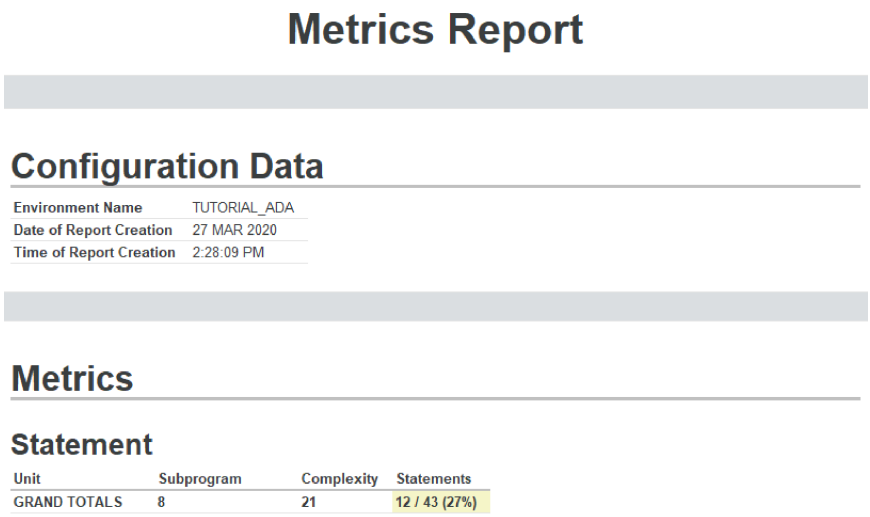
Statement

Unit	Subprogram	Complexity	Statements
MANAGER	**6**	**19**	**10 / 41 (24%)**
DATABASE	**2**	**2**	**2 / 2 (100%)**
GRAND TOTALS	**8**	**21**	**12 / 43 (27%)**

Choosing "Show only Grand Totals" (Unit_Detail) causes the Metrics report to show only the very last

line, the GRAND TOTALS for each column.

Show only Grand Totals (UNIT_DETAIL)



Note: If the option "Sort Metrics report by directory" is also set, then the "Show only Grand Totals" setting causes the Metrics report to have a single GRAND TOTAL line for each directory.

```
clicast -lada Option METRICS_TOTAL_LEVEL_DISPLAY No_Filtering | Subprogram_
Detail | Unit_Detail

Specify which result to filter out of the Metrics Report. The default value
is No_Filtering.
```

Display/Check Global Data Report Options

Display/Check Global Data After

The Display Global Data After feature gives you control over when global data objects are displayed in the Test Execution report. If an expected value for the global object is set, then that expected value is compared to its actual value at the same time.

Ranging from displaying more frequently to less frequently, you can choose to display global data after:

- Each event
- Each range iteration
- Each slot iteration
- Each test case (default)

Each Event displays the current value(s) of global data objects at every transfer of control flow.

Each range iteration displays the current value(s) of global data objects after every iteration caused

by a parameter's range of input values. If no parameter has an iteration, then the globals are displayed at the end of the test execution.

Each slot iteration displays the current value(s) at the end of each iteration of a test case in a compound test case. A compound can have one or more test cases in it, each with one or more iterations. This setting causes the globals to be displayed (and compared) when each test case finishes an iteration. If you apply this setting to a test case and then execute that test case alone (i.e. not in a compound), then the globals are displayed at the end of test execution, because it is the only "slot." In the example below, the global values would be displayed twice, at the end of each slot iteration.

Slot	Units	Subprograms	Test Cases	Iterations
1	manager	Get_Check_Total	Get_Check_Total.001	2

The default **Each test case** displays the current value(s) of global data at the end of a slot in a compound test case. A compound can have one or more test cases in it; this setting causes the globals to be displayed and compared when each test case in a compound finishes executing all of its iterations. If you apply this setting to a test case and then execute that test case alone (i.e. not in a compound), then the globals are displayed at the end of test execution, because it is the only "slot."

```
clicast -lada option GLOBALS_DISPLAY <Each_event|Range_Iteration|Slot_
Iteration|Testcase>

When to display and check global values in execution results: at each
event; at the end of each range iteration; at the end of each slot
iteration; at the end of each slot. Its default value is Testcase.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.GLOBAL_DATA_DISPLAY. Its default setting is the environment-wide setting for the Display global data option, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.

Example of Display Global Data

Suppose you insert a test case for Get_Check_Total, from the tutorial environment. It is named GET_CHECK_TOTAL.001. Get_Check_Total only calls the stubbed function Get_Table_Record. To set up the example, enter the following Input and Expected values:

	Input	Expected
VCAST_USER_GLOBALS		
<<GLOBALS>>		
VECTORCAST_INT1	44	55
Get_Check_Total		
Table	**2..3/1**	

Execute the test case GET_CHECK_TOTAL.001. In the execution results, you see 6 events, three for

the iteration in which Table is 2, and three for the iteration in which Table is 3. Because Each Test Case is the default setting, the global data value is displayed following Event 6, at the end of the test case.

The following table shows when the global data is displayed based on the setting of the “Display Global Data After” option. Global data is never compared to its expected value at the start of an iteration.

	Global Data Displayed After:			
	Each Event	**Each Range Iteration**	**Each Slot Iteration**	**Each Test Case**
Event 1: call UUT Range Iteration 1 (Table is 2)	Yes	No	No	No
Event 2: call stub	Yes	No	No	No
Event 3: return from UUT	Yes	Yes	No	No
Event 4: call UUT Range Iteration 2 (Table is 3)	Yes	No	No	No
Event 5: call stub	Yes	No	No	No
Event 6: return from UUT	Yes	Yes	Yes	Yes

Execution Report Options

Show Only Data with Expected Results

In some cases, a test may contain a huge number of global data input, but you may not want to capture all of this data in the Execution Reports. In the test case execution report, global and parameter data will only be displayed in the test execution report if expected values are provided; globals and parameters with input values only are not displayed.

```
clicast -lada option VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES True | False

When this option is set, global and parameter data will only be displayed
in the test execution report if expected values are provided. This option
will result in smaller execution reports and faster test execution in cases
where there is a lot of global and / or parameter data input. The default
value is False.
```

```
TEST.VALUE:<<OPTIONS>>.SHOW_ONLY_DATA_WITH_EXPECTED_RESULTS:TRUE | FALSE

If specified in a test case script, this option causes only events that
have Expected values to be included in the Execution Report. It overrides
the environment-wide setting for “Show only events with expected results.”
If not specified in a test script, the default value is the environment-
```

wide setting, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.

Show Only Events with Expected Results

Setting this option causes VectorCAST to leave out any events in the Execution Report that do not have expected data for any parameter or global value. This option will result in smaller execution reports in cases where there is a lot of global and /or parameter data input.

clicast -lada option VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES True | False

When this option is set, events will only be displayed in the test execution report if expected values exist for that event. This option will result in smaller execution reports in cases where there is a lot of global and / or parameter data input. The default value is False.

TEST.VALUE:<<OPTIONS>>.SHOW_ONLY_EVENTS_WITH_EXPECTED_RESULTS:TRUE | FALSE

If specified in a test case script, this option causes only events that have Expected values to be included in the Execution Report. It overrides the environment-wide setting for "Show only events with expected results." If not specified in a test script, the default value is the environment-wide setting, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.

Allow Expected Value Comparisons Before First UUT Call

This option is used to cause VectorCAST to make comparisons of expected values to actual values before the UUT is even called. It is useful if a stub is called before you get to the UUT call (from user code in an <<INIT>> test case, for example), and you want to verify that a value was passed in correctly. By default, this option is false. Set it to true if you want to compare expected values before the UUT call.

clicast -lada option VCAST_EXPECTED_BEFORE_UUT_CALL True | False

Setting this option enables comparisons between expected values and actual values to occur before the call to the UUT. The default value is False.

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.COMPARE_BEFORE_UUT. Its default value is the environment-wide setting for Compare Expected Values Before UUT Call, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.

Convert Octal / Hex-Encoded Strings to ASCII

By default, strings with unprintable characters are displayed in octal in the Execution report. To display these strings in hex, select **Tools => Options = Execute tab** and set the option "**Hex notation for unprintable chars**."

Setting the option, “**Convert octal/hex-encoded strings to ASCII**” under Execution Report Options causes the printable portions of these strings to be displayed in ASCII in the Execution report, while the unprintable characters remain in octal or hex.

```
clicast -lada option VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII True | False.

This option is used to determine whether or not to convert "unprintable"
strings encoded in octal or hexidecimal notation to ASCII encoding in the
report output. It doesn not affect the actual string comparisons in the
test execution. Its default value is TRUE.
```

Report Options

Verbose Management Report

When True, this option copies the contents of the 'Test Notes' box in the Test Case Editor to the Test Case Management Report. One possible use for this is for test requirement numbers to be correlated to test cases in the Test Case Management Report.

```
clicast -lada option VCAST_VERBOSE_MANAGEMENT_REPORT True | False

Copies the contents of the 'Test Notes' box in the Test Case Editor to the
Test Case Management Report. Its default value is
```

Old-style Results in Management Report

When this option is set to True, the Test Case Management Report uses the "old-style" pass/fail results in the main table of the report. The Overall Results section does not change.

The "old-style" Test Case Management Report combined the Expected Values matched with the Control Flow results, signals, exceptions, and terminations in the Pass/Fail column. Enable this option to once again generate the Test Case Management Report with the data displayed in this way.

```
clicast -lada option VCAST_OLD_STYLE_MANAGEMENT_REPORT <True | False>

When set, this option reverts the Test Case Management Report to the
behavior found in VectorCAST version 6.0 and prior, in which the main body
of the report combined the Expecteds and Control Flow in one fraction in
the "Pass/Fail" column. The default value is False.
```

Show the Version of VectorCAST in the Reports

When this option is set, the version and build date of VectorCAST is added to the Configuration section of the following reports:

- Test Case Management report
- Aggregate Coverage report
- Metrics report

- Test Case Data report
- Execution Results report

```
clicast -lada option VCAST_RPTS_SHOW_VERSION True | False

Show the VectorCAST version in the configuration section of the reports.
The default value is False
```

Show Notes / Requirements in the Full Report

By default, the Notes / Requirements section is included in the Full Report. When cleared, the option "Show notes / requirements in the Full Report" excludes the Notes / Requirements in the Full Report. Note that the section is always included in the Test Case Data Report.

```
clicast -lada  Option SHOW_NOTES_IN_FULL_REPORT True | False

This option will add a section to the Full Report listing test notes and
requirements. The default value is True.
```

Floating Point Digits of Precision

This value controls exactly how the decimal part of a floating point number will be displayed. It modifies the amount of precision that VectorCAST will use to print a floating point number.

```
clicast-lada option VCAST_FLOAT_PRECISION <integer number>

Controls the number of digits used when the test harness prints floating
point Actual Values for the Execution Report or Range Data. This option's
setting is significant in the comparison of Expected Values for floating
point numbers as well as determining the min and max values.

In an Ada environment, floating point values are printed in scientific
notation and the precision value controls the number of digits to the right
of the decimal, with a single digit to the left of the decimal. A precision
of 0 results in the maximum avaliable digits being printed.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.FLOAT_POINT_DIGITS_OF_PRECISION. Its default value is the environment-wide setting for Floating Point Digits of Precision, accessible on the **Reports** tab of the **Tools => Options** dialog.

File Version Command

If a command is supplied, the command is executed on each UUT source file and any non-stubbed units. The results of the command are included in the Test Case Management Report and the Full Test Report.

For example, setting this option to **`ls -l`** (on Linux) causes the following information to be added to the two reports:

```
FILE VERSION REPORT
  Unit Name: File Name and Version Information (ls -l)
  manager : -rwxr-xr-x   1 mdm     None        2012 May 18 14:31
C:\Cygwin\home\mdm\cvs\sqa_testing\SOURCE\C\TUTORIAL\manager.c
```

The analogous command on Windows is: **`dir /b`**.

On Linux, VectorCAST executes the following command:

```
<supplied command> <full path to each UUT>
```

On Windows, VectorCAST executes the following command:

```
cmd /c "<supplied command> <full path to each UUT>"
```

The full path to each UUT is derived from the Source directories used in the environment.

`clicast -lada option VCAST_FILE_VERSION_COMMAND <command>`

Command executed for each source file in the test environment, and a section added to the Test Case Management and Full reports listing the name of each unit in the test environment and the information generated by this command. If this command is empty, then the File Version section fo the report will not be created. <command> is a quoted string. It has no default value.

Notes Section Template

In the Test Case Notes tab, Parameter User Code, and Test Case User Code, you can import a text template. This feature is useful if you want to document the requirements tested by each test case following a standard format. Before importing the template, first specify the template file. To do this, choose **Tools => Options** dialog, **Report** tab, **Content** sub-tab. Enter the full path to the template file for the option "Notes section template."

To import the template at the cursor location in a User Code editor, first click the checkbox next to Enable. Right-click in the blank text area and choose **Import template**. The text from the template file appears in the text area.

To use the template in test case Notes, first specify the template file. If the Notes tab of a test case is empty and a template file is specified, then the template is automatically loaded when the test case is opened. You can also right-click in the Notes tab and choose Import template. The text from the template file appears in the Notes tab at the cursor location.

`clicast -lada option VCAST_NOTES_SECTION_TEMPLATE <Path to Template text file>`

This option allows you to enter a path to a text file that contains a template for the Notes section of the test case.

Custom Script Header Text File

This option specifies a text file containing text you want VectorCAST to place at the top of exported test scripts. Use the full path to the file, or a path relative to the environment directory (not working directory). In the exported test script, the text is preceded with a comment marker (--) automatically, in order to retain the custom text when the environment is rebuilt or updated.

```
clicast -lada option VCAST_DEFAULT_SCRIPT_HEADER <Path to Template text
file>

This option allows you to enter a path to a text file that contains a
custom header for all generated test scripts.
```

Report Header

This option prepends a text string to the title or adds a new section to the Full Report, just below the title, to include the contents of a file in TEXT or HTML format. Input can be a text string, or a file containing several lines.

If an HTML file is provided, the contents should provide a ** ** section with a table. For example:

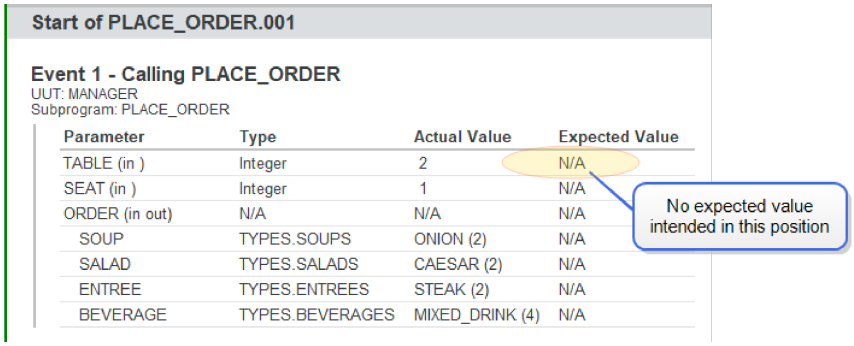
```
<div class='report-block'>
  <h2>Additional Data</h2>
  <table class='table table-small'>
    <tr><th>Tests created by</th><td>Fred Williams</td></tr>
    <tr><th>Source revision</th><td>123abc789</td></tr>
  </table>
 </div>
```

```
clicast -lada option VCAST_RPTS_HEADER <text to prepend to title> | <header
text> | <html file>

Prepend a text string to the title or add a new section to the Full Report,
just below the title, to include the contents of a file in TEXT or HTML
format. Input can be a text string, or a file containing several lines. If
an HTML file is provided, the contents should provide a <div> section with
a table.
```

Blank Cells Text

This option directs VectorCAST to include a text string in the Execution Report and Test Case Data Report, as well as those sections in the Full Report, in place of empty cells in the Input and Expected Value tables.

For example, use the text "N/A" for blank cells in reports to indicate that no Input Value or Expected Value is intended in this position.

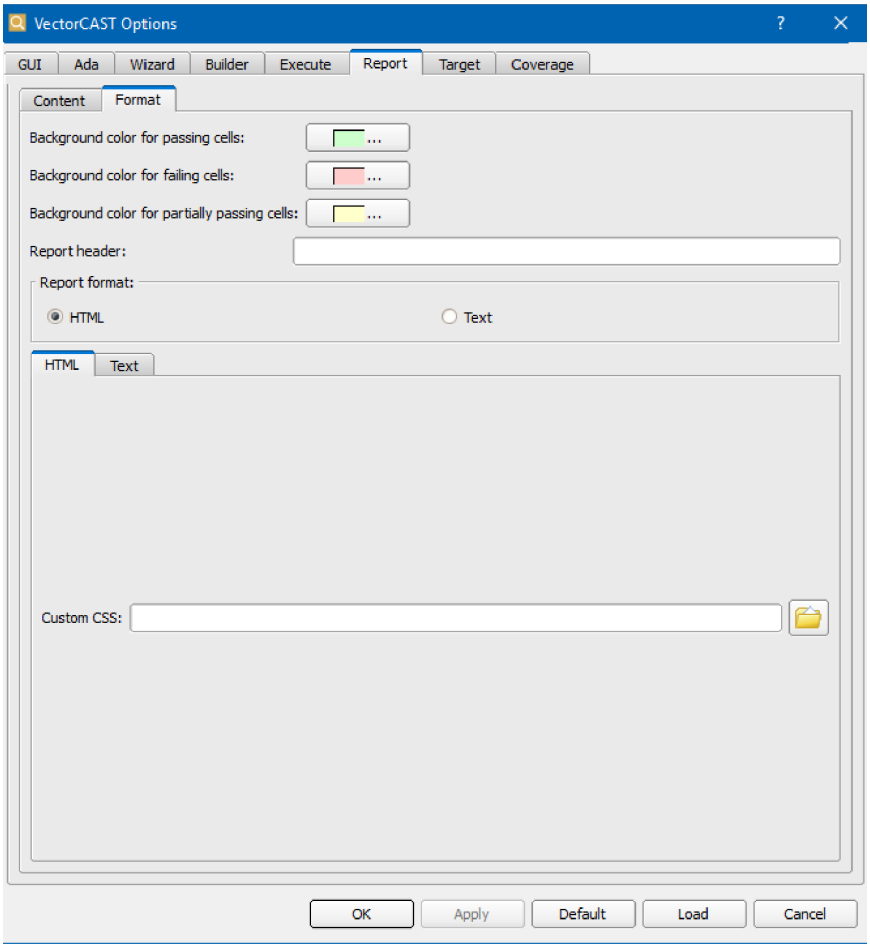



```
clicast -lada option VCAST_RPTS_EMPTY_DATA_STRING <string>

Text to show in blank cells in the Execution and Test Case Data report
sections. The default value is unset.
```

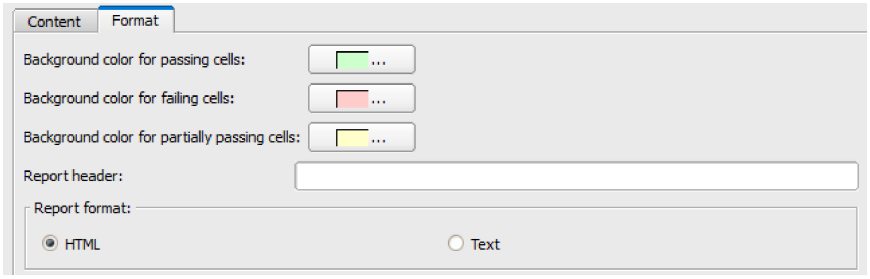
Report Format Options

Choose **Tools => Options** and click the **Report** tab , then click the **Format** sub-tab.

The **Format** sub-tab is used to change the formatting options for HTML and text reports. Pass your cursor over any of the options to see an explanation of that option in a tool-tip.



Setting Format Options



Background Color for Passing Cells

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab. This background color is used for passing data cells in the Parameter Tree.

```
clicast VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR <color>

Background color used for passing data cells in tables in the UI. The
default color is #ccffcc, a light green.
```

Background Color for Failing Cells

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab. This background color is used for failing data cells in the Parameter Tree.

```
clicast option VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR <color>

Background color used for failing data cells in tables in the UI. The
default color is #ffffcc, a light pink.
```

Background Color for Partially Passing Cells

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab. This background color is used for partially passing data cells in the Parameter Tree.

```
clicast option VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR <color>

Background color used for partially passing data cells in tables in the UI.
The default color is #ffffcc, a light yellow.
```

Report Format

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab.

The Report Format option determines if reports are displayed in HTML or Text. The default setting is HTML. If you choose HTML, you can view the reports within VectorCAST or in an external browser. If you choose text, you can view them within VectorCAST or in an external text editor.

Note that a test does not need to be re-executed in order to see the Execution report in a different format.

```
clicast -lc option VCAST_CUSTOM_REPORT_FORMAT HTML | TEXT

Output format for VectorCAST reports: HTML or TEXT. The default value is
HTML.
```

Setting Custom CSS

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab, and the **HTML** sub-tab.

To apply a custom CSS file to use in the HTML reports, enter the path to the custom CSS file.Your custom style sheet is appended when reports are generated and the contents of the custom style sheet are embedded in the **<style>** section following the default styles, when the reports are

generated.

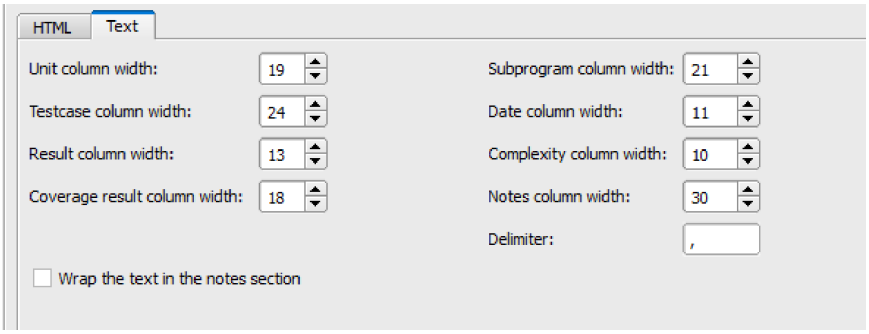
If the option is not set or the file cannot be found, the default CSS is used. The default CSS is located at **$VECTORCAST_DIR/python/vector/apps/ReportBuilder/css**.

```
clicast -lada option VCAST_RPTS_CUSTOM_CSS <path to custom.css file

Reports use the default cascading style sheets (*.css) located in
$VECTORCAST_DIR/python/vector/apps/ReportBuilder/css/, unless a custom
style sheet is specified in this option. Environment variables in the path
to the custom style sheet are supported, using this syntax $(ENV_VAR). When
used, the contents of the custom style sheet are embedded in each report,
between the <style>...</style> tags in the header section after the default
styles allowing for CSS style overriding.
```

Text Report Options

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab, and the **Text** sub-tab.

The **Text** sub-tab has options that are honored when the Report Format is Text.



Unit Column Width

The Unit Column Width option specifies the width of the “Units” column in Text reports.

The default Unit column width is 19 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_UNIT_COLUMN_WIDTH <width>

Width (in characters) of unit column in text reports. Its default value is
19.
```

Subprogram Column Width

The Subprogram Column Width option specifies the width of the “Subprogram” column in Text reports.

The default Subprogram column width is 21 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH <width>

Width (in characters) of subprogram columns in text reports. Its default
value is 21.
```

Testcase Column Width

The Testcase Column Width option specifies the width of the “Test Case” column in Text reports.

The default Testcase column width is 24 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_TESTCASE_COLUMN_WIDTH <width>

Width (in characters) of testcase column in text reports. Its default value
is 24.
```

Date Column Width

The Date Column Width option specifies the width of the “Date” column in Text reports.

The default Date column width is 11 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_DATE_COLUMN_WIDTH <width>

Width (in characters) of date column in text reports. Its default value is
11.
```

Result Column Width

The Results Column Width option specifies the width of the “Results” (i.e. “Pass/Fail”) column of Text reports.

The default Results column width is 16 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_RESULT_COLUMN_WIDTH <width>

Width (in characters) of result columns in text reports. Its default value
is 16.
```

Complexity Column Width

The Complexity Column Width option specifies the width of the “Complexity” column in Metrics tables of Text reports.

The default Complexity column width is 10 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH <width>

Width (in characters) of complexity column in text reports. Its default
value is 10.
```

Coverage Result Column Width

The Coverage Results Column Width option specifies the width of the “Coverage Results” (i.e. “Statement Coverage”) column of Text reports.

The default Coverage Results column width is 18 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH <width>

Width (in characters) of coverage result columns in text reports. Its
default value is 18.
```

Notes Column Width

The Notes Column Width option specifies the width of the “Notes” column in Text reports.

The default Notes column width is 30 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_NOTES_COLUMN_WIDTH <width>

Width (in characters) of notes columns in text reports. Its default value
is 30.
```

Delimiter

The character separator for the Cover report created by the CLICAST report commands:**`cover report csv_metrics`** and **`reports alternate`**. Enter the character in quotes, as in “@”. To enter a tab, use “\t”. Valid delimiters are: ? , ' ; | { } [ ] @ ~ # $ _ \t \n

```
clicast Option VCAST_RPTS_DELIMITER <delimiter>

Character separator used in two CLICAST report commands cover report csv_
metrics and reports alternate. The default value is comma ",".
```

Wrap Text in the Notes Section

Setting this option causes the text in the Notes section to wrap at 80 characters in the Test Case Data and Full Reports in either TEXT or HTML format.

```
clicast Option VCAST_RPTS_WRAP_NOTES True | False

This option will cause the text in the notes section to wrap at the first
space before 80 characters. The default value is FALSE.
```

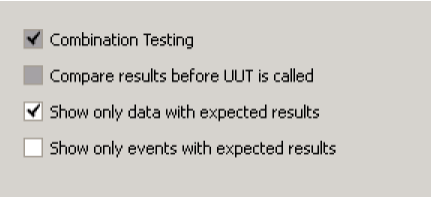
Setting Test Case Options

The Testcase Options Tab

Most of the options that are available in the Testcase Options tab are also available on the **Tools => Options dialog, Execute tab**. The environment-level settings are “inherited” by each test case, but can be over-ridden by setting a testcase-specific option here.

The states of a test case option are indicated below:

- Gray checkbox with a check – option is enabled and it is inherited.
- Gray checkbox with no check – option is disabled and it is inherited.
- White checkbox with a check – option is enabled and overrides inherited setting.
- White checkbox with no check – option is disabled and overrides inherited setting.



Display Integer Results in Hexadecimal

In VectorCAST version 5.0, this option has been removed from the GUI, as it has been replaced by the based notation feature in the Parameter Tree. It is still available as a test case option, however.

By choosing this option, the actual results for Integer types are displayed in hexadecimal rather than in the default decimal format in the execution reports.

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.DISPLAY_INTEGER_RESULTS_IN_HEX. Its default value is FALSE.

Floating Point Digits of Precision

This value controls exactly how the decimal part of a floating point number will be displayed. It modifies the amount of precision that VectorCAST will use to print a floating point number.

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.FLOAT_POINT_DIGITS_OF_PRECISION. Its default value is the environment-wide setting for Floating Point Digits of Precision, accessible on the Reports tab of the **Tools => Options** dialog.

Floating Point Tolerance (%)

The Floating Point Tolerance option enables you to set the tolerance of expected results for floating point numbers. This tolerance applies to all test case data in the environment.

Note: There is no tolerance around the value 0.

```
clicast -lada option FLOATING_POINT_TOLERANCE <tolerance>

Percentage that a floating point actual value can vary from its expected
value and still be considered a match. <percentage> is a floating point
number. Its default value is 0.000000.
```

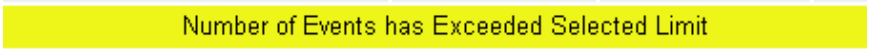
In a test script, the testcase-wide floating point tolerance is specified using:

TEST.FLOATING_POINT_TOLERANCE:<tolerance>

Note: For very small tolerances, the floating point representation of the value may not be exact due to the resolution of floating point numbers; in this case, the floating point tolerance may show more digits than the user expects. For example, a tolerance of 0.000000001 may actually be displayed as 0.00000000099999.

Event Limit

The Event Limit option enables you to control the maximum number of events per test harness execution. The event limit defaults to 1000. If this option is set to 0, no results data is captured, although coverage data is captured. If the event limit is exceeded, VectorCAST automatically stops test execution if your compiler supports signals, and the execution report shows:



The Execution Report shows <limit>+1 events.

The event limit applies to each test case. Each time a test executes, it resets the event counter to 1. If you execute several test cases at once using Ctrl+Shift click to select them, or execute them all by selecting the environment name and choosing **Test => Execute**, the counter starts over with each test case (because the test harness is being executed multiple times).

Note: An individual test case can have its own event limit, which overrides the environment event limit when that test case is executed. This option is accessed via the **Options tab** in the **Test Case Editor**.

When executing a compound test case, the event limit is not reset for each slot; the Event Limit setting applies to the Compound Test Case as a whole.

Note: The Event Limit feature is implemented through the raising of signals. If your target does not support signals, the event limit capability will not be available.

```
clicast -lada option EVENT_LIMIT <limit>

Maximum number of events for the harness to process. If this option is set
to 0, no results data is captured, although coverage data is captured. Its
default value is 1000.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.EVENT_LIMIT. Its default value is the environment-wide setting for Event Limit, accessible on the Execute tab of the **Tools => Options** dialog.

Number of Data Partitions

This option specifies the number of iterations to span an entire range of a scalar parameter. It is used in conjunction with Range Values to allow a specification of iterations instead of using a range delta.

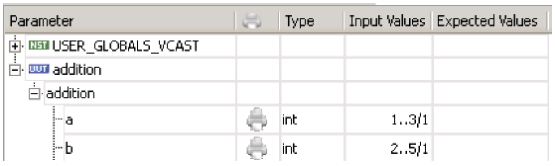
```
clicast -lada Option NUMBER_OF_DATA_PARTITIONS <positive number>

Number of iterations to span an entire range of a scalar parameter. Used
with Range Values to allow specification of iterations instead of range
delta. The default value is 1 partition.
```

Combination Testing

If a test case has a parameter with an input range or list, then the test harness executes the test case once for each iteration. When two or more parameters have a range iteration, then each parameter's iteration count is incremented at the same time, by default.

Combination testing can be turned on for an individual test case, using Test Case Options, or for the whole environment, using the **Tools => Options** dialog, **Execute tab**. If combination testing is enabled, VectorCAST will determine the combinations of input values and use those values as stimulus values.

For example, if parameter A has a range 1 to 3 with a delta of 1 and parameter B has a range 2 to 5 with a delta of 1, and Combination Testing is off, then the test case executes this way:



Range Iteration #1:

A is 1
B is 2

Range Iteration #2:

A is 2
B is 3

Range Iteration #3:

A is 3
B is 4

Range Iteration #4:

A is 3 again
B is 5

If the combination testing option is enabled, then the test case executes this way:

Range Iteration #1:

A is 1
B is 2

Range Iteration #2:

A is 1

B is 3

Range Iteration #3:

A is 1

B is 4

Range Iteration #4:

A is 1
B is 5

Range Iteration #5:

A is 2

B is 2

Range Iteration #6:

A is 2

B is 3

Range Iteration #7:

A is 2

B is 4

Range Iteration #8:

A is 2

B is 5

Range Iteration #9:

A is 3

B is 2

Range Iteration #10:

A is 3

B is 3

Range Iteration #11:

A is 3

B is 4

Range Iteration #12:

A is 3

B is 5

This feature is not available with an environment built with VectorCAST version 3.2 or earlier. Rebuilding the environment will enable this feature's functionality.

```
clicast -lada option VCAST_DO_COMBINATION true | false

Use all combinations of the values in the range expressions for test
stimulus values. The default value is False.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.DO_COMBINATION. Its default value is the environment-wide setting for Combination testing, accessible on the Execute tab of the **Tools => Options** dialog.

Compare Results Before UUT is Called

This option is used to cause VectorCAST to make comparisons of expected values to actual values before the UUT is even called. It is useful if a stub is called before you get to the UUT call (from user code in an <<INIT>> test case, for example), and you want to verify that a value was passed in correctly. By default, this option is false. Set it to true if you want to compare expected values before the UUT call.

```
clicast -lc option VCAST_EXPECTED_BEFORE_UUT_CALL <True | False>

Setting this option enabled comparisons between expected values and actual
values to occur before the call to the UUT. Its default value is FALSE.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.COMPARE_BEFORE_UUT. Its default value is the environment-wide setting for Compare Expected Values Before UUT Call, accessible on the Reports tab of the **Tools => Options** dialog.

Show Only Data with Expected Results

In some cases, a test may contain a huge number of global data input, but you may not want to capture all of this data in the Execution Reports. In the test case execution report, global and parameter data will only be displayed in the test execution report if expected values are provided; globals and parameters with input values only are not displayed.

```
clicast -lc option VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES True | False

When this option is set, global and parameter data will only be displayed
in the test execution report if expected values are provided. This option
will result in smaller execution reports and faster test execution in cases
```

where there is a lot of global and / or parameter data input. The default value is False.

TEST.VALUE:<<OPTIONS>>.SHOW_ONLY_DATA_WITH_EXPECTED_RESULTS:TRUE | FALSE

If specified in a test case script, this option causes only events that have Expected values to be included in the Execution Report. It overrides the environment-wide setting for "Show only events with expected results." If not specified in a test script, the default value is the environment-wide setting, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.

Show Only Events with Expected Results

Setting this option causes VectorCAST to leave out any events in the Execution Report that do not have expected data for any parameter or global value. This option will result in smaller execution reports in cases where there is a lot of global and /or parameter data input.

clicast -lada option VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES True | False

When this option is set, events will only be displayed in the test execution report if expected values exist for that event. This option will result in smaller execution reports in cases where there is a lot of global and / or parameter data input. The default value is False.

TEST.VALUE:<<OPTIONS>>.SHOW_ONLY_EVENTS_WITH_EXPECTED_RESULTS:TRUE | FALSE

If specified in a test case script, this option causes only events that have Expected values to be included in the Execution Report. It overrides the environment-wide setting for "Show only events with expected results." If not specified in a test script, the default value is the environment-wide setting, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.

Display Global Data Options

The Display Global Data After feature gives you control over when global data objects are displayed in the test execution report. If an expected value for the global object is set, then that expected value is compared to its actual value at the same time.

Ranging from displaying more frequently to less frequently, you can choose to display global data after:

- Each event (default)
- Each range iteration
- Each slot iteration
- Each test case

The default **Each Event** displays the current value(s) of global data objects at every transfer of

control flow.

Each range iteration displays the current value(s) of global data objects after every iteration caused by a parameter's range of input values. If no parameter has an iteration, then the globals are displayed at the end of the test execution.

Each slot iteration displays the current value(s) at the end of each iteration of a test case in a compound test case. A compound can have one or more test cases in it, each with one or more iterations. This setting causes the globals to be displayed (and compared) when each test case finishes an iteration. If you apply this setting to a test case and then execute that test case alone (i.e. not in a compound), then the globals are displayed at the end of test execution, because it is the only "slot." In the example below, the global values would be displayed twice, at the end of each slot iteration.

Slot	Subprograms	Test Cases	Iterations	Delay
1	Get_Check_Total	GET_CHECK_TOTAL.001	2	0

Each test case displays the current value(s) of global data at the end of a slot in a compound test case. A compound can have one or more test cases in it; this setting causes the globals to be displayed and compared when each test case in a compound finishes executing all of its iterations. If you apply this setting to a test case and then execute that test case alone (i.e. not in a compound), then the globals are displayed at the end of test execution, because it is the only "slot."

```
clicast -lada option GLOBALS_DISPLAY Each_event|Range_Iteration|Slot_
Iteration|Testcase

When to display and check global values in execution results. Its default
value is EACH_EVENT.
```

In a test case script, this option is

```
 TEST.VALUE:<<OPTIONS>>.GLOBAL_DATA_DISPLAY.

Its default setting is the environment-wide setting for the Display global
data option, accessible on the Report tab of the Tools => Options dialog.
```

Deprecated Test Case Options

The test case option "Display Integer Results in Hexadecimal" was deprecated in VectorCAST version 4.2. You can now enter the value in a based notation in the Parameter Tree. The base used for the Expected value is the one that appears in the Execution Report for that parameter.

TEST.VALUE:<<OPTIONS>>.DISPLAY_INTEGER_RESULTS_IN_HEX

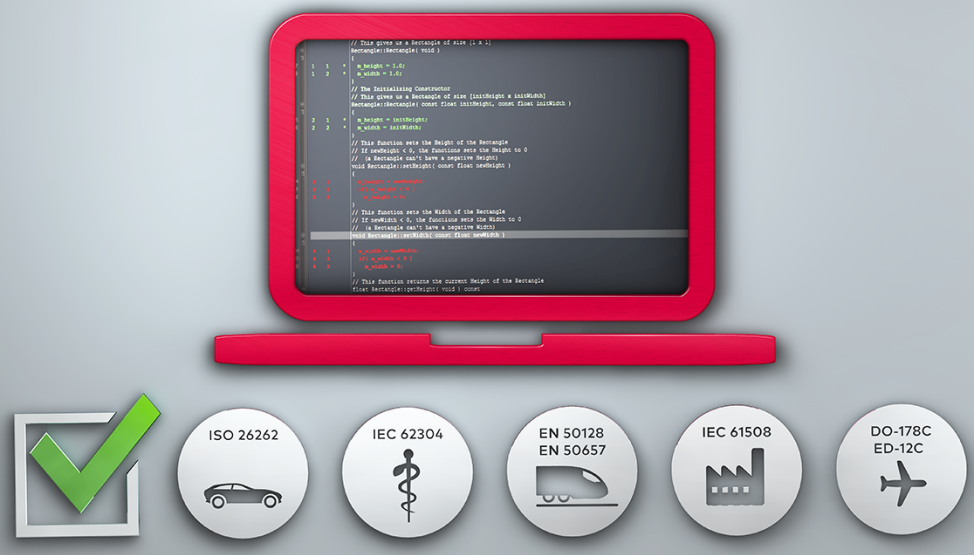
Deprecated Report Options and Commands

VCAST_COMPACT was deprecated in VectorCAST version 4.2. It is always true.

`clicast execute extract coverage` was deprecated in VectorCAST version 4.0. use **`clicast reports custom coverage`**.

`Clicast execute extract results` was deprecated in VectorCAST version 4.0. use **`clicqast reports custom actual`**.

`clicast reports report` was deprecated in VectorCAST version 4.0. Use **`clicast reports custom full`**.



Incremental Rebuild

About Change-Based Testing

The VectorCAST Project Manager uses the Incremental Rebuild feature in VectorCAST/Ada to allow users to perform Change-Based Testing (CBT). The VectorCAST Project Manager automatically identifies the minimum tests that must be run for each code change. VectorCAST scans the code for changes since the last test run, identifies the sub-set of tests that are affected by the change and automatically re-runs only those tests.

Change-Based Testing results in greatly reduced test time, allowing for more frequent testing, and ensuring that bugs are fixed when they are introduced, instead of during a full test cycle at a later date.

Change-Based Testing in the VectorCAST Project Manager:

- Builds the environment if unbuilt
- Builds incrementally for function scope changes, or
- Builds fully for global (or file) scope changes, and
- Executes any new tests, or tests affected by the source change.

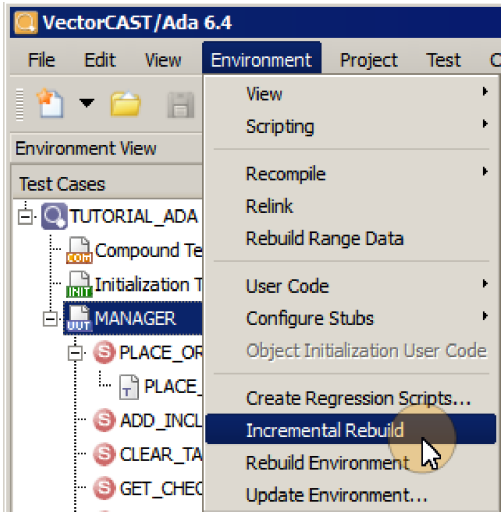
This section describes the Incremental Rebuild feature that the VectorCAST Project Manager uses in Change-Based Testing. Incremental Rebuild can be used in a unit test environment by the user in the same manner that the VectorCAST Project Manager uses it.

Performing an Incremental Rebuild

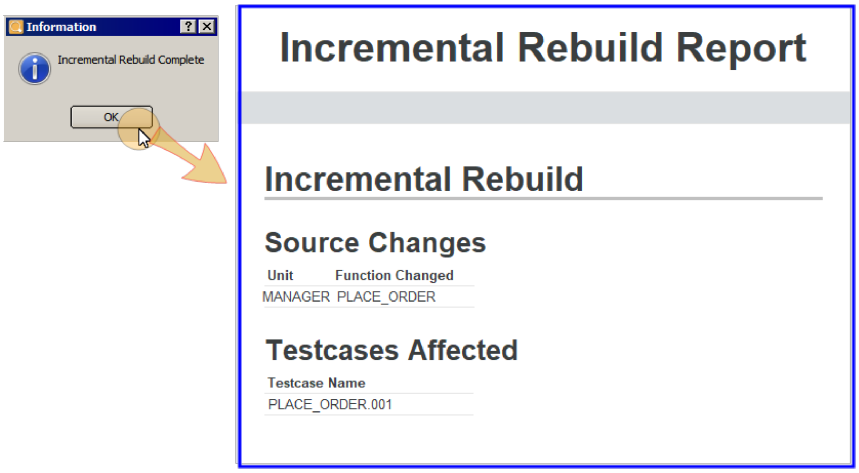
The Incremental Rebuild feature allows you to incorporate source code changes into the test harness without having to fully rebuild the environment. The environment must be whitebox, must have coverage instrumented and tests executed in order for VectorCAST to be able to detect which subprograms have been modified and which test cases have been affected.

If the environment does not have coverage, a full rebuild is required because the changes are considered global in scope.

To perform an incremental rebuild, first make any modifications to a UUT or dependent body. Next, save the changes and recompile the source. Then select **Environment => Incremental Rebuild** from the Menu Bar.



Upon selection of the Incremental Rebuild option, if VectorCAST detects a change to UUTs or non-stubbed dependents, it determines which package bodies or specs have changed, updates the associated portions of the test harness, and recompiles and then re-instruments those portions. A pop-up message is displayed indicating the incremental rebuild is complete. Select the **OK** button and the Incremental Rebuild Report is displayed in the MDI window.

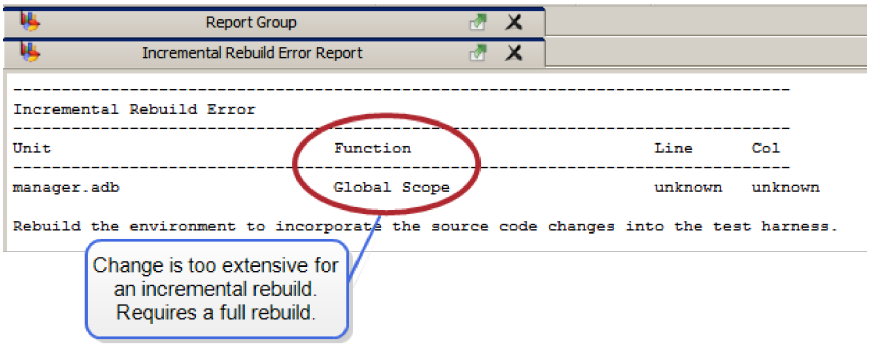


The Incremental Rebuild Report indicates which units were modified and which subprograms were affected by the change. If execution results exist, the report also indicates which test cases were affected.

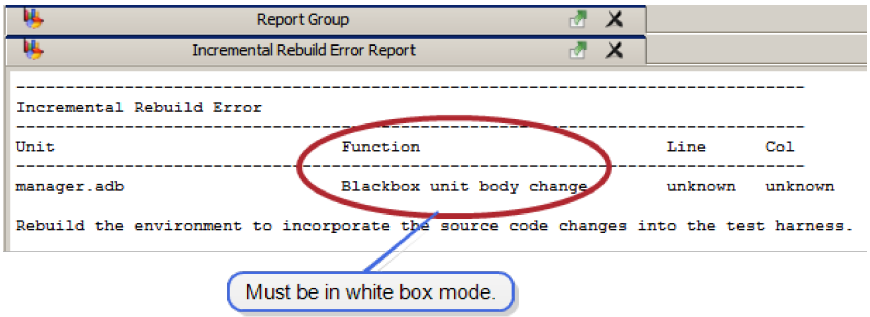
Execution and coverage status are selectively removed following an incremental rebuild. Assuming the environment has coverage instrumented, the test cases that call the modified function (and those whose non-stubbed subprograms call the modified function) are considered "affected," and their coverage data and execution data are removed from the environment. Unaffected test cases retain their execution and coverage data.

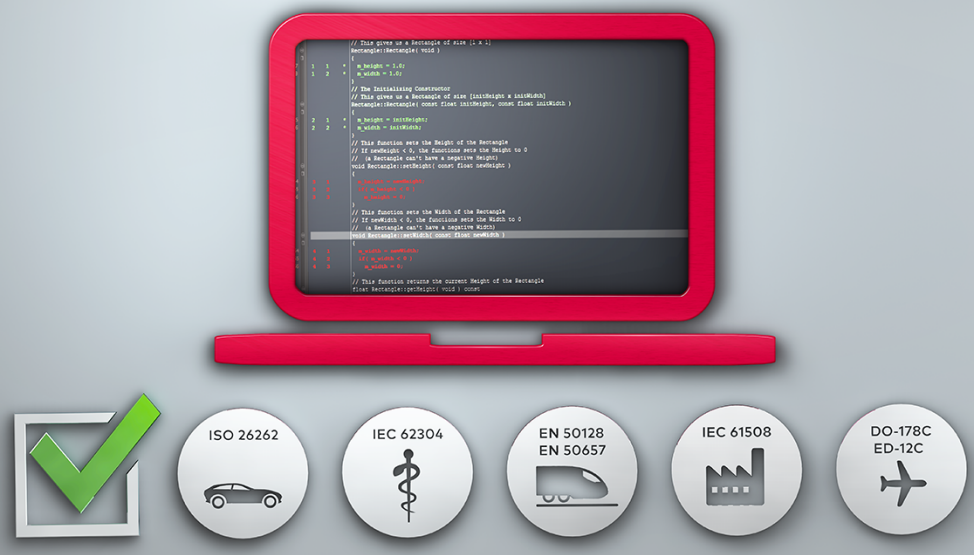
Note: The Parameter Tree does not reflect changes to the source code after an Incremental Rebuild; only the test harness is affected. To see changes such as renaming an enumerated value, changing a parameter name, etc. reflected in the Parameter Tree, you must fully rebuild the environment.

Global source code changes require a full rebuild. If a source code modification adds a new dependency, such as a new parameter, when incremental rebuild is initiated the user is notified that the change is too large to be rebuilt incrementally. The Incremental Rebuild Error Report prompts the user to do a full rebuild of the environment to incorporate the source code changes into the test harness.



VectorCAST/Ada supports incremental rebuild of Ada whitebox environments. Attempting to perform an incremental rebuild on a blackbox unit produces the following Incremental Rebuild Error Report. The environment must be updated to a whitebox test environment.





Using Code Coverage

Code Coverage

The Code Coverage tool determines which lines of source code (statement), which branches of source code (branch), or which equivalence pairs (MC/DC) have been executed by one or more sets of test case data. The reports show you the completeness of your test suite. By analyzing the untested code, you can easily work backward designing test cases to test these portions of your unit.

VectorCAST initializes the following coverage types:

- **Statement** - reports on which executable lines of source code are being executed as the program runs.
- **Branch** - displays which outcomes have occurred for each branch point in the program.
- **MC/DC** - similar to branch coverage, but in addition to reporting on the outcomes of all Boolean expressions, it also reports on the values of all Boolean components of a complex conditional.
- **Statement + MC/DC** - reports on which executable lines of source code are being executed as the program runs and on the outcomes of all Boolean expressions, and reports on the values of all Boolean components of a complex conditional.
- **Statement + Branch** - reports on which executable lines of source code are being executed as the program runs and displays which outcomes have occurred for each branch point in the program.
- **Basis Paths** - identifies all possible paths of execution and consists of two parts: the list of basis paths and an associated annotated source listing.

The current Industry Mode (see “To Set The Industry Mode” in Section 1) determines the set of coverage choices. If no Industry Mode is set, Default is used. The following is a list of coverage types listed by Industry Mode:

- Avionics (DO-178 B/C)
  - Level A (Statement, Branch, MC/DC),
  - Level B (Statement, Branch)
  - Level C (Statement)
- Automotive (ISO-26262)
  - Unit Level ASIL A (Statement)
  - Unit Level ASIL B/C (Statement, Branch)
  - Unit Level ASIL D (Statement, Branch, MC/DC)
- Industrial (IEC-61508)
  - SIL4 (Statement, Branch, MC/DC)
  - SIL3 (Statement, Branch)
  - SIL 1/2 (Statement)
- Railway (EN-50128)
  - SIL4 (Statement, Branch, MC/DC),
  - SIL3 (Statement, Branch)
  - SIL 1/2 (Statement)

- Medical (IEC-62304)
  - Class C (Statement, Branch, MC/DC)
  - Class B (Statement, Branch)
  - Class A (Statement)
- Default
  - Statement
  - Branch
  - Basis Paths
  - MC/DC
  - Statement + Branch
  - Statement + MC/DC

Initializing a type of coverage results in the instrumentation, compilation, and linking of an instrumented version of the test harness which enables VectorCAST to monitor coverage during the execution of test cases.



Note: Because code coverage runs in parallel with normal VectorCAST functionality, test cases run with coverage ON can always be re-run on the non-instrumented test harness to verify real-time performance.

To Initialize Statement Coverage

Statement coverage reports on which executable lines of source code are being executed as the program runs. Each executable statement in the file under test appears with a corresponding subprogram number and executable line number.

```
1 5     *    switch (Order.Entree)
             {
             case NO_ENTREE :
1 6            break;
             case STEAK :
1 7     *      Table_Data.Check_Total = Table_Data.Check_Total + 14.0;
1 8     *      break;
             case CHICKEN :
1 9     *      Table_Data.Check_Total = Table_Data.Check_Total + 10.0;
1 10    *      break;
               ...
```

To initialize Statement coverage, select a source file and choose **Coverage => Initialize => Statement**.

Statement coverage maps to the Industry Modes as shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Statement	Level A Level B Level C	Unit Level ASIL A Unit Level ASIL B/C Unit Level ASIL D	SIL 1/2 SIL 3 SIL 4	SIL 1/2 SIL 3 SIL4	Class A Class B Class C

To initialize Statement coverage when using Industry Modes, choose **Coverage => Initialize => *<appropriate type>*** based on the current Industry Mode.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> TOols Cover Statement [ <unitname> | ALL]

Initialize coverage instrumentation of all UUT(s) in the environment for
Statement coverage. Similar to Initialize Custom, the optional argument
<unitname> is the name of a non-stubbed unit to be instrumented in addition
to the UUTs, or "ALL" to instrument all non-stubbed units in the
environment as well.
```

To Initialize Branch Coverage

Branch coverage displays which outcomes have occurred for each branch point in the program. Each source line that contains a branch point is displayed with a subprogram number, the branch point number, and space for the condition value.



Note: The branch point numbers will not necessarily be consecutive.

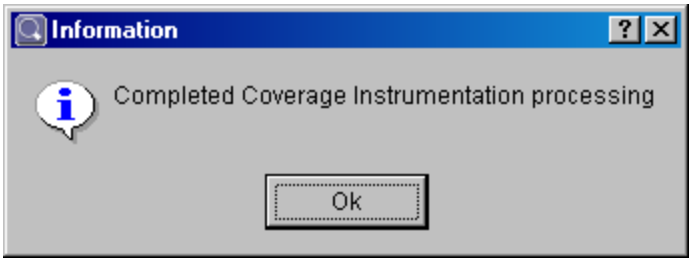
Each branch point will have space for either one or two condition values. Boolean decision points (i.e., if statements) are displayed with two place holders, because the condition can be either true or false, as in the following example:

```
1 0       (T)    Add_Included_Dessert
1 1       (T) (F)    if((Order->Entree == STEAK &&
                         Order->Salad == CAESAR &&
                         Order->Beverage == MIXED_DRINK))
                   {
                      Order->Dessert = PIE;
                   }
                   else
1 3       (T) (F)       if((Order->Entree == LOBSTER &&
                            Order->Salad == GREEN &&
                            Order->Beverage == WINE))
                         {
                            Order->Dessert = CAKE;
                         }
                 }
```

Case statements are handled by treating each "when" as a single decision branch point, as in the following example:

```
1 1 case ORDER.ENTREE is

1 2 (T) when TYPES.NO_ORDER =>
null;

1 3 ( ) when TYPES.STEAK =>
TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL + 14;

1 4 ( ) when TYPES.CHICKEN =>
TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL + 10;

1 5 (T) when TYPES.LOBSTER =>
TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL + 18;

1 6 ( ) when TYPES.PASTA =>
TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL + 12;
end case;
```

To initialize Branch coverage, choose **Coverage => Initialize => Branch**. Once VectorCAST has completed instrumenting the test harness, the following message is displayed:



Branch coverage maps to the Industry Modes as shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Branch	Level A Level B	Unit Level ASIL B/C Unit Level ASIL D	SIL 3 SIL 4	SIL 3 SIL4	Class B Class C

Note: Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> TOols Cover Branch [<unitname> | ALL]

Initialize instrumentation of all UUT(s) in the environment for Branch
coverage. Similar to Initialize Custom, the optional argument <unitname> is
the name of a non-stubbed unit to be instrumented in addition to the UUTs,
or "ALL" to instrument all non-stubbed units in the environment as well.
```

To Initialize MC/DC Coverage

MC/DC coverage is similar to branch coverage, but in addition to reporting on the outcomes of all

Boolean expressions, it also reports on the values of all Boolean components of a complex conditional. A complex conditional is one in which there is more than one "term" in the conditional. For example: if (a && b) is a complex conditional with two terms.



Note: VectorCAST displays a progress bar when initializing MC/DC or Level A coverage on a source file having an expression with more than 10 sub-expressions.

MC/DC coverage maps to the Industry Modes as shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
MC/DC	Level A	Unit Level ASIL D	SIL 4	SIL4	Class C

Pairs Status

When MC/DC coverage is initialized, an additional “Pairs” metric is available in the coverage Metrics Report, **Test => View => Metrics Report** and the Test Case Management Report, **Test => View => Test Case Management Report**

Pairs Status consists of the total number of MC/DC pairs satisfied compared to the total number of MC/DC pairs in the unit.

To initialize MC/DC coverage, select a source file, and choose **Coverage => Initialize => MC/DC**.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> TOols Cover MCDC [<unitname> | ALL]

Initialize instrumentation of all UUT(s) in the environment for MC/DC
coverage. Similar to Initialize Custom, the optional argument <unitname> is
the name of a non-stubbed unit to be instrumented in addition to the UUTs,
or "ALL" to instrument all non-stubbed units in the environment as well.
```

Suppressing MC/DC Initialization on Compile Error

In some rare cases, VectorCAST cannot successfully instrument MC/DC coverage on an expression in your source code. As a workaround (be sure to contact Technical Support), you may wish to suppress MC/DC coverage for a particular statement or set of statements in your application. It is possible to do this by adding comment tags to the source code.

The Comment Tag Method

To disable MC/DC coverage for a certain line of source code, place the comment VCAST_DONT_DO_MCDC before the line. The comments must exist on lines with no other text, immediately preceding the source code line for which you want to modify the coverage. The text can be upper-

or lowercase.

Note: For Ada source code, the comments should be inside the **begin** block. There can be no spaces after the comment characters.

For example:

```
-- VCAST_DONT_DO_MCDC
specific statement to which you do not want MC/DC coverage applied.
-- VCAST_DO_MCDC
specific statement to which you DO want MC/DC coverage applied
```

To Initialize Statement+MC/DC

Statement+MC/DC coverage maps to the Industry Modes as is shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Statement + MC/DC	Level A	Unit Level ASIL D	SIL 4	SIL 4	Class C

To initialize Statement+MC/DC coverage when using Industry Modes, choose **Coverage => Initialize => <*appropriate type*>** based on the current Industry Mode.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> Tools COVER STATEMENT+MC/DC [<unitname> | ALL]

Initialize instrumentation of all UUT(s) in the environment for
STATEMENT+MC/DC coverage. The optional argument <unitname> is the name of a
non-stubbed unit to be instrumented in addition to the UUTs, or "ALL" to
instrument non-stubbed units in the environment as well.
```

To Initialize Statement+Branch

Statement+Branch coverage maps to the Industry Modes as is shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Statement + Branch	Level B	Unit Level ASIL B/C	SIL 3	SIL 3	Class B

To initialize Statement + Branch coverage when using Industry Modes, choose **Coverage =>**

Initialize => *<appropriate type>* based on the current Industry Mode.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

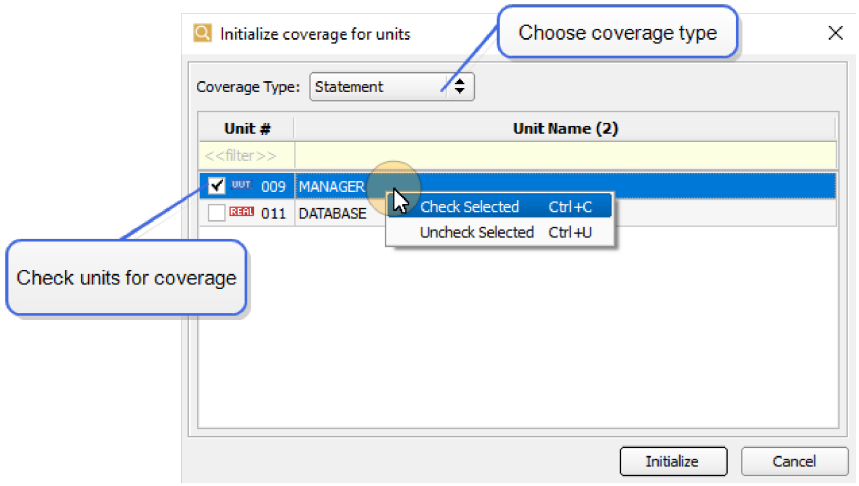
Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> TOols Cover STATEMENT+BRANCH [<unitname> | ALL]

Initialize instrumentation of all UUT(s) in the environment for
STATEMENT+BRANCH coverage. The optional argument <unitname> is the name of
a non-stubbed unit to be instrumented in addition to the UUTs, or "ALL" to
instrument all non-stubbed units in the environment as well.
```

To Initialize Coverage for Units Other than UUT

Coverage = > Initialize Custom... enables you to instrument multiple units (the UUTs and non-stubbed dependents, for example). When you choose **Coverage => Initialize Custom..**, a dialog appears listing all units that can be instrumented. The unit(s) under test (UUTs) are always listed first in the list. Each UUT displayed is capable of being selected for coverage.

In the figure below, the **MANAGER** unit is a UUT (UUT), and **DATABASE** is a non-stubbed unit (REAL). Any unit that is not-stubbed or ignored is given the icon REAL .



You may select individual units for coverage by checking the box next to the unit name or right-clicking and choosing **Check Selected**. After selecting the units, choose a coverage type from the drop-down list, and click the **Initialize** button to initialize coverage for the selected units.

Once you initialize a coverage type for real units (that is, non-stubbed or ignored units), any subsequent use of **Coverage => Initialize** affects those units as well.

Note: It is not possible to specify different types of coverage for different units.

To exit the dialog without initializing coverage, click the **Cancel** button.

```
clicast -e <env> Tools <coverage-type> [<unit_name1> <unit_name2…] | [ALL]

where <coverage-type> is one of < STATEMENT | STATEMENT+MC/DC |
STATEMENT+BRANCH | MC/DC | BASIS_PATHS | NONE >

Initialize instrumentation of all UUT(s) in the environment for <coverage-
type>. Similar to Initialize Custom, the optional arguments <unit_Xname> is
the name of a non-stubbed units to be instrumented in addition to the UUTs,
or "ALL" to instrument all non-stubbed units in the environment as well.
```

To disable coverage for a particular UUT, use the following CLICAST command:

```
clicast -e <env> -u <unit> TOols Cover UUT_Disable <coverage-type>

where <coverage-type> is one of Statement | Branch | Basis_paths | MCDC |
STATEMENT+MC/DC | STATEMENT+BRANCH | None

Disable instrumentation of <unit> the next time coverage is instrumented.
The <coverage-types> argument specifies the type of coverage to initialize
for the other UUTs which still have coverage enabled.
```

To Avoid Instrumenting Sections of Source Code

There may be sections of your source code that you do not want to instrument for code coverage.

You can instruct VectorCAST to ignore sections of your source code by delimiting the section with the following source code comment markers:

```
(Ada)--VCAST_DONT_INSTRUMENT_START

Code that should not be instrumented for code coverage…

--VCAST_DONT_INSTRUMENT_END
```

The text can be upper- or lowercase, but there can be no spaces between the comment characters and the text of the delimiter. Care must be used to ensure that the start and end delimiters are within the same scope. That is, if you have a **`case`** statement, you do not insert the delimiters such that half of the case statement is not instrumented and half is instrumented for code coverage.

Note: For Ada source code, the comments should be inside the **`begin`** block. There can be no spaces after the comment characters.

To Uninstrument an Environment

To completely remove coverage instrumentation from unit test environments, select **Coverage =>**

Uninstrument from the Menu Bar. Selecting the Uninstrument command removes all the coverage data and the environment is no longer instrumented.



Note: Execution results are not removed when the environment is uninstrumented.

```
clicast -e <env> TOols Cover None

Remove coverage from all UUTs and non-stubbed units with custom coverage in
a unit test environment.
```

To Enable Coverage

The **Coverage => Enable** command turns on code coverage for the current open environment. This command is only valid for environments that previously had coverage initialized and disabled. Subsequent to enabling coverage, all test cases will be run against the instrumented test harness (UUT_INST.EXE).



Note: The status bar indicates that coverage is enabled by displaying "Coverage type: <type>."

```
clicast -e <env> TOols Cover Enable

Run subsequent test cases with coverage monitoring enabled, using the
instrumented test harness.
```

To Disable Coverage

The **Coverage = > Disable** command turns off coverage for the current environment. This command is only available for environments that have coverage initialized and enabled. All subsequent test case execution will use the non-instrumented harness. The purpose of disabling coverage is to enable you to run test cases using the original, non-instrumented test harness (UUT_INTE.EXE), so that you can ensure that the coverage instrumentation does not affect the test results.

If this menu item is dimmed, coverage is already disabled or has not been initialized.



Note: The status bar indicates that coverage is disabled by displaying "Coverage type: <type>" (disabled)."

```
clicast -e <env> TOols Cover Disable

Run subsequent test cases with coverage monitoring disabled, using the un-
instrumented test harness.
```

Viewing Coverage Results

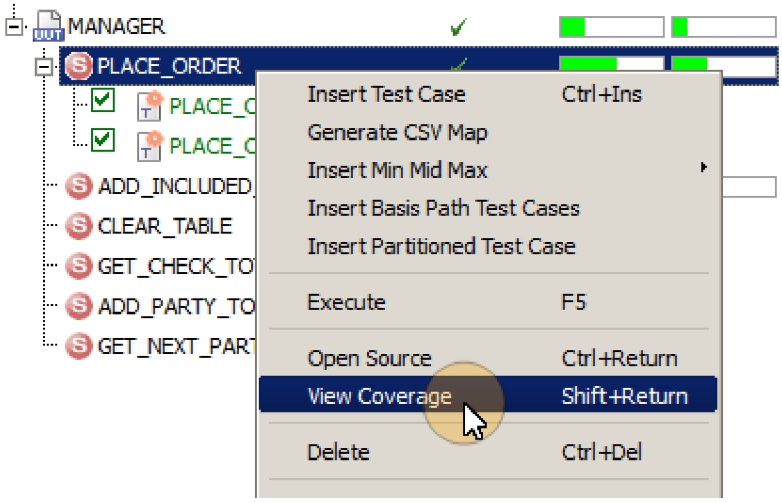
To Turn on Coverage Results

Once coverage has been initialized and test case(s) run, there are several ways to view the achieved

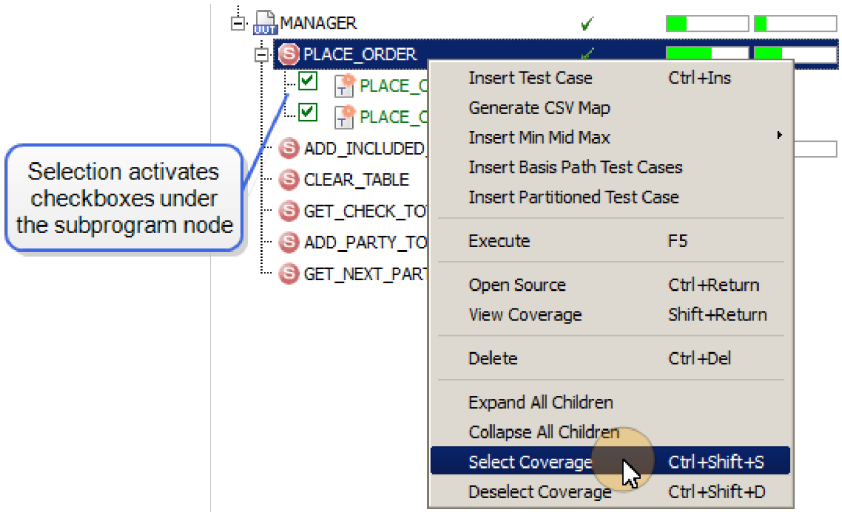
coverage. Each unit has its own Coverage Viewer which contains an annotated version of the source file, colorized to indicate the coverage level achieved.

The Coverage Viewer is opened by highlighting unit, subprogram or test case nodes in the Test Case Tree and clicking the View Coverage icon in the Toolbar.

You can also open the Coverage Viewer for any number of units by selecting those units in the Test Case Tree, right-clicking, and selecting **View Coverage** from the context menu.



To view coverage achieved by multiple test cases, first select multiple test case, subprogram, or unit nodes in the Test Case Tree. Then right-click and choose **Select Coverage**, which activates the checkboxes next to all selected test cases. The Coverage Viewer reflects the coverage achieved by the selected test cases.

