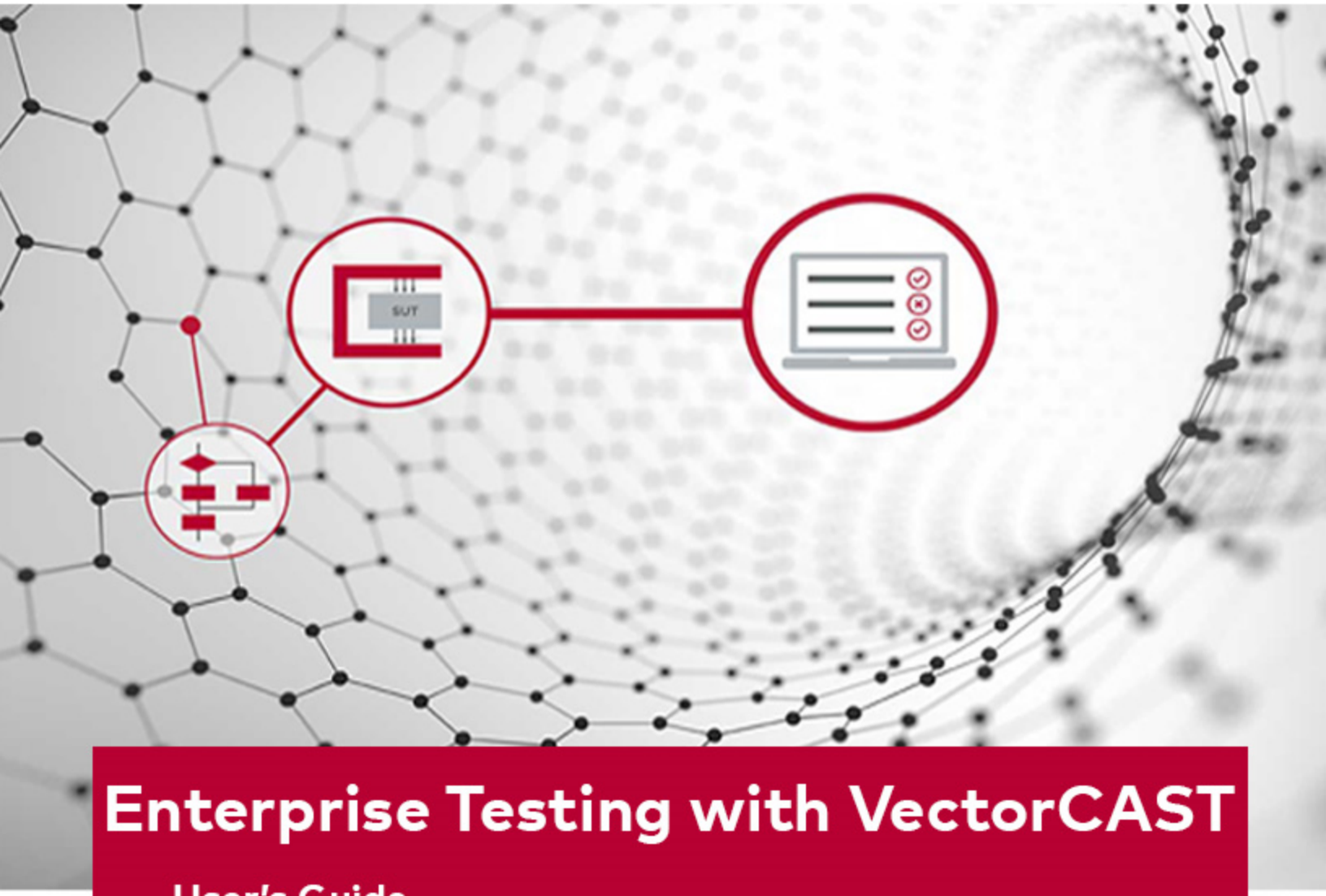
VECTOR



Enterprise Testing with VectorCAST

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: 6/8/2026, 8:19 PM

Rev: d2bd547

Part Number: Enterprise Testing with VectorCAST User's Guide for VectorCAST 2026 SP2

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

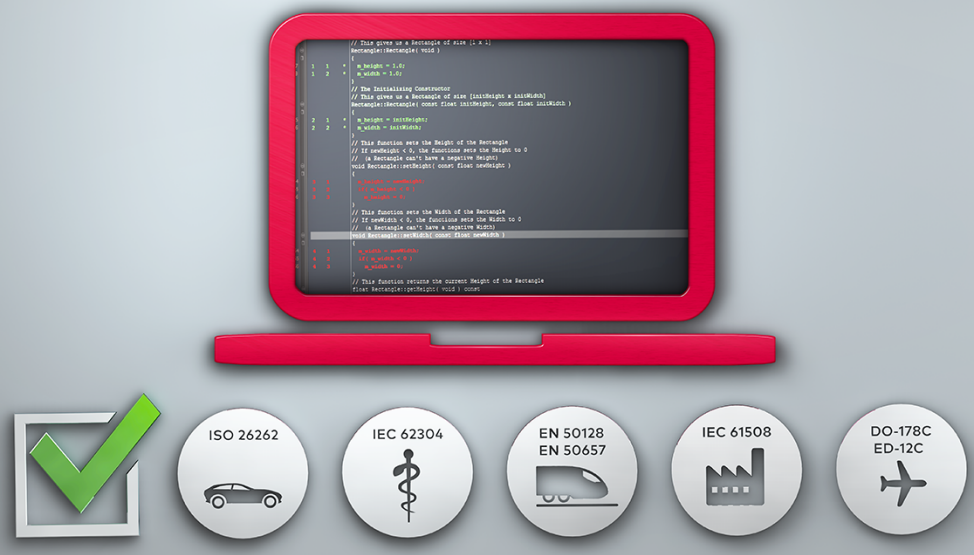
Vector reserves the right to modify any information and/or data in this user documentation without notice. Vector disclaims all liabilities for the completeness or correctness of the contents and for damages which may result from the use of this documentation. This documentation nor any of its parts 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.

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



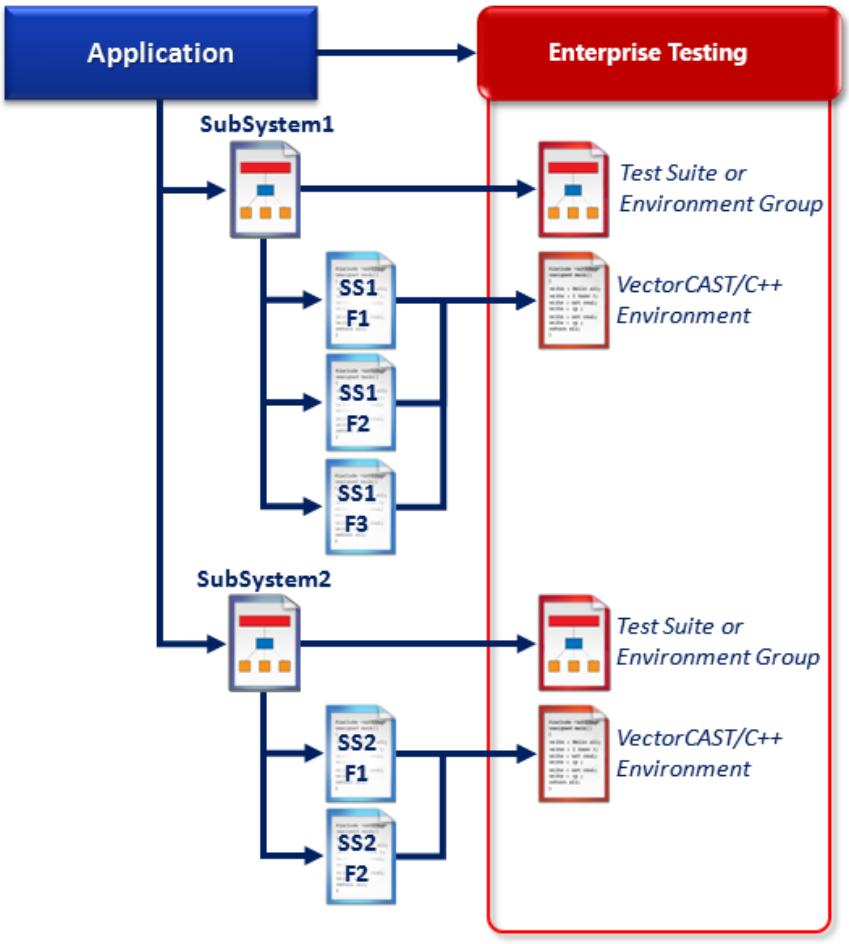
Introduction

Enterprise Testing with VectorCAST

This User Guide describes in detail how to use the extended testing functionality of Enterprise Features such as multi-configuration testing, change-based testing, and continuous and parallel testing.

A VectorCAST Project is a collection of VectorCAST unit and system test environments that allows tests and test results to be shared across the enterprise, providing a single point-of-control for all unit- and integration-test activities. At-a-glance logs, summary reports, and color-coded pass/fail criteria highlight the status of each test within the regression suite.

Within an Enterprise Testing Project, Test Environments can be grouped into larger “Environment Groups” and “Test Suites.” Environments can be members of multiple Environment Groups, and Environment Groups can be assigned to multiple Test Suites. This enables users to structure their VectorCAST project to match the architecture of their application. For example, the application sub-systems will map onto Enterprise Testing Test Suites or Environment Groups, and individual source files will map onto Enterprise Testing Test Environments.

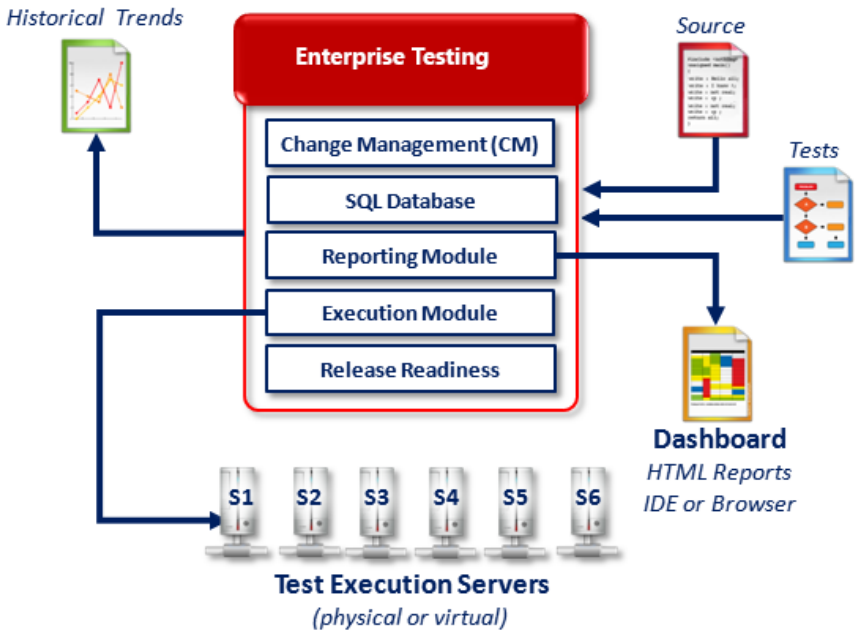


Because Environment Groups and Test Suites can be easily duplicated, the same tests can be run using various source baselines, on different host platforms, or with a different compilers.

The integrated SQL database and graphing facility in Enterprise Testing enables users to view historical trend data for an individual software component, or any group of software components. This makes it easy to analyze regression trends across the software-testing life cycle.

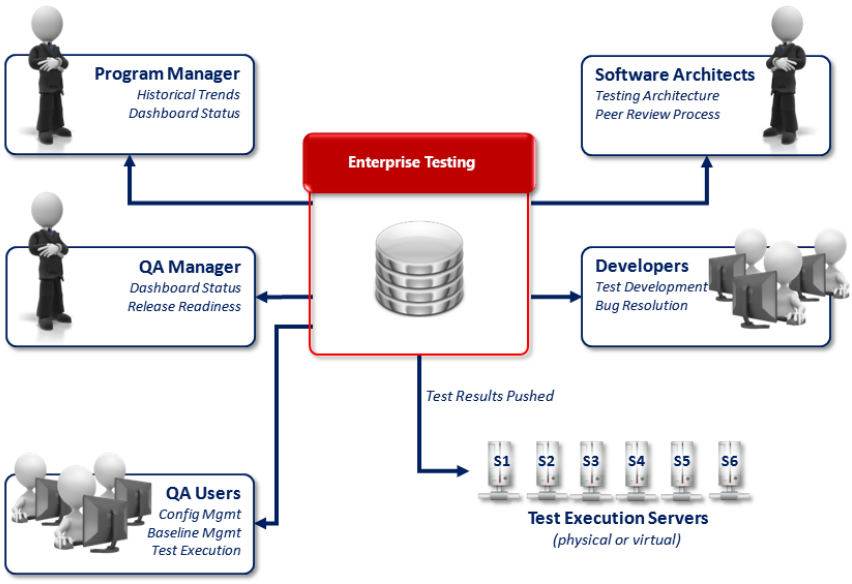
The Status Panel and HTML reports enable users to view the current status of each test case. Data is automatically recorded for build status and duration, test execution status and duration, and code coverage achieved. Using the integrated Python interpreter, additional comparisons can be added for each component and a column for the comparisons added to the report. For example, a user might want to compare the test execution time to some threshold or might want to run a static analysis tool on the source files in the Enterprise Testing project.

The Test Execution Module allows tests to be distributed across multiple physical or virtual servers, with all results from the remotely executed tests incorporated in the Enterprise Testing Database.



The VectorCAST project allows multiple users simultaneous access to run tests and to view status and results. The entire development team can use Enterprise Testing.

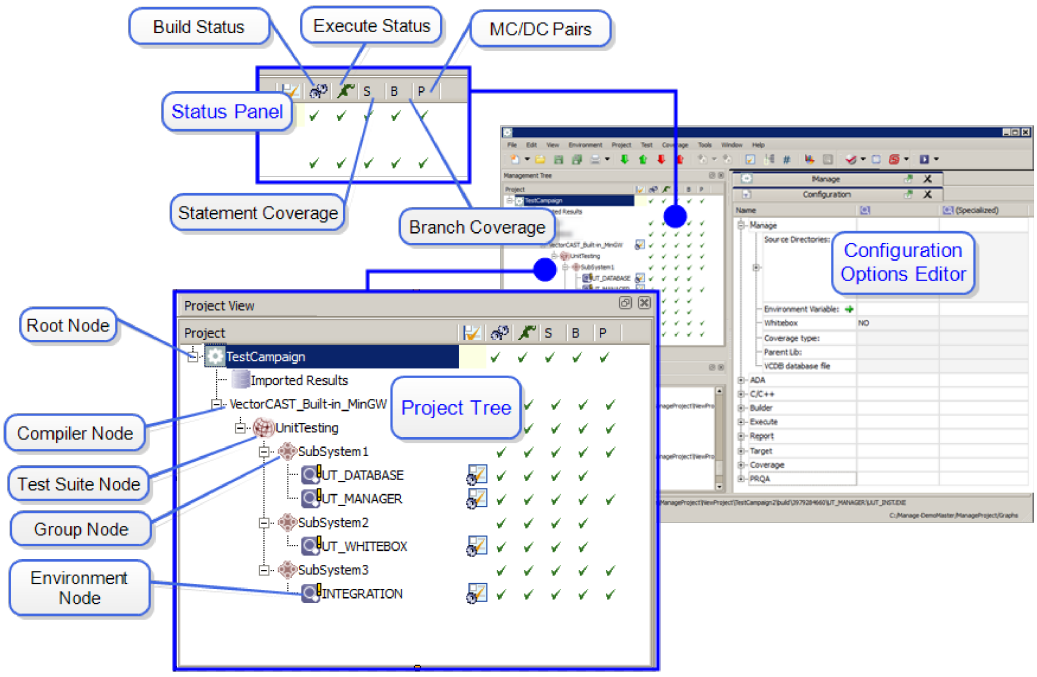
- Program Managers use the high-level reports and graphs to track testing progress.
- QA Managers use the status panel and reports to track release readiness.
- QA Engineers control multiple test configurations and test scheduling.
- Software Engineers create new tests, and identify and fix bugs.
- Software Architects control the over-all design of the VectorCAST project as well as controlling the peer review and change control process.
- Engineers use the tool to easily design test campaigns and monitor release readiness.



The VectorCAST Project

A VectorCAST *project* is the set of all files, directories and test components managed and controlled by Enterprise Testing. A VectorCAST project is made up of the project file (*yourprojectname.vcm*) and the project directory. The project file is an XML file and is typically kept under source code control. The project directory is the directory in which VectorCAST stores the files to build and execute environments and the SQL historical database. It is named the same as the project.

A VectorCAST project can be created from a set of existing VectorCAST Test Environments, using the New Project Wizard, or as an empty project. The interface of a typical VectorCAST project is shown below.



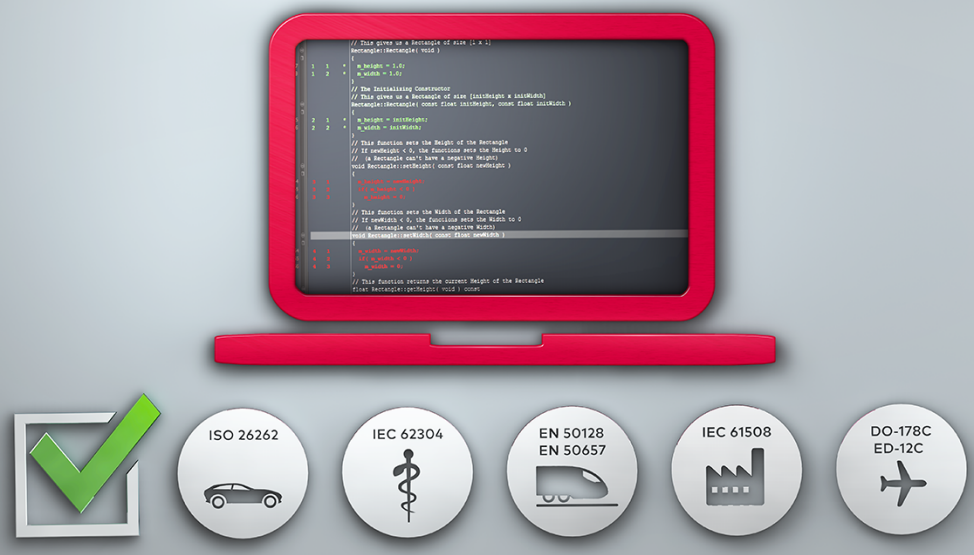
The *project tree* for the VectorCAST project is displayed in the upper left hand corner of the application window. It depicts a hierarchical view of the components of the project as a set of nodes on a tree. The following is a list of Project Tree nodes from the root or top-most level to the leaves or bottom-most level:

- The *root node* is the overarching container that contains all the other nodes within a project. There is a single root node for each project. Its name reflects the name of the project.
- The *compiler node* contains the compiler and target-specific options for a set of tests. For example, the compiler node might be Visual Studio, the GNU Compiler, or GreenHills Multi. The compiler node carries the default VectorCAST compiler settings for that compiler.
- The *test suite node* encapsulates and groups a set of environments that contain a similar set of common options such as coverage or whitebox testing. Configuration options specific to the test suite can be set at this level.
- The *group node* is a conceptual grouping for the test environments for a given test suite. Groups do not have any configuration associated with them; they are just a way to organize and group test environments within a test suite.
- The *environment node*, the lowest level node in the project tree represents a VectorCAST test environment.

After an environment is built and executed, the build status, execution status, and coverage achieved for each environment is displayed in the *Status Panel* (to the right of the Project Tree).

The results from each test run are saved into an SQL Database to allow team members to see historical test and code coverage trends.

Additionally, a full Python interface to the underlying data is provided for easy integration with other development tools.



Work Flow Scenarios

Using Enterprise Testing with VectorCAST in Your Organization

Enterprise Testing with VectorCAST can be used by a single developer or used across an entire enterprise. Using the Enterprise Testing platform for testing allows for:

- Easier team collaboration and sharing of tests
- Easy deployment of the same tests to multiple configurations
- Change-Based Testing to shorten test cycles
- and Parallel Testing to reduce total test time.

This section provides an overview of how to configure VectorCAST for some common work flow scenarios for a project team consisting of the following members:

- Bob - Architect
- Carlos - Developer
- Eric - Developer
- Beth - QA Engineer

The root directory of the VectorCAST project described in the following work flows is referred to as `$PROJECT_BASE`.