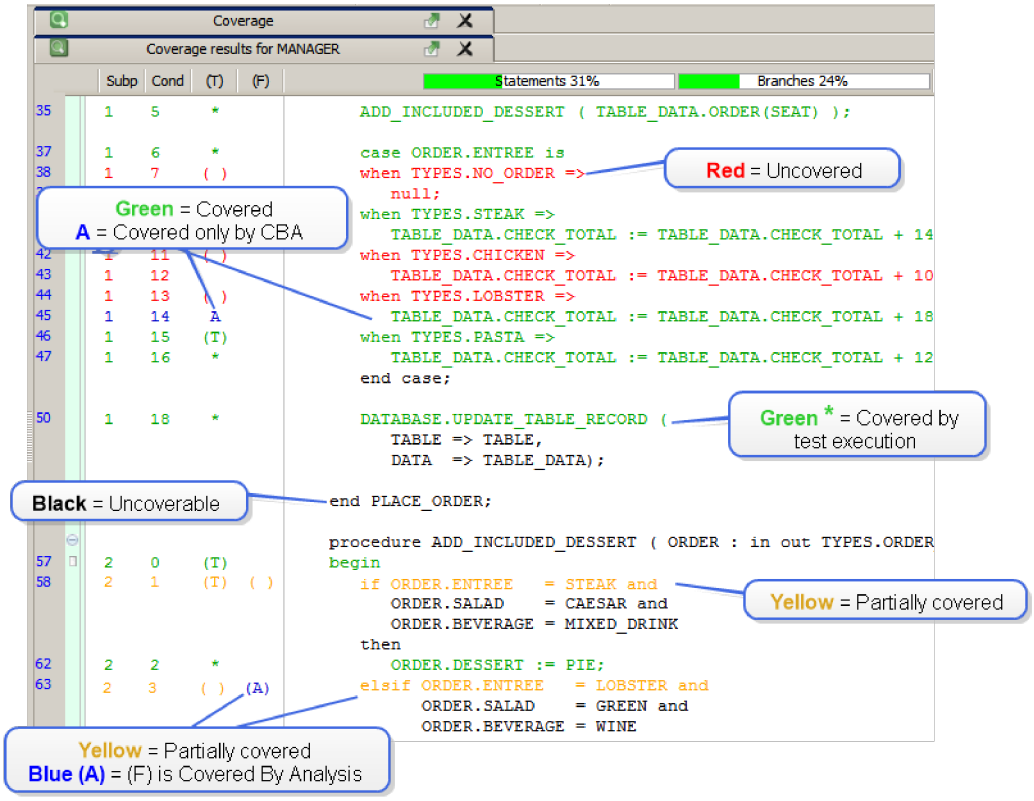


The Coverage Viewer

The Coverage Viewer contains an annotated version of the source file, colorized to indicate the

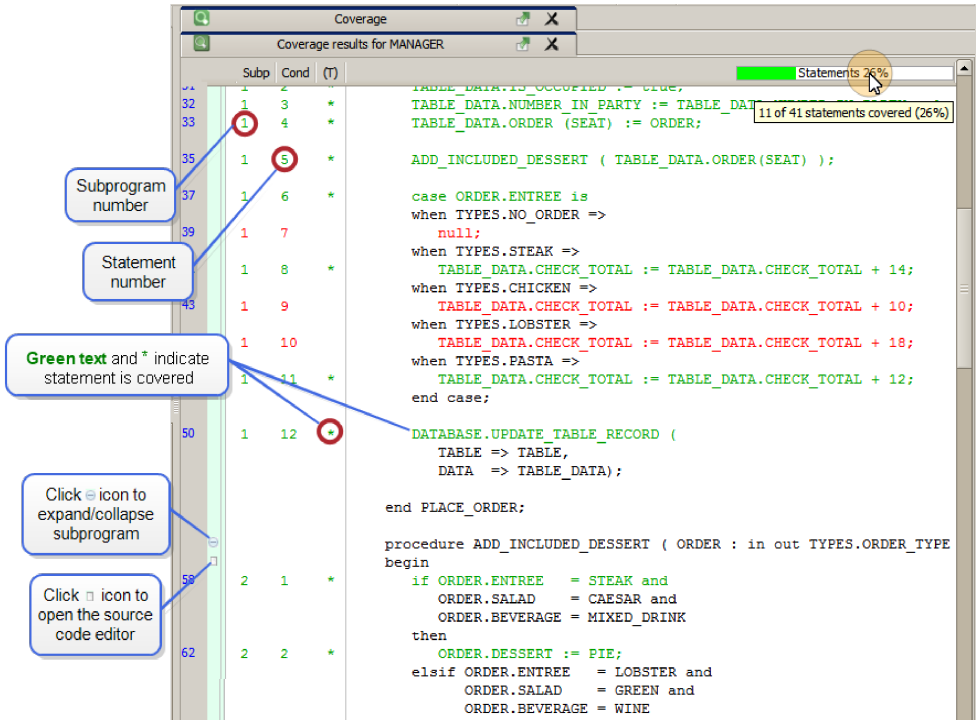
coverage level achieved:

- Covered lines are displayed in green
- Lines covered only by test execution results are displayed in green (indicating the line is covered) with a green asterisk "*" (indicating the line is covered *only* by test execution results)
- Lines covered only by CBA results are displayed in green (indicating the line is covered) with a blue "A" (indicating the line is covered *only* by CBA)
- Lines covered by both regular execution results *and* CBA results are displayed in green (indicating the line is covered) and with a blue asterisk "*" (for statement) or a blue "T" or "F" (for branch), which indicates that it is *also* covered by CBA
- Uncovered lines are displayed in red
- Partially covered lines are displayed in yellow
- Uncoverable lines are displayed in black



Statement Coverage

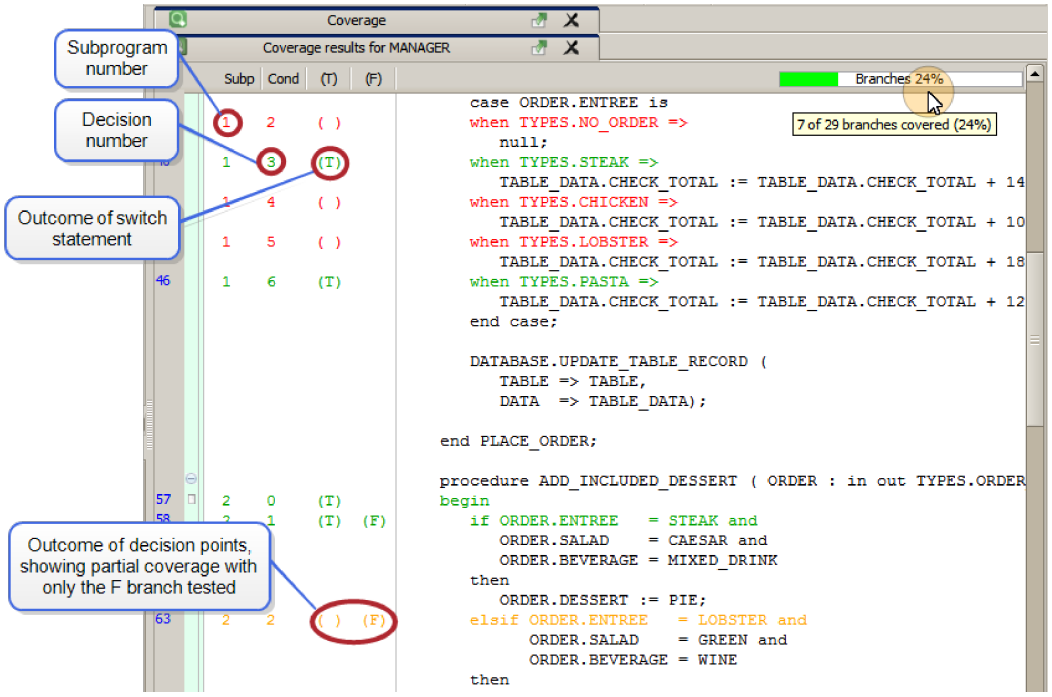
When Statement coverage is active, the Coverage Viewer looks like the following example:



The numbers along the left-hand margin indicate the subprogram number and the statement number. An asterisk (*) indicates that the statement was covered (along with the color-coding).

Branch Coverage

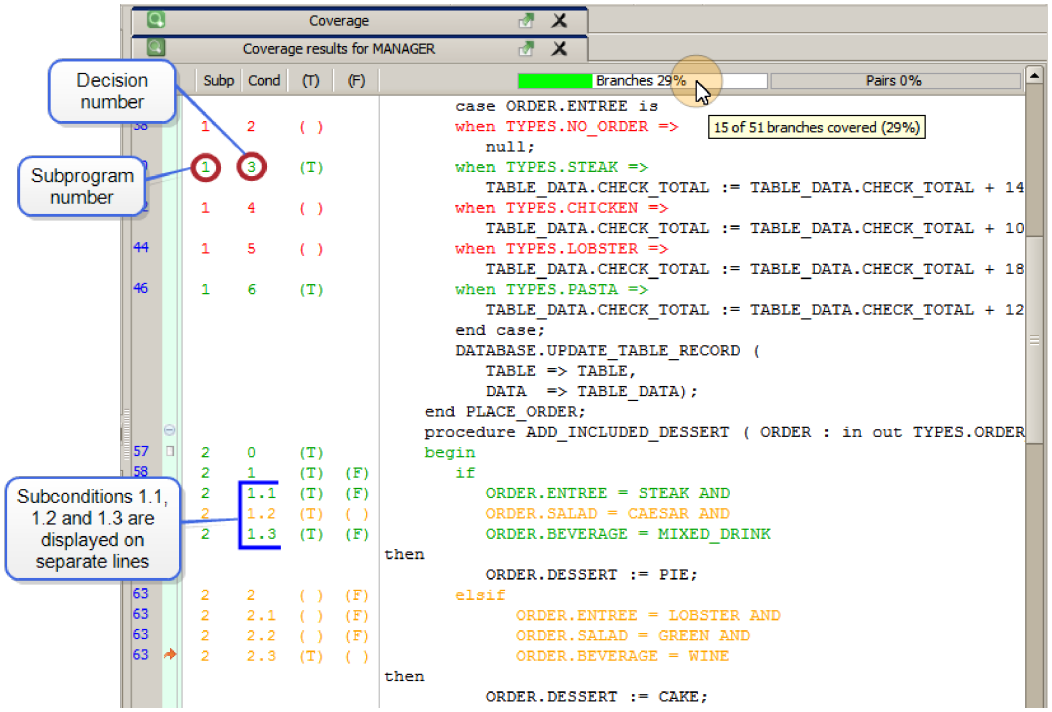
When Branch coverage is active, the Coverage Viewer looks like the following example:



The left-most number indicates the subprogram number, and the second number indicates the decision number. For Boolean decision points, two sets of parentheses are visible and are filled in with a T or F when the True or False outcomes have been tested. For switch statements, one set of parentheses is visible and a **(T)** is filled in when that branch outcome has been tested.

MC/DC Coverage

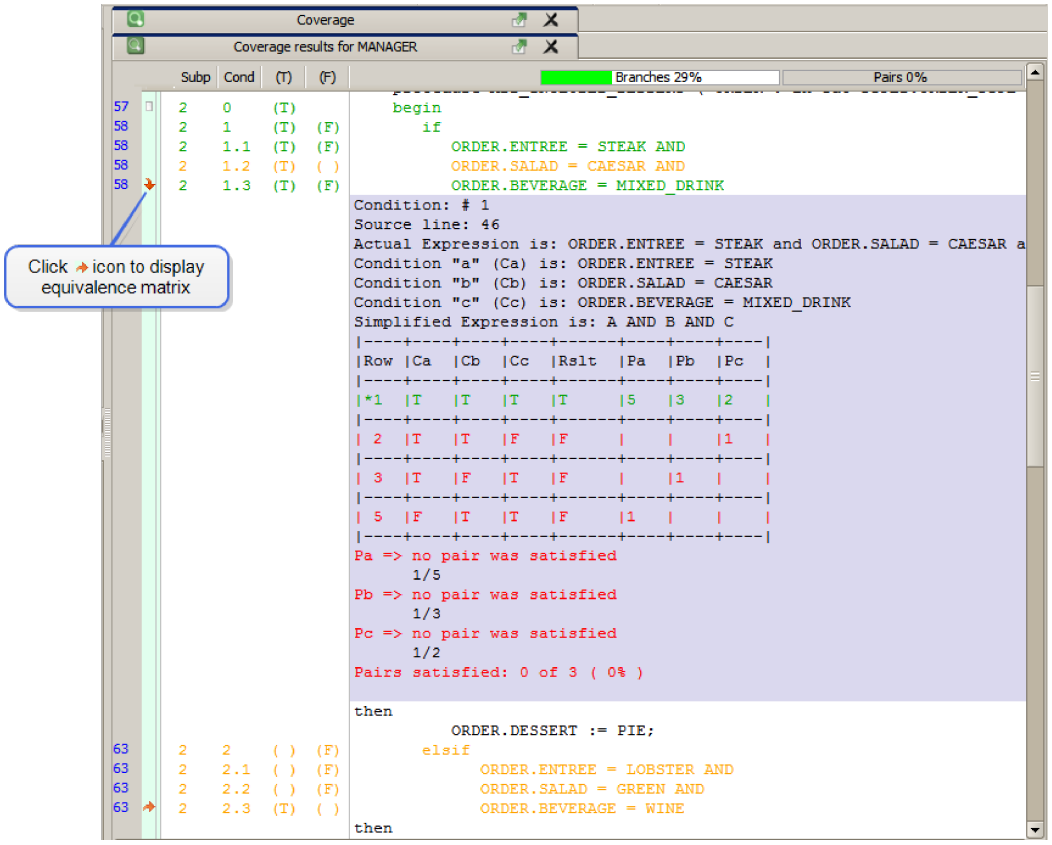
When MC/DC coverage is active, the Coverage Viewer looks like the following example:



The left-most number indicates the subprogram number, and the second number indicates the decision number. For Boolean decision points, two sets of parentheses are visible and are filled in with a T or F when the True or False outcomes have been tested. For decision points that have multiple components (e.g., `a || b`), each sub-condition is displayed on a separate line. The sub-conditions are numbered `.1`, `.2` to `.n`, where `n` is the total number of sub-conditions.

Note: When you re-initialize coverage, rebuild or update the environment, or regenerate basis paths, the coverage results are removed.

For MC/DC or STATEMENT+MC/DC coverage, the left margin includes a red arrow that, when clicked, opens the Equivalence Matrices below the associated subprogram in the Coverage Viewer.



Navigating Coverage Results

Four buttons are available in the toolbar to allow navigation of the coverage listing.

 Previous Covered Result (green)

 Next Covered Result (green)

 Previous Uncovered Result (red)

 Next Uncovered Result (red)

When the arrow buttons are clicked, the coverage source window scrolls to show the next covered or uncovered line. Partially-covered lines are considered both covered and uncovered. Contiguous blocks of lines are skipped over, until a line of a different block type is found or a line off-screen is found.

If there are no lines found, the message “Search reached bottom” or “Search reached top” appears, depending on which direction you were searching.

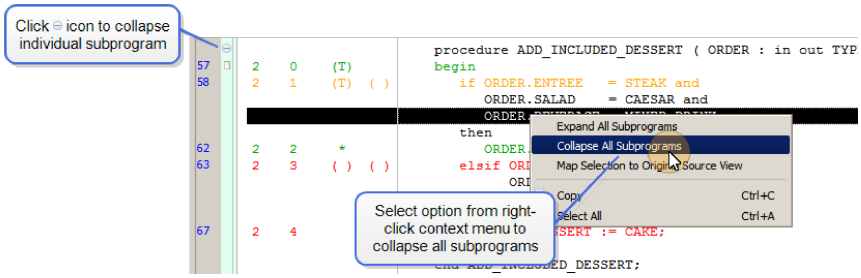
In addition, pressing the **Home key** or **End key** (or alternatively, pressing **Ctrl+Home** or **Ctrl+End**)

brings the cursor to the beginning or end of the file, respectively.

To Customize the Coverage Viewer

By default, each unit's tab in the Coverage Viewer shows all subprograms in that unit as fully expanded. You can customize the display by collapsing and expanding subprograms.

To customize the display in the Coverage Viewer, right-click anywhere in the window. The context menu appears allowing you to expand or collapse all subprograms in the viewer.



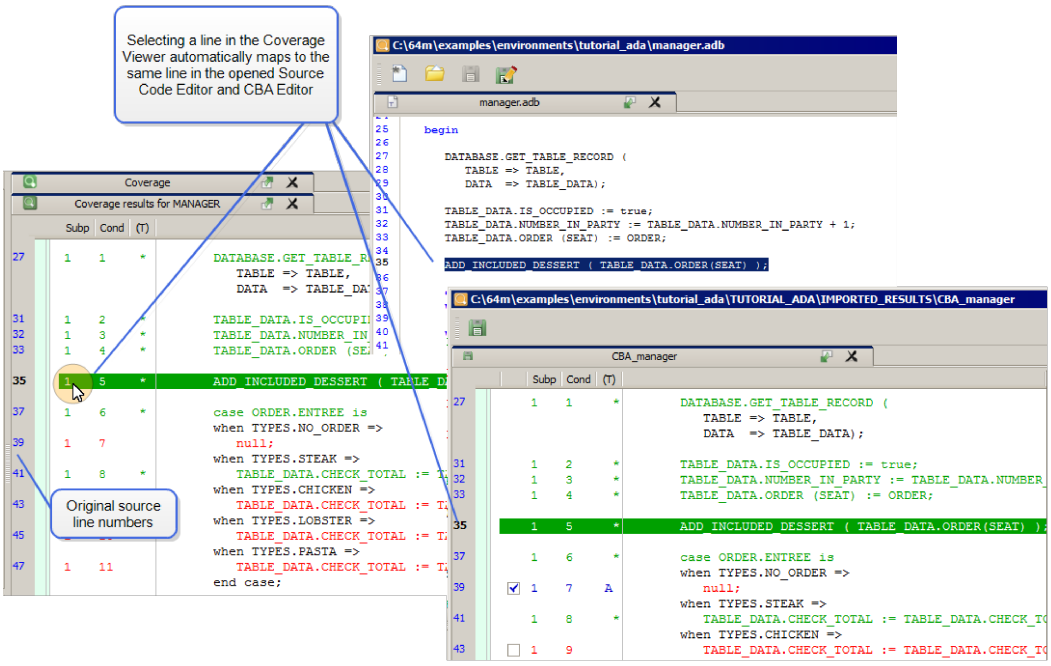
Use the Expand and Collapse menu items as "filters" or "layers" to get the exact display you want in the Coverage Viewer.

You can also expand and collapse individual subprograms by clicking the (collapse) and (expand) icons located on the left of the viewer.

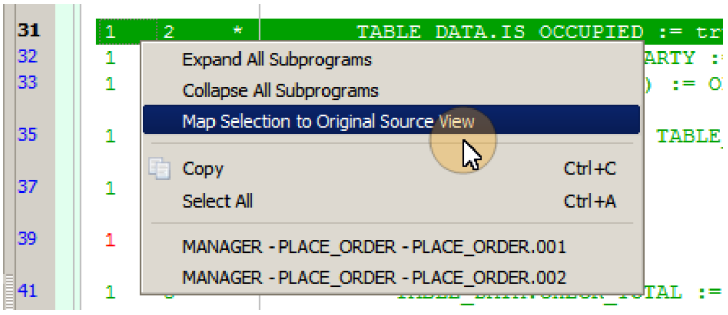
Collapsing and expanding the subprograms in the Coverage Viewer does not affect any of the Coverage reports.

Map Line Selection to Original Source View

The Coverage Viewer displays the line numbers of the original source file on the left side of the viewer. If both the Coverage Viewer and the Original Source are open concurrently, any line selection in one viewer will be reflected in the other viewer. If the CBA Editor is also open concurrently, the source line mapping is also enabled in that editor.



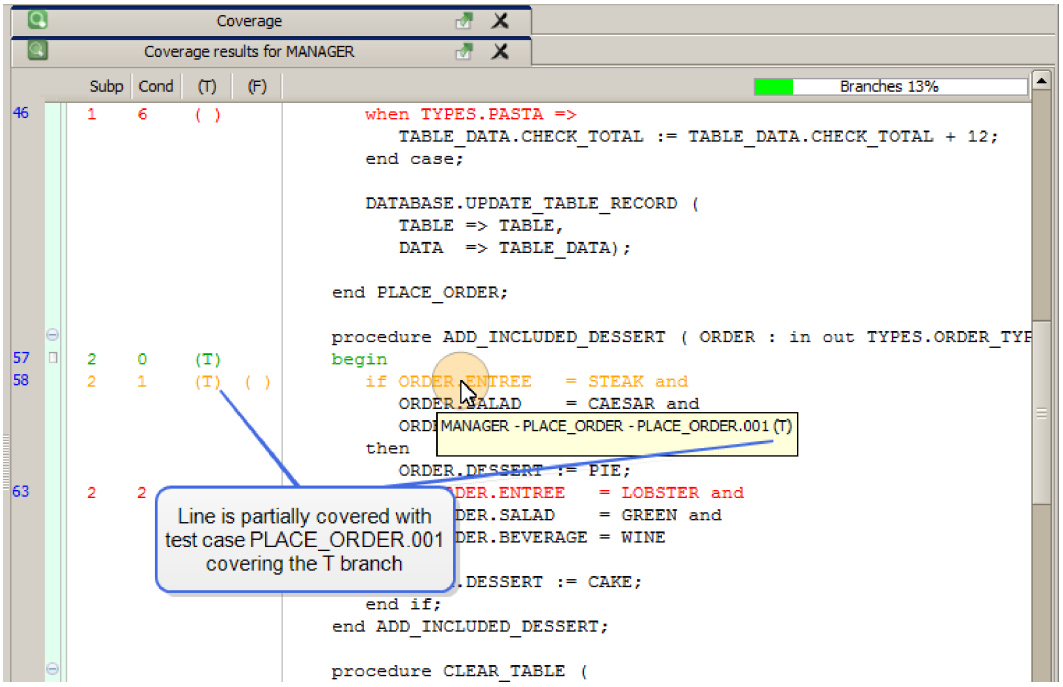
By default, selecting a line in the Coverage Viewer, Original Source text editor or CBA Editor will automatically scroll to the corresponding line in any other open editor. To change the behavior, right-click within the Coverage Viewer or CBA Editor and uncheck the **Map Selection to Original Source View** option on the context menu.



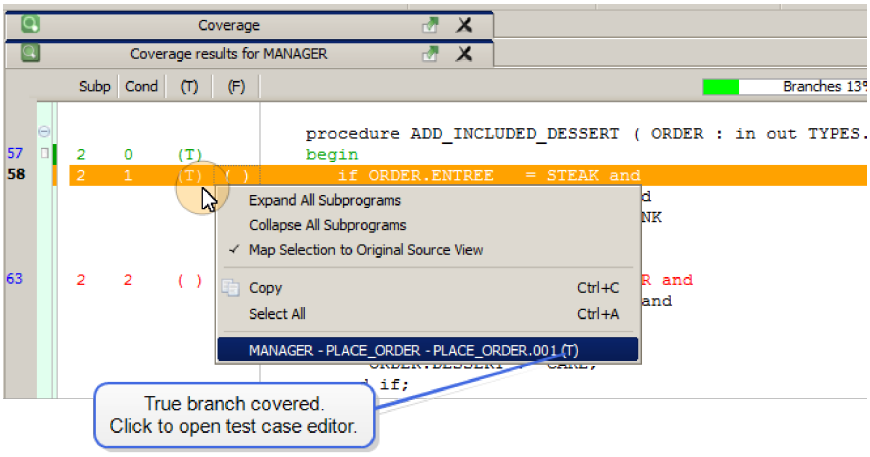
Alternatively, in the Original Source text editor, right-click and uncheck the **Map Current Line to Coverage View** option on the context menu.

To Open a Test Case That Covers a Line

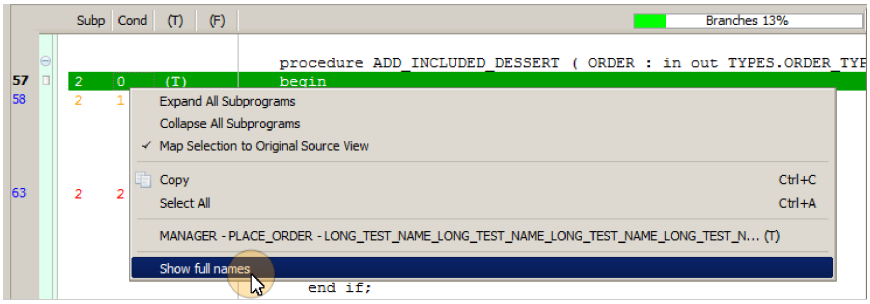
When viewing code coverage, if you hover the cursor over a covered (green) or partially-covered (yellow) line, a pop-up list is displayed that shows the names of all test cases that caused that line to be executed. Hovering over a branch will display (T) if the test case executes the True branch, (F) if the test case executes the False branch, and (TF) if the test case executes both the True and False branches.



Right-click a covered line in the Coverage Viewer to see a pop-up menu listing the test case(s) that cover it. True (T), False (F) or both True and False (TF) branch coverage is also indicated. To open that test case in the Test Case Editor, right-click, and select the test case name from the pop-up list.



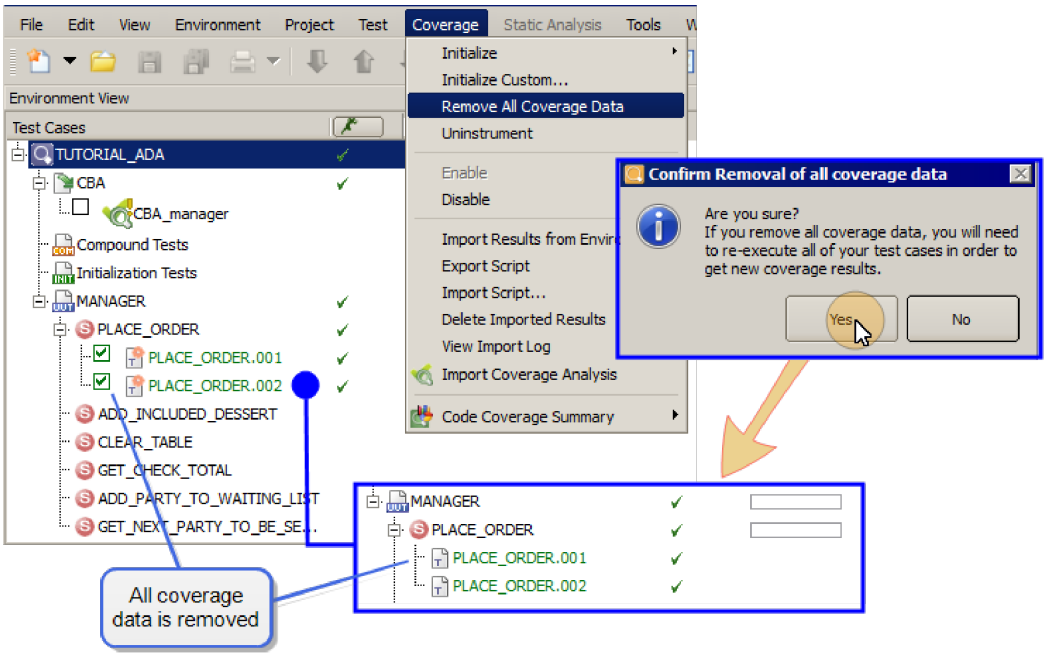
Note: Long test case names are truncated in the pop-up menu. Select the **Show full names** option to display the full test case name. This option is only available when test case names are truncated.



To Remove All Coverage Results

If you wish to discard all existing coverage data for an environment, choose **Coverage => Remove All Coverage Data**. A dialog appears asking for confirmation.

Clicking Yes in this dialog removes coverage checkboxes from the Test Case Tree, as shown below. Coverage remains initialized.



To View the Aggregate Coverage Report

An Aggregate Coverage Report includes for each unit selected in the Test Case Tree:

- an annotated source-listing, indicating the lines covered
- a metrics table giving the complexity and the level of coverage achieved for each subprogram

The report is sensitive to what level is selected in the Test Case Tree. If you select the top-level node for the environment, the report includes coverage achieved by test cases executed in all UUTs and non-stubbed units that have coverage instrumented using Custom Initialize. You can narrow the

content down by selecting particular units in the Test Case Tree for which you want coverage achieved. Once you have made a unit selection, choose **Test => View => Aggregate Coverage Report**.

The report contains coverage data from all test cases within the selection, regardless of whether they are toggled on to show coverage or not.

The annotated source code is color-coded to mark the exercised lines (green), the un-exercised lines (red), lines covered by analysis (blue) and partially exercised lines (yellow), similar to what is seen in the Coverage Viewer.

```
clicast -e <env> [-u <unit>] REports Custom Coverage <outputfile>

Create an Aggregate Coverage report and output to HTML file <env>_
aggregate_coverage_report.html, standard output as text, or to the
specified output file. If the Unit parameter is specified, the report
includes the annotated source code for the specified unit, and the Metrics
table includes only the specified unit; otherwise, the report includes the
annotated source code for all units and the Metrics table includes all
units in the environment.
```

To View the Metrics Report

A Metrics Report includes a metrics table giving the complexity and the level of coverage achieved for each subprogram in the selected unit(s) for each unit selected in the Test Case Tree.

The report is sensitive to what level is selected in the Test Case Tree. If you select the top-level node for the environment, the Metrics table includes coverage achieved by test cases executed in all UUTs and non-stubbed units that have coverage instrumented using Custom Initialize. You can narrow the content down by selecting particular units in the Test Case Tree for which you want coverage achieved. Once you have made a unit selection, choose **Test => View => Metrics Report**.

The report contains coverage data from all test cases within the selection, regardless of whether they are toggled on to show coverage or not.

If coverage data is not available, then only the Unit, Subprogram, and Complexity columns are present in the Metrics Report.

```
clicast -e <env> [-u <unit>] Reports Custom Metrics <outputfile>

Create a Metrics report and output to HTML file <env>_metrics_report.html,
standard output as text, or to the specified output file. If the Unit
parameter is specified, the Metrics table includes only the specified unit;
otherwise, the Metrics table includes all units in the environment.
```

If the report format is HTML, the Metrics Report looks similar to the following:

Metrics Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	27 MAR 2020
Time of Report Creation	2:38:09 PM

Metrics

Statement

Unit	Subprogram	Complexity	Statements
MANAGER	PLACE_ORDER	5	8 / 12 (66%)
	ADD_INCLUDED_DESSERT	3	2 / 3 (66%)
	CLEAR_TABLE	2	0 / 7 (0%)
	GET_CHECK_TOTAL	1	0 / 2 (0%)
	ADD_PARTY_TO_WAITING_LIST	2	0 / 5 (0%)
	GET_NEXT_PARTY_TO_BE_SEATED	6	0 / 12 (0%)
TOTALS	**6**	**19**	**10 / 41 (24%)**
GRAND TOTALS	**6**	**19**	**10 / 41 (24%)**

Pairs Status

When MC/DC or STATEMENT+MC/DC coverage is initialized, an additional “Pairs” column is added to the Metrics Report. MC/DC Pairs consists of total number of MC/DC pairs satisfied compared to the total number of MC/DC pairs in the unit.

If the report format is HTML, the report looks similar to the following:

Metrics Report

Configuration Data

Environment Name	TUTORIAL_ADA
Date of Report Creation	27 MAR 2020
Time of Report Creation	2:42:06 PM

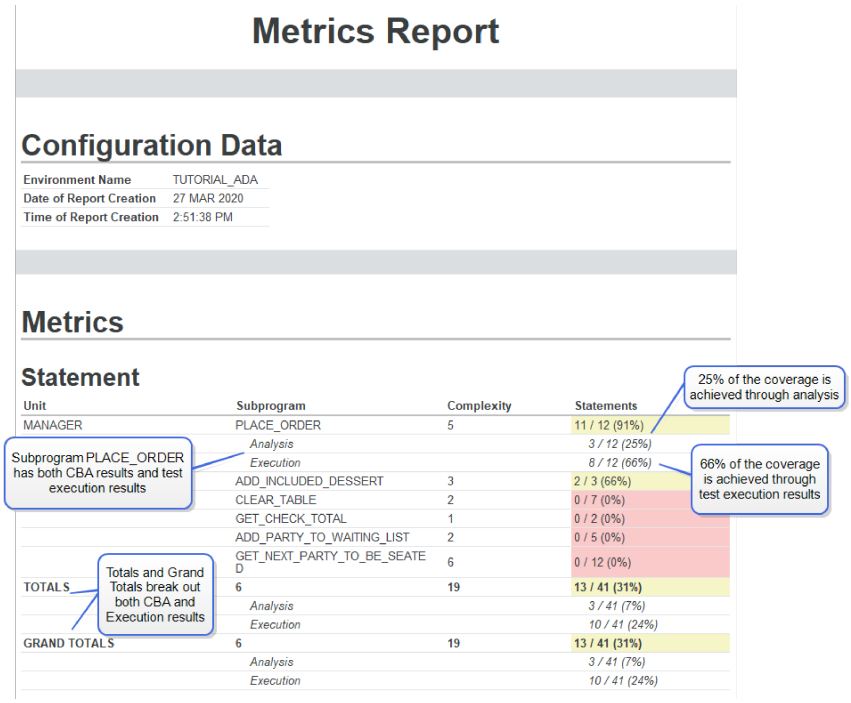
Metrics

Statement+MC/DC

Unit	Subprogram	Complexity	Statements	Branches	Pairs
MANAGER	PLACE_ORDER	5	9 / 17 (52%)	2 / 6 (33%)	
	ADD_INCLUDED_DESSERT	3	2 / 4 (50%)	5 / 17 (29%)	0 / 6 (0%)
	CLEAR_TABLE	2	0 / 7 (0%)	0 / 3 (0%)	
	GET_CHECK_TOTAL	1	0 / 2 (0%)	0 / 1 (0%)	
	ADD_PARTY_TO_WAITING_LIST	2	0 / 5 (0%)	0 / 5 (0%)	0 / 1 (0%)
	GET_NEXT_PARTY_TO_BE_SEATED	6	0 / 13 (0%)	0 / 19 (0%)	0 / 4 (0%)
TOTALS	**6**	**19**	**11 / 48 (22%)**	**7 / 51 (13%)**	**0 / 11 (0%)**
GRAND TOTALS	**6**	**19**	**11 / 48 (22%)**	**7 / 51 (13%)**	**0 / 11 (0%)**

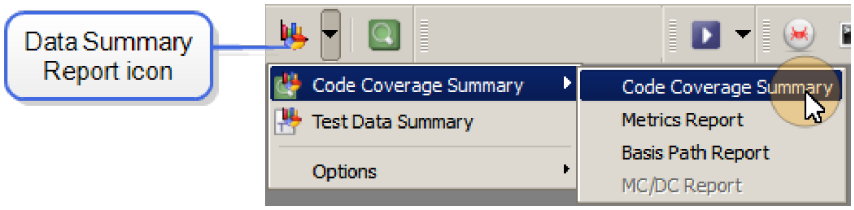
Covered By Analysis Results

In the Metrics Report, when CBA results are present in the environment, they are displayed in italics in the row below the subprogram and the coverage achieved by test execution is displayed below the CBA results.



To View Code Coverage Summary

The Code Coverage Summary table allows the user to easily view coverage information for all the units in an environment or all the environments of a Manage project. To open the summary from the Toolbar, select **Code Coverage Summary** from the Data Summary Report icon drop-down menu.

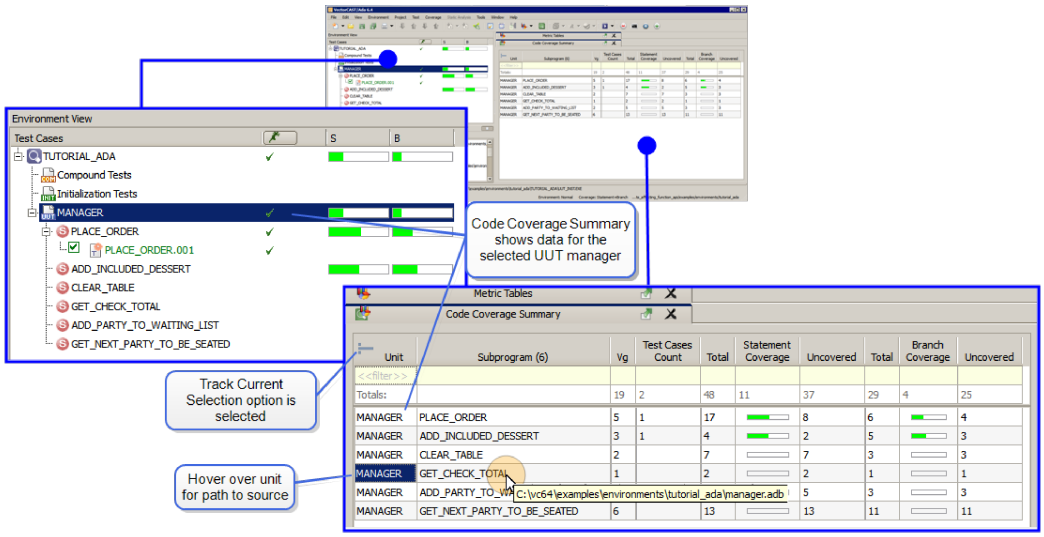


Alternatively, from the Menu Bar, select **Coverage => Code Coverage Summary => Code Coverage Summary**.

Viewing Code Coverage

By default, the contents of the Code Coverage Summary reflect the UUTs selected in the Test Case Tree. A tracking icon is displayed at the top of the Unit column indicating that the Code Coverage Summary is currently tracking selected UUTs from the Test Case Tree.

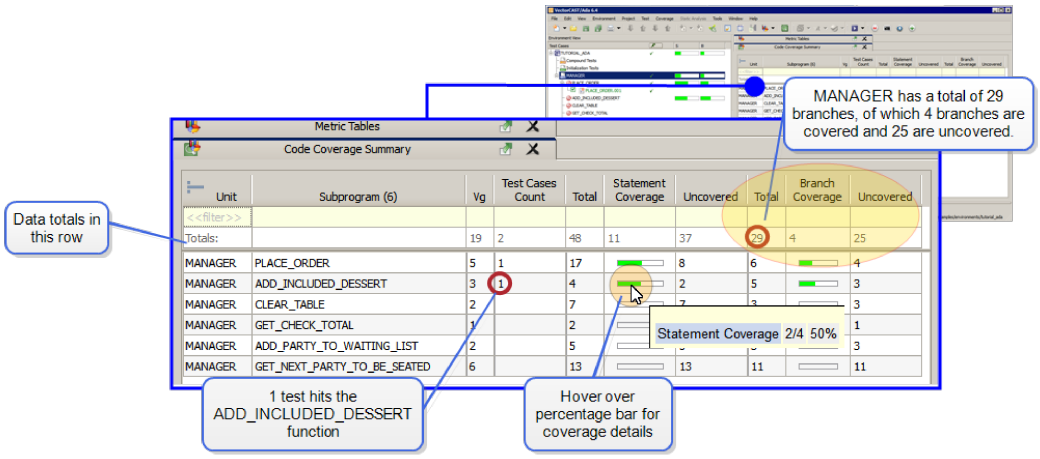
To override tracking of selected UUTs in the Test Case Tree, open the drop-down menu for the Data Summary Report and select **Options => Track Current Selection**. Remove the check next to the option. The tracking icon on the Summary table will change to gray to indicate that the summary is now tracking all units in the Test Case Tree.



The Code Coverage Summary table is dynamic. When the **Track Current Selection** option is enabled, as units are selected and deselected in the Test Case Tree, the Code Coverage Summary table updates in real time reflecting the selections.

The data displayed in the Code Coverage Summary includes the Unit name, the Subprogram name, the Cyclomatic Complexity (Vg), the Test Cases Count (showing the number of unique test cases that have hit a particular subprogram during execution, with compound tests being counted as one test regardless of the number of slots in the compound), and the achieved coverage for each coverage type.

Note: When I/O type is set to "Animation" (e.g. when using Basis Path coverage), the Test Cases Count column indicates the number of times a function is entered. This total can also include multiple slot iterations of a compound test.



Unit	Subprogram (6)	Vg	Test Cases Count	Total	Statement Coverage	Uncovered	Total	Branch Coverage	Uncovered
<<filter>>									
Totals:		19	2	48	11	37	29	4	25
MANAGER	PLACE_ORDER	5	1	17		8	6		4
MANAGER	ADD_INCLUDED_DESSERT	3	1	4		2	5		3
MANAGER	CLEAR_TABLE	2		7					3
MANAGER	GET_CHECK_TOTAL	1		2					1
MANAGER	ADD_PARTY_TO_WAITING_LIST	2		5					3
MANAGER	GET_NEXT_PARTY_TO_BE_SEATED	6		13		13	11		11

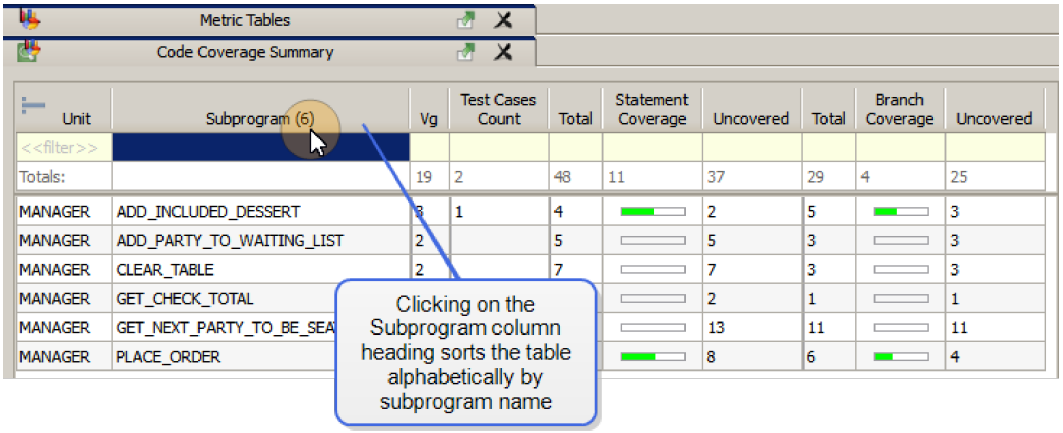
The Totals row at the top of the table displays the totals for each data column. In the example above, note that in the unit MANAGER, we have a total of 29 branches of which 4 branches are covered and 25 are uncovered.

The Summary table updates whenever coverage data is updated. For example, the table refreshes when coverage is initialized, or following test execution.

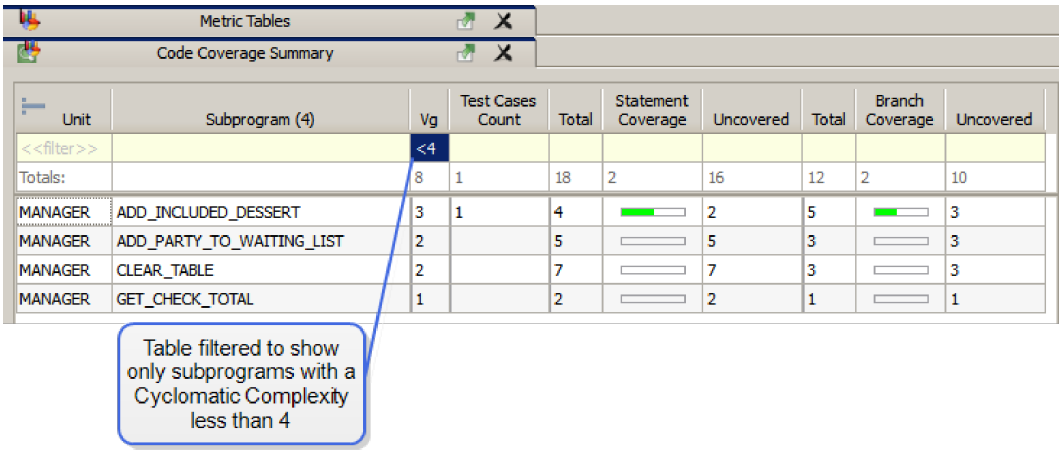
Double-clicking on a line in the Code Coverage Summary opens the corresponding UUT in the Coverage Viewer.

Sorting and Filtering Coverage Data

Sorting and filtering is available to locate data of interest. Sort by clicking on any column heading. The data will sort in alphabetic or numeric order, as appropriate. Clicking the heading again reverses the order.



Coverage data can be accessed all the way down to the individual subprogram level by filtering. Access the filter by typing into the top row of any column. Clear the filter by right-clicking in the top row and selecting **Clear Filter** from the context menu. In the example below, the table has been filtered to only show subprograms with Cyclomatic Complexity less than 4.



Filtering supports the following symbols: <, >, =. For example, the summary can be filtered to show only subprograms with a Cyclomatic Complexity greater than 2. Other examples of filtering inputs are:

10 - lists subprograms matching the specific value of "10" in the selected column
>50 - lists subprograms greater than the value of "50" in the selected column
<90 - lists subprograms less than the value of "90" in the selected column
=100 - lists subprograms matching the specific value of "100" in the selected column
< - lists subprograms with empty values in the selected column
> - lists subprograms with non-empty values in the selected column
= - lists subprograms with non-empty values in the selected column

Saving and Printing Code Coverage Summary

Saving the Code Coverage Summary in .html or .csv format is supported. See "To Save a Script or

Text File " on page 82 for more information.

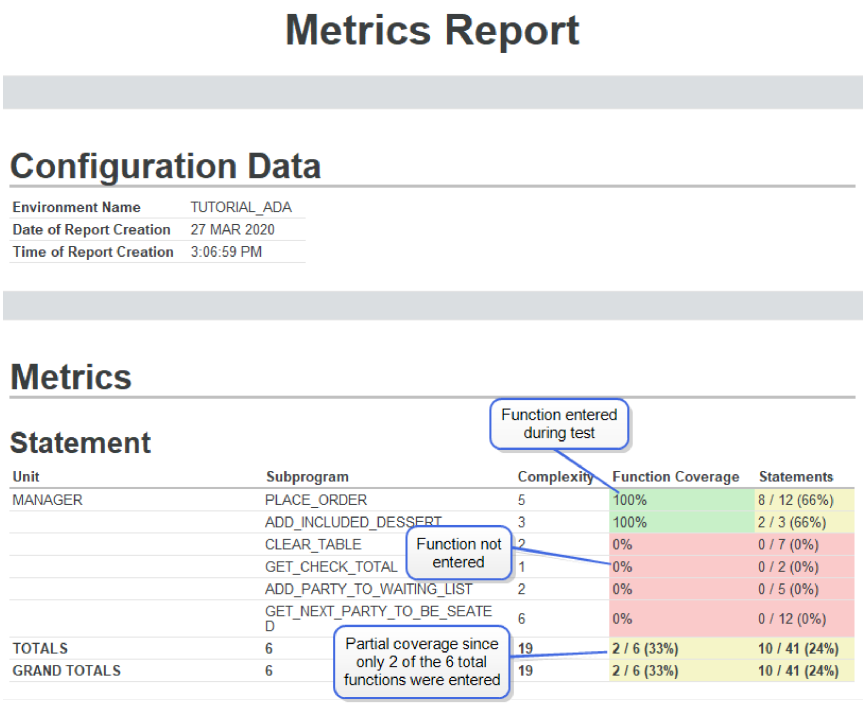
Printing the contents of the Code Coverage Summary is supported. See "To Print an Open Window" on page 83 for more information.

To View Function Coverage Results

When the Function Coverage option is enabled, the Metrics Report displays a Function Coverage column indicating whether or not a subprogram has been entered during a test.

To enable the Function Coverage option, select **Tools => Options** from the Menu Bar. Select the **Coverage => Options => Instrumentation Options** tab. Check the box for 'Always display function coverage' and select the **Apply** button.

In the Function Coverage column, 100% coverage is displayed in green and no coverage is displayed in red.



The Function Coverage column is displayed in each report instance that includes the Metrics report (Full Report, Test Case Management Report, and Aggregate Coverage Report).

Understanding Basis Paths

The output from the Basis Path Analysis consists of two parts: the list of basis paths and an associated annotated source listing. The following example shows a single basis path along with the corresponding annotated source listing for the example.

```
  Test Path 1
         (1) if ( value < min ) ==> FALSE
         (3) if ( value > max ) ==> FALSE
  Test Path 2
         (1) if ( value < min ) ==> FALSE
         (3) if ( value > max ) ==> TRUE
  Test Path 3
         (1) if ( value < min ) ==> TRUE

    Complexity: 3

  Source Code Listing

                int Compute (int min, int max, int value) {
                    int return_value;
1 0     ( )   Compute
1 1     ( )( )    if ((value < min)){
                        return_value = min;
                    }
                    else
1 3     ( )( )       if ((value > max)){
                          return_value = max;
                       }
                       else{
                          return_value = value;
                       }
                    return return_value;
                 }
```

To Build Test Cases from Basis Path Analysis

When using the basis path analysis to build test cases, you should build at least one test case for each basis path ("Test Path"). The test case should contain the collection of data that will set the conditions specified in the basis path listing.

Using the above example, Test Path 1 specifies that **`value < min`** should be False, and **`value > max`** should also be False. If we use 4 for **`min`** and 10 for **`max`**, setting **`value`** to 5 satisfies this Test Path.

Test Path 2 specifies that **`value < min`** should be False, and **`value > max`** should be True. If we use 4 for **`min`** and 10 for **`max`**, setting **`value`** to 11 satisfies this Test Path.

Test Path 3 specifies that **`value < min`** should be True. We can use 4 for **`min`** and 3 for **`value`**. There is no need to set **`max`**; it is irrelevant.

Executing these three test cases gives us 100% branch coverage and 100% basis paths coverage.

VectorCAST can automatically build the basis path test cases. For details, see "To Insert Basis Path Test Cases" on page 164

To View a Basis Paths Report

The **Tools => Basis Path => View Basis Path** command enables you to view the results of the computation of test paths for test case building. The contents of this report reflect the items selected in the Test Case Tree.

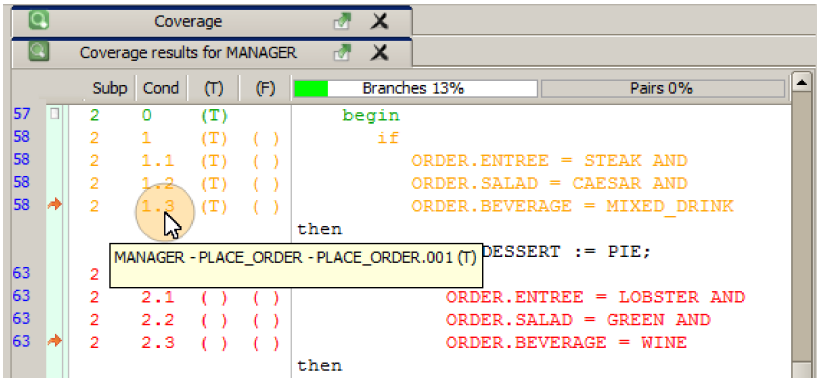
The number of paths identified equals the code complexity, also referred to as V(g). The code complexity corresponds to the number of test cases that must be developed to exercise the unique paths in a subprogram.

This information can also be viewed by accessing the Basis Paths tab of the Coverage window.

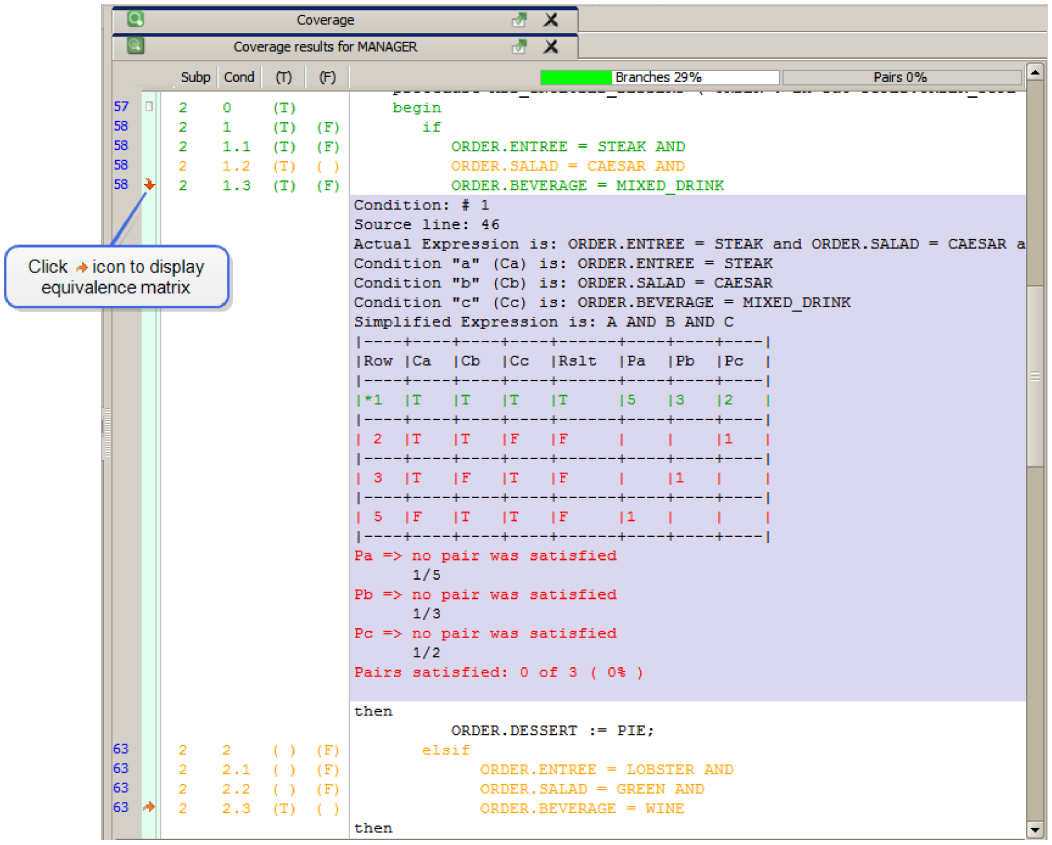
```
clicast -e <env> [-u <unit>] TOols Basis_path [<outputfile>]

Extract the basis path analysis for all units or the specified units to
standard output or the specified file. If VCAST_CUSTOM_REPORT_FORMAT is
HTML and <outputfile> is not specified, the file is saved to <env>_basis_
path_report.html.
```

MC/DC Coverage

When you have the Coverage Viewer open, and MC/DC or STATEMENT+MC/DC coverage is initialized, if you hover your cursor over any fully or partially covered line of code, a tooltip displays the test case(s) which cover the line.



For MC/DC or STATEMENT+MC/DC coverage, the left margin includes a red arrow that, when clicked, opens the Equivalence Matrix below the associated subprogram in the Coverage Viewer.



Clicking the arrow again closes the Equivalence Matrix. The matrices can also be viewed by selecting **Tools => MC/DC => View Equivalence Matrices** and viewing the MC/DC Condition Tables Report. See"To View the Equivalence Matrices Report" on page 354 for more information.

Understanding MC/DC Analysis

The output from the MC/DC analysis consists of two parts: an equivalence pair matrix, and an annotated source listing.

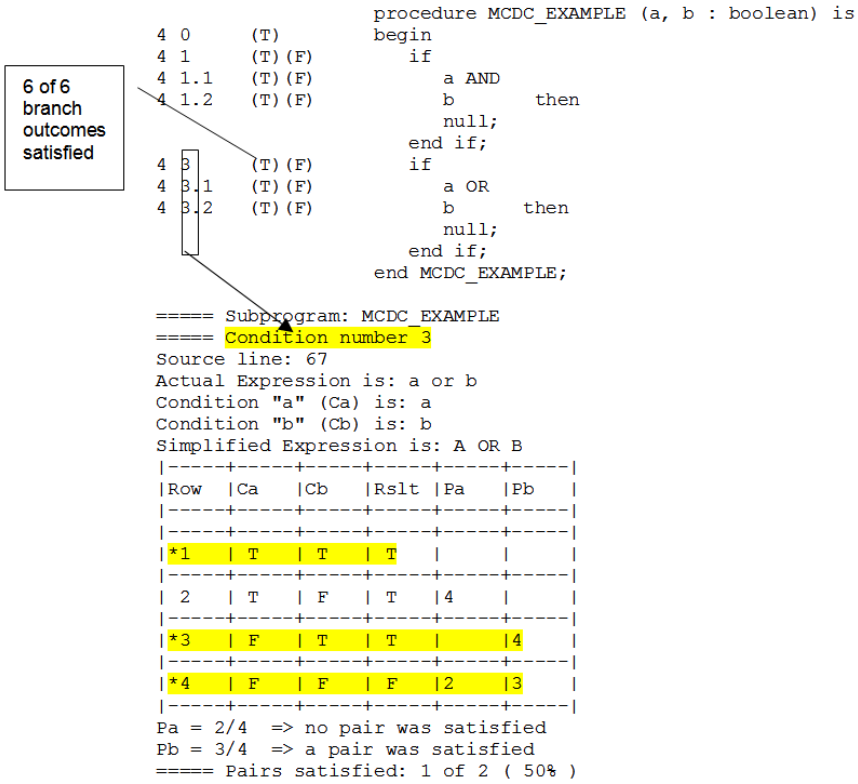
For each Boolean conditional there is a corresponding matrix. Each row in the matrix shows a unique combination of values for the components of the complex conditional, as well as the result of the condition for each combination. The equivalence pairs are pairs of rows which demonstrate that the result of the conditional changes from True to False as one component changes, while all other components are held constant.

The Equivalence Pair Matrix shows both the static pair analysis and the pair coverage achieved. The Pair A through Pair *n* columns (where *n* is the number of sub-conditions) show candidate row pairs for proving condition components. An asterisk next to a row number indicates that the row has been covered. When an equivalence pair has been covered, the Pa through P*n* summary lines reflect that information.

The Equivalence Pair Matrix shows only rows having equivalence pair information. Rows that do not have any pair infomation are eliminated from the display because they add no useful information.

The following example shows a single complex conditional with the corresponding equivalence pair

matrix.



This source listing shows that the “if” condition has been executed with A=True, B=True (Row 1); A=False, B=True (Row 3); and A=False, B=False (Row 4). The MC/DC branch coverage is 6/6. Six out of a total of six outcomes have been satisfied, and the pair coverage is 1 of 2 pairs satisfied.

To View the Equivalence Matrices Report

Choose **Tools => MC/DC => View Equivalence Matrices** to display the MC/DC Condition Tables report. This report includes the equivalence matrices for all units instrumented for MC/DC coverage or DO-178B Level A coverage in the environment as well as the MC/DC equivalence pair status.

The MC/DC Condition Tables report shows only rows having equivalence pair information. When VectorCAST instruments for MC/DC or Statement+ MC/DC coverage, it calculates the Equivalence Pair matrix. Each row represents a combination of possible outcomes for each condition in the expression, and the resulting output. A pair is a set of two rows which demonstrate that the result of the conditional (Rslt column) changes from True to False as one component (Ca or Cb) changes, while all other components are held constant.

For example, in the matrix below, rows 1 and 3 form a pair, and rows 1 and 2. However, row 4 does not form a pair with any other row because changing one of the components, such as Ca, from F to T does not change the result of the conditional (F) to True.

```
Source line: 179
```

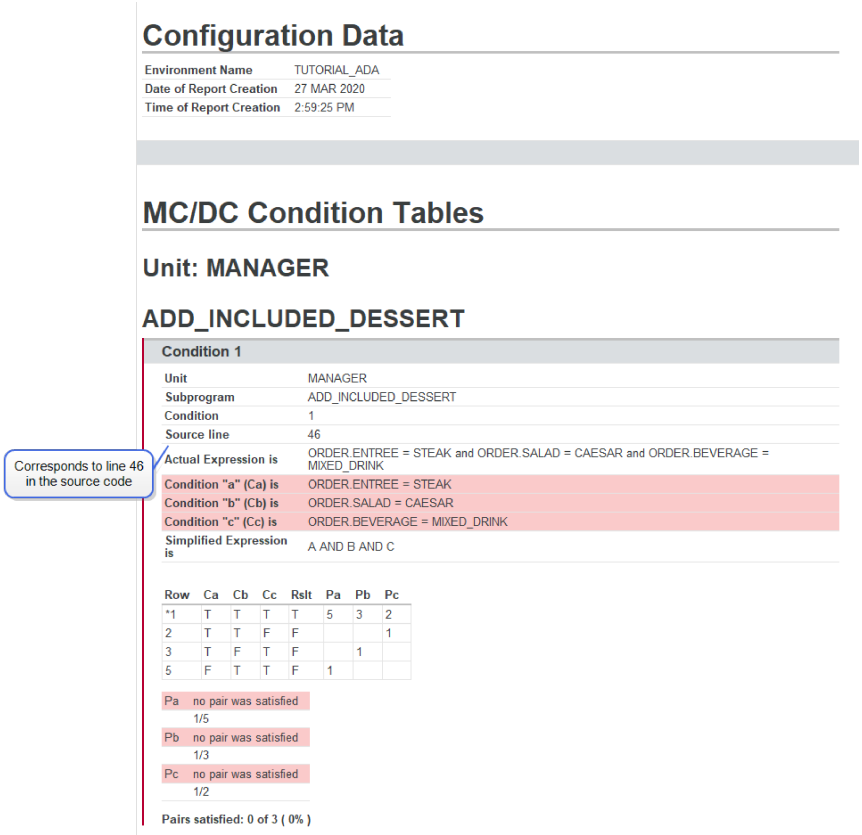
```
Actual Expression is: ( p && q )
Condition "a" (Ca) is: p
Condition "b" (Cb) is: q
Simplified Expression is: ( a && b )
|-----+-----+-----+-----+-----+-----|
|Row |Ca  |Cb  |Rslt |Pa  |Pb  |
|-----+-----+-----+-----+-----+-----|
|-----+-----+-----+-----+-----+-----|
| 1  | T  | T  | T  |3   |2   |
|-----+-----+-----+-----+-----+-----|
| 2  | T  | F  | F  |    |1   |
|-----+-----+-----+-----+-----+-----|
| 3  | F  | T  | F  |1   |    |
|-----+-----+-----+-----+-----+-----|
|4    | F  | F  | F  |    |    |
|-----+-----+-----+-----+-----+-----|
```

VectorCAST displays the rows that do not have any pair information as having blank boxes under the Pair column (Pa or Pb, in the example). These rows are eliminated from the display of the Equivalence Pair matrix because they add no useful information.

clicast -e <env> [-u <unit>] Reports Custom Equivalence_matrices [<outputfile>]

Extract MC/DC Equivalence matrices for all units or the specified unit to standard output or the specified output file. If VCAST_CUSTOM_REPORT_FORMAT is HTML and <outputfile> is not specified, the file is saved to <env>_mcdc_tables_report.html.

If the report format is HTML, the report looks similar to the following:



The **Source line #** refers to the source code line in the Coverage Viewer that corresponds to the expression being evaluated in the matrix. After noting the line number, right-click the unit in the Test Case Tree, and choose **View Coverage**. Then, in the Coverage Viewer, type **Ctrl+F** (or choose **Edit => Find** from the Menu Bar), and type in the line number.

Viewing Execution Flow with Animated Coverage

Coverage Animation allows you to view the flow of execution for a test case in the Coverage Viewer. For example, you can view individual statements being executed as the test executes. If a unit calls a subprogram, you can view the execution flow to the called subprogram.

Coverage Animation displays the flow of control from one unit to another, using the test's coverage data. Calls to stubbed units are not included in Animation. Not-stubbed units are included if they have coverage initialized using **Coverage => Initialize Custom**.

Coverage can be animated for only one test case in the Test Case Tree at a time. The item currently activated for Animation has a check in the box to the left of its name in the Test Case Tree. If you check another item, the Animation toolbar dims until you activate another test result for animation.



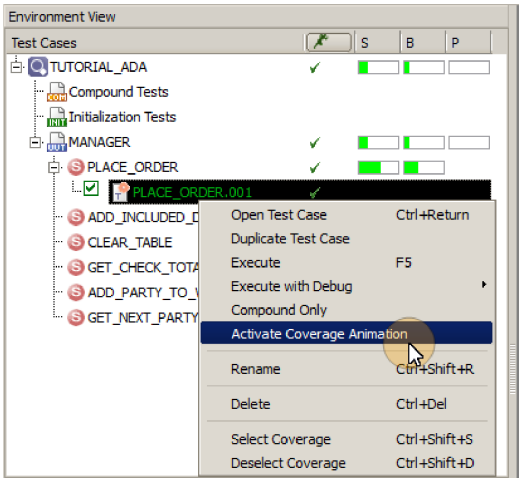
Note: If the Animation toolbar is dim, activate a test case for animation.

To Activate Coverage Animation

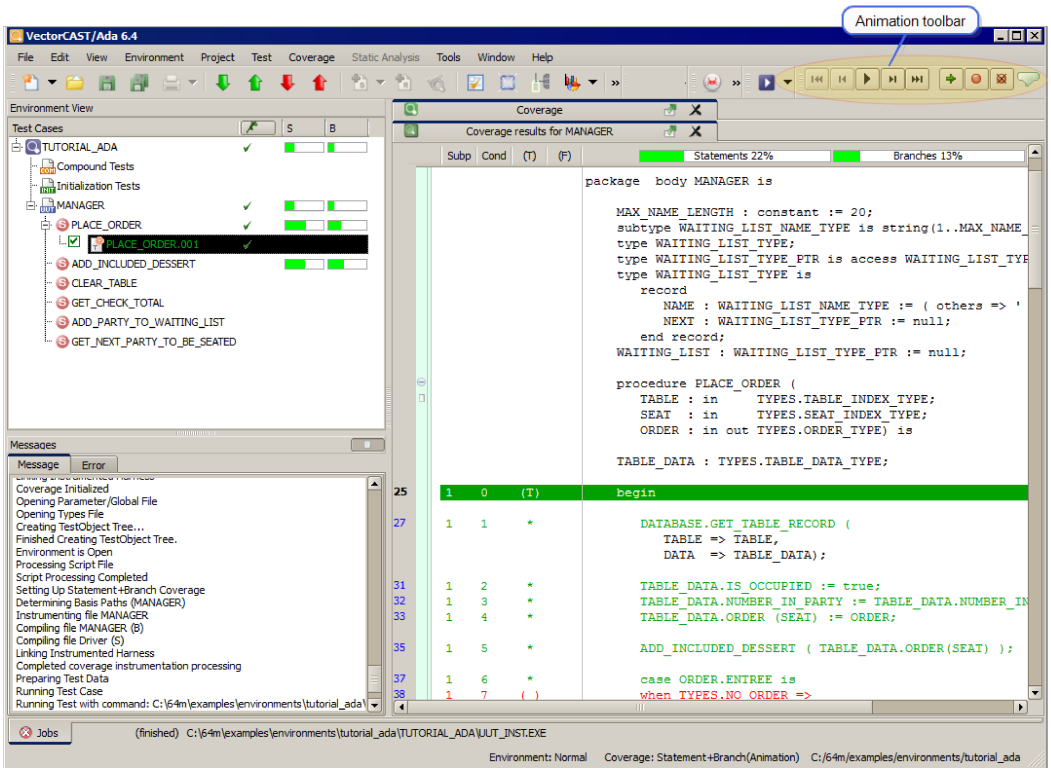
To use the Animation feature, select **Tools => Options** dialog, Coverage tab, and then select Animation for the **Coverage I/O Type**.

After setting the Coverage I/O type to Animation, instrument for a type of coverage. If you want to see each statement being executed, you would instrument for Statement coverage; if you want to see the choice made at each branch point, you would instrument for Branch coverage. To see each entry point, decision point, and statement covered, choose MC/DC, STATEMENT+MC/DC, or STATEMENT+BRANCH.

Next, execute a test case. Right-click the test case in the Test Case Tree, and then choose **Activate Coverage Animation** from the popup menu:



The Coverage Viewer for the unit containing the first covered line in the test result opens, with the current line on the first covered line. The coverage checkbox next to the test case is checked.



If not already open, the Coverage Animation toolbar (pictured below) appears in the main toolbar.



To Play the Coverage Animation

Click the **Play** button  to begin the animation. In the Coverage Viewer, the current line progresses steadily from the first covered line to the last. The speed of each step is 1 second. The source code in the Coverage Viewer scrolls up, keeping the current line in the middle. If a subprogram in another unit is called and that unit has coverage initialized, then its tab comes to the front of the Coverage Viewer and animation continues in that tab.

In the Coverage Viewer:

- the blue arrow  indicates the current line in the animation of control flow
- the green arrow  indicates the first line
- the red arrow  indicates the last line

The Animation Toolbar

When a test case is activated for animation, the Animation toolbar is displayed in the main toolbar. To display or hide the Animation toolbar manually, right-click the main toolbar and toggle **Coverage**

Animation. If there is a checkmark, the toolbar is visible; otherwise it is hidden. Alternatively, choose **View => Dock Control** and toggle the setting for Coverage Animation.

The following controls are available on the Coverage Animation toolbar:

	**Rewind**	Returns the current-line indicator to the first executed line.
	**Back**	Returns the current-line indicator to the previous executed line.
	**Play**	Starts dynamic display of the execution; continues dynamic display after an interruption. See Pause and Breakpoint.
	**Pause**	Pauses dynamic display of the execution. See Continue and Play.
	**Next**	Moves the current-line indicator to the next executed line.
	**Fast Forward**	Moves the current-line indicator to the last line executed.
	**Continue**	Jumps the current-line indicator to the next breakpoint or to the last line of animation.
	**Set Breakpoint**	Sets a breakpoint at the current line.
	**Remove Breakpoint**	Removes a breakpoint from the current line.
	Progress Indicator / Scroll Knob	Indicates the relative position of the current line during dynamic display of the execution. Drag this knob to set the position of the current line.

To Set a Breakpoint

Setting a breakpoint causes the animated display of the execution flow to pause on the line marked with the breakpoint  . To set a breakpoint, select the line to make it current, then click the **Set Breakpoint** button

.