For all of these examples we will use VectorCAST/C test environments that are based on the VectorCAST tutorial source code for C. The source code for the following examples is located in the VectorCAST installation directory: **`$VECTORCAST_DIR/Tutorial/c`**. Because the examples are based on the VectorCAST/C tutorial code, you should be able to easily duplicate the processes.

These examples are not intended to be step-by-step tutorials, but rather an overview of the tool features that are applicable to each work flow. All examples were run using Windows.

Note: Environment Groups are no longer created. If you are migrating a Project created in an older version, the Group level will still appear. If the Test Suite you are working with does not have any groups, then you can substitute the Test Suite for the Group in the directions listed below.

Single User Setting Up an Initial VectorCAST Project

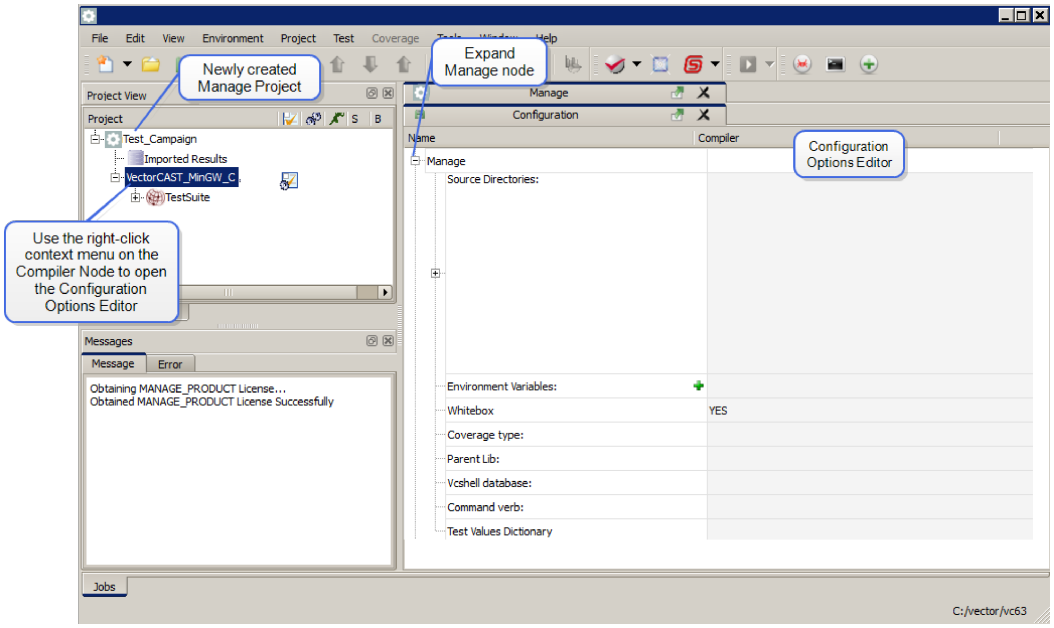
A typical Enterprise Testing use case scenario is a single user setting up an Initial VectorCAST Project. This scenario demonstrates building an empty VectorCAST Project and adding the environment for UNIT_TEST_MANAGER.

For this example the Project Architect, Bob, creates a new VectorCAST project to formalize the testing configuration for the Test_Campaign project. The following sections describe the work flow.

Create An Empty Project

To begin, select **File => New => VectorCAST Project => Empty Project** from the Menu Bar. Enter the name Test_Campaign in the Project Name field, select **C/C++ > VectorCAST MinGW > C** from the Compiler drop down menu and enter the path to the Base Directory. For this example, the base directory path is `C:/VCAST/tutorial/c`. Select the **Create** button.

The Project Tree for the Test_Campaign project opens. Note that the compiler node is labeled with the VectorCAST MinGW_C compiler selected when initially creating the project.

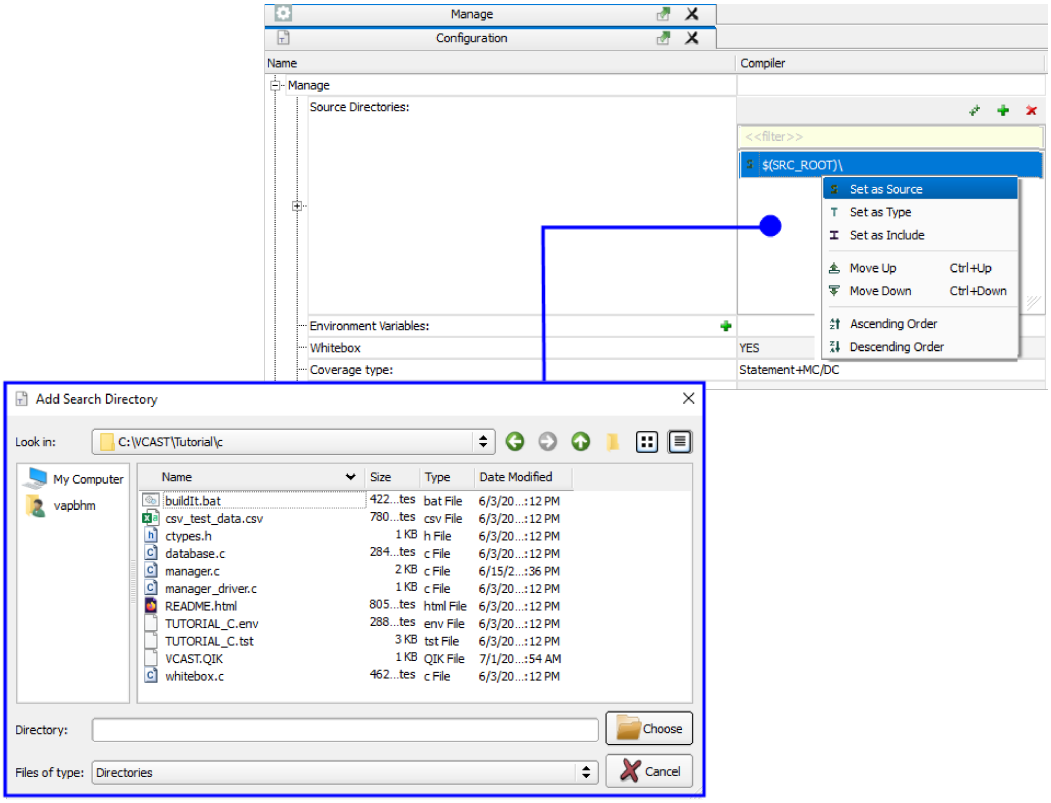


Customize the Configuration

Before building any test environments, customize the build configuration for the project. Two options that are typically updated are the source code search directories and the code coverage type. To set these options, open the Configuration Options Editor by right-clicking on the compiler node and select **Open Configuration** from the context menu. Bob finds using the right-click context menu on any Project root, Platform, Compiler or Test Suite node to be a quick way to open the Configuration Options Editor and configure a project.

Expand the Manage node to set the Source Directory. Open the **Add Search Directory** browser dialog, navigate to the location of the Source Directory and click the **Choose** button. The directory

path displays in the Configuration Options Editor. Right-click on the path and select **Set as Source** from the context menu. Save the changes.



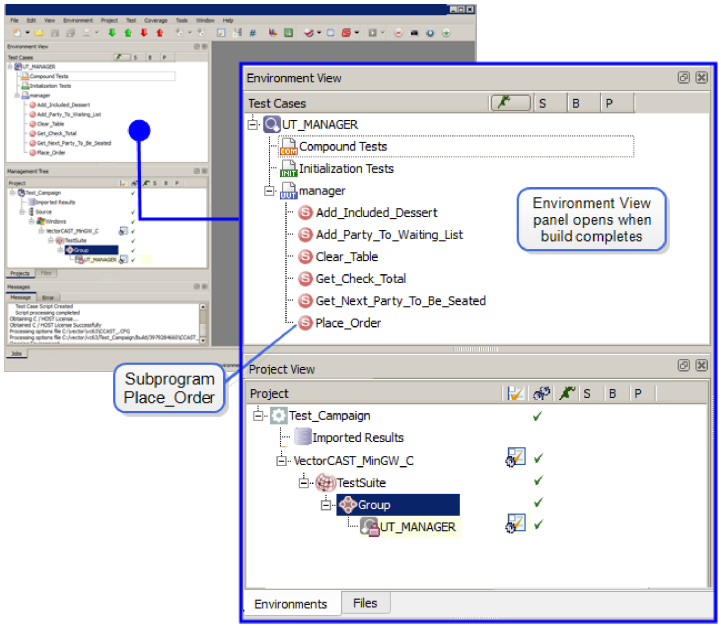
Select **Statement + MC/DC** from the Coverage Type drop-down menu to update the Coverage Type for the Test_Campaign project. Save the change.

Bob has now completed building a Configuration Node for the Test_Campaign project. This configuration will be used by all the members of the team, and at this point the project could be deployed to the developers. The developers would then create test environments using the configuration. However, Bob has a test environment that he wants to create before he shares the project with the team.

Create a New Environment

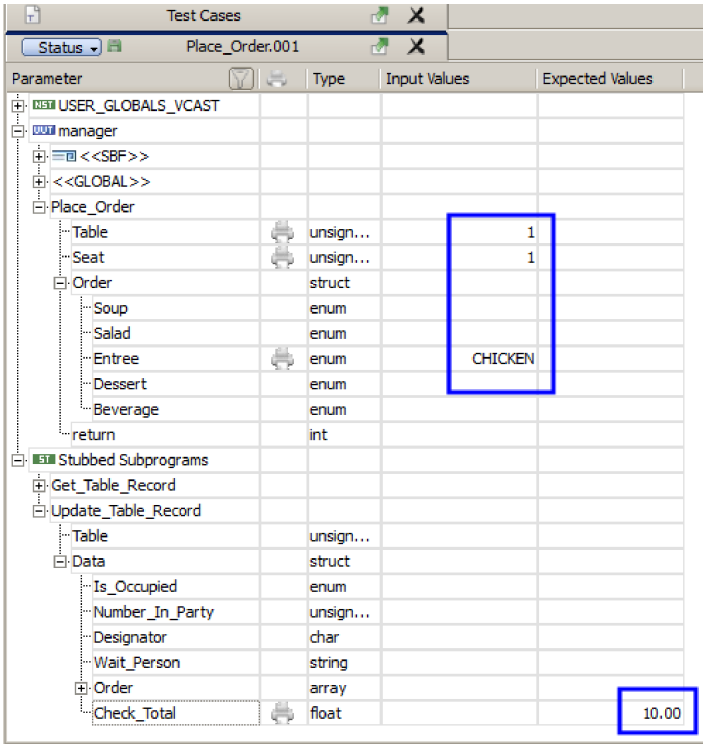
Right-click on the Group node in the Project Tree and select **Create Unit Test Environment > Interactive** from the context menu. Enter the Environment Name: **UT_MANAGER**. Note that the VectorCAST MinGW_C compiler, the Statement + MC/DC coverage type, and the search path specified in the Configuration Node are inherited and do not need to be entered here. Select **manager.c** as the UUT and select the **Build** button.

When the environment build completes the new environment is automatically opened in the Environment View panel to allow test case creation.



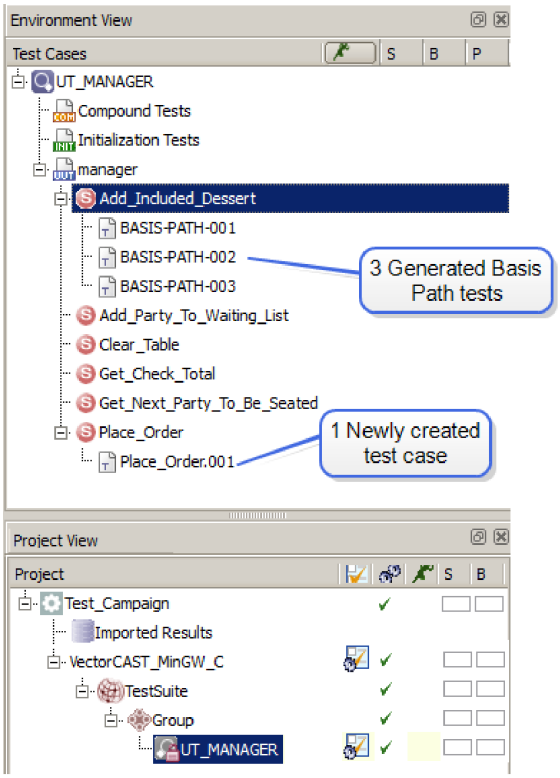
Add Test Cases

Right-click on the subprogram Place_Order and select **Insert Test Case** from the context menu. The test case Place_Order.001 is created. Enter the following input and expected values and save:

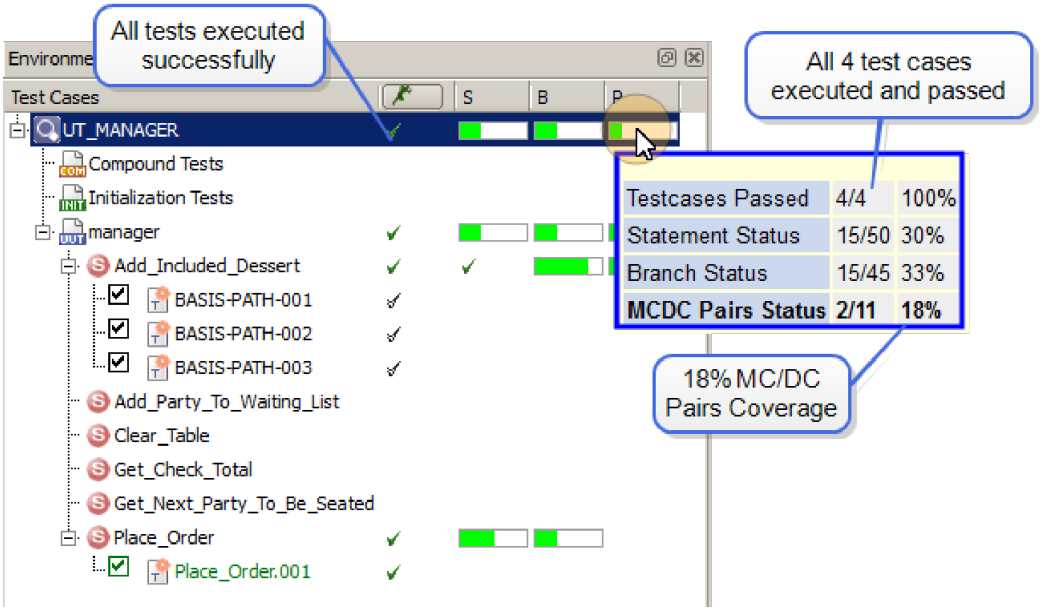


Add Basis Path Tests for the subprogram Add_Included_Dessert by using the right-click context

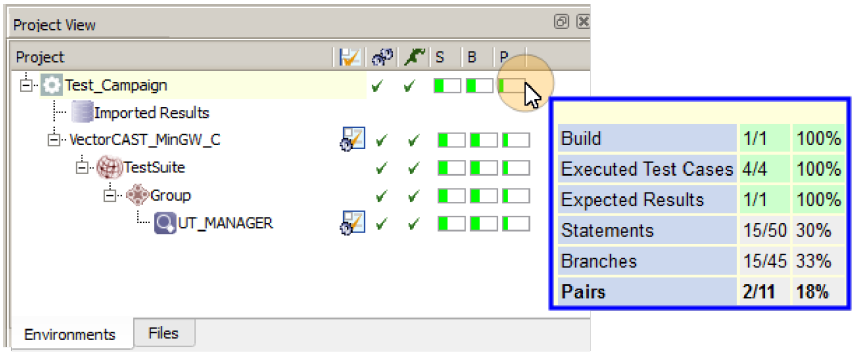
menu to select the **Insert Basis Path Test Cases** option.



Finally, execute the test cases and verify that they all pass.



Bob does a full build and execute, accepting changes to the test scripts and environment when prompted, and generates the Manage Report, noting that all the builds are successful, all expected values were met and all four test cases have passed. The Project Tree updates to display test status and code coverage.



Bob is satisfied with his configuration and environment, and selects **File => Close Project**, selecting to store all results when prompted.

Sharing a VectorCAST Project With Your Team

This Enterprise Testing use case scenario demonstrates putting a project under source code management and then having the team build on the project's initial configuration work. Regression tests are set up to run automatically each evening and generate result reports to be reviewed by the team each morning.

Put a Project Under Source Code Management

In this work flow, VectorCAST has been configured to use the SCM system Git to provide locking and version control. Bob, the Project Architect, puts the Test_Campaign project under Source Code Management.

Bob commits the project's environment/* directory to the team's Git repository, being sure that the following key files are included:

- Test_Campaign/environment/UT_MANAGER/UT_MANAGER.cfg (configuration file)
- Test_Campaign/environment/UT_MANAGER/UT_MANAGER.env (environment file)
- Test_Campaign/environment/UT_MANAGER/UT_MANAGER.mfg (Manage configuration options file)
- Test_Campaign/environment/UT_MANAGER/UT_MANAGER.tst (test script file)
- Test_Campaign.vcm (VectorCAST project file)

These additional files can also be placed under source control if desired:

- Test_Campaign/python/build-instrumented-executable.py
- Source files

Now that the Test_Campaign project is under source control, multiple team members can work on the project simultaneously knowing they are using the most current project files.

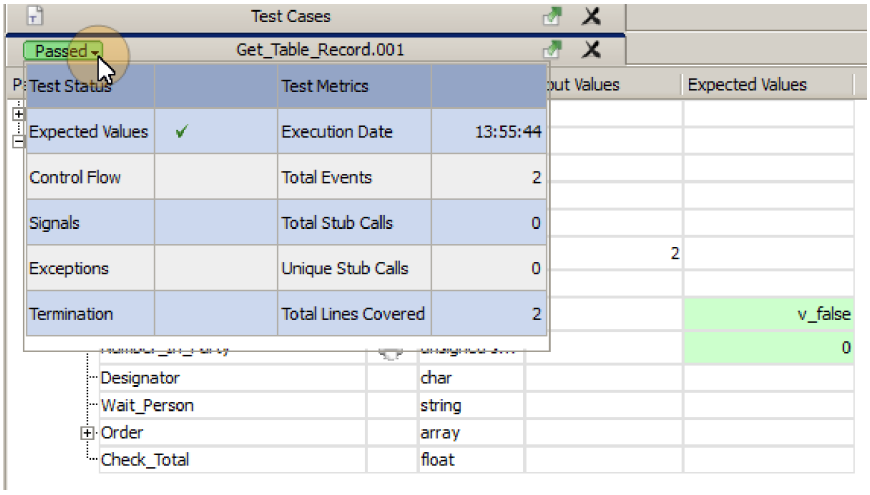
Add New Unit Test Environment

Carlos, a developer, clones the repository of Test_Campaign files locally using Git. Carlos needs to add a new unit test environment for database.c.

To begin, open the Test_Campaign Project by selecting **File => Open** from the Menu Bar and using the browser to navigate to **`$PROJECT_BASE/Test_Campaign.vcm`**. Click the **Open** button and VectorCAST creates the Test_Campaign project locally.

Right-click on the Group node in the Project Tree and select **Create Unit Test Environment > Interactive** from the context menu. Enter the Environment Name: **UT_DATABASE**. Note that the VectorCAST MinGW_C compiler, the Statement + MC/DC coverage type, and the search path specified in the Configuration Node are already set because Carlos is using the configuration from Bob's initial creation of the project. Select **database.c** as the UUT and select the **Build** button.

Carlos adds a test for the subprogram Get_Table_Record, entering an input value of 2 for Table and entering expected values of v_false for Is_Occupied and 0 for Number_In_Party. Carlos executes and the test passes.



Following a successful local build and execute for his new environment, Carlos commits his changes and pushes the changed code to the branch in the team's Git repository. Now each member of the team is able to pull this most recent change from Git as they continue work on the project.

Create Automatic Regression Test Script

Beth, an engineer in the QA department, is required to pull in each day's updates, automatically run regression tests and generate a report each evening to capture the results of the day's work. The reports are saved in a publicly accessible folder for the team to review each morning.

To do this, Beth creates a shell script file, TESTCAMPAIGN_DAILY_RUN.BAT, using the Manage command line interface (CLI). This script is executed each evening using an execution scheduler (such as Windows' "Scheduled Tasks").