From here you can:

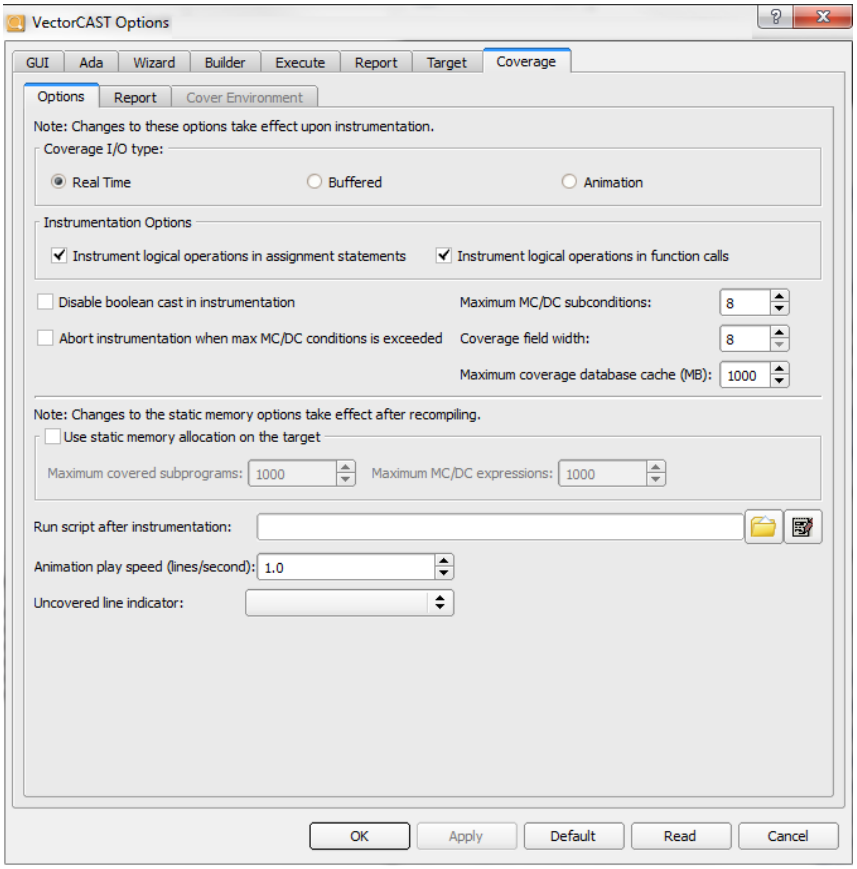
- play from this point 
- step to the next line in the flow of execution 
- jump to the next breakpoint  (or the end, if there is none)
- remove the breakpoint 

There is no limit to the number of breakpoints you can set.

Setting Coverage Options

Options: Coverage Tab

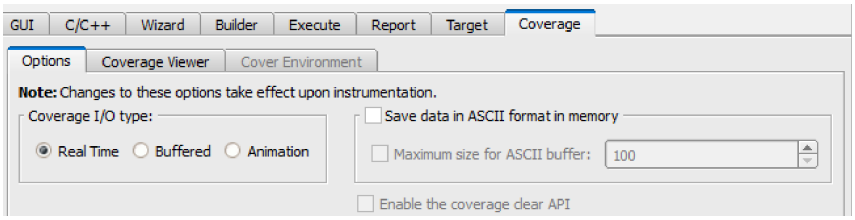
The Coverage tab of the **Tools => Options dialog** enables you to control the fonts and colors used in the Coverage Viewer. By default, covered lines are shown in green, uncovered lines are shown in red, and non-executable statements are shown in black. It is recommended that you use a monospaced font for the Coverage Viewer (e.g. Courier) as that allows the tables and report to be aligned properly.



See also "To Set VectorCAST Options" on page 242.

Coverage I/O Type

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.



Three coverage I/O types are supported: Real-time, Buffered, and Animation. Changes to the Coverage I/O type take effect upon coverage instrumentation.

- **Real-time:** Default. Coverage data is gathered as the test runs, and output to the results file immediately. This I/O type is optimized.
- **Buffered:** Coverage data is stored and then output when the test finishes executing. This method is useful is your target platform is slow. This I/O type is optimized. Buffered I/O gives better performance, but requires more memory than Real-time coverage I/O.

> **Animation:** This I/O type is not optimized. Instead, coverage data is gathered in the order encountered, in preparation to animate the test result's coverage in the Coverage Viewer. The animated coverage results reveal the flow of control.

After changing this option, you must reinitialize coverage.

```
clicast option COVERAGE_IO_TYPE Real_Time | Buffered | Animation

Type of coverage I/O that the instrumented harness will perform. The
default value is Real_Time.
```

Maximum MC/DC Subconditions

The "Maximum subconditions for MC/DC table pre-calculation" option enables you to limit the application of MC/DC analysis to conditions that contain a particular number of sub-conditions. The default value for this option is 8. Any statement containing more than 8 conditions will not be analyzed. The maximum value for this option is 26.

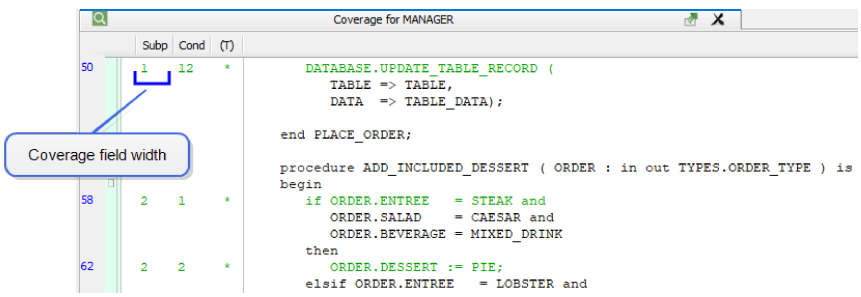
After changing this option, you must reinitialize coverage.

```
clicast -lada option VCAST_MAX_MCDC_CONDITIONS <integer number>

VectorCAST generates equivalence matrices for up to this many subconditions
in an MC/DC expression. (Example: (A || !B || C) has operands, so it has
three subconditions.) If an expression exceeds this limit, equivalence
pairs are calculated as results are added to the environment, and only
table rows that contribute to a satisfied pair are displayed. MC/DC tests
are not created for expressions exceeding the limit. Its default value is
8.
```

Coverage Field Width

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab if it is not selected.

The "Coverage field width" option specifies the width, in characters, of the far left column in the Coverage tab in the Coverage Viewer. Increase this number if you have a large number of subprograms in the unit or a subprogram with a large number of statements or branches.



After changing this option, you must reinitialize coverage.

```
clicast option VCAST_COVERAGE_FIELD_WIDTH <integer number>

The number of digits to allow for the numeric identifier displayed in the
left column of a coverage report. The default is 8. Increase this number if
you have a large number of subprograms or a subprogram with a large number
of statements or branches. Its default value is 8 characters.
```

Maximum Coverage Database Cache

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab.

This option allows you to adjust the coverage database cache size. Increasing the value may increase performance, but care should be taken not to set the value greater than available memory. Changes to this option take effect when re-opening an environment.

```
clicast option VCAST_COVERAGE_DATABASE_CACHE_SIZE <integer number>

Increasing the maximum coverage database cache size may increase
performance, but take care not to set it to a value greater than the amount
of available system memory. For a system with 2GB, a 1000 MB maximum cache
is probably sufficient. Changes to this option take effect when reopening
an environment.
```

Abort Instrumentation when 'Max MC/DC Conditions' is Exceeded

Stop instrumenting a unit for MC/DC or DO-178B Level A when the maximum number of MC/DC Subconditions is exceeded.

After changing this option, you must reinitialize coverage.

```
clicast option VCAST_MAX_MCDC_ABORT_INST True | False

 While instrumenting with MC/DC, if a conditional statement contains more
conditions than possible to instrument, then abort instrumenting that file.
For Ada, the maximum is 26.
```

Instrument Logical Operations in Assignment Statements

By default, branch and MC/DC coverage is implemented on conditional statements, such as **`if`** statements, and not implemented on assignment statements. When this option is on, VectorCAST will instrument branch and MC/DC coverage on assignment statements with boolean operators or the conditional operator "?". Comparison operators in C, such as "==", are also considered logical operators.

You can override the default behavior on a statement-by-statement basis, so that coverage is disabled on a conditional statement that would, by default, have instrumentation applied, or enabled on assignment statements that would by default, not have instrumentation applied.

To enable coverage for a specific statement, precede the statement with the comment **VCAST_DO_MCDC**. The comment must exist on lines with no other text, immediately preceding the source code line for which you want to modify the coverage. The text can be upper- or lowercase, but there can be no spaces after the comment characters.

After changing this option, you must reinitialize coverage.

This option was formerly named “Cover assignment statements in MC/DC.”

Instrument Logical Operations in Function Calls

When this option is set, VectorCAST performs Branch and MC/DC analysis on function calls that contain boolean expressions as parameters. For example, instrumenting source code with a function call like this for branch coverage:

```
foo(a&&b);
```

will have branch coverage instrumented on the logical expression "a&&b".

```
clicast option VCAST_INSTRUMENT PARAMETERS [True | False]

Setting this option will cause parameters in function calls to be covered
by branch and MC/DC coverage. The default value is True.
```

Disable Boolean Cast in Instrumentation

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab if it is not selected.

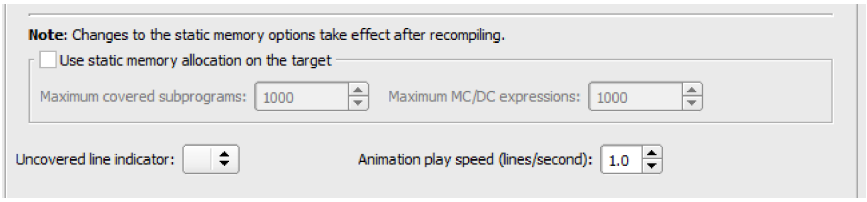
Set this option to prevent VectorCAST from casting conditional expressions to boolean when instrumenting for Branch or MC/DC coverage.

After changing this option, you must reinitialize coverage.

```
clicast -lada Option VCAST_DISABLE_BOOLEAN_CAST True | False

Select this option to prevent VectorCAST from casting conditional
expressions to boolean when instrumenting for branch or MC/DC coverage. The
default value is False.
```

Use Static Memory Allocation on the Target

Choose **Tools => Options** and click the **Coverage tab**. Then click the **Options tab** if it is not selected.



When the “Use Static Memory Allocation on the Target” option is checked, VectorCAST uses a static memory model when allocating memory for coverage data. By default, the instrumentation routines allocate memory as needed (dynamic). If you choose the static memory model, you can specify limits for the maximum number of subprograms and MC/DC expressions for which memory should be pre-allocated.

Note: After changing this option, you must recompile.

```
clicast option VCAST_USE_STATIC_MEMORY [True | False]

This option tells VectorCAST that only static data allocation can be used
on the target platform. This is used when collecting coverage data. If this
is set to True, then use the options VCAST_MAX_MCDC_STATEMENTS and VCAST_
MAX_COVERED_SUBPROGRAMS to configure the static allocation. Its default
value is FALSE.
```

Maximum Covered Subprograms

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab if it is not selected.

To set the “Maximum covered subprograms” option, you must check the box next to “Use static memory allocation” and use the Buffered Coverage I/O type.

The “Maximum covered subprograms” option tells VectorCAST how many unique subprogram calls it should allocate memory for. If a test case or compound test case execution encounters 500 different subprograms calls, and the option is set to 499 or lower, then a text error message is added to the coverage data indicating that the limit was reached. This text error message is used to generate popup messages and CLICAST output messages.

Note: After changing this option, you must recompile.

```
clicast option VCAST_MAX_COVERED_SUBPROGRAMS <integer number>

If "Use static memory allocation" is True and Coverage I/O is Buffered,
VectorCAST must allocate a specific amount of space to store coverage data
each time a different subprogram call is encountered. This number tells
VectorCAST how many different subprogram calls for which it should set
aside memory. Its default value is 1000.
```

Maximum MC/DC Expressions

Choose **Tools => Options** and click the **Coverage tab**. Then click the **Options tab** if it is not selected. To set the “Maximum MC/DC expressions” option, you must check the box next to “Use static memory allocation.” You can use any type of Coverage I/O with this option.

The “Maximum MC/DC expressions” option tells VectorCAST how many unique MC/DC expressions it should allocate memory for. If a test case or compound test case execution encounters 500 MC/DC expressions with unique values for its parameters, and the option is set to 499 or lower, then a text error message is added to the coverage data indicating that the limit was reached. This text error message is used to generate popup messages and CLICAST output messages.



Note: After changing this option, you must recompile.

```
clicast option VCAST_MAX_MCDC_STATEMENTS <integer number>

If "Use static memory allocation" is True, VectorCAST must allocate a
specific amount of space to store coverage data each time a unique MC/DC
expression is encountered. This number tells VectorCAST how many MC/DC
expressions for which it should set aside memory. Its default value is
1000.
```

Run Script After Instrumentation

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab if it is not selected.

The “Run script after instrumentation” option provides a means to execute a batch file (Windows) or shell script (Linux) once after each unit in the environment is instrumented. This feature is useful for performing post-processing on the unit’s instrumented source code. Specify the full or relative (to the environment directory) path to the batch file or shell script in the option. The full path to the unit is passed as an argument to the script.

A very simple example of a Windows batch file (run_script.bat) is:

```
echo %1 >> report.txt
type %1 >> report.txt
```



Once the path to the example script is specified in the “Run script after instrumentation” option and the environment is instrumented for coverage, this example script causes the file path to the unit and the instrumented file itself to be appended to the file **`report.txt`**, located in the environment directory. The first few lines of **`report.txt`** are shown below. These lines are repeated once for each unit in the environment.

```
c:\vcast\Environments\TUTORIAL_C\manager_inst.c
```

```
/* VectorCAST/Cover */
#ifndef VCAST_CONDITION_TYP
#define VCAST_CONDITION_TYP int
#endif
#ifdef __cplusplus
extern "C" {
#endif
#ifdef VCAST_USE_STD_STRING
#include <string.h>
#endif
/*
---------------------------------------
-- Copyright 2017 Vector Informatik GmbH --
-- North Kingstown, Rhode Island USA --
---------------------------------------
*/
...
```

If you have an external text editor defined on the GUI tab, clicking the Edit File button () opens the script in the specified external text edit.

clicast option VCAST_CMD_AFTER_INSTRUMENTATION <path to script>

This script will be run after VectorCAST has instrumented a file. The path to the source file is passed as an argument to the script. It has no default value.

Animation Play Speed

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

The “Animation play speed” option enables you to change the speed of coverage animation progress. By default, coverage animation proceeds at the rate of one line per second. To go faster, increase the value. To go slower, decrease the value. The ratio 1.0 represents 1 line / 1 second.

clicast option COVERAGE_ANIMATION_PLAY_SPEED_RATIO <floating point number>

The speed ratio for coverage animation. The default value is 1.0, which is 1 line per 1000 msec.

Uncovered Line Indicator

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

The “Uncovered line indicator” option allows the selection of a character to be used to mark "uncovered" lines in the Coverage Viewer and coverage reports. It allows easy identification of uncovered lines in the VectorCAST reports outside of the VectorCAST GUI.

The feature can be used with Statement, DO-178B Level A, B, and C coverage types.

Valid character choices are: '#', '$', '%', '&', or '+'. The default is "no character".

```
clicast -lc option VCAST_UNCOVERED_LINE_INDICATOR ['#' | '$' | '%' | '&' |
'space']

The uncovered line indicator. The default value is space, that is none.
```

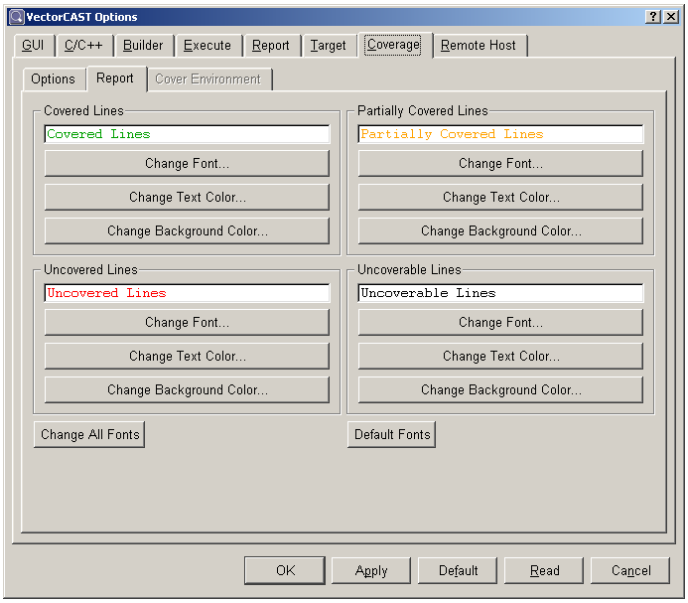
Reinstrumentation is necessary after changing this option. **Coverage => Initialize**, and select **coverage type**

Coverage Viewer Options

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

The **Coverage Viewer** tab enables you to control the fonts and colors used in the Coverage Viewer and the various coverage reports. By default, covered lines are shown in green, uncovered lines are shown in red, and non-executable statements are shown in black. It is recommended that you use a monospaced font for the Coverage Viewer (e.g. Courier) as that allows the tables and report to be aligned properly.

Note: These options do not have a CLICAST counterpart; they are strictly for the Coverage Viewer.



Disable Boolean Cast in Instrumentation

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab if it is not selected.

Set this option to prevent VectorCAST from casting conditional expressions to boolean when instrumenting for Branch or MC/DC coverage.

After changing this option, you must reinitialize coverage.

```
clicast -lada Option VCAST_DISABLE_BOOLEAN_CAST True | False

Select this option to prevent VectorCAST from casting conditional
expressions to boolean when instrumenting for branch or MC/DC coverage. The
default value is False.
```

To Format the Text in the Coverage Viewer

Choose **Tools => Options** and click the Coverage tab. Then click the Report sub-tab. Choose one of the four contexts you want to change.

Covered Lines – lines or branches that have been completely covered.

Partially Covered Lines – lines that have multiple outcomes where some but not all outcomes have been tested.

Uncovered Lines – lines or branches that have not yet been executed.

Uncoverable Lines – non-executable statements (e.g., comments).

To Reset the Fonts to Default

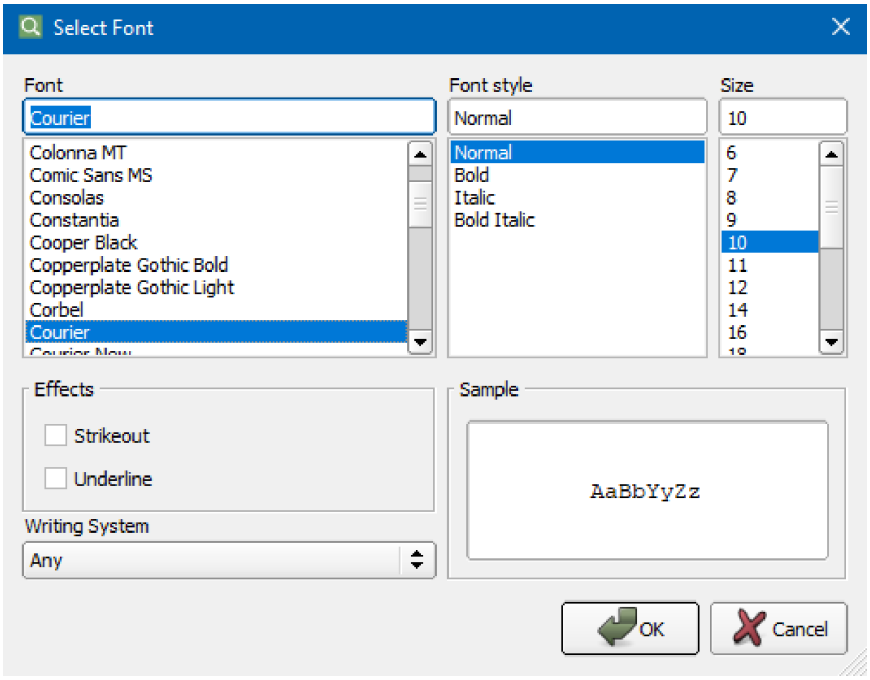
Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

Click the **Default Fonts** button. This option returns the font and color for each line type to the default settings: Courier font, Normal Font style, 10 pt Size.

To Change the Fonts

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

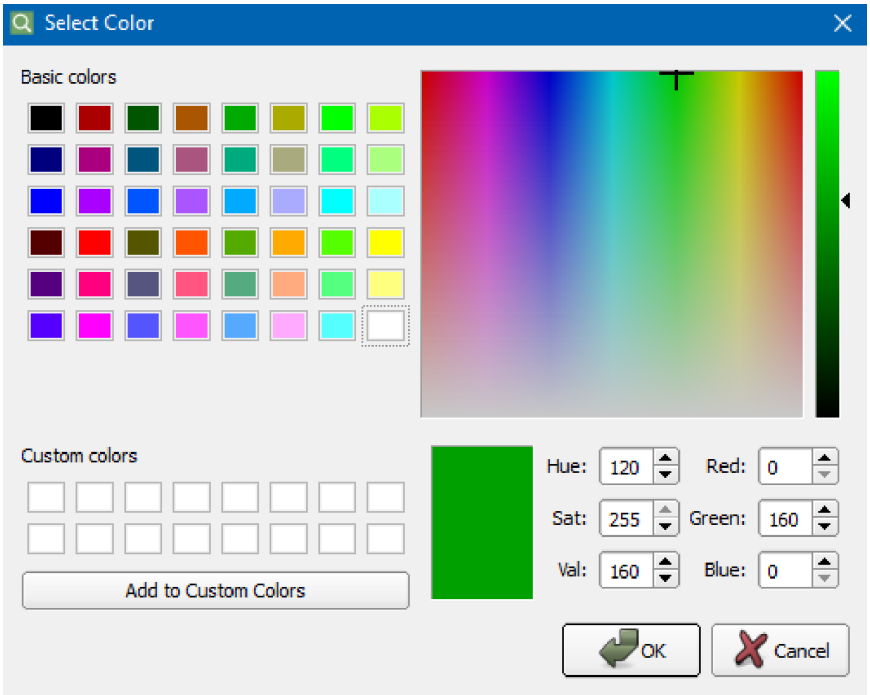
The **Font...** button for each line type and the **Change All Fonts...** button enable you to select the font, font style and size for one or all line types in the Coverage Viewer. The **Default Fonts** button enables you to revert to the default font settings. Default font settings are: Courier font, Normal Font style, 10 pt Size.



To Change Text Color

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

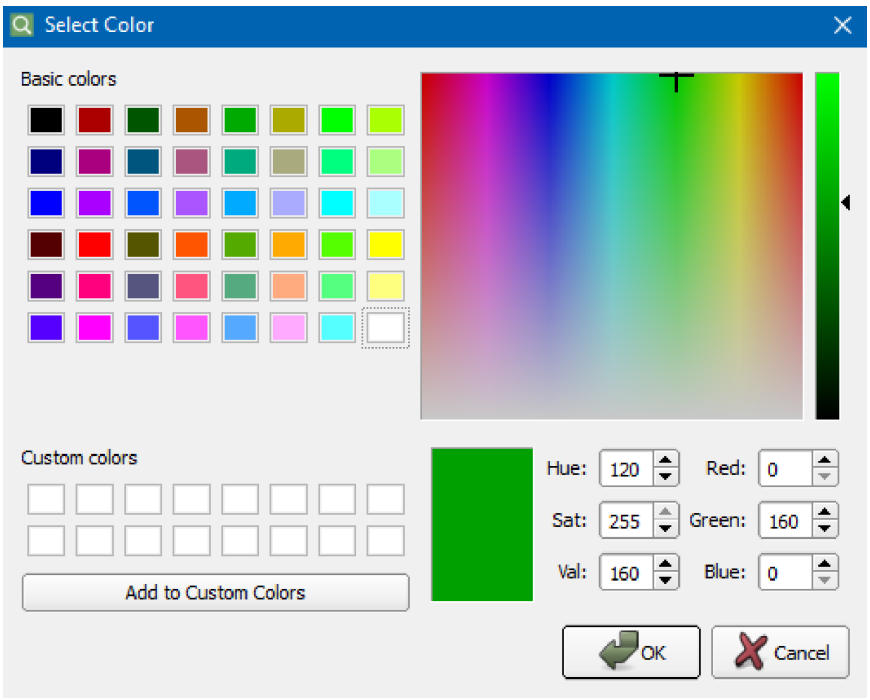
The **Text Color...** buttons enable you to change the text color for each of the five line types.



To Change Background Color

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

The **Background Color...** buttons enable you to change the background color for the five line types.



Importing Coverage Results

The **Coverage => Import Results from Environment** feature is useful if you have used another VectorCAST environment to generate some coverage data, and want to import that data into the current test environment to increase the code coverage. Importing coverage data enables you to achieve complete coverage using any combination of unit, integration, and functional tests.

To import coverage results, choose **Coverage => Import Results from Environment**. If the menu item is dimmed, first initialize coverage by choosing **Coverage => Initialize => *coverage-type***. If coverage has been initialized but disabled, you must enable it for the menu item to be available.

Preparing to Import Results

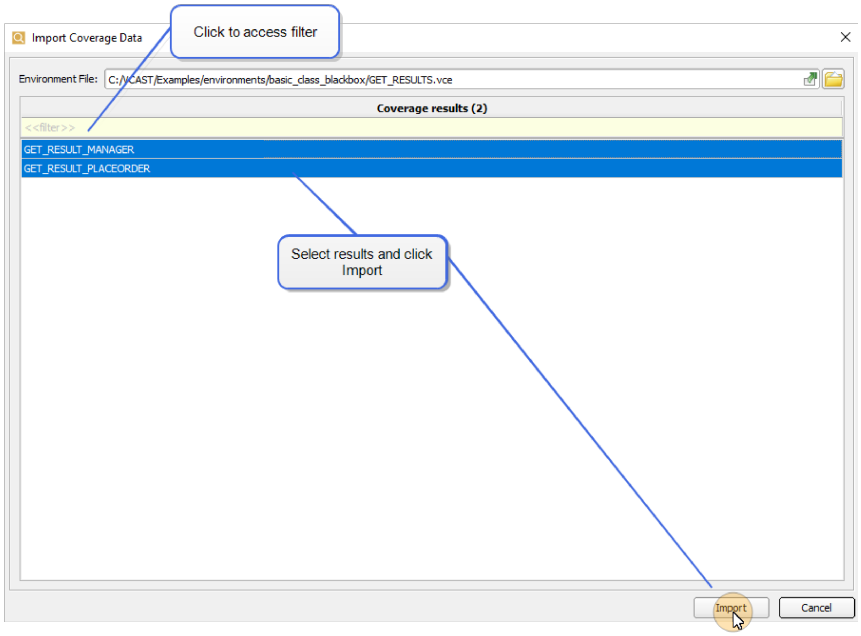
In order to import coverage, several conditions must be met.

- The same files must be used in both environments. VectorCAST compares the filenames and file checksums to ensure that the files are the same or copies of each other.
- “Instrument Files in Place” in the VectorCAST/Cover environment may be used with VectorCAST version 5.0+, otherwise, “Instrument Files in Place” may not be used.
- The same type of coverage must be used in both environments.

Importing the Results

Choose **Coverage => Import Results from Environment**, and a dialog appears which enables you to browse for a VectorCAST Cover environment (.vcp) or a VectorCAST environment (.vce). Note that you select an environment from which to import, not a test result.

Select the results files from that environment to import. You can choose to import the other environment's test execution results or imported results or both.



A filter is provided at the top of the pane making it easier to filter out some results when working with a large set of test results. By entering text or a regular expression in the <<filter>> field, the user can filter by name. Clear the filter by right-clicking in the top row and selecting **Clear Filter** from the context menu.

When you are ready to import the coverage results, click the **Import** button. To exit the dialog box without importing, click **Cancel**.

After the import is complete, a Coverage Import/Export log is displayed showing a series of status (S) or error (E) messages. Some of the messages you may see in the log file:

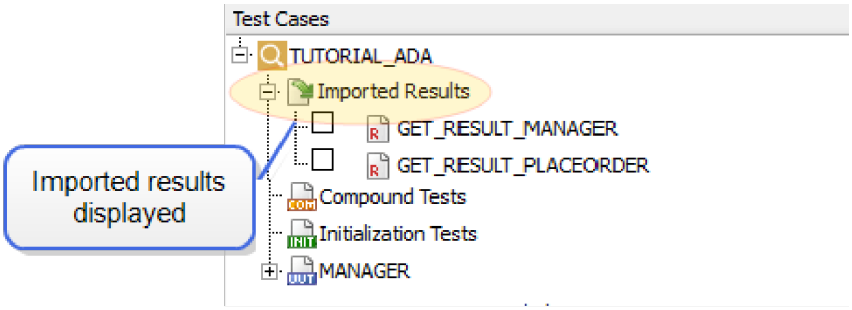
- **`(S) Source file matched`**
  The source file exists and was successfully matched.
- **`(S) No match for file <file> referenced in script file`**
  `>>> References to unit <unit number> will be ignored.`
  Not an error. The coverage results that were imported refer to a unit that is not present in the environment to which the results are being imported. This situation is typical if <file> is stubbed in the environment to which the results are being imported, but not-stubbed in the other environment. Therefore, the coverage data for <file> are just being ignored.
- **`(E) Coverage not on for source file`**
  The source file exists in the environment to which results are being imported, but coverage is not

initialized for that unit. An example is when there are non-stubbed dependent units in the environment but only the UUT was initialized for coverage. Use Initialize Custom to enable coverage for non-stubbed dependent units.

- **(E) No match found for source file**
  Although the file names may be the same, the checksums of the source files differ.
- **(E) Coverage types differ**
  The type of coverage in the importing environment is different than the type of coverage in the environment to which results are being imported. They must match for a successful import.
- **(E) No translatable data for result**
  A result was found in the importing environment, but the data did not match any source file in the current environment.
- **(S) Coverage data was loaded**
  A result file was successfully imported.

The Coverage Import/Export Log file is named **IMPORT.LOG** and resides in the environment directory. You can view it at any time by selecting **Coverage => View Import Log**.

The imported results appear in the Test Case Tree, near the top. They are listed in a folder named "Imported Results", as shown below.



To Export a Script of Coverage

To aid in regression testing, you can export a script to achieve the coverage results that you imported. Use **Coverage => Export Script** to create the script file.

This script can be imported to VectorCAST/Cover or into another VectorCAST environment. VectorCAST automatically generates a coverage script when regression test scripts are created if coverage results have been imported.

After the script file is exported, a log is displayed showing a series of status (S) or error (E) messages. Some of the messages you may see in the log file:

- (S) Script creation started – the process of creating the script file (.cvr) is beginning.
- (S) Looking for results directory IMPORTED_RESULTS – VectorCAST uses the results in the directory IMPORTED_RESULTS, located in the environment directory, to create the script file containing imported results.
- (E) Error reading imported results directory – there was an error reading the directory IMPORTED_RESULTS, located in the environment directory. Make sure the results have not been deleted or corrupted.

- (E) No imported results found – VectorCAST could not find any results file in the directory IMPORTED_RESULTS. The script file could not be successfully created.
- (E) Script creation aborted – the script file could not be created due to other errors.
- (S) Script creation completed – the script file was created successfully.

An example of a coverage script (.cvr) file is as follows:

```
-- VectorCAST 5.0 (09/11/08)
-- Imported Coverage Results Script
IMPORT.BEGIN
IMPORT.SOURCE.BEGIN
IMPORT.SOURCE.UNIT:1
IMPORT.SOURCE.ORIG_FILENAME:c:\vcast\Environments\database.c
IMPORT.SOURCE.TIMESTAMP:1221235044
IMPORT.SOURCE.COVERAGE_STATUS:TRUE
IMPORT.SOURCE.COVERAGE_TYPE:STATEMENT
IMPORT.SOURCE.FILE_CHECKSUM:2106310038
IMPORT.SOURCE.END
IMPORT.SOURCE.BEGIN
IMPORT.SOURCE.UNIT:2
IMPORT.SOURCE.ORIG_FILENAME:c:\vcast\Environments\manager.c
IMPORT.SOURCE.TIMESTAMP:1221235044
IMPORT.SOURCE.COVERAGE_STATUS:TRUE
IMPORT.SOURCE.COVERAGE_TYPE:STATEMENT
IMPORT.SOURCE.FILE_CHECKSUM:3071222853
IMPORT.SOURCE.END
IMPORT.SOURCE.BEGIN
IMPORT.SOURCE.UNIT:3
IMPORT.SOURCE.ORIG_FILENAME:c:\vcast\Environments\manager_driver.c
IMPORT.SOURCE.TIMESTAMP:1221235044
IMPORT.SOURCE.COVERAGE_STATUS:TRUE
IMPORT.SOURCE.COVERAGE_TYPE:STATEMENT
IMPORT.SOURCE.FILE_CHECKSUM:2039996053
IMPORT.SOURCE.END
RESULT.NEW
RESULT.NAME:IMPORTED_RESULTS\PLACE_ORDER-PATH-1
RESULT.DATA:1 1 1
RESULT.DATA:1 2 1
RESULT.DATA:2 1 1
RESULT.DATA:2 1 3
RESULT.DATA:2 2 1
RESULT.DATA:2 2 2
RESULT.DATA:2 2 3
RESULT.DATA:2 2 4
RESULT.DATA:2 2 5
RESULT.DATA:2 2 6
RESULT.DATA:2 2 14
RESULT.DATA:2 2 15
RESULT.DATA:2 2 16
```

```
RESULT.DATA:2 2 17
RESULT.END
RESULT.NEW
RESULT.NAME:IMPORTED_RESULTS\PLACE_ORDER-PATH-2
RESULT.DATA:1 1 1
RESULT.DATA:1 2 1
RESULT.DATA:2 1 1
RESULT.DATA:2 1 3
RESULT.DATA:2 2 1
RESULT.DATA:2 2 2
RESULT.DATA:2 2 3
RESULT.DATA:2 2 4
RESULT.DATA:2 2 5
RESULT.DATA:2 2 6
RESULT.DATA:2 2 7
RESULT.DATA:2 2 16
RESULT.DATA:2 2 17
RESULT.END
RESULT.NEW
RESULT.NAME:IMPORTED_RESULTS\PLACE_ORDER-PATH-3
RESULT.DATA:1 1 1
RESULT.DATA:1 2 1
RESULT.DATA:2 1 1
RESULT.DATA:2 1 3
RESULT.DATA:2 2 1
RESULT.DATA:2 2 2
RESULT.DATA:2 2 3
RESULT.DATA:2 2 4
RESULT.DATA:2 2 5
RESULT.DATA:2 2 6
RESULT.DATA:2 2 8
RESULT.DATA:2 2 9
RESULT.DATA:2 2 16
RESULT.DATA:2 2 17
RESULT.END
RESULT.NEW
RESULT.NAME:IMPORTED_RESULTS\PLACE_ORDER-PATH-4
RESULT.DATA:1 1 1
RESULT.DATA:1 2 1
RESULT.DATA:2 1 1
RESULT.DATA:2 1 3
RESULT.DATA:2 2 1
RESULT.DATA:2 2 2
RESULT.DATA:2 2 3
RESULT.DATA:2 2 4
RESULT.DATA:2 2 5
RESULT.DATA:2 2 6
RESULT.DATA:2 2 10
```

```
RESULT.DATA:2 2 11
RESULT.DATA:2 2 16
RESULT.DATA:2 2 17
RESULT.END
RESULT.NEW
RESULT.NAME:IMPORTED_RESULTS\PLACE_ORDER-PATH-5
RESULT.DATA:1 1 1
RESULT.DATA:1 2 1
RESULT.DATA:2 1 1
RESULT.DATA:2 1 3
RESULT.DATA:2 2 1
RESULT.DATA:2 2 2
RESULT.DATA:2 2 3
RESULT.DATA:2 2 4
RESULT.DATA:2 2 5
RESULT.DATA:2 2 6
RESULT.DATA:2 2 12
RESULT.DATA:2 2 13
RESULT.DATA:2 2 16
RESULT.DATA:2 2 17
RESULT.END
IMPORT.END
```

clicast -e <env> TOols EXPORT_Results_coverage <scriptfile>

Create a coverage script containing test execution results that are present in <env>, including imported results. Coverage scripts have the extension .cvr.

clicast -e <env> TOols EXPORT_Coverage <scriptfile>

Create a coverage script containing only imported coverage results that are present in <env>.

To Import a Coverage Script

The **Coverage => Import Script** command imports a coverage script that was previously generated using the Export Script command. The default file extension is .cvr.

clicast -e <env> TOols Import_coverage <scriptfile> | <env>

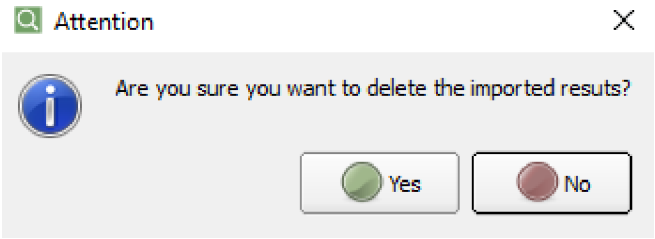
Import a coverage script named <scriptfile> or import all coverage results from <env>.

Any errors that occur during importing of a coverage script are logged in a file named Import.log. To view this file, choose **Coverage => View Import Log**.

To Delete Imported Results

To delete all imported coverage results, choose **Coverage => Delete Imported Results**. A dialog

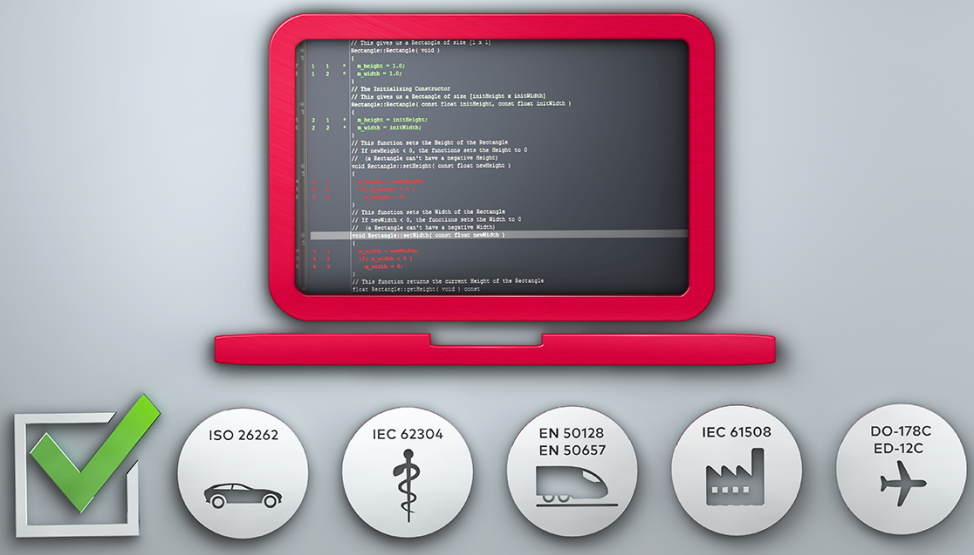
appears asking you to verify that you want to continue with this operation, as shown below.



Clicking **No** cancels the operation without removing the imported coverage results. Clicking **Yes** removes the imported coverage results from the environment and the **IMPORTED_RESULTS** directory from the environment directory. There is no clicast command to delete all imported results.

To View the Coverage Import Log

Any errors that occur during importing of a coverage script are logged in a file named Import.log. To view this file, choose **Coverage => View Import Log**.



Using VectorCAST Covered By Analysis

Covered By Analysis (CBA)

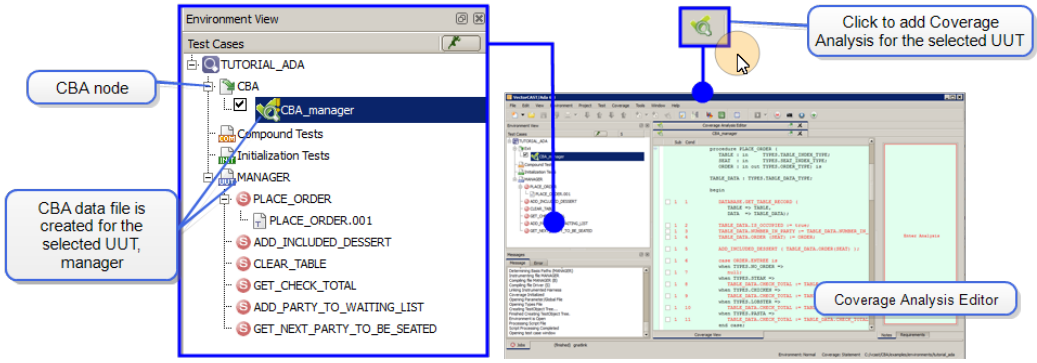
Covered By Analysis (CBA) allows users to augment test coverage metrics with coverage analysis data sets. Small portions of embedded applications are commonly impossible to test, but regulated industries require documented analysis of uncovered code to meet the requirement of 100% structural coverage.

VectorCAST CBA provides the ability to do code analysis directly within VectorCAST using the Coverage Analysis Editor and combines the test and analysis coverage metrics in a single report. VectorCAST CBA can also import analysis files, including those generated by third party tools.

To Add Coverage Analysis

The environment must first be instrumented for coverage. A CBA Analysis result is associated with one UUT or non-stubbed unit. To add Coverage Analysis for a selected unit, select the **CBA** button  on the Toolbar. Clicking the **CBA** button requires a unit to be selected. Alternatively, right-click a UUT in the Test Case Tree and select **Add Coverage Analysis** from the context menu.

The Coverage Analysis Editor opens in the MDI Window and a CBA node displays in the Test Case Tree. A CBA data file is created for the selected UUT and this Analysis result displays beneath the CBA node. When an Analysis result is first created, it is empty and contains no data. Empty Analysis results display the  icon, even if they contain notes or requirements. Each CBA data file corresponds to a single UUT, but you can create multiple Analysis results per UUT.



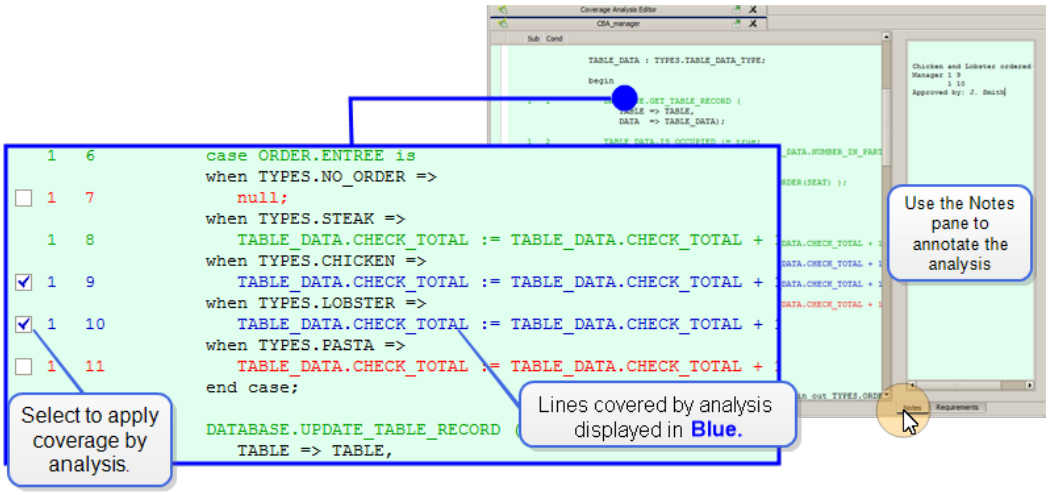
Using the Coverage Analysis Editor

Once an Analysis result has been created, the Coverage Analysis Editor can be opened at any time by double-clicking on the Analysis result.

In the Coverage Analysis Editor, the Notes tab on the right is a free-form text editor allowing you to annotate the associated analysis. Use the Requirements pane to trace the Project Requirements and Test Case Requirements associated with the selected code. The Requirements tab is populated by VectorCAST Requirements Gateway. Use the **Save** button to save inputs.

Statement Coverage

With Statement coverage instrumented, the Coverage Analysis Editor displays boxes on the left for statement outcomes that are uncovered. To mark a statement or condition as "considered covered", select the check box. Lines covered by analysis are displayed in blue.



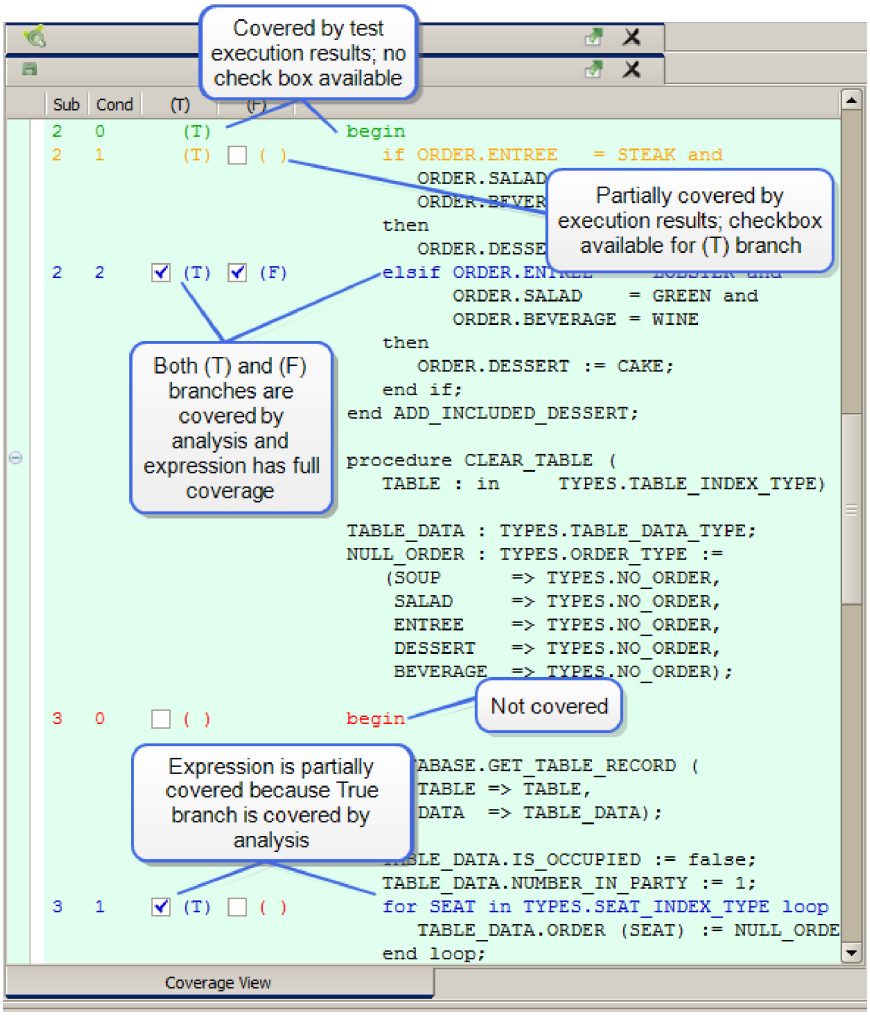
In the Coverage Analysis Editor, a check box does not appear next to a line if it is already covered by a test case execution result. Regular test case results, if they are present in the environment, take precedence over CBA results.

Branch Coverage

When Branch coverage is instrumented in the environment, the Coverage Analysis Editor displays each subprogram with a True branch (T) for the entry result, and True and False branches (T) (F) for each expression.

To mark a condition as having the True branch covered, select the check box in the (T) column. The " (T)" is displayed in blue. The branch is now partially covered because the True branch is Covered By Analysis, and the False branch is not covered.

To mark an expression as having the False branch covered, select the check box in the (F) column in the Coverage Analysis Editor. The "(F)" is displayed in blue.

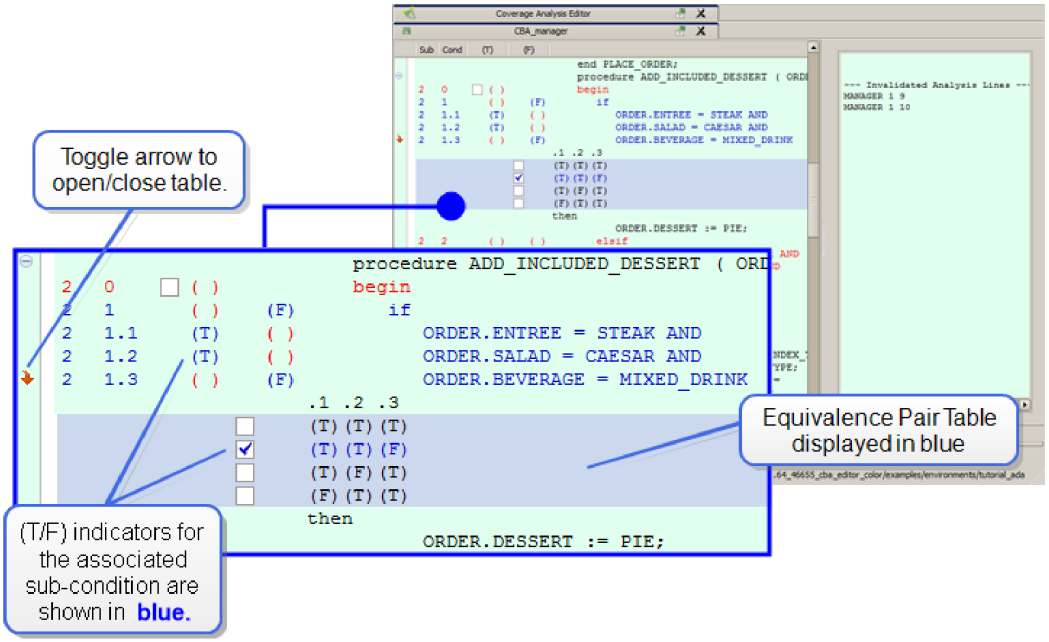


As with Statement coverage, if either or both of the True and False branches is already covered by regular test case execution results, then the check box is not available in the Coverage Analysis Editor, and the expression shows yellow if partially covered, or green if already fully covered.

MC/DC Coverage

To add coverage analysis for a condition with multiple sub-conditions when MC/DC Coverage is instrumented, you annotate that one or more rows of the Equivalence Pairs table is covered.

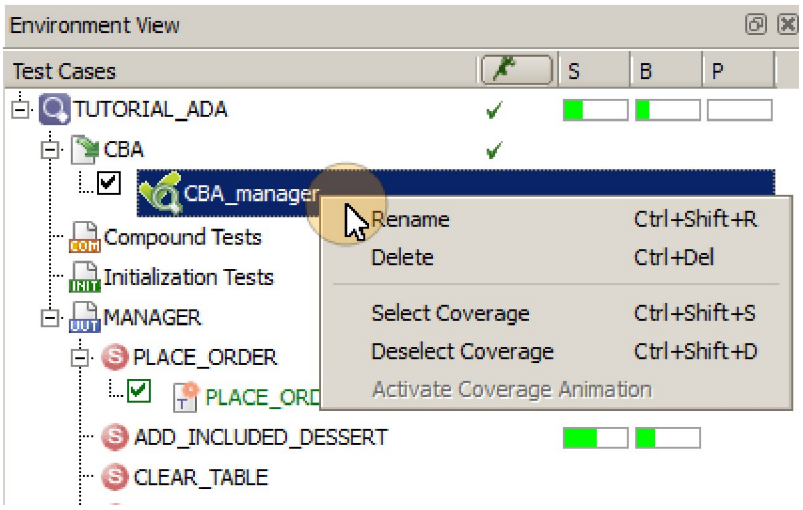
To access the equivalence pair table, click the arrow to the left of the condition. The truth table opens and a check box is displayed for each row. When a checkbox is selected, the associated sub-condition is considered "covered" and displayed in blue.



To Edit an Existing Analysis Result

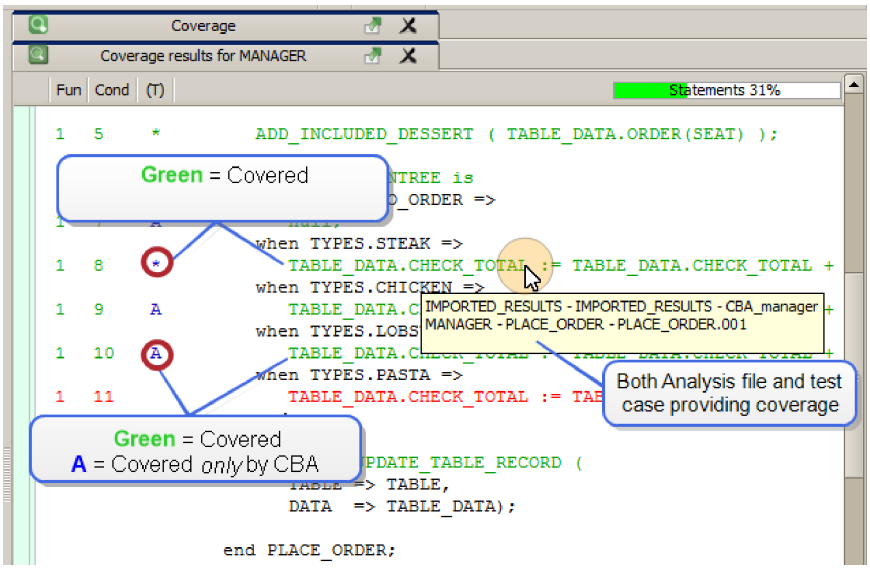
Any existing Analysis results can be edited, regardless of how they were created. This includes Analysis results that are imported via **`.cvr`**, **`.cba`**, and after rebuilding an environment.

Double-click on an existing Analysis result in the Test Case Tree to open the Coverage Analysis Editor. Right-clicking on the Analysis result provides a contextual menu allowing you to Rename, Delete, Select Coverage and Deselect Coverage. Select and Deselect turns On/Off the Analysis result in the Coverage Viewer. Deleting an Analysis result removes its data, notes, and requirements from the environment.



Viewing CBA Coverage in the Coverage Viewer

CBA coverage is displayed in blue in the Coverage Viewer. When viewing CBA lines in the Coverage Viewer, hover over a CBA line to see the Analysis result providing coverage.

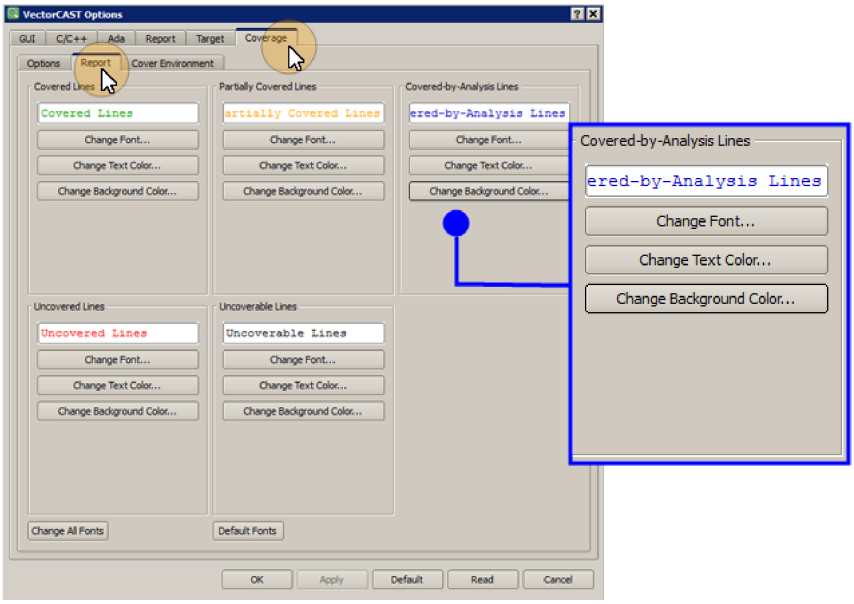


Lines covered only by CBA results are displayed in green (indicating the line is covered) with a blue "A" (indicating the line is covered *only* by CBA).

Lines covered by both regular execution results *and* CBA results are displayed in green (indicating the line is covered) and with a blue asterisk "*" (for statement) or a blue "T" or "F" (for branch), which indicates that it is *also* covered by CBA.

To Change Covered-By-Analysis Display Color

The default Covered-By-Analysis display color is blue. To change the color select **Tools => Options** from the Menu bar. On the **Tools => Options** dialog, select the **Coverage** tab and the **Report** sub-tab.



Use the buttons provided in the Covered-By-Analysis group box to change the font, text color and background color. When changes are complete, select the **Apply** button to save the changes to the .CFG file.

Working With Analysis Files

To Import Analysis Files

Importing Analysis Files allows the creation of analysis files from other tools. To import CBA data from a text file, select **Coverage => Import Coverage Analysis** from the Menu bar. An Open File dialog is displayed allowing you to select the Analysis file to be imported. The supported file type is **.dat.** Imported CBA Analysis results cannot be edited.

For Statement coverage, the .dat file syntax is:

Unit Number	Subprogram Number	Line Number
9	1	1
9	1	2

The .dat file above covers the first two lines of Subprogram 1 of Unit 9.

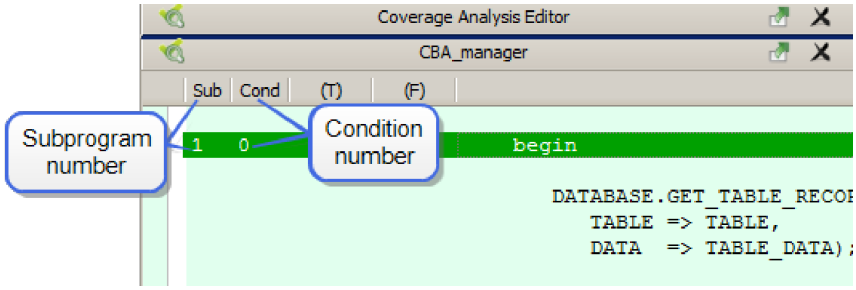
For Branch coverage, the .dat file syntax is:

Unit Number	Subprogram Number	Condition	Outcome Covered
9	1	0	T
9	1	1	T

In the .dat file above, 9 1 0 T is the entry point to the first Subprogram of Unit 9. 9 1 1 T is the True

branch for the first condition of the first Subprogram of Unit 9.

Unit Test environments always have Unit 9 as the first UUT, Unit 11 as the second UUT, Unit 12 as the third, etc. The user can refer to the Coverage Viewer to locate the Subprogram number and Condition number, whether T or F branch is covered, and then fill in the Unit number.



To Export Analysis Files

To export CBA data, select **Coverage => Export Script => Analysis Results...** from the Menu bar. A Save As dialog is displayed allowing you to name the coverage file to be created. By default, the file extension "**.cvr**" is selected as the supported file type.

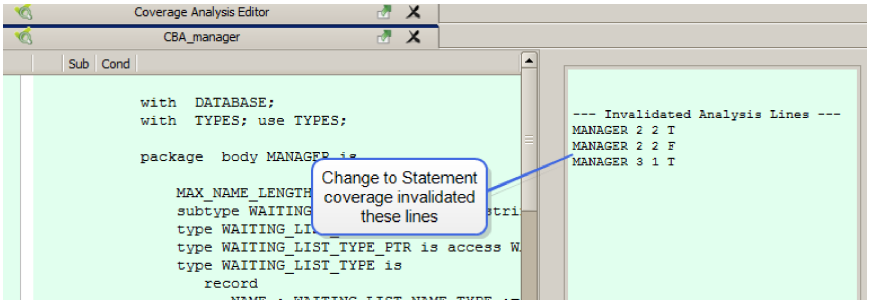
```
clicast -e <env> TOols EXPORT_CBA <scriptfile>

Generate a script containing Coverage By Analysis results.
```

To Remove Analysis Files

To remove CBA data files, right-click on the Analysis result and select **Delete** from the context menu. The CBA coverage data, Notes, and Requirements are removed.

Re-Instrumenting With CBA Data

When you change the coverage type in the environment, the Notes and Requirements for CBA are automatically retained and include information on the data that was invalidated. The data that is invalidated is recorded in the Notes section. VectorCAST writes information about the deleted data in the Notes tab, appending to any Notes that were already there. When you see a CBA Analysis result with the icon indicating it is empty , double-click the icon to view the information in the Notes.

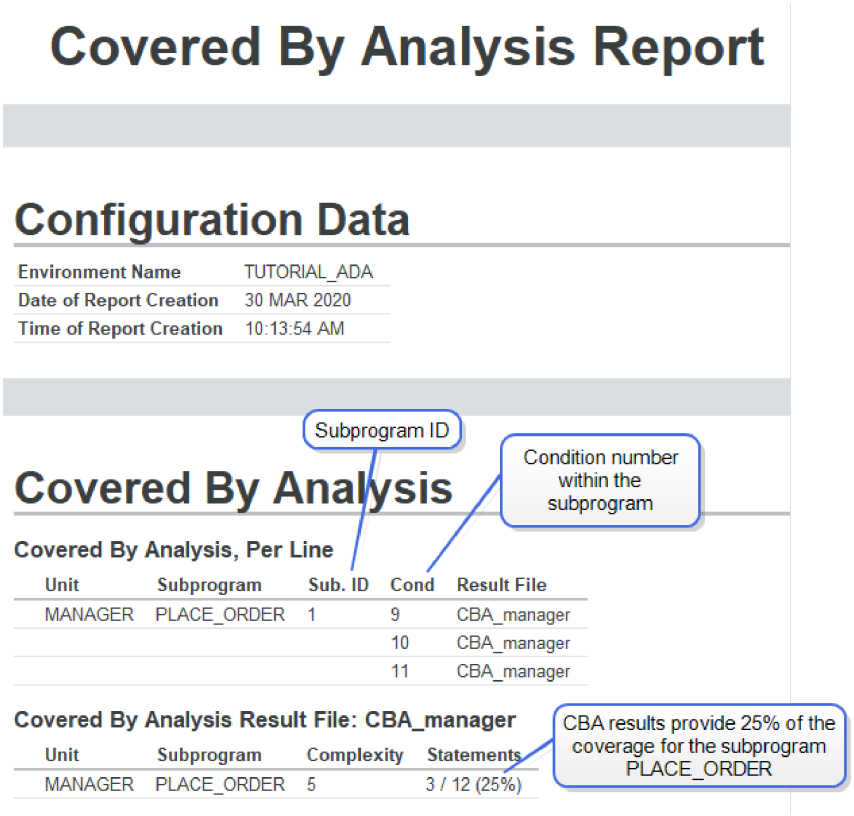


Viewing Analysis Data in Reports

Covered By Analysis Report

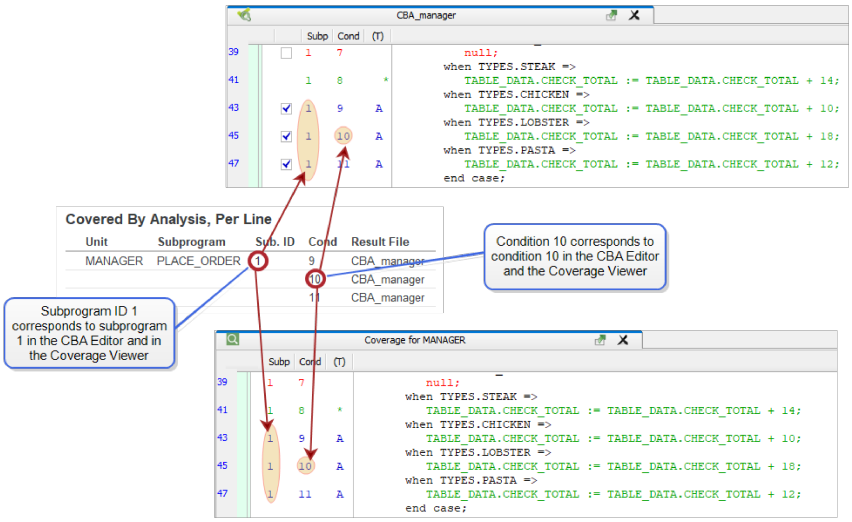
The Covered By Analysis (CBA) Report provides an overview of the Analysis data, including the total number of lines or conditions covered and any associated notes and requirements. The CBA Report includes only CBA results and disregards any regular test case results which normally take precedence over CBA results.

To access the Covered By Analysis Report, select **Test => View => Covered By Analysis Report** from the Menu Bar.



The CBA Report has two sections: The **Covered By Analysis, Per Line** section and the **Covered By Analysis Result File** section(s).

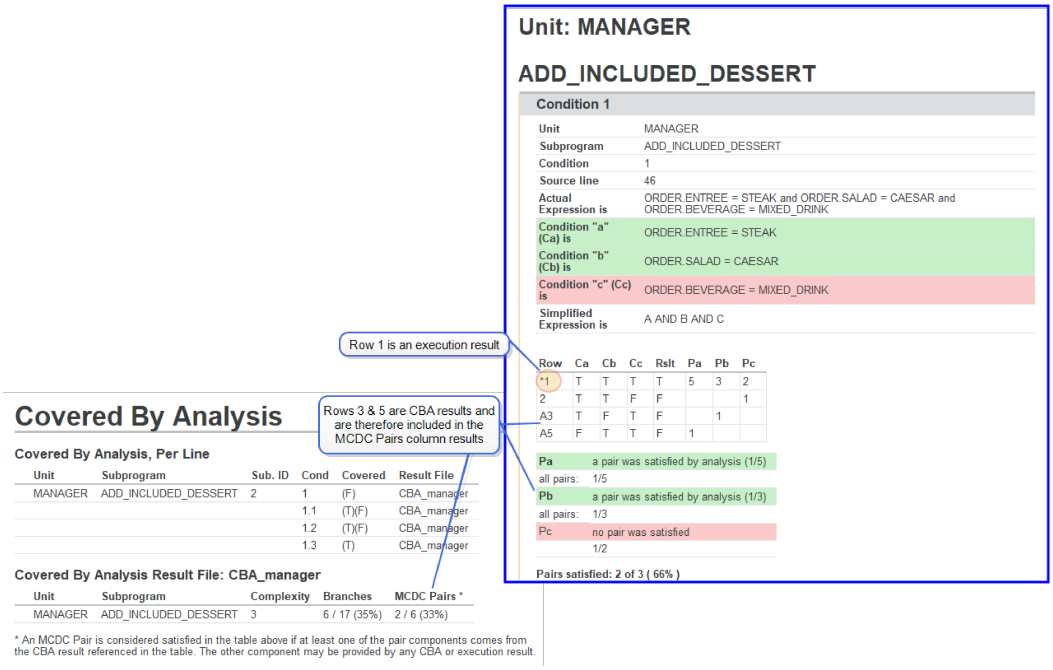
The **Covered By Analysis, Per Line** section lists each covered line in the unit, and identifies which CBA result file covers that line. The Subprogram ID corresponds to the left-hand number in the Coverage Viewer and CBA Editor. The Line number corresponds to the right-hand number in the Coverage Viewer and CBA Editor when the coverage type is Statement. The Line number represents the branch or decision number when the coverage type is other than Statement. There may be more than one CBA result covering a line. If the CBA result covers the True (T) outcome of a branch or decision, "(T)" is displayed in the line column. Similarly, if it covers the False (F) outcome, "(F)" is displayed.



The **Covered By Analysis Result File** section includes one table for each CBA result file. The table is similar to the Metrics Report in that it shows the number of statements and/or branches covered by that CBA result. For Statement and Branch coverage, only the number of lines or conditions that are Covered by Analysis are shown.

For MC/DC or Level A coverage, the number of conditions covered by CBA are shown. However, for the MCDC Pairs column, both CBA results and execution results are considered. An MC/DC Pair is considered satisfied if at least one of the pair components (or row) is a CBA result. The remaining component may be either a CBA result or an execution result.