Here is a full listing of the script. Let's look at the script line by line.

```
REM Line 1: Establish a timestamp to be used in the report file names
for /f "tokens=2,3,4 usebackq delims=:/ " %a in ('%date%') do (set mydate=%c-%a-%b)

REM Line 2: Create a time stamped file name including the path for the report
set FULL_REPORT_NAME=%mydate%_FullStatus.html
set SharedLocationForReports=S:\Public\Test_Campaign_Daily_Reports

REM Line 3: Fully build or rebuild environments and then execute all tests
%VECTORCAST_DIR%\manage --project=Test_Campaign --build-execute --workspace=S:\Test_
Campaign\build

REM Line 4: Store the status in project database
%VECTORCAST_DIR%\manage --project=Test_Campaign --store-status

REM Line 5: Generate HTML reports with full status of each environment in project
%VECTORCAST_DIR%\manage --project=Test_Campaign --full-status=%FULL_REPORT_NAME% --
output=%SharedLocationForReports%
```

Line 1 retrieves the system date and creates a string to be used for the report file name.

Line 2 uses the date string from the previous line to create a path and file name string that is used when a report is generated later on in the script. A sample filename created by the script is **`04-07-15_FullStatus.html`**

After execution (line 3), the status results are saved in Line 4.

And finally, in line 5, an HTML report is generated and saved in the publicly accessible directory, **`S:\Public\TestCampaign_Daily_Reports`** with a date stamp as part of the file name.

Here is the report:

Manage Report

Configuration Data

Date of Report Creation	04 FEB 2020
Time of Report Creation	2:34:36 PM
Version	20 revision 00c6a14 (02/03/20)

Full Status Section

	BUILD	BUILD TIME	EXPECTED	TESTCASES	EXECUTE TIME	Statements	Branches	Pairs
ALL	2/2 (100%)	-	3/3 (100%)	5/5 (100%)	00:02	16/52 (30%)	16/47 (34%)	2/11 (18%)
VectorCAST_MinGW_C	2/2 (100%)	-	3/3 (100%)	5/5 (100%)	00:02	16/52 (30%)	16/47 (34%)	2/11 (18%)
TestSuite	2/2 (100%)	-	3/3 (100%)	5/5 (100%)	00:02	16/52 (30%)	16/47 (34%)	2/11 (18%)
Group	2/2 (100%)	-	3/3 (100%)	5/5 (100%)	00:02	16/52 (30%)	16/47 (34%)	2/11 (18%)
UT_DATABASE	NORMAL	-	2/2 (100%)	1/1 (100%)	00:01	1/2 (50%)	1/2 (50%)	-
UT_MANAGER	NORMAL	-	1/1 (100%)	4/4 (100%)	00:01	15/50 (30%)	15/45 (33%)	2/11 (18%)

For more detailed information on the Manage command lines used in the scripts, see "Command Line Interface" on page 190.

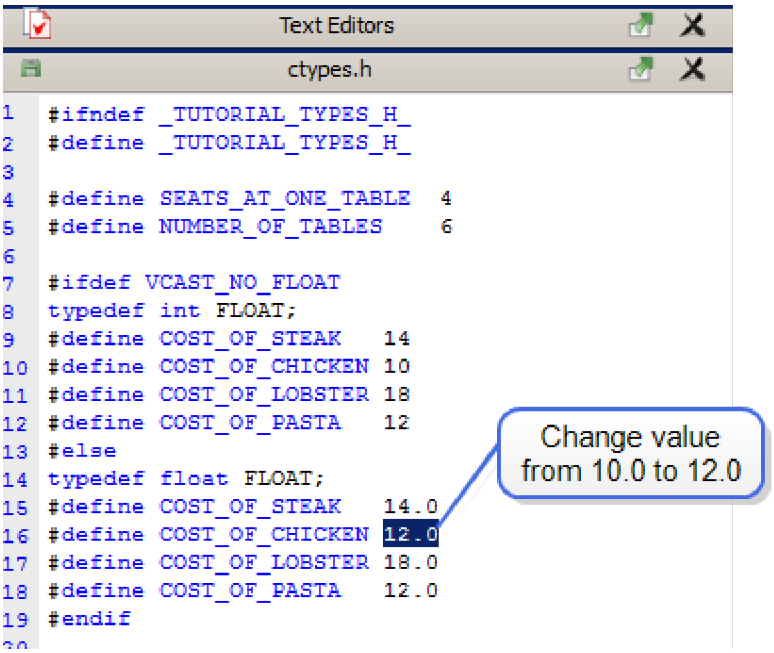
Performing Change-Based Testing

This Enterprise Testing use case scenario has a developer making a change to a single line of source code and then using Change-Based Testing to automatically run only the sub-set of tests affected by the code change. The developer identifies and fixes a broken test and then verifies there has been no regression due to the code change.

Modify Source Code

Carlos has a new requirement to increase the cost of the Chicken entree to 12.00. This will require a change to the **`manager.c`** source code.

Open the source code by right-clicking on **`ctypes.h`** in the Files tab and selecting **View => Original Source Code** from the context menu. Change the value in the header file for COST_OF_CHICKEN from 10.00 to 12.00 and save the change.



Perform Incremental Build/Execute

Doing an incremental build and execute allows VectorCAST to identify the sub-set of tests affected by the code change and run only those tests. Right-click on the Test_Campaign project node in the Project Tree and select **Build/Execute => Incremental** from the context menu.

The Manage Incremental Rebuild Report is produced, showing that of the 5 total tests, only one test was affected by the code change and needed to be executed. In a complex project, executing all tests on each software change can take a great deal of time. Only running the affected tests dramatically reduces test time and speeds up test cycles.

Manage Incremental Rebuild Report

Configuration Data

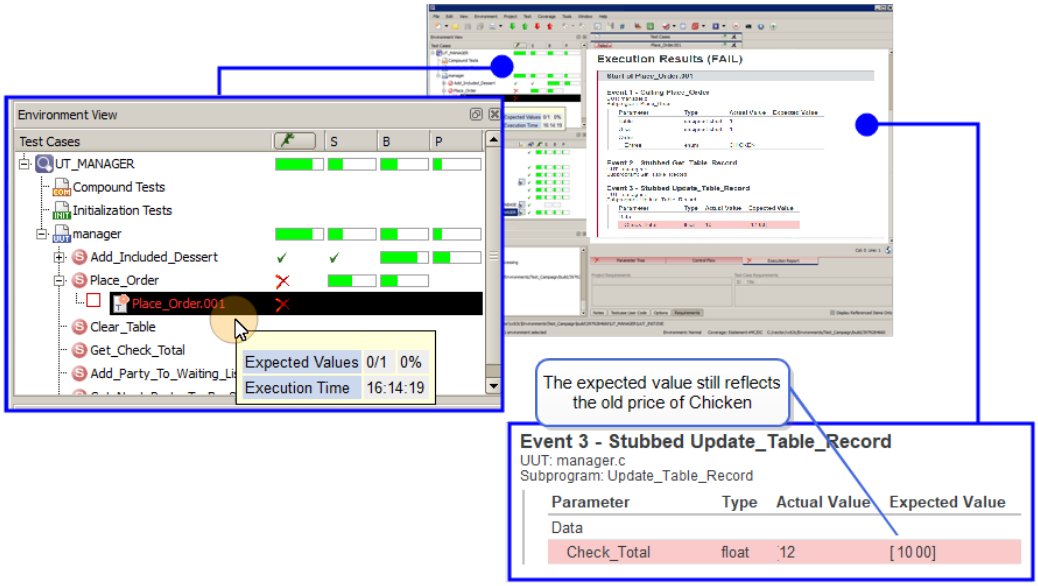
Date of Report Creation	06 FEB 2020
Time of Report Creation	11:05:27 AM

Manage Incremental Rebuild

Environments Affected

Environment	Rebuild Status	Unaffected Tests	Affected Tests	Total Tests
VectorCAST_MinGW_C / TestSuite / UT_DATABASE	Incremental Build Succeeded	1	0	1
VectorCAST_MinGW_C / TestSuite / UT_MANAGER	Incremental Build Succeeded	3	1	4
Totals	**2 / 2 (100%)**	**4**	**1**	5

Change-Based Testing allows developers to quickly assess the impact of source code changes. Carlos opens the UT_MANAGER environment and sees that test case Place_Order.001 is now failing as a result of his change.



Open the test case Place_Order.001 and update the expected value for the Check_Total from 10.00 to 12.00 and save the change. Repeat the **Build/Execute => Incremental** command from the Test_Campaign node, accepting the changes to the test script and environment when prompted.

Once again, the Incremental Rebuild Report shows that only the one changed test needed to be re-executed. Not having to execute the entire test suite saved the project valuable time. Carlos generates a Manage Report and sees that all the builds were successful and all test cases passed. He is now confident that his code change does not introduce any regression and commits and pushes his changes to the Git repository.

Testing with Multiple Configurations

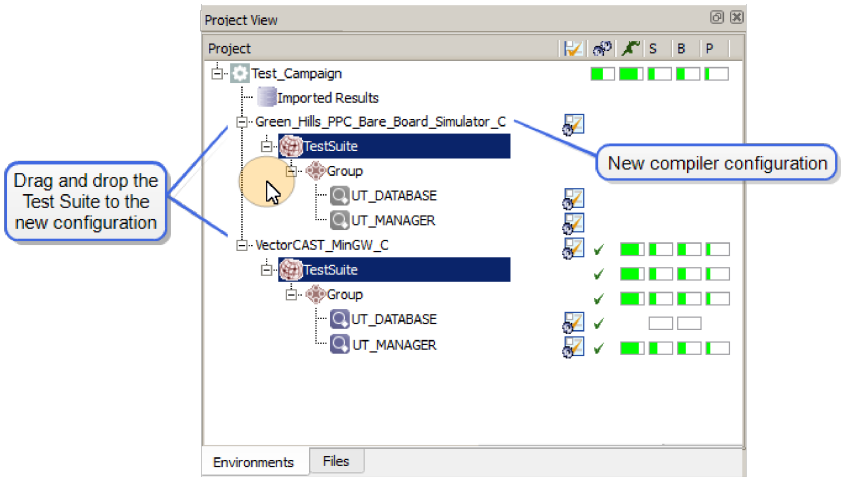
This Enterprise Testing use case scenario has a developer reusing the Test_Campaign project tests in a new configuration. The ability to easily share tests and test configurations improves work flow and increases team productivity.

Add a New Compiler Configuration

Eric has been tasked with porting the application to a new target processor: PowerPC using Green Hills Multi compiler. His first step is to clone the existing project so that he can add a configuration node for this new processor.

Open the local project, then right-click on the project root node and select **Add Compiler => C/C++ => Green Hills => PPC => Bare Board => Simulator => C** from the context menu. A new Green Hills compiler node is displayed in the Project Tree.

Drag and drop the TestSuite node from the VectorCAST_MinGW_C node to the new compiler node. All the children of the TestSuite node are cloned and are now visible under the new Green Hills compiler node. It is important to note that these are not copies of the original test suite, but are the exact same set of tests as in the original node, being used in a new configuration.



Build / Execute the New Compiler

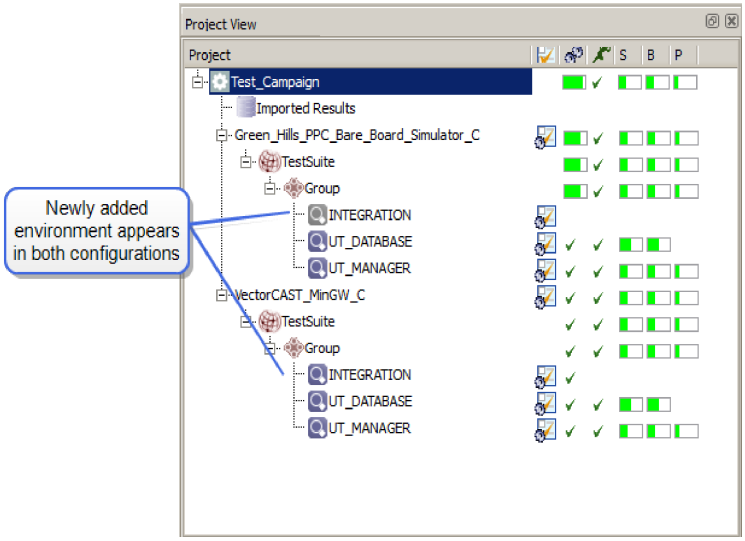
Eric has quickly and efficiently added a new compiler configuration to the project and is ready to build and execute. Right-click on the Green Hills compiler node and select **Build/Execute => Full** from the context menu.

Now the results are exactly the same for both compiler nodes. The same tests have been run in two different configurations. In the future, any changes made to a test in one configuration will update that test in all configurations where it is located and that test is then re-run.

Add a New Environment to the Configuration

Eric also needs to add a new environment and a new test case to the project. To do so he adds the

environment to an existing test suite and it is cloned into each configuration node where that test suite is used. Right-click on the MinGW_C Group node and select **Create Unit Test Environment > Interactive** from the context menu. Name the environment INTEGRATION. Choose database.c and manager.c for the units under test and select the **Build** button.



Note: The new environment is also appearing under the Green Hills compiler node that Eric just added to the project.

Eric creates a test case for Place_Order.001 using the following test case data:

Input Test Data:

UUT: database.c
  Subprogram: Get_Table_Record
    return
      Number_In_Party: 0
      Check_Total: 0.0

UUT: manager.c
  Subprogram: Place_Order
    Table: 2
    Seat: 0
    Order
      Soup: ONION
      Salad: CAESAR
      Entree: STEAK
      Beverage: MIXED_DRINK

Expected Test Data:

UUT: database.c
  Subprogram: Update_Table_Record
    Data
      Is_Occupied: v_true
      Number_In_Party: 1

```
Order
 [0]
   Dessert: PIE
Check_Total: BETWEEN:12.0 AND:16.0
```

Next, Eric executes his test case and verifies that the expected results matched 100%. Satisfied with the results, he will commit and push his changes to the Git repository. Because Eric is able to easily share his work, the rest of the team can leverage his changes and this in turn increases the productivity and success of the entire project.

Sharing Tests and Results Between Multiple Users

One of the key benefits of Enterprise Testing is the ability to share tests and test results across an enterprise. In this use case scenario a developer imports results from the master Test_Campaign project and uses those results to identify any regression in his suite of tests due to the previous day's changes. Any broken tests are fixed locally and pushed to the master project.

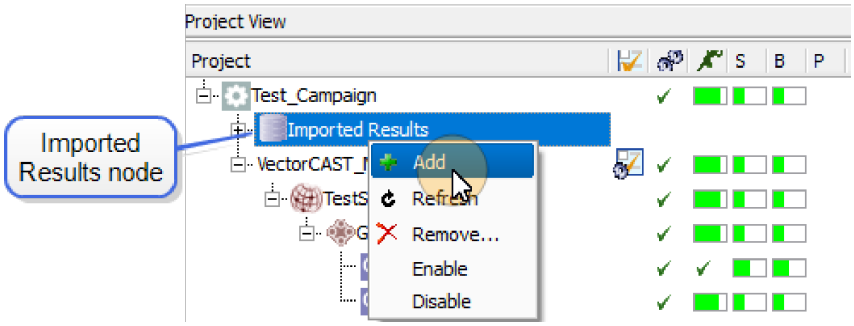
Import Results from Master Project

The master Test_Campaign project continues to remain under Git source control. The project is built each night using Jenkins automated continuous integration and a farm of test machines.

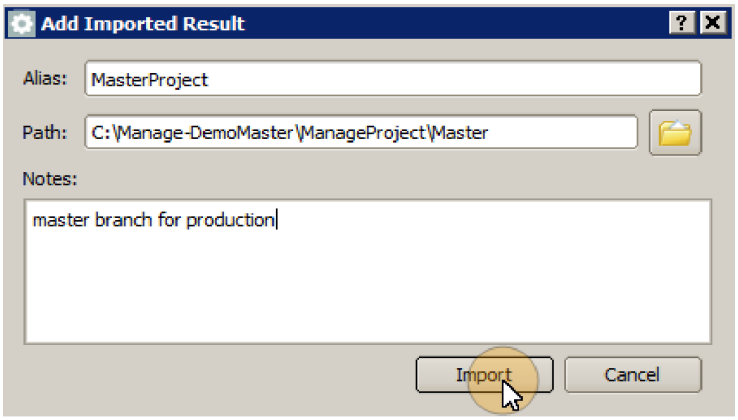
Carlos is responsible for the UT_MANAGER tests. He will use the nightly build, execution and coverage results of the master Test_Campaign project to verify that his tests have not been impacted by any of the previous day's changes. Carlos checks out a new copy of the Test_Campaign project from configuration control, and before doing any local testing, he makes a connection to the Master SQA project.

First, open the local Test_Campaign project. This is a clone of the master Test_Campaign project which is checked into Git.

Right-click on the Imported Results node and select **Add** from the context menu.



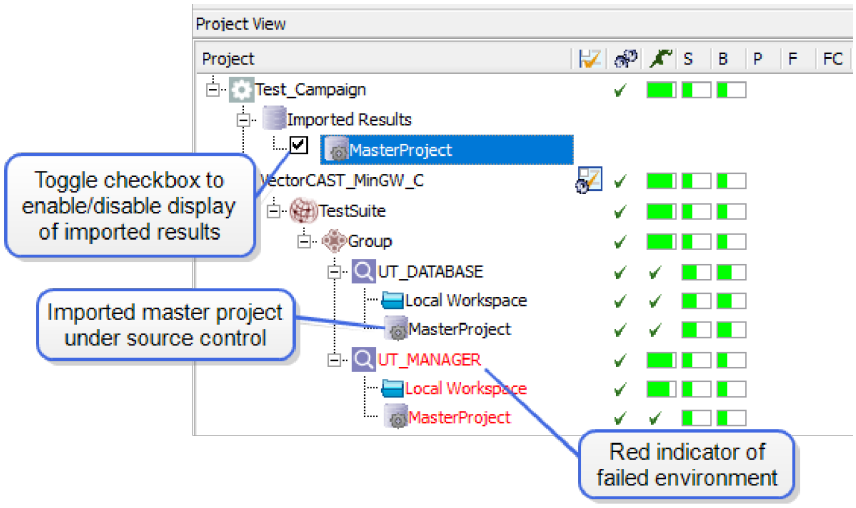
Enter the **Alias** MasterProject for the imported results. Although not required, an *alias* is a nickname for the imported results making it easier to identify.



Enter the path to the master project and select the **Import** button. See "Working With Imported Results" on page 142 for more information about importing results.

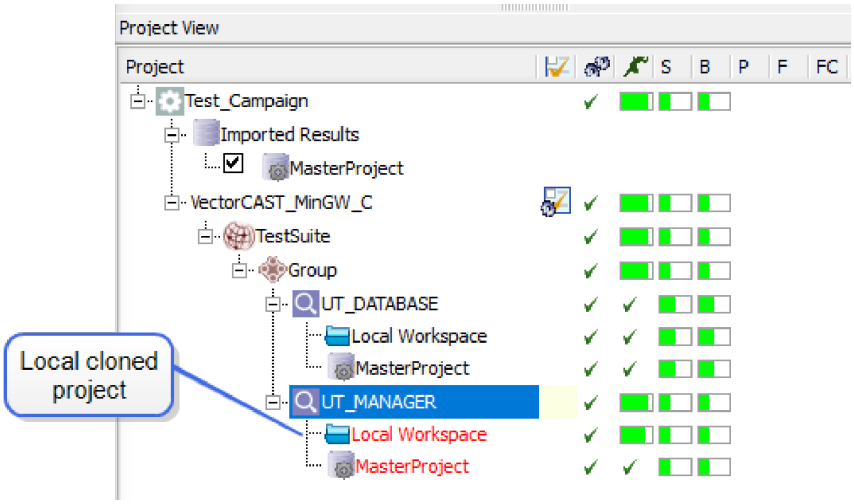
Imported results are a snapshot of the data in the master project. Results imported by the clone project's .vcm file are automatically refreshed every time the project is opened or they can be manually refreshed by right-clicking on the Imported Results node and choosing **Refresh**.

Right-click on the project node and select **Expand All Children** from the context menu. Use the checkbox next to the MasterProject imported results to enable and disable the display of the imported data.

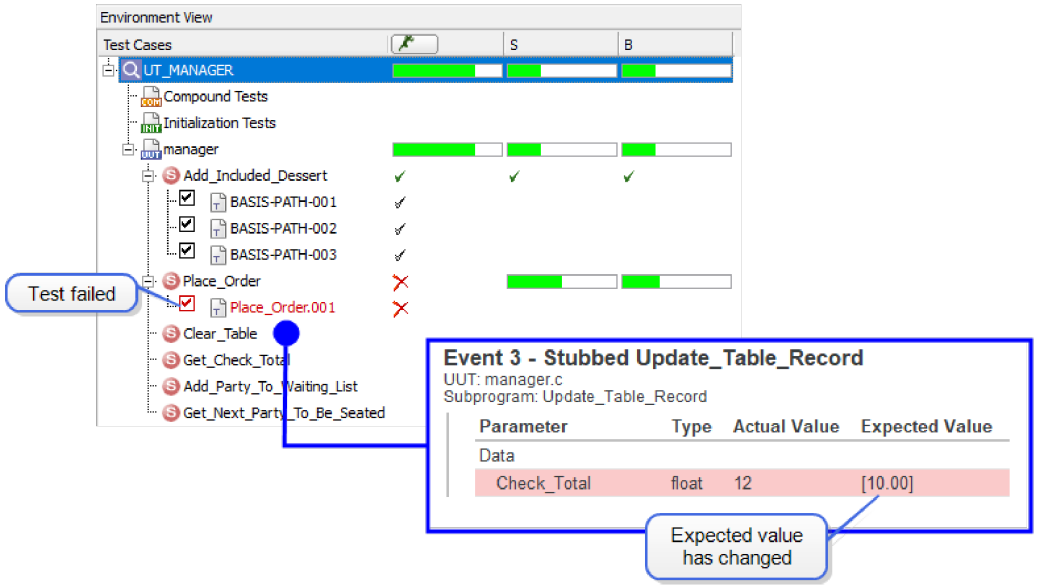


Fix Test Failures Locally

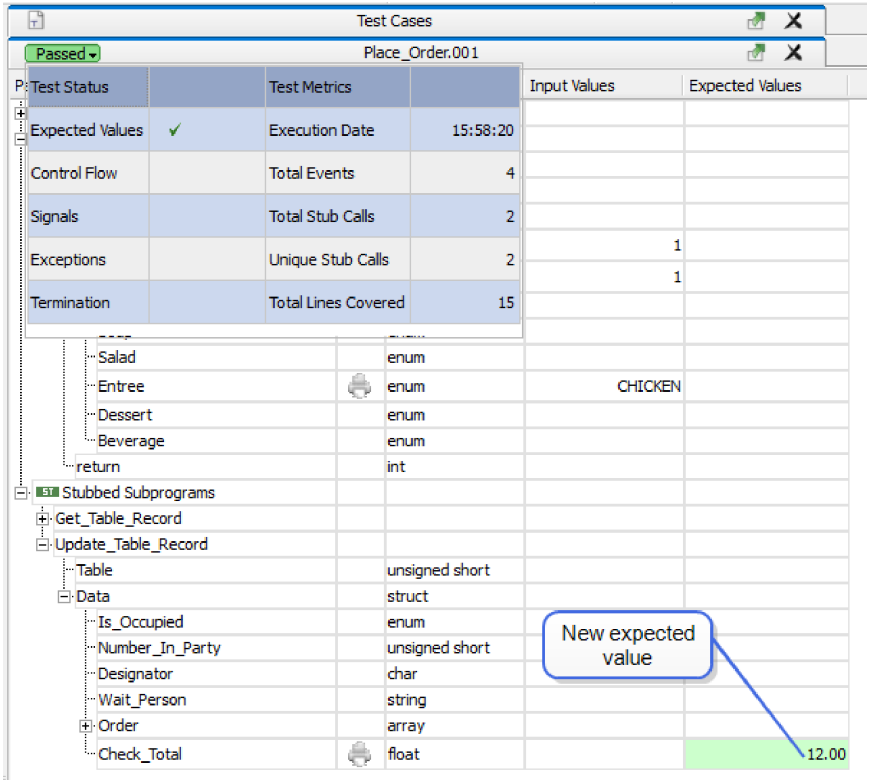
In our example the UT_MANAGER environment is displayed in red, indicating failure. Since Carlos is responsible for the UT_MANAGER tests, he wants to take a closer look and opens the environment to investigate. To debug this test failure, Carlos builds a local copy of the failing test by right-clicking on UT_MANAGER and choosing **Build/Execute => Full** from the context menu.



Expand Place_Order and note that the test Place_Order.001 is now failing. Double-click on the test to open the Execution Report. The Execution Report shows that one of the four results failed. The parameter Check_Total had an expected value of 10.0, but returned an actual value of 12.

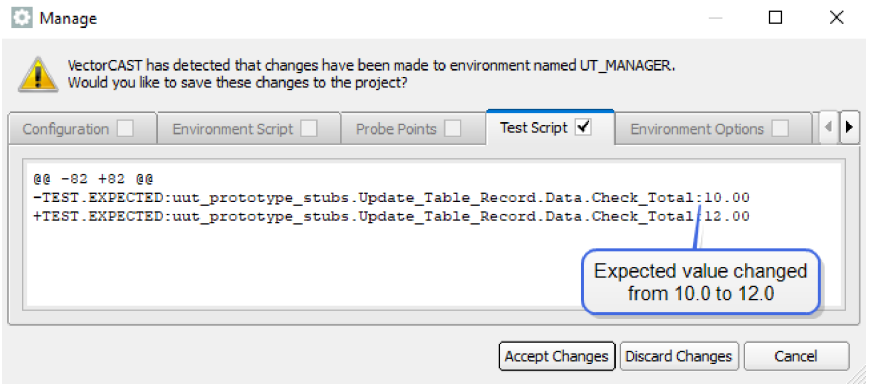


Examining the source code reveals that the price of the chicken entree has been increased to 12.00. Update the expected value to 12.0 and save the change. Carlos executes the test and it now passes.



Commit Changes to Source Code Management

Satisfied that the problem is solved, Carlos chooses **File => Close Environment**. The changes made to the test script are detected by VectorCAST and a difference checker displays the changes for review and acceptance. Accept the changes by clicking **Accept Changes**.



Anytime a change is made to .env, .cfg, .tst and .mfg files (see "Environment Directory" on page 48), and before the changes are saved, a difference checker is displayed, giving you the opportunity to review and accept the changes. The difference checker dialog displays the changes using four tabs, one for each of the file types. A check mark in the tab indicates that a change was made. By clicking an active tab, the change details are shown and you can accept or reject the changes

Having fixed the problem in the local work area, Carlos commits his changes to Git. When the master

Test_Campaign project does its nightly build, Carlos's fix is included.

Using Imported Results With Change-Based Testing

In this use case scenario a developer modifies source code and leverages the common test results of the master Test_Campaign project to perform Change-Based Testing on the local branch to ensure the change hasn't broken any tests.

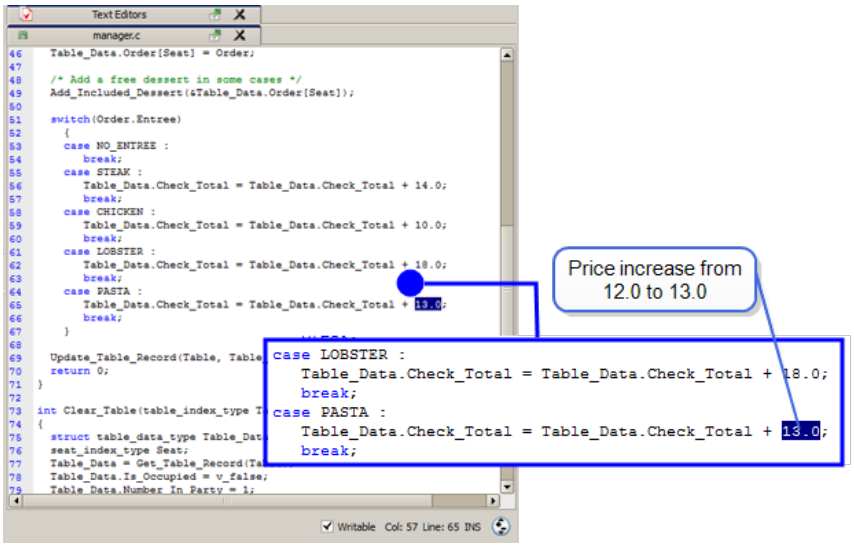
Import Results

Like our Test_Campaign project, large test projects commonly have a master branch for the production version of the source code and test artifacts. The master branch is continuously built and integrated and all tests run daily. Developers are able to use the common test results of the master branch to perform Change-Based Testing on their local branch.

Eric needs to make a change to the source code for manager.c because the price of the Pasta entree has increased to 13.00. Eric has already cloned the Test_Campaign master project locally. To be sure that he has the most current data, he manually refreshes the imported results for the Test_Campaign project by right-clicking on the Imported Results node and selecting **Refresh** from the context menu. See "Working With Imported Results" on page 142 for more information about importing results.

Make Modifications

Open the local source code for manager.c in the text editor and change the Table_Data.Check_Total in the Pasta case from 12.0 to 13.0. Save the change.



Eric isn't sure if the code change he just made will break any of the project's tests. One way to verify this would be to run all of the tests locally. In a complex project, executing all tests on each software change can take a great deal of time. A better option is the VectorCAST CBT feature that allows you to only run the tests affected by a source change. Eric decides to use Change-Based Testing and run only the affected tests to save time and speed up the test cycle.

Perform Incremental Rebuild

After making the change to the source code, right-click on the Test_Campaign project node in the Project Tree and select **Build/Execute => Incremental** from the context menu.

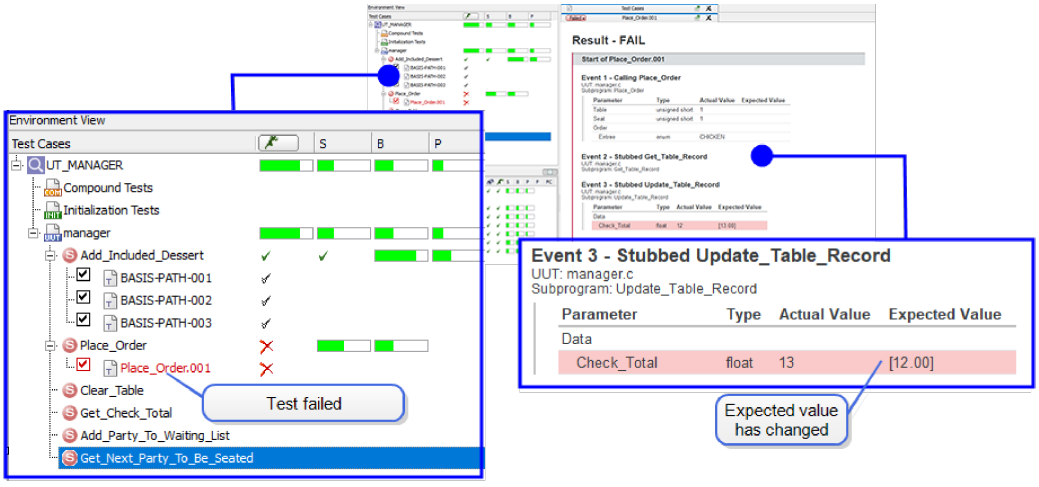
VectorCAST will compute the sub-set of tests affected by the source change and run only those tests. When the Incremental testing is complete, a report is displayed to summarize the updates. The Manage Incremental Rebuild Report shows that Eric's change only affected 2 of the 9 total tests, and that no rebuild at all was necessary for the UT_DATABASE environment, saving time and resources.

Manage Incremental Rebuild

Environments Affected

Environment	Rebuild Status	Unaffected Tests	Affected Tests	Total Tests
VectorCAST_MinGW_C / TestSuite / INTEGRATION	Incremental Build Succeeded	3	1	4
VectorCAST_MinGW_C / TestSuite / UT_DATABASE	Rebuild Unnecessary	1	0	1
VectorCAST_MinGW_C / TestSuite / UT_MANAGER	Incremental Build Succeeded	3	1	4
Totals	**2 / 2 (100%)**	**7**	**2**	**9**

Open the UT_MANAGER environment. Note that the test Place_Order.001 is now failing due to the source code change. Double-click on the test to open the Test Case Editor. The Execution Report shows that one of the four results failed. The parameter Check_Total had an expected value of 12.00, but returned an actual value of 13.00.



By using Change-Based Testing and leveraging the master project's test results, Eric has been able to quickly test the impact of his source code change locally.

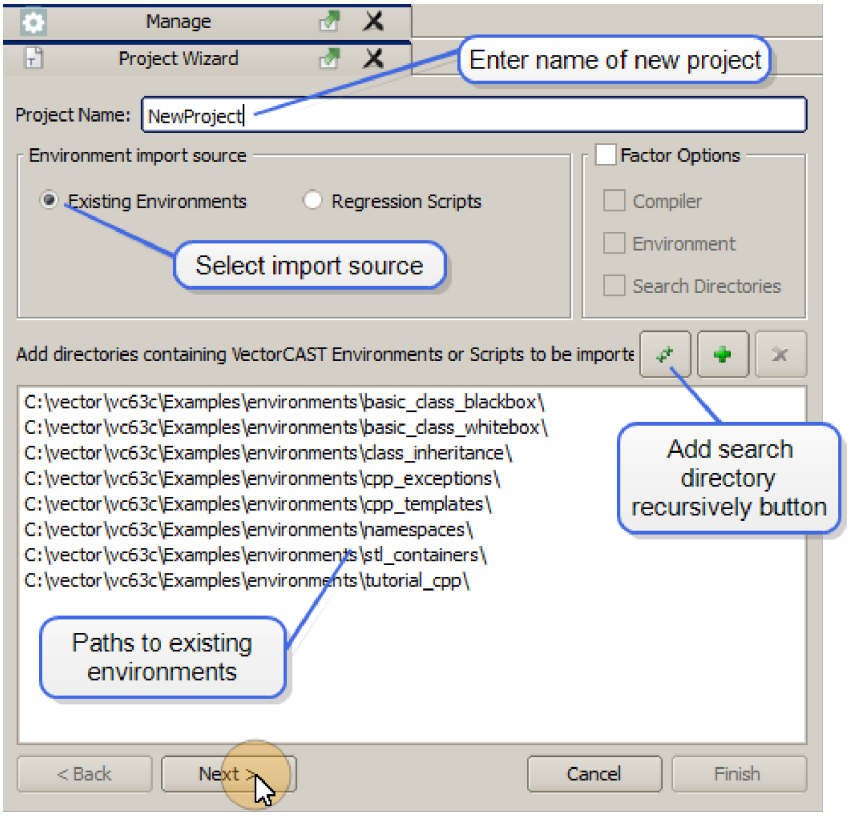
Building a VectorCAST Project From Existing Environments

Many enterprises have a number of existing test environments. This scenario demonstrates building a new VectorCAST Project by leveraging a set of existing environments.

For this example Bob, the Project Architect, creates a new VectorCAST project for a new test campaign using environments that had been previously created for another project. The following sections describe the work flow.

Create a New Project

To begin, select **File => New => VectorCAST Project => From Existing Environments** from the Menu Bar. The Project Wizard opens. Enter the name NewProject in the Project Name field and select **Existing Environments** as the Environment Import Source.

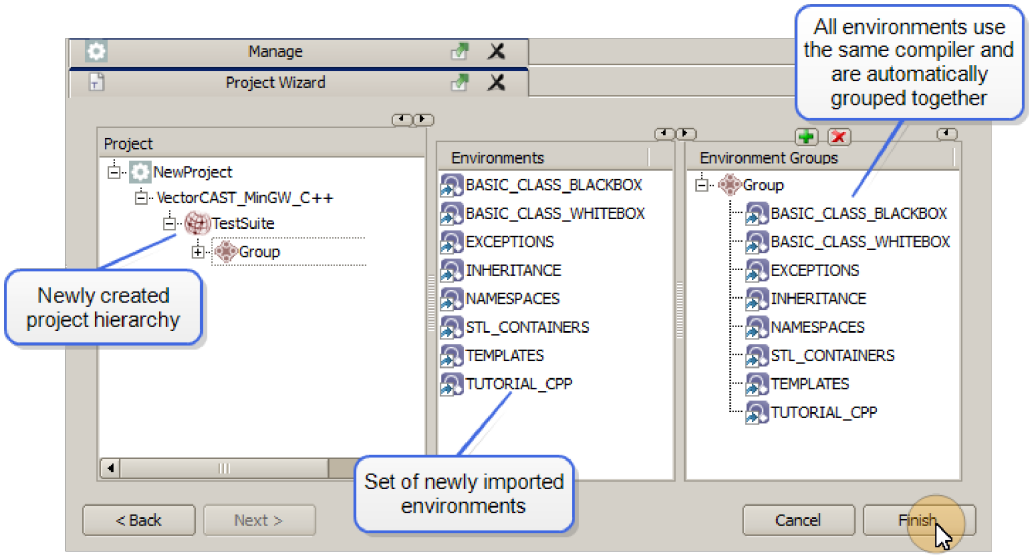


Use the **Add Search Directory Recursively** button to navigate to the parent directory of the existing environments to be imported. The Project Wizard will search this directory and all those below it for built environments to import.

Select the **Next** button. VectorCAST begins to import the test environments it finds in the specified search directories. Environments that have a build error (such as compile or link errors) are not imported.

The second page of the Project Wizard opens, displaying a list of the imported environments and

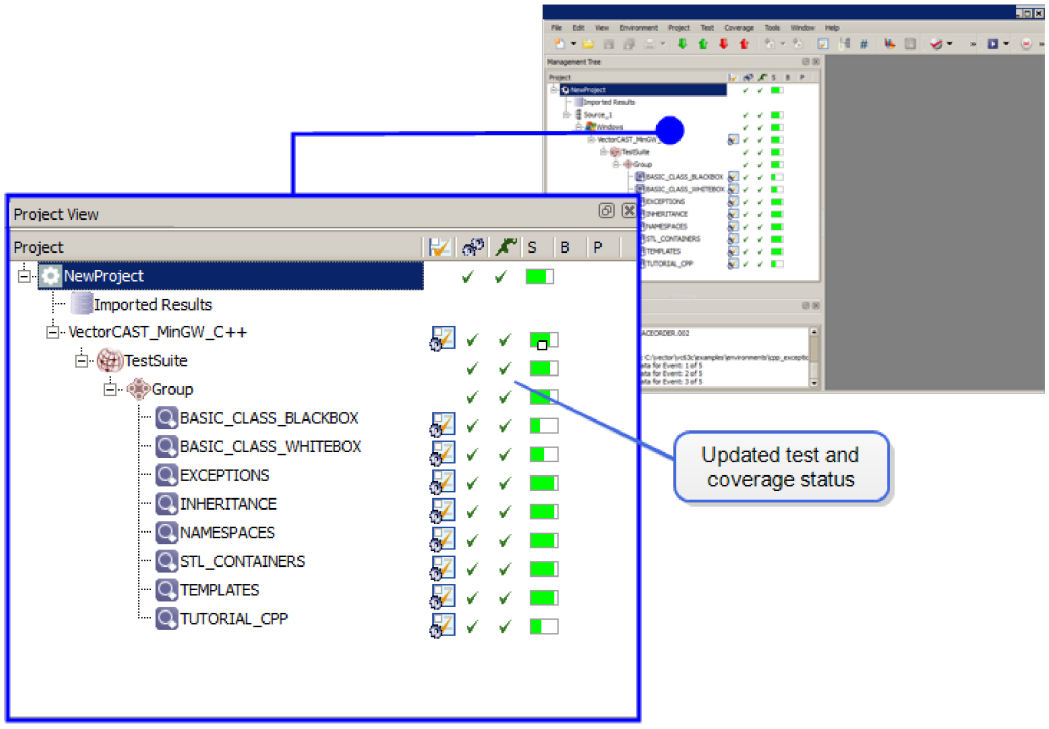
building the environment groups.



Before selecting the **Finish** button, Bob looks to see if the new project needs any modifications. Use the right-click context menu on the nodes to add additional items, rename existing items or delete items.

Note: Deleting a node in the Project Tree panel does not remove the environment from the list in the Environments panel. However, removing an environment from the Environments panel will remove the environment from the Project Tree, Environments and Environment Groups panels.

Satisfied that the current setup will meet his needs, Bob selects **Finish** and creates the new VectorCAST project. The Project Tree automatically builds. Bob migrates the TestSuite to his workspace and then executes his tests by right-clicking on the TestSuite node and selecting **Build/Execute => Full** from the context menu. As the tests execute, the Project Tree updates with test and coverage status.