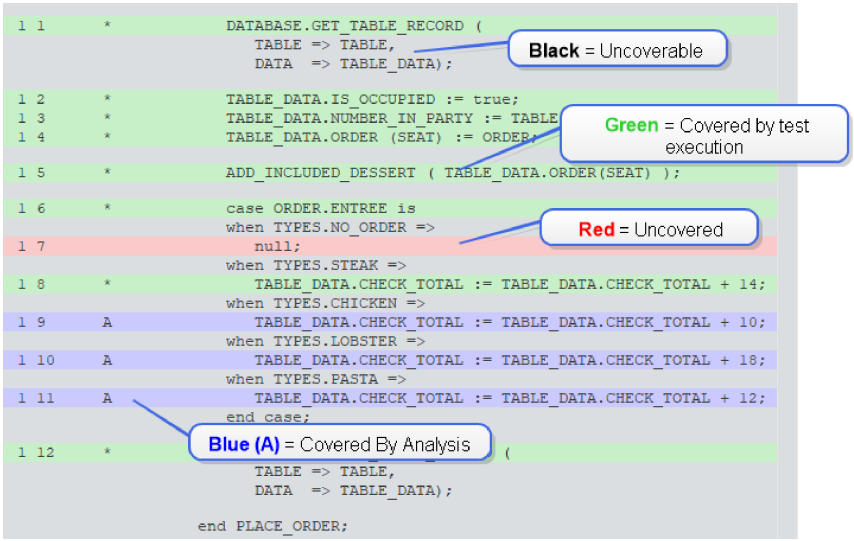
In the example below, two pairs are covered. Each pair has one component covered by the CBA result file ADD_INCLUDED_DESSERT (rows 2 and 3). The other component (row 1) is covered by an execution result.




```
clicast -e <env> Reports Custom CBA <outputfile>

Create a CBA report and output to HTML file <env>_covered_by_analysis_
report.html, standard output as text, or to the specified output file.
```

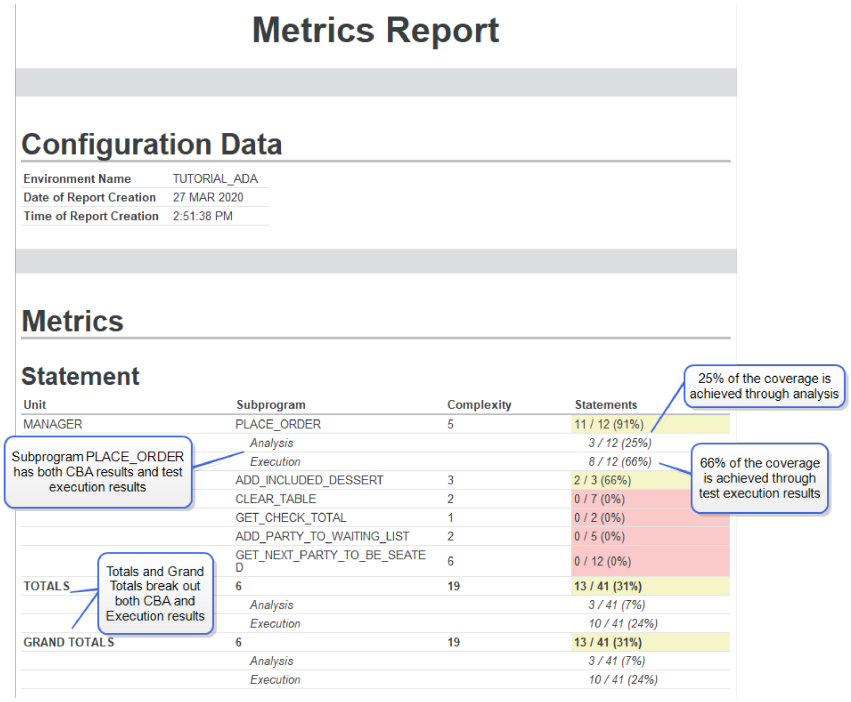
Aggregate Coverage Report

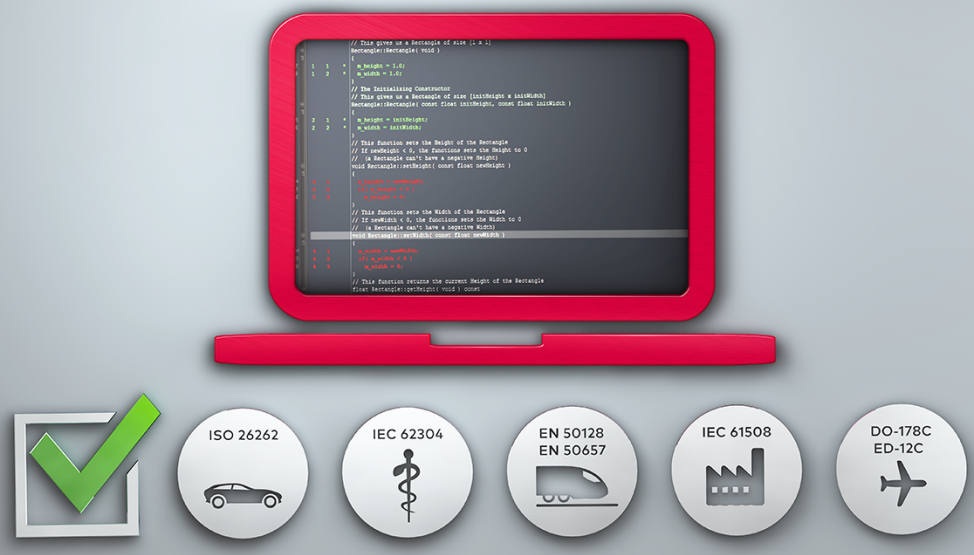
In the Aggregate Coverage Report, items that are only Covered by Analysis are indicated by the character 'A' and a blue background. Otherwise, the execution results are displayed using a green background to indicate covered by execution. Red background indicates uncovered.



Metrics Report

In the Metrics Report, when CBA results are present in the environment, they are displayed in italics in the row below the subprogram and the coverage achieved by test execution is displayed below the CBA results.





User Code

Understanding User Code

Types of User Code

VectorCAST's User Code capability enables you to write Ada-language expressions to set or check the value of data objects, or to do some other dynamic test case setup. With user code, you can write code to read data from a file, call initialization routines, or assign and verify data objects based on dynamic criteria.

There are several types of user code: environment user code, test case user code, individual parameter user code, and stub user code.

- Environment User Code is best suited for operations that relate to all test cases; loading data from a file or initializing a database are two examples.
- Stub User Code is added to functions that have been stubbed, allowing you to specify extra logic to be executed when the stub is called.
- Test Case User Code is used to include input value or expected value user code which is not associated with a specific parameter. It applies to simple test cases, not compound test cases. It is accessed by clicking the Testcase User Code tab in the Test Case Editor window.
- Parameter User Code is used to set a parameter value based on a dynamic expression, or to verify a parameter value against some expression.

Order of Execution

User code is executed in the following order during test case execution:

- Environment User Code – Harness Init
- Environment User Code – Test Case Init
- Test Case Input User Code
- Parameter Input User Code
- Timer Start
- [Invocation of a Unit Under Test (Test Harness calls UUT)
    - [Invocation of a stubbed subprogram]
    - Timer Stop
    - Configure Stubs User Code (Beginning of Stub)
    - Environment User Code (tags only) – Stub Entry User Code
    - Environment User Code – Stub Processing
    - Parameter Input User Code for Stub
    - Environment User Code (tags only) – Stub Exit User Code
    - Configure Stubs User Code (End of Stub)

>> Timer Start

>> [Return from stubbed subprogram]

> [Return from UUT to Test Harness]
> Timer Stop
> Environment User Code – Test Case Term
> Parameter Expected User Code
> Test Case Expected User Code
> Environment User Code – Harness Term

Editing User Code

Each type of user code is entered by accessing its user code dialog:

> Environment User Code is accessed by choosing **Environment => User Code => Edit**.
> Stub User Code is accessed by choosing **Environment => Configure Stubs => Edit**.
> Test Case User Code is accessed in the Testcase User Code tab of the Test Case Editor.
> Parameter User Code is accessed by right-clicking on a parameter in the Parameter Tree and selecting “Properties…” or double-clicking the parameter and selecting the User Code tab.

User Code Tags

User code can consist of any legal Ada source code. Beyond that, there are some special tags that can be used to refer to VectorCAST-generated variable names. When the test harness is generated, VectorCAST creates a global variable for each parameter of each subprogram in each unit that is either under test or stubbed. These variable names may change each time a test harness is rebuilt. To accommodate this, you refer to these variables using a special tag syntax.

Some examples of the tag syntax follow (see the next section for information on obtaining the correct tag syntax automatically):

> Parameter objects are specified using the notation:

```
<<unitname.subprogram.parameter>>
```

> Structures and arrays are specified using the notation:

```
<<unitname.subprogram.array_name>>.element
```

> Pointer to a structure is specified using the notation:

```
<<unitname.subprogram.parameter>>[index].element
```

> Array objects are specified using the notation:

```
<<unitname.subprogram.parameter>>.Array[index].element
```

> If **`element`** is an array itself, keep going, as in:

```
<<unitname.subprogram.parameter>>.Array[index].element
```

> When the unit is a stubbed-by-prototype unit, the **unitname** is

```
uut_prototype_stubs.
```

> Global objects are specified using the notation:

```
<<unitname.<<GLOBAL>>.parameter>> ...
```

In addition to assigning values to parameters and global objects, you can also compare values against an expected expression. To have the result of a user code comparison show up in the test results, you must enclose the comparison in double curly braces, **{{ }}**. For example, **{{ <<unitname.subprogram.parameter>> == value }}** would result in a comparison being performed between the value of the parameter and **value**. Of course, a dynamic expression of any sort can be substituted for **value**.

When specifying Expected User Code, the code between the double braces (**{{...}}**) must evaluate to a boolean expression. The boolean expression will be evaluated as the test runs, and the value will be reported in the test results listing. The result of the evaluation of the boolean expression will be included in the pass/fail analysis of the test case. A boolean expression can be combined with other Ada-code to accomplish a more complex comparison.

Features Common to All User Code

In addition to a common syntax, all user code edit areas support an external editor, importing from a file, and automatic insertion of User Code tags.

Insert User Code Tag

To make it easier to enter user code tags for objects, you can click the User Code Tags button  in any of the user code dialogs or on the toolbar. Alternatively, select **View => Dock Control => User Code Tags** from the Menu Bar. A new window opens in non-docked mode, which displays a hierarchal tree of objects in the environment. Using this tree, navigate to the object whose tag you want to insert.

A Find Banner is available to assist in searching and filtering the text. The Find Banner appears at the bottom of the window and is off by default. Access the Find Banner using the **Ctrl + F** keyboard shortut. For more information on using this feature, see "To Search for Text Using the Find Banner" on page 85.

In the screen shot below, the MANAGER file is a UUT in the environment, and DATABASE is stubbed. The TABLE parameter in the subprogram GET_TABLE_RECORD is expanded. For each object in the tree, you can:

> right-click the object and choose **Copy** – puts both the assignment and conditional forms into your copy buffer. For example:

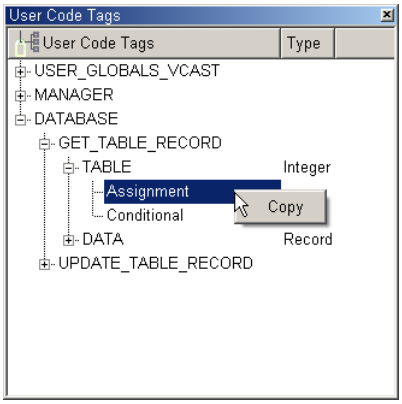
```
<<DATABASE.GET_TABLE_RECORD.TABLE>>:=(expression);
{{ <<DATABASE.GET_TABLE_RECORD.TABLE>> = (expression) }}
```

> right-click the **Assignment** node and choose **Copy** – puts the assignment form into your copy buffer. For example:

```
<<DATABASE.GET_TABLE_RECORD.TABLE>>:=(expression);
```

> right-click the **Conditional** node and choose **Copy** – puts the conditional form into your copy buffer. For example:

{{ <<DATABASE.GET_TABLE_RECORD.TABLE>> = (expression) }}



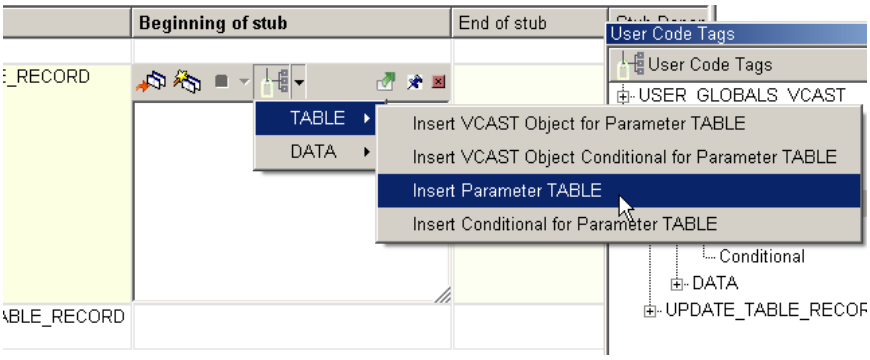
For Configure Stubs user code only, the following two additional insertion types are available:

> **Insert Parameter *param*** – inserts the actual parameter name. For the subprogram Get_Table_Record, parameter Table, the text inserted looks similar to the following:

```
Table
```

> **Insert Conditional for Parameter*param*–** inserts a VectorCAST user code conditional expression for the parameter. For the subprogram sum, parameter R, the text inserted looks similar to the following:

```
{{ Table == (expression) }}
```

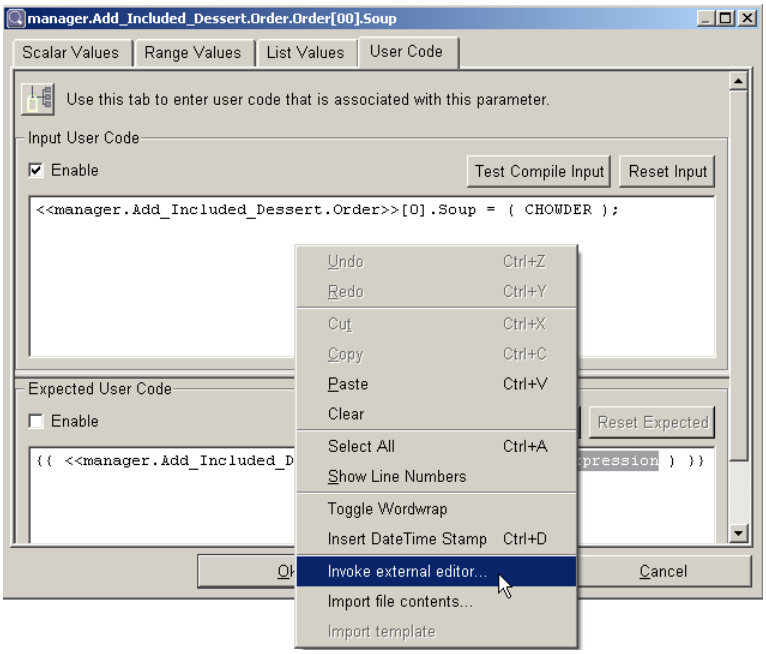


> **Insert VCAST Object for Parameter*param*** is the same as right-clicking Assignment for *param* and choosing Copy, then Paste

> **Insert VCAST Object Conditional for Parameter*param*** is the same as right-clicking Conditional for *param* and choosing Copy, then Paste.

Using an External Editor

If you chose an external text editor (notepad or emacs, for example) in the **Tools => Options** dialog, GUI tab, then you can invoke that editor by right-clicking in any of the user code text areas, and selecting **Invoke external editor** from the pop-up menu. When you are finished editing, save and exit the editor and the text is placed in the user code section.

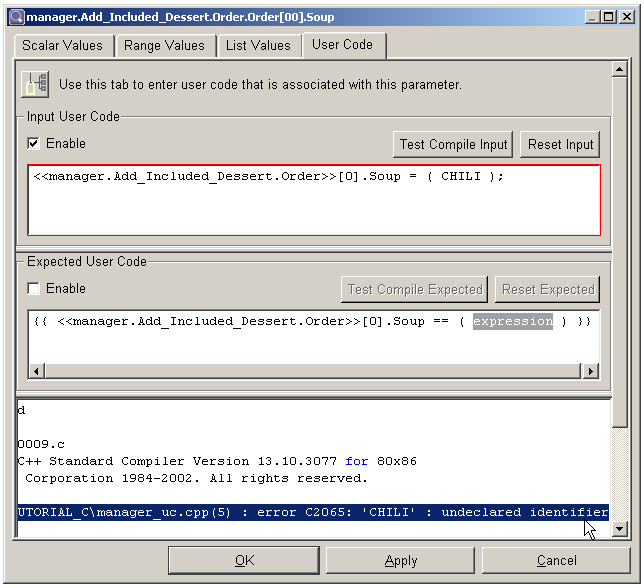


Importing Contents of a File

In addition to typing in user code, you can add the contents of a file. To import the contents of a file to any of the user code dialogs, right-click in the text area, and select **Import file contents** from the pop-up menu. An Open File dialog appears, allowing you to choose a file. Once you've chosen a file, click **Open**, and the text from the file is added to the user code section.

Test Compile Button

Before committing your user code to the test harness, a **Test Compile** button enables you to determine if there are any compilation errors in the code. The various user code dialogs have the Test Compile button in different locations, and some have a separate one for input user code and expected user code, but they all behave in essentially the same way. When you click a Test Compile user code button, the user code is compiled, and VectorCAST informs you if it was successful or there were errors, and displays the errors. It may be necessary to scroll to the bottom of the dialog to see the errors displayed.



Environment User Code

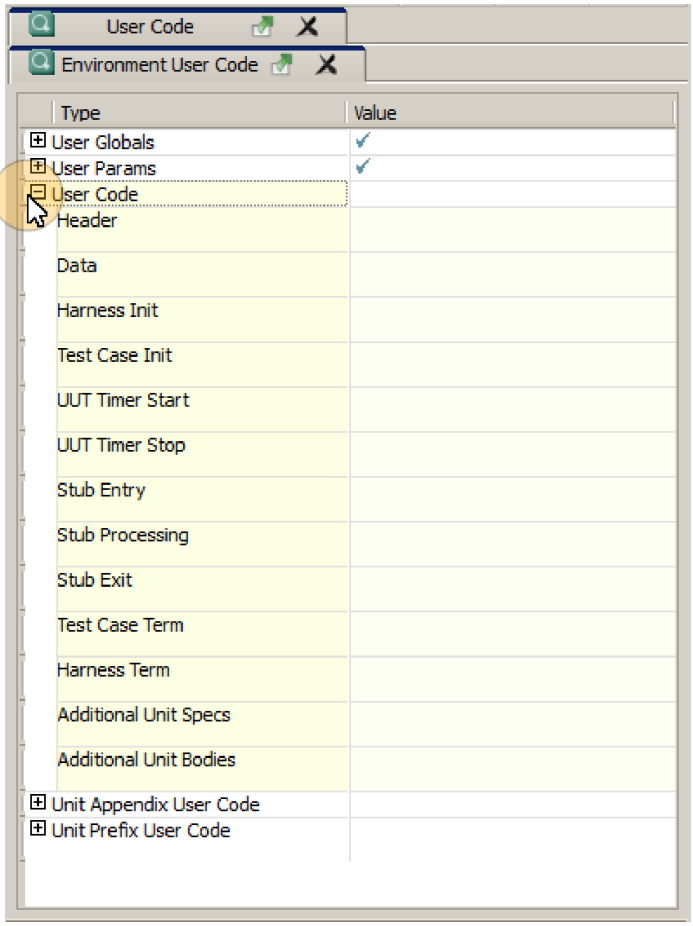
Types of Environment User Code

VectorCAST's User Code capability enables you to write source code to handle some of the harness tasks that are not easily accomplished with static data. With User Code, you can write code to read data from a file, call initialization routines, or assign and verify data objects based on dynamic criteria. *Environment User Code* is best suited for operations that relate to the harness as a whole; loading data from a file or initializing a database unit are two examples.

Choose **Environment => User Code => Edit** to enter or modify the environment user code.

The Environment User Code dialog enables access to User Globals, User Parameters, Environment User Code, Unit Appendix User Code, and Unit Prefix User Code.

Expand the node for "User Code" to see the input boxes for each type of Environment User Code.



Header: Source code that normally appears before “package body” declaration. Usually consists of “with” clauses, as well as “use” or “use type” clauses.

By default, the “with” or “use” clauses are inserted in several places: the User Code package, the nested User Code package in a whiteboxed UUT, and the stub bodies. In some cases, it is not desirable to put the same “with” statements in every location. You can control where these “with” and “use” clauses are inserted by adding an Ada comment before the “with” clause: `--unit_name`, where *`unit_name`* is the name of the UUT or any stub, for which you want the “with” clause to apply. The comment `--uut` may be used to specify the UUT. If you use `--user_code_adacast`, then the “with” clause is inserted only in the declarative section of the User Code package.

For example,

```
--database with types;
```

Data – Source code that normally occurs near the top of a source file. This could consist of type and data declarations, as well as any support subprograms that might be needed.

Harness Init – Source code in this area gets executed once per harness execution, immediately after elaboration has completed. This could include calling initialization routines.

Test Case Init – Source code in this area gets executed immediately after the test case’s data is loaded, but before the UUT is called by the harness. This usually includes initializing data for a

specific test case. If your test case sets global values, the Test Case Init code has the opportunity to write over those values.

UUT Timer Start – Source code in this area gets executed just prior to entering the call to the UUT. Used to start timer.

UUT Timer Stop – Source code in this area gets executed just after returning from the call to the UUT. Used to stop timer.

Stub Entry – Source code in this area gets executed upon entry to a stubbed subprogram, after Configure Stubs Beginning User Code.

Stub Processing – Source code in this area is executed every time any stub is called.

Stub Exit – Source code in this area gets executed upon exit from a stubbed subprogram, before Configure Stubs Ending User Code

Test Case Term – Source code in this area gets executed immediately after the UUT returns control to the harness. This usually includes examining data affected by specific test cases.

Harness Term – Source code in this area gets executed once per harness execution, immediately before exiting. This could include saving data to a file or other cleanup processing.

Additional Unit Specs – Specification portion of an additional code unit.

Additional Unit Bodies – Implementation portion of additional code unit, to be added to test harness.

There are two additional types of stub processing that you can add for all stubs; user code to process upon entry, and user code to process upon exit.

To enter user code for a *particular* stubbed subprogram, see "To Enter Configure Stub User Code" on page 418.

See "Environment Script Language" on page 454 for more information on the user code tags in the environment scripting language.

To Edit Environment User Code

To enter code for environment user code, follow these steps:

1. **Environment => User Code => Edit**.
2. Expand the node **"User Code"**
3. Choose which type of environment user code you need
4. Double-click in the **"Value"** column for that type

An edit box opens in which you can type, copy/paste, and test compile  source code.

To test compile this code, click the Test Compile button  . Test compiling in one cell of Environment User Code incorporates the source code in all other Environment User Code cells. Therefore if you have an error in any cell, the Test Compile Errors report is displayed.

The cell sports several icons and buttons:

 – contains code (not empty)

 – needs saving

 – needs to be compiled and linked

 – currently has an error

 – test compile the unit with the appendix user code

 – Save, or Save, Compile and Link (from the drop-down menu) the unit, but does not affect the Parameter Tree of a test case until environment is rebuilt

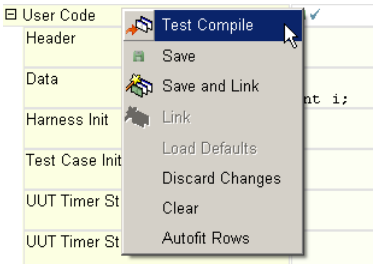
 – pop cell out into its own window for easier editing

 – pin, or preserve size

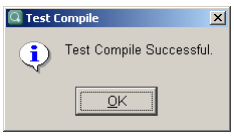
 – close the edit box for the cell

To Test Compile Environment User Code

There are several ways to test compile environment user code. In each individual cell, click the **Test Compile** button  to compile the code before it is saved to the test harness. Or, right-click a node and choose Test Compile. Either way, *all* environment user code is test compiled, not just the individual cell.

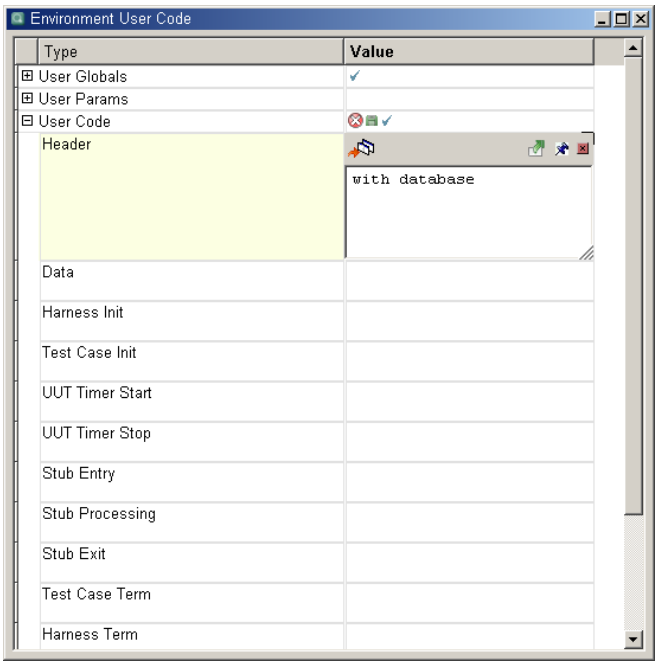


If successful, a message appears informing you, as shown in the following figure:

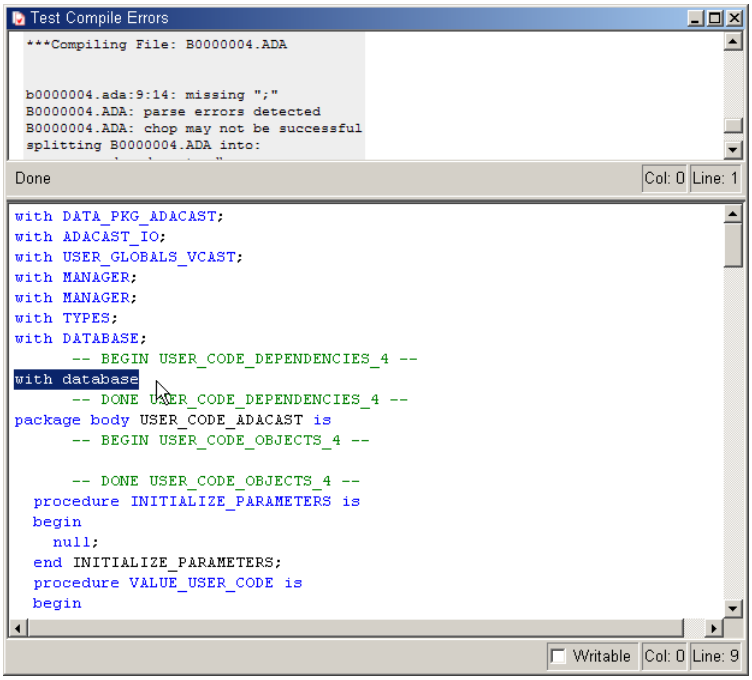


and any error state icons  are removed.

If there are compile errors, a dialog informs you and a window explains the errors. For example, suppose the following user code is entered in the Header section. Note the missing semicolon at the end of the line.



When the test compile button is clicked, the MDI window shows the following:



The top pane shows the compiler error and the line in which the error occurs.

```
***Compiling File: B0000004.ADA
b0000004.ada:9:14: missing ";"
B0000004.ADA: parse errors detected
B0000004.ADA: chop may not be successful
splitting B0000004.ADA into:
user_code_adacast.adb
```

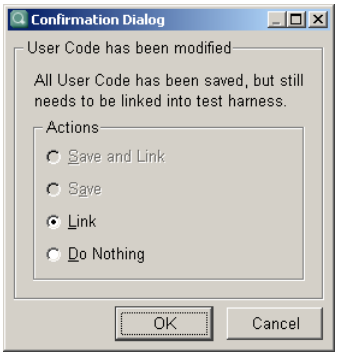
The lower pane shows the user code file with the temporary user code inserted. Use Ctrl+G (Goto Line) to go to the line with the error.

```
with DATA_PKG_ADACAST;
with ADACAST_IO;
with USER_GLOBALS_VCAST;
with MANAGER;
with TYPES;
with DATABASE;
-- BEGIN USER_CODE_DEPENDENCIES_4 --
with database
-- DONE USER_CODE_DEPENDENCIES_4 --
package body USER_CODE_ADACAST is
-- BEGIN USER_CODE_OBJECTS_4 --

-- DONE USER_CODE_OBJECTS_4 --
```

To close the Test Compile Errors window, use the  in the upper right corner or **File => Close**.

To Save Environment User Code

To save all sections of Environment User Code, click the **Save** button on the Toolbar , right-click a node and choose **Save**, or choose **File => Save**. The following dialog appears.

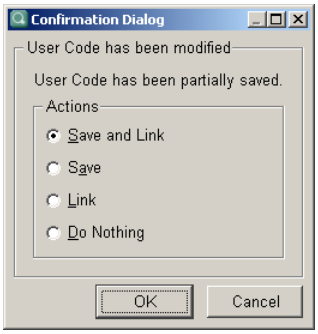


If you are ready to compile the user code and link it into the test harness, leave the radio button on **Link** and click **OK**.

If you plan to rebuild the environment anyway, put the radio button on **Do Nothing**, and click **OK** or **Cancel**.

If you attempt to close the Environment User Code window (using the  window control) and there

are any unsaved sections, the same dialog appears, but this time indicating that a Save needs to be performed.



Choosing the **Save and Link** radio button saves, compiles and links the user code into the test harness before closing the window. Choosing the **Save** radio button saves your changes before closing the window. Choose **Environment => User Code => Compile and Link later**. Choosing the **Link** radio button when some cells have been modified discards the changes and compiles and links the user code. Clicking the **Do Nothing** radio button discards your changes and closes the Environment User Code window.

User Globals

User Globals is a collection of global objects for use by the User Code functionality and for manipulation of objects of SYSTEM.ADDRESS type. You can also use this area to define temporary data objects. These objects can also be accessed in the Create New Environment wizard, User Code page, and are visible in the Parameter Tree of a test case.

Choose **Environment => User Code => Edit** to enter or modify the user globals. If you want to see the change in the Parameter Tree of a test case, you will need to rebuild the environment. Otherwise, compiling and linking the change into the test harness may suffice.

The Environment User Code dialog enables access to User Globals, User Parameters, Environment User Code, Unit Appendix User Code and Unit Prefix User Code.

Expand the node for “User Globals” to see the edit box of default user globals. The User Globals section provides a mechanism for user-defined types and objects to be included into the test harness. By default, the file has five integer objects, five floating point objects, five string objects, and an array of 200 integer elements. All data objects that are defined in the User Globals file when the environment is built can be manipulated as test data when building a test case.

You can add to these objects at the time you create the environment using the User Code page in the Create New Environment wizard.

The User Globals file is accessible in the Create New Environment dialog, the Environment User Code dialog, and the USER_GLOBALS_VCAST section of a test case’s Parameter Tree. The following list explains how to move between the three:

- Any User Globals added in the Create New Environment wizard, User Code page, are seen in the Environment User Code dialog, User Globals section and in the Parameter Tree of a test case.

> Adding any User Globals in the Environment User Code dialog, User Globals section, requires you to rebuild the environment before they are visible in the Parameter Tree of a test case. To see them in the Update Environment wizard, choose **Environment => Update Environment**.

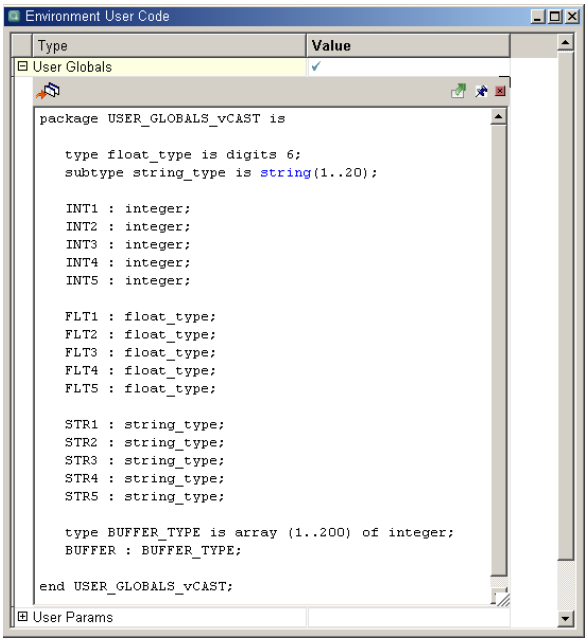
> The default version of the User Globals is delivered in source code format in the VectorCAST Installation Directory, subdirectory DATA, filename GLOBALS.ADA. If you modify the default file, the changes are reflected in the User Code page in the Create New Environment wizard for each new environment.

To Edit User Globals

To enter code for environment user code, follow these steps:

1. **Environment => User Code => Edit**.
2. Expand the node “User Globals”
3. Double-click in the yellow cell to edit

An edit box opens in which you can type, copy/paste, and test compile  source code.



To test compile this code, click the Test Compile button . Test compiling in one cell of Environment User Code incorporates the source code in all other Environment User Code cells. Therefore if you have an error in any cell, the Test Compile Errors report is displayed.

The cell sports several icons and buttons:

 – contains code (not empty)

 – needs saving

 – needs to be compiled and linked

 – currently has an error

 – test compile the unit with the appendix user code

 – Save, or Save, Compile and Link (from the drop-down menu) the unit, but does not affect the Parameter Tree of a test case until environment is rebuilt

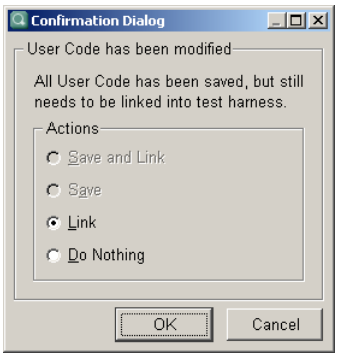
 – pop cell out into its own window for easier editing

 – pin, or preserve size

 – close the edit box for the cell

To Save User Globals

To save the User Globals, click the **Save** button on the toolbar  , right-click the User Globals node and choose **Save**, or choose **File => Save**. The following dialog appears.

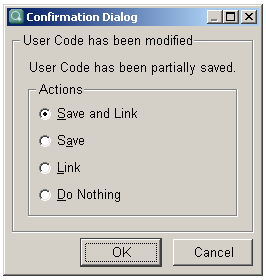


To see the changes to the User Globals in the Parameter Tree of a test case, you'll need to rebuild the environment. Put the radio button on **Do Nothing**, and click **OK**. Choose **Environment => Rebuild later**. If you have other Environment User Code changes, you probably want to use Link.

If you only need access to the new User Globals (without seeing them in the Parameter Tree), leave the radio button on **Link** and click **OK**.

To cancel the save operation without saving, click the **Cancel** button.

If you attempt to close the Environment User Code window (using the  window control) and there are any unsaved sections, the same dialog appears, but this time indicating that a Save needs to be performed.



Choosing the **Save and Link** radio button saves, compiles, and links the all environment user code into the test harness before closing the window. Choosing the **Save** radio button only saves your changes before closing the window; it does not compile and link. Choose **Environment => User Code => Compile and Link later**. Choosing the **Link** radio button when some cells have been modified discards their changes and compiles and links the user code. Clicking the **Do Nothing** radio button discards your changes and closes the Environment User Code window.

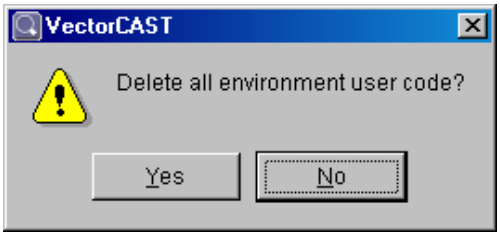
To Recompile Environment User Code

The **Environment => User Code => Compile** and Link command causes a compilation of the user code file to be performed. This command enables you to compile and re-link the modified USER_CODE_VCAST unit with the VectorCAST-created executable test harness. Use this command when you are ready to compile the environment user code after you have edited the environment user code, but you chose “Save Only” rather than “Save and Link” at the time you edited the user code.

```
clicast -e <env> ENvironment User COmpile

Compile and link Environment User Code into test harness.
```

To Remove Environment User Code

The **Environment => User Code => Clear** command will clear the environment user code from the test harness and the Environment User Code dialog. Once selected, you will see the following dialog box confirming that you wish to delete the environment user code.




```
clicast -e <env> ENvironment User CLear yes

Remove Environment User Code from test harness.
```

Example: To Perform Timing Analysis

As an example of using environment user code, assume a timing analysis scenario. Suppose you

would like to know how long it takes for the harness to execute, as well as how long each test case takes to execute. The following code would be added to each area of the Environment User Code dialog:

Header

```
with CALENDAR; use type CALENDAR.TIME;
with TEXT_IO;
with USER_GLOBALS_VCAST; use USER_GLOBALS_VCAST;
```

Data

```
package DURATION_IO is new TEXT_IO.FIXED_IO(CALENDAR.DAY_DURATION);
HARNESS_START : CALENDAR.TIME;
TESTCASE_START : CALENDAR.TIME;
```

Harness Init

```
HARNESS_START := CALENDAR.CLOCK;
```

Test Case Init

```
TESTCASE_START := CALENDAR.CLOCK;
```

Test Case Term

```
TEXT_IO.PUT ( "Time of testcase: " );
DURATION_IO.PUT ( CALENDAR.CLOCK - TESTCASE_START );
TEXT_IO.NEW_LINE;
```

Harness Term

```
TEXT_IO.PUT ( "Time of harness: " );
DURATION_IO.PUT ( CALENDAR.CLOCK - HARNESS_START );
TEXT_IO.NEW_LINE;
```

Example: To Read Test Data from a Database

In some test applications it is the case that you may have real-world data that you want to include into your test environment, and you do not want to have each tester enter all of this data in manually. Data of this sort is often numeric and the opportunity for errors is great.

In this case it is may be desirable for one person on the project to write a database interface that will access the data and to include this interface into all VectorCAST test harnesses, via the user code package. Consider this following example:

```
package TARGET_DATABASE is

type TARGET_RECORD is
 record
   ID    : integer;
   X_POSITION : YARDS_TYPE;
   Y_POSITION : YARDS_TYPE;
   COURSE  : DEGREES_TYPE;
   SPEED  : KNOTS_TYPE;
 end record;

type TARGET_INDEX_TYPE is new integer range 1..1000;
type TARGET_ARRAY is array (TARGET_INDEX)
 of TARGET_RECORD;

TARGETS : TARGET_ARRAY;
end TARGET_DATABASE;
```

In this case you could set up a portion of the target database using the VectorCAST test case utility, however it is unlikely that you would set up all 1000 items in the target array this way. To read the target data from a database you could write some code that could read the target data from a database and populate the TARGETS data object. We will not expand all of this code here except to show how it relates to the VectorCAST user code capability.

The following code would be added to two areas of the Environment User Code dialog:

Header

```
with DATABASE_IF;
with TARGET_DATABASE;
```

Harness Init

```
for I in TARGET_DATABASE.TARGETS'range loop
 TARGET_DATABASE.TARGETS (I) :=
DATABASE_IF.READ_TARGET_ITEM (I);
end loop;
```

User Parameters

User Parameters enable the tester to specify an object defined elsewhere in the harness and use it in place of a VectorCAST-generated parameter. The User Params section of the Environment User Code editor is where you indicate the association between the two objects. The syntax is:

```
<<unit.subprogram.parameter>> user_parameter
```

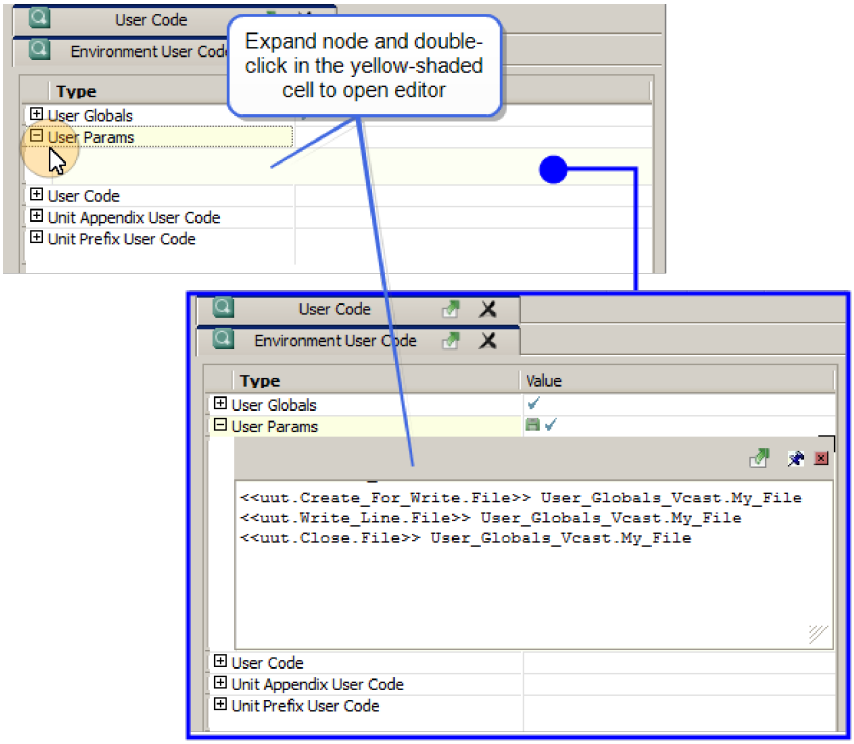
where unit, subprogram, and parameter are all VectorCAST-generated. For example,

```
<<uut.Create_For_Write.File>> User_Globals_Vcast.My_File
<<uut.Write_Line.File>> User_Globals_Vcast.My_File
<<uut.Close.File>> User_Globals_Vcast.My_File
```

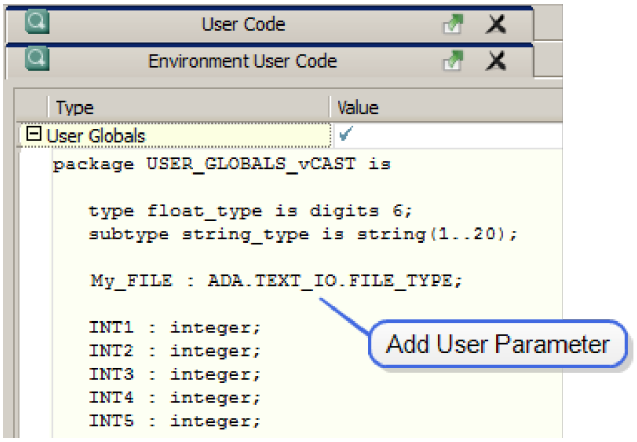
To enter code for environment user code, follow these steps:

1. From the Menu Bar, select **Environment => User Code => Edit**.
2. Expand the node **User Params**.
3. Double-click in the yellow cell to open the edit box.

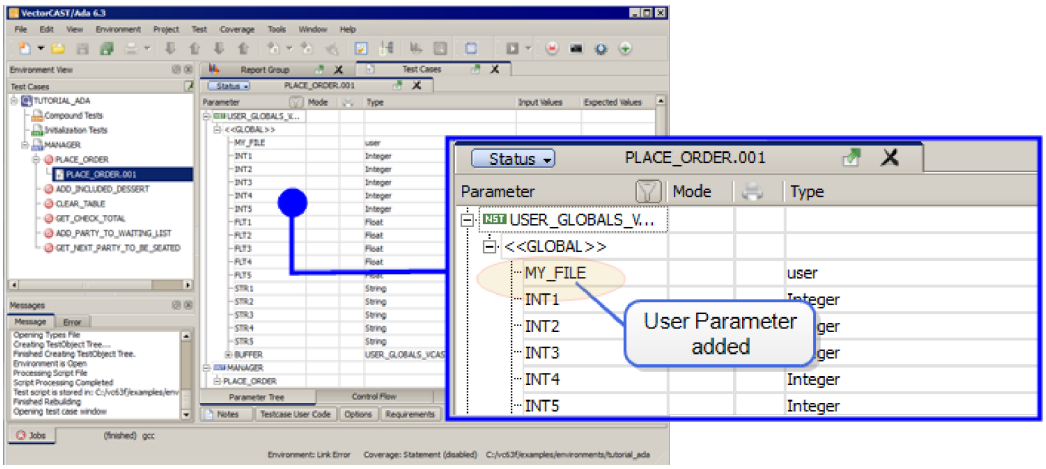
In our example, we will cut and paste the example code above into the edit box:



In addition, (for this example) you must declare your parameter in the User Globals section and then rebuild the environment.



As a result, the User Parameter MY_FILE is available in the USER_GLOBALS_VCAST section of the test cases in the environment. When you set an input or expected value for MY_FILE, you are actually setting it for the “File” parameter for the Create_For_Write, Write_Line, and Close functions.



Example: Chaining Data Values With User Parameters

In a typical Unit Test scenario for a File I/O package, you might have a function to create a file, write to the file, and close the file, You could use Parameter User Code (see "Parameter User Code" on page 412) to copy the file pointer data returned from ‘Create’ into the file pointer parameters for ‘Write’ and ‘Close’. However, in Ada 2005, Ada.Text_Io.File_type is “limited”, so copying is not allowed. You need a mechanism so that all three functions read/write the same object – that mechanism is User Parameters.

In this example, if the package specification for the Unit Under Test looked like:

```
with ADA.TEXT_IO;
package UUT is
 procedure CREATE_FOR_WRITE ( FILE : in out ADA.TEXT_IO.FILE_TYPE; FILENAME : in
   string );
 procedure WRITE_LINE ( FILE : in ADA.TEXT_IO.FILE_TYPE; LINE : in string );
 procedure CLOSE ( FILE : in out ADA.TEXT_IO.FILE_TYPE );
end UUT;
```

You could then create an environment using a script like:

```
      ENVIRO.NEW

      ENVIRO.NAME:ENV

      ENVIRO.COMPILER:GNAT

      ENVIRO.PARENT_LIB:.

      ENVIRO.UUT:UUT

      ENVIRO.USER_PARAMETERS:

      <<UUT.CREATE_FOR_WRITE.FILE>> user_globals_vcast.my_file

      <<UUT.WRITE_LINE.FILE>> user_globals_vcast.my_file

      <<UUT.CLOSE.FILE>> user_globals_vcast.my_file

      ENVIRO.END_USER_PARAMETERS:

      ENVIRO.USER_GLOBALS:

      with Ada.Text_Io;

      package USER_GLOBALS_vCAST is

      My_FILE : Ada.Text_Io.File_Type;

      end USER_GLOBALS_vCAST;

      ENVIRO.END_USER_GLOBALS:

      ENVIRO.END
```

In this scenario, all three subprograms would use the my_file object as a file pointer.

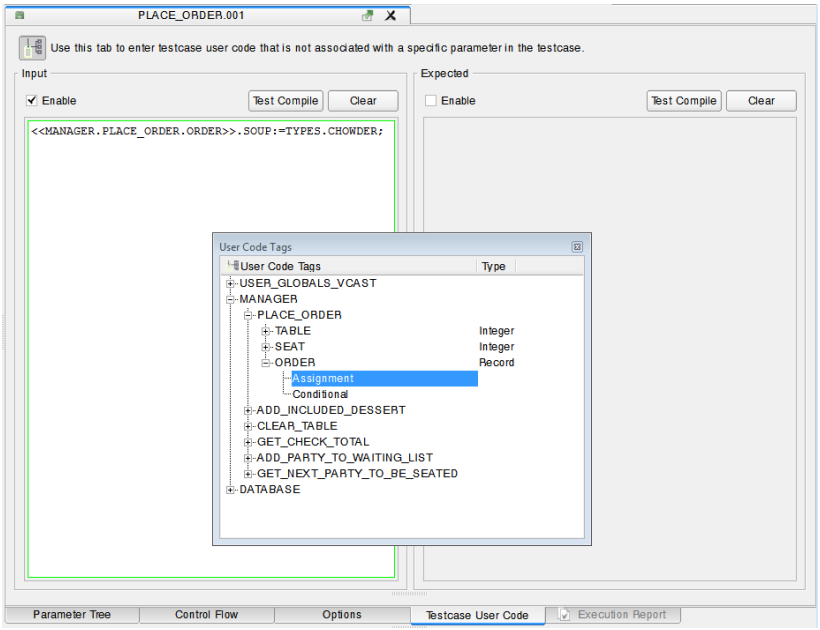
Test Case User Code

Test Case User Code is used to include input value or expected value user code which is not associated with a specific parameter. It applies to simple test cases, not compound test cases.

To Enter Test Case User Code

To enter input or expected test case user code, first insert or open a test case. Click the **Testcase User Code** tab along the bottom of the MDI window. Click the **Enable checkbox**, then you can enter code into the edit boxes.

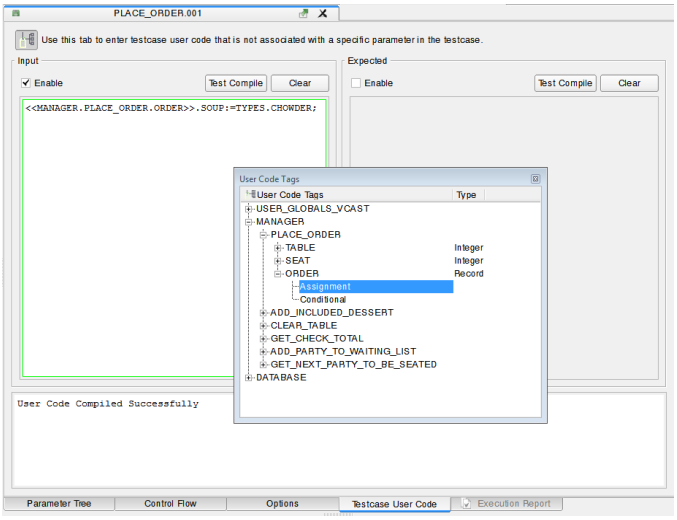
Use the User Code Tags window to easily insert the symbol for a parameter, using the correct notation. To use this feature in either the Input or Expected edit box, click the User Code Tags button  to open the window.



See also "User Code Tags" on page 392

Test Compile Testcase User Code

After entering user code in this way, clicking the **Test Compile Input** button or the **Test Compile Expected** button compiles the entered code, and displays any errors. Clicking either Test Compile button does not commit any changes to the testcase user code.



Once the Test Compile has completed, a message window is displayed at the bottom of the dialog. You may need to re-size the dialog or scroll the window to see this message window. You can move the splitter between sections of the window by dragging it up or down. If the test compile was successful, the message “User Code Compiled Successfully” is displayed. If there are errors, the compiler diagnostic messages are displayed.

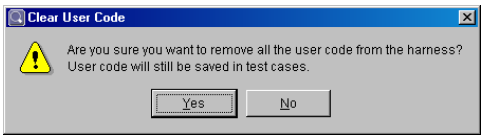
To delete all the text in the Input User Code section and start over, click the **Clear Input** button. To delete all the text in the Expected User Code section and start over, click the **Clear Expected** button.

To Clear Test Case User Code from Test Harness

The **Test => User Code => Clear All** command removes all test case user code from the test harness. The user code is not removed from the test cases (once it has been saved). This means that the next time a test case that contains user code is executed, its user code is added back into the user code unit.

The purpose of this command is to allow you to reset the user code unit in the test harness to its default (empty) state, and then selectively add the user code back in as you run tests. This process may be useful in cases where you experience unexpected results because of conflicts in your user code.

When you choose **Test => User Code => Clear All**, the following dialog box appears:



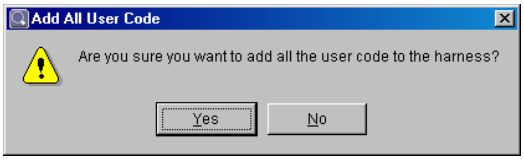
If you wish to remove all the user code from the harness, click **Yes**.

```
clicast -e <env> TESt User Clear

Remove all Test Case User Code from harness.
```

Add Test Case User Code Back Test Harness

This command adds the user code for all test cases that have user code into the user code unit of the test harness.

The following dialog box appears:



If you wish to add all the user code to the harness, click **Yes**.

```
clicast -e <env> TESt User Add

 Add all Test Case User Code into harness.
```

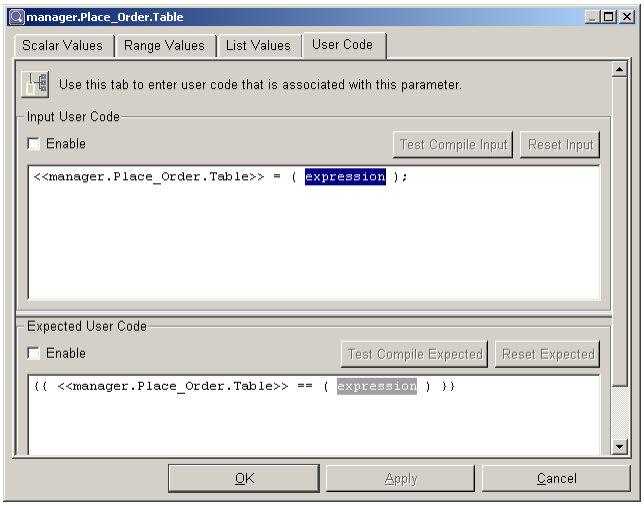
Parameter User Code

The Parameter User Code tab enables you to set a parameter value based on a function call, or to set

or verify a parameter value against some dynamic expression.

To Enter Parameter User Code

Double-click a parameter in the Parameter Tree to access the dialog for that parameter and then click the User Code tab. The Parameter User Code tab is separated into two sections. The Input User Code section is used to set objects to the value of the parameter. The Expected User Code section is used to verify the value of a parameter. Parameter object names are specified using the VectorCAST notation of <<unitname.subprogram.parameter>>.



See also "Setting Input and Expected Values" for details on the notation for parameter object names.

To Test Compile Parameter User Code

The code that you enter is automatically compiled into the test harness when the test case is run. The **Test Compile** buttons enable you to make sure your user code compiles before saving it. Any compiler errors will be displayed in the bottom pane of the window. The **Reset** buttons will delete any entered user code for the parameter and reset the dialog section to its initial configuration.

Input User Code Example

Input user code is used to set a value for a parameter before a test is run.

User code may consist of any number of lines of source code. For example, you may set an array parameter value using the following user code:

```
for I in 1..200 loop
<<unit.subprogram.a>>(I) := 201 - I;
end loop;
```

This code initializes an array “a” with the values 200 to 1.

```
<<USER_GLOBALS_VCAST.<<GLOBAL>>.BUFFER>> for <<unit.subprogram.a>>.
```

Expected User Code Example

Expected user code is used to provide an expression to test the value of the parameter after a test is run.

A boolean expression that is inside the {{ and }} markers is translated into a test expression that will output its results to the test results report. Variables are expanded as in input user code, and any user not inside {{ and }} markers is executed as (non-conditional) normal code.

For example, you could check the value of an array parameter using the following user code:

```
for I in 1..200 loop
{{ <<unit.subprogram.a>> (I) = (201-I) }}
end loop;
```

This code will test all 200 values of the array “a”.

Another User Code Example

In some cases, you may want to use a parameter that is returned from one function as an input to another function. Parameter user code allows you to do this. Assume a unit writes data to a file. To test the unit, you would want the file identifier returned from the creation routine to be passed as input to the write routine.

The package specification for the unit under test in this example might look like:

```
package DATA_FILE_OPERATIONS is
type FILE_TYPE is private;
procedure OPEN_DATA_FILE (
  NAME : in    string;
  FILE : in out FILE_TYPE);
procedure WRITE_DATA_TO_FILE (
  FILE : in out FILE_TYPE;
  DATA : in    integer);
procedure CLOSE_DATA_FILE ( FILE : in out FILE_TYPE );
function IS_OPEN ( FILE : FILE_TYPE ) return boolean;
private
type FILE_TYPE is new integer;
end DATA_FILE_OPERATIONS;
```

The package body for the unit under test in this example might look like this:

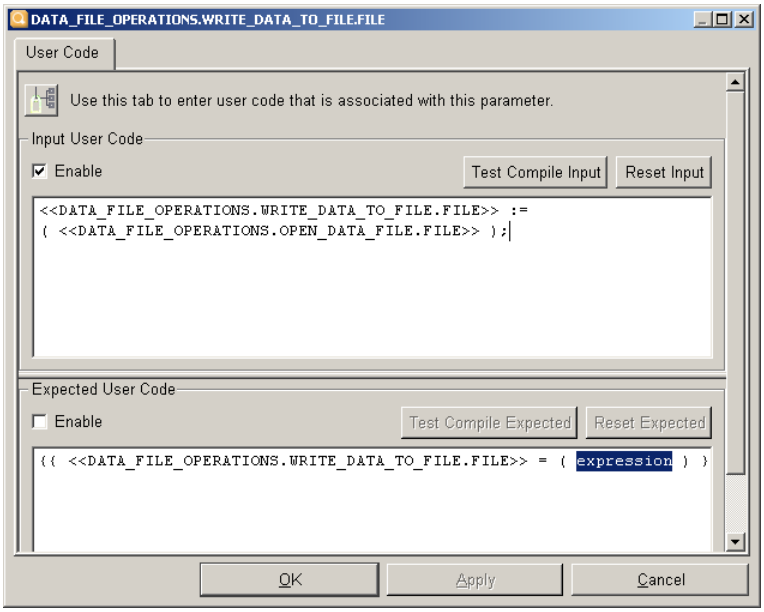
```
with TEXT_IO;
package body DATA_FILE_OPERATIONS is
FILES : array ( FILE_TYPE range 1 .. 10 ) of TEXT_IO.FILE_TYPE;
CURRENT : FILE_TYPE := 0;
procedure OPEN_DATA_FILE (
  NAME : in    string;
  FILE : in out FILE_TYPE) is
begin
```

```
  CURRENT := CURRENT + 1;
  FILE := CURRENT;
  TEXT_IO.CREATE ( FILES(FILE), TEXT_IO.OUT_FILE, NAME );
end OPEN_DATA_FILE;
procedure WRITE_DATA_TO_FILE (
  FILE  : in out FILE_TYPE;
  DATA : in    integer) is
begin
  TEXT_IO.PUT_LINE ( FILES(FILE), integer'image ( DATA ) );
end WRITE_DATA_TO_FILE;
procedure CLOSE_DATA_FILE ( FILE : in out FILE_TYPE ) is
begin
  TEXT_IO.CLOSE ( FILES(FILE) );
end CLOSE_DATA_FILE;
function IS_OPEN ( FILE : FILE_TYPE ) return boolean is
begin
  return TEXT_IO.IS_OPEN ( FILES(FILE) );
end IS_OPEN;
end DATA_FILE_OPERATIONS;
```

You would create a test case for OPEN_DATA_FILE, which, when called as the first slot in a Compound test case, would create a file with the specified name, which is one of the parameters.

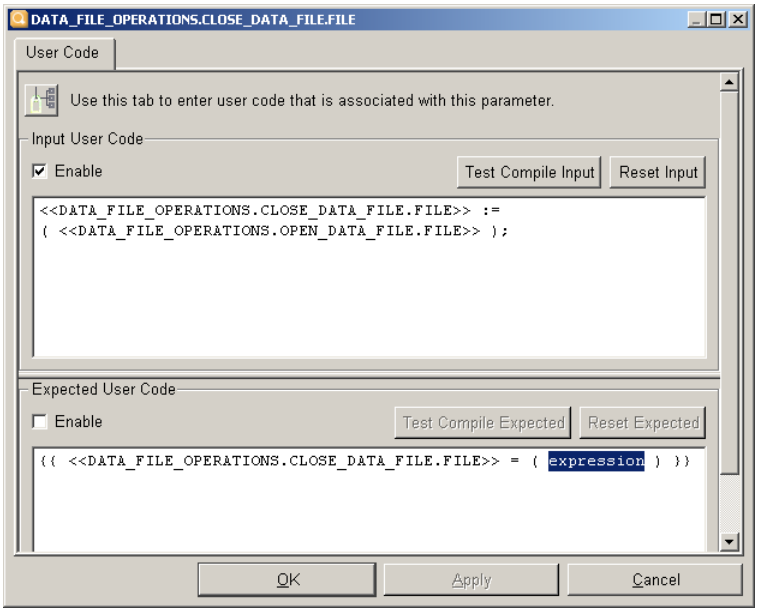
Next, create a test case for WRITE_DATA_TO_FILE. For the File parameter, we want to pass the file pointer from OPEN_DATA_FILE. This can be done by double-clicking on the File parameter, then using Input User Code as shown:

```
<<DATA_FILE_OPERATIONS.WRITE_DATA_TO_FILE.FILE>> := @
 ( <<DATA_FILE_OPERATIONS.OPEN_DATA_FILE.FILE>> );
```

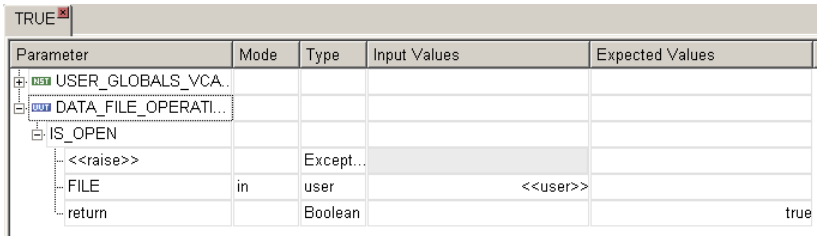


The Parameter User Code for a CLOSE_DATA_FILE test case is similar. We want to pass the same

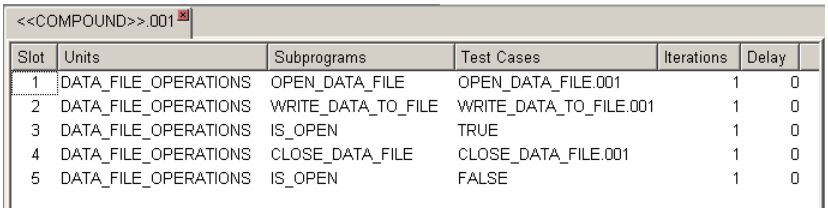
file pointer to this subprogram as well.



The test cases for IS_OPEN are simple; pass the file pointer (again), and set the Expected value to True for one test case, and False for the other.

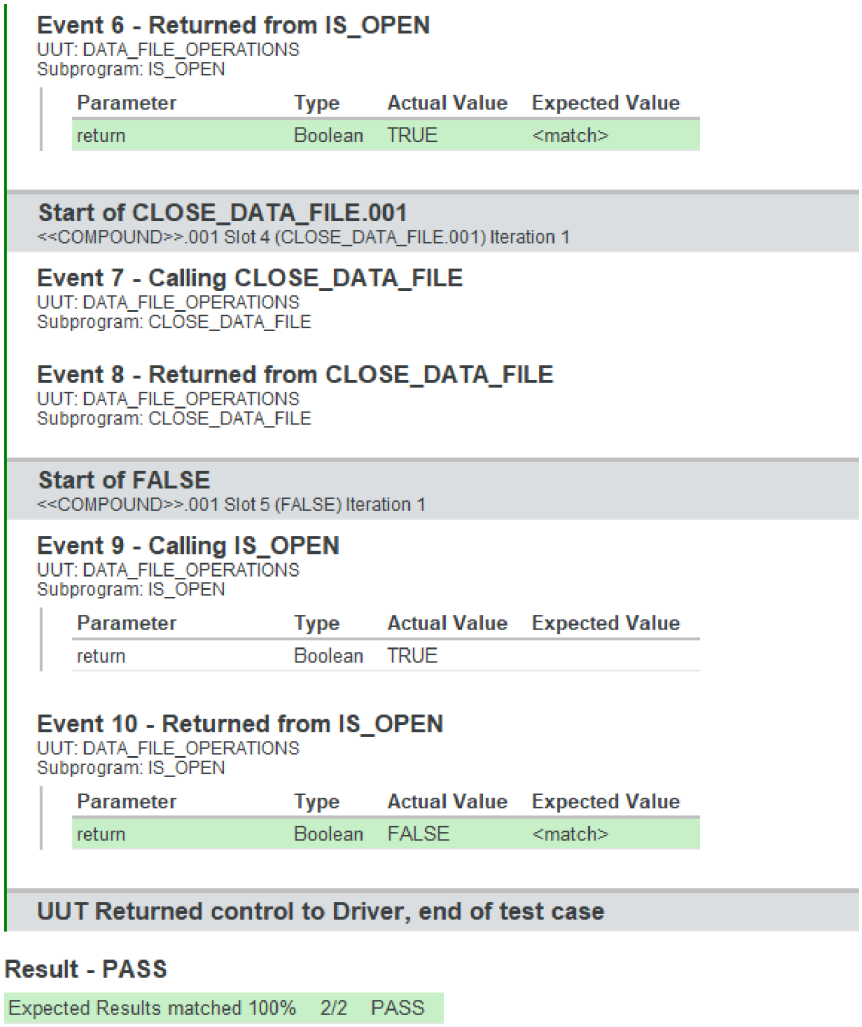


The Compound test case has these test cases as slots:



Slot	Units	Subprograms	Test Cases	Iterations	Delay
1	DATA_FILE_OPERATIONS	OPEN_DATA_FILE	OPEN_DATA_FILE.001	1	0
2	DATA_FILE_OPERATIONS	WRITE_DATA_TO_FILE	WRITE_DATA_TO_FILE.001	1	0
3	DATA_FILE_OPERATIONS	IS_OPEN	TRUE	1	0
4	DATA_FILE_OPERATIONS	CLOSE_DATA_FILE	CLOSE_DATA_FILE.001	1	0
5	DATA_FILE_OPERATIONS	IS_OPEN	FALSE	1	0

When the Compound test case is executed, the Execution Results shows that the file was open after the WRITE_DATA_TO_FILE subprogram returned, and was not open after the CLOSE_DATA_FILE subprogram returned.



Event 6 - Returned from IS_OPEN
UUT: DATA_FILE_OPERATIONS
Subprogram: IS_OPEN

Parameter	Type	Actual Value	Expected Value
return	Boolean	TRUE	<match>

Start of CLOSE_DATA_FILE.001
<<COMPOUND>>.001 Slot 4 (CLOSE_DATA_FILE.001) Iteration 1

Event 7 - Calling CLOSE_DATA_FILE
UUT: DATA_FILE_OPERATIONS
Subprogram: CLOSE_DATA_FILE

Event 8 - Returned from CLOSE_DATA_FILE
UUT: DATA_FILE_OPERATIONS
Subprogram: CLOSE_DATA_FILE

Start of FALSE
<<COMPOUND>>.001 Slot 5 (FALSE) Iteration 1

Event 9 - Calling IS_OPEN
UUT: DATA_FILE_OPERATIONS
Subprogram: IS_OPEN

Parameter	Type	Actual Value	Expected Value
return	Boolean	TRUE	

Event 10 - Returned from IS_OPEN
UUT: DATA_FILE_OPERATIONS
Subprogram: IS_OPEN

Parameter	Type	Actual Value	Expected Value
return	Boolean	FALSE	<match>

UUT Returned control to Driver, end of test case

Result - PASS

Expected Results matched 100% 2/2 PASS

Stub User Code

There are several different kinds of user code for stubs: you can enter user code for a parameter, a test case, or the whole environment. Each type of stub user code has distinct characteristics, as described below.

- **Parameter User Code** – To enter expected data or input values for a stubbed subprogram, use the Parameter Tree as you would for a UUT. The user code entered here us implemented via two procedure calls (one at the start of the stub, which is *expected user code,* and one at the end of the stub, which is *input user code*). Parameter User Code is entered in the User Code tab of the Parameter Dialog. User Code that is entered in this way will only be executed when the stubbed function is called. An application of this type of user code would be to set the value of a global variable when a particular stubbed subprogram is called, or to set the value returned from a stub to a dynamic expression (e.g. `foo()`).
- **Environment User Code** – To enter user code for all stubbed subprograms, use Environment User Code, Stub Processing. The Stub Processing section of the **Environment => User Code => Edit dialog** is implemented in the same was as Parameter and Test Case User Code, but it

is executed whenever any stubbed subprogram in the environment is called. For example, if you added stub processing user code that incremented a counter, at the end of each test execution you would have the total number of stubbed subprograms that were called during the test.

> To add user code for all stubbed subprograms to be processed upon entry to or exit from the stub, use the Stub Entry and Stub Exit section of the **Environment => User Code => Edit dialog**.

> **Configure Stubs** – Configure Stubs user code enables you to insert code directly into the definition of the stubbed subprogram. As a result, you have direct access to the named parameters of the function. An application of this type of user code is to cause a function to return a value based on the value of the input parameters.

To Enter Configure Stub User Code

Choose **Environment => Configure Stubs => Edit** to access the Configure Stubs dialog.