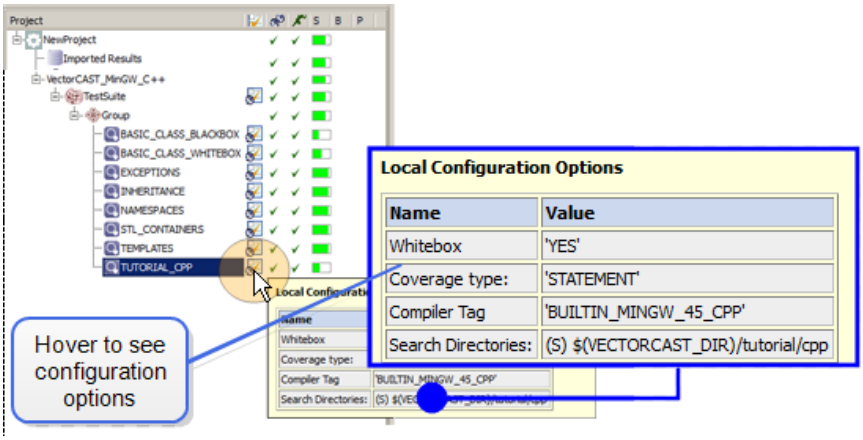
By choosing to create a new project with existing environments, Bob has used shared resources efficiently and helped to shorten the overall project time line.

Managing Configuration Options for Multiple Environments

The ability to control test configurations across multiple test nodes is a powerful time-saving feature of Enterprise Testing. In this use case scenario the project architect applies common configuration options to all of the project's environments.

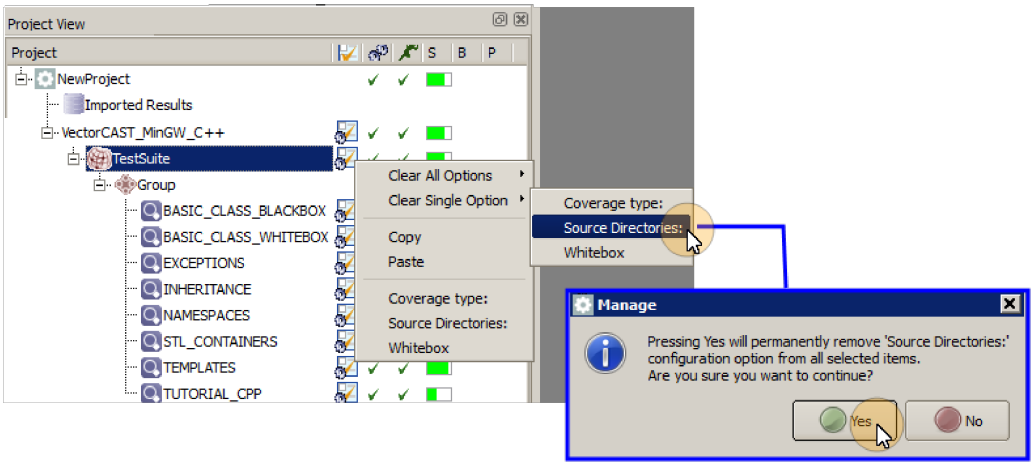
Set Up Common Configuration Options

Bob wants to use a common set of configuration options for all of his test environments. Select environments and hover over the Configuration node icon to see the Local Configuration Options for each. The TUTORIAL_CPP environment has the Whitebox option set to Yes and the Coverage type set to Statement. These are the options that Bob wants to make common for all of his test nodes.

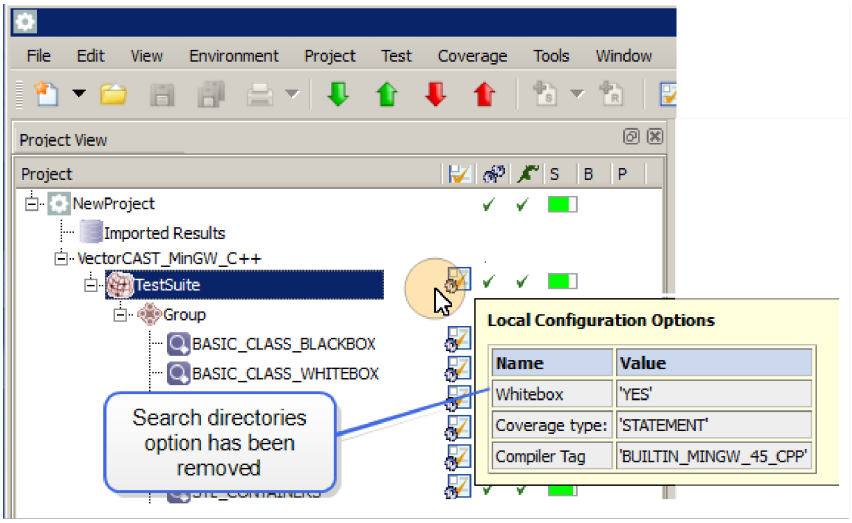


Right-click on the Configuration node icon next to the TUTORIAL_CPP environment and select **Copy** from the context menu. Right-click in the Configuration Options column next to the TestSuite node and select **Paste** from the context menu. A Configuration node icon will appear in the column.

Next, Bob will clear the Search Directories option from the common configuration. Right-click on the TestSuite Configuration node icon and select **Clear Single Option => Source Directories:** from the context menu. Select the **Yes** button on the dialog prompt.

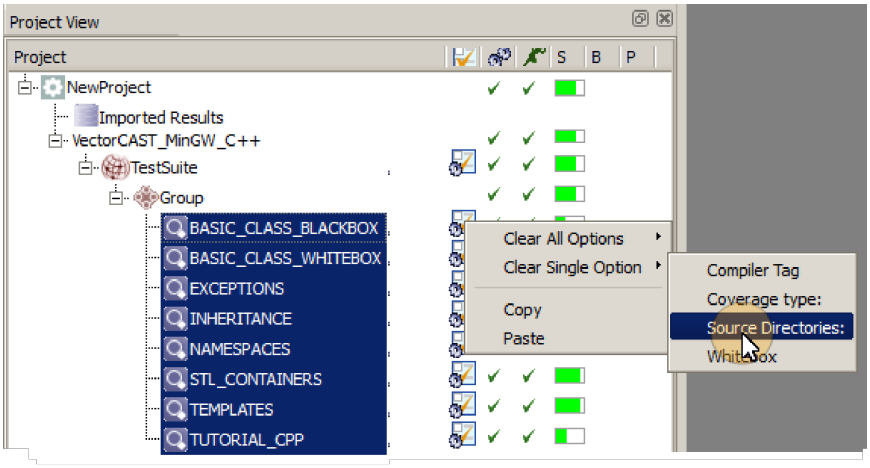


Note that the Search Directories option has been removed from the TestSuite node's Local Configuration.



Clear Elevated Configuration Options

Now that the common configuration options are set up, it's time to clear the elevated options from the individual environment nodes. **Shift-click** or **Control-click** to select all 8 of the environment nodes. Right-click on the Configuration icon next to any selected environment and select **Clear Single Option => Coverage Type** from the context menu. Select the **Yes** button on the dialog prompt. Repeat the process to clear the Whitebox option.

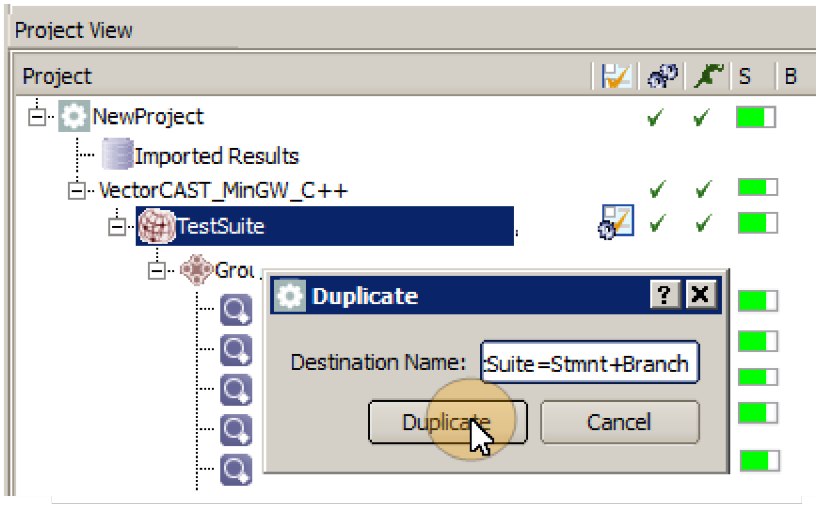


The Whitebox and Coverage Type configuration options have now all been moved to the single TestSuite node. Any future changes to the configuration can now be efficiently accomplished by editing a single TestSuite node, rather than editing each individual environment node.

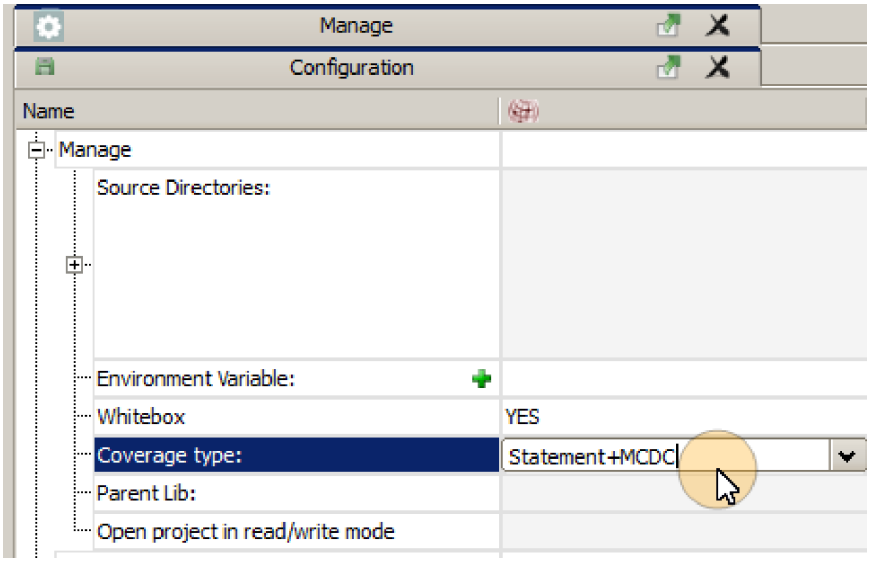
Add a New Test Suite Node and Reconfigure Options

Bob needs to run all of these tests with Statement + MC/DC coverage. Because the configuration is managed from the TestSuite node, he can easily accomplish this by duplicating the first TestSuite and then applying the new coverage type.

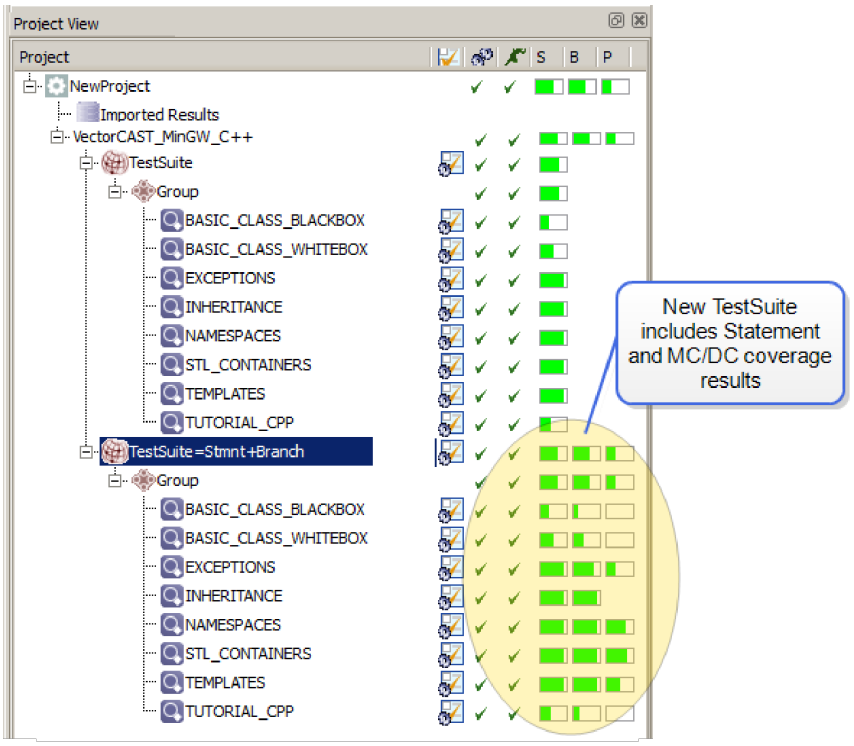
Right-click on the TestSuite node and select **Duplicate** from the context menu. Enter the Destination Name: **TestSuite=Stmnt+Branch** and select the **Duplicate** button.



The new TestSuite node is built in the Project Tree. Right-click on the Configuration node icon for the new TestSuite=Stmnt+Branch node and select **Coverage Type** from the context menu. In the Configuration Editor, change the Coverage type to **Statement + MCDC**. Save the change.



In the Project Tree, right-click on the new TestSuite=Stmnt+Branch node and select **Build/Execute => Full** from the context menu. All of the tests are now built with Statement, Branch and MC/DC coverage enabled.



With just a few simple clicks, Bob has created a new set of tests sharing a unique configuration.

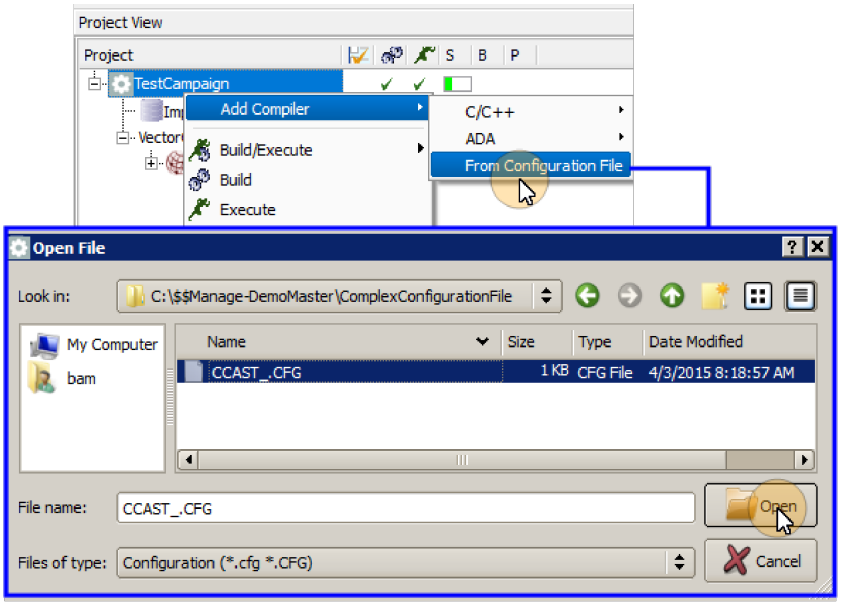
Creating a Compiler Node Using an Existing .CFG File

Enterprises often have existing CCAST_.CFG files that contain complex compiler and project settings. Enterprise Testing provides a feature allowing the import of .CFG files to create new Compiler nodes. In this use case scenario a developer loads an existing .CFG file into Enterprise Testing and creates a new compiler node where the team can begin creating new environments.

Load the .CFG File

Carlos needs to create a new compiler node for the Test_Campaign project. He could create a new compiler node using the supplied template and then modify each of the default settings in the Configuration Editor to meet his project's needs. A better option would be to load an existing CCAST_.CFG file, which already contains the customized options, into the project.

Right-click on the Project root node and select **Add Compiler => From Configuration File** from the context menu. Navigate to the location of the .cfg file and select the **Open** button.



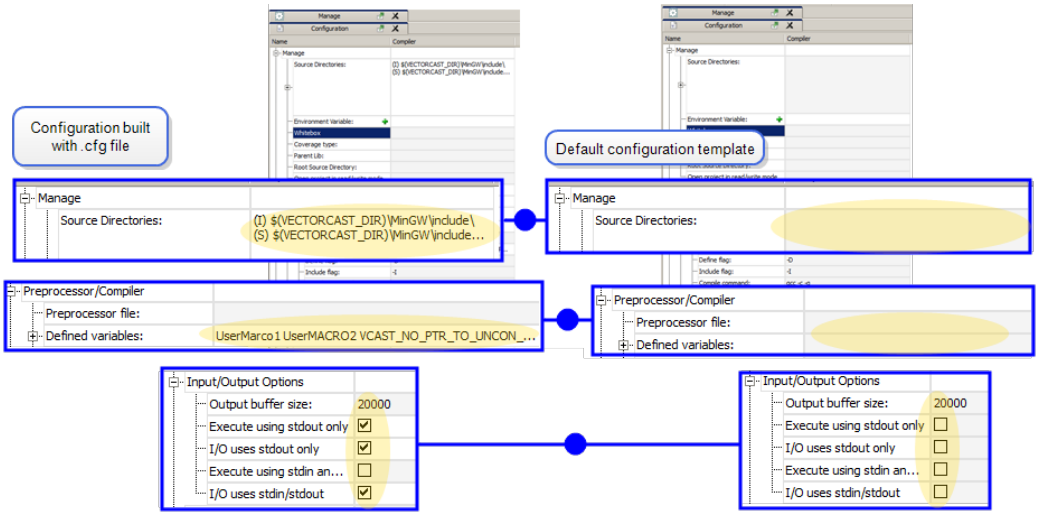
Open the Configuration Editor

Right-click on the new compiler node and select **Open Configuration** from the context menu to open the Configuration Editor. Because he used the imported configuration file option, VectorCAST has already set up the compiler configuration for Carlos.

To see the pre-populated configuration, expand the **Manage** node and note that the Search Directories are already set up. Expand the **C/C++ => Preprocessor/Compiler** node and note that the Defined Variables are set up. Expand the **Target => C/C++ => Input/Output Options** node and note that the desired options are already selected.

The image below compares the default compiler template configuration with the complex configuration. Using the **Add Compiler From Configuration File** option has saved Carlos valuable

time and the team can immediately begin building environments.



Change-Based Testing with VectorCAST/QA

A typical VectorCAST/QA use case scenario uses the System Testing node to invoke the Incremental Build and Execute feature. This scenario demonstrates taking a Enterprise Testing project with a Cover environment, using the Python script to configure the application, and incrementally building and executing after modifying the source.

The VectorCAST installation directory contains a subdirectory, **`Examples/SystemTesting`**, that contains a set of source files and shell scripts that allow you to explore the System Testing and CBT features of VectorCAST/QA. A **`ReadMe.txt`** file provides detailed steps for building the examples. The examples discussed below were run using Windows and are based on the **`SystemTesting/ManagerExample`** directory files.

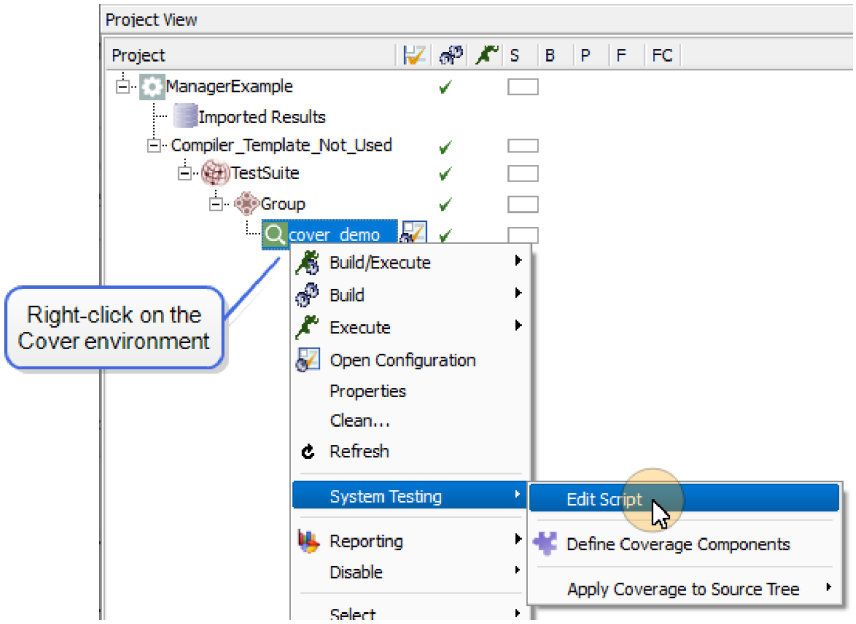
Change-Based Testing processing consists of the following:

- Evaluating file changes and map those changes to the test cases
- Deleting coverage data for all test cases affected by changes
- Re-instrumenting affected source files and re-build the application
- Re-running affected tests and update coverage data
- Providing an Incremental Build Report identifying which tests were re-run

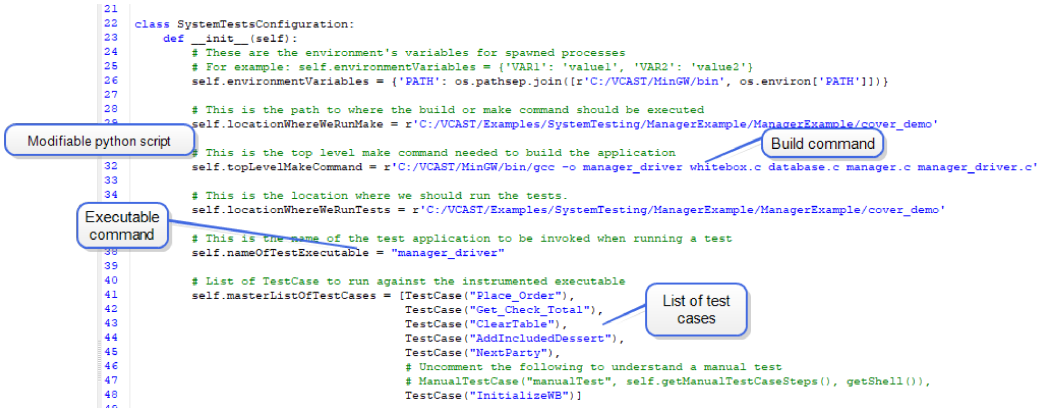
For this example, the user has a VectorCAST project with a Cover environment and uses the System Testing functionality to incrementally build and execute following changes to the source code. The following sections describe the work flow.

Using the Python Script

Eric has a ManagerExample project containing a Cover environment named cover_demo. His task involves making changes to his source code. First, Eric builds his instrumented executable. He opens the Python configuration script **`cover_demo_system_tests.py`** by right-clicking on the cover-demo node and selecting **System Testing => Edit Script** from the context menu.



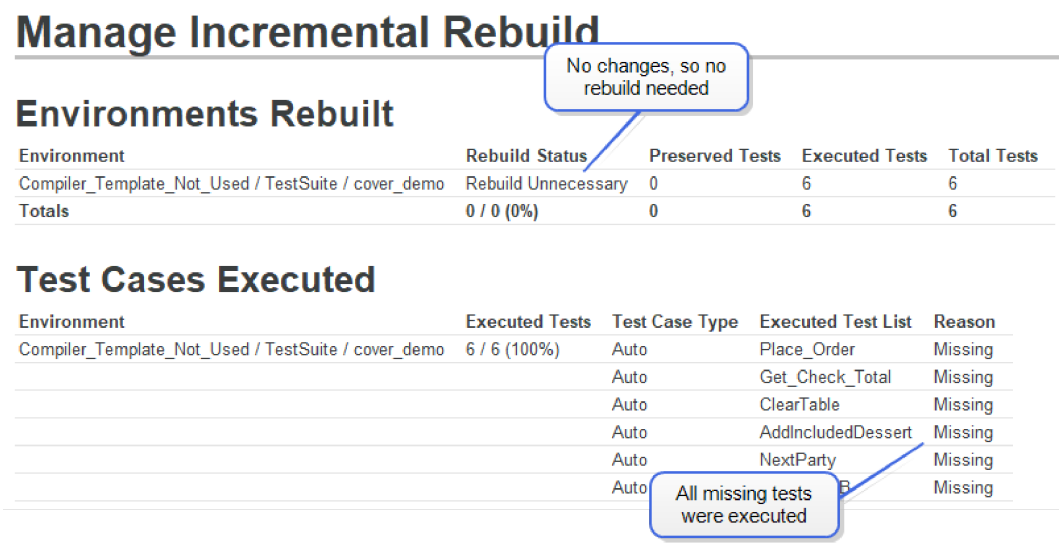
The Python Configuration Script opens in the Script Editor. The script contains functions that are invoked by VectorCAST to provide custom build and execute commands based on the name of a VectorCAST/Cover environment in a VectorCAST project. The configuration script is fully customizable and allows the user to define the application build commands, list of test cases, and test execution commands for an application.



To build the instrumented executable using the **`cover_demo_system_tests.py`** configuration script, right-click on the cover_demo node in the Project Tree and select **Build=>Full** from the context menu.

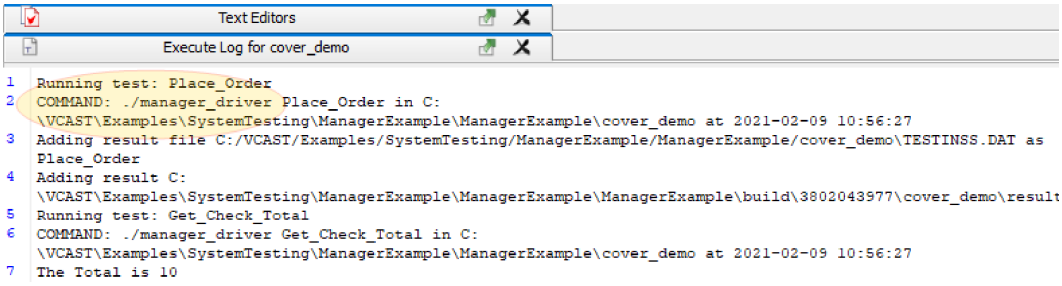
As the Cover environment builds, the Messages area updates in real time and Eric can verify that the build command is executed. For more detail on the build process, view the Build Log by right-clicking on the environment node and selecting **View Build Log** from the context menu.

Next, Eric executes the list of tests as specified in the configuration script. Right-click on the cover_ demo node in the Project Tree and select **Execute => Incremental** from the context menu. Note the Manage Incremental Rebuild Report displays following execution.



Once the Python configuration has been set up, the incremental build and execute functionality performs only the work that needs to be done, based on the current state of the project. At this point, no source changes have been made, so no files were instrumented and only the six missing tests were executed.

Right-clicking on the environment node and selecting **View Execute Log** from the context menu provides further details on the execution.



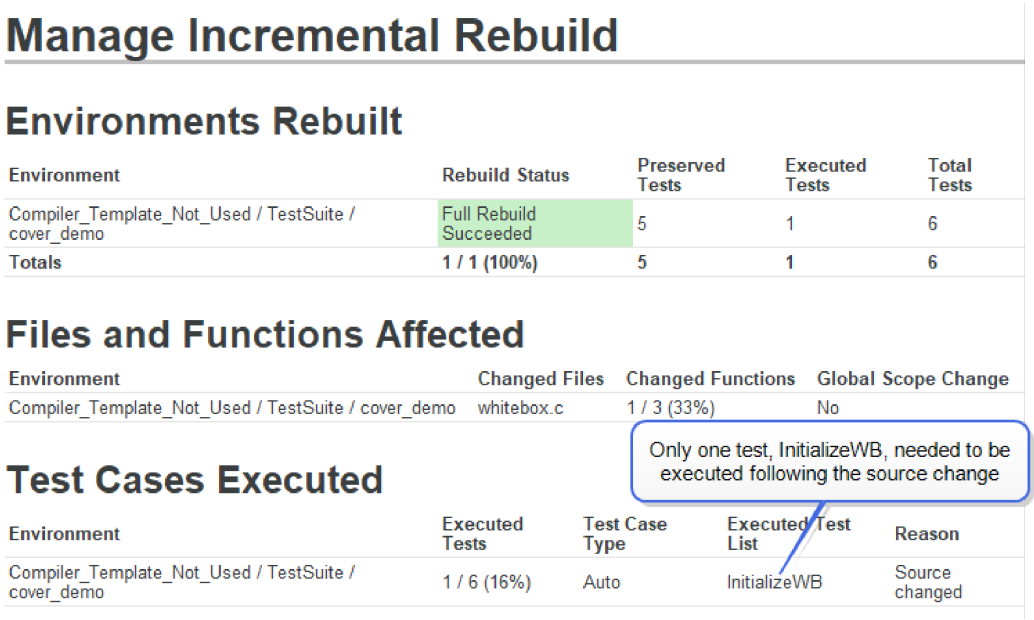
Incremental Build and Execute

Now that the Python configuration is set up for the application, Eric makes a change to the project's source files and executes a Build/Execute command. The incremental build and execute functionality will perform only the work that needs to be done based on the source change.

Next, Eric edits the source file, **`whitebox.c`**. He changes the P.DataValue in the last line from 12 to 11 and saves the change.

```
20 static void InitColor(enum COLOR Val)
21 {
22   CurrentColor = Val;
23 }
24
25 void Initialize()
26 {
27   InitDay(WEDNESDAY);
28   InitColor(BLUE);
29   P.DataIndex = 1;
30   P.DataValue = 11;
31 }
32
33
```

Right-click on the cover_demo node and select **Build/Execute => Incremental** from the context menu. VectorCAST re-instruments the changed file, **`whitebox.c`**, performs the build and re-runs the test InitializeWB, the only test affected by the source change.



Manage Incremental Rebuild

Environments Rebuilt

Environment	Rebuild Status	Preserved Tests	Executed Tests	Total Tests
Compiler_Template_Not_Used / TestSuite / cover_demo	Full Rebuild Succeeded	5	1	6
Totals	**1 / 1 (100%)**	**5**	**1**	**6**

Files and Functions Affected

Environment	Changed Files	Changed Functions	Global Scope Change
Compiler_Template_Not_Used / TestSuite / cover_demo	whitebox.c	1 / 3 (33%)	No

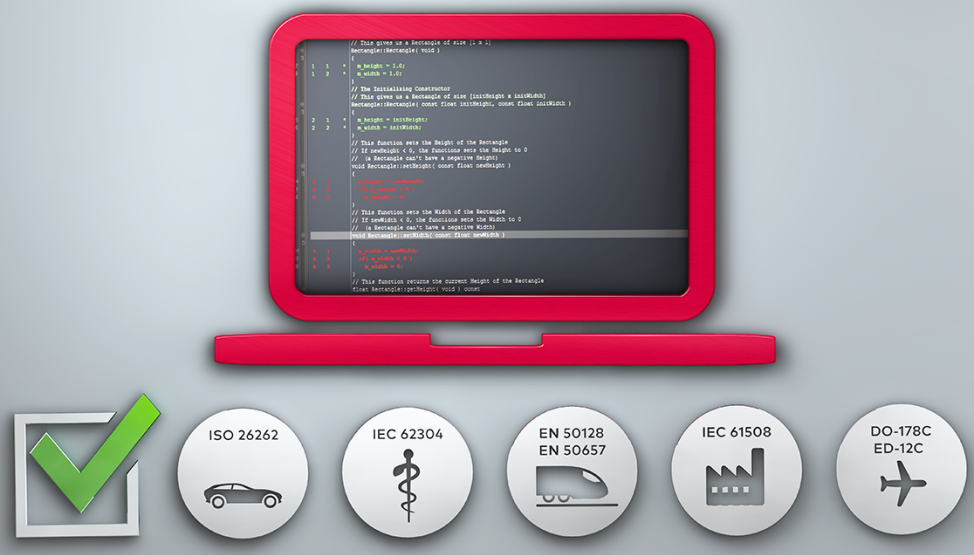
Test Cases Executed

Only one test, InitializeWB, needed to be executed following the source change

Environment	Executed Tests	Test Case Type	Executed Test List	Reason
Compiler_Template_Not_Used / TestSuite / cover_demo	1 / 6 (16%)	Auto	InitializeWB	Source changed

The Manage Incremental Rebuild Report shows that only one file was instrumented and only one test, InitializeWB, was run as a result of the source code change. By using VectorCAST/QA to identify the minimum tests that must be run for his code change and automatically re-running only that test, Eric has been able to quickly test the impact of his source code change.

For more detailed discussion of VectorCAST/QA's features, see "VectorCAST/QA - System Test Automation" on page 174.



Enterprise Testing Tool Reference

The VectorCAST Project Structure

A VectorCAST Project consists of a set of files and directories that are created, maintained and updated through user interaction with VectorCAST. This section explores the VectorCAST Project Structure.

The project.vcm File

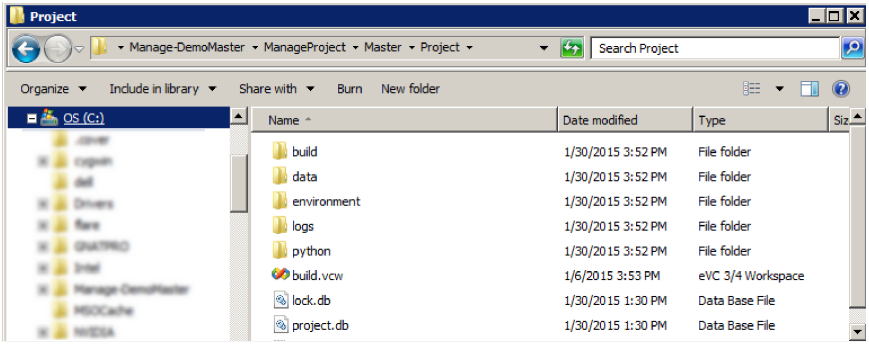
VectorCAST creates a .vcm file with the same name as the project name assigned by the user during project creation. It is an XML file that contains the items analogous to those found in the Project Tree.

Double-click on a .vcm file in a Windows Explorer window and VectorCAST is automatically invoked for that project. This file is maintained and updated by VectorCAST, and should normally not be edited directly by the user.

Note: This file must be kept under Configuration Management (SCM). See the section "File Hooks Scripting" on page 139.

Project Directory

VectorCAST creates a directory with the project name assigned by the user at project creation. It contains all the files and directories that comprise a VectorCAST Project. The contents of the Project Directory are for the internal use of VectorCAST and should not be accessed by the user.



Environment Directory

The *environment directory* contains sub-directories for each test environment. Each environment subdirectory contains a flat list of files. The base name for each file is the name of the parent test environment directory.

Note: The Environment Directory must be placed under Configuration Management (SCM). See the section "File Hooks Scripting" on page 139.

For Unit environments, each environment subdirectory contains a flat list of files with file extensions of .cfg, .env .mfg and .tst.

- The .cfg file contains tool and compiler options specific to this environment.
- The .env file is the VectorCAST/C++ or VectorCAST/Ada environment script. For a description, please refer to Appendix B of the VectorCAST/C++ or VectorCAST/Ada User Guides.
- The .mfg file contains the Manage Configuration Options.
- The .tst files are the Test Case Script files. For a detailed description, please refer to Appendix C of the VectorCAST/C++ or VectorCAST/Ada User Guides.

For Cover environments, each environment subdirectory contains a flat list of files with file extensions of .enc .cfg .mfg and .cvr.

- The .enc file contains the options to generate the Cover environment (for example, base directories, source files, etc.,).
- The .cfg file contains tool and compiler options specific to this environment.
- The .mfg file contains the Manage Configuration Options.
- The .cvr file contains test execution results, including imported coverage results.

Using the Enterprise Testing Interface

The Project Tree, Environments tab, Files tab and Project Editor are the primary interfaces of the Enterprise Testing GUI, and are discussed in detail below.

Understanding the Project View

The Project View is displayed in the upper left-hand corner of the application window and depicts a hierarchical view of the project's components and a visual summary of the project's status. The two tabs at the bottom of the Project View pane allow users to view the project from two different perspectives: **Environments** or **Files**.

The default view is **Environments**. It looks at the project from the perspective of the environments and components which make up the project.

Alternately, selecting the **Files** view allows users to look at the source files contained within the project.

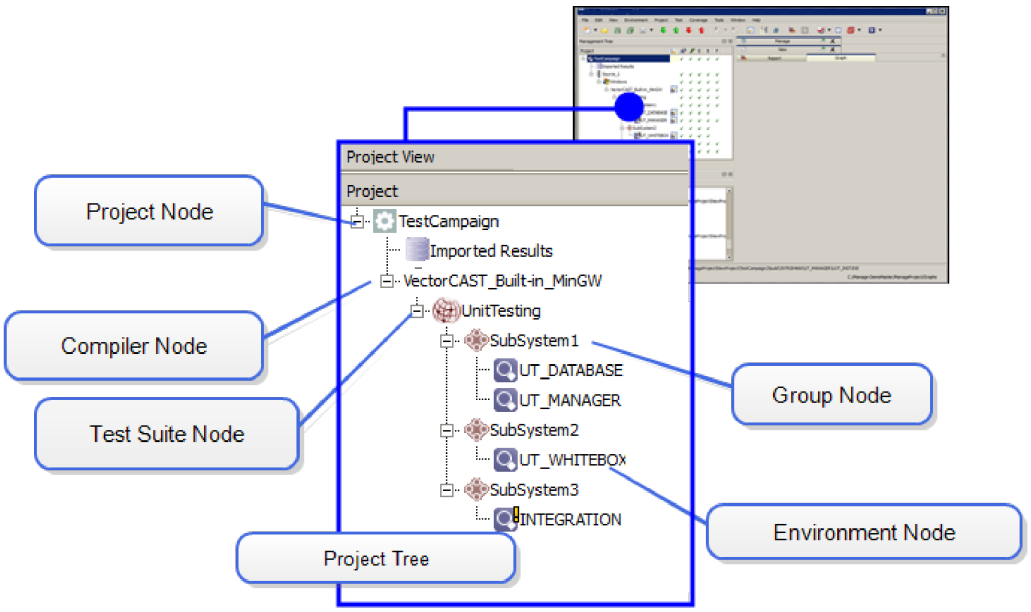
Each of these views are discussed in more detail below.

Understanding the Environments Tab

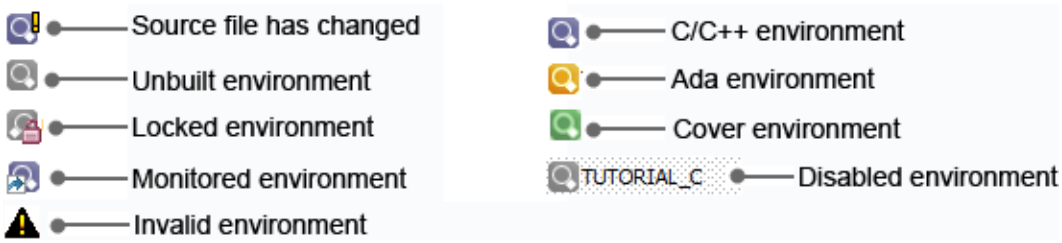
The **Environments** tab is the default view for the project. It is composed of a Project Tree and a Status Panel, which are discussed in detail below.

The Project Tree

The *Project Tree* provides a hierarchical view of the components of the project as a set of nodes on a tree.



The Environment node utilizes several icons to identify an environment:



Project Tree Context Menu

Right-click on any node in the Project Tree and a context menu for the selected node type is displayed.

Use the context menu to:

- **Add Compiler** – (Root node) Add a new Compiler node to the parent Project node. A cascade menu with the following compiler options is provided:
  - **C/C++** – Add a supported C/C++ compiler template.
  - **Ada** – Add a supported Ada compiler template.
  - **From Configuration File** – Import an existing customized .CFG file into the project.
- **Add TestSuite** – (Compiler node) Add a new Test Suite node to the parent Compiler node.
- **Add Group** – (Test Suite node) Add a new Group node to the parent Test Suite node.
- **Create Unit Test Environment** – (Group node) Create a new Unit Test environment from an existing script using the Create New Environment Wizard. The environment is built using the configuration from the nodes above the Group node as well as any settings you configure in the Create New Environment Wizard or the .env script itself. A cascade menu with the following options is provided:
  - **Interactive** – Launches the Create New Environment Wizard
  - **From Script** – Allows you to navigate to an environment script (.env).
- **Create System Test Environment** – (Group node) Create a new System Test environment using the Create New Environment Wizard. A cascade menu with the following option is provided:
  - **Interactive** – Launches the Create New Environment Wizard
- **Build/Execute** – (Root node, Compiler node, Testsuite node, Group node, Environment node) Combine build and execute options into one operation. A cascade menu with the following options is provided for Unit Environment nodes:
  - **Full** – (Environment node) Fully build or rebuild environments and execute all tests.
  - **Incremental** – (Environment node) Build the environment if unbuilt, or build incrementally if source code has changed, or rebuild fully if the incremental rebuild detected a global source code change, and execute only those tests that are un-executed.
- **Build** – (Root node, Compiler node, Testsuite node, Group node, Environment node) Fully build or rebuild environments. When building a Cover environment, VectorCAST calls the

SystemTests.build member in the system_tests.py script, which is used to automate the reinstrumenting of source files and the building of an instrumented executable. A cascade menu with the following options is provided for Unit Environment nodes:

>> **Full** – (Environment node) Fully build or rebuild environment.

>> **Interactive** – (Environment node) For an environment that has not yet been built, build using the Create New Environment wizard. For previously built environments, build using the Update Environment wizard.

> **Execute** – (Root node, Compiler node, Testsuite node, Group node, Environment node) Execute all test cases in all test environments in the project or in the selected nodes. For Cover environments, execute the test cases that don't already exist in the Cover environment. (Note that in Cover environments, "test cases" map to result files.) A cascade menu with the following options is provided for Environment nodes:

>> **Full** – (Environment node) Execute all not-run automated and manual tests. For unit test environments, perform a batch execute.