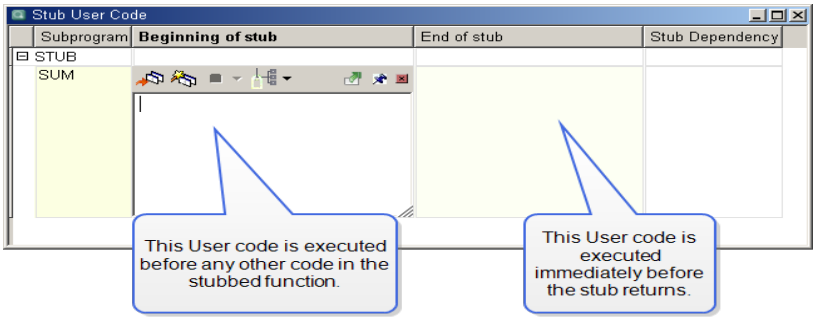
There are two editable fields on the Configure Stubs dialog. The first is marked “Beginning of Stub”, the second is marked “End of Stub.” Any code that is put into “Beginning of Stub” is placed before any other code in the stubbed function, and is the first code executed in the stub.

Any code that is put into “End of Stub” will be placed after all other code in the stubbed function (immediately preceding the `return()` statement).

To enter stub user code, follow these steps:

1. **Environment => Configure Stubs => Edit**.
2. Expand the node for the unit that contains the stub you are interested in.
3. Double-click in the yellow cell under “Beginning of Stub” or “End of Stub” to edit

An edit box opens in which you can type, copy/paste, and test compile  source code.



The cell sports several icons and buttons:

 – contains code (not empty)

 – needs saving

 – needs to be compiled and linked

 – currently has an error

 – test compile the unit with the appendix user code

 – Save, or Save, Compile and Link (from the drop-down menu) the unit, but does not affect the Parameter Tree of a test case until environment is rebuilt

 – pop cell out into its own window for easier editing

 – pin, or preserve size

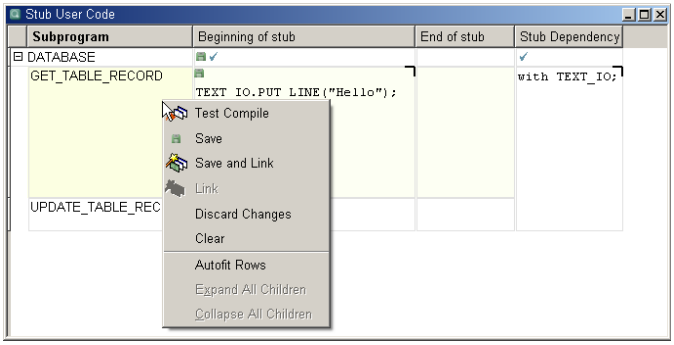
 – close the edit box for the cell

The Stub Dependency Column

The Stub Dependency column appears when dependents of the UUT are stubbed by implementation, rather than by prototype. In the Configure Stubs dialog, the Stub Dependency column appears to the right side of the dialog, and is used to add `with` statements needed by the configure stubs user code. Code entered via the Stub Dependency column is inserted into the global (file) scope of the stubbed unit.

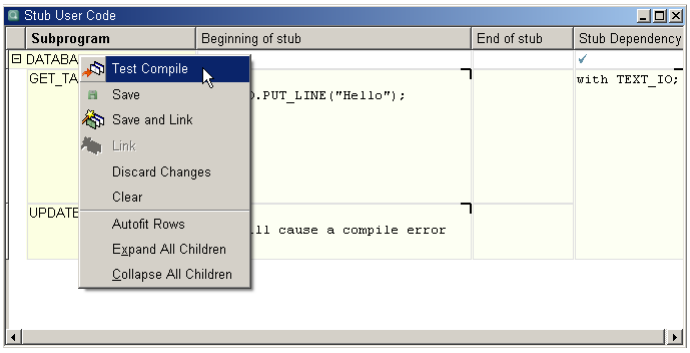
To Test Compile Stub User Code

There are several ways to test compile configure stubs user code. In each individual cell, click the **Test Compile** button  to compile the code before it is saved to the test harness. Or, right-click a node and choose **Test Compile**. Either way, *both* the Beginning of Stub and End of Stub user code is test compiled, not just the individual cell.

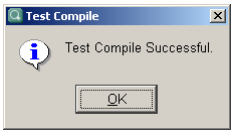


You can also right-click a node higher in the hierarchy, such as the unit or the top level, Stubbed Subprograms. In this case, choosing Test Compile performs a test compilation on all of the cells in the scope of the right-click. The light-yellow shading of the cells indicates the current scope of a right-click. For example, in the following screen shot, the user has right-clicked on the unit manager.c. Therefore, the test compile is performed over both subprograms in the unit, and both the Beginning of

Stub and End of Stub columns. As a result, the bad code in the second subprogram will cause a compile error.

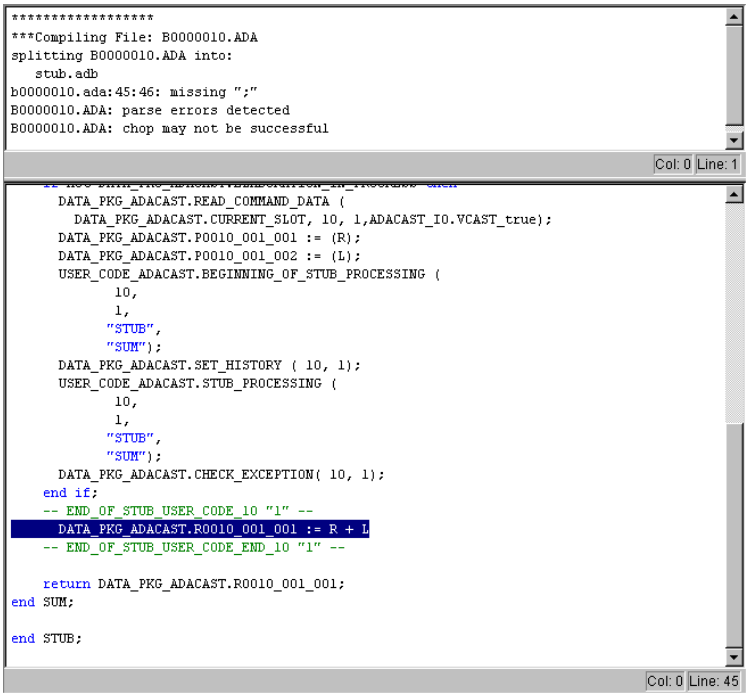


If successful, a message appears informing you, as shown in the following figure:



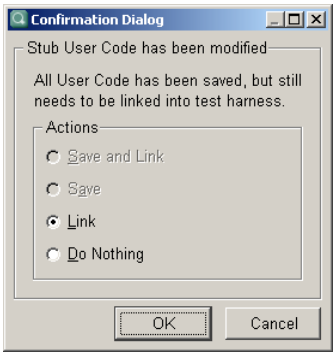
and any error state icons are removed.

If there are errors in your user code, clicking the Test Compile button opens a Stub Test Compile Errors window. The top pane shows the compile error; the lower pane shows the stub file. In the example below, the code “this will cause a compile error” is present. By using Ctrl+G in the lower pane you can easily go to the line number where the error occurred. To correct the error, close the Stub Test Compile Errors window by clicking the X in the upper right corner, edit the user code, and click the Test Compile button again until it compiles successfully.



To Save Stub User Code

To save all sections of Configure Stubs User Code, click the **Save** button on the Toolbar  , right-click a node and choose **Save**, or choose **File => Save**. The following dialog appears.

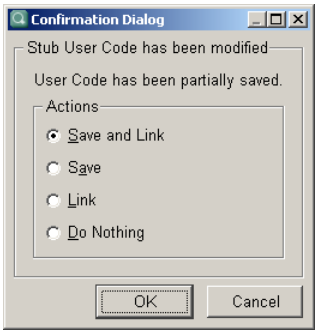


If you are ready to compile the user code and link it into the test harness, leave the radio button on **Link** and click **OK**.

If you plan to rebuild the environment anyway, put the radio button on **Do Nothing**, and click **OK**.

To cancel the save operation without saving, click the **Cancel** button.

If you attempt to close the Stub User Code window (using the  window control) and there are any unsaved sections, the same dialog appears, but this time indicating that a Save needs to be performed.



Choosing the **Save and Link** radio button saves, compiles and links the user code into the test harness before closing the window. Choosing the **Save** radio button saves your changes before closing the window. Choose **Environment => Configure Stubs => Compile and Link later**. Choosing the **Link** radio button when some cells have been modified discards the changes and compiles and links the user code. Clicking the **Do Nothing** radio button discards your changes and closes the Stub User Code window.

To Recompile Stub User Code

The **Environment => Configure Stubs => Compile and Link** command causes a compilation of stub user code file to be performed. Use this command when you are ready to compile the stub user code after you have edited it, but you chose **Save** rather than **Save and Link**.

```
clicast -e <env> ENvironment STub_user COmpile

Compile and link Configure Stub User Code into test harness.
```

To Remove Stub User Code

The **Environment => Configure Stubs => Clear command** will clear the stub user code from the test harness and the Configure Stubs dialog. Once selected, you see the following dialog box confirming that you wish to delete the environment user code.



VectorCAST will automatically re-compile the user code file after the user code has been removed.

```
clicast -e <env> ENvironment STub_user CLear YES

Remove Configure Stub User Code from test harness.
```

Example of Stub User Code

The following example demonstrates how to use the Configure Stubs feature to force the stub for a subprogram SUM to return the sum of its parameters. Assume that you have a subprogram **uut()** that calls **sum()** as in the following example.

uut.adb

```
with STUB;
package body UUT is
    function FUNC return integer is
        RET : integer := 0;
    begin
        for I in 1 .. 10 loop
            RET := RET + STUB.SUM ( I, I );
        end loop;
        return RET;
    end FUNC;
end UUT;
```

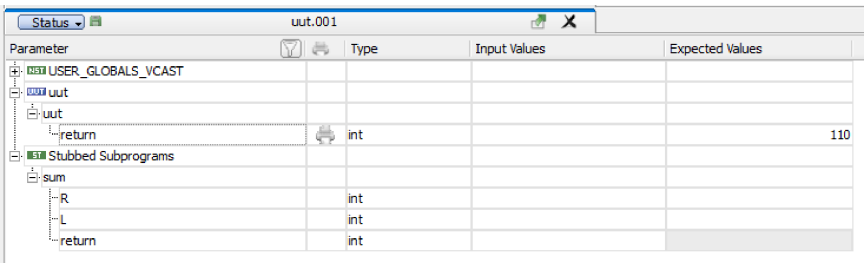
uut.ads

```
package UUT is
    function FUNC return integer;
end UUT;
```

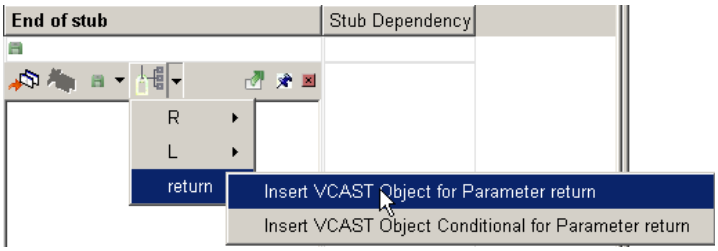
stub.ads

```
package STUB is
    function SUM ( R : integer;
                   L : integer )
                   return integer;
end STUB;
```

1. Create an environment using the source code listing above.
2. Insert a test case for the function uut. Enter the value 110 for the expected value for uut.return. We expect the value 110 to be returned from the stub because (1+1) + (2+2) + (3+3) + (4+4) + (5+5) + (6+6) + (7+7) + (8+8) + (9+9) + (10 + 10) = 110.



3. Choose **Environment => Configure Stubs => Edit**. Double-click in the End of Stub cell to open it for editing.
4. Click the User Code Tags button . Click the arrow to see the sub-menu.



5. VectorCAST inserts the following user code object:

```
<<STUB.SUM.return>>
```

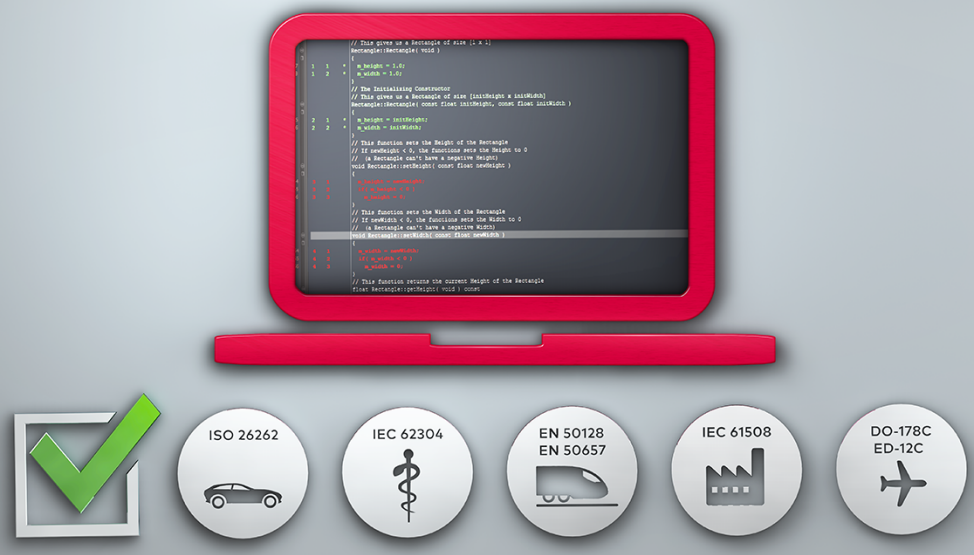
6. Edit to assign the return parameter R + L:

```
<<STUB.SUM.return>> := R + L;
```

Note: Rather than typing “R” and “L,” you can have VectorCAST insert the parameter names automatically. To do this, place your cursor ":=", click the arrow in the User Code Tags button, and choose **R => Insert Parameter R**.

7. Click the **Test Compile** button . If you have any compile errors, fix them.
8. Once the user code compiles successfully, click the **Save** button on the Toolbar .
9. In the message that appears, Click the **Save and Link** button.
10. Execute the test case FUNC.001. At the end of the execution results file, you see that the stub user code summed the inputs R and L each time it was called, and the returned value was 110, which is what was expected.

Tip: To see the returned value after each event, choose **Tools Options => Execute Tab**. Click the checkbox **Expand Report Parameters** and execute the test case again.



Using the Requirements Gateway

Requirements Gateway (RGW)

For assistance in migrating any legacy RGW databases to the new RGW data model (using `.json` files), contact Vector Tech Support at support@vector.com or visit https://support.vector.com/.

VectorCAST Requirements Gateway (RGW) provides traceability between software requirements and test cases and allows the import and mapping of requirements to test cases. Requirements Gateway provides a way to:

- Import testing requirements from your Requirements Management Tool (RMT) to a VectorCAST RGW repository
- Assign specific requirements to testcases in a unit test environment
- Export the resulting pass or fail status back to the RMT tool

In RGW, the requirements database is implemented using JSON, allowing the user to commit the repository to Source Control Management (SCM).

In addition to CSV files, VectorCAST RGW supports the following Requirements Management Systems:

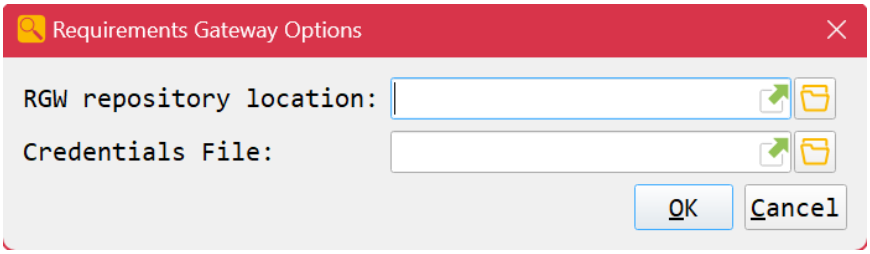
- Codebeamer
- IBM® Rational® DOORS®
- IBM ELM
- Jama
- Polarion®
- Visure

Using Requirements Gateway

The RGW repository location includes a directory named **`requirements_gateway`** which contains:

- **`export data`** - a directory which contains a separate file for each environment.
  - **`$ENV_NAME.json`** - this contains the test case data.
- **`settings.json`** - this contains the settings for the third party tool integration.
- **`requirements.json`** - this contains imported requirements data.

The location of the repository can be assigned to a directory by selecting **Tools => Requirements Gateway => Options...** from the Menu Bar.



The RGW credentials file can store the username and password for a specific user, to save time

when logging in to the RMT. The location of the credentials file can be assigned to a `.json` file by selecting **Tools => Requirements Gateway => Options...** from the Menu Bar.

Use the **Browse** button to navigate to either the location of the RGW repository or the location of the credentials file and then select the **OK** button. The paths to these values are typically specified as a full path, but can be a relative path.

Create a Requirements Repository

To use the Requirements Gateway, a location must be specified for the Repository. If no location is specified, a prompt opens allowing the user to select the location. A standard Choose a Directory dialog opens, prompting the user to enter the path to the directory.

A repository location can also be specified by selecting **Tools => Requirements Gateway => Options...** from the Menu Bar. In the **RGW Repository Location** field, enter the path to the repository and select the **OK** button.

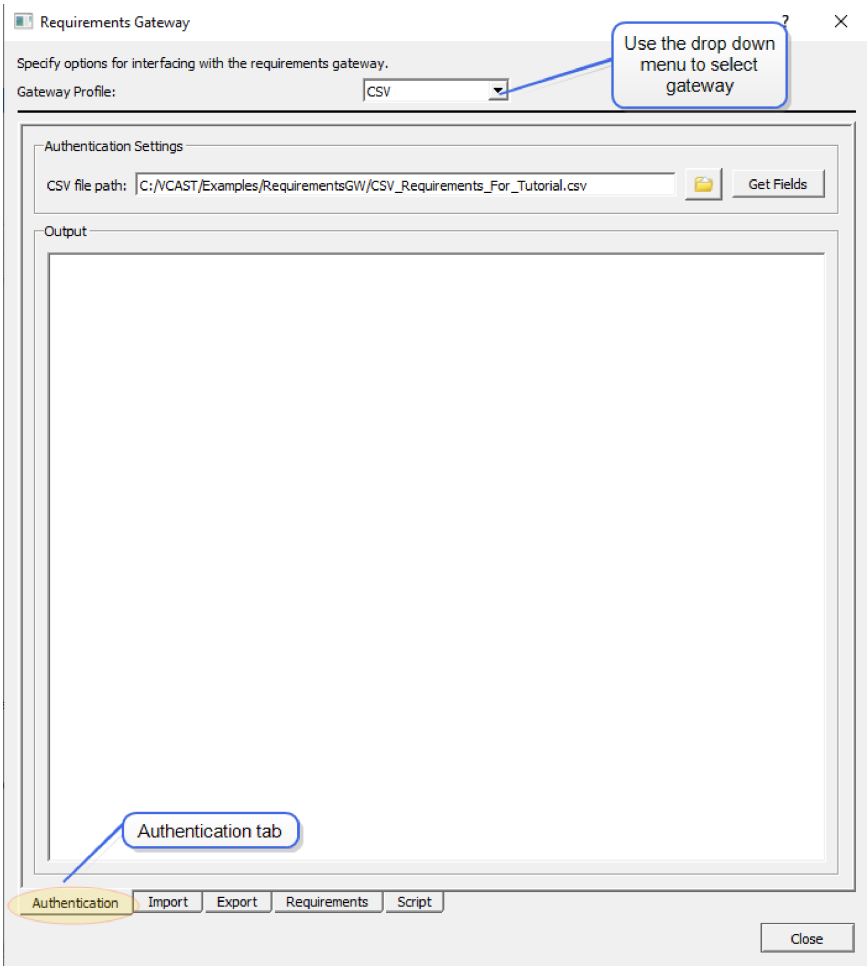
To open Requirements Gateway, select one of the following methods:

- Find the **Open Requirements Gateway**  button on the Toolbar. Select the drop-down menu next to the **Open Requirements Gateway** on the Toolbar and select the name of the gateway.
- Select **Tools => Requirements Gateway => <gateway name>** from the Menu Bar.

The Requirements Gateway dialog opens.

The Authentication Tab

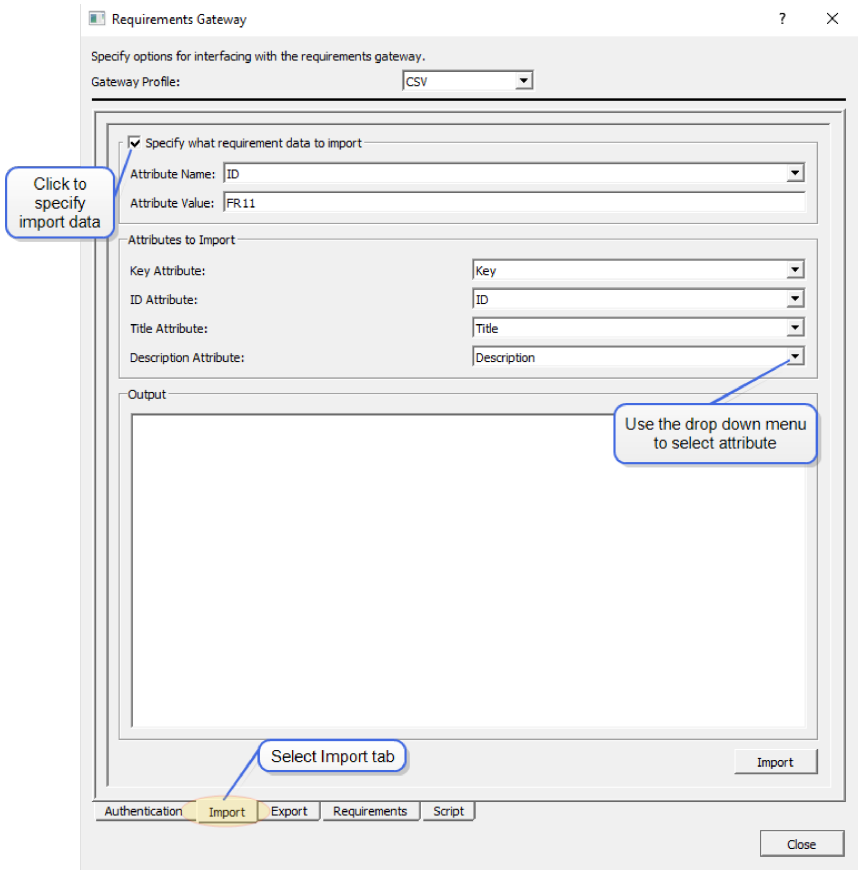
The **Authentication** tab allows the user to specify configuration options. Configuration options vary according to the gateway selected. See "Working With Supported Gateways" on page 437 to identify the configuration options for your subsystem. Use the **Gateway Profile** drop-down menu to select the gateway.



The Import Tab

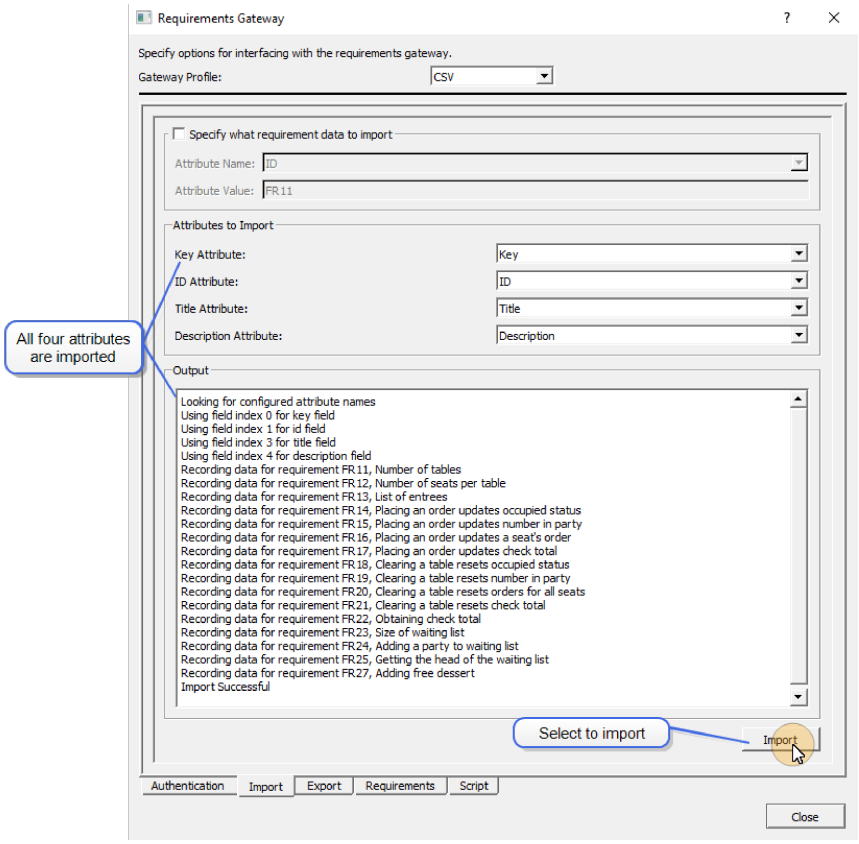
To import the requirements from the Requirements subsystem into the Repository, select the **Import** tab at the bottom of the Requirements Gateway dialog.

Configure the system to identify the data attributes to be imported. Available attributes vary according to the gateway selected. See "Working With Supported Gateways" on page 437 to identify the attributes required for your gateway. Our example below is based on a CSV gateway.



Note that the imported requirement data may be filtered by selecting the **Specify what requirement data to import** checkbox. When this option is selected, only objects which have an attribute with a specified **Attribute name** and **Attribute value** are imported.

Select the **Import** button to import the requirement data from the requirements subsystem. In our example, the option to specify what data to import was not selected, and all four attributes were imported.



The Export Tab

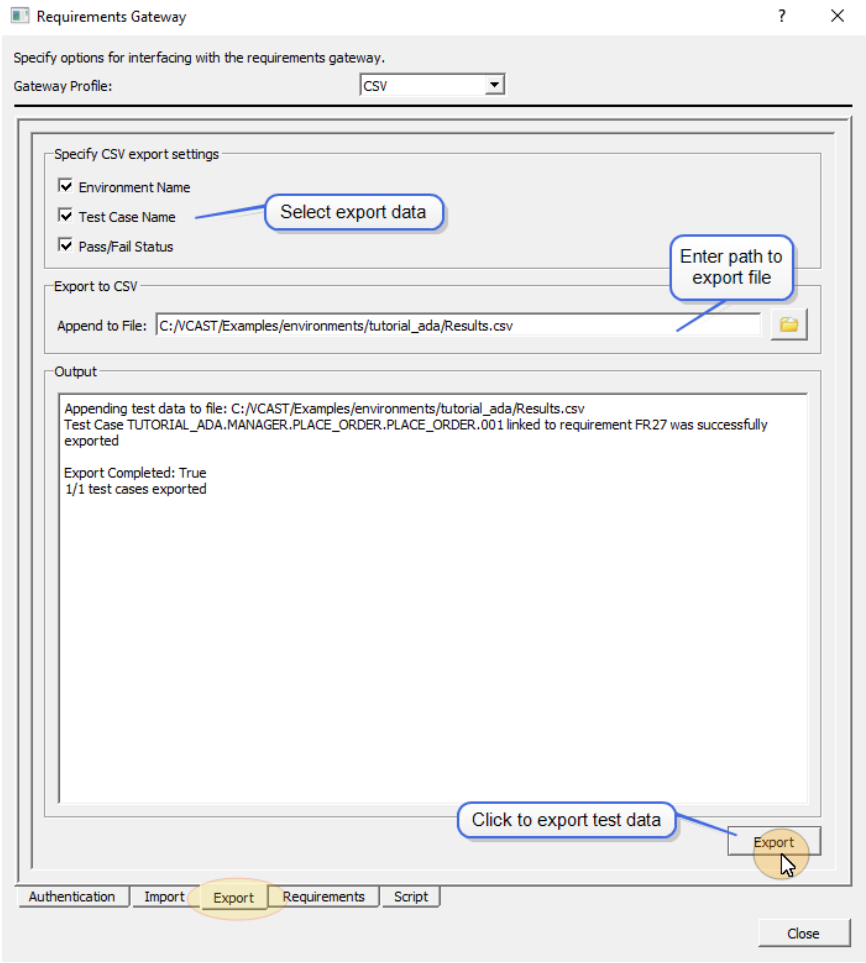
Once test case results have been obtained, the results can be exported to a new file. To export, select the **Export** tab on the Requirements Gateway dialog.

Available export settings vary according to the gateway selected. See "Working With Supported Gateways" on page 437 to identify the settings available for your gateway.

Our example shows export settings for a CSV gateway. The following test data can be exported:

- Environment name
- Test case name
- Pass/fail status

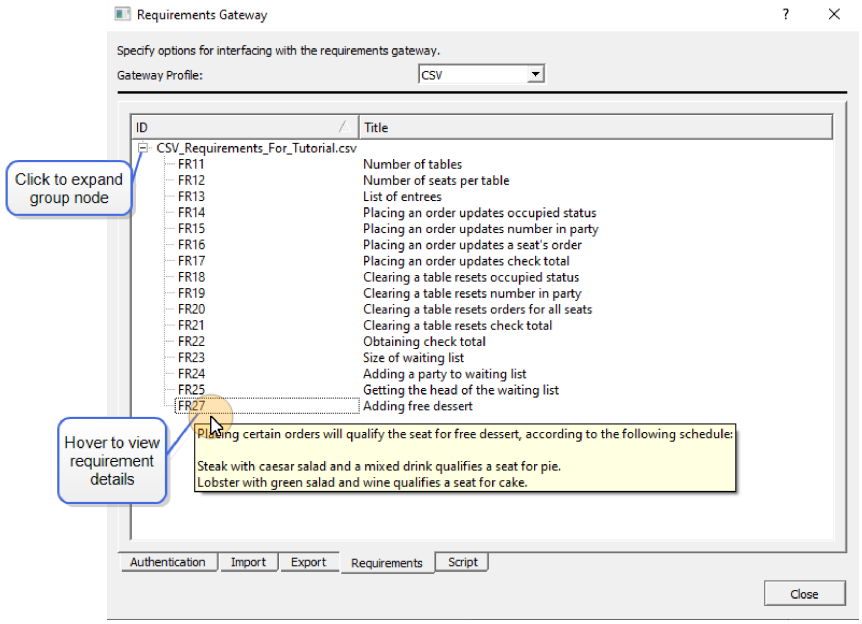
Use the browser dialog to enter the path to the export file.



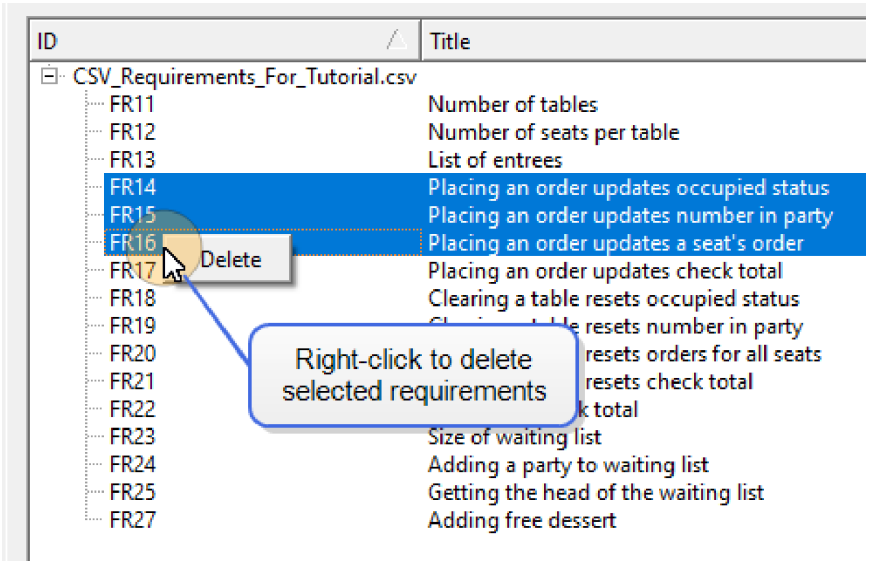
Select the **Export** button. The test data is appended to the export file.

The Requirements Tab

Select the **Requirements** tab at the bottom of the Requirements Gateway dialog to view a listing of all imported requirements. Hovering over a requirement displays a tool tip containing a detailed description of the requirement.

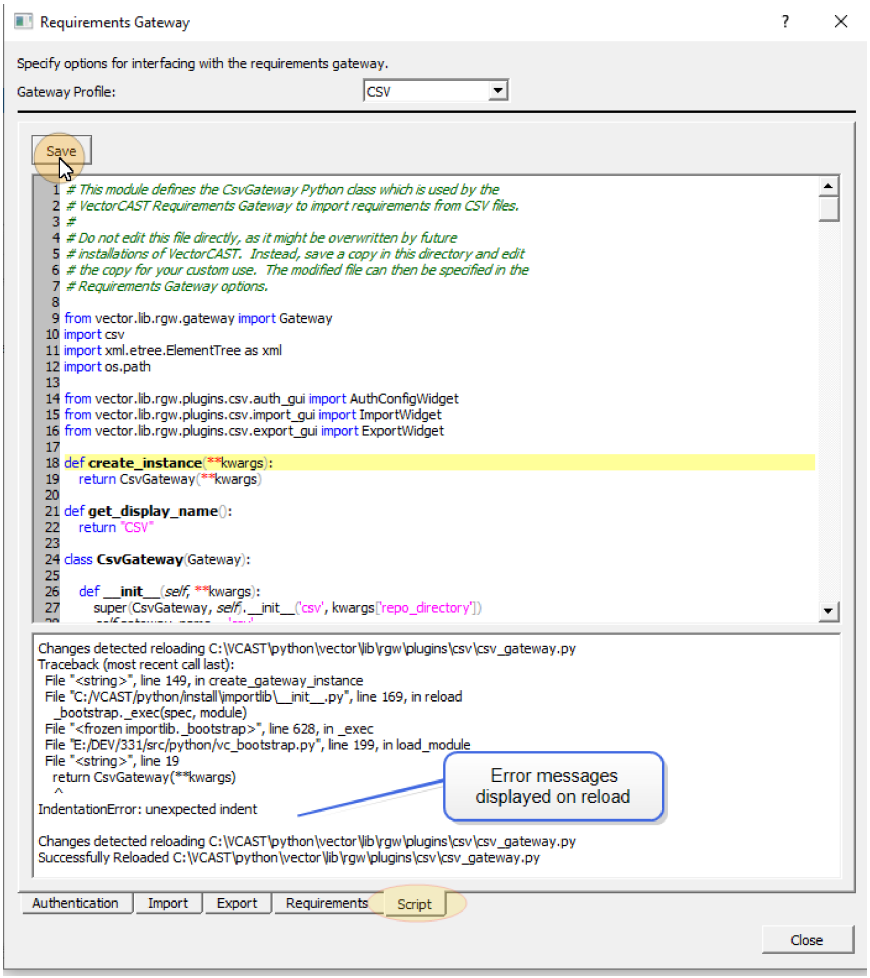


Single requirements, multiple requirements and entire requirement groups can be deleted from the repository. To delete a single requirement, select the requirement and select **Delete** from the right-click menu. To delete multiple requirements from the repository, use **ctrl-click** or **shift-click** to select the requirements and select **Delete** from the right-click menu. An entire group of requirements can be deleted by selecting the group node and selecting **Delete** from the right-click menu.



The Script Tab

To view and edit the **`csv_gateway`** Python script used to import requirements from the CSV file, select the **Script** tab located at the bottom of the Requirements Gateway dialog.



Once edits are complete, select the **Save** button to save changes and reload the Python file. If VectorCAST is unable to reload the file after editing, error messages are displayed in the message box below the script.

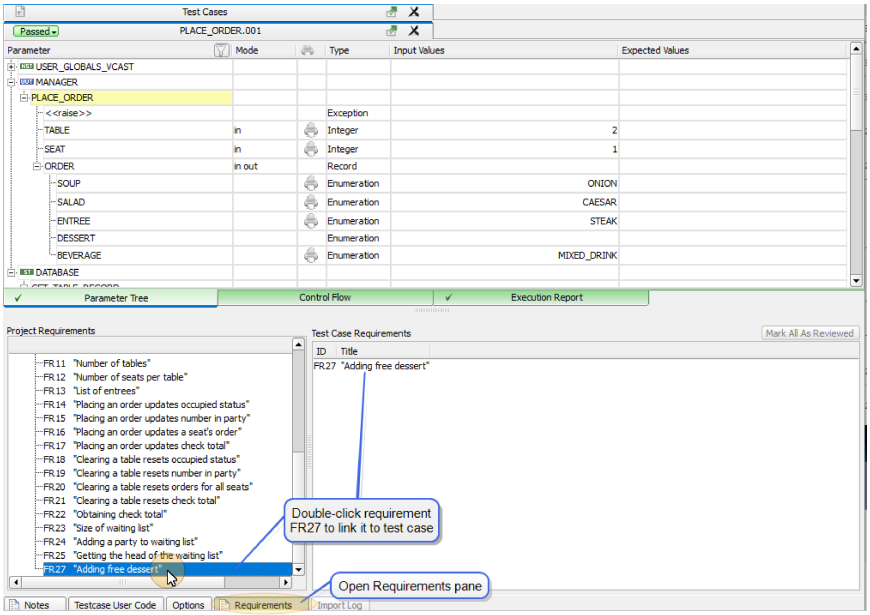
Note: Best practice is to not edit this file directly. The file may be overwritten by future VectorCAST installations. Instead, make a copy of the file and save it in the **`$VECTORCAST_DIR/python/vector/lib/rgw/plugins`** directory. Modify the file as required. Your custom file can then be specified in the Requirements Gateway options.

Link Requirements to Test Cases

To view or edit the requirements associated with a test case, open the test case and click on the **Requirements** tab in the Test Case Editor. The **Project Requirements** pane in the lower left of the MDI window contains a listing of all requirements imported into the repository. The **Test Case Requirements** pane in the lower right of the MDI window shows the requirements associated with the selected test case.

To link a requirement to a test case, double-click on the requirement in the **Project Requirements** pane. The requirement will be added to the list in the **Test Case Requirements** pane. In our example, we have linked the requirement **FR27 Adding free dessert** to test case PLACE_

ORDER.001.

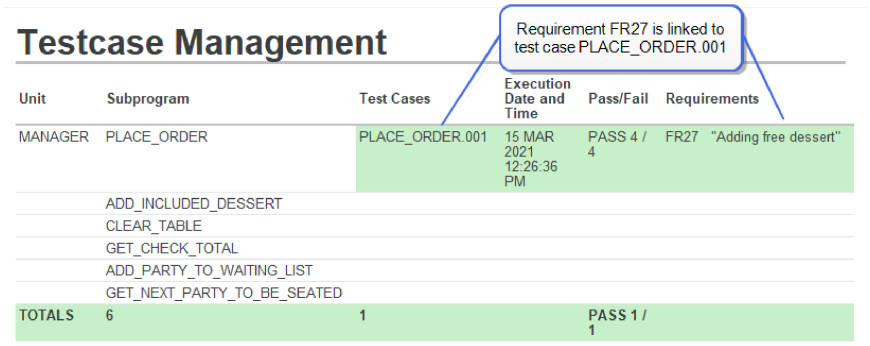


To remove a requirement from a test case, double-click on the requirement in the **Test Case Requirements** pane.

Save any changes by clicking on the **Save** icon in the Toolbar.

Once requirements have been associated with a test case, run the test case to generate a result. The result is then linked to that requirement in the repository.

Requirements that are associated with a test case are printed in the **Test Case Management Report**, which can be accessed by selecting **Test => View => Test Case Management Report** from the Menu Bar.



Tracking Requirement Change Impacts on Test Cases

VectorCAST provides users with the ability to track how VectorCAST test cases are impacted when requirements change.

The work flow for tracking the impact of requirement changes on test cases consists of the following:

1. Configure the Requirements Gateway and import your requirements for the first time.

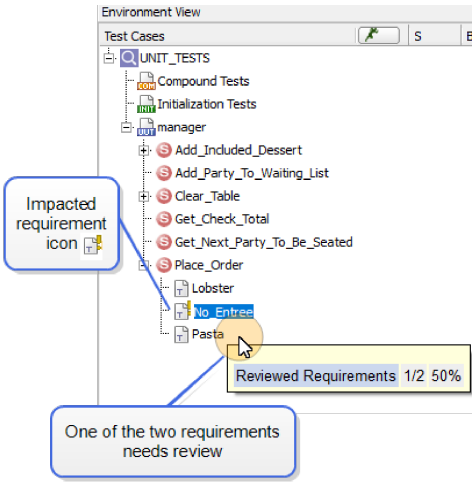
2. Link these requirements to VectorCAST test cases.

3. Make a change to some requirements in your chosen Requirement Management Tool (RMT).

4. In Requirements Gateway, re-import the requirements.

If VectorCAST detects that a requirement has been updated and it is linked to a test case in a VectorCAST unit test environment, then the requirement is considered "impacted". Any test cases linked to impacted requirements are then marked as needing review.

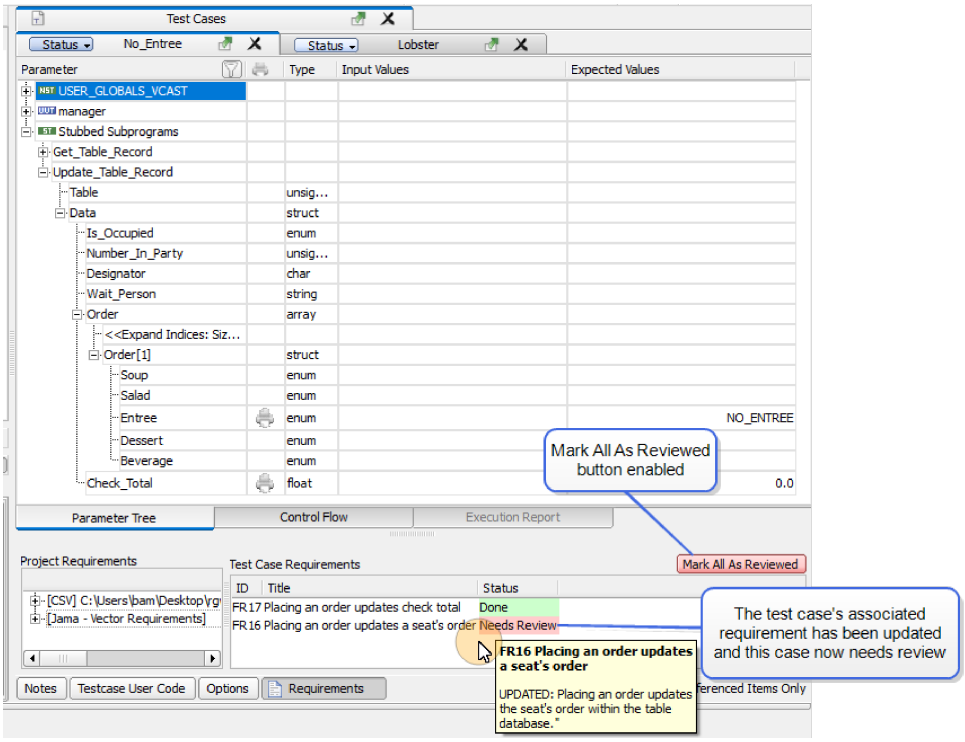
In the GUI, test cases linked to impacted requirements that need review are displayed in the Test Case Tree with a yellow "!" icon  . The test's tooltip also indicates how many requirements need review.



Note that the user is then prevented from changing the list of linked requirements and from exporting test case pass / fail data back to the RMT.

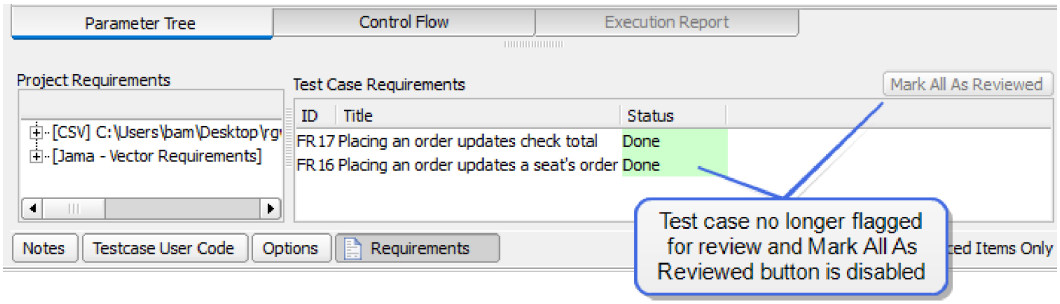
5. Verify that the test case still satisfies the test's associated requirement(s), perhaps modifying Input or Expected Values and executing as needed.

When such a test case is opened in the Test Case Editor, the impacted requirements are displayed as having a "Needs Review" status, and a **Mark All As Reviewed** button is enabled.



6. When done, mark the test case as having been reviewed.

Clicking the **Mark All As Reviewed** button clears the "Needs Review" status of the test case, and marks all impacted requirements as "Done" in that test case.



This action then frees up the restrictions on the test case, allowing the user to once again change the list of linked requirements and export the pass / fail status back to the RMT.

In the GUI, test cases linked to impacted requirements that have been reviewed are displayed in the Test Case Tree with a green check mark icon icon .

The Mark As Reviewed CLICAST command marks a specified test case as having been reviewed and approved, and clears the "Marked for Review" flag.

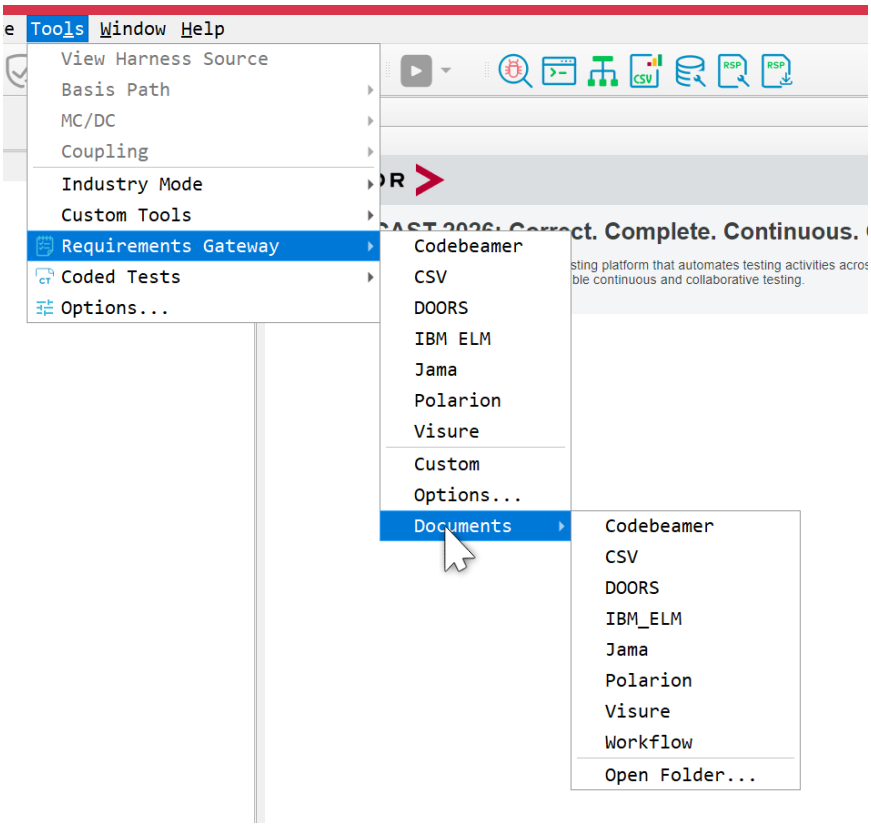
```
clicast -lc -e <env> -u <unit> -s <sub> -t <test> RGW Testcase MArk_as_
reviewed

Where a Test Case is flagged as having been impacted by a Requirements
change, this marks the selected Test Case as reviewed.
```

Working With Supported Gateways

VectorCAST RGW supports several Requirements Management Systems, including:

- CSV
- Codebeamer
- IBM® Rational® DOORS®
- IBM ELM
- Jama
- Polarion®
- Visure

Instructions for these integrations are available from the Menu bar by selecting **Tools => Requirements Gateway => Documents** and then selecting the appropriate system from the menu.



To view the document in the system's Markdown (MD) viewer, select the **Open Folder...** option and then select the desired guide. For an enhanced Markdown rendering experience, the "Markdown

Preview Enhanced" extension in Visual Studio Code provides excellent formatting and display capabilities.

The following sections discuss in detail how to set up and use the Requirements Gateway with the CSV Gateway.

Example Work Flow for RGW Command Line

The following example illustrates the process of setting up a CSV Requirements subsystem and importing and exporting requirements using the RGW command line.

This example covers:

- Creating and configuring the Repository
- Initializing the Gateway
- Importing requirements
- Linking requirements to test cases
- Executing test cases
- Exporting test data

This work flow uses the VectorCAST example environment Tutorial for Ada. Before starting the work flow example, you can automatically build the environment by selecting **Help => Example Environments => Ada => Tutorial for Ada** from the Menu Bar. Be sure that the **TUTORIAL_ADA** environment is closed before beginning the work flow. Close the environment by selecting **File => Close Environment** from the Menu Bar.

The .csv file used in our example is located in the VectorCAST installation directory: **$VECTORCAST_DIR/Examples/RequirementsGW/CSV_Requirements_For_Tutorial.csv**.

Step 1 - Set the Repository

From a Windows command prompt, navigate to your VectorCAST Working Directory (which in our example is located at **C:\VCAST\examples\environments\tutorial_ada**). Enter the following commands to first create a directory for the repository. For our example, the repository directory is named **REPO3**. Then, set the path to the repository directory:

```
$ %VECTORCAST_DIR%\mkdir REPO3

$ %VECTORCAST_DIR%\clicast -lada Options VCAST_REPOSITORY
C:\VCAST\examples\environments\tutorial_ada\REPO3

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
```

Note that for the purposes of our example, the path to the new repository is located in the working directory. In actual practice, the repository is normally located in a different directory.

Step 2 - Initialize the Gateway

Enter the following commands to initialize the gateway:

```
$ %VECTORCAST_DIR%\clicast -lada RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Warning: you will not be able to rebuild your environment without search
directories.

$ %VECTORCAST_DIR%\clicast -lada RGw INitialize
C:\VCAST\examples\environments\tutorial_c\REPO3

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Attempting to create RGW Repository...
Creating RGW Repository in 'C:\VCAST\examples\environments\tutorial_
ada\REPO3'
```

Step 3 - Set Configuration Options

Next, we configure the newly created Repository. The recommended workflow is to enter these options from the GUI, using the option selection found on the Requirements Gateway dialog's Authentication and Import tabs.

From the command line, enter the following configuration options:

```
$ %VECTORCAST_DIR%\clicast -lada RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Current Gateway was set to 'CSV'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set csv_selective_fetch 1

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Key 'csv_selective_fetch' set to '1' for Subsystem Profile 'csv'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set csv_path
C:\VCAST\Examples\RequirementsGW\CSV_Requirements_For_Tutorial.csv

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Key 'csv_path' set to 'C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG' for Subsystem Profile 'csv'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set id_attribute ID
```

```
VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Key 'id_attribute' set to 'ID' for Subsystem Profile 'csv'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set key_attribute Key

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Key 'key_attribute' set to 'Key' for Subsystem Profile 'csv'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set title_attribute Title

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Key 'title_attribute' set to 'Title' for Subsystem Profile 'csv'.


$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set description_attribute
Description

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Key 'description_attribute' set to 'Description' for Subsystem Profile
'csv'.
```

Step 4 - Import Requirements

Now that all the required system settings have been configured, import the requirements from the CSV Requirements system into the Repository by entering:

```
$ %VECTORCAST_DIR%\clicast -lada RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Current Gateway was set to 'CSV'.

$ %VECTORCAST_DIR%\clicast -lada RGw IMport

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
********* starting import *********
********* import finished *********
```

Step 5 - Link Requirements to a Test Case

Next, we link specific requirements to a particular test case. In our example we will link the requirements FR11, FR12 and FR13 to the test case PLACE_ORDER.001 using the following commands:

```
$ %VECTORCAST_DIR%\clicast -lada -e TUTORIAL_ADA -u MANAGER -s PLACE_ORDER
-t PLACE_ORDER.001 RGw Testcase Link FR11

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Opening Environment
Opening Parameter/Global File
Opening Types File
Environment is Open
Successfully linked Test Case 'PLACE_ORDER.001' to Requirement with Key
'FR11'.

$ %VECTORCAST_DIR%\clicast -lada -e TUTORIAL_ADA -u MANAGER -s PLACE_ORDER
-t PLACE_ORDER.001 RGw Testcase Link FR12
.
.
.
Successfully linked Test Case 'PLACE_ORDER.001' to Requirement with Key
'FR12'.

$ %VECTORCAST_DIR%\clicast -lada -e TUTORIAL_ADA -u MANAGER -s PLACE_ORDER
-t PLACE_ORDER.001 RGw Testcase Link FR13
.
.
.
Successfully linked Test Case 'PLACE_ORDER.001' to Requirement with Key
'FR13'.
```

Step 6 - Run the Test Case

Now we will run the test case and gather the execution and coverage data. To run a test case from the command line, enter:

```
$ %VECTORCAST_DIR%\clicast -lada -e TUTORIAL_ADA -u MANAGER -s PLACE_ORDER
-t PLACE_ORDER.001 Execute Run

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Opening Environment
Opening Parameter/Global File
Opening Types File
Environment is Open
Preparing Test Data
Running Test Case
Running Test with command: C:\VCAST\examples\environments\tutorial_
```

```
ada\TUTORIAL_ADA\UUT_INST.EXE
   Running Command: C:/Users/****\.vcastProcessMonitorLogs\1_Command
   Command Returned Exit Code: 0
Processing Execution Data
Updating Coverage Data
PLACE_ORDER.001 Test Execution Complete
TEST RESULT: pass
```

Step 7 - Configure the Export Settings

From the command line, enter the following export configuration options:

```
$ %VECTORCAST_DIR%\clicast -lada RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Current Gateway was set to 'CSV'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set csv_export_path
C:\VCAST\examples\environments\tutorial_ada\Results.csv

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\Environments\ADACAST_.CFG
Key 'csv_export_path' set to 'C:\VCAST\examples\environments\tutorial_
ada\Results.csv' for Subsystem Profile'csv'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set test_name_attribute true

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\Environments\ADACAST_.CFG
Key 'test_name_attribute' set to 'true' for Subsystem Profile'csv'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set test_status_attribute
true

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\Environments\ADACAST_.CFG
Key 'test_status_attribute' set to 'true' for Subsystem Profile'csv'.

$ %VECTORCAST_DIR%\clicast -lada RGw Configure Set environment_name_
attribute true

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\Environments\ADACAST_.CFG
Key 'environment_name_attribute' set to 'true' for Subsystem Profile'csv'.
```

Step 8 - Export the Test Data

Now we export the test case data into the new file, **`Results.csv`**, by entering:

```
$ %VECTORCAST_DIR%\clicast -lada RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
Current Gateway was set to 'CSV'.

$ %VECTORCAST_DIR%\clicast -lada RGw Export CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
ada\ADACAST_.CFG
**************starting export*************
**************export finished*************
```

APPENDIX A: VECTORCAST/ADA MESSAGES

Start-up Messages

VECTORCAST_DIR Environment Variable is not set

This message indicates that an error was made in the installation of VectorCAST. Check all environment variables, symbols and path links that are required.

License Messages

FLEXlm: Licensed number of users already reached

This message indicates that an attempt was made to use more VectorCAST licenses than are authorized.

FLEXlm: Cannot find license file

This message indicates that license processing could not be completed. Check all environment variables, symbols and path links that are required.

FLEXlm: Feature has expired

This message indicates that the VectorCAST license has expired.

LICENSE ERROR: License Failure

This message indicates that some license anomaly was detected subsequent to the initial invocation of VectorCAST and will not allow VectorCAST to continue. All data will be saved and VectorCAST will be automatically terminated. In general, a FLEXlm error number is displayed. The meaning of the error code is available in the FLEXlm End User's Guide: **VectorCAST installation directory => DOCS => fnp_LicAdmin.pdf**.

Environment Building Informational Messages

Source not Found: *unit name*

This message indicates that the source for a dependent of the UUT could not be found. The environment construction will continue but you will not have the option of creating a stub for this unit. Also, you will not be able to modify any data values which rely on types from this dependent unit.

INFO: Environment built but Run-Time error exists

This message indicates that there is an exception being raised during the elaboration processing of the unit under test. This is most often caused by function calls that are made by the code under test during program elaboration.

Copying The Parent View For Testing

(Apex only) This message is displayed before VectorCAST calls the Apex copy_view command to create a new view in the current sub-system for test purposes.

Performing A Private Check-Out Of Source Files

(Apex only) This message is displayed performs a private check-out of the source files that exist in the test view.

Parsing UUT body for whitebox conversion
Processing whitebox data

These message are displayed during the environment construction when VectorCAST ios performing the parsing and conversion of the unit under test that is required for whitebox testing.

Processing UUT *unit name*

This message indicates that the parsing of the Unit Under Test is being accomplished by the VectorCAST environment builder.

Processing *unit name*

This message indicates that the indicated unit is being parsed and added to the current environment.

***unit_name* is a generic instantiation - whitebox processing cannot be performed**

This message will be displayed if the unit under test is a generic instantiation and you choose the whitebox test option.

***unit_name* is a private package - whitebox processing automatically invoked**

This message will be displayed if the unit under test is a private package and you DO NOT choose the whitebox test option.

Building Driver

This message indicates that the Ada language driver program is being built by the VectorCAST environment builder and generator.

Building Harness for *unit_name*

This message is displayed when VectorCAST is building the data handling functions for a particular unit.

Compiling *unit name*

This message indicates that elements of the environment have been created and they are now being compiled into the Test Ada Library by VectorCAST.

Linking Environment

This message indicates that all elements of the environment have been created and they are now being linked into the Test Ada Library by VectorCAST.

Environment Built Successfully

This message indicates that all elements of the VectorCAST environment have been successfully constructed.

Environment Building Error Messages

ERROR: Source not Found: *unit name* (S) [(B)]

This message indicates that the source file for a Unit Under Test spec or body could not be found. This error is fatal for the unit under test and informational for dependent units due to the fact that the UUT source is required to build a test harness.

Source not Found: *dependent unit name*

This message indicates that the source for a dependent of a UUT could not be found. The environment construction will continue but you will not have the option of creating a stub for this unit. Also, you will not be able to modify any data values which rely on types from this dependent unit.

ERROR: Syntax Error in Unit: *unit name*

This message indicates that there is an Ada construct not recognized by the VectorCAST parser. The next screen output by VectorCAST will be a scrollable buffer window containing parser diagnostic message.

ERROR: Internal Tool Error (Parser) - Write STR

This message indicates that there is some construct not being parsed correctly by VectorCAST. This error is fatal, no recovery is possible. After the error message is cleared, a scrollable buffer window will be presented with amplifying information about the parser error. This information will also be written to a file. The file will be stored in the default directory from which VectorCAST was invoked. The filename will be "VEIPARSE.ERR". Print this file and submit the listing with a description of the error to Customer Support support@vector.com.

ERROR: Stub Could not be Created for: *name*

This message indicates that when the operator chose to create a stub for a dependent unit the creation failed. This message results from an unexpected error while the VectorCAST code generator executed. This error is not fatal; environment creation will continue.

ERROR: Compile Failed, unit: *unit name*

This message is displayed to indicate that the compilation of a particular harness source file has failed. The compiler diagnostic message will be displayed to help resolve the issue.

ERROR: Compile Failed, stubbed unit: *unit name*

This message indicates that during the compilation of the VectorCAST Test Harness components, a compile error occurred. The compiled error listing will be displayed in the VectorCAST display and will also be written to the file ACOMPILE.LIS.

ERROR: Abnormal Termination of Compile and Link

This message is displayed when VectorCAST encounters an unexpected error during the compilation or link of the test harness. For example this message may reflect the inability to create a file because of disk space or permission problems.

ERROR: Environment Directory Creation Failed

This message indicates that some sort of operating system or protection error occurred when VectorCAST attempted to create the disk directory that will contain the environment files

ERROR: Cannot Overwrite Environment

This message indicates that you have attempted to create a VectorCAST environment in a situation where a directory already exists with the same name. The directory may or may not be a VectorCAST environment.

ERROR: Cannot Build Environment for: *unit name*

This message indicates that the UUT is a generic package or subprogram. The environment creation process will be abandoned. You cannot test a generic unit directly, you must use a stand-alone instantiation to test a generic. Instantiate this unit and choose the instantiation as the VectorCAST unit under test.

ERROR: Driver Program did not link

ERROR: Instrumented Harness did not link

ERROR: Data Program did not link

These messages indicate that an executable program created by VectorCAST did not link properly. The linker diagnostic listing will be displayed and will be written to the file AALINKER.LIS.

ERROR: Driver Could not be Invoked

ERROR: Data Interface Could not be Invoked

These messages indicates that VectorCAST cannot execute one of the executables that it builds. Probable Cause: a previous error during environment creation, an operating system protection error on program execution, or "." is not on the user's path.

ERROR: Environment Creation Failed

This message indicates that the VectorCAST environment builder was not able to complete its processing normally. Nothing is usable in the partially created environment, and it will be deleted.

ERROR: *environment name* Could not be Created

This message indicates that a serious error occurred during environment creation. File protection or limit conditions may exists.

ERROR: *environment name* Could not be Deleted

This message indicates that there are unexpected files in the environment subdirectory or for some other reason the files in the environment subdirectory could not be deleted.

ERROR: *environment name* Could not be Opened

This message indicates that the environment selected either does not exist or has been corrupted.

INFO: Environment Built but not Compiled

This message indicates that the environment creation is complete, however there was a compile failure during creation that must be resolved before continuing.

INFO: Environment built but not linked

This message indicates that the environment creation is complete, however, there was a link failure during creation that must be resolved before continuing.

INFO: Environment built but Run-Time error exists

This message indicates that there is an exception being raised during the elaboration processing of the unit under test. This is most often caused by function calls that are made by the code under test during program elaboration. The following Ada source fragment illustrates this point:

```
package unit_under_test is
object : integer := get_object_value;
```

Deleting Files From Environment Build

This message is displayed if the environment creation aborts for any reason. VectorCAST will automatically delete all files that were created during the aborted environment build.

Test Case Messages

ERROR: Test Execution Failed. Could not execute UUT interface.

This message indicated that the execution of the test harness did not produce the expected output files for VectorCAST to produce the test reports. Common causes are a fatal error in the initialization of the test harness, or problems with the configuration of the target if you are using VectorCAST/RSP. To determine the cause, run the test case using the VectorCAST option: “Execute with debug” and step through the code under test.

ERROR: No Change Made to Test Data

This message is displayed during the creation of a test case, this error message is displayed to notify the operator that the entered change was not affected.

ERROR: Illegal Numeric Entry

This message indicates that an operator entered parameter value entered during test case generation is not consistent with the numeric value required by the context.

ERROR: *type mark* Could not be Found

This message indicates that the displayed type mark is of a type that could not be found in the environment.

ERROR: Value Out of Range - no Change Made to Data

This message indicates that a particular test data item is outside the allowable range for that type, and that no further processing will be performed.

ERROR: Field is not in active part of variant

This indicates that the a field in a variant structure has been selected for modification, but that the field cannot be changed because the current value of the discriminant makes the field inactive.

INFO: Index Expression Error

This message indicates that an operator entered array index value is invalid for the array specified.

Can't Change Discriminant / Type has non-defaulted Discriminants

This indicates that the a field in a variant structure has been selected for modification, but that the field cannot be changed with in a VectorCAST test case because the type definition did not provide a default value for the discriminant.

INFO: Invalid Test Case Name

This message indicates that the selected test case name contains illegal characters or space characters.

INFO: No Expected Results Exist

This message indicates that you have selected an operation dependent on the expected results file. However, no expected results exist for this test case.

INFO: No Results Exist

This message indicates that you have selected an operation dependent on the actual results file. However, no expected results exist for this test case.

INFO: Null Record => No Components

This message is displayed when a structure type with no components is encountered.

INFO: Type Not Supported

This message is displayed when a type is encountered that is not supported by VectorCAST. See "Limitations and Restrictions" on page 479 for more information.

Building Test Case Data

This message indicates that the ASCII data is being created which will allow viewing and printing of test cases.

General Messages

Building Environment Script file

Building Test Case Script file

Building the CLI Script

These message are displayed when VectorCAST is building the three script files required to for regression testing.

Creating environment script

This message is displayed while VectorCAST is creating an environment script file

Script Building Completed

This message is displayed when VectorCAST is done building a test case script or an environment script.

Building Test Case Script Template

This message is displayed while VectorCAST is creating an test case script template

Linking Instrumented Harness

This message is displayed when you initialize coverage, or re-link the test environment and the instrumented test harness is being linked.

Getting Size and Range Data

This message is displayed when VectorCAST is building its database of range data for all of the various parameters, global objects and data types that are part of the test environment.

Processing Global Objects [Phase1 | Phase 2 | Phase 3]

Processing Parameters [Phase1 | Phase 2 | Phase 3]

These message report on the progress of computing the correct data ranges for global objects and parameters. This processing is the processing that enables VectorCAST to display the specific range of legal values for all data types used in the test environment.

Running 'MAKE' Command

This message is displayed when VectorCAST is performing “make” processing to make the test ada library up-to-date when there are obsolete units. This processing is only required in the case of whitebox testing.

Down Loading Target Processor

Running transfer to execute UUT on: *board_name*

Running tornserv to execute target program

(Target Only) These message are displayed while VectorCAST is connecting to the target and downloading the test harness executable.

INFO: Invalid Environment Name

This message indicates that the entered environment name contains a space or some other character that is an illegal directory name on the host platform.

***environment name* Deleted Successfully**

This message indicates that the environment chosen for deletion has been completed.

Could not open environment *environment_name*

This message is displayed when there is an error opening an environment. Possible reasons are disk space and file protection problems.

Determining Basis Paths

This message is displayed when VectorCAST is performing the basis path computation for a particular unit.

Error Computing Basis Paths

This message is displayed when VectorCAST cannot compute the basis path listings for a module. Errors of this type should be reported to VectorCAST Technical Support (support@vector.com)

Error Initializing Coverage

This message is displayed when VectorCAST cannot perform the coverage instrumentation processing for a module. Errors of this type should be reported to VectorCAST Technical Support (support@vector.com)

Rebuilding instrumented harness

This message is displayed when the user selects the “re-link” command and coverage is enabled. VectorCAST will automatically retrieve the latest version of the code under test, and re-instrument the code.

Error compiling instrumented file

This message is displayed when the compilation of an instrumented source module fails. Errors of this type should be reported to VectorCAST Technical Support (support@vector.com).

Building Min Mid Max Test Cases

This message is displayed while VectorCAST is generating the Min, Mid or Max test cases.

Environment has not been Compiled/Compile Environment Before Proceeding!

Environment has Unresolved Compile Errors/Resolve Errors Before Proceeding!

This message is displayed when an environment has been opened in VectorCAST and there are unresolved compile errors. This message indicates that no testing can be performed until the compile errors are resolved.

Environment has Unresolved Link Errors/Resolve Errors Before Proceeding!

This message is displayed when an environment has been opened in VectorCAST and there are unresolved link errors. This message indicates that no testing can be performed until the link errors are resolved.

Harness has Run-Time Errors / Resolve Errors Before Proceeding!

This message is displayed when an environment has been opened in VectorCAST and there are unresolved run-time errors. This message indicates that no testing can be performed until the run-time errors are resolved.

Updating Coverage Data

Updating Aggregate Coverage Report

This message is displayed after a test case execution when VectorCAST is updating the coverage data to reflect the test that was just run.

Environment Data not Expanded

Cannot Create Min Mid or Max Test Cases

This message indicates that there was some kind of unexpected error during the generation of the parameter and global data range information. As a result, testing may be performed, but the generation of Min, Mid, Max test cases is unavailable.

Reading Instrumentation Data Line: *line_number*

Sorting Instrumentation Data Line: *line_number*

These message are displayed to provide an update to the user when large coverage data files are being processed . The line numbers will update in 1000's as the file is processed.