>> **Interactive** – (Environment node) Open a dialog for the selection of tests to execute. For unit test environments, open the environment and perform a batch execute.

> **Add Existing Environment** – (Root node, Group node) Add an existing Unit Test (.vce) or Cover (.vcp) environment to the project's Root node or a Group node to create a monitored environment. When adding an environment at the Root node level, VectorCAST will either identify an existing location or create a new location for the environment based on the environment's configuration. Use the Group node level to add an environment to a specific group in the project.

> **Open Configuration** – (Root node, Compiler node, Test Suite node, Environment node) Open the Configuration Editor for the selected node.

> **Properties** – (Environment node) Open the Properties - Workspace Attributes dialog for information on the Environment's build directory, build type, dependent files and workspace attributes.

> **Refresh** – (Root node, Compiler node, Testsuite node, Group node, Environment node) Refresh the Status Panel to reflect changes in the workspace.

> **System Testing** – (Environment node) Edit Python Configuration Script, open the Component editor and control location and usage of instrumented files. A cascade menu with the following options is provided:

>> **Edit Script** – Open a text editor for the Python Configuration Script

>> **Define Coverage Components** – Open the Component Editor

>> **Apply Coverage to Source Tree** – Control where instrumented files are stored and how those instrumented files are used

> **Reporting** – (Root node, Compiler node, Testsuite node, Group node, Environment node) Open the Full Status Report and Change Impact Report.

> **Disable** – (Root node, Compiler node, Test Suite node, Environment node) Disable the display of data. Disabled nodes are displayed in the Project Tree using dimmed gray text. The Environment node menu provides a cascade menu with the following options:

>> **Entire Project** – Disables the selected environment throughout the entire project

>> **Test Suite** – Disables the selected environment only in this Test Suite

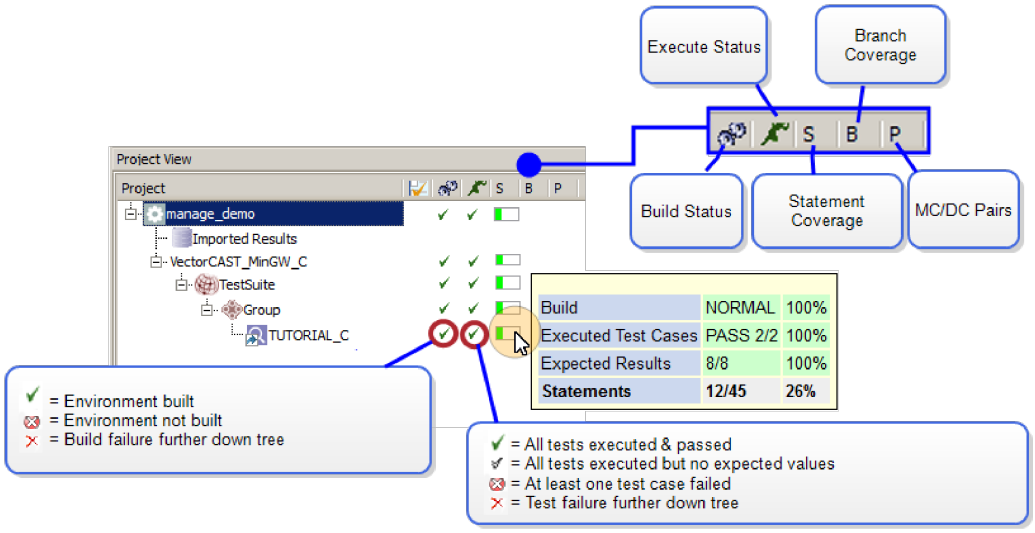
> **Rename** – (Root node, Compiler node, Test Suite node, Group node, Environment node) Rename node with a new, unique name.

- **Duplicate** – (Compiler node, Test Suite node, Environment node) Create a clone of the selected node.
- **Migrate to Workspace** – (Root node, Compiler node, Test Suite node, Environment node) Permanently migrate the selected nodes into the Workspace.
- **Select** – (Root node, Compiler node, Testsuite node, Group node, Environment node) A cascade menu with the following options is provided:
  - **All Children** – Select all environments of children of selected nodes.
  - **Environments with source changes** – Select only the environments of child nodes with source changes.
  - **Environments not yet built** – Select only the unbuilt environments of child nodes.
- **Delete** – (Test Suite node, Compiler node) Permanently remove selected nodes from the Project Tree.
- **Remove** – (Group node) Permanently remove Group node from the Project Tree.
- **Remove Environment** – (Environment node) A cascade menu with the following options is provided:
  - **From Project** – (Environment node) Permanently remove the selected environment from the project.
  - **From Group** – (Environment node) Permanently remove the selected environment from the group.
- **View Build Log** – (Environment node) Open the Build Log for the selected environment.
- **View Execute Log** – (Environment node) Open the Execute Log for the selected environment.
- **Unlock Environment** – (Environment node) Only available for monitored environments. Release lock on selected monitored environment.
- **Expand All Children** – (Root node, Compiler node, Test Suite node, Group node) For a selected node that contains sub-items, display all child nodes.
- **Collapse All Children** – (Root node, Compiler node, Test Suite node, Group node) For a selected node that contains sub-items, hide all child nodes.

The Status Panel

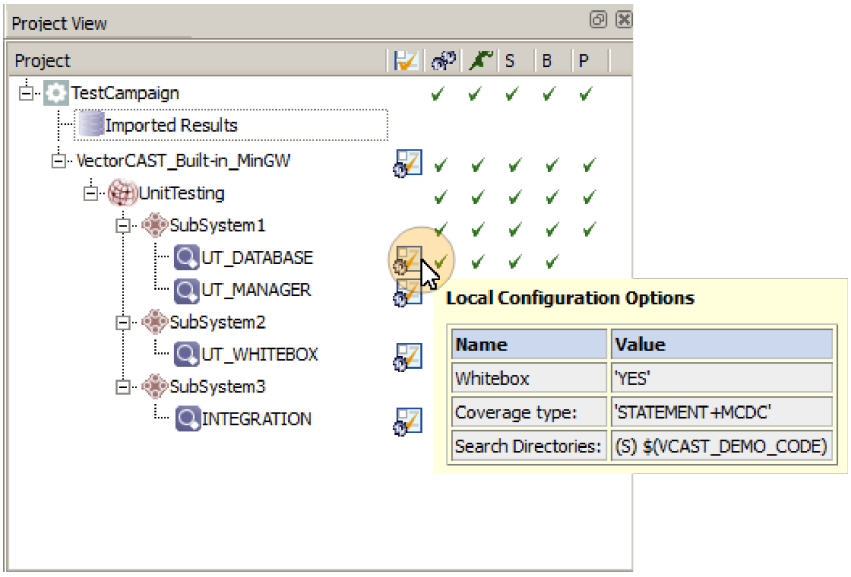
The *Status Panel* on the Project View provides a quick, visual summary of the project's status. Status categories are placed in columns and appear to the right of the node names in the Project Tree.

The width of each status column can be adjusted by selecting and dragging the column separator controls found at the top of the Project Tree.

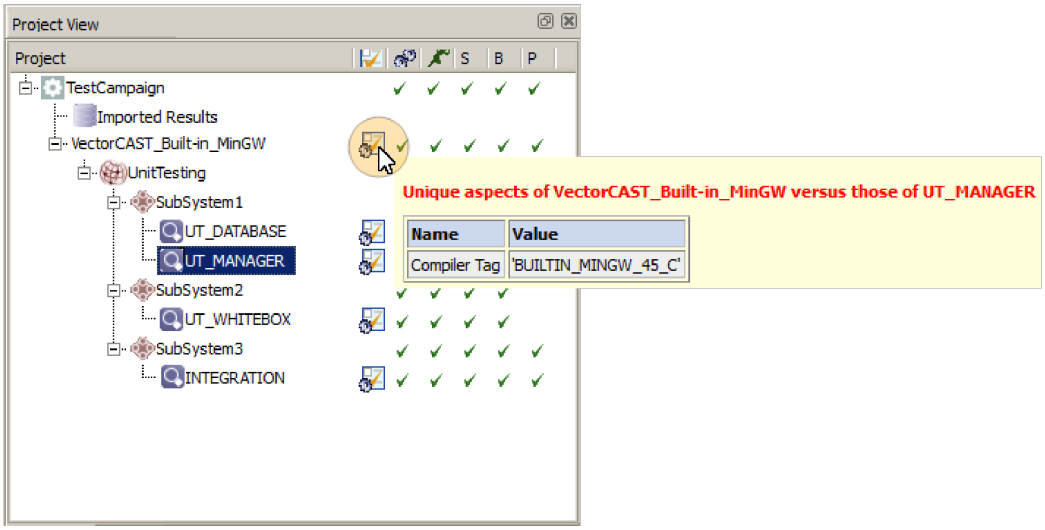


Status Panel Columns

 *Configuration Options* – An icon appearing in this column indicates that there is configuration options information for the node. To quickly see the information hover over the configuration options icon and a tool-tip appears. Double-click the icon next to a migrated environment to open the Options window for that environment.



If you select a node in the tree, and then hover over a *configuration options* icon for some other node, a tool-tip is displayed showing the differences in configurations between the two nodes.



For more information on Configurations, see "Editing and Modifying Configurations" on page 123.

Build Status – A check mark ✓ in this column indicates that the build was successful. An X ✕ in this column indicates that the build failed. If the build was partially successful, a status bar is displayed.

Double-click in the Build Status column next to an environment to open the corresponding Build Log for that environment.

Execute Status – A check mark ✓ indicates that all expected values in all test cases have matched. Test cases with no expected values are counted as passed.

An X ⊗ indicates that *all tests have failed*. If some tests have passed and some have failed, then a *status bar* is displayed. The status bar graphically shows the percentage of tests passed in green and the percentage failed in white.

Test cases that have not executed are counted as failed. A test case with one or more expected values that do not match is counted as failed.

An environment with no test cases is neither passed nor failed.

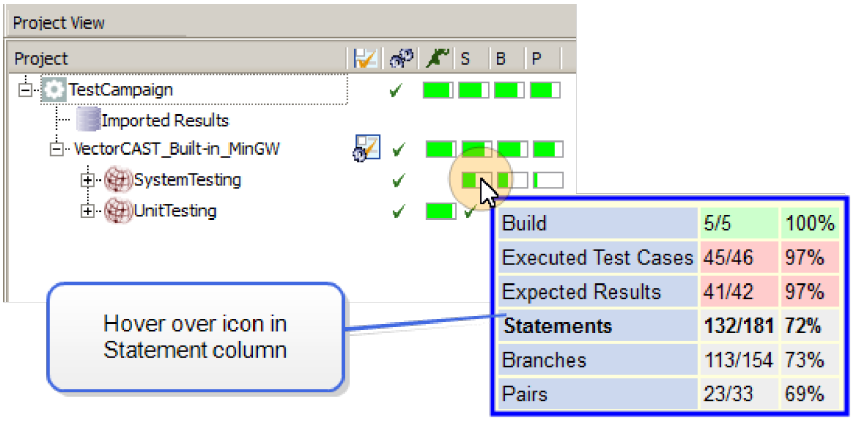
Double-click in the Execute Status column next to an environment to open the corresponding Execute Log for that environment.

Coverage Status – Coverage is represented by five columns, one for each of the coverage types, S = *Statement*, B = *Branch*, P = *MC/DC Pairs*, F = *Function*, and FC = *Function Call*. Branch (B) coverage applies to both standard branch coverage, as well as MC/DC branches. A status bar is used by all three coverage columns to indicate percent coverage achieved. The status bar is all green when there is 100% coverage, all white when there is 0% coverage, or a combination of white and green, when the coverage percentage is between 0% and 100%.

If one of the coverage columns is not applicable to the coverage type set for the environment, the column is blank.

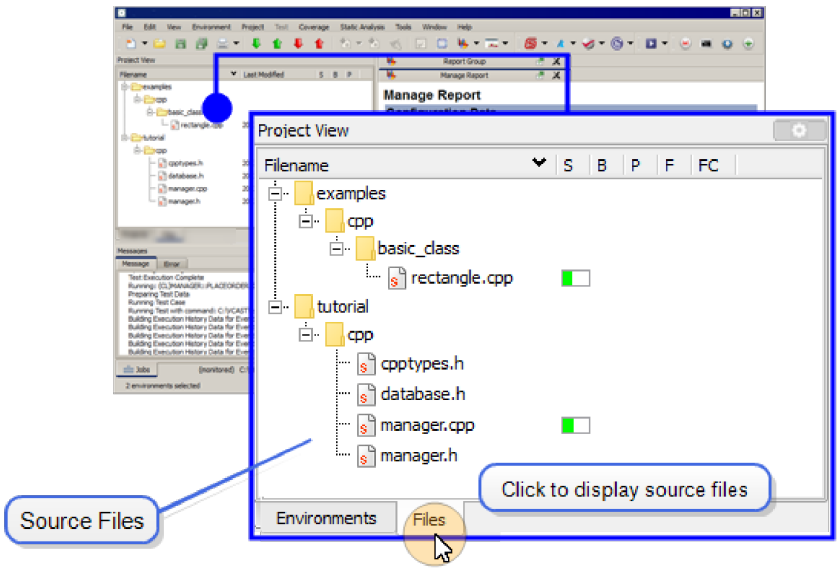
Status Column Tool-Tips

If you hover over any of the status icons in the *build*, *execute* or *coverage* columns, a summary tool-tip will appear. Build, execution, and coverage information appears for all columns, and the data specific for the selected column is shown in bold. The row headings that are displayed are similar to the column headings contained in the *summary report*.



Understanding the Files Tab

The **Files** tab displays a "file" view of all source files in the VectorCAST project. Access the Files view by clicking the Project View **Files** tab.



Using this tab, you can:

- View source files used in the project, including or excluding header files
- Change the list to display the files in a flat list or hierarchically by directory

- Open an environment that uses a source file
- Edit the source file
- View the coverage achieved on a source file
- Open the Coverage Viewer for a source file

File Context Menu

Right-click on any file name or file sub-entry and a context menu is displayed.

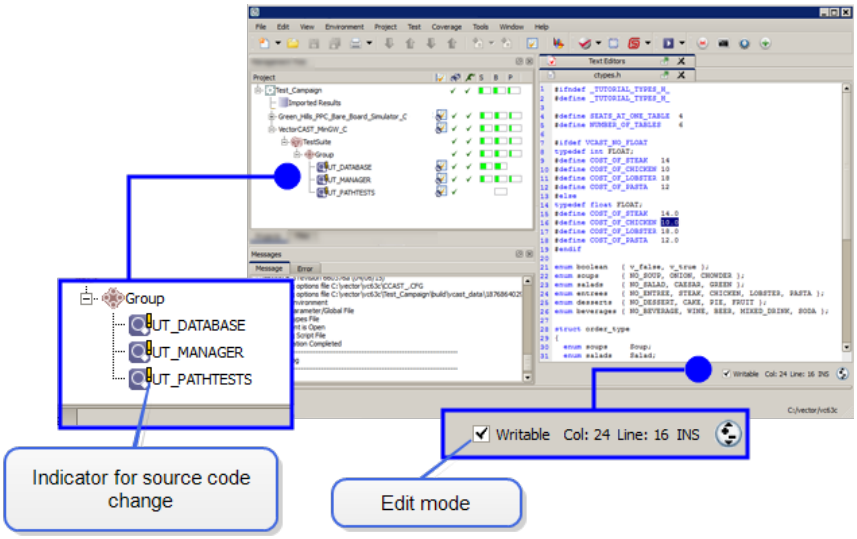
Use the context menu to:

- **View** – view original source code, Metrics Report, Aggregate Coverage Report, Environment Coverage Report, or open the Coverage Viewer.
- **Open Environment With File** – select this item and a list of environments associated with the item is displayed. Select an environment to open its Environment View.
- **Rescan Dependent Files** – this will refresh the source code view.
- **Flatten** – when enabled, this entry will have a checkmark. A flattened view of the tree for the selected file removes the plus and minus tree controls from view.
- **Show Headers** – when enabled, this entry will have a checkmark and header files are included in the list of files. Disable this feature and the header files are removed from the display.
- **Expand All Children** – for an item in the tree that contains sub-items, all sub-items are displayed.
- **Collapse All Children** – for an item in the tree that contains sub-items, all sub-items are hidden.

Editing Source Files

Right-click on the source file and select **View => Original Source Code** from the context menu. Alternatively, you can double-click on a source file without coverage to open it in the text editor. To put the file in edit mode, select the Writable checkbox in the lower right of the text editor.

The Project Tree will display a yellow exclamation point indicating that an update is required whenever an environment depends upon a changed source file. Right-click on the Project Tree and choose **Select => Environments with source changes** to select all environments with source changes to dependent files.



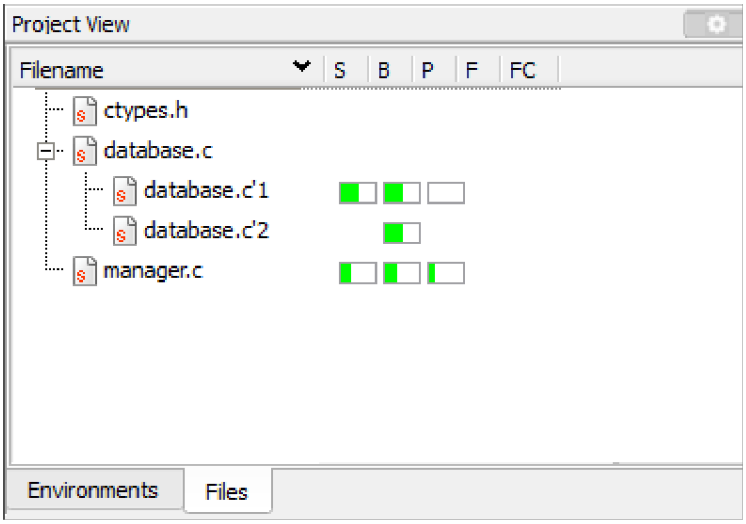
By default, VectorCAST will scan all dependent files for changes when a project is opened. You can also manually control this scanning by selecting **Rescan Dependent Files** from the Files tab context menu.

Coverage Achieved

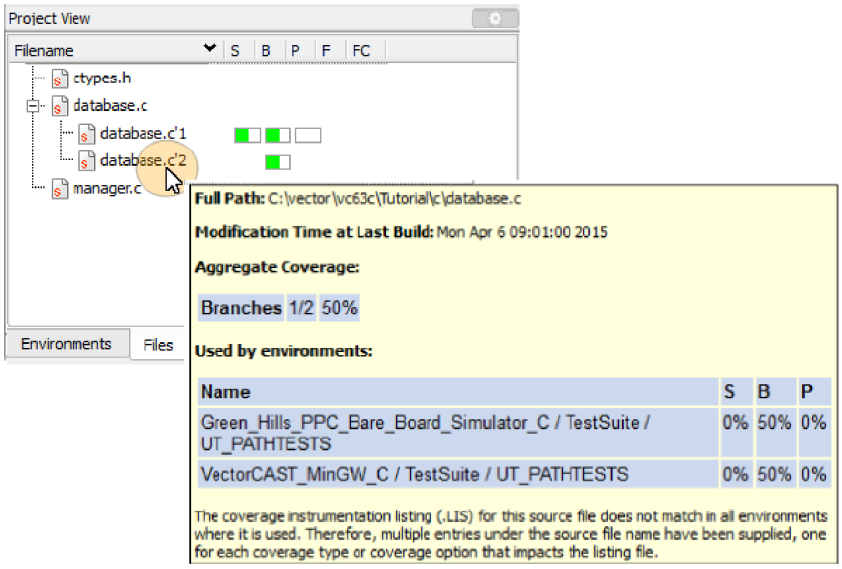
Code coverage metrics are displayed on the Files Tab allowing you to see the code coverage for each file. Additionally, double-clicking on a source file name with coverage will result in the VectorCAST Coverage Viewer being displayed for that file, enabling you to see exactly which code constructs have been covered.

For each source file used in a test environment that has been built and executed with coverage, a green coverage percentage bar displays the coverage achieved for the file across all of its test execution contexts. A separate percentage bar is displayed for each coverage type (Statement, Branch, MC/DC pairs).

For Translation Unit Perspective only, if a file is tested with differing coverage types in different data sources, a separate sub-entry file name is created in the “Filename” column for each coverage type with an associated coverage bar. In the example shown below, database.c was executed with statement and MC/DC coverage in one data source and with branch coverage in another data source. Sub-entries “database.c’1” and “database.c’2” were created to differentiate the coverage types.