ERROR: Could Not Build Test History

This message indicates a serious error occurred during the test execution and the expected output data did not get produced by the test harness. With Target testing this can indicate that the target communication failed. With host testing this can indicate a file lock, disk space, or permission problem.

CLICAST Messages

Environment must be specified on command line

Environment *environment_name* is invalid

These messages are displayed when you issue a clicast command that requires the name of an environment, and you do not provide an environment name or provide an invalid name as a command option.

Subprogram must be specified on command line

Subprogram *subprogram_name* is invalid

This message is displayed when you issue a clicast command that requires the name of a subprogram, and you do not provide a subprogram name or you provide an invalid name as a command option.

Test case must be specified on command line

Test Case *testcase_name* is invalid

This message is displayed when you issue a clicast command that requires the name of a test case, and you do not provide a test case name or you provide an invalid name as a command option.

No files specified

This message is displayed when a clicast command requires an output filename and no filename is provided.

Compiler *compiler_name* not supported

This message is displayed when an unknown compiler name is provided for a clicast command

CLICAST failed - parameter inconsistency

This message is displayed when you attempt to invoke clicast with a combination of parameters and options that VectorCAST cannot understand.

CLICAST failed with unknown exception

This message is displayed when the execution of a clicast command fails and there is no additional information available.

APPENDIX B: ENVIRONMENT SCRIPT LANGUAGE

The VectorCAST environment scripting enables you to build environments using text files. Scripting is provided as an alternative to entering commands interactively via the VectorCAST GUI.

The Environment Scripting language corresponds to the data required to construct an Environment using the GUI. Each command is entered on a single line. The script commands are not case sensitive, though the data may be.

Note the following conventions:

- All commands start with "ENVIRO." to indicate that an Environment command follows.
- All commands that require data must use a colon (:) to indicate that start of the associated data.
- Comments may be inserted, and are indicated by inserting the VectorCAST comment delimiter "--" as the first item on the line.

The following is a list of the valid script commands with a description of each command.

Enviro Command List

ENVIRO.NEW

Required. This command indicates that a new environment is to be created. ENVIRO.NEW must be the first non-comment line in the environment script.

ENVIRO.NAME:

Required. This command is used to provide a name for the Environment. Unlike test case scripts, the "name" command is mandatory. Since the Environment name will ultimately be the name of a sub-directory, it must be unique within its parent directory and not contain spaces.

ENVIRO.UUT:

This command indicates a Unit Under Test filename. If you have more than one UUT, list each unit name separately, one per line, each with its own ENVIRO.UUT command. If a unit is listed twice, in two different commands (i.e. ENVIRO.UUT and ENVIRO.STUB), then the first time the unit's name is encountered takes precedence over other appearances.

The environment script must include at least one ENVIRO.UUT line or ENVIRO.STUB_BY_ FUNCTION line.

ENVIRO.COMPILER:

This command tells VectorCAST which compiler to use when building the Environment. The supported compilers and corresponding script names are as follows:

Compiler Name	Script Name
ActivAda	ACTIVADA
AdaWorld	ADAWORLD
APEX	APEX
DDCI Windows	DACS
DDCI Unix	DDCI_UNIX
DEC Ada	DEC_ADA
GNAT	GNAT
Green Hills Ada83	GREEN_HILLS
Green Hills Ada95	GREEN_HILLS_95
LegacyAda	LEGACY_ADA
Max Ada	MAXADA
ObjectAda	OBJECTADA
PowerAda	POWERADA
RiscAda	RISCADA
VADS	VADS
Score	SCORE
SunAda	VADS

ENVIRO.CUSTOM_COVERAGE: *<unit>:<ON | OFF>*

Unit-specific coverage enable flag. Supports the specification of coverage on non-stubbed dependent units in test environments. Custom coverage on these units can be applied during environment build and retained during environment rebuild.

ENVIRO.PARENT_LIB:

Required. This command is used to tell VectorCAST the name of the parent library where the UUT exists. If the parent library is not a subdirectory of the current default directory, then the full path must be given. On platforms where the operating system is case sensitive (i.e. Linux) the exact case matched path must be entered.

The environment variable VCAST_PARENT_LIBRARY can be used to override the value set in the environment script, which can be useful in running regression tests with different libraries for comparison.

ENVIRO.WHITE_BOX:

This command is optional and if not given, defaults to blackbox. If the Whitebox option is desired then the command "YES" is used.

ENVIRO.COVERAGE_TYPE: *type*

This command is optional and if not given, defaults to no coverage (None). To specify a coverage type, *type* can be None, STATEMENT+MC/DC, STATEMENT+BRANCH, MC/DC, BASIS_PATHS, BRANCH, STATEMENT .

ENVIRO.STUB:
ENVIRO.STUB:ALL
ENVIRO.STUB:NONE
ENVIRO.STUB: *unit name to stub*

This command is used to tell VectorCAST which dependent units it should stub *by implementation* during the environment creation. The syntax for this command is one value per line. If all of the dependent units are to be stubbed, then use ENVIRO.STUB:ALL. If none of the dependent units are to be stubbed, then use ENVIRO.STUB:NONE.

To specify individual units to stub, each unit is named, one per line. By default, units that are not specified by name are not stubbed.

See also the environment variable VCAST_NEVER_STUB_LIST.

ENVIRO.IGNORE:
ENVIRO.IGNORE:ALL
ENVIRO.IGNORE: *unit name to ignore*

This command is used to tell VectorCAST to ignore all dependent units or to ignore particular ones when building the environment. Ignoring a unit causes VectorCAST to not parse it or any of its dependents, thus shortening the time needed to build an environment. If all of the dependent units are to be ignored, then use ENVIRO.IGNORE:ALL. To specify individual units to ignore, each unit is named, one per line.



Note: See “VectorCAST Closure Concepts” for additional information.

ENVIRO.IMPORT: *path to subsystem*

Import the specified subsystem into the environment’s work view. For Apex users.

ENVIRO.MAX_VARY_RANGE: *num*

This command is used to tell VectorCAST how many scalars can be varied at one time. The default is 20. The setting is indicated in the Tools => Options dialog, Builder tab.

ENVIRO.USER_GLOBALS:
ENVIRO.END_USER_GLOBALS:

The ENVIRO.USER_GLOBALS and ENVIRO.END_USER_GLOBALS delimit the lines of the script file to be put into the User Globals File. This command is optional (VectorCAST 4.2+) and if not given, defaults the following standard lines.

This file can be permanently altered by editing the GLOBALS.ADA file in the DATA directory in the VectorCAST installation directory. The User Globals are also accessible by choosing **Environment => User Code => Edit**, and then clicking the **User Globals** node.

```
package USER_GLOBALS_vCAST is
     type float_type is digits 6;
     subtype string_type is string(1..20);
     INT1 : integer;
     INT2 : integer;
     INT3 : integer;
     INT4 : integer;
     INT5 : integer;

     FLT1 : float_type;
     FLT2 : float_type;
     FLT3 : float_type;
     FLT4 : float_type;
     FLT5 : float_type;

     STR1 : string_type;
     STR2 : string_type;
     STR3 : string_type;
     STR4 : string_type;
     STR5 : string_type;

     type BUFFER_TYPE is array (1..200) of integer;
     BUFFER : BUFFER_TYPE;

end USER_GLOBALS_vCAST;
```

ENVIRO.USER_PARAMETERS:
ENVIRO.END_USER_PARAMETERS:

The code you add between this pair of commands creates a user-defined parameter to use in place of a parameter, instead of a VectorCAST-generated parameter. The syntax consists of a pair on each line: the case-sensitive VectorCAST-generated parameter, followed by your parameter object.

For example:

```
ENVIRO.USER_PARAMETERS:
<<MANAGER.PLACE_ORDER.TABLE>> USER_GLOBALS_vCAST.MY_TABLE
ENVIRO.END_USER_PARAMETERS:
```

defines the object MY_TABLE to be used in place of MANAGER.PLACE_ORDER.TABLE.

User Parameters can be created by selecting **Environment => User Code => Edit** from the Menu Bar and expanding the User Params node. Double-click to open the edit box. Alternatively, User Parameters can be created in the Create New/Update Environment wizard by selecting Step 6 User Code and opening the User Params node.

ENVIRO.USER_CODE_DEPENDENCIES:
ENVIRO.END_USER_CODE_DEPENDENCIES:

These commands delimit the dependencies (**`#include`** statements) for environment user code. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Header** section.

```
ENVIRO.USER_CODE_DEPENDENCIES:
          Header section
ENVIRO.END_USER_CODE_DEPENDENCIES:
```

ENVIRO.USER_CODE_OBJECTS:
ENVIRO.END_USER_CODE_OBJECTS:

These commands delimit the object definitions (type definitions, globals object, and subprograms) for environment user code. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Data** section.

```
ENVIRO.USER_CODE_OBJECTS:
          Data section
ENVIRO.END_USER_CODE_OBJECTS:
```

ENVIRO.USER_CODE_ONE_SHOT_INIT:
ENVIRO.END_USER_CODE_ONE_SHOT_INIT:

These commands delimit the harness initialization environment user code. This user code is

processed upon test harness initialization. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Harness Init** section.

```
ENVIRO.USER_CODE_ONE_SHOT_INIT:
          Harness initialization section
ENVIRO.END_USER_CODE_ONE_SHOT_INIT:
```

ENVIRO.USER_CODE_INITIALIZE: ENVIRO.END_USER_CODE_INITIALIZE:

These commands delimit the test case initialization environment user code. This user code is processed immediately before the harness calls the UUT. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Test Case Init** section.

```
ENVIRO.USER_CODE_INITIALIZE:
          Test Case initialization section
ENVIRO.END_USER_CODE_INITIALIZE:
```

ENVIRO.USER_CODE_CAPTURE: ENVIRO.END_USER_CODE_CAPTURE:

These commands delimit the test case termination environment user code. This user code is processed immediately before the harness itself terminates. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Test Case Term** section.

```
ENVIRO.USER_CODE_CAPTURE:
          Test case termination section
ENVIRO.END_USER_CODE_CAPTURE:
```

ENVIRO.USER_CODE_ONE_SHOT_TERM: ENVIRO.END_USER_CODE_ONE_SHOT_TERM:

These commands delimit the test harness termination environment user code. This user code is processed upon harness termination. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Harness Term** section.

```
ENVIRO.USER_CODE_ONE_SHOT_TERM:
          Harness termination section
ENVIRO.END_USER_CODE_ONE_SHOT_TERM:
```

ENVIRO.USER_CODE_STUB_PROCESSING: ENVIRO.END_USER_CODE_STUB_PROCESSING:

These commands delimit the test stub processing environment user code. This user code is common to all stubs, and is processed between Parameter Expected User Code and Parameter Input User Code. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Stub Processing** section.

```
ENVIRO.USER_CODE_STUB_PROCESSING:
          Stub Processing section
ENVIRO.END_USER_CODE_STUB_PROCESSING:
```

ENVIRO.STUB_ENTRY_USER_CODE: ENVIRO.END_STUB_ENTRY_USER_CODE:

These commands delimit the stub subprogram entry configuration user code. It is processed for every stub upon entry, right after any Configure Stubs Beginning User Code and before any Expected Parameter User Code of the stub. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Stub Entry** section.

```
ENVIRO.STUB_ENTRY_USER_CODE:
          Stub subprogram entry configuration code
ENVIRO.END_STUB_ENTRY_USER_CODE:
```

ENVIRO.STUB_EXIT_USER_CODE: ENVIRO.END_STUB_EXIT_USER_CODE:

These commands delimit the stub subprogram exit configuration user code. It is processed for every stub upon exit, right before any Configure Stubs Ending User Code and after any Input Parameter User Code of the stub. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Stub Exit** section.

```
ENVIRO.STUB_EXIT_USER_CODE:
          Stub subprogram exit configuration code
ENVIRO.END_STUB_EXIT_USER_CODE:
```

ENVIRO.STUB_USER_CODE_FILE: ENVIRO.END_STUB_ USER_CODE_FILE:

These commands delimit the beginning of stub and end of stub user code for Configure Stubs. There are two sub-commands: one for the beginning of the stub

Repeat for each unit or subprogram that has stub user code to process at the beginning of the stub

and one for the end of stub. You can repeat these any number of times for different units or different

subprograms within the same unit.

You can add or modify this user code by choosing **Environment => Configure Stubs => Edit**.

```
ENVIRO.STUB_USER_CODE_FILE:

 BEGINNING_OF_STUB.unit.subprogram
Configure Stubs dialog, Beginning of Stub
END_BEGINNING_OF_STUB.unit.subprogram
```



Note: Repeat these sections for each unit or subprogram that has stub user code to process at the beginning of the stub.

```
END_OF_STUB.unit.subprogram
Configure Stubs dialog, End of Stub
END_END_OF_STUB.unit.subprogram
```



Note: Repeat these sections for other units and/or subprograms at the end of stub.

```
ENVIRO.END_STUB_USER_CODE_FILE:
```

ENVIRO.STUB_DEPEND_USER_CODE_FILE: ENVIRO.END_STUB_DEPEND_USER_CODE_FILE:

These commands delimit the stub dependencies user code for Configure Stubs. You can add or modify this user code by choosing **Environment => Configure Stubs => Edit**, and opening the **Stub Dependencies** cell.

For each stubbed unit for which you want to write stub dependency user code, you enclose that unit's name with `BEGIN_UC` and `END_UC`.



Note: Repeat the following for each unit that has a dependency

```
ENVIRO.STUB_DEPEND_USER_CODE_FILE:
BEGIN_UC:
unit
          Configure Stubs dialog, Stub Dependencies column
END_UC:
ENVIRO.END_STUB_DEPEND_USER_CODE_FILE:
```

Example: Suppose a subprogram vcast_out is defined in a package named IO. In the following example, that subprogram is called at the beginning of stub and end of stub for the unit GUI_STUB and subprogram STUB_CONFIG_STUBS, as shown in the first portion of the example below. The second portion shows the package IO "withed" in the Stub Dependency section.

```
ENVIRO.STUB_USER_CODE_FILE:
BEGINNING_OF_STUB.GUI_STUB.STUB_CONFIG_STUBS
IO.vcast_out("BEGIN CONFIG STUBS");
END_BEGINNING_OF_STUB.GUI_STUB.STUB_CONFIG_STUBS
END_OF_STUB.GUI_STUB.STUB_CONFIG_STUBS
IO.vcast_out("END CONFIG STUBS");
END_END_OF_STUB.GUI_STUB.STUB_CONFIG_STUBS
ENVIRO.END_STUB_USER_CODE_FILE:
ENVIRO.STUB_DEPEND_USER_CODE_FILE:
BEGIN_U\\\\DATASERVER
GUI_STUB
with IO;
END_U\\\\DATASERVER
ENVIRO.END_STUB_DEPEND_USER_CODE_FILE:
```

ENVIRO.ADDITIONAL_UNIT_BODIES
ENVIRO.END_ADDITIONAL_UNIT_BODIES

These two commands delimit the implementation portion of additional units that you want added to the test harness, as though it were another source file (unit). For example, you may have a print utility, a timing routine, or a data setup routine that is not part of your deliverable, but necessary for unit testing. The code here is not parsed by VectorCAST during environment building. These tags are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Additional Unit Bodies** section.

```
ENVIRO.ADDITIONAL_UNIT_BODIES:
          Implementation for additional units
ENVIRO.END_ADDITIONAL_UNIT_BODIES:
```

ENVIRO.ADDITIONAL_UNIT_SPECS
ENVIRO.END_ADDITIONAL_UNIT_SPECS

These two commands delimit the specification portion of an additional code unit, for example, a class header. These tags are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Additional Unit Specs** section.

```
ENVIRO.ADDITIONAL_UNIT_SPECS:
          Specifications for additional units
ENVIRO.END_ADDITIONAL_UNIT_SPECS:
```

ENVIRO.USER_CODE_TIMER_START:
ENVIRO.END_USER_CODE_TIMER_START:

Code to start timer after the test harness performs data setup, but just prior to calling the UUT. These tags are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code

node, or by choosing **Environment => User Code => Edit**, and opening the **UUT Timer Start** section.

```
ENVIRO.USER_CODE_TIMER_START:
          Code to start timer
ENVIRO.END_USER_CODE_TIMER_START:
```

ENVIRO.USER_CODE_TIMER_STOP: ENVIRO.END_USER_CODE_TIMER_STOP:

Code to stop timer immediately after returning from the call to the UUT. These tags are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **UUT Timer Stop** section.

```
ENVIRO.USER_CODE_TIMER_STOP:
          Code to stop timer
ENVIRO.END_USER_CODE_TIMER_STOP:
```

ENVIRO.INDUSTRY_MODE

Code to set the Industry Mode. Industry Modes are set as follows for each Industry Mode type:

```
ENVIRO.INDUSTRY_MODE:DO-178 B/C (Avionics)
ENVIRO.INDUSTRY_MODE:ISO-26262 (Automotive)
ENVIRO.INDUSTRY_MODE:IEC-61508 (Industrial)
ENVIRO.INDUSTRY_MODE:EN-50128 (Railway)
ENVIRO.INDUSTRY_MODE:IEC-62304 (Medical)
ENVIRO.INDUSTRY_MODE:Default
```

ENVIRO.END

Required. This command marks the end of the script data for a particular environment. It must be the last line in the environment script. When this command is encountered, VectorCAST attempts to build an environment using the values read since ENVIRO.NEW was read.

Sample Environment Script

```
-- VectorCAST Environment Script
-- Copyright 2017 Vector Informatik, GmbH
--
ENVIRO.NEW
ENVIRO.NAME:DEMO
ENVIRO.COMPILER:GREEN_HILLS_95
ENVIRO.PARENT_LIB:\\\\DATASERVER\VCAST\Test
ENVIRO.UUT:MANAGER
ENVIRO.WHITE_BOX:NO
```

```
ENVIRO.STUB:DATABASE
ENVIRO.END
```

APPENDIX C: TEST SCRIPT LANGUAGE

TEST.SCRIPT Command List

The scripting language corresponds to the data required to build a test case using the GUI. Each command is entered on a single line. The Script reader is case sensitive, conforming to the normal case sensitivity rules of the language.

Note the following conventions:

- All commands start with "TEST." to indicate that a test command follows.
- All commands that require data must use a colon (:) to indicate the start of the associated data.
- Comments may be inserted, and are indicated by inserting the comment delimiter "--" as the first item on the line.

Script Commands

The following is a list of the valid script commands with a description of each command.

TEST.NEW | TEST.ADD | TEST.REPLACE | TEST.REMOVE:

Required. Tell VectorCAST that the test case definition that follows defines a new test case, adds values to an existing test case, replaces an existing test case, or removes an existing test case. You can only use one (NEW, ADD, REPLACE or REMOVE) for each test case definition.

TEST.NEW is used to tell VectorCAST that the following commands are to start the construction of a new test case.

TEST.ADD is used to add additional input or expected data values to an existing test case.

TEST.REPLACE is used to delete an existing test case with the same name and replace it, using the data that follows in the test script. The script log informs you that the test case is being replaced.

TEST.REMOVE is used to remove an existing test case.

TEST.ATTRIBUTES: <data name>:<attribute expression>

Attributes for scalar data. These are specific to attributes contained in the test case that are unrelated to input or expected values. For example, you may right-click on the input or expected value field of a particular parameter and set the base for the value to be hex or octal. This attribute gets stored in the test script so that the same attribute is used when it is re-imported into the same or a different environment.

TEST.AUTOMATIC_FINALIZATION

Specify a <<COMPOUND>> or <<INIT>> test case as an Automatic Finalization test case. It sets a flag indicating a <<COMPOUND>> or <<INIT>> test case should be run automatically after other test cases. A single <<COMPOUND>> or <<INIT>> test case can be both Automatic Initialization and Automatic Finalization. An Automatic Finalization test case can be executed on its own, but it cannot

be run before or after itself. There can only be one Automatic Finalization test per test environment. When exporting a test script, the Automatic Finalization test cases are also exported.

TEST.AUTOMATIC_INITIALIZATION

Specify a <<COMPOUND>> or <<INIT>>test case as an Automatic Initialization test case. It sets a flag indicating a <<COMPOUND>> or <<INIT>> test case should be run automatically before other test cases. A single <<COMPOUND>> or <<INIT>> test case can be both Automatic Initialization and Automatic Finalization. An Automatic Initialization test case can be executed on its own, but it cannot be run before or after itself. There can only be one Automatic Initialization test per test environment. When exporting a test script, the Automatic Initialization test cases are also exported.

TEST.BASIS_PATH: *x* of *y*

When basis path test cases are generated automatically (using **Tools => Basis Path => Generate Tests**), and a test script is exported for these test cases, the TEST.BASIS_PATH command is used to identify them as basis path test cases. The text "*x* of *y*" indicates that particular test case out of the total number of basis path test cases generated for that subprogram.

TEST.COMPOUND_ONLY

This command is used to tell VectorCAST that the test case is to be executed only from within a compound test case. During Batch Execute All, the test case is not executed unless it is part of a compound.

TEST.CSV_NAME_COL:

This command is used in a CSV MAP test case. It specifies the number of the column containing the test case name. Enter a value of "0" if you do not specify the test case name in the .csv file.

TEST.CSV_NOTES_COL:

This command is used in a CSV MAP test case. It specifies the number of the column containing test case notes. Enter a value of "0" if you do not specify Notes in the .csv file.

TEST.END

This command is used to indicate the end of a test case and causes the previously entered test case data to be stored into the VectorCAST environment.

TEST.EXPECTED: *identifier* : *value*

This command is used to provide expected values for parameters and global objects.

See the section "Setting Input and Expected Values" on page 471 for a full description of the <value> syntax.

TEST.EXPECTED_GLOBALS_USER_CODE: *global_value*
<enter user code here>
TEST.END_EXPECTED_GLOBALS_USER_CODE:

Together, these commands delimit the expected user code for global values. This user code is evaluated when other global values are evaluated, so it is possible for this user code to be evaluated when a test ends due to the event limit being reached.

```
TEST.EXPECTED_GLOBALS_USER_CODE:USER_GLOBALS_VCAST.<<GLOBAL>>.VECTORCAST_
INT2
{{ <<USER_GLOBALS_VCAST.<<GLOBAL>>.VECTORCAST_INT2>> == ( 2 ) }}
TEST.END_EXPECTED_GLOBALS_USER_CODE:
```

TEST.EXPECTED_USER_CODE: <<testcase>>
<enter user code here>
TEST.END_EXPECTED_USER_CODE:

Together, these commands delimit the test case's expected user code when "<<testcase>>" is used (without the quotes), and is not associated with any single parameter.

TEST.EXPECTED_USER_CODE: *unit.subprogram.parameter*
<enter user code here>
TEST_END_EXPECTED_USER_CODE:

Together, these commands delimit a parameter's expected user code in the UUT. Specify the parameter to which the user code applies using the notation *unit*.*subprogram*.*parameter*.

TEST.FLOATING_POINT_TOLERANCE: *number*%
(or)
TEST.FLOATING_POINT_TOLERANCE: *decimal number*

Using one of these commands enables you to set a floating-point tolerance at the test case level. You can set the floating-point tolerance at the environment level, the test case level, and the parameter level. Each level overrides the previous one. Therefore, you can set an environment-level tolerance (**Tools => Options dialog, Execute tab**), specify a test case tolerance for a particular test case, and within that test case, specify a tolerance for one parameter.

In the test script, you can specify the floating-point tolerance as a percent, or as the decimal equivalent. For example, to set the test case floating-point tolerance to 1.25%, use either of the following notations:

```
TEST.FLOATING_POINT_TOLERANCE: 1.25%
TEST.FLOATING_POINT_TOLERANCE: 0.0125
```

Upon test script import, the text "Test case floating point tolerance is: *number*%" is displayed in the log.

Note: For very small tolerances, the floating point representation of the value may not be exact due to the resolution of floating point numbers; in this case, the floating point tolerance may show more digits than the user expects. For example, a tolerance of 0.000000001 may actually be displayed as 0.00000000099999.

TEST.FLOW:
<unit> . <subprogram>
TEST.END_FLOW:

This is the expected sequence of subprograms called in the Unit Under Test and stubbed units. For each <unit>.<subprogram> that is called, there is a line in this section. For example, if the UUT is unit A, the test harness calls unit A's subprogram A() first, then unit B's subprogram B(), then unit C's subprogram C(), then the UUT returns control to the test harness:

```
TEST.FLOW
A.A
B.B
C.C
A.A
TEST.END_FLOW
```

TEST.IMPORT_FAILED

VectorCAST adds this line to an exported test script file if the option "Strict test case importing" is on, and there was an error when the script file was originally imported.

See also "Strict Test Case Importing" on page 289 for more information.

TEST.NAME: *name*

This command is used to provide a name for the test case. The *name* is case-sensitive, if you are using it with TEST.ADD. This command is not required. If a TEST.NAME command is not provided, VectorCAST will generate an automatic name as it does when creating test cases in the Test Case Editor. If this command is used, the *name* provided must be unique. Otherwise, it will be ignored and an automatic name will be assigned when the test script is processed.

TEST.NOTES:
<enter text here>
TEST.END_NOTES:

Together, these commands delimit a list of requirements or notes for the test case. Everything that exists between these delimiters will be included in the requirements section of the test case. Note that each of these commands must exist, by itself, on separate lines. Anything that follows these commands on the same line will be ignored. If you have the Report option "Verbose management report" turned on, then the first 89 characters in the Notes section are displayed in the Test Case Management Report.

TEST.REQUIREMENT_KEY: *requirement_key*

When an environment has imported requirements through the Requirements Gateway, a requirement can be associated with a testcase. Upon test script export, this command is used to identify the requirement key associated with each test case. The requirement key is imported, and should not be changed in the test script. If a test case has more than one requirement, then multiple commands are used, with one key per line. (VectorCAST with Requirements Gateway license)

TEST.SCRIPT_FEATURE: *special feature*

The TEST.SCRIPT_FEATURE line is automatically added by VectorCAST on test script export and should not need to be modified by the user. These commands tell VectorCAST if any special features are present in the environment which the test script importer should support.

TEST.SCRIPT_FEATURE:ADA_DIRECT_ARRAY_INDEXING which causes arrays with direct indexing such as -10 to 10 to be output in the script as:

```
TEST.VALUE:ADA_TESTCASE_VERSION2.FUNC_VECTOR_IMPLICIT_INTEGER.P(-10..10):3
```

However, if you turn on the option 'Disable direct array indexing' on the **Tools => Options dialog**, **Builder** tab, then the ADA_DIRECT_ARRAY_INDEXING script feature is not written to the script and the TEST.VALUE line is output in the script as:

```
TEST.VALUE:ADA_TESTCASE_VERSION2.FUNC_VECTOR_IMPLICIT_INTEGER.P(1..21):3
```

TEST.SLOT: *slot num, unit, subprogram name, number of iterations, testcase name, delay, print*

The command TEST.SLOT provides information on a slot in a compound test. It should be present when the TEST.SUBPROGRAM line specifies <<COMPOUND>>.

- *slot* can be any number. The test cases in the compound are added in the order they are listed in the TEST.SLOT command. *slot* is a quoted string. Example: “1”
- *unit* is a string identifying the unit under test. The name is case sensitive. *unit* is a quoted string. Example: “file_io”. If the slot is a nested compound slot, *unit* is specified as <<COMPOUND>> in quotes.
- *subprogram name* is a string identifying the subprogram in which this test case appears. *subprogram name* is a quoted string. Example: “CreateFile”. If the slot is a nested compound slot, *subprogram name* is specified as <<COMPOUND>> in quotes.
- *testcase name* is the name of the test case in the slot. *testcase name* is a quoted string. Example: “CREATEFILE.001”
- *number of iterations* is an integer specifying how many times the test case should repeat. To “turn off” a test case while keeping it in the compound, set the iterations to 0. Doing so causes the test case to be skipped during execution of the compound, while remaining in the compound test case. Note that setting iterations to 0 causes the test case to be skipped and not executed, and is not the same as setting PRINT=FALSE, which still executes the test case but does not gather the execution data. *number of iterations* is a quoted string. Example: “1”

- *delay*, in seconds indicates how many seconds (to a precision of tenths of a second) to delay after executing the slot, before executing the next slot. *delay* is a quoted string. Example: "1.2" If not included, the delay is assumed to be 0.
- *print* is either a TRUE or FALSE value and is used to prevent the reporting of data for a slot. The default value is TRUE and if no value is specified, the value TRUE is assumed. The value FALSE prevents the harness from reporting on executing data. The test is executed, but the test harness does not gather any data for the execution report. Note that PRINT=FALSE is not the same as setting *iterations* to 0 for the slot, where the test case will be skipped and not executed. If the value is FALSE, the test case in the slot is required to have no expected values. Example: "PRINT=FALSE".

For example, the following test script specifies a compound test case which has two slots with the test case PLACEORDER.001 and .002 included in it. Note that the script for the test cases PLACEORDER.001 and PLACEORDER.002 must be defined above the definition for the compound or already exist in the environment at the time that the compound test script is encountered.



Note: Compound test case below doesn't get a TEST.UNIT line itself.

```
-- COMPOUND TESTS
TEST.SUBPROGRAM:<<COMPOUND>>
TEST.NEW
TEST.NAME:<<COMPOUND>>.001
TEST.NOTES:
TEST.END_NOTES:
TEST.SLOT: "1", "MANAGER", "PLACE_ORDER", "1", "PLACEORDER.001"
TEST.SLOT: "2", "MANAGER", "PLACE_ORDER", "1", "PLACEORDER.002"
TEST.END
```



Note: Unit manager Requires TEST.SCRIPT_FEATURE:MULTIPLE_UNIT_SUPPORT.

TEST.STUB_EXP_USER_CODE: *unit.subprogram.parameter*
<enter user code here>
TEST.END_STUB_EXP_USER_CODE:

Together, these commands delimit a parameter's expected user code in a stub.

TEST.STUB_VAL_USER_CODE: *unit.subprogram.parameter*
<enter user code here>
TEST.END_STUB_VAL_USER_CODE:

Together, these commands delimit a parameter's input user code in a stub.

TEST.SUBPROGRAM: *subprogram name*

Required. This command is used to indicate a subprogram for which test cases will be built. The

subprogram name corresponds to any subprogram of the Unit Under Test specified in the TEST.UNIT line. If several test cases are to be built for the same subprogram this command does not have to be repeated until a different subprogram is desired. To create an <<INIT>> test case, use the subprogram name <<INIT>>.

TEST.UNIT: *unit name*

Required. This command identifies the unit that the test case is for. The name is not case sensitive. This command can be used once to indicate that all of the following test case definitions are for a single unit, until another TEST.UNIT line appears. This line is not required for <<INIT>> or <<COMPOUND>> test cases.

TEST.VALUE: *identifier* : *value*

This command is the most important of the test case scripting commands. It is used to provide the individual input values for parameters and global objects.

See the section "Setting Input and Expected Values" on page 471 for a full description of the <value> syntax.

TEST.VALUE_USER_CODE: <<testcase>>
<enter user code here>
TEST.END_VALUE_USER_CODE:

Together, these commands delimit the test case's input user code, when "<<testcase>>" is used (without the quotes), and is not associated with any single parameter.

TEST.VALUE_USER_CODE: *unit.subprogram.parameter*
<enter user code here>
TEST.END_VALUE_USER_CODE:

Together, these commands delimit a parameter's input user code in the UUT. Specify the parameter to which the user code applies using the notation *unit*.*subprogram*.*parameter*.

Setting Input and Expected Values

Most of the commands in a test script will be TEST.VALUE or TEST.EXPECTED commands. This section explains the syntax of these commands, and provides examples intended to cover as many combinations as practical.

All test value commands have three fields. The first field is the test command and is always set to "TEST.VALUE:" or "TEST.EXPECTED". The second field is called the *identifier* field and the third field is called the *value* field. Each of the three fields is separated by colons.

The *identifier* field consists of the unit name, a dot, the subprogram name, a dot, and the parameter name. For example: manager.Place_Order.Seat.



Note: VectorCAST is not sensitive to spaces around the Value field. The colon marks the start of the value and the end-of-line marker marks the end of the value.

Numeric Types

With numeric types, the value field is simply the number to be assigned. For example, the test value commands to set values for the two numeric parameters of subprogram Place_Order would be as follows:

```
TEST.VALUE:manager.Place_Order.Table: 3
TEST.VALUE:manager.Place_Order.Seat: 1
```



Note: When working with numeric parameters you may use based notation or exponential notation as you would when building test cases in the GUI.

Structure Types

For structure types, the Identifier field is extended using the standard C syntax of referring to the fields by their names with a '.' separating the field name from the rest of the identifier.

```
TEST.VALUE:manager.Place_Order.Order.Salad: CAESAR
TEST.VALUE:manager.Place_Order.Order.Entree: STEAK
TEST.VALUE:manager.Place_Order.Order.Dessert: CAKE
```

Enumeration Types

With enumeration types, the value field will be the enumeral that you are selecting. In the previous example you may have noticed that the fields were all enumeration types. In this case the value field of the commands are the enumerals.

Note that full named notation is neither required nor permitted when referring to the enumerals. VectorCAST will determine the validity of the value from the context of the definition.



Note: Out of Range enumeral values are specified with test case scripting by entering a number in place of the enumeral.

Boolean Types

Boolean types are similar to enumeration types, with the Value field limited to TRUE or FALSE.

Character and String Types

When setting the value field for a character or string, the standard Ada character and string delimiters (' and ") with the following exceptions:

- You must use the character delimiter if the character to be entered is itself the delimiter character (apostrophe) or a blank character.
- You must use the string delimiter if the string contains blank characters or contains string delimiters.

The following examples illustrate some of the variations:

```
TEST.VALUE:database.Get_Table_Record.Data.Designator: d
TEST.VALUE:database.Get_Table_Record.Data.Designator: ' '
TEST.VALUE:database.Get_Table_Record.Data.Designator: '''
TEST.VALUE:database.Get_Table_Record.Data.Wait_Person: Tim
TEST.VALUE:database.Get_Table_Record.Data.Wait_Person: ""Me""
TEST.VALUE:database.Get_Table_Record.Data.Wait_Person: "J Smith"
```



Note: As you would expect, the case of any character or string value is maintained.

Access Types

When working with access types, the keyword "ALL" must be used as is consistent with standard Ada syntax. For example, assume the following package specification is a unit under test or a stub.

```
package TEST is
  type POINTER is access integer;
  procedure TEST_IT (VALUE : POINTER);
end TEST;
```

The associated test value command would be as follows:

```
TEST.VALUE:TEST.TEST_IT.VALUE.ALL: 4
```

Array Types

When working with Arrays, the same conventions must be used as are used when building array values interactively. That is, the array index expression must be the Ada 'POS attribute of the array item that you wish to select. The array index will always be enclosed in parenthesis. The following example illustrates this:

```
TEST.VALUE:DATABASE.GET_TABLE_RECORD.DATA.ORDER(2).SALAD: GREEN
```

Unconstrained Array Parameters

When setting the value of an unconstrained array parameter, proceed exactly as you would for a normal array parameter. To set the values of the constraints for the array, use the keywords "'FIRST" or "'LAST" to set the upper or lower constraint. For example given the following package specification:

```
package TEST is
  type INDEX is new integer range 1..100;
  type BUFFER is array (INDEX range <>) of integer;
  procedure TEST_IT (VALUE : BUFFER);
end TEST;
```

The associated test value commands might be as follows:

```
TEST.VALUE:TEST.TEST_IT.VALUE(1): 1
TEST.VALUE:TEST.TEST_IT.VALUE(2): 2
TEST.VALUE:TEST.TEST_IT.VALUE(3): 3
TEST.VALUE:TEST.TEST_IT.VALUE'first: 1
TEST.VALUE:TEST.TEST_IT.VALUE'last: 3
```

Record Types

For record types the Identifier field is extended using the standard Ada syntax of referring to the fields by their names:

```
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SALAD: CAESAR
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.ENTREE: STEAK
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.DESSERT: CAKE
```

Subprogram Return Parameters

Up to this point we have been dealing with examples where we are setting the values of subprogram parameters. In the case of function return parameters, all of the same syntax applies, you simply use the keyword "RETURN" in place of the parameter name as in the following:

```
TEST.VALUE:MANAGER.GET_CHECK_TOTAL.RETURN: 23.0
```

Global Objects

To set the value of global objects rather than subprogram parameters, the basic syntax is UNIT.<<GLOBAL>>.OBJECT. All of the syntax related to the specific type of the object is the same as it is for subprogram parameters. The following example uses the same types from the Basic Tutorial, but we assume that there is a global object in unit MANAGER called ORDER_OBJECT.

```
TEST.VALUE:manager.<<GLOBAL>>.ORDER_OBJECT.ENTREE: STEAK
```

Test Case Options

These options can be set in the Options tab for a test case. By default, these options are not specified (set to false).

See the section "The Testcase Options Tab" on page 312for information on what each option does. To set them to true or a value in the scripting language, the syntax is:

```
TEST.VALUE:<<OPTIONS>>.COMPARE_BEFORE_UUT:TRUE | FALSE
TEST.VALUE:<<OPTIONS>>.DATA_PARTITIONS:<value>
TEST.VALUE:<<OPTIONS>>.DISPLAY_FULL_STRING_DATA:TRUE | FALSE
TEST.VALUE:<<OPTIONS>>.DO_COMBINATION:TRUE | FALSE
TEST.VALUE:<<OPTIONS>>.EVENT_LIMIT:<value>
TEST.VALUE:<<OPTIONS>>.FLOAT_POINT_DIGITS_OF_PRECISION:<value>
TEST.VALUE:<<OPTIONS>>.FLOAT_POINT_TOLERANCE:<value>
TEST.VALUE:<<OPTIONS>>.GLOBAL_DATA_DISPLAY:EACH_EVENT | RANGE_ITERATION | SLOT_
ITERATION | TESTCASE
```

```
TEST.VALUE:<<OPTIONS>>.HEX_NOTATION_FOR_UNPRINTABLE_CHARS:TRUE | FALSE
TEST.VALUE:<<OPTIONS>>.MULTI_RETURN_SPANS_COMPOUND_ITERATIONS: TRUE | FALSE
TEST.VALUE:<<OPTIONS>>.MULTI_RETURN_SPANS_RANGE: TRUE | FALSE
TEST.VALUE:<<OPTIONS>>.MULTI_RETURN_SPANS_TESTCASES: TRUE | FALSE
TEST.VALUE:<<OPTIONS>>.REFERENCED_GLOBALS: TRUE | FALSE
```

Range of Values for Input Data

During normal test case generation, individual scalar values are specified as input to the UUT or as output from stubs of dependent units. VectorCAST also provides a mechanism that enables you to define a single test case, which will test all values within a specified range of data. This range testing is useful in cases where it is desirable to exhaustively exercise a unit to verify that there is never an abnormal termination or that the routine always returns a legal value regardless of the input data.

To accommodate this type of testing, VectorCAST enables range expressions to be used in place of individual scalar values for any parameter. This is accomplished with the following script syntax:

```
vary from: <value> to: <value> by: <iteration value>
```

A simple example of using range expressions would be in the testing of trig functions.

Consider the following code fragment:

```
package math is
type ANGLE is new float range 0.0..359.99;
  function SINE (A : ANGLE) return float;
  function ATAN (X, Y : ANGLE) return float;
```

Using the range testing capability you could build a single test case that would exercise the sine function in the first quadrant every one tenth of a degree. The test script to accomplish this is shown here:

```
TEST.SUBPROGRAM:SINE
TEST.NEW
TEST.VALUE: MATH.SINE.A:vary from: 0.0 to: 90.0 by: 0.1
TEST.EXPECTED: MATH.SINE.return:0.0..1.0
```



Note: The range values defined allow you to loop forward or backward through the specified range as in the following examples:

Forward looping

```
TEST.VALUE:MATH.SINE.A:vary from: 0.0 to: 90.0 by: 0.1
```

Backward looping

```
TEST.VALUE:MATH.SINE.A:vary from: 90.0 to: 0.0 by: - 0.1
```

Varying Multiple Parameters Simultaneously

When setting up a test case for range testing, you may vary different parameters simultaneously. This means that each parameter can have a different range expressions within a single test case. This capability enables you to vary multiple values simultaneously, or vary one value while holding others constant

The following test script shows how two parameters can be varied simultaneously:

```
TEST.SUBPROGRAM:atan
TEST.NEW
TEST.VALUE:MATH.atan.X:vary from 1.0 to: 90.0 by: 1.0
TEST.VALUE:MATH.atan.Y:vary from 1.0 to: 90.0 by: 1.0
TEST.EXPECTED:MATH.ATAN.return:45.0
TEST.END
```

Note that in this example the expected result is used to ensure the value returned by the ATAN function is always 45 (ATAN of 1).

The following script shows a test case where one parameter is held constant while another paramter is varied:

```
TEST.SUBPROGRAM:atan
TEST.NEW
TEST.VALUE:MATH.atan.X:0
TEST.VALUE:MATH.atan.Y:vary from 1.0 to: 90.0 by: 1.0
TEST.EXPECTED:MATH.ATAN.return:0.0
TEST.END
```

Again in this example the expected result is used to ensure the value returned by the ATAN function is always 0 (ATAN of 0).



Note: By default you may simultaneously vary up to 20 parameters at a time in a given test case. If you need to vary more, edit the Maximum Varied Parameters option. It is located in the Tools => Options dialog, Builder tab. For this option to take effect, you will need to rebuild the current environment. The easiest way to do this is with Environment => Rebuild Environment. If multiple values are varied in the same test case, you can choose to have the values vary in parallel or combinatorially.

Range of Values for Expected Results

The expected results capability enables the entry of a range of values for any numeric scalar value. When entering a value in a test case script for an expected value, simply use the syntax <min> .. < max>. By using the range feature for the expected results you can also perform some cursory checking of the results. In this case we know that the sine of any angle in the first quadrant should be between 0 and 1. The power of this capability is that you can use a single test case to verify that the sine function did not terminate abnormally over the entire range of values and also that the computed value is within a broad range of acceptable values for this function.

Because all of the actual values would show up in the test results from this test case, it would also be possible to go back and verify the output value for any single data point.

List of Values

When entering a list of values in a test case script, simply use the syntax item1, item2, itemN as in the following examples:

```
TEST.VALUE:MANAGER.PLACE.ORDER.SEAT:1,2,4
```

If a certain value should be repeated, rather than list it multiple times, just include the repeat count in parentheses before the item.

```
TEST.VALUE:MANAGER.PLACE_ORDER.SEAT:1, (4)2, 4
```

In a list of expected values, if any value is acceptable, use the special notation <<ANY>>. Using <<ANY>> indicates that any value that is compared to that spot in the list is acceptable. <<ANY>> doesn't make a test case pass or fail; it is used as a placeholder in a list of expected values.

```
TEST.VALUE:MANAGER.PLACE_ORDER.SEAT:1,(4)2,<<ANY>>
```



Note: For a list of values, if the number of values entered is less than the number of calls to the particular subprogram, VectorCAST will use the last parameter value for the remaining calls.

Test Values Spanning Multiple Lines

The continuation character '\' enables TEST.VALUE commands to span many lines. This is useful when a script value line becomes large. The continuation character may be placed either after a value or used to break a value between lines. When VectorCAST creates a test case script it will format every line to fit in eighty columns. However, it is not necessary to format the script file in this way in order for VectorCAST to read it.

Example:

```
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.BEVERAGE:WIiNnEe,NOo_ORDER,\
BEER,WINE,SODA,MIXED_DRINK,\
MIXED_DRINK,NO_ORDER
```



Note: The Script Reader will ignore all white space so you may indent and space the values for maximum readability.

Working with Overloaded Subprograms

When two or more subprograms with the same name exist in the same unit, each instance of the overloaded subprogram will have the parameter type list appended to the subprogram name. The following example has commands to set the value of the parameter VAL for the four instances of the overloaded subprogram P that have parameter types of: integer, character, string and boolean.

```
TEST.VALUE:TEST1.P(integer).VAL: 3
TEST.VALUE:TEST1.P(character).VAL:'d'
TEST.VALUE:TEST1.P(string).VAL:"this is a test"
```

```
TEST.VALUE:TEST1.P(boolean).VAL:TRUE
```

Adding User Code to a Test Script

In this example, the unit under test has three subprograms:

- InitializeFile – returns a file pointer to the specified file name
- WriteToFile – write a string to the file pointed by the file pointer passed in
- CloseFile – close the file pointed to by the file pointer passed in

To test these subprograms in VectorCAST, you would create a test case for each subprogram and then call the three test cases from a compound test case. The problem is there is no static way for VectorCAST to pass the file pointer created by InitializeFile to the other two subprograms. With parameter user code, you have the ability to set data dynamically. In this example, you want to pass the file pointer returned from InitializeFile as an input into WriteToFile and CloseFile. With user code, you use the name of the file pointer returned by InitializeFile as a parameter for the other two files. The following test case script shows how to do that:

```
-- Test Case: INITIALIZEFILE.001
TEST.SUBPROGRAM:INITIALIZEFILE
TEST.NEW
TEST.NAME:INITIALIZEFILE.001
TEST.VALUE:FILEIO.INITIALIZEFILE.FILENAME:"DUMMY.TXT"
TEST.END
--
-- Test Case: WRITETOFILE.001
TEST.SUBPROGRAM:WRITETOFILE
TEST.NEW
TEST.NAME:WRITETOFILE.001
TEST.VALUE_USER_CODE:FILEIO.WRITETOFILE.FILE
<<FILEIO.WRITETOFILE.FILE>> := <<FILEIO.INITIALIZEFILE.return>>;
TEST.END_VALUE_USER_CODE:
TEST.END
--
-- Test Case: CLOSEFILE.001
TEST.SUBPROGRAM:CLOSEFILE
TEST.NEW
TEST.NAME:CLOSEFILE.001
TEST.VALUE_USER_CODE:FILEIO.CLOSEFILE.FILE
<<FILEIO.CLOSEFILE.FILE>> := <<FILEIO.INITIALIZEFILE.return>>;
TEST.END_VALUE_USER_CODE:
TEST.END
```

The highlighted lines show the scripting language used to set the parameter “FILE” to be the return value from the call to InitializeFile. You can also put any other executable code between the TEST.VALUE_USER_CODE and TEST.END_VALUE_USER_CODE delimiters.

APPENDIX D: LIMITATIONS AND RESTRICTIONS

The following are the limitations for compilers and of the VectorCAST tools. When possible a work-around or option has been suggested for the specific limitation identified.

Compiler Limitations

VectorCAST is subject to all limitations identified in the documentation provided with each of the compilers supported by VectorCAST. The following sections identify these specific limitations and apply to all platforms for each compiler supported by VectorCAST.

All Compilers

Support for Pre-defined Language Components

With some of the compilation systems, the source files for units which are part of the pre-defined language environment (CALENDAR, SYSTEM etc.) are not provided. This makes it impossible to manipulate the values of objects or parameters which use types from these packages. To work around this problem, VectorCAST provides a capability for users to supply their own source files for any unit that is not available to VectorCAST.



Note: A unit is not available when VectorCAST produces the status alert: "Source not Found: {unit_name}"

To make use of this feature, store a copy of the source code that you want VectorCAST to use for a particular unit into a file with the name of the unit suffixed with ".a" (e.g. for package CALENDAR use filename calendar.a). This file must be stored in the VectorCAST distribution area on the following path:

```
Linux => $VECTORCAST_DIR/source/compiler_name
DOS => %VECTORCAST_DIR%\SOURCE\compiler_name
```

where *compiler_name* is one of the following:

ada_world	apex	ddci	dec_ada
gh95	gnat	green_hills	legacy
obj_ada	powerada	risc	vads



Note: VectorCAST will only refer to this directory if it cannot find the source file for a unit through the normal means. You **CANNOT** override the compiled version of a source file by inserting an alternate version into this directory structure!

It is not necessary for the user supplied source file to be a complete specification of the unit, it is only required that the source provided is syntactically correct. Therefore if you are only using one type from a pre-defined package you can create a file with only that type definition.

This file will NOT be compiled, it is only used by the VectorCAST parser to allow users to pass information into VectorCAST.

As an example, assume we are using the Alsys Ada World compiler under Linux and our code to be tested uses the type CALENDAR.DAY_NUMBER. Since Alsys does not supply the source code with for CALENDAR, you would be unable to manipulate a parameter or object of type DAY_NUMBER with VectorCAST. To work around this, simply create a file called **`calendar.a`** and store it into **`$VECTORCAST_DIR/source/ada_world`**

```
package CALENDAR is
type DAY_NUMBER is range 1..31;
end CALENDAR;
```

VADS / SunAda

Library units should be compiled from separate files.

For example: package specifications and package bodies should be in separate source files. Combining specifications and bodies in the same source file may result in unpredictable behavior.

Loader Commands

Any commands that are normally passed to the loader should be either embedded in the source file for the unit under test using the pre-defined pragma "link_with" or should be inserted into the parent Ada library "ada.lib" file as "WITHn" directives.

VADS unsigned_types package

If you use the types that are provided via the VADS UNSIGNED_TYPES package you will not normally be able to set values for parameters or objects of these types with VectorCAST. The reason for this is that an Ada source for this package does not exist, only a pseudo source file exists and therefore VectorCAST has no Ada source to parse to determine the types.

If you wish to set values of parameters and/or objects which use types from the UNSIGNED_TYPES package you will need to modify the source file for this package. The source file can be found by setting you working directory to any VADS Ada library and executing the VADS "which" command for unit "unsigned_types". If you then edit this file (root privilege may be required) you will find that the file consists of all comment lines. To allow VectorCAST to parse the types in this package you need to uncomment the package declaration, the three types definitions (unsigned_integer, unsigned_short_integer, and unsigned_tiny_integer), and the end package declaration. An example of the five uncommented lines is shown below:

```
package unsigned_types is

-- 0..4294967295

  type unsigned_integer is range 0 .. (2**32 - 1);

-- 0..65535
```

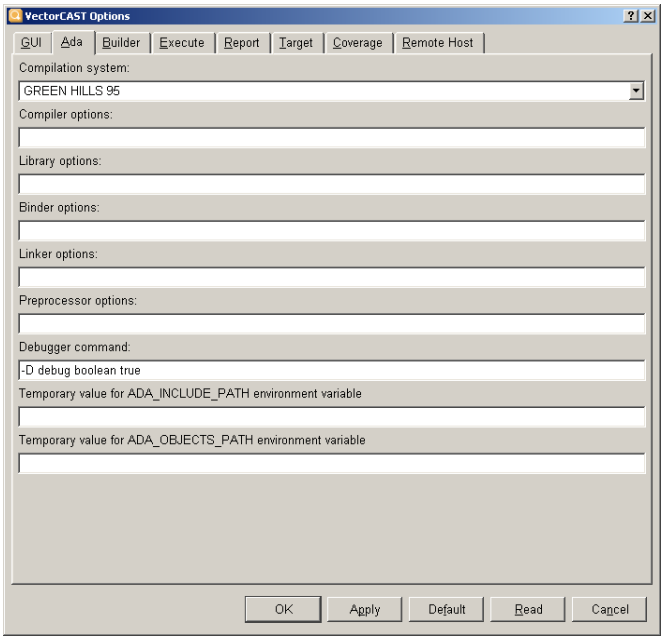
```
   type unsigned_short_integer is range 0 .. (2**16 - 1);

-- 0..255

   type unsigned_tiny_integer is range 0 .. (2**8 - 1);

end unsigned_types;
```

After making these changes simply run VectorCAST normally and an environment will be built that enables you to manipulate objects and parameters of these types.

VectorCAST Option to Control VADS Ada Preprocessor Utility

A VectorCAST option exists to allow VADS users to supply a list of command line declarations to the Ada Pre-Processor. This option should be used to supply the entire option text including the "-D".

For example, to define a boolean variable called debug and set its value to "true," set the option to be: "-D debug boolean true"



AdaWorld / ActivAda

Compiling your Source Files

To run VectorCAST with AdaWorld, all units in the unit under test closure must have been compiled with the "copy=yes" option. As shown below:

ada compile copy=yes my_unit.ada

This enables VectorCAST to extract the source file from the Ada library.

Source lines equal-to or greater-than 80 characters should be avoided.

AdaWorld inserts a new line marker into source lines that are 80 characters or longer in the extracted version. Even for comment lines! This may yield unpredictable results for the VectorCAST parser since the resulting source file is not compilable Ada.

A work around is to use an AdaWorld start-up file that contains the following command:

```
global.terminal (width=255)
```

Refer to the compiler documentation for information regarding start-up files.



Note: For ActivAda, the location of the **ADA.DEF** file is passed to VectorCAST via the **ADADEF** environment variable. Set the value of this environment variable to point at the **ADA.DEF** file to be used.

To run VectorCAST with the AdaWorld debugger (AdaProbe), the unit under test must have been compiled with the "debug=yes" option:

```
ada compile debug = yes copy=yes my_unit.ada
```

Setting Compiler Options

All compile, link, and command level options should be provided in the appropriate AdaWorld initialization file as described in the AdaWorld documentation.

VectorCAST (host) requires the following parameters to be set up with default values via this method:

```
default family = /home/bsmith/my_family
global.terminal (width=255)
default copy = yes
```

VectorCAST (TARGET) requires the following additional parameters to be set up with default values via this method:

```
default directives = /home/bsmith/pgm130.cmd
default modules = /home/bsmith/user130c.o
default target = <target_name>
```

Source Not Found problems

The status message: "ERROR" Source not Found <unit> may be generated if modifications have been made to the source file for the <unit> library unit. If the source file that the <unit> was compiled from has been moved or modified since the time of compile it will not be found. This is an AdaWorld limitation.

Some library units that are part of the Ada Run Time, (i.e. SYSTEM, TEXT_IO) cannot be accessed by VectorCAST under AdaWorld because AdaWorld does not have source files for these entities.

ObjectAda

Specifying the Parent Library

The path to the parent Ada library is the relative or absolute path to the directory in which your code to be tested has been compiled. This is the directory that contains the UNIT.MAP file and the ADA.LIB file.

Legacy Ada / PowerAda

Specifying the Parent Library

When prompted for the path to the parent library when building a new environment you should enter the full path name (including the file name) of the Legacy Ada library that contains the units to be tested. For example, if the Ada library file is called: "alib.list" and is located in "/home/test" then you would enter:

```
/home/test/alib.list
```

The parent library that is used to build a VectorCAST environment **must** have absolute path names for **all** libraries For example your Ada library file should contain something similar to the following:

```
/home/first_subsystem/adalib

/home/second_subsystem/adalib

/home/base_types/adalib
```

Library units should be compiled from separate files. Library units should not be combined in the same source file. For example package specifications and package bodies should be in separate source files. Combining specifications and bodies in the same source file may result in unpredictable behavior.

Green Hills Ada

Specifying the Parent Library

When prompted for the path to the parent library when building a new environment, you should enter the full path (absolute, or relative to the working directory) to the directory that contains the Green Hills Ada library file (ADA.LIB) that contains the units to be tested.

If the parent library contains links to other libraries, these links should utilize absolute path names **not** relative path names.

Using older versions of Green Hills Ada

The command verbs for the GH95 compiler have changed for version 1.8.9. VectorCAST contains the new command verbs as the default. If you are using Green Hills Ada on Windows, and using a pre-1.8.9 version, you must set the configuration option VCAST_OLD_GH95 to TRUE.

Compiling your Source Files

Library units should be compiled from separate source files. Combining specifications and bodies in the same file or compiling multiple unrelated units in the same file can lead to unpredictable behavior during environment construction.

APEX

Starting VectorCAST

VectorCAST must be run from an Apex shell window in order for VectorCAST to have access to the Apex Command Line Interface commands. The Apex shell window is accessed under the Apex Tools menu. It is also possible to integrate VectorCAST into the Apex GUI; refer to your Apex documentation for details on modifying the Apex Menu Files.

VectorCAST must be run from within an Apex sub-system for Summit, or from within a Rational subsystem for ClearCase. Generally you will run VectorCAST from the parent directory that contains the Apex View subdirectories; you cannot run from within a view directory (Summit).

Specifying the Parent Library (Summit)

The path to the parent Ada library is the relative or absolute path to the Apex View that contains the versions of the units to be tested. VectorCAST will be creating a copy of the "source view " that is provided as the "parent" library. If the source view contains checked-out files, the copy view will fail. You should make sure to check-in all files in the "source view" before building a new VectorCAST environment based on this view.

Specifying the Parent Library (ClearCase)

The path to the Ada library is the relative or absolute path to the Rational subsystem top level directory. The directory name will generally end in ".rss".

Compiling the VectorCAST Tutorials

When building the tutorial examples from the provided source files you will need to change the names to correspond to the Apex unit naming conventions. Change the .ads extensions to **`.1.ada`** and the .adb extensions to **`.2.ada`**.

GNAT

VectorCAST is compatible with GNAT 3.04 and higher. Versions of GNAT prior to 3.04 will not work with VectorCAST.

Specifying the Parent Library

The path to the parent Ada library is the relative or absolute path to the directory in which your code to be tested has been compiled (the directory that contains the .ali files).

File names

GNAT normally requires every unit to be in a single file, where the file name matches the unit name (e.g. the specification for package **fred_pkg** will be in file fred_pkg.ads). When source code does not follow this requirement, GNAT has two methods for resolving discrepancies. The easiest method to use is the GNAT utility "gnatchop" – this utility will copy Ada code from a file into the correctly named file (e.g. if file fred.ada contains the specification and body for **fred_pkg**, running "gnatchop" on fred.ada will yield two files: fred_pkg.ads and fred_pkg.adb). Another method is to use the gnat.adc file, which enables the user to map unit names to file names regardless of conventions (e.g. **pragma SOURCE_FILE_NAME (FRED_PKG, SPEC_FILE_NAME => "fred.ads");**).

Setting GNAT Environment Variables

When compiling or linking, GNAT searches the current directory and the GNAT libraries for package and linker object information. If you are using a hierarchical library structure (units in one directory depend on units in another directory), you can either specify the other directories on the command line or you can set environment variables for the search locations. The environment variable **ADA_INCLUDE_PATH** should contain a list of directories to search for Ada unit information. The environment variable **ADA_OBJECTS_PATH** should contain a list of directories to search for Ada linker objects. Normally, you do not need to include the GNAT Ada Run Time libraries in these environment variables. However, because VectorCAST does not know where the Run Time libraries are located, you need to add the directory containing the standard Ada units in the **ADA_INCLUDE_PATH** environment variable, and the directory containing the standard Ada Run Time libraries in the **ADA_OBJECTS_PATH** environment variable. Typically, the directory containing the standard Ada units will be the adalib directory in the installation area, and the directory containing the standard Ada Run Time libraries will be the **adainclude** directory in the installation area.

An example using the GNAT Environment Variables

Assume that you have two Ada libraries "lib1" and "lib2" as shown below.

/home/lib1

types.ads

/home/lib2

manager.ads

manager.adb

database.ads

database.adb

manager_driver.adb

To compile the "lib2" files that depend on types.ads you would enter the following command:

```
gcc -c -I/home/lib1 manager.adb
gcc -c -I/home/lib1 database.adb
gcc -c -I/home/lib1 manager_driver.adb
```

To “bind” and link the main procedure: manager_driver, you would enter the following command:

```
gnatbind -aO/home/lib1 manager_driver.ali
gnatlink -aO/home/lib1 manager_driver.ali
```

Using the environment variables, you can omit the “-I” and “-aO” options as follows:

```
setenv ADA_INCLUDE_PATH /home/lib1
setenv ADA_OBJECTS_PATH /home/lib1
gcc manager.adb
gcc database.adb
gcc manager_driver.adb
gnatbind manager_driver.ali
gnatlink manager_driver.ali
```

DDC-I Native (WinNT)

To use VectorCAST with the DACS compiler on Windows NT, you must set the ADA_LIBRARY environment variable to point to the appropriate Ada Run-Time library. For example

\\\\DATASERVER\DACS\NATIVE\root.alb.

Specifying the Parent Library

The path to the parent Ada library is the relative or absolute path to the DDC-I library file (having a file extension of ".alb").

VectorCAST-Specific Limitations

The following is a list of limitations imposed by VectorCAST and work-around solutions.

Subprogram Renames are not available for testing in the compilation unit where they are renamed.

Option: Test all subprograms in the context of the unit where they are defined.

All required external information must be provided for the unit under test to be run. That is, if the unit solicits keyboard input during processing, this input must be provided by the tester. Similarly any external input retrieved from any other hardware device should be simulated or actually provided to ensure normal test function.

Options: None, tests will either hang or not run properly if expected input is not provided.

Maximum number of dimensions for any single array is 26.

Options: Provide data values for arrays with more than 26 dimensions within the User Code package.

Stand-alone subprogram bodies cannot be stubbed.

Options: Provide a manually generated stub for these units and use the Re-Link command to regenerate the executable test harness.

Code that contains an infinite loop and does not call a stubbed unit should be stubbed or modified to have a finite number of loop iterations. Without doing this, control will never be passed back to the operating system. The result of this is that VectorCAST will hang after each test case execution.

Options: Use the appropriate operating system command to kill the VectorCAST test harness after each test execution.

Multiple library units within one source file may generate unpredictable results.

Options: If multiple units exist in a single file the "end" statements for the packages or procedure must include the package /procedure name. This does not guarantee success, the best option is to separate units into separate source files.

When building environments in which some of the parameters are unconstrained array types you may get compile warnings, link errors, or execution errors if the maximum size of the unconstrained array type is larger than the compilation system enables.

Options: See the Usage Notes: "Working with Unconstrained Arrays".

Parameters Defined with a 'class attribute are treated by ignoring the 'class attribute and allowing manipulation of the parameter as if the type mark was the base type without the 'class.

Option: Use the User Code to set values for the parameter using classes derived from the parent class.

The Unit Under Test can be a private package; VectorCAST automatically turns on whitebox testing. A private package cannot be stubbed by VectorCAST. Private packages CAN be part of the execution closure.

Option: None

APPENDIX E: ENVIRONMENT VARIABLES

The following environment variables affect the behavior of VectorCAST. To set the value of an environment variable, use the following syntax:

- **Linux (csh):** setenv VCAST_MAX_LINE_LENGTH 7000
- **Linux (sh):** export VCAST_MAX_LINE_LENGTH=7000
- **Windows (DOS):** set VCAST_MAX_LINE_LENGTH 7000
- **Windows:** Use **Control Panel => System icon => Advanced system settings => Environment Variables...**

You must set the environment variables *before* starting VectorCAST.

Environment Variables

VCAST_ALTERNATE_REPORT

If this environment variable is defined, then a tab-separated report is generated after test execution, in addition to the regular execution results. Use **`clicast reports alternate`** to extract the report to a file or standard output.

VCAST_APEX_COPY_FILES

When building environments with Apex, copy all source files to local view rather than creating Linux links.

VCAST_DISABLE_ADA_ARRAY_RANGE_CHECKING

Disables the ability to use index notation for arrays, leaving only positional notation. This environment variable overrides the Builder option “Disable direct array indexing.”

VCAST_DISABLE_FILE_DELETE

Do not delete temporary files before running the driver.

VCAST_GNAT_HIE_DASH_E

Add ‘-e’ for GNAT HIE Linker.

VCAST_GNAT_LD_OPTIONS

Options passed to GNAT linker.

VCAST_IGNORE_COMPILE_ERRORS

Do not stop compiling when an error is encountered. For use if your compiler returns an exit code not equal to zero even when successful.

VCAST_IGNORE_FILE_EXISTS_CHECK_FOR_COVERABLE

Allow coverage even if VectorCAST can’t find the source file.

VCAST_IGNORE_TYPEMARKS

VCAST_IGNORE_TYPEMARKS can be used to stop VectorCAST from building type processing functions for particular types. In the Parameter Tree, the types will require user code to be set. This feature is useful for types from third party libraries that will never be manipulated by the user. Multiple type marks should be separated by commas.

VCAST_LEGACY_CFG_FILE_INHERITANCE

Read *.CFG files from installation directory and user's home directory before reading from the local directory. If the environment variable VCAST_LEGACY_CFG_FILE_INHERITANCE is set, any options inherited from the installation directory or user's home directory are saved to the local *.CFG file, creating a complete set of options from all sources in one place (the local *.CFG file). Once this is done for each environment, the environment variable no longer needs to be set when running the regression script.

VCAST_MAX_INSTRUMENTATION_LINE_LENGTH

Set the maximum length of a source line generated by the coverage instrumentation. When the value of the environment variable is set to 80 or less, lines in the instrumented source code are wrapped at a convenient location around 80 characters. If the value is greater than 80, lines are wrapped at that number of characters.

VCAST_NEVER_STUB_LIST

A comma-separated list of unit names (without the file extension) that should not be stubbed even when the STUB ALL option is selected. This environment variable can also be set to the full path to a file, with one unit per line. This feature is useful when the never-stub list is long, or when sharing the list with others.

VCAST_NO_STANDARD_PKG_USAGE

By default, the Multiunit Whitebox harness redefined TRUE and FALSE by using STANDARD.BOOLEAN. This option disables that override.

VCAST_NO_USER_EXCEPTION_SUPPORT

Do not support user-defined exceptions.

VCAST_OLD_ACTIVADA

Indicates ActivAda version 5.2 or below.

VCAST_PARENT_LIBRARY

Over-rides the setting for ENVIRO.PARENT_LIB in an environment script that was created for a regression test. Useful for running a regression test with a different Parent Library than then one that was used when the regression test was created.

VCAST_RAVEN_HEAP_SIZE

For RAVEN target, set size of heap VectorCAST creates for each pointer type.

VCAST_RAVEN_STDOUT

Run Raven target using standard output.

VCAST_RECURSION_DEPTH

Maximum number of levels to recurse when trying to generate basis path data.

VCAST_REGRESSION_COMMAND

An additional command that should be added to regression test shell script or batch file.

See also "To Post-Process the Regression Scripts " on page 142.

VCAST_RPTS_PRETTY_PRINT_HTML

This environment variable has turned into a Report option.

VCAST_S68_LOCATION

Define location of Green Hills 68000 simulator.

VCAST_TIC_INDICATES_MACHINE_CODE

Tic mark (`) at beginning of line indicates line contains machine code.

VCAST_TORNADO_OLD_SCRIPT

When running vxWorks, do not complain about missing symbols. In VectorCAST version 5.0b, this environment variable was renamed **VCAST_DONT_CHECK_FOR_VXWORKS_UNRESOLVED_SYMBOLS**.

VCAST_USE_REMOVE_IMPORTS

Use Apex command to remove imports from copied views; otherwise, the data files are modified directly.

VCAST_WHITEBOX

Force whitebox environment building. This environment variable overrides the setting of the Whitebox option in the **Tools => Options dialog, Builder tab**.

VCV_EDIT_LINE_NUMBER

This environment variable is set when opening files in an external editor. This variable is set to the line number of the selected function. The external editor can then be set to a script command that uses the environment variable VCV_EDIT_LINE_NUMBER to go to a specific line number in the external editor. When VCV_EDIT_LINE_NUMBER is set to 0, the file opens and sets the cursor to the top.

VCV_ENVIRONMENT_DIR

This environment variable is set to the full path to the current environment directory whenever the environment is opened.

This environment variable is available to all commands invoked by VectorCAST from the open environment, including:

- C_PREPROCESS_CMD,
- C_COMPILE_CMD,
- C_LINK_CMD,
- VCAST_CMD_DURING_ENV_OPEN,
- VCAST_CMD_DURING_ENV_CLOSE,
- VCAST_PRE_EXECUTE_CMD, and
- C_EXECUTE_CMD.

VCV_ENVIRONMENT_NAME

This environment variable is set to the name of the environment while the environment is open.

This environment variable is available to all commands invoked by VectorCAST from the open environment, including:

- C_PREPROCESS_CMD,
- C_COMPILE_CMD,
- C_LINK_CMD,
- VCAST_CMD_DURING_ENV_OPEN,
- VCAST_CMD_DURING_ENV_CLOSE,
- VCAST_PRE_EXECUTE_CMD, and
- C_EXECUTE_CMD.

VECTORCAST_DIR

The location of VectorCAST binary executables (installation directory).

VECTOR_LICENSE_FILE

The full path to VectorCAST's vendor-specific FLEXlm license file.

APPENDIX F: CLICAST - COMMAND LINE VECTORCAST

Quick Start

The following is a list of useful commands to get online CLICAST help. Each command is preceded by **$VECTORCAST_DIR/** (Linux) or **%VECTORCAST_DIR%** (Windows).

To see a list of all commands, type:

```
clicast help all
```

To see a list of categories, type:

```
clicast help
```

To see a list of commands in a single category, type:

```
clicast help <category>
```

where **<category>** is one of the following:

- ENvironment (or just “EN”)
- EXecute
- Report
- TEst
- TOol
- Option
- Get_option
- Language
- Cover

To see a list of all targets, type:

```
clicast -x tgt
```

To see a list of all options, type:

```
clicast help options all
```

To see a list of option categories, type:

```
clicast help options
```

To see a list of options in a single category, type:

```
clicast help options <category>
```

where **<category>** is one of the following:

- Language
- Builder
- Execute
- Report
- Target
- Coverage

To find out the value of an option, type:

```
clicast get_option <option>
```

Command Format

CLICAST is invoked using the following syntax:

Linux systems:

```
$VECTORCAST_DIR/clicast -eenv -uunit -ssub -ttestcase command arguments
```

Windows systems:

```
%VECTORCAST_DIR%\clicast /e:env /u:unit /s:sub /t:testcase command arguments
```

where ***env*** is the name of the environment, ***unit*** is the name of a unit under test, ***sub*** is the name of a subprogram, and ***testcase*** is the name of a testcase. If the environment has only one UUT, then **-u unit** is optional.

Throughout this User Guide, the corresponding CLICAST command is presented after each feature or option available in the VectorCAST application. The CLICAST command uses the following format:

```
clicast -lc -e <env> command | option arguments

Description of the command, option, and arguments.
```

The following conventions are used:

- The vertical bar | stands for “or”, indicating that one of several choices is to be included in the command.
- Square brackets around a parameter indicate that a parameter is optional. For example,

  ```
  -e <env> [-u <unit>]
  ```

  indicates that the environment name must be specified, but the unit name can optionally be

specified on the command line.

- Angle brackets indicate that a source file name or script file name must be substituted. For example,

```
<output file>
```

indicates that an output file name must be provided.

- Square brackets and angle brackets can be combined to indicate that a substitution is optional. For example,

```
[<output file>]
```

indicates that if the optional output filename is not present, standard output is used.

- Capital letters in a command name indicate the minimum that needs to be typed to uniquely identify the command. For example,

```
clicast ENvironment Build <scriptfile>
```

indicates that the abbreviated command **`clicast EN B <scriptfile>`** can be used. To specify <<COMPOUND>> or <<INIT>> as the subprogram or when used in a testcase, enclose the name in quotes, as in :

```
clicast -e TUTORIAL_C -u manager -s "<<COMPOUND>>" -t "<<COMPOUND>>.001" exec
run
```

The Options Append Command

For **`clicast`**, an append command is available to append values to an existing option. The main benefit is adding to an existing option, for example a missing flag, without having to repeat the entire option. A common use case is to update older commands to add the missing 64-bit parser flags.

In this example the original flags might be:

```
C_EDG_FLAGS: -w --g++ --gnu_version 60300 --type_info_in_std --type_traits_helpers -
-c++98
```

This is fixed with the '**`OPTIONS_Append`**' command:

```
clicast -e C_EDG_FLAGS OPTIONS_Append '--x86_64 --64_bit_target'
```

```
clicast -lada  OPTIONS_Append <option> <value>

Append value to specified text-based option. Appended values are preceded
by an inserted space, except for VCAST_ENVIRONMENT_FILES and VCAST_
SUPPRESS_COVERABLE_FUNCTIONS, which use a comma instead.
```

Console Report Format

The Console Report provides command line output with passes and failures at the iteration level and Expected/Actual value mismatch at the iteration level. After all tests have run, a test summary is

shown along with a list of all failing tests.

The clicast-only option VCAST_CONSOLE_REPORT_FORMAT has four levels of output:

- DEFAULT
- LEGACY
- VERBOSE
- QUIET

```
clicast -lada VCAST_CONSOLE_REPORT_FORMAT CONSOLE_REPORT_DEFAULT | CONSOLE_
REPORT_LEGACY | CONSOLE_REPORT_VERBOSE | CONSOLE_REPORT_QUIET

By default (CONSOLE_REPORT_DEFAULT), VectorCAST shows command line output
with passes and failures at the iteration level, and Expected/Actual value
mismatch at the iteration level. After all tests have run, a test summary
is shown along with a list of all failing tests. With Legacy output
(CONSOLE_REPORT_LEGACY), the previous style of console output is shown.
Verbose output(CONSOLE_REPORT_VERBOSE) prints both Default and Legacy
Output. When set to quiet (CONSOLE_REPORT_QUIET), minimal test/run
information is shown in the console output.
```

The following sections illustrate the output for the four levels.

Default Output

By default (CONSOLE_REPORT_DEFAULT), VectorCAST shows command line output with passes and failures at the iteration level, and Expected/Actual value mismatch at the iteration level. After all tests have run, a test summary is shown along with a list of all failing tests. Output for testcases failing due to a signal failure (such as segmentation violation, for example) are not currently supported.

This example output shows that PLACE_ORDER.002 failed due to one of the range iterations expecting the CheckTotal to be 17, but it was 18.

```
[  PASS  ] MANAGER.PLACE_ORDER - PLACE_ORDER.001
[  FAIL  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002
[  PASS  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002.rangeIter1
[  FAIL  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002.rangeIter2
[        ]   Expected: CheckTotal = [17]
[        ]   Actual:   CheckTotal = [18]
[  FAIL  ] <<COMPOUND>> - <<COMPOUND>>.001
[========] 2 tests ran.
[ PASSED ] 1 tests.
[        ] the following 1 tests failed:
[ FAILED ]   MANAGER.PLACE_ORDER - PLACE_ORDER.002
[        ]   MANAGER.PLACE_ORDER - PLACE_ORDER.002.rangeIter2
```

Legacy Output

With Legacy output (CONSOLE_REPORT_LEGACY), the previous style of console output is shown and is displayed by default for VectorCAST Project output.

```
Running all MANAGER.PLACE_ORDER test cases
Running: PLACE_ORDER.001
Preparing Test Data
Running Test Case
Running Test with command: ...\UUT_INST.EXE
Processing Execution Data
Updating Coverage Data
Test Execution Complete
Running: PLACE_ORDER.002
Preparing Test Data
Running Test Case
Running Test with command: ...\UUT_INST.EXE
Processing Execution Data
Test Execution Complete
Completed Batch Execution processing
```

Verbose Output

Verbose output (CONSOLE_REPORT_VERBOSE) prints both Default and Legacy output.

```
Running all MANAGER.PLACE_ORDER test cases
Running: PLACE_ORDER.001
Preparing Test Data
Running Test Case
Running Test with command: ...\UUT_INST.EXE
Processing Execution Data
[  PASS  ] MANAGER.PLACE_ORDER - PLACE_ORDER.001
Updating Coverage Data
Test Execution Complete
Running: PLACE_ORDER.002
Preparing Test Data
Running Test Case
Running Test with command: ...\UUT_INST.EXE
Processing Execution Data
[  FAIL  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002
[  PASS  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002.rangeIter1
[  FAIL  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002.rangeIter2
[        ]   Expected: CheckTotal = [17]
[        ]   Actual:   CheckTotal = [18]
Updating Coverage Data
Test Execution Complete
[========] 2 tests ran.
[ PASSED ] 1 tests.
[        ] the following 1 tests failed:
[ FAILED ]   MANAGER.PLACE_ORDER - PLACE_ORDER.002
[        ]   MANAGER.PLACE_ORDER - PLACE_ORDER.002.rangeIter2
 Completed Batch Execution processing
```

Quiet Output

When set to quiet (CONSOLE_REPORT_QUIET), minimal test/run information is shown in the console output.

Compound Test Output

Compound test execution is displayed in a similar manner. The beginning of the compound is listed at the top with no status in the brackets. The end of the compound test execution is listed with the status ("PASS" or "FAIL"). The individual slots in the compound are also listed with a status, but do not add to the number of tests executed, of course.

The following output demonstrates the default display for a compound test with three slots:

- an <<INIT>> test case that compares a global string and is failing
- PLACE_ORDER.001 that passes
- and PLACE_ORDER.002 that fails in the second range iteration

```
[        ] <<COMPOUND>> - <<COMPOUND>>.001
[  FAIL  ] MANAGER.<<INIT>> - <<INIT>>.001.slot1
[        ]   Expected: STR1 = [expected string]
[        ]   Actual: STR1 = [Init]
[  PASS  ] MANAGER.PLACE_ORDER - PLACE_ORDER.001.slot2
[  FAIL  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002.slot3
[  PASS  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002.slot3.rangeIter1
[  FAIL  ] MANAGER.PLACE_ORDER - PLACE_ORDER.002.slot3.rangeIter2
[        ]   Expected: CheckTotal = [17]
[        ]   Actual:   CheckTotal = [18]
[  FAIL  ] <<COMPOUND>> - <<COMPOUND>>.001
[========] 1 tests ran.
[ PASSED ] 0 tests.
[        ] the following 1 tests failed:
[ FAILED ]   <<COMPOUND>> - <<COMPOUND>>.001
[        ]   MANAGER.<<INIT>> - <<COMPOUND>>.001.slot1 - <<INIT>>.001
[        ]   MANAGER.PLACE_ORDER - <<COMPOUND>>.001.slot3 - PLACEORDER.002
[        ]   MANAGER.PLACE_ORDER - PLACE_ORDER.002.slot3.rangeIter2
```

APPENDIX G: CLICAST - TOOL OPTIONS REFERENCE

This reference contains an alphabetized list of VectorCAST's configurable options.

Tool Options

ADA_DEBUG_CMD

```
clicast -lada Option ADA_DEBUG_CMD <debugger command>
```

Category: Language Options

Description: This option will over-ride the default debug command that VectorCAST uses to invoke the debugger. For example, the default debug command for GNAT is: gdb, if you prefer to use gvd, you may enter that command here.

ADA_EXECUTE_CMD

```
clicast -lada Option ADA_EXECUTE_CMD <execute command>
```

Category: Target Options

Description: This option must be set when 'Execution process uses stdin/stdout for file I/O' is True. When using this execution method, you provide the path to your custom script, batch file, or other mechanism that handles downloading to the target, test execution, and any other preparation or post-processing. The script is passed a single argument with the name of the test harness executable (UUT_INTE for the normal harness, or UUT_INST for the coverage-instrumented harness). Use this option to override the default test execution method for your particular target. (This option will override the 'Execute command' option).

ADACAST_IO_BODY

```
clicast -lada Option ADACAST_IO_BODY <path to 'adacast_io' package body>
```

Category: Target Options

Description: Package body containing I/O functionality used by harness.

ADACAST_IO_SPEC

```
clicast -lada Option ADACAST_IO_SPEC <path to 'adacast_io' package spec>
```

Category: Target Options

Description: Package spec containing I/O functionality used by harness.

BINDER_OPTIONS

```
clicast -lada Option BINDER_OPTIONS <binder options>
```

Category: Language Options

Description: GNAT Only. Command-line options to be used when invoking the binder.

COMPILATION_SYSTEM

```
clicast -lada Option COMPILATION_SYSTEM <Ada compiler name>
```

Category: Language Options

Description: Ada compiler to be used to compile and link the test harness.

COMPILER_OPTIONS

```
clicast -lada Option COMPILER_OPTIONS <compiler options>
```

Category: Language Options

Description: Command-line options to be used when invoking the compiler.

COMREADER_BAUD

```
clicast -lada Option COMREADER_BAUD <baud rate>
```

Category: Target Options

Description: Serial port baud rate.

COMREADER_COMPORT

```
clicast -lada Option COMREADER_COMPORT <comport name>
```

Category: Target Options

Description: Name of serial port (e.g. 'COM1').

COMREADER_DATA_BITS

```
clicast -lada Option COMREADER_DATA_BITS <number of data bits>
```

Category: Target Options

Description: Number of data bits in communication protocol.

COMREADER_ENABLED

```
clicast -lada Option COMREADER_ENABLED True | False
```

Category: Target Options

Description: Use utility to read data from serial port.

COMREADER_PARITY

```
clicast -lada Option COMREADER_PARITY Y | N
```

Category: Target Options

Description: Communication protocol uses parity bits.

COMREADER_STOP_BITS

```
clicast -lada Option COMREADER_STOP_BITS number of stop bits
```

Category: Target Options

Description: Number of stop bits in communication protocol.

COVERAGE_ANIMATION_PLAY_SPEED_RATIO

```
clicast -lada Option COVERAGE_ANIMATION_PLAY_SPEED_RATIO <floating point
number>
```

Category: Coverage Options

Description: By default, coverage animation proceeds at the rate of one line per second. To go faster, increase the value. To go slower, decrease the value.

COVERAGE_IO_TYPE

```
clicast -lada Option COVERAGE_IO_TYPE Real_Time | Buffered | Animation
```

Category: Coverage Options

Description: Type of coverage I/O that the instrumented harness uses to record coverage data.

- Real time coverage I/O gathers coverage data and writes it to the results file immediately.
- Buffered coverage I/O gathers coverage data and buffers it in memory until the end of the test case, or on test harness termination if the option VCAST_DUMP_COVERAGE_AT_EXIT is set to TRUE. For Cover environments, buffered I/O gathers coverage data and buffers it in memory until VCAST_DUMP_COVERAGE_DATA() is called. If the option VCAST_DUMP_COVERAGE_AT_EXIT is set to TRUE then the data is instead written to the results file when the application terminates and no call to VCAST_DUMP_COVERAGE_DATA() is needed.
- Animation coverage I/O writes data to the results file in the order it was encountered, and does not remove duplicates. Animation coverage I/O is required for Basis Path coverage, and supports replaying the order of execution of the code in the Coverage Viewer.

COVERAGE_TARGET

```
clicast -lada Option COVERAGE_TARGET <integer number>
```

Category: Coverage Options

Description: Coverage is considered to Fail if it doesn't meet specified percentage.

COVERAGE_TYPE

```
clicast -lada Option COVERAGE_TYPE None | STATEMENT+MC/DC |
STATEMENT+BRANCH | MC/DC | BASIS_PATHS | BRANCH | STATEMENT
```

Category: Builder Options

Description: Type of coverage instrumentation to perform immediately after building environment. By setting this option to a value other than None coverage will be initialized each time you build a new environment.

EVENT_LIMIT

```
clicast -lada Option EVENT_LIMIT <integer number>
```

Category: Execute Options

Description: Maximum number of events for the harness to process. The harness will abort execution when this limit is exceeded. This is useful for testing code with infinite loops. If this limit is 0, no results data will be captured (coverage data will still be captured).

EXECUTABLE_EXTENSION

```
clicast -lada Option EXECUTABLE_EXTENSION <extension>
```

Category: Language Options

Description: File extension of executables created by VectorCAST.

EXECUTE ALL

```
clicast -e <env> EXecute All [<report filename>]
```

Category: Execute Commands

Description: This command executes all test cases in the specified environment. Regardless of the pass or fail status of the test execution, if a report file name is given, then additionally a Testcase Management Report is generated with the given filename. The exit status from the 'execute all' command is zero if all tests pass and non-zero otherwise.

FLOATING_POINT_TOLERANCE

```
clicast -lada Option FLOATING_POINT_TOLERANCE <floating point number>
```

Category: Execute Options

Description: Percentage that a floating point value can vary from its expected value and still be considered a match.

GLOBALS_DISPLAY

```
clicast -lada Option GLOBALS_DISPLAY Each_event | Range_Iteration | Slot_
Iteration | Testcase
```

Category: Report Options

Description: When to display / check global values in execution results: at each event; at the end of each range iteration; at the end of each slot iteration; at the end of each slot.

GNATCHOP_OPTIONS

```
clicast -lada Option GNATCHOP_OPTIONS <gnatchop options>
```

Category: Language Options

Description: GNAT Only. Command-line options to be used when invoking 'gnatchop'.

GNATMAKE_OPTIONS

```
clicast -lada Option GNATMAKE_OPTIONS <'gnatmake' options>
```

Category: Language Options

Description: GNAT Only. Switches used to control behavior of gnatmake (such as '-j8' to use multiple processes).

GPR_OPTIONS

```
clicast -lada Option GPR_OPTIONS <GNAT Project File options>
```

Category: Language Options

Description: GNAT Only. Switches used to control behavior of GPR file (such as '-Xname=value' for GPR external variables).

LIBRARY_OPTIONS

```
clicast -lada Option LIBRARY_OPTIONS <library options>
```

Category: Language Options

Description: Command-line options to be used when creating the test Ada library.

LINES_PER_PAGE

```
clicast -lada Option LINES_PER_PAGE <integer number>
```

Category: Report Options

Description: Number of lines between form-feeds in extracted reports. A value of 0 indicates no pagination will take place.

LINKER_OPTIONS

```
clicast -lada Option LINKER_OPTIONS <linker options>
```

Category: Language Options

Description: Command-line options to be used when invoking the linker.

MANAGE_ENVIRONMENT_VARIABLE

```
clicast -lada Option MANAGE_ENVIRONMENT_VARIABLE
```

Category: Language Options

Description: Environment variables used when building, executing, importing, or executing python scripts.

MANAGE_PARENT_LIB

```
clicast -lada Option MANAGE_PARENT_LIB
```

Category: Language Options

Description: Ada parent lib directory.

MANAGE_ROOT_SOURCE_DIR

```
clicast -lada Option MANAGE_ROOT_SOURCE_DIR <root source directory>
```

Category: Language Options

Description: The common root directory of environment source files.

MAX_VARY_RANGE

```
clicast -lada Option MAX_VARY_RANGE <integer number>
```

Category: Target Options

Description: Maximum number of scalars that can be varied in one test case. Changes to this value will take effect after the environment is rebuilt. Reducing this value to 0 will completely remove list and range processing from the test harness, significantly reducing the size of the harness.

METRICS_TOTAL_LEVEL_DISPLAY

```
clicast -lada Option METRICS_TOTAL_LEVEL_DISPLAY No_Filtering | Subprogram_
Detail | Unit_Detail
```

Category: Report Options

Description: This option allows you to specify which results to filter out.

NO_DELAY_STATEMENTS

```
clicast -lada Option NO_DELAY_STATEMENTS True | False
```

Category: Target Options

Description: If this option is set, the harness will be built with no 'delay' statements (typically used for allowing tests with tasks to run more smoothly).

NUMBER_OF_DATA_PARTITIONS

```
clicast -lada Option NUMBER_OF_DATA_PARTITIONS <positive number>
```

Category: Execute Options

Description: Number of iterations to span an entire range of a scalar parameter. Used with Range Values to allow specification of iterations instead of range delta.

PAGE_WIDTH

```
clicast -lada Option PAGE_WIDTH <integer number>
```

Category: Report Options

Description: Number of columns to use when building reports.

PREPARE_COMPILED_IN_DATA

```
clicast -lada Option PREPARE_COMPILED_IN_DATA <compiled-in data preparation
script>
```

Category: Target Options

Description: Path to your custom script, batch file, or other mechanism that will read the standard input file generated by the tool and generate source code to populate the file input object in the test harness. This script is executed every time the harness is run; it is immediately followed by a compile of the input object and a link of the executable.

PREPROCESSOR_OPTIONS

```
clicast -lada Option PREPROCESSOR_OPTIONS <preprocessor options>
```

Category: Language Options

Description: Command-line options to be used when invoking the Ada preprocessor (only available for some compilers).

RANGE_CHECK

```
clicast -lada Option RANGE_CHECK Full | Disable | None
```

Category: Execute Options

Description: Indicates whether range checking should be performed for input and expected test case values. 'All', means that type ranges will be enforced, and out-of-range test data will not be accepted. 'Disable' allows out-of-range test data values to be used. 'None' disables all range processing. This is useful when target communication is slow.

REPORT_OBJECTS

```
clicast -lada Option REPORT_OBJECTS Never | Always
```

Category: Report Options

Description: When preparing execution reports, if this option is set, VectorCAST will display ALL global objects. If the option is not set, then only global objects that are part of the test case (as input or expected values) will be displayed.

REPORT_PARAMETERS

```
clicast -lada Option REPORT_PARAMETERS Never | Always
```

Category: Report Options

Description: When preparing execution reports, if this option is set, VectorCAST will display ALL parameters for each test event. If the option is not set, then only parameters that are part of the test case (as input or expected values) will be displayed.

SHOW_NOTES_IN_FULL_REPORT

```
clicast -lada Option SHOW_NOTES_IN_FULL_REPORT True | False
```

Category: Report Options

Description: This option will add a section to the Full Report listing test notes and requirements.

STANDARD_ERROR

```
clicast -lada Option STANDARD_ERROR Normal | Redirect
```

Category: Execute Options

Description: If this option is set, any standard error from the test will be captured to a file for inclusion in the test execution report.

STANDARD_OUTPUT

```
clicast -lada Option STANDARD_OUTPUT Normal | Redirect
```

Category: Execute Options

Description: If this option is set, any standard output from the test will be captured to a file for inclusion in the test execution report.

STUB_DEPENDENCIES

```
clicast -lada Option STUB_DEPENDENCIES Yes | No | Custom
```

Category: Builder Options

Description: Default setting for Stub Dependencies option in the environment builder. ALL means stub all dependencies. NONE means do not stub any dependencies.

TARGET_AE653_MAKE_FILE

```
clicast -lada Option TARGET_AE653_MAKE_FILE <directory containing the
Makefile>
```

Category: Target Options

Description: This option provides the command used to perform a 'make' for the vxWorks 653 target.

TARGET_APEX_LIBRARY

```
clicast -lada Option TARGET_APEX_LIBRARY full | mark
```

Category: Target Options

Description: For Apex Exec. This option tells VectorCAST which ada library to use. If this option is set to Full, the full Ada run-time library will be used. If this option is set to MARK, the reduced run-time will be used and the test harness will be built with the appropriate restrictions.

TARGET_BOARD_NAME

```
clicast -lada Option TARGET_BOARD_NAME <target name>
```

Category: Target Options

Description: Name of target board. For vxWorks this is the name of the target server, for other targets it is the hostname or IP address of the target.

TARGET_BOOT_HOSTNAME

```
clicast -lada Option TARGET_BOOT_HOSTNAME <host name>
```

Category: Target Options

Description: For vxWorks. This is the name that the target knows the boot host as. If this option is not set, VectorCAST assumes that the hostname of the machine it is running on, is the boot host.

TARGET_BUILD_FILE

```
clicast -lada Option TARGET_BUILD_FILE <filename>
```

Category: Target Options

Description: For Green Hills Compilers only. This is the name of the default build file. The '.bld' extension should NOT be used (e.g. ppc, sim800). Builder flags can also be provided in this option (e.g. :language=ada).

TARGET_COMMAND_OPTIONS

```
clicast -lada Option TARGET_COMMAND_OPTIONS <target command options>
```

Category: Target Options

Description: Options used when loading the target.

TARGET_COMMAND_VERB

```
clicast -lada Option TARGET_COMMAND_VERB <download command>
```

Category: Target Options

Description: Command prefix, used to invoke the compiler.

TARGET_CPU

```
clicast -lada Option TARGET_CPU Ppc603
```

Category: Target Options

Description: Used for SingleStep debugger. The name of the 'CPU' to be passed to the 'debug' command.

TARGET_INTEGRITY_ADDRESS_SPACE_MODE

```
clicast -lada Option TARGET_INTEGRITY_ADDRESS_SPACE_MODE kernel | virtual
```

Category: Target Options

Description: For Green Hills Integrity. This option tells VectorCAST how to link the application. If this option is set to Kernel, the harness will be linked with the OS as a kernel space application. If the option is set to Virtual, a virtual address space application will be created.

TARGET_INTEGRITY_LIBRARY

```
clicast -lada Option TARGET_INTEGRITY_LIBRARY full | gmart | gstart
```

Category: Target Options

Description: For Green Hills Integrity. This option tells VectorCAST which ada library to use. If this option is set to Full, the full Ada run-time library will be used. If this option is set to GMART, or GSTART, the reduced run-time will be used and the test harness will be built with the appropriate restrictions.

TARGET_IO_DIRECTORY

```
clicast -lada Option TARGET_IO_DIRECTORY <directory>
```

Category: Target Options

Description: Directory where Input/Output files for target test execution will be stored. This option should be used when the target does not have read/write permission for the test environment directory.

TARGET_SIM_CMD

```
clicast -lada Option TARGET_SIM_CMD <simulator command>
```

Category: Target Options

Description: Command used to execute on the target or simulator.

TARGET_SUPPORTS_TIC_IMAGE

```
clicast -lada Option TARGET_SUPPORTS_TIC_IMAGE True | False
```

Category: Target Options

Description: Set this option to True when the target run-time does not support the 'image by default, but the run-time being used has been modified to support 'image.

TARGET_TFTP_DIR

```
clicast -lada Option TARGET_TFTP_DIR <directory>
```

Category: Target Options

Description: This is the directory that the executable files will be copied to so that the target can be booted via tftp.

TARGET_USE_RTSERV_COMM

```
clicast -lada Option TARGET_USE_RTSERV_COMM True | False
```

Category: Target Options

Description: If this option is set, and you are building a Kernel application, VectorCAST will copy the executable files to the tftp directory, and then prompt you to manually boot the target board. This option is assumed set if you are building a Virtual Address Space applicaton

TARGET_USE_TORNSERV

```
clicast -lada Option TARGET_USE_TORNSERV True | False
```

Category: Target Options

Description: Tornado command is 'tornserv' instead of the default 'torn2serv'.

TARGET_VARIANT

```
clicast -lada Option TARGET_VARIANT <target code>
```

Category: Target Options

Description: This option describes the compiler / target combination used to build an environment.

TARGET_XFER_METHOD

```
clicast -lada Option TARGET_XFER_METHOD Serial | Ethernet
```

Category: Target Options

Description: Method of target data transfer. This option controls the value passed to the Aonix transfer command when a test is run.

TEST_CASE_TIMEOUT

```
clicast -lada Option TEST_CASE_TIMEOUT <integer number>
```

Category: Target Options

Description: Option TEST_CASE_TIMEOUT <integer number>. Maximum amount of time, in seconds, to wait before VectorCAST terminates a test execution. The default value is 0 which means wait 'forever'.

UNCON_ARRAY_SIZE

```
clicast -lada Option UNCON_ARRAY_SIZE <integer number> | <integer
number>:<integer number>
```

Category: Builder Options

Description: Default size for unconstrained arrays where the index range is unknown or very large. May be specified as [lower bound]:[upper bound] or just [upper bound].

VCAST_ADA_ACCESSOR_FUNCTIONS

```
clicast -lada Option VCAST_ADA_ACCESSOR_FUNCTIONS True | False
```

Category: Target Options

Description: If this option is set, VectorCAST will access Ada options from accessor functions. If this is set to false, VectorCAST will access Ada options from global objects.

VCAST_ADA_FILE_EXTENSIONS

```
clicast -lada Option VCAST_ADA_FILE_EXTENSIONS <ada file extensions>
```

Category: Language Options

Description: Use this option to specify the ada file extensions.

VCAST_ADA_LANGUAGE_SPECIFICATION

```
clicast -lada Option VCAST_ADA_LANGUAGE_SPECIFICATION <ADA_83 | ADA_95 |
ADA_05 | ADA_12>
```

Category: Builder Options

Description: Language specification to which the compiler conforms.

VCAST_ADAOPTS

```
clicast -lada Option VCAST_ADAOPTS True | False
```

Category: Builder Options

Description: Set this option to tell VectorCAST/Ada that you are using a version of the ObjectAda compiler that is newer than version 7.2. If 'adaopts -l -u' reports an error, this flag needs to be set.

VCAST_ADD_DONT_STUB_UNITS

```
clicast -lada Option VCAST_ADD_DONT_STUB_UNITS True | False
```

Category: Builder Options

Description: This option will force the environment builder to add units specified as DONT_STUB in the environment script into the environment database. This is especially useful to get custom coverage on units that have otherwise been ignored.

VCAST_ALLOW_STUB_OF_STD_LIB

```
clicast -lada Option VCAST_ALLOW_STUB_OF_STD_LIB True | False
```

Category: Builder Options

Description: Set this option if you want to enable stubbing of the standard Ada packages (VCAST_IO, SYSTEM, etc).

VCAST_APEX_CLEARCASE

```
clicast -lada Option VCAST_APEX_CLEARCASE True | False
```

Category: Builder Options

Description: This option tells VectorCAST that Apex is being used with the ClearCase CM tool rather

than the default Summit CM Tool. In ClearCase mode, the structure of the Apex views and sub-systems is quite different. This option allows VectorCAST to accommodate those differences.

VCAST_APEX_INSTALLATION_DIR

```
clicast -lada Option VCAST_APEX_INSTALLATION_DIR <path search string>
```

Category: Builder Options

Description: Comma-separated list of strings to search for in paths to imported subsystems that identify standard ada libraries that VectorCAST will not parse/stub. Default (unspecified) value for this option is 'base/ada'.

VCAST_APEX_MUTUAL_IMPORTS

```
clicast -lada Option VCAST_APEX_MUTUAL_IMPORTS True | False
```

Category: Builder Options

Description: When this option is set, VectorCAST will process source files found in the mutual imports of the parent view.

VCAST_APPEND_TO_TESTINSS

```
clicast -lada Option VCAST_APPEND_TO_TESTINSS True | False
```

Category: Coverage Options

Description: This option tells VectorCAST to open the coverage data file for appending, rather than always creating the data file.

VCAST_CMD_AFTER_INSTRUMENTATION

```
clicast -lada Option VCAST_CMD_AFTER_INSTRUMENTATION <cmd>
```

Category: Coverage Options

Description: This command will be run after VectorCAST has instrumented a file.

VCAST_CMD_DURING_ENV_CLOSE

```
clicast -lada Option VCAST_CMD_DURING_ENV_CLOSE <cmd>
```

Category: Target Options

Description: Command to be executed while closing a unit test environment, where **<cmd>** is an executable application or script to be executed in the environment directory. The command is executed just as the environment begins to close. If running a script, the interpreter should be specified.

Note that when using clicast, the user should be aware that every command taking the argument **-e <env>** opens and closes the environment. Therefore, the options can be set immediately before calling the clicast command of interest, and unset afterwards.

VCAST_CMD_DURING_ENV_OPEN

```
clicast -lada Option VCAST_CMD_DURING_ENV_OPEN <cmd>
```

Category: Target Options

Description: Command to be executed while opening a unit test environment, where **<cmd>** is an executable application or script to be executed in the environment directory. The command is executed right before building the range data during environment open. If running a script, the interpreter should be specified.

Note that when using clicast, the user should be aware that every command taking the argument **-e <env>** opens and closes the environment. Therefore, the options can be set immediately before calling the clicast command of interest, and unset afterwards.

VCAST_COMMAND_LINE_DEBUGGER

```
clicast -lada Option VCAST_COMMAND_LINE_DEBUGGER True | False
```

Category: Language Options

Description: This option causes VectorCAST to bring up a shell window to run the debugger.

VCAST_COMPACT

```
clicast -lada Option VCAST_COMPACT True | False
```

Category: Report Options

Description: This option causes VectorCAST to compress the execution results for scalar array elements that are the same. If a five element array has all values = 0, this option will enable the report to have one output line of (1..5) rather than 5 lines of one value per line.

VCAST_COMPILE_DUMB_STUBS

```
clicast -lada Option VCAST_COMPILE_DUMB_STUBS True | False
```

Category: Builder Options

Description: When a dependent unit is stubbed, VectorCAST creates a 'dumb' stub for any unit 'with'ed by the stubbed units specification. By setting this option, these 'dumb' bodies will be compiled and linked into the harness, allowing the execution closure to be completed with less effort.

VCAST_COMPILE_GENERIC_STUBS

```
clicast -lada Option VCAST_COMPILE_GENERIC_STUBS True | False
```

Category: Builder Options

Description: If you set this option, VectorCAST/Ada will compile stubs that are created for generic packages.

VCAST_CONFIGURABLE_ADA_IO

```
clicast -lada Option VCAST_CONFIGURABLE_ADA_IO True | False
```

Category: Target Options

Description: VectorCAST will generate VCAST_STDIN.DAT containing all input data for harness, and assume all output data will be written to standard output.

VCAST_CONSOLE_REPORT_FORMAT

```
clicast -lada VCAST_CONSOLE_REPORT_FORMAT CONSOLE_REPORT_DEFAULT | CONSOLE_
REPORT_LEGACY | CONSOLE_REPORT_VERBOSE | CONSOLE_REPORT_QUIET
```

Category: Report Options

Description: This clicast-only option by default (CONSOLE_REPORT_DEFAULT), shows command line output with passes and failures at the iteration level, and Expected/Actual value mismatch at the iteration level. After all tests have run, a test summary is shown along with a list of all failing tests. With Legacy output (CONSOLE_REPORT_LEGACY), the previous style of console output is shown. Verbose output (CONSOLE_REPORT_VERBOSE) prints both Default and Legacy Output. When set to quiet (CONSOLE_REPORT_QUIET), minimal test/run information is shown in the console output.

VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII

```
clicast -lada Option VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII True | False
```

Category: Report Options

Description: This option is used to determine whether or not to convert "unprintable" strings encoded in octal or hexidecimal notation to ASCII encoding in the report output. It does not affect the actual string comparisons in the test execution.

VCAST_COVER_SEPARATE_TYPES_FILES

```
clicast -lada Option VCAST_COVER_SEPARATE_TYPES_FILES <ada file extensions>
```

Category: Coverage Options

Description: Put unit coverage utilities in separate file.

VCAST_COVERAGE_DATABASE_CACHE_SIZE

```
clicast -lada Option VCAST_COVERAGE_DATABASE_CACHE_SIZE <integer number>
```

Category: Coverage Options

Description: Increasing the maximum coverage database cache size may increase performance, but take care not to set it to a value greater than the amount of available system memory. For a system with 2GB, a 1000 MB maximum cache is probably sufficient. Changes to this option take effect when re-opening an environment.

VCAST_COVERAGE_DATABASE_SIMPLIFIED_LOCKING

```
clicast -lada Option VCAST_COVERAGE_DATABASE_SIMPLIFIED_LOCKING True |
False
```

Category: Coverage Options

Description: Certain file system configurations, especially some using NFS, do not have a properly functioning POSIX locking implementation. Enabling this option makes the database layer use an alternate, simpler locking method which should allow the database to function on any file system. However, when enabled, only one user or instance of VectorCAST will be able to access the database at a time.

VCAST_COVERAGE_FIELD_WIDTH

```
clicast -lada Option VCAST_COVERAGE_FIELD_WIDTH <integer number>
```

Category: Report Options

Description: The width (in characters) of the left margin of the coverage viewer for C/C++ files instrumented prior to VectorCAST 2020 and all Ada files. Increase this number if you have a large number of subprograms or a subprogram has a large number of statements or branches.

VCAST_CUSTOM_REPORT_FORMAT

```
clicast -lada Option VCAST_CUSTOM_REPORT_FORMAT HTML | TEXT
```

Category: Report Options

Description: Output format for VectorCAST reports: HTML or text.

VCAST_DEFAULT_SCRIPT_HEADER

```
clicast -lada Option VCAST_DEFAULT_SCRIPT_HEADER <Path to default header
text file>
```

Category: Report Options

Description: This option allows you to enter a path to a text file that contains a custom header for all generated test scripts.

VCAST_DETECT_UNUSED_EXPECTED_UC

```
clicast -lada Option VCAST_DETECT_UNUSED_EXPECTED_UC True | False
```

Category: Execute Options

Description: When enabled, this option detects test-specific expected user code that has not been executed during a test. Such code is most likely to be associated with a stub that has not been called. When True and some testcase or parameter user code has not been executed, the test fails, and the Execution Report displays the unused values, the unit, and whether that unit is stubbed or not-stubbed.

Disable this option if you have added parameter user code to a stub that you expect to never get called. (You might add code to an uncalled stub if you want the test to show a failure if the stub ever gets called.)

VCAST_DISABLE_AUTO_BASIS_PATH_GEN

```
clicast -lada Option VCAST_DISABLE_AUTO_BASIS_PATH_GEN True | False
```

Category: Coverage Options

Description: When this option is set, VectorCAST will not generate Basis Path information unless specifically requested by the user.

VCAST_DISABLE_BOOLEAN_CAST

```
clicast -lada Option VCAST_DISABLE_BOOLEAN_CAST True | False
```

Category: Coverage Options

Description: Select this option to prevent VectorCAST from casting conditional expressions to Boolean when instrumenting for branch or MC/DC coverage.

VCAST_DISABLE_DIRECT_ARRAY_INDEXING

```
clicast -lada Option VCAST_DISABLE_DIRECT_ARRAY_INDEXING True | False
```

Category: Builder Options

Description: This option is used to allow VectorCAST to know how to display the subscripts of an array in the Parameter Tree and in the test script. For example, if you have an array subscripted from -10 to 10, if this option is set, VectorCAST displays the subscripts from 1 to 21. If the option is not set (default), VectorCAST displays the subscripts from -10 to 10.

VCAST_DISABLE_TEST_MAINTENANCE

```
clicast -lada Option VCAST_DISABLE_TEST_MAINTENANCE True | False
```

Category: Builder Options

Description: This clicast-only option enables Test Script Maintenance by default and will try to update renamed parameters, subprograms, and other parts of test scripts to reflect changes in source code when importing test scripts. The default value is False. To disable the utility from running set the option to True.

VCAST_DISPLAY_EMPTY_COVERAGE

```
clicast -lada Option VCAST_DISPLAY_EMPTY_COVERAGE <True | False>
```

Category: Report Options

Description: Display annotated source code in reports even if no run-time coverage data exists. If this option is disabled, the message "No Coverage Data Exists" will be displayed instead.

VCAST_DISPLAY_FUNCTION_COVERAGE

```
clicast -lada Option VCAST_DISPLAY_FUNCTION_COVERAGE <True | False>
```

Category: Report Options

Description: Extra column that reports if a function was entered. This option is only used if the

coverage type is not Function.

VCAST_DO_COMBINATION

```
clicast -lada Option VCAST_DO_COMBINATION True | False
```

Category: Execute Options

Description: When this option is on, and multiple parameters have a range of values, all combinations of the range values will be used as input data. (e.g. Two parameters being varied from 1..5 will result in 25 calls to the UUT) If this option is grey, it means that the open environment does not support combination testing. Rebuild the environment to use this feature.

VCAST_DONT_STUB_ANCESTORS

```
clicast -lada Option VCAST_DONT_STUB_ANCESTORS True | False
```

Category: Builder Options

Description: If this option is set, VectorCAST will not prompt for stubbing ancestor packages of the Unit Under Test. Also, VectorCAST will add any sub-programs defined in the ancestors to the sub-program list of the Unit Under Test.

VCAST_EMPTY_TESTCASE_FAIL

```
clicast -lada Option VCAST_EMPTY_TESTCASE_FAIL True | False
```

Category: Execute Options

Description: If this option is set, then empty test cases will be marked as failed.

VCAST_ENHANCED_FOR_LOOP_COVERAGE

```
clicast -lada Option VCAST_ENHANCED_FOR_LOOP_COVERAGE True | False
```

Category: Coverage Options

Description: When this option is true, coverage for Ada 'for' statements will show both True if the loop condition has ever evaluated as True, and False if it has ever evaluated as False. When this option is not set, only the True condition is checked.

VCAST_ENVIRONMENT_FILES

```
clicast -lada Option VCAST_ENVIRONMENT_FILES <files>
```

Category: Target Options

Description: These files will be copied into the environment directory during an environment build or rebuild.

VCAST_EXECUTE_WITH_STDOUT

```
clicast -lada Option VCAST_EXECUTE_WITH_STDOUT
```

Category: Target Options

Description: This option tells VectorCAST to compile input test data into the harness, and map standard output to VCAST_STDOUT.DAT. The normal Execute Command is still used to execute the test.

VCAST_EXPECTED_BEFORE_UUT_CALL

```
clicast -lada Option VCAST_EXPECTED_BEFORE_UUT_CALL True | False
```

Category: Report Options

Description: If a stub is called before the first UUT call, allow comparisons of expected values during the stub execution.

VCAST_EXTENDED_EXCEPTION_INFORMATION

```
clicast -lada Option VCAST_EXTENDED_EXCEPTION_INFORMATION True | False
```

Category: Builder Options

Description: When an exception is propagated out of a UUT call, setting this flag to TRUE will cause ADA.EXCEPTIONS.EXCEPTION_MESSAGE to be called.

VCAST_FIELD_DOT_NOTATION

```
clicast -lada Option VCAST_FIELD_DOT_NOTATION True | False
```

Category: Report Options

Description: This option is used to tell VectorCAST's execution results and test case data listing reports to use field dot notation whenever a field name is printed.

VCAST_FILE_ENCODING

```
clicast -lada Option VCAST_FILE_ENCODING <encoding format>
```

Category: Builder Options

Description: Indicates what file encoding is used for source files being processed by VectorCAST. <encoding format> is one of the following:

WCEM=h for Hex ESC encoding

WCEM=u for Upper half encoding

WCEM=s for Shift-JIS encoding

WCEM=e for EUC encoding

WCEM=8 for UTF-8 encoding

WCEM=b for Brackets encoding

The default is empty, which means "Hex ESC encoding".

VCAST_FILE_INDEX

```
clicast -lada Option VCAST_FILE_INDEX True | False
```

Category: Target Options

Description: Select this option to enable VectorCAST to use a file indexing mode that significantly reduces file I/O required for running coverage or unit tests.

VCAST_FILE_VERSION_COMMAND

```
clicast -lada Option VCAST_FILE_VERSION_COMMAND <command>
```

Category: Report Options

Description: This command will be executed for each source file in the test environment, and a section will be added to the Full report and Test Case Management report listing the name of each unit in the test environment and the information generated by this command. If this command is empty, then the File Version section of the report will not be created.

VCAST_FLOAT_PRECISION

```
clicast -lada Option VCAST_FLOAT_PRECISION <integer number>
```

Category: Report Options

Description: Controls the number of digits used when the test harness prints floating point Actual Values for the Execution Report or Range Data. This option's setting is significant in the comparison

of Expected Values for floating point numbers as well as determining the min and max values.

In an Ada environment, floating point values are printed in scientific notation and the precision value controls the number of digits to the right of the decimal, with a single digit to the left of the decimal. A precision of 0 results in the maximum avaliable digits being printed.

VCAST_FORCE_REPORT_DATES

```
clicast -lada Option VCAST_FORCE_REPORT_DATES True | False
```

Category: Report Options

Description: This option will force all the dates and times in the report configuration table and the management table to all be the same. This helps diff the reports much more easily.

VCAST_GEN_ALL_EXECUTION_REPORT_FORMATS

```
clicast -lada Option VCAST_GEN_ALL_EXECUTION_REPORT_FORMATS True | False
```

Category: Report Options

Description: This option is used to determine whether or not to generate the execution reports in all the available formats (e.g. HTML or TEXT).

VCAST_GH_INT_FILE

```
clicast –lada Option VCAST_GH_INT_FILE <intex filename>
```

Category: Target Options

Description: This is the custom integrate file passed to the Green Hills 'intex' command. This file should follow the general format of the default file found in the VectorCAST installation directory, which means it should contain a 'Filename' line with the text VCAST_FILE (to be replaced with the VectorCAST executable name) and a 'Starit' line with the value 'true'.

VCAST_GH_INTEX_CMD

```
clicast -lada Option VCAST_GH_INTEX_CMD <intex command>
```

Category: Target Options

Description: When set, VectorCAST will invoke the Green Hills intex utility with this command immediately after linking the test harness. Refer to the VCAST_GH_INT_FILE environment variable for information on how to specify a custom integrate file. Example: intex -kernel=C:\GHS\int408\sim800\kernel -bspdir=C:\GHS\int408\sim800 -

target=C:\GHS\int408\sim800\default.bsp vcast.int.

VCAST_GH_PP_COMMAND

```
clicast -lada Option VCAST_GH_PP_COMMAND <full path to ada preprocessor
command with args>
```

Category: Language Options

Description: This option allows you to assign a preprocessor for VectorCAST to use on ADA files when using the Green Hills compiler.

VCAST_GMART_FILE_IO

```
clicast -lada Option VCAST_GMART_FILE_IO True | False
```

Category: Target Options

Description: If this option is set, VectorCAST will use File I/O to communicate with the target. If it is not set, asynchronous monitor communication will be used.

VCAST_GNAT_INCLUDE_PATH

```
clicast -lada Option VCAST_GNAT_INCLUDE_PATH <directory path>
```

Category: Language Options

Description: GNAT Only. This value will override the ADA_INCLUDE_PATH environment variable used by GNAT to find source files.

VCAST_GNAT_OBJECTS_PATH

```
clicast -lada Option VCAST_GNAT_OBJECTS_PATH <directory path>
```

Category: Language Options

Description: GNAT Only. This value will override the ADA_OBJECTS_PATH environment variable used by GNAT to find source files.

VCAST_GPR_TEMPLATE

```
clicast -lada Option VCAST_GPR_TEMPLATE <gpr file template>
```

Category: Language Options

Description: This file will be used by VectorCAST to make a local GPR build structure to be used as the parent for the VectorCAST environment.

VCAST_GPR_USES_NAMING

```
clicast -lada Option VCAST_GPR_USES_NAMING True | False
```

Category: Language Options

Description: The GNAT Project file uses 'package NAMING' to override GNAT's default file naming conventions.

VCAST_IGNORE_INCOMPLETE_TESTS

```
clicast -lada Option VCAST_IGNORE_INCOMPLETE_TESTS True | False
```

Category: Execute Options

Description: When building Basis Path or MC/DC test cases there are three outcomes for each test: Complete, Partial, and Template. When this option is on, VectorCAST will discard the Partial and Template tests, and only load the Complete tests.

VCAST_IMPORT_FOREIGN_OBJECTS

```
clicast -lada Option VCAST_IMPORT_FOREIGN_OBJECTS True | False
```

Category: Builder Options

Description: When the harness needs to create an object at a specific memory location with the 'use at <address>' construct, if this option is set to TRUE then a 'pragma import' command will be used to control compiler initialization of the object.

VCAST_INSTRUMENT_ASSIGNMENTS

```
clicast -lada Option VCAST_INSTRUMENT_ASSIGNMENTS True | False
```

Category: Coverage Options

Description: Setting this option to TRUE will cause assignment statements with boolean or comparison operators to be covered by branch and MC/DC coverage. If this option is TRUE, MC/DC will be suppressed for statements preceded by the comment: VCAST_DONT_DO_MCDC. If this option is FALSE, MC/DC will still be done on assignment statements preceded by the comment: VCAST_DO_MCDC.

VCAST_INSTRUMENT_PARAMETERS

```
clicast -lada Option VCAST_INSTRUMENT_PARAMETERS True | False
```

Category: Coverage Options

Description: Setting this option will cause parameters in function calls to be covered by branch and MC/DC coverage.

VCAST_MAX_COVERED_SUBPROGRAMS

```
clicast -lada Option VCAST_MAX_COVERED_SUBPROGRAMS <integer number>
```

Category: Coverage Options

Description: If 'Use static memory allocation' is True and Coverage I/O is Buffered, VectorCAST must allocate a specific amount of space to store coverage data each time a different subprogram call is encountered. This number tells VectorCAST how many different subprogram calls for which it should set aside memory.

VCAST_MAX_MCDC_ABORT_INST

```
clicast -lada Option VCAST_MAX_MCDC_ABORT_INST True | False
```

Category: Coverage Options

Description: While instrumenting with MC/DC, if a conditional statement contains more conditions than possible to instrument, then abort instrumenting that file. For Ada, the maximum is 26.

VCAST_MAX_MCDC_CONDITIONS

```
clicast -lada Option VCAST_MAX_MCDC_CONDITIONS <integer number>
```

Category: Coverage Options

Description: VectorCAST generates equivalence matrices for up to this many subconditions in an MC/DC expression. (Example: (A || !B || C) has operands, so it has three subconditions.) If an expression exceeds this limit, equivalence pairs are calculated as results are added to the environment, and only table rows that contribute to a satisfied pair are displayed. MC/DC tests are not created for expressions exceeding the limit. Its default value is 8.

.

VCAST_MAX_MCDC_STATEMENTS

```
clicast -lada Option VCAST_MAX_MCDC_STATEMENTS <integer number>
```

Category: Coverage Options

Description: If 'Use static memory allocation' is True, VectorCAST must allocate a specific amount of space to store coverage data each time an MC/DC expression is encountered. This number tells VectorCAST how many MC/DC expressions for which it should set aside memory.

VCAST_MAX_STRING_LENGTH

```
clicast -lada Option VCAST_MAX_STRING_LENGTH <integer number>
```

Category: Target Options

Description: This specifies the size of temporary character arrays that VectorCAST creates in the test harness. Longer strings are needed specifically when test cases use long lists of values as stimulus data. For example, a stimulus value for the sine function of: 10,20,30,40,50,60 ... If you are running in a target environment with limited heap and stack resources, making this value smaller will reduce the VectorCAST test harness use of heap and stack. Changes to this value will take effect after the environment is recompiled.

VCAST_MAX_TABLE_SUBCONDITIONS

```
clicast -lada Option VCAST_MAX_TABLE_SUBCONDITIONS <integer number>
```

Category: Coverage Options

Description: When VectorCAST generates equivalence matrices for an expression, if the expression has more than this many subconditions, no table information will be displayed. Pair information will still be calculated.

VCAST_MAX_TARGET_FILES

```
clicast -lada Option VCAST_MAX_TARGET_FILES <integer number>
```

Category: Target Options

Description: This option limits the total number of files that the test harness is allowed to open while running a compound test using 'Execution process uses stdin/stdout for file IO' (VCAST_ CONFIGURABLE_ADA_IO is TRUE). This option is useful for executing compound tests on a target with limited memory. If set too low, test execution for the test does not start. Environment must be recompiled for changes to take effect.

VCAST_MULTIBYTE_CHARACTERS

```
clicast -lada Option VCAST_MULTIBYTE_CHARACTERS True | False
```

Category: Builder Options

Description: Enable this clicast-only option if source code files are encoded using Shift JIS.

VCAST_NEW_ARCHITECTURE

```
clicast -lada Option VCAST_NEW_ARCHITECTURE True | False
```

Category: Builder Options

Description: Set this option to allow manipulation of private types from all units in the environment, not just the Unit Under Test.

VCAST_NO_EXCEPTION_HANDLERS

```
clicast -lada Option VCAST_NO_EXCEPTION_HANDLERS True | False
```

Category: Target Options

Description: If this option is set, the harness will be built with no exception-catching capabilities.

VCAST_NO_RECURSIVE_CALLS

```
clicast -lada Option VCAST_NO_RECURSIVE_CALLS True | False
```

Category: Builder Options

Description: By default, the VectorCAST test harness uses recursive calls to improve processing. If the 'NO_RECURSION' restriction is activated in the code under test, the harness needs to be built differently.

VCAST_NO_STANDARD_PKG_USAGE

```
clicast -lada Option VCAST_NO_STANDARD_PKG_USAGE True | False
```

Category: Builder Options

Description: By default, the Multiunit Whitebox harness redefines TRUE and FALSE by using STANDARD.BOOLEAN. This option disables that override, necessary when user source files use 'STANDARD' in a different context.

VCAST_NOTES_SECTION_TEMPLATE

```
clicast –lada Option VCAST_NOTES_SECTION_TEMPLATE <Path to Template text
file>
```

Category: Report Options

Description: This option allows you to enter a path to a text file that contains a template for the Notes section of the test case.

VCAST_OLD_STYLE_MANAGEMENT_REPORT

```
clicast -lada Option VCAST_OLD_STYLE_MANAGEMENT_REPORT True | False
```

Category: Report Options

Description: When set, this option reverts the Testcase Management Report to the behavior found in VectorCAST version 6.0 and prior, in which the main body of the report combined the Expecteds and Control Flow in one fraction in the "Pass/Fail" column.

VCAST_POST_LINK_COMMAND

```
clicast -lada Option VCAST_POST_LINK_COMMAND <command to execute>
```

Category: Builder Options

Description: Command to execute after linking executable.

VCAST_PRAGMA_IMPORT

```
clicast -lada Option VCAST_PRAGMA_IMPORT True | False
```

Category: Builder Options

Description: This option tells VectorCAST to build stubs for subprograms in stubbed units that are defined using 'pragma import'.

VCAST_PRE_EXECUTE_CMD

```
clicast -lada Option VCAST_PRE_EXECUTE_CMD <command to execute before test
execution>
```

Category: Target Options

Description: The specified executable or batch file script is executed before the test harness is called during test case execution. Useful for rebooting your target, for example. If a relative path to this command is used, it should be relative to the environment directory.

VCAST_PRECOMPILE_SCRIPT

```
clicast –lada Option VCAST_PRECOMPILE_SCRIPT <script>
```

Category: Builder Options

Description: This command is executed prior to full compilations of the test harness. It can be used to make modifications to harness files before the initial compilation. Note that this command may also be called for subsequent harness compilations, but it is not called before every compilation.

VCAST_PREPEND_TO_PATH_DIRS

```
clicast -lada Option VCAST_PREPEND_TO_PATH_DIRS <path>
```

Category: Builder Options

Description: These directories will be prepended to the PATH environment variable value. This can be used to put your compiler on the PATH. Multiple directories should be separated by ';' (Windows) or ':' (Linux). Note that some compilers require other environment variables to be set as well in order for the compiler to work properly.

VCAST_PSC_MAKE_FILE_BASE_DIRECTORY

```
clicast -lada Option VCAST_PSC_MAKE_FILE_BASE_DIRECTORY <directory path>
```

Category: Target Options

Description: This option provides the path to the vxWorks 653 build directory containing the existing partition build. This directory should contain a directory (with the same name as the Target Board option) containing the files necessary to complete an executable build.

VCAST_RAVEN_FILE_IO

```
clicast -lada Option VCAST_RAVEN_FILE_IO True | False
```

Category: Target Options

Description: Run Raven target using file I/O instead of standard output.

VCAST_RECOMPILE_SEPARATES

```
clicast -lada Option VCAST_RECOMPILE_SEPARATES True | False
```

Category: Builder Options

Description: This option tells VectorCAST that you want to recompile the body of the Unit Under Test before linking, but after all stubs are compiled. This is useful when the Unit Under Test calls 'in-lined' subprograms from stubbed units.

VCAST_REFERENCED_GLOBALS

```
clicast -lada Option VCAST_REFERENCED_GLOBALS True | False
```

Category: Execute Options

Description: When this option is on, the VectorCAST test case editor will display only those global objects and stubs that are referenced by the function under test.

VCAST_REPOSITORY

```
clicast -lada Option VCAST_REPOSITORY <directory path>
```

Category: Language Options

Description: This is the path to a directory containing a repository of settings.

VCAST_RPTS_AUTOMATIC_BGCOLOR

```
clicast -lada Option VCAST_RPTS_AUTOMATIC_BGCOLOR <color>
```

Category: Report Options

Description: This color is used for text indicating automatic initialization and finalization tests.

VCAST_RPTS_BODY_BGCOLOR

```
clicast -lada Option VCAST_RPTS_BODY_BGCOLOR <color>
```

Category: Report Options

Description: Primary background color for HTML reports.

VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of complexity columns in text reports.

VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of coverage result columns in text reports.

VCAST_RPTS_CUSTOM_CONFIG

```
clicast -lada Option VCAST_RPTS_CUSTOM_CONFIG <config text> | <config file>
```

Category: Report Options

Description: Custom report configuration. If this option points to an existing file, then the contents of that file is used as the custom report configuration, otherwise the contents of the option is used.

VCAST_RPTS_CUSTOM_CSS

```
clicast -lada Option VCAST_RPTS_CUSTOM_CSS <css>
```

Category: Report Options

Description: Reports use the default cascading style sheets (**`*.css)`** located in **`$VECTORCAST_DIR/python/vector/apps/ReportBuilder/css/`**, unless a custom style sheet is specified in this option. Environment variables in the path to the custom style sheet are supported, using this syntax **`$(ENV_VAR)`**. When used, the contents of the custom style sheet are embedded in each report, between the **`<style>...</style>`** tags in the header section after the default styles allowing for CSS style overriding.

VCAST_RPTS_DATE_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_DATE_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of date columns in text reports.

VCAST_RPTS_DEFAULT_FONT_COLOR

```
clicast -lada Option VCAST_RPTS_DEFAULT_FONT_COLOR <color>
```

Category: Report Options

Description: Default text color for HTML reports.

VCAST_RPTS_DEFAULT_FONT_FACE

```
clicast -lada Option VCAST_RPTS_DEFAULT_FONT_FACE <font>
```

Category: Report Options

Description: Default font for HTML reports.

VCAST_RPTS_DEFAULT_FONT_SIZE

```
clicast -lada Option VCAST_RPTS_DEFAULT_FONT_SIZE <font size>
```

Category: Report Options

Description: Default font size for HTML reports.

VCAST_RPTS_EMPTY_DATA_STRING

```
clicast -lada Option VCAST_RPTS_EMPTY_DATA_STRING <string>
```

Category: Report Options

Description: Text to show in blank cells in the Execution and Test Case Data report sections. The default value is unset.

VCAST_RPTS_DELIMITER

```
clicast -lada Option VCAST_RPTS_DELIMITER <delimiter>
```

Category: Report Options

Description: Character separator used in two CLICAST report commands: cover report cvs_metrics and reports alternate. To enter a tab, use \t.

Valid delimiters are ? , ' ; | { } [ ] @ ~ # $ _

VCAST_RPTS_FAIL_COLOR

```
clicast -lada Option VCAST_RPTS_FAIL_COLOR <color>
```

Category: Report Options

Description: This color is used for text indicating failing test cases or failing totals.

VCAST_RPTS_HEADER

```
clicast -lada Option VCAST_RPTS_HEADER <text to prepend to title> | <header
text> | <html file>
```

Category: Report Options

Description: Prepend a text string to the title or add a new section to the Full Report, just below the title, to include the contents of a file in TEXT or HTML format. Input can be a text string, or a file containing several lines. If an HTML file is provided, the contents should provide a ** ** section with a table. For example:

```
<div class='report-block'>
  <h2>Additional Data</h2>
  <table class='table table-small'>
    <tr><th>Tests created by</th><td>Fred Williams</td></tr>
    <tr><th>Source revision</th><td>123abc789</td></tr>
  </table>
 </div>
```

VCAST_RPTS_HEADING_FONT_SIZE

```
clicast -lada Option VCAST_RPTS_HEADING_FONT_SIZE <font size>
```

Category: Report Options

Description: Heading font size for HTML reports.

VCAST_RPTS_NOTES_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_NOTES_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of notes columns in text reports.

VCAST_RPTS_PASS_COLOR

```
clicast -lada Option VCAST_RPTS_PASS_COLOR <color>
```

Category: Report Options

Description: This color is used for text indicating passing test cases or passing totals.

VCAST_RPTS_PRETTY_PRINT_HTML

```
clicast -lada Option VCAST_RPTS_PRETTY_PRINT_HTML True | False
```

Category: Report Options

Description: This option enables whitespace indentation in HTML and XML report-related files. It is off by default to allow better performance and reduced memory usage.

VCAST_RPTS_REQUIREMENTS_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_REQUIREMENTS_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of requirements columns in text reports.

VCAST_RPTS_RESULT_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_RESULT_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of result columns in text reports. The default value is 16.

VCAST_RPTS_SECTION_TITLE_BGCOLOR

```
clicast -lada Option VCAST_RPTS_SECTION_TITLE_BGCOLOR <color>
```

Category: Report Options

Description: Background color for section titles in HTML reports.

VCAST_RPTS_SELF_CONTAINED

```
clicast -lada Option VCAST_RPTS_SELF_CONTAINED True | False
```

Category: Report Options

Description: When True, HTML reports are created as a single, self-contained file with an embedded stylesheet and embedded images (for all built-in, non user-modified reports). Set the option to True (default value) when planning to email reports. When False, HTML reports are created as multiple files, storing stylesheets and images in the same directory as the report. Set the option to False when

serving reports from a webserver.

VCAST_RPTS_SHOW_VERSION

```
clicast -lada Option VCAST_RPTS_SHOW_VERSION True | False
```

Category: Report Options

Description: Show the VectorCAST version in the configuration section of the reports.

VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of subprogram columns in text reports.

VCAST_RPTS_TABLE_BORDER_SIZE

```
clicast -lada Option VCAST_RPTS_TABLE_BORDER_SIZE <border size>
```

Category: Report Options

Description: Table border size for tables in HTML reports.

VCAST_RPTS_TABLE_CELL_PADDING

```
clicast -lada Option VCAST_RPTS_TABLE_CELL_PADDING <padding>
```

Category: Report Options

Description: Table cell padding for tables in HTML reports.

VCAST_RPTS_TABLE_DATA_BGCOLOR

```
clicast -lada Option VCAST_RPTS_TABLE_DATA_BGCOLOR <color>
```

Category: Report Options

Description: Background color for table data in HTML reports.

VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR

```
clicast -lada Option VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR <color>
```

Category: Report Options

Description: This background color is used for failing data cells in reports and tables.

VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR

```
clicast -lada Option VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR <color>
```

Category: Report Options

Description: This background color is used for partially passing data cells in reports and tables, as well as warning messages in reports.

VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR

```
clicast -lada Option VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR <color>
```

Category: Report Options

Description: This background color is used for passing data cells in reports and tables.

VCAST_RPTS_TABLE_DATA_TEXT_ALIGNMENT

```
clicast -lada Option VCAST_RPTS_TABLE_DATA_TEXT_ALIGNMENT left | right |
center
```

Category: Report Options

Description: Text alignment (left, right, or center) for table data in HTML reports.

VCAST_RPTS_TABLE_HEADING_BGCOLOR

```
clicast -lada Option VCAST_RPTS_TABLE_HEADING_BGCOLOR <color>
```

Category: Report Options

Description: Background color for table headings in HTML reports.

VCAST_RPTS_TESTCASE_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_TESTCASE_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of testcase columns in text reports.

VCAST_RPTS_UNIT_COLUMN_WIDTH

```
clicast -lada Option VCAST_RPTS_TESTCASE_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of unit columns in text reports.

VCAST_RPTS_USER_REPORTS_DIR

```
clicast -lada Option VCAST_RPTS_USER_REPORTS_DIR <directory>
```

Category: Report Options

*Description:*This option points to a directory containing additional user reports.

VCAST_RPTS_WRAP_NOTES

```
clicast -lada Option VCAST_RPTS_WRAP_NOTES True | False
```

Category: Report Options

Description: This option will cause the text in the notes section to wrap at the first space before 80 characters.

VCAST_SCRIPT_EXPAND_ARRAYS

```
clicast -lada Option VCAST_SCRIPT_EXPAND_ARRAYS True | False
```

Category: Execute Options

Description: By default, when an array of scalars is written to a test script, identical values are condensed to one script line. By setting this option, each array element will get its own script line.

VCAST_SCRIPT_IN_CREATION_ORDER

```
clicast -lada Option VCAST_SCRIPT_IN_CREATION_ORDER True | False
```

Category: Execute Options

Description: By default, test scripts are exported in alphabetical order within unit and subprogram. Setting this option will cause the script to be generated in the order that the test cases were created.

VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS

```
clicast -lada Option VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS True | False
```

Category: Report Options

Description: In script logs for large test scripts, error messages can be overwhelmed by the number of status messages. Setting this flag prevents normal status messages from being added to the log - only error messages remain.

VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES

```
clicast -lada Option VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES True | False
```

Category: Report Options

Description: When this option is set, global and parameter data will only be displayed in the test execution report if expected values are provided. This option will result in smaller execution reports and faster test execution in cases where there are is a lot of global and/or parameter data input.

VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES

```
clicast -lada Option VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES True |
False
```

Category: Report Options

Description: When this option is set, events will only be displayed in the test execution report if expected values exist for that event. This option will result in smaller execution reports in cases where there are is a lot of global and/or parameter data input.

VCAST_SHOW_STDOUT_CONSOLE

```
clicast -lada Option VCAST_SHOW_STDOUT_CONSOLE True | False
```

Category: Execute Options

Description: When this option is set, a new console will be opened during test harness execution to facilitate Standard I/O. If 'Redirect standard output' or 'Redirect standard error' is set, this option is ignored. (Windows Only)

VCAST_SIMPLIFIED_CONDITION_COVERAGE

```
clicast -lada Option VCAST_SIMPLIFIED_CONDITION_COVERAGE True | False
```

Category: Coverage Options

Description: This option suppresses the computation and reporting of Equivalence Pairs when MC/DC Coverage is active. Since Equivalence Pairs are not computed, MC/DC tests cannot be generated. This option is primarily used for Medical Device testing, and is defaulted to True when IEC-62304 (Medical) Class C Code Coverage is selected. The default is False for all other cases.

VCAST_SORT_METRICS_RPT_BY_DIR

```
clicast -lada Option VCAST_SORT_METRICS_RPT_BY_DIR <True | False>
```

Category: Report Options

Description: Sort Metrics report by directory.

VCAST_SPEC_FILE_EXTENSIONS

```
clicast -lada Option VCAST_SPEC_FILE_EXTENSIONS <spec file extensions>
```

Category: Language Options

Description: Use this option to specify the ada spec file extensions.

VCAST_SPLIT_INSTRUMENTED_FILE

```
clicast -lada Option VCAST_SPLIT_INSTRUMENTED_FILE <integer number>
```

Category: Builder Options

Description: Approximate number of lines in an source file. Used when source unit has separates, and the compiler cannot handle a single file containing the unit and all its separates.

VCAST_SPLIT_LARGE_FILE

```
clicast -lada Option VCAST_SPLIT_LARGE_FILE True | False
```

Category: Builder Options

Description: When the Unit Under Test or a Custom Coverage unit contains 'separates', VectorCAST will combine them into one file for parsing and instrumentation. Set this option if this combined file is too large for the compiler. (Only works for Green Hills, ObjectAda, and ActivAda compilers.)

VCAST_STRICT_TEST_CASE_IMPORT

```
clicast -lada Option VCAST_STRICT_TEST_CASE_IMPORT True | False
```

Category: Execute Options

Description: Normally, if there are errors when importing a test script, VectorCAST will ignore any object errors. When this option is selected, errors in the script will cause the testcase to not be allowed to execute until the script is fixed or until the testcase is edited.

VCAST_SUPPORT_TAGGED_INHERITANCE

```
clicast -lada Option VCAST_SUPPORT_TAGGED_INHERITANCE True | False
```

Category: Builder Options

Description: When set, the tool will add functions defined in other units that contain a parent of a tagged type defined in the current unit.

VCAST_SUPPRESS_COVERABLE_FUNCTIONS

VCAST_SUPPRESS_DEBUG_FLAG

```
clicast -lada Option VCAST_SUPPRESS_DEBUG_FLAG True | False
```

Category: Builder Options

Description: This option will force VectorCAST to compile source code without debug information.

VCAST_TEST_ANCESTORS

```
clicast -lada Option VCAST_TEST_ANCESTORS True | False
```

Category: Builder Options

Description: When set, the tool will add functions defined in the ancestors of the UUT to the list of functions that can be tested for the environment.

VCAST_TESTCASE_FAIL_ON_NO_EXP_RETURN

```
clicast -lada Option VCAST_TESTCASE_FAIL_ON_NO_EXP_RETURN True | False
```

Category: Execute Options

Description: If this option is set, then testcases that have no expected return value or expected user code for the return value will be marked as failed. The default value is FALSE. Note that this option is specific to the return value of the subprogram under test only.

VCAST_TESTCASE_FAIL_ON_NO_EXPECTED

```
clicast -lada Option VCAST_TESTCASE_FAIL_ON_NO_EXPECTED True | False
```

Category: Execute Options

Description: If this option is set, any test cases that have no expected results or expected user code will be marked as failed. Its default value is FALSE.

VCAST_UNCONSTRAINED_STRING_SIZE

```
clicast -lada Option VCAST_UNCONSTRAINED_STRING_SIZE <integer number>
```

Category: Builder Options

Description: The maximum length an unconstrained string can be in Ada.

VCAST_UNCOVERED_LINE_INDICATOR

```
clicast -lada Option VCAST_UNCOVERED_LINE_INDICATOR <'#' | '$' | '%' | '&'
| '+' | ' '>
```

Category: Coverage Options

Description: Character to be used to indicate an uncovered statement in a coverage report. Valid coverage indicators are:

$ % & +

VCAST_UNEXPECTED_EXCEPTIONS_FAIL

```
clicast -lada Option VCAST_UNEXPECTED_EXCEPTIONS_FAIL True | False
```

Category: Execute Options

Description: If this option is set, then if an exception is raised during the execution of a testcase and no exception was expected, the testcase is marked as failed.

VCAST_UNEXPECTED_SIGNALS_FAIL

```
clicast -lada Option VCAST_UNEXPECTED_SIGNALS_FAIL True | False
```

Category: Execute Options

Description: If this option is set, then if a signal is raised during the execution of a testcase then testcase is marked as failed.

VCAST_UNIT_TYPE

```
clicast -lada Option VCAST_UNIT_TYPE UUT | SBF
```

Category: Builder Options

Description: Default setting for Unit type in Create New Environment Wizard. UUT allows stubbing only for functions external to unit being tested; Stub By Function (SBF) allows stubbing of functions within the unit being tested.

VCAST_USE_COMPOUND_FOR_BATCH

```
clicast -lada Option VCAST_USE_COMPOUND_FOR_BATCH True | False
```

Category: Target Options

Description: This option allows you to run all of your test cases during a single test driver execution. This is useful if there are many steps involved in running on your target platform. When this option is set, and you choose to Batch Execute All, VectorCAST creates a temporary compound test case to execute all tests. The temporary compound contains all Initialization test cases, followed by all simple and Compound test cases.

VCAST_USE_GH_BLD_FILE

```
clicast -lada Option VCAST_USE_GH_BLD_FILE <.bld filename>
```

Category: Language Options

Description: Set this to an appropriate '.bld' file so that VectorCAST will compile and link using the 'build' command instead of compiling each file separately.

VCAST_USE_GNATMAKE

```
clicast -lada Option VCAST_USE_GNATMAKE True | False
```

Category: Language Options

Description: This option forces VectorCAST to use 'gnatmake' to compile, bind and link the test

harness. If this option is not set, VectorCAST will use 'gcc' to compile the test harness files, 'gnatbind' to bind, and 'gnatlink' to link.

VCAST_USE_GPRBUILD

```
clicast -lada Option VCAST_USE_GPRBUILD True | False
```

Category: Builder Options

Description: This option forces VectorCAST to use 'gprbuild' to compile/link environments built from GPR files. This may slow the build process, as this command will compile objects in other languages that are part of the build process.

VCAST_USE_RELATIVE_PATHS

```
clicast -lada Option VCAST_USE_RELATIVE_PATHS True | False
```

Category: Builder Options

Description: If set, the new environment dialog will use relative paths. The target I/O directory will also use relative paths.

VCAST_USE_STATIC_MEMORY

```
clicast -lada Option VCAST_USE_STATIC_MEMORY True | False
```

Category: Coverage Options

Description: This option tells VectorCAST that only static data allocation can be used on the target platform. This is used when collecting coverage data. If this is set to True, then use the options VCAST_MAX_MCDC_STATEMENTS and VCAST_MAX_COVERED_SUBPROGRAMS to configure the static allocation.

VCAST_USER_CODE_FILE

```
clicast -lada Option VCAST_USER_CODE_FILE <script filename>
```

Category: Builder Options

Description: Set this to a file containing environment user code (in scripting format) to get this code automatically inserted into every environment built.

VCAST_USER_CONSTRAINTS

```
clicast -lada Option VCAST_USER_CONSTRAINTS <typemark[,typemark...]>
```

Category: Builder Options

Description: List of array index types that have a large range and are used for unconstrained arrays.

VCAST_USING_GNAT_5

```
clicast -lada Option VCAST_USING_GNAT_5 True | False
```

Category: Builder Options

Description: Newer versions of GNAT use different syntax for the 'gnatchop' command. If 'gcc --version' returns 3.4.4 or greater, this option should be set to TRUE.

VCAST_VERBOSE_MANAGEMENT_REPORT

```
clicast -lada Option VCAST_VERBOSE_MANAGEMENT_REPORT True | False
```

Category: Report Options

Description: This option will copy the first line from the 'Test Notes' box in the test case editor to the test case management report. One possible use for this is for test requirement numbers to be correlated to test cases in the test case management report.

VCAST_VXWORKS_RTP_MODE

```
clicast -lada Option VCAST_VXWORKS_RTP_MODE True | False
```

Category: Target Options

Description: Set this option to run VxWorks executables as a Real-Time process (via the 'rtpSp' command) rather than as a standard process (via the 'ld' and 'sp' commands).

VCAST_WIDE_CHARACTER_SUPPORT

```
clicast -lada Option VCAST_WIDE_CHARACTER_SUPPORT True | False
```

Category: Builder Options

Description: Allow VectorCAST to parse Ada code containing wide characters by using ADA.WIDE_TEXT_IO when reading code. When this option is off, VectorCAST reverts to using ADA.TEXT_IO.

VCAST_WRAP_TEXT_REPORT

```
clicast -lada Option VCAST_WRAP_TEXT_REPORT True | False
```

Category: Report Options

Description: In some text reports, if text is too long for a table cell then this option will allow the text to wrap in a new row.

WHITEBOX

```
clicast -lada Option WHITEBOX No | Yes
```

Category: Builder Options

Description: Default setting for Whitebox checkbox in environment builder. (See the Whitebox Processing section of User's Guide for more details.)

APPENDIX H: RGW - COMMAND LINE REFERENCE

This reference contains an alphabetized list of VectorCAST/Requirements Gateway (RGW) commands.

Requirements Gateway supports the following RGW operations from the command line:

- Import and Export
- Profile Settings
- Repository Manipulation
- Test Automation



Note: Command line operations for RGW must be run from a directory with a CCAST_.CFG file that indicates where the repository is located.

When RGW begins, it loads the ADACAST_.CFG file located in the current working directory and looks for the repository path in VCAST_REPOSITORY. If it cannot locate an entry for VCAST_REPOSITORY, it returns this text: "VCAST_REPOSITORY not set in .CFG file. Please set it using 'clicast option'."

The command line syntax is:

```
clicast -lada RGw -r <command> <arguments>
```

where:

<command>	is the command to execute
<arguments>	are the arguments for the command

RGW Commands

Configure Set

Syntax	**clicast -lada RGw Configure Set <Subsystem Name> <Option Name> <Option Value>**
Description	This command sets any of the configuration options for a particular subsystem. Use Configure Test to determine the names of options that need to be set for a particular subsystem.
Example	**clicast -lada RGw Configure Set DOORS user Administrator** **Key 'user' set for Profile 'DOORS'**

Export

Syntax	**`clicast -lada RGw Export <Subsystem Name>`**
Description	This command exports Requirements Test Data from the RGW Repository to an external Requirements System.
Example	**`clicast -lada RGw Export DOORS`**

Import

Syntax	**`clicast -lada RGw IMport <Subsystem Name>`**
Description	This command imports Requirements into the RGW Repository from an external Requirements system. **`<Subsystem Name>`** specifies the RGW subsystem that the import will work on. The following subsystems are supported: - CSV - Polarion - Jama
Example	**`clicast -lada RGw Import DOORS`**

Initialize

Syntax	**`clicast -lada RGw INitialize`**
Description	This command creates a **`requirements_gateway`** directory with **`repository.json`** and **`settings.json`** files in the location specified by VCAST_REPOSITORY.
Example	**`clicast -lada RGw INitialize`**

Requirement Report

Syntax	**`clicast lada RGw Requirement REPort <Query Type> <Query ARg 1> <Query Arg2>`**
Description	This command returns Requirements that match the supplied query. Query Types can be: **All** - Return all Requirements in the Repository **Pending** - Return all Requirements with pending Test Data to be exported. **Changed** - Return all Requirements that have been changed between the two specified dates in Query Arguments 1 and 2. Date must be specified in the format 'YYYY-MM-DD hh:mm:ss'.

Example	`clicast -lada RGw Requirement REPort Changed 2017-01-02 12:00:00 2017-01-03 12:00:00`

Set Gateway

Syntax	`clicast -lada RGw Set Gateway <Gateway Name>`
Description	Select an RGW integration to use.
Example	`clicast  -lada RGw Set Gateway csv`

Testcase Link

Syntax	`clicast -lada -e <env> -u <unit> -s <sub> -t <test> RGw Testcase Link <Requirement Key>`
Description	This command links a Requirement to the Test Case specified by the Test Case identification flags: `-e -u -s -t`.
Example	`clicast -lada -e ENVIRONMENT_1 -u manager -s Clear_Table -t Clear_Table.001 RGW Testcase Link FR11`

Testcase Mark as Reviewed

Syntax	`clicast -lada -e <env> -u <unit> -s <sub> -t <test> RGw Testcase MArk_as_reviewed`
Description	Where a testcase is flagged as having been impacted by a Requirements change, this command marks the test case specified as having been reviewed and approved, and clears the "Marked for Review" flag.
Example	`clicast -lada -e ENVIRONMENT_1 -u manager -s Clear_Table -t Clear_Table.001 RGW MArk_as_ reviewed`

Testcase Migrate_links

Syntax	`clicast -lada -e <env> RGw Testcase Migrate_links`
Description	This command migrates all legacy test case requirement links from the specified Environment into the Requirements Repository.
Example	`clicast -lada -e ENVIRONMENT_1 RGw Testcase Migrate_links`

Testcase Report

Syntax	**clicast -lada -e <env> -u <unit> -s <sub> -t <test> RGw Testcase Report**
Description	This command returns all Requirements linked to the Test Case specified in the flags.
Example	**clicast -lada -e ENVIRONMENT_1 -u manager -s Clear_Table -t Clear_Table.001 RGW Testcase Report**

Index

*
    indicating a statement covered 335
AALINKER.LIS 118, 447
Abort button 264
aborting test case execution 264
access types 188
actual results 20
Ada language specification 132
ADA.EXCEPTIONS.EXCEPTION_MESSAGE 130
Ada83 135
Add button
    in List tab 215
additional information about exception occurrence 130
aggregate coverage report 343
All Values button 201, 205
allow stubs of Ada runtime 128
alphabetize parameter tree parameters 180
alphabetize test cases 65, 158
alternate direct array indexing 129
animated control flow 356
animation coverage type 361
animation play speed 367
Any button
    in List Values tab 183, 215
    in Range Values tab 213
    in Scalar Values tab 210
Apex mutual imports 129
apply to array indices" 203, 211
apply to expanded array indices only" 203, 211
array constraints
    modifying 206
array types
    entering multi-dimension indices 206
    unconstrained arrays 206
arrays
    clearing data 205, 213
    collapsing unused elements 200
    entering data 203, 211
    expanding certain elements 201
    expanding entire 201
    expanding with + 198-200
    expanding with <<Expand Indices>> dialog 201, 205
    ways to expand 205
automate test script maintenance 223
automatic finalization 170
automatic initialization 170
automatically clear test user code 289
background color for failing cells 308
background color for partially passing cells 308
background color for passing cells 307
background colors
    changing for coverage output 371
backgrounds
    printing 83
basis path
    edit script for 245
    viewing for subprogram 244, 351
basis path analysis 351
basis path output
    building test cases from 351
basis paths
    do not automatically generate 245
batch execute all 263
batch file 138-140, 143, 490
binder/linker log file 118
blank cells text 305
Branch coverage 328
breakpoints
    removing 360

setting in coverage animation 359
buffered coverage I/O 361
build options
set coverage 104
whitebox 104
build stubs for pragma imports 129
builder option
multiunit whitebox 106
building test cases 19
CBA
aggregate coverage report 388
change cba display color 383
covered by analysis report 386
edit analysis result 382
export analysis files 385
import analysis files 384
metrics report 389
re-instrumenting with cba data 385
remove analysis files 385
view cba data in reports 386
view cba in coverage viewer 383
cba editor
branch coverage 380
mcdc coverage 381
statement coverage 380
cbt
change based testing 321
CCAST_.CFG
opening 82
change-based testing
see cbt 321
changing application font 67
changing industry mode 30
character types 472
choose compiler 102
circular references 106
class-based testing 192
Clear Expected button 412
Clear Input button 412
clear unapplied test script data 220
ClearCaseTM
integrating regression scripts with 144
CLICAST
accessing in Linux 494
accessing in Windows 494
command format 494
console report 495
cover initialize branch 328
cover report aggregate 344
deprecated
execute extract coverage 261
execute extract results 261
reports report 261
environment build 115
environment delete 147
environment extract build_log 118
environment extract compile errors 117
environment extract link errors 118
environment extract overview 116
environment re_build 147
environment rebuild_range_data 153
environment recompile automatic 149
environment recompile create script 150
environment recompile instrumented 149
environment recompile run 152
environment relink 146, 153
environment script create 102, 148
environment script quick 148
environment script run 148-149
environment stub_user clear 422
environment stub_user compile 422
environment user clear 405

environment user compile 405
execute all command 503
execute batch 263
execute clear control flow 267
execute debug 264
execute run 255
execute save control flow 267
IF DOCPROPERTY Project C 344
messages 452
options
ADA_DEBUG_CMD 122, 499
ADA_EXECUTE_CMD 499
ADACAST_IO_BODY 499
ADACAST_IO_SPEC 499
BINDER_OPTIONS 121, 500
CAST_WIDE_CHARACTER_SUPPORT 27, 131
COMPILATION_SYSTEM 120, 500
COMPILER_OPTIONS 120, 500
COMREADER_BAUD 500
COMREADER_COMPORT 500
COMREADER_DATA_BITS 500
COMREADER_ENABLED 501
COMREADER_PARITY 501
COMREADER_STOP_BITS 501
COVERAGE_ANIMATION_PLAY_SPEED_RATIO 367, 501
COVERAGE_IO_TYPE 362, 501
COVERAGE_TARGET 502
COVERAGE_TYPE 124-126, 502
EVENT_LIMIT 287, 314, 502
EXECUTABLE_EXTENSION 122, 502
FLOATING_POINT_TOLERANCE 288, 313, 503
GLOBALS_DISPLAY 299, 318, 503
GNAT_CHOP_OPTIONS 122
GNATCHOP_OPTIONS 503
GNATMAKE_OPTIONS 504
GPR_OPTIONS 121, 504
LIBRARY_OPTIONS 121, 504
LINES_PER_PAGE 504
LINKER_OPTIONS 121, 504
MANAGE_ENVIRONMENT_VARIABLE 504
MANAGE_PARENT_LIB 505
MANAGE_ROOT_SOURCE_DIR 505
MAX_VARY_RANGE 505
METRICS_TOTAL_LEVEL_DISPLAY 298, 505
NO_DELAY_STATEMENTS 505
NUMBER_OF_DATA_PARTITIONS 506
OPTIONS_Append 495
PAGE_WIDTH 506
PREPARE_COMPILED_IN_DATA 506
PREPROCESSOR_OPTIONS 121, 506
RANGE_CHECK 287, 507
REPORT_OBJECTS 507
REPORT_PARAMETERS 507
reports extract 71
SHOW_NOTES_IN_FULL_REPORT 303, 507
STANDARD_ERROR 289, 507
STANDARD_OUTPUT 288, 508
STUB_DEPENDENCIES 123, 508
TARGET_AE653_MAKE_FILE 508

TARGET_APEX_LIBRARY 508

TARGET_BOARD_NAME 508

TARGET_BOOT_HOSTNAME 509

TARGET_BUILD_FILE 509

TARGET_COMMAND_OPTIONS 509

TARGET_COMMAND_VERB 509

TARGET_CPU 509

TARGET_INTEGRITY_ADDRESS_SPACE_MODE 510

TARGET_INTEGRITY_LIBRARY 510

TARGET_IO_DIRECTORY 510

TARGET_SIM_CMD 510

TARGET_SUPPORTS_TIC_IMAGE 511

TARGET_TFTP_DIR 511

TARGET_USE_RTSERV_COMM 511

TARGET_USE_TORNSERV 511

TARGET_VARIANT 511

TARGET_XFER_METHOD 512

TEST_CASE_TIMEOUT 512

UNCON_ARRAY_SIZE 512

VCAST_ADA_ACCESSOR_FUNCTIONS 512

VCAST_ADA_FILE_EXTENSIONS 122, 512

VCAST_ADA_LANGUAGE_SPECIFICATION 132, 513

VCAST_ADAOPTS 513

VCAST_ADD_DONT_STUB_UNITS 513

VCAST_ALLOW_STUB_OF_STD_LIB 129, 513

VCAST_APEX_CLEARCASE 513

VCAST_APEX_INSTALLATION_DIR 514

VCAST_APEX_MUTUAL_IMPORTS 129, 514

VCAST_APPEND_TO_TESTINSS 514

VCAST_AUTO_CLEAR_TEST_USER_CODE 290

VCAST_CMD_AFTER_INSTRUMENTATION 367, 514

VCAST_CMD_DURING_ENV_CLOSE 514

VCAST_CMD_DURING_ENV_OPEN 515

VCAST_COMMAND_LINE_DEBUGGER 515

VCAST_COMPACT 515

VCAST_COMPILE_DUMB_STUBS 128, 515

VCAST_COMPILE_GENERIC_STUBS 132, 516

VCAST_CONFIGURABLE_ADA_IO 516

VCAST_CONSOLE_REPORT_FORMAT 495, 516

VCAST_CONVERT_OCTAL_HEX_STRINGS_ASCII 302

VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII 516

VCAST_COVER_SEPARATE_TYPES_FILES 517

VCAST_COVERAGE_DATABASE_CACHE_SIZE 363, 517

VCAST_COVERAGE_DATABASE_SIMPLIFIED_LOCKING 517

VCAST_COVERAGE_FIELD_WIDTH 363, 517

VCAST_CUSTOM_REPORT_FORMAT 308, 518

VCAST_DEFAULT_SCRIPT_HEADER 305, 518

VCAST_DETECT_UNUSED_EXPECTED_UC 290, 518

VCAST_DISABLE_AUTO_BASIS_PATH_GEN 134, 518

VCAST_DISABLE_BOOLEAN_CAST 364, 369, 518

VCAST_DISABLE_DIRECT_ARRAY_INDEXING 519

VCAST_DISABLE_TEST_MAINTENANCE 519

VCAST_DISPLAY_EMPTY_COVERAGE 296, 519

VCAST_DISPLAY_FUNCTION_COVERAGE 296, 519

VCAST_DO_COMBINATION 293, 316, 520

VCAST_DONT_STUB_ANCESTORS 128, 520

VCAST_EMPTY_TESTCASE_FAIL 520

VCAST_ENHANCED_FOR_LOOP_COVERAGE 520

VCAST_ENVIRONMENT_FILES 520

VCAST_EXECUTE_WITH_STDOUT 521

VCAST_EXPECTED_BEFORE_UUT_CALL 301, 316, 521

VCAST_EXTENDED_EXCEPTION_INFORMATION 130, 521

VCAST_FIELD_DOT_NOTATION 521

VCAST_FILE_ENCODING 522

VCAST_FILE_INDEX 522

VCAST_FILE_VERSION_COMMAND 304, 522

VCAST_FLOAT_PRECISION 303, 522

VCAST_FORCE_ADA83 135

VCAST_FORCE_REPORT_DATES 523

VCAST_GEN_ALL_EXECUTION_REPORT_FORMATS 523

VCAST_GH_INT_FILE 523

VCAST_GH_INTEX_CMD 523

VCAST_GH_PP_COMMAND 122, 524

VCAST_GMART_FILE_IO 524

VCAST_GNAT_INCLUDE_PATH 121, 524

VCAST_GNAT_OBJECTS_PATH 121, 524

VCAST_GPR_TEMPLATE 524

VCAST_GPR_USES_NAMING 134, 525

VCAST_IGNORE_INCOMPLETE_TESTS 290, 294, 525

VCAST_IMPORT_FOREIGN_OBJECTS 525

VCAST_INSTRUMENT_ASSIGNMENTS 525

VCAST_INSTRUMENT_PARAMETERS 364, 526

VCAST_MAX_COVERED_SUBPROGRAMS 365, 526

VCAST_MAX_MCDC_ABORT_INST 363, 526

VCAST_MAX_MCDC_CONDITIONS 362, 526

VCAST_MAX_MCDC_STATEMENTS 366, 526

VCAST_MAX_STRING_LENGTH 527

VCAST_MAX_TABLE_SUBCONDITIONS 527

VCAST_MAX_TARGET_FILES 527

VCAST_MULTIBYTE_CHARACTERS 527

VCAST_NEW_ARCHITECTURE 129, 528

VCAST_NO_EXCEPTION_HANDLERS 528

VCAST_NO_RECURSIVE_CALLS 528

VCAST_NO_STANDARD_PKG_USAGE 134, 528

VCAST_NOTES_SECTION_TEMPLATE 304, 528

VCAST_OLD_STYLE_MANAGEMENT_REPORT 302, 529

VCAST_POST_LINK_COMMAND 529

VCAST_PRAGMA_IMPORT 129, 529

VCAST_PRE_EXECUTE_CMD 529

VCAST_PRECOMPILE_SCRIPT 131, 530

VCAST_PREPEND_TO_PATH_DIRS 530

VCAST_PREPROCESS_ADA_FILES 135

VCAST_PSC_MAKE_FILE_BASE_DIRECTORY 530

VCAST_RAVEN_FILE_IO 530

VCAST_RECOMPILE_SEPARATES 128, 530

VCAST_REFERENCED_GLOBALS 531

VCAST_REPOSITORY 128, 531

VCAST_RHI_CWD 122

VCAST_RHI_HOSTNAME 122

VCAST_RPTS_AUTOMATIC_BGCOLOR 531

VCAST_RPTS_BODY_BGCOLOR 531

VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH 311, 531

VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH 311, 532

VCAST_RPTS_CUSTOM_CONFIG 68, 532

VCAST_RPTS_CUSTOM_CSS 75, 309, 532

VCAST_RPTS_DATE_COLUMN_WIDTH 310, 532

VCAST_RPTS_DEFAULT_FONT_COLOR 532

VCAST_RPTS_DEFAULT_FONT_FACE 533

VCAST_RPTS_DEFAULT_FONT_SIZE 533

VCAST_RPTS_DELIMITER 272, 533

VCAST_RPTS_EMPTY_DATA_STRING 306, 533

VCAST_RPTS_FAIL_COLOR 533

VCAST_RPTS_HEADER 305, 534

VCAST_RPTS_HEADING_FONT_SIZE 534

VCAST_RPTS_NOTES_COLUMN_WIDTH 311, 534

VCAST_RPTS_PASS_COLOR 534

VCAST_RPTS_PRETTY_PRINT_HTML 535

VCAST_RPTS_REQUIREMENTS_COLUMN_WIDTH 535

VCAST_RPTS_RESULT_COLUMN_WIDTH 310, 535

VCAST_RPTS_SECTION_TITLE_BGCOLOR 535

VCAST_RPTS_SELF_CONTAINED 76, 535

VCAST_RPTS_SHOW_VERSION 303, 536

VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH 310, 536

VCAST_RPTS_TABLE_BORDER_SIZE 536

VCAST_RPTS_TABLE_CELL_

PADDING 536

VCAST_RPTS_TABLE_DATA_BGCOLOR 536

VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR 308, 536

VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR 308, 537

VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR 308, 537

VCAST_RPTS_TABLE_DATA_TEXT_ALIGNMENT 537

VCAST_RPTS_TABLE_HEADING_BGCOLOR 537

VCAST_RPTS_TESTCASE_COLUMN_WIDTH 310, 537

VCAST_RPTS_UNIT_COLUMN_WIDTH 309, 538

VCAST_RPTS_USER_REPORTS_DIR 538

VCAST_RPTS_WRAP_NOTES 538

VCAST_SCRIPT_EXPAND_ARRAYS 538

VCAST_SCRIPT_IN_CREATION_ORDER 538

VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS 294, 539

VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES 300, 316, 539

VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES 301, 317, 539

VCAST_SHOW_STDOUT_CONSOLE 291, 539

VCAST_SIMPLIFIED_CONDITION_COVERAGE 540

VCAST_SORT_METRICS_RPT_BY_DIR 295, 540

VCAST_SPEC_FILE_EXTENSIONS 123, 540

VCAST_SPLIT_INSTRUMENTED_FILE 540

VCAST_SPLIT_LARGE_FILE 540

VCAST_STRICT_TEST_CASE_IMPORT 289, 541

VCAST_SUPPORT_TAGGED_INHERITANCE 541

VCAST_SUPPRESS_COVERABLE_FUNCTIONS 541

VCAST_SUPPRESS_DEBUG_FLAG 130, 134, 541

VCAST_TARGET_DIRECT_ARRAY_INDEXING 130

VCAST_TEST_ANCESTORS 541

VCAST_TESTCASE_FAIL_ON_NO_EXP_RETURN 293, 541

VCAST_TESTCASE_FAIL_ON_NO_EXPECTED 289, 542

vcast_unconstrained_string_size 131

VCAST_UNCONSTRAINED_STRING_SIZE 542

VCAST_UNCOVERED_LINE_INDICATOR 368, 542

VCAST_UNEXPECTED_EXCEPTIONS_FAIL 289, 542

VCAST_UNEXPECTED_SIGNALS_FAIL 293, 542

VCAST_UNIT_SPECIFIC_USER_CODE_INSERTION 133

VCAST_UNIT_TYPE 124, 543

VCAST_USE_COMPOUND_FOR_BATCH 543

VCAST_USE_GH_BLD_FILE 122, 543

VCAST_USE_GNATMAKE 132, 543

VCAST_USE_GPRBUILD 133, 544

VCAST_USE_RELATIVE_PATHS 544
VCAST_USE_STATIC_MEMORY 365, 544
VCAST_USER_CODE_FILE 544
VCAST_USER_CONSTRAINTS 544
VCAST_USING_GNAT_5 545
VCAST_VERBOSE_MANAGEMENT_REPORT 302, 545
VCAST_VXWORKS_RTP_MODE 545
VCAST_WIDE_CHARACTER_SUPPORT 545
VCAST_WRAP_TEXT_REPORT 545
WHITEBOX 127, 546
reports alternate 272
reports custom actual 257, 271
reports custom cba 388
reports custom coverage 344
reports custom equivalence_matrices 355
reports custom full 273
reports custom management 273
reports custom test 269
reports diagnostic 49, 252
rgw testcase mark as reviewed 437
test compound_only 173
test csv2tst create 241
test csv2tst load 239
test csv2tst template 234
test delete 159-160
test script create 220
test script log 221
test script run 221
test script template 221
test unapplied clear 220
test user clear 412
tools auto_test_generation 166
tools basis_path 244, 352
tools cover <coverage-type> 332
tools cover disable 332-333
tools cover enable 333
tools cover level_B 331
tools cover none 333
tools cover statement 327
tools execute_commands 140
tools export cba 385
tools export_coverage 376
tools export_results_coverage 376
tools import_coverage 376
tools regression_test 139, 143
tools update_basis_path 245
clicast actuals to expected 178, 268
closing environment 136
closure concept 17
ignoring units 18
minimizing 17
code complexity 244, 351
code coverage summary 347
collapse all subprograms 340
collapsing arrays 200
colors 83
combination testing 291, 314
compare expected before UUT call 301, 316
compile dumb stubs 128
compile generic stubs 132
compile log file 117
compile UUT after stubs 128
components
of test harness 16
compound Only test cases 173
compound test cases
automatic initialization finalization 170
building 167

deleting test case from 172
enable/disable reporting for slots 176
executing 261
find slot in test case tree 172
nested 173
opening Coverage Viewer from 172
opening test case from 171
rearranging order of test cases in 171
search editor 172
configuration file
created by options dialog 242
how to use several of 242
opening 82
configuration management 138
configure stubs 417
how to edit 418
console report
compount test output 497
default ouput 496
format 495
legacy output 496
quiet output 497
verbose output 497
constrained array types 198
contacting
technical support 50
continuation character 477
control flow
animated 356
clearing 267
edit 264
edit a control flow 265
edit context menu 265
edit subprogram name 266
executing a test case with a control flow that has changed 266
expand/collapse subprogram tree 264
manually creating a control flow 265
saving 267
subprogram identifier 266
subprograms panel 264
test case editor 264
test execution report 266
view 264
convert octal/hex-encoded strings to ASCII 301
copying text 84
cover assignment statements in MC/DC 363
coverage
aggregate coverage report 343
animation play speed 367
before using 325
coverage viewer 334
customizing display 340
deleting imported results 376
disabling 333
enabling 333
exporting 373
importing 376
importing from VectorCAST/Cover 371
in status bar 40
initializing Branch 328
initializing custom 331
initializing DO-178B Level A 330
initializing DO-178B Level B 330
initializing DO-178B Level C 327
initializing EN-50128 SIL 1/2 327
initializing EN-50128 SIL 3 330
initializing EN-50128 SIL 4 330
initializing IEC61508 SIL 1/2 327
initializing IEC61508 SIL 3 330
initializing IEC61508 SIL 4 330
initializing IEC-62304 Class A 327
initializing IEC-62304 Class B 330

initializing IEC-62304 Class C 330
initializing ISO-26262 ASIL A 327
initializing ISO-26262 ASIL B/C 330
initializing ISO-26262 ASIL D 330
initializing MC/DC 328
initializing statement 326
initializing Statement 327
initializing Statement+BRANCH 330
initializing Statement+MC/DC 330
not instrumenting sections of source code 332
opening test case which caused 341
pairs status 329, 345
recompile instrumented code 149
removing data 343
results navigation 339
uncovered line indicator 367
uninstrument 332
coverage analysis editor 379
coverage animation
setting breakpoints 359
toolbar 358
coverage data used in reports 268
Coverage menu 325
Export Script 373
Import Results 371
Initialize Custom 331
Remove All Coverage Data 343
Uninstrument 332
View Import Log 373
coverage reports
printing colors 83
coverage type
set default 105
coverage viewer 333-334
branch coverage 336
mcdc coverage 337
numbers along left-hand side 335
open coverage viewer 333
statement coverage 335
viewing cba lines 383
COVERAGE_IO_TYPE 361
covered by analysis
see CBA 379
covered by analysis report 386
covered lines 369
create CSV template
CSV options panel 229
parameter tree 229
create csv template file 228
create tab delmited template file 228
create test
from CSV map 238
creating
regression test 138
creating new reports 76
creating reports 20
css 68
csv
mapping a column 237
CSV
edit map test case 239
csv file 228
CSV map
create from existing file 234
csv map
create tests 238
mapping view 237
rows and columns in file 234
table.csv 234
csv mapping
save 237

csv options
Load all rows 236
Load rows from to 236
Row containing column headings 236
Rows per testcase 236
Rows to skip 236
Test name column 236
Test notes column 236
CSV options panel 229
CSV Options panel
Create Template type selection 229
csv options panel
comma separted values 230
tab separted values 230
value seperator type 230
CSV reports 272
csv template
create template file 232
populate with data 233
csv template headers 231
csv template map
modify column order 232
custom
ignored 113
not stubbed 113
stub 113
custom css
add 74
add from command line 75
add from gui 74
create 73
embedding style sheets 76
custom script header text file 305
custom style sheet 73
custom templates 70
configure 71
custom tools
adding 247
default tools 246
deleting 250
import export 251
settings for 249
sharing 251
testing 248
customer number
obtaining 50
customize reports
add footer 72
add logo 72
add section 68
create new 76
custom css 73
existing 68
order section 70
remove section 68
replace section 69
report components 76
run new report 79
sections 71
style 73
templates 70
customizing Coverage Viewer 340
customizing report sections 71
customizing report style 73
customizing reports 68
cutting text 84
data
entering directly in MDI window 179
entering for parameters 209
entering for test cases 182
entering in Scalar Values tab 210
entering lists 183

entering on List Values tab 183, 215
entering on Range Values tab 213
entering ranges 182
for parameters 412
global 182
ranges
testing all combinations 291, 314
reading from a database 406
two ways to enter 178
data types 185
access 188
constrained arrays 198
entering number for enumeration 186
entering number in different bases 186
entering real numbers in scientific notation 186
enumeration 185
null strings 184
of type string 187
records 187
system address types 188
variant record types 187
database
reading data from
user code example 406
Default button 242
default coverage type 124
delay between test executions
specifying 170
deleting an environment 147
deleting test cases 158-159
deleting test cases, part of compound 159
delta value
in ranges 214
dependent unit
meaning of 16
smart stubs 16
disable boolean cast in instrumentation 364, 368
disable coverage 333
display coverage in reports (always) 296
display theme 60
do not allow recursive calls in harness 106
do not automatically generate basis paths 134, 245
do not stub ancestors 128
do not use package STANDARD 134
docking window
floating 34
moving 33
returning to default layout 34-35
Down button
in List tab 216
driver program
meaning of 16
duplicate names in exported scripts 220
duplicating test cases 158
edit existing map test case 239
Edit Harness Source 243
Edit menu
Copy 84
Cut 84
Find 85
Find again 87
Find Again 94
Goto Line 94
Paste 84
Redo 84
Undo 84
enable coverage 333
enumerals
testing out-of-range values 191
enumeration types 185, 472
entering number for 186

ENVIRO.ADDITIONAL_UNIT_BODIES 462
ENVIRO.ADDITIONAL_UNIT_SPECS 462
ENVIRO.COMPILER 454
ENVIRO.COVERAGE_TYPE 456
ENVIRO.END 463
ENVIRO.IGNORE 456
ENVIRO.IMPORT 456
ENVIRO.INDUSTRY MODE 463
ENVIRO.MAX_VARY_RANGE 456
ENVIRO.NAME 454
ENVIRO.NEW 454, 463
ENVIRO.PARENT_LIB 455
ENVIRO.STUB 456
ENVIRO.STUB_DEPEND_USER_CODE_FILE 461
ENVIRO.STUB_ENTRY_USER_CODE 460
ENVIRO.STUB_EXIT_USER_CODE 460
ENVIRO.STUB_USER_CODE_FILE 460
ENVIRO.USER_CODE_CAPTURE 459
ENVIRO.USER_CODE_DEPENDENCIES 458
ENVIRO.USER_CODE_INITIALIZE 459
ENVIRO.USER_CODE_OBJECTS 458
ENVIRO.USER_CODE_ONE_SHOT_INIT 458
ENVIRO.USER_CODE_ONE_SHOT_TERM 459
ENVIRO.USER_CODE_STUB_PROCESSING 460
ENVIRO.USER_CODE_TIMER_START 462
ENVIRO.USER_CODE_TIMER_STOP 463
ENVIRO.USER_GLOBALS 457
ENVIRO.USER_PARAMETERS 458
ENVIRO.UUT 454
ENVIRO.WHITE_BOX 455
environment
- build options 104
- building new 114
- closing 136
- constructor 19
- coverage type 104-105
- deleting 147
- editing 136
- files created during building 117
- loading 104
- meaning of 15
- naming 104
- opening 81, 135
- rebuilding 146
- rebuilding range data for 153
- regression test, creating 138
- renaming 136
- scripting language 454
- select parent library directory 107
- sending to Tech Support 50
- setting working dir before loading script with relative paths 104
- un-deleting 147
- unit(s) under test 108
- updating 136
- user code 391
- whitebox 104
- whitebox option 105

environment builder log file 118
Environment menu
- Configure Stubs 391, 417
  - Clear 422
- Rebuild Environment 146
- Rebuild range data 153
- Rebuild Range Data 223
- Recompile
  - Automatic 149
  - Create Script 149
  - Instrumented Code 149
  - Run Script 152

Relink 145, 153
Scripting
Create 102, 148
Run 147-148
Update Environment 136
User Code 391
Clear 405
Compile 405, 422
Edit 396
View
Compile Errors 117
Environment Build Log 118
Link Errors 118
Metrics Report 329, 344
Overview 115
Test Case Management Report 273, 329
Undefined Entities Log 118
environment name 104
duplicate name 104
environment script
opening 81
environment scripts
saving user code in 457
environment user code in designated uut 132
environment variables
setting 488
VCAST_ALTERNATE_REPORT 488
VCAST_APEX_COPY_FILES 488
VCAST_DISABLE_ADA_ARRAY_RANGE_CHECKING 488
VCAST_DISABLE_FILE_DELETE 488
VCAST_GNAT_HIE_DASH_E 488
VCAST_GNAT_LD_OPTIONS 488
VCAST_IGNORE_COMPILE_ERRORS 489
VCAST_IGNORE_FILE_EXISTS_CHECK_FOR_COVERABLE 489
VCAST_IGNORE_TYPEMARKS 489
VCAST_LEGACY_CFG_FILE_INHERITANCE 489
VCAST_MAX_INSTRUMENTATION_LINE_LENGTH 489
VCAST_NEVER_STUB_LIST 456, 489
VCAST_NO_STANDARD_PKG_USAGE 489
VCAST_NO_USER_EXCEPTION_SUPPORT 490
VCAST_OLD_ACTIVADA 490
VCAST_PARENT_LIBRARY 455, 490
VCAST_RAVEN_HEAP_SIZE 490
VCAST_RAVEN_STDOUT 490
VCAST_RECURSION_DEPTH 490
VCAST_REGRESSION_COMMAND 490
VCAST_RPTS_PRETTY_PRINT_HTML 490
VCAST_S68_LOCATION 490
VCAST_TIC_INDICATES_MACHINE_CODE 490
VCAST_TORNADO_OLD_SCRIPT 491
VCAST_USE_REMOVE_IMPORTS 491
VCAST_USER_CONSTRAINTS 208
VCAST_WHITEBOX 491
VCV_EDIT_LINE_NUMBER 491
VCV_ENVIRONMENT_DIR 491
VCV_ENVIRONMENT_NAME 491
VECTOR_LICENSE_FILE 492
VECTORCAST_DIR 492
environment wizard
choose compiler 102
choose UUTs & stubs 108
load an existing environment 103
load environment script 104
name environment 103

specify custom stubbing interactively 112
unit appendix user code 114
update environment 104
user code 113
user globals 113
user params 113
equivalence matrices 354
event
defined 16
event limit 287, 313
EVENT_LIMIT 287, 313, 474
examples
of environment user code 405-406, 409
user code 414
execute all command 503
execute commands, CLICAST 263
execute option
automatically clear test user code 289
fail if no expected return 293
executing compound test cases 261
executing test cases 19, 255
executing test cases with your debugger 264
execution status viewer 57
exiting 25
exiting VectorCAST 25
expand all subprograms 340
expanding arrays 198-200
expected results
range of values 476
unused” 259
using ANY in list of values 477
expected user code 414
expected user code comparisons
width of 258
expected values
any (placeholder) 209-210, 218
defined 178, 182
entering as a list 183
entering as a range 182
entering as a scalar 182
range of 208, 217
set expected to actual 178, 268
export test scripts 219
external browser
setting up 64
using when viewing html files 269
external editor
scrolling to first line of function 63
setting up 61
using in user code 395
using when viewing text files 269
fail if no expected return 293
fail on unexpected exceptions 289
fail on unexpected signals 293
File menu
Close 42, 45
Close All 45
Close Environment 136
Delete Environment 147
Exit 25
New 101
new document 81
Open 81, 136
Print 83
Print Setup 82
Recent Environments 135-136
Save 82
Save All 82
Save As 82
Set Working Directory 99, 136
file version command 303

files
.CFG 242
.vcast-qt 35, 65, 67
AALINKER.LIS 118, 447
configuration 242
created by environment 117
environment_name.bat 138
environment_name.cvr 138
environment_name.env 138, 147
environment_name.sh 138
environment_name.tst 138
exporting scripts 219
GLOBALS.C 457
importing 395
importing scripts 220
opening 81
opening using right-click menu 40
saving 82
saving automatically 67
scripting log 220
TEST_DELETE.SAV 159-160
user code 396
USER_CODE_VCAST 405
viewing in external editor 61
filter
parameter tree 93
test case tree 88
filter Metrics report 296
find
tear off menu 88
find banner 85
finding text
in a text file 85
FLEXlm 444
floating point digits of precision 303, 312
floating point tolerance 288, 313
FLOATING_POINT_TOLERANCE 288, 313
fonts
changing for coverage output 369
default for coverage output 369
force Ada83 conformance 135
full dot notation 208
gcc 132
generic packages 190
global data
about 182
initializing in INIT test case 162
global objects 15, 474
global values
when to display 298, 317
GLOBALS_DISPLAY 298, 317
GNAT version 5 or greater 130
gnatbind 132
gnatlink 132
Gnatmake 132
gridlines in test case, showing/hiding 65
groups
cascading 42
closing 41
closing all 45
explained 40
opening 41
switching between 41
header and footer 83
Help menu
About 50
Available Licenses 48
Diagnostics
Create Diagnostic Report 252
Troubleshooting 252
Tech Support 50
Tech Support 50

User Guides 49
hex notation for unprintable chars 301
hexadecimal
display integer results 312
import test script 220
importing
contents of file 395
importing coverage results 371
include directory
adding to an environment 136
incremental rebuild
incremental rebuild report 322
perform 321
industry mode
automotive 27
avionics 27
changing 30
changing from branch, basis path or MC/DC 30
default 27
industrial 27
medical 27
railway 27
reports 29
setting the mode 27
status bar 29
INIT test cases
using to initialize global data 162
Initialize button 331
initializing global data
in INIT test case 162
inlined subprograms 128
input user code 413
input values 15
<<keep>> 183
defined 178, 182
entering as a list 183
entering as a range 182
entering as a scalar 182
Insert button
in List tab 216
instrument logical operations in assignment statements 363
Instrument Logical Operations in Function Calls 364
instrumented code
recompiling 149
job status viewer 54
jobs monitor 52
execution status viewer 57
job status viewer 54
keyboard navigation
Parameter Tree 180
shortcuts 45
Test Case Tree 156
Level A coverage 330
Level B coverage 327, 330
license messages 444
licenses
determining which your organization has 48
link errors 118
linker errors
resolving 118
list
<<keep>> 183
entering data as 183
list of values for expected results 477
List Values tab 183, 215
Load button 104
looping 475
map line selection 340
map test case 228
create test script 240
edit test script 240

export 228
import 228
mapping view 237
margins 83
maximize button 42
maximum MC/DC Conditions 362
maximum MC/DC Conditions, abort when exceeded 363
maximum number of test cases 157
maximum varied parameters 214, 476
MC/DC 124-126, 327-328
maximum conditions 362
maximum conditions, abort on 363
understanding analysis 353
viewing 245
MDI window
closing all files in 45
entering data directly into 179
Notes tab 182
Parameter Tree 179
member functions
making visible with Whitebox 105
message window
collapse 31
error tab 31
expand 31
undocking and docking 33
messages
<environment name> could not be created 447
<environment name> could not be deleted 447
<environment name> could not be opened 447
<environment> deleted successfully 450
<type mark> could not be found 448
Abnormal Termination of Compile and Link 446
after exporting coverage script 373
after importing coverage results 372
Building Driver 445
Building Harness for unit_name 445
Building Min Mid Max test cases 451
Building test case data 449
Building test case script file 449
Building test case script template 449
Building the CLI script 449
Can't change discriminant/Type has non-defaulted discriminants 449
Cannot Build Environment for
unit name 447
Cannot create min-mid-max or max test cases 451
Cannot Overwrite Environment 447
Clicast failed - parameter inconsistency 452
Clicast failed with unknown exception 453
Compile Failed, stubbed unit
unit name 446
Compile Failed, unit
unit name 446
Compiler not supported 452
Compiling unit name 445
Copying The Parent View For Testing 444
Could not build test history 452
Could not open environment 450
Creating environment script 449
Data interface could not be invoked 447
Data Program did not link 447
Deleting files from environment build 448
Determining basis paths 450
Driver could not be invoked 447
Driver Program did not link 447
Environment build but not compiled 447
Environment build but not linked 447

Environment build but run-time error exists 448
Environment built but run time error exists 444
Environment Built Successfully 445
Environment creation failed 447
Environment data not expanded 451
Environment Directory Creation Failed 446
Environment has not been compiled 451
Environment has unresolved compile errors 451
Environment has unresolved link errors 451
Environment must be specified on command line 452
Environment name is invalid on command line 452
Error compiling instrumented file 451
Error computing basis paths 450
Error initializing coverage 451
Field is not in active part of variant 448
FLEXlm Cannot find license file 444
FLEXlm Feature has expired 444
FLEXlm License failure 444
FLEXlm Licensed number of users already reached 444
Getting size and range data 450
Harness has run-time errors 451
Illegal numeric entry 448
Index expression error 448
instrumentation data line <number> 452
Instrumented Harness did not link 447
Internal Tool Error (Parser) - Write STR 446
Invalid environment name 450
Invalid test case name 449
Linking Environment 445
Linking Instrumented Harness 450
No change made to test data 448
No expected results exist 449
No files specified 452
No results exist 449
Null record 449
Parsing UUT body for whitebox conversion 445
Performing A Private Check-Out Of Source Files 445
Processing global objects 450
Processing parameters 450
Processing unit name 445
Processing UUT unit name 445
Processing whitebox data 445
Rebuilding instrumented harness 451
Running \ 450
Script building completed 449
Sorting instrumentation data line <number> 452
Source not Found
unit name 446
unit name (S) [(B)] 446
Source not found <unitname> 444
Stub Could not be Created for
name 446
Subprogram must be specified on command line 452
Subprogram name is invalid 452
Syntax Error in Unit
unit name' 446
Test case must be specified on command line 452
Test case name is invalid 452
Test execution failed 448
Type not supported 449
Updating aggregate coverage report 451
Updating coverage data 451
Value out of range 448

VECTORCAST_DIR Environment variable is not set 444
metrics report 344
METRICS_TOTAL_LEVEL_DISPLAY 296
min-mid-max 162
modifications
to source code and relinking 145, 153
multi-dimensional arrays 206
multiple value syntax 216
multiunit whitebox 106, 129, 134
multi-unit whitebox
using to manipulate private types 197
nested compound
double-click nested test 174
nested compound test case 173
drop of compound test into itself error 175
recursion detection error 175
nested compound tests 173
no change made to data 448
no components 449
NO_FILTERING 296
non-defaulted discriminants 187
notes
entering for test cases 182
notes section template 304
Notes tab 302
null strings
passing as input or setting as expected 184
number of data partitions 288, 314
NUMBER_OF_DATA_PARTITIONS 288, 314
numeric types 472
object oriented programming and testing 192
only show errors in script logs” 293
open a console for standard I/O 290
opening
environment 135
files, scripts, environments, etc. 81
opening test cases 157
options 120
Ada language specification 132
additional information about exception occurrence 130
allow stubs of Ada runtime 128
alphabetize parameters 180
alphabetize test cases 65
alternate direct array indexing 129
always display coverage in reports 296
animation coverage type 361
animation play speed 367
Apex mutual imports 129
ask before closing multiple tabs 67
assignment statements in MC/DC 363
automatically save test cases 67
blank cells text 305
buffered coverage I/O 361
build stubs for pragma imports 129
builder 127
disable direct array indexing 202
change application font 67
change background color 371
change fonts 369
change text color 370
combination testing 291, 314
compare expected before UUT call 301, 316
compile dumb stubs 128
compile generic stubs 132, 189
compile UUT after stubs 128
convert octal/hex-encoded strings to ASCII 301
coverage 360
default fonts 369
Instrument Logical Operations in Function Calls 364

VCAST_INSTRUMENT_PARAMETERS 364
coverage field width 362
coverage I/O types 361
coverage type 124
covered lines 369
default script header file 305
deprecated
DISPLAY_INTEGER_RESULTS_IN_HEX 318
RANGE_FILE_MAX_SIZE 135
VCAST_COMPACT 261
VCAST_EXPAND_THRESHOLD 135
VCAST_EXPECTED_USER_CODE_EACH_EVENT 261
VCAST_USE_APEX_LINK 135
detect unused expected user code 290
disable boolean cast in instrumentation 364, 368
display global data after 298, 317
display subprogram coverage 296
Do not allow recursive calls in harness 106
do not automatically generate basis paths 245
do not stub ancestors 128
do not use package STANDARD 134
environment user code in designated uut 132
event limit 287, 313
execute 285
floating point tolerance 288, 313
range check 191
expand scalar array elements 290
external browser 64
external editor 61
fail on no expected values 289
fail on unexpected exceptions 289
fail on unexpected signals 293
file version command 303
filter Metrics report 296
floating point precision 303, 312
force Ada83 conformance 135
GNAT version 5 or greater 130
harness precompile script command 131
hex notation for unprintable chars 301
ignore incomplete tests 294
instrument logical operations in assignment statements 363
maximum covered subprograms 365
maximum MC/DC conditions 362
maximum MC/DC conditions, abort when exceeded 363
maximum MC/DC expressions 366
multibyte characters 27
multi-unit whitebox 129
notes section template 304
number of data partitions 288, 314
only show errors in script logs 293
open a console for standard I/O 290
partially covered lines 369
preprocess gnat source files 134
range check 286
real-time coverage I/O 361
redirect standard output 288
report 294, 306
report complexity column width 310
report coverage results column width 311
report date column width 310
report format 308
report header 305, 308
report notes column width 311
report results column width 310
report subprogram column width 309

report testcase column width 310
report unit column width 309
run script after instrumentation 366
setting custom css 308
shift jis character encoding 27
show gridlines in test case 65
show only data with expected results 300, 316
show only events with expected results 301, 317
show version of VectorCAST in reports 302
sort Metrics report by directory 295, 298
strict test case importing 289
stub dependencies 123
suppress debug information 134
text report options 309
unconstrained string size 131
uncoverable lines 369
uncovered line indicator 367
uncovered lines 369
use as text file viewer 62
use DOS formatted files 62
use Gnatmake 132
use gprbuild 133
use naming 133
use static memory allocation on the target 365
verbose management report 302, 468
whitebox 127
wide-character support 131
wrap text in Notes 311
Options tab
for test cases 474
original source code line numbers 340
out-of-date units 106
out-of-range values
testing 191
output values 15
overloaded member functions
naming of 155
overloaded subprograms
working with 477
overview 15
PAGE_WIDTH 258
pairs status 329, 345
parameter dialog box 210
parameter tree 229
alphabetize parameters 180
defined 179
filter 93
filter indicators 93
keyboard navigation 180
parameter checkboxes 229
parameter user code example 414
parameters
entering data for 209
user code 391
parent library directory
selecting 107
partially covered lines 369
pasting text 84
performance testing
real-time vs. instrumented 326
pragma import 129
precompile script command 131
preprocess gnat source files 134
print
print preview 84
print preview 84
print setup
coverage 83
header and footer 83
margins 83

use backgrounds 83
Print Setup
reverse foreground and background 83
printing
files 83
setting up 82
private ancestors 129
protected type and task type testing 189
quitting 25
raising exceptions from stubs 184
range
entering data as 182
range check 286
range data
rebuilding 153
range of values for expected results 476
range of values for test creation 475
range testing 214
range values tab 213
RANGE_CHECK 286
ranges as expected results 214
reading data from database
user code example 406
real-time coverage I/O 361
rebuild environment 146
recompiling
automatic 149
create script 149
instrumented code 149
run script 152
record types 187
redirect standard output 288
redoing an edit action 84
regression scripts
creating 138
integrating with ClearCaseTM 144
post-processing automatically 142
regression test 138
relinking
test harness 153
remember last working directory 99
Remove All button
in List tab 216
Remove button
in List tab 216
removing
breakpoints 360
removing a custom tool 250
renaming
environment 136
renaming a test case 157
renaming test cases 158
Replace button
in List tab 216
report complexity column width 310
report components 76
layout template 78
report file 77
section file 78
report coverage results column width 311
report customization 68
report date column width 310
report format 308
report header 305, 308
report notes column width 311
report results column width 310
report subprogram column width 309
report testcase column width 310
report unit column width 309
reports
aggregate coverage report 343
controlling the test execution report 259

coverage summary 20
creating 20
diagnostic 252
execution history 20
expand report objects 259
expand report parameters 259
expanding information included 259
full 272
metrics 344
overview of environment 115
scripting log 220
summary of results 20
test case management 273, 468
test comparison 278
test data 20
viewing in external browser 65
requirements
displaying in report 468
requirements gateway
authentication tab 427
command line 547
configure export 442
example workflow 438
export 443
export tab 430
import requirements 440
import tab 428
initialize repository 438
link requirements 433, 441
requirements repository 427
requirements tab 431
run test case 441
script tab 432
set configuration options 439
set repository 438
supported subsytems 437
tracking requirement change impacts 434
using rgw 426
requirements gateway commands
configure set 547
export 548
import 548
initialize 548
requirement report 548
set gateway 549
testcase link 549
testcase mark as reviewed 549
testcase migrate_links 549
testcase report 550
Reset Expected button
in Parameter User Code tab 413
Reset Input button
in Parameter User Code tab 413
restoring deleted test cases 160
rgw
see requirements gateway 426
right-click menu
(Array) Properties 202
Close (tab) 42
Collapse Unused Children 202
Compound Only 173
copy/clear text in Message window 32
Delete 158-159
Delete, from a Compound 172
Deselect Coverage 333
Duplicate Test Case 158
Execute 255, 263
Execute with Debug 264
Expand All Array Indices 202
Expand First Array Index 202
find slot in test case tree 172
Import file contents 395

Insert Basis Path Test Cases 164
Insert Min Mid Max 162
Insert user code tag 393
Insert usercode tag 410
Invoke external editor 395
Move Down, in a Compound 171
Move Up, in a Compound 171
Open Source 161
Open Test Case 157
Open, from a Compound 171
opening files 40
Rename 157-158
Select Coverage 333
Show (tab) 45
toggle Message window 31
View Coverage 161
View Coverage, from a Compound 172
run new reports 79
from command line 79
from gui 79
run-time errors 444
Save and Link button 401, 421
save backup copies 159
Save button 102
save control flow 267
Save Only button 401
saving
files 82
test cases automatically 67
saving a file to another name 82
SBF
meaning of 16
scalar values tab 210
scientific notation
supported for real numbers 186
scripting 219
exporting for regression test 373
scripting utility
building test cases with scripts 81, 219
clear unapplied test script data 220
duplicate names in exported scripts 220
exporting scripts 219
importing scripts 220
opening scripts 81
setting test values 471
scripts
environment, creating 81
test case, creating 81
search
case sensitive 86
close 87
execute 86
match case 86
next instance 86
previous instance 86
repeat 87
search directory
adding to an environment 136
searching 85
segmentation violation
in execution results 166
set display theme 60
set expected values to actual values 178, 268
set working directory 99
setting the industry mode 27
shell script 138-140, 143, 490
shortcut keys 45
show only data with expected results 300, 316
show only events with expected results 301, 317
show version of VectorCAST in reports 302
SHOW_NOTES_IN_FULL_REPORT 303

smart stubs 16
sort Metrics report by directory 295, 298
sort test cases alphabetically 158
source code
    making modifications to and relinking 145, 153
source code file
    opening 81
specify custom stubbing interactively 112
standard I/O 290
STANDARD.BOOLEAN 134
STANDARD_ERROR 288
STANDARD_OUTPUT 288
starting VectorCAST 22
starting VectorCAST and opening environment 22
starting VectorCAST and opening options dialog 22
starting VectorCAST in C or Ada mode 24
starting VectorCAST, building and opening environment 22
start-up messages 444
statement coverage 326
Statement coverage 327
STATEMENT+BRANCH 330
Statement+Branch coverage 330
STATEMENT+MC/DC 330
Statement+MC/DC coverage 330
static functions
    making visible with Whitebox 105
status bar 40
strict test case importing 289
string type
    null 184
string types 187, 472
strings
    passing null 184
structure types 472
Stub by function
    meaning of 16
stub dependencies 123
    all (by prototype) 110
    choosing 110
    custom 110
    none 110
stub user code
    compiling 422
stubbing
    private ancestors 129
stubs
    do not stub ancestors 128
    generic packages 189
    in/out parameters 185
    meaning of 16
    protected types 189
    raising exceptions 184
    special kinds of 189
    task types 189
SUBPROGRAM_DETAIL 297
subprograms
    inlined 128
    overloaded 477
    viewing coverage of 161
suppress debug information 134
system address types 188
T
    indicating a branch covered 336
tab delimited file 228
tabs
    closing 42
    closing all 45
    closing the current 42, 45
    showing from behind 45

task type and protected type testing 189
tech support
    troubleshooting VectorCAST 252
technical support
    contacting 50
    FTP environment to 50
template
    for exported test scripts 305
    for Notes tab 304
    for user code 304
template file
    creation 229
    csv 229
    tab delimited 229
templates 221
test case
    (MAP) 229
    create from CSV Map 238
    CSV File 228
    import/exprt map test case 228
    map test 228
    meaning of 16
    tab delimited file 228
test case execution
    detect unused expected user code 290
    expand scalar array elements 290
    fail on unexpected signals 293
    ignore incomplete tests 294
    no expected values 289
    strict importing 289
test case notes
    displaying in report 468
    in MDI window 182
test case options
    display integer results in hexadecimal 312
test case script
    opening 81
Test Case Tree 32
    filter 88
    navigation 156
test case tree
    changing search term for filter 89
    coverage viewer 333
    filter indicators 89
    filter test case reports 90
    filter to selectively apply coverage 92
    filtering to export test scripts 91
test cases
    aborting execution 264
    automatic range testing 475
    automatically naming 157
    batch execute all 263
    building 19
    building compound 167
    building from basis path analysis 351
    compound only 173
    creating INIT 162
    creating regression test for 138
    creating simple 157
    definition of compound 156
    definition of simple 156
    deleting 158-159
    deleting from Compound 172
    deleting test case part of compound 159
    duplicating 158
    entering data 178
    entering data for 182
    executing 19, 255
    executing compound 261
    executing with debugger 264
    find slot, from Compound 172

generating automatically for full basis path 164
management report 273
maximum number of 157
Min-Mid-Max 162
nested compound 173
opening for editing 157
opening, from Compound 171
rearranging order of, in Compound 171
renaming 157-158
restoring deleted test cases 160
saving automatically 67
saving control flow 267
scripting language 465
selecting multiple 156
selecting subprogram 157
sorting alphabetically 158
specifying input values in scripting 471
specifying options in scripting 474
types of 156
user code 391
viewing 268
viewing coverage of 161
viewing coverage of, from Compound 172
viewing execution results 270
Test Compile button 395, 399, 419
Test Compile Expected button 411
in Parameter User Code tab 413
Test Compile Input button 411
in Parameter User Code tab 413
test data summary 274
test driver
meaning of 16
test execution
animated 356
test execution report
controlling data displayed 179, 182, 259
test executions
delay between 170
specifying the number of iterations of 168
test harness
components 16
creating a recompile script 149
editing of 243
meaning of 15
recompiling 149
recompiling without reinstrumenting 149
relinking 153
running a recompile script 152
test harness architecture 192
Test menu
Batch Execute All 263
Control Flow
Clear 267
Save 267
Delete All 159
Execute 255
Scripting
Duplicate names in exported scripts 220
Export Script 219
Import Script 220
Template 221
User Code
Add All 412
Clear All 412
View 268
Aggregate Coverage Report 343
Execution Results 270
Full Report 272
Raw Data Set 285
Scripting Log 220

Test Case Data 269
test menu
coverage data used in reports 268
test requirements 182
test script header template 305
Test Script Log 220
test scripts
automating maintenance 223
conversion processing 228
convert multiple solutions 224
convert rebuilt environments with no test cases 226
delete translation 226
language 219
test script converter 223
view messages from last translation 227
view test script log 226
view test script maintenance difference listing 227
test values
spanning multiple lines 477
TEST.ADD 465
TEST.ATTRIBUTES 465
TEST.AUTOMATIC_FINALIZATION 465
TEST.AUTOMATIC_INITIALIZATION 466
TEST.BASIS_PATH 466
TEST.COMPOUND_ONLY 466
TEST.CSV_NAME_COL 466
TEST.CSV_NOTES_COL 466
TEST.END 466
TEST.EXPECTED 466
TEST.EXPECTED_USER_CODE 467
TEST.FLOATING_POINT_TOLERANCE 467
TEST.FLOW 468
TEST.IMPORT_FAILED 468
TEST.NAME 468
TEST.NEW 465
TEST.NOTES 468
TEST.REMOVE 465
TEST.REPLACE 465
TEST.REQUIREMENT_KEY 469
TEST.SLOT 469
TEST.STUB_EXP_USER_CODE 470
TEST.STUB_VAL_USER_CODE 470
TEST.SUBPROGRAM 470
TEST.UNIT 471
TEST.VALUE 471
TEST.VALUE_USER_CODE 471
TEST_DELETE.SAV 159-160
testcase
options
COMPARE_BEFORE_UUT 301, 316
EVENT_LIMIT 287, 314
FLOATING_POINT_DIGITS_OF_PRECISION 303, 312
FLOATING_POINT_TOLERANCE 288, 313, 468
GLOBAL_DATA_DISPLAY 299, 318
NUMBER_OF_DATA_PARTITIONS 288
SHOW_ONLY_DATA_WITH_EXPECTED_RESULTS 300, 317
SHOW_ONLY_EVENTS_WITH_EXPECTED_RESULTS 301, 317
testing custom tool 248
text colors
changing for coverage output 370
tiling groups 43
timing analysis
user code example 405
toolbar
coverage animation 358
New 101
Open button 81, 135

Print button 83
print preview 84
Redo button 84
Save All button 82
Save button 82, 401, 404, 421
Undo button 84
Tools
MC/DC
View Analysis Report 245
options 242
Tools menu
Basis Path
Edit Script 245
View Analysis Report 244, 351
Coverage
Delete Imported Results 376
Disable 333
Enable 333
Import Script 376
Custom Tools 245
Edit Harness Source 243
MC/DC
View Equivalence Matrices 354
menu selections 243
Options dialog 242
troubleshooting
<<unknown>> values in template 222
environment data not expanded 222
segmentation violation caused by Min, Mid, or Max values 166
spaces in working directory 100
suppressing some statement in MC/DC coverage 329
VectorCAST 252
true branch 336
types
character 472
enumeration 472
numeric 472
string 472
structure 472
unconstrained array parameters
working with 207
unconstrained array size 206
unconstrained arrays 206
unconstrained string size 131
uncoverable lines 369
uncovered line indicator 367
reinstrument 368
uncovered lines 369
undoing an edit action 84
uninstrument coverage 332
Unit Under Test
choosing 108
meaning of 16
UNIT_DETAIL 297
units
viewing coverage of 161
Up button
in List tab 216
updating an environment 136
use as text file viewer 62
use DOS Formatted Files 62
use Gnatmake 132
use gprbuild 133
user code
add all 412
clear all 412
configure stubs
saving and linking 421
Stub Dependency column 419

test compiling 419
editing 396
entering for test cases 410
environment
additional unit bodies 398
additional unit specs 398
clearing 405
compiling 405
data 397
global data 402
harness init 397
harness term 398
saving and linking 401
stub entry 398
stub exit 398
stub processing 398
test case init 397
test case term 398
test compiling 399
UUT timer start 398
UUT timer stop 398
header 397
importing contents of a file 395
order of execution 391
parameter example 414
parameter, expected 414
parameter, input 413
reading from database example 406
stub
clearing 422
entering 418
timing analysis example 405
types of 391
user globals 408
user parameters example 409
user params 407
using external editor 395
using external editor for 64
User Code tab
for a parameter 412
user code template 304
user globals
adding 402
changing the default 403
default 402
how to edit 403
introduction 402
saving and linking 404
test compiling 403
User Globals in environment scripts 457
user guides
HTML 49
PDF filenames for 49
user parameters
user code example 409
user params
how to edit 408
how to use 409
USER_CODE_VCAST 405
UUT
meaning of 16
V(g) 244, 351
variant record types 187
non-defaulted discriminants 187
vcast use naming 133
VCAST_ALTERNATE_REPORT 488
VCAST_APEX_COPY_FILES 488
VCAST_CONSOLE_REPORT_FORMAT 495
VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII 301

VCAST_COVERAGE_IO_ANIMATION 361
VCAST_COVERAGE_IO_BUFFERED 361
VCAST_COVERAGE_IO_REAL_TIME 361
VCAST_CUSTOM_REPORT_FORMAT 308
VCAST_DETECT_UNUSED_EXPECTED_UC 290
VCAST_DISABLE_ADA_ARRAY_RANGE_CHECKING 488
VCAST_DISABLE_FILE_DELETE 488
VCAST_DISPLAY_EMPTY_COVERAGE 296
VCAST_DISPLAY_FUNCTION_COVERAGE 296
VCAST_DO_COMBINATION 291, 314
VCAST_DO_MCDC 329
VCAST_DONT_DO_MCDC 329
VCAST_DONT_INSTRUMENT_END 332
VCAST_DONT_INSTRUMENT_START 332
VCAST_EXPECTED_BEFORE_UUT_CALL 301, 316
VCAST_FILE_VERSION_COMMAND 303
VCAST_FLOAT_PRECISION 303, 312
VCAST_GNAT_HIE_DASH_E 488
VCAST_GNAT_LD_OPTIONS 488
VCAST_IGNORE_COMPILE_ERRORS 489
VCAST_IGNORE_FILE_EXISTS_CHECK_FOR_COVERABLE 489
VCAST_IGNORE_INCOMPLETE_TESTS 294
VCAST_IGNORE_TYPEMARKS 489
VCAST_INSTRUMENT_PARAMETERS 364
VCAST_INSTRUMENT_PARAMETERS VCAST_INSTRUMENT_PARAMETERS 364
VCAST_LEGACY_CFG_FILE_INHERITANCE 489
VCAST_MAX_INSTRUMENTATION_LINE_LENGTH 489
VCAST_NEVER_STUB_LIST 456, 489
VCAST_NO_STANDARD_PKG_USAGE 489
VCAST_NO_USER_EXCEPTION_SUPPORT 490
VCAST_NOTES_SECTION_TEMPLATE 304
VCAST_OLD_ACTIVADA 490
VCAST_OLD_STYLE_MANAGEMENT_REPORT 302
VCAST_PARENT_LIBRARY 455, 490
VCAST_RAVEN_HEAP_SIZE 490
VCAST_RAVEN_STDOUT 490
VCAST_RECURSION_DEPTH 490
VCAST_REGRESSION_COMMAND 490
VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH 310
VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH 311
VCAST_RPTS_CUSTOM_CSS 308
VCAST_RPTS_DATE_COLUMN_WIDTH 310
VCAST_RPTS_EMPTY_DATA_STRING 305
VCAST_RPTS_HEADER 305, 308
VCAST_RPTS_NOTES_COLUMN_WIDTH 311
VCAST_RPTS_PRETTY_PRINT_HTML 490
VCAST_RPTS_RESULT_COLUMN_WIDTH 310
VCAST_RPTS_SHOW_VERSION 302
VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH 309
VCAST_RPTS_TESTCASE_COLUMN_WIDTH 310
VCAST_RPTS_UNIT_COLUMN_WIDTH 309
VCAST_RPTS_WRAP_NOTES 311
VCAST_S68_LOCATION 490
VCAST_SCRIPT_EXPAND_ARRAYS 290
VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS 293
VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES 300, 316
VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES 301, 317
VCAST_SHOW_STDOUT_CONSOLE 290

VCAST_SORT_METRICS_RPT_BY_DIR 295, 298
VCAST_STRICT_TEST_CASE_IMPORT 289
VCAST_TESTCASE_FAIL_ON_NO_EXPECTED 289
VCAST_TIC_INDICATES_MACHINE_CODE 490
VCAST_TORNADO_OLD_SCRIPT 491
VCAST_UNEXPECTED_EXCEPTIONS_FAIL 289
VCAST_UNEXPECTED_SIGNALS_FAIL 293
VCAST_USE_REMOVE_IMPORTS 491
VCAST_USER_CONSTRAINTS 208
VCAST_VERBOSE_MANAGEMENT_REPORT 302
VCAST_WHITEBOX 491
VCV_EDIT_LINE_NUMBER 63, 491
VCV_ENVIRONMENT_DIR 491
VCV_ENVIRONMENT_NAME 491
VECTOR_LICENSE_FILE 492
VectorCAST
  exiting 25
  starting 22
  starting and opening environment 22
  starting and opening options dialog 22
  starting in C or Ada mode 24
  starting, building and opening environment 22
  troubleshooting 252
VECTORCAST_DIR 444, 492, 494
verbose management report 302, 468
version control 138
version number
  obtaining 50
View menu
  Dock control 31-32
viewing
  execution results 270
  full report 272
  raw data file 285
  test cases 268
  test comparison report 278
visible subprogram
  meaning of 16
whitebox 127
  changes made to source 105
  using to manipulate private types 197
whitebox test utility
  circular references 106
  out-of-date units 106
  restrictions and limitations 106
wide-character support 131
Window menu
  Cascade 42
  Group 41
  Tile 43-44
windows
  closing all 45
windshell script
  opening 81
working directory 136
  in status bar 40
  invalid 99
  remembering last 99
  setting 99
  uses of 99
  when starting from command line 99
  when starting from Start menu 99
wrap text in Notes 311