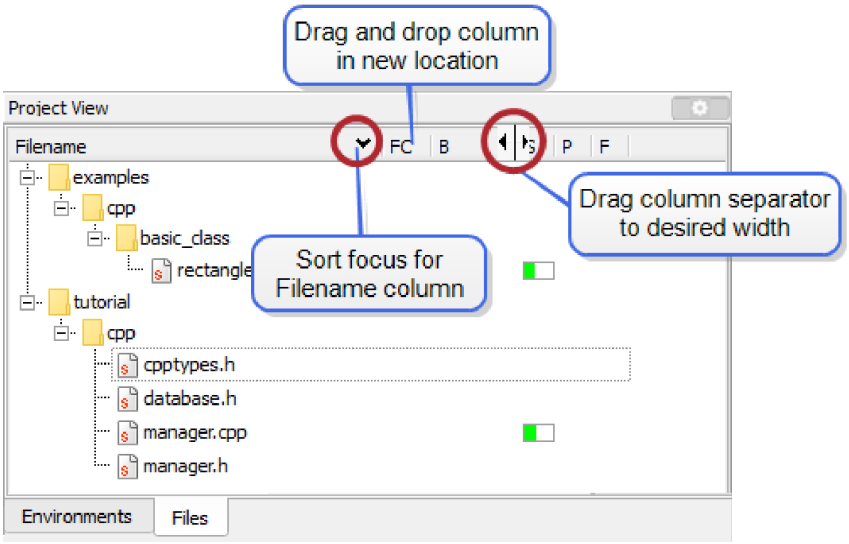


Hovering over an item displays the coverage percentage for coverage types.



Modifying the Files Display

You can change the order and width of the columns and sort the table of source files on the Files tab.



To Change Column Width

Column widths can be adjusted by moving the cursor to a column separator in the header and dragging it to the desired width.

To Sort the Table

The table can be sorted alphabetically by Filename. A column has sort focus when an arrow is displayed in the header for the column. Clicking on the arrow will toggle the sort between an ascending and descending sort. If the column does not have sort focus, click the column heading and the arrow will appear. Then click the arrow to toggle the sort.

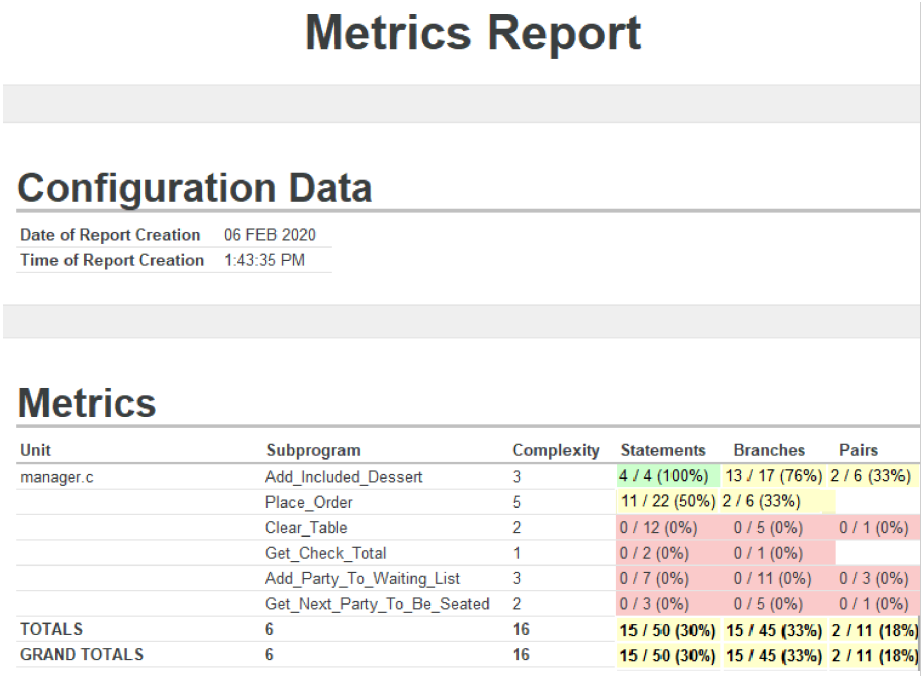
Viewing Coverage Reports

Reports are generated in HTML format by default. To modify the default report format, open the VectorCAST Options dialog and select the **Report** tab. In the **Report format** pane, select the HTML or Text radio button and select the **Apply** button.

The following sections discuss the available report types.

Metrics Report

The Metrics Report shows total coverage for the selected source files. Multiple source files may be selected. To access the Metrics Report from the Files tab, right-click on the source file and select **View => Metrics Report** from the context menu.



Metrics Report

Configuration Data

Date of Report Creation	06 FEB 2020
Time of Report Creation	1:43:35 PM

Metrics

Unit	Subprogram	Complexity	Statements	Branches	Pairs
manager.c	Add_Included_Dessert	3	4 / 4 (100%)	13 / 17 (76%)	2 / 6 (33%)
	Place_Order	5	11 / 22 (50%)	2 / 6 (33%)	
	Clear_Table	2	0 / 12 (0%)	0 / 5 (0%)	0 / 1 (0%)
	Get_Check_Total	1	0 / 2 (0%)	0 / 1 (0%)	
	Add_Party_To_Waiting_List	3	0 / 7 (0%)	0 / 11 (0%)	0 / 3 (0%)
	Get_Next_Party_To_Be_Seated	2	0 / 3 (0%)	0 / 5 (0%)	0 / 1 (0%)
TOTALS	6	16	15 / 50 (30%)	15 / 45 (33%)	2 / 11 (18%)
GRAND TOTALS	6	16	15 / 50 (30%)	15 / 45 (33%)	2 / 11 (18%)

The Metrics Coverage reports can be generated by using the **`--create-report`** command.

The syntax for the **`--create-report`** command is:

```
manage -p <project> --create-report=<aggregate|csv_metrics|metrics> [--
output=<PATH>] [--path=<PATH>]
```

where:

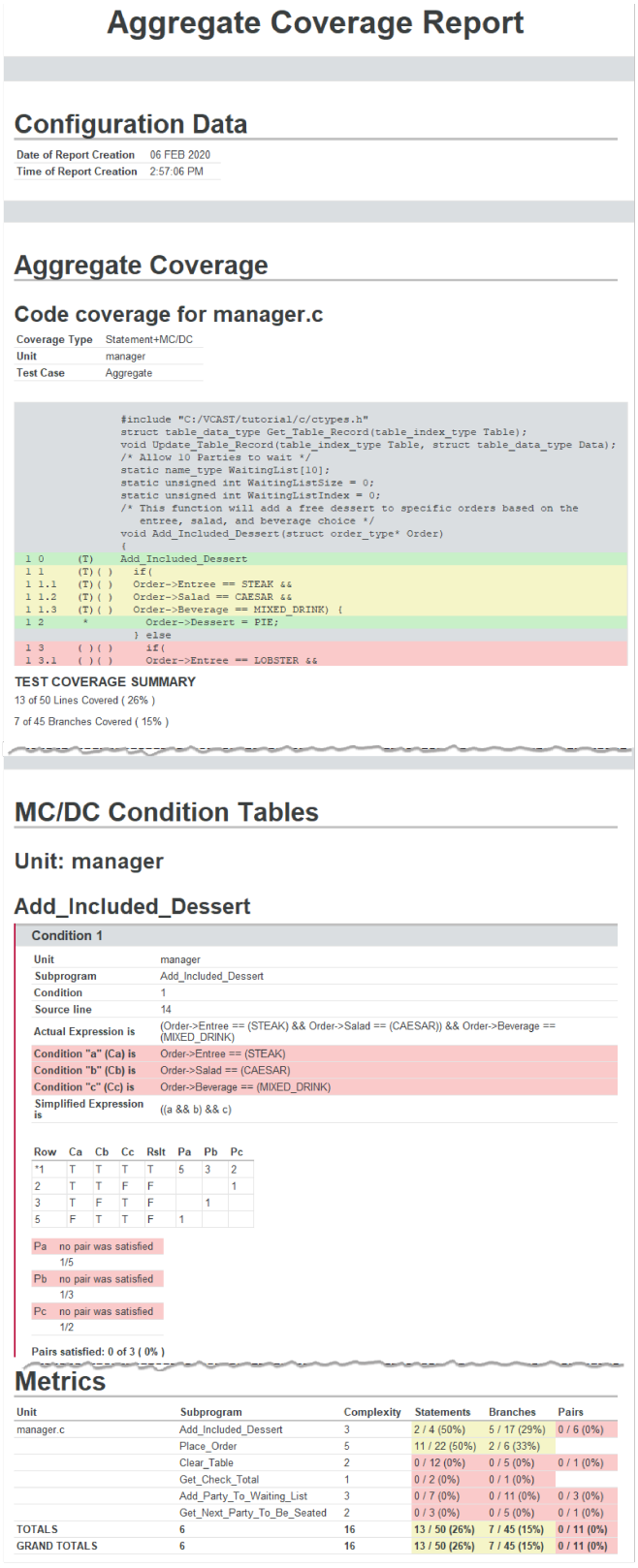
`<create-report>` is the report type: Aggregate, CSV_Metrics or Metrics.

`<output>` is the location to write the report.

`<path>` is the path to a source file or directory specifying the unit(s) to include in the report. May be provided multiple times on the command line.

Aggregate Coverage Report

The Aggregate Coverage Report shows a line or branch listing of the total coverage for the selected source files. Multiple source files may be selected. To access the Aggregate Coverage Report from the Files tab, right-click on the source file and select **View => Aggregate Coverage Report** from the context menu.



Aggregate Coverage Report

Configuration Data

Date of Report Creation	06 FEB 2020
Time of Report Creation	2:57:06 PM

Aggregate Coverage

Code coverage for manager.c

Coverage Type	Statement+MC/DC
Unit	manager
Test Case	Aggregate

```
                #include "C:/VCAST/tutorial/c/ctypes.h"
                struct table_data_type Get_Table_Record(table_index_type Table);
                void Update_Table_Record(table_index_type Table, struct table_data_type Data);
                /* Allow 10 Parties to wait */
                static name_type WaitingList[10];
                static unsigned int WaitingListSize = 0;
                static unsigned int WaitingListIndex = 0;
                /* This function will add a free dessert to specific orders based on the
                   entree, salad, and beverage choice */
                void Add_Included_Dessert(struct order_type* Order)
                {
1 0      (T)    Add_Included_Dessert
1 1      (T)( ) if(
1 1.1    (T)( ) Order->Entree == STEAK &&
1 1.2    (T)( ) Order->Salad == CAESAR &&
1 1.3    (T)( ) Order->Beverage == MIXED_DRINK) {
1 2       *       Order->Dessert = PIE;
                } else
1 3      ( )( )   if(
1 3.1    ( )( )   Order->Entree == LOBSTER &&
```

TEST COVERAGE SUMMARY

13 of 50 Lines Covered (26%)

7 of 45 Branches Covered (15%)

MC/DC Condition Tables

Unit: manager

Add_Included_Dessert

Condition 1

Unit	manager
Subprogram	Add_Included_Dessert
Condition	1
Source line	14
Actual Expression is	(Order->Entree == (STEAK) && Order->Salad == (CAESAR)) && Order->Beverage == (MIXED_DRINK)
Condition "a" (Ca) is	Order->Entree == (STEAK)
Condition "b" (Cb) is	Order->Salad == (CAESAR)
Condition "c" (Cc) is	Order->Beverage == (MIXED_DRINK)
Simplified Expression is	((a && b) && c)

Row	Ca	Cb	Cc	Rslt	Pa	Pb	Pc
*1	T	T	T	T	5	3	2
2	T	T	F	F			1
3	T	F	T	F		1	
5	F	T	T	F	1		

Pa no pair was satisfied
1/5

Pb no pair was satisfied
1/3

Pc no pair was satisfied
1/2

Pairs satisfied: 0 of 3 (0%)

Metrics

Unit	Subprogram	Complexity	Statements	Branches	Pairs
manager.c	Add_Included_Dessert	3	2 / 4 (50%)	5 / 17 (29%)	0 / 6 (0%)
	Place_Order	5	11 / 22 (50%)	2 / 6 (33%)	
	Clear_Table	2	0 / 12 (0%)	0 / 5 (0%)	0 / 1 (0%)
	Get_Check_Total	1	0 / 2 (0%)	0 / 1 (0%)	
	Add_Party_To_Waiting_List	3	0 / 7 (0%)	0 / 11 (0%)	0 / 3 (0%)
	Get_Next_Party_To_Be_Seated	2	0 / 3 (0%)	0 / 5 (0%)	0 / 1 (0%)
TOTALS		6	16	13 / 50 (26%)	7 / 45 (15%)	0 / 11 (0%)
GRAND TOTALS		6	16	13 / 50 (26%)	7 / 45 (15%)	0 / 11 (0%)

The Aggregate Coverage reports can be generated by using the **`--create-report`** command.

The syntax for the **--create-report** command is:

```
manage -p <project> --create-report=<aggregate|csv_metrics|metrics> [--
output=<PATH>] [--path=<PATH>]
```

where:

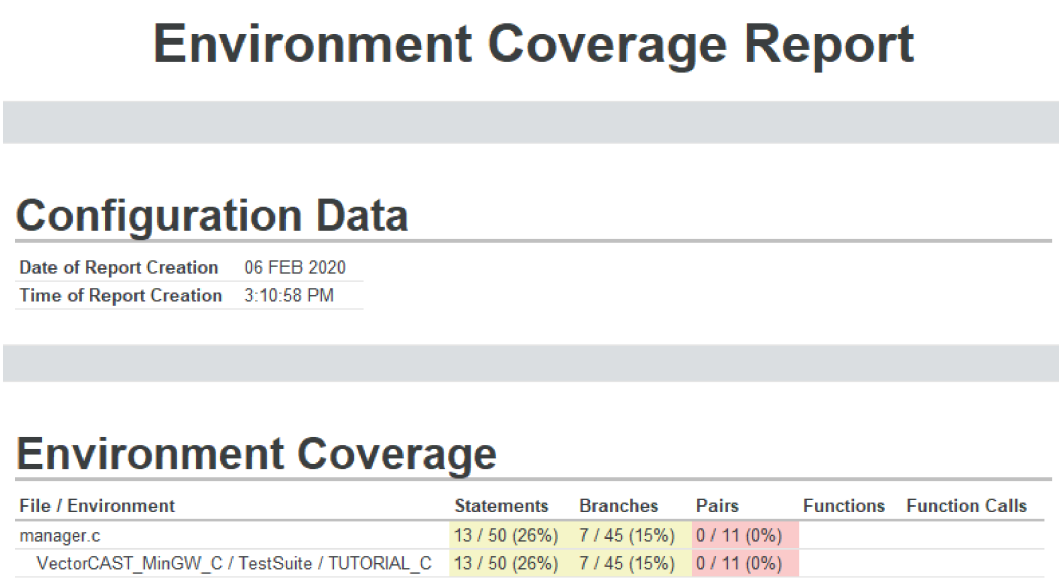
<create-report> is the report type: Aggregate, CSV_Metrics, or Metrics.

<output> is the location to write the report.

<path> is the path to a source file or directory specifying the unit(s) to include in the report. May be provided multiple times on the command line.

Environment Coverage Report

The Environment Coverage Report shows the total coverage from all environments that include the source file and eliminates duplicate covered lines or branches. Multiple source files may be selected. To access the Environment Coverage Report from the Files tab, right-click on the source file and select **View => Environment Coverage Report** from the context menu.



Environment Coverage Report

Configuration Data

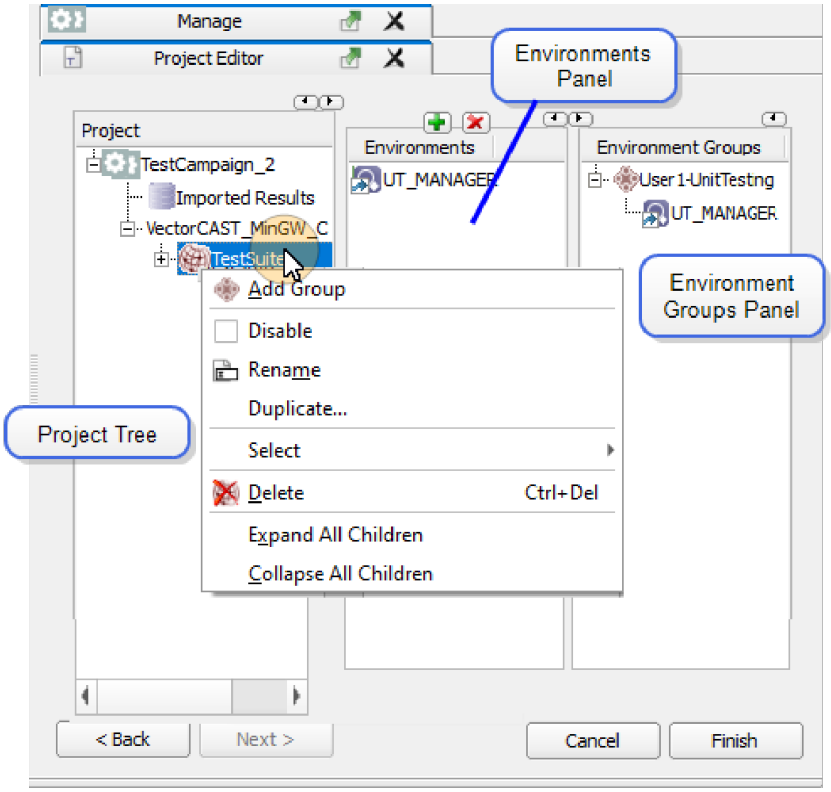
Date of Report Creation	06 FEB 2020
Time of Report Creation	3:10:58 PM

Environment Coverage

File / Environment	Statements	Branches	Pairs	Functions	Function Calls
manager.c	13 / 50 (26%)	7 / 45 (15%)	0 / 11 (0%)		
VectorCAST_MinGW_C / TestSuite / TUTORIAL_C	13 / 50 (26%)	7 / 45 (15%)	0 / 11 (0%)		

The Project Editor

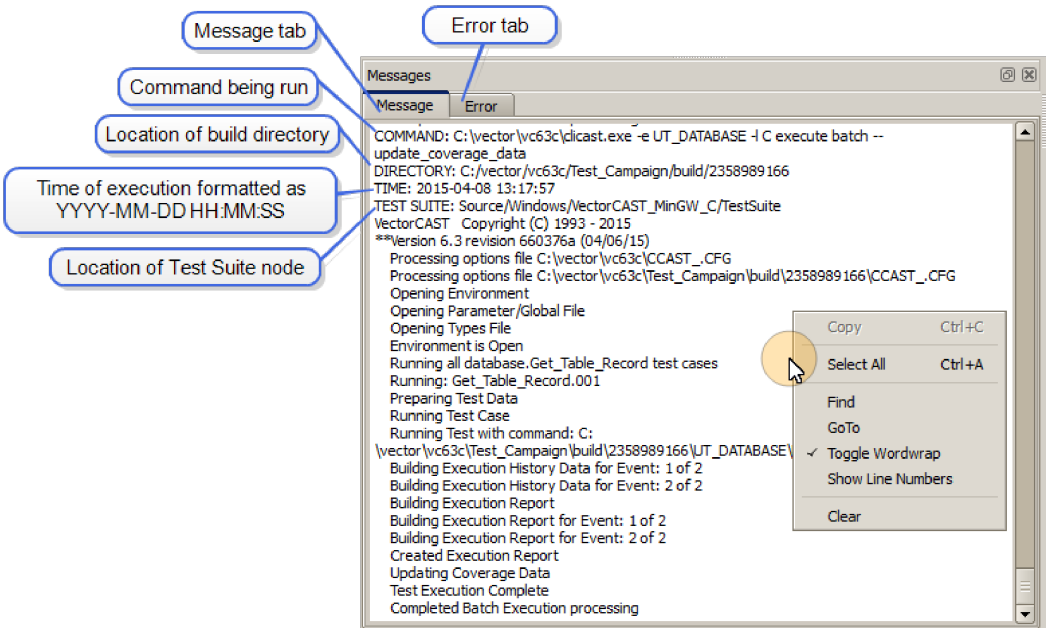
Access the Project Editor by selecting **Project => Update Project** from the Menu bar or from the New Project Wizard. You can make modifications to any item contained in the panels of the Project Editor by right-clicking and then selecting an operation from the context menu. You can add additional items, delete, or rename an item as well as create environment groups and add them to the Project Tree.



See"Using the Project Update Editor" on page 120 for more details on how to update an Enterprise Testing project.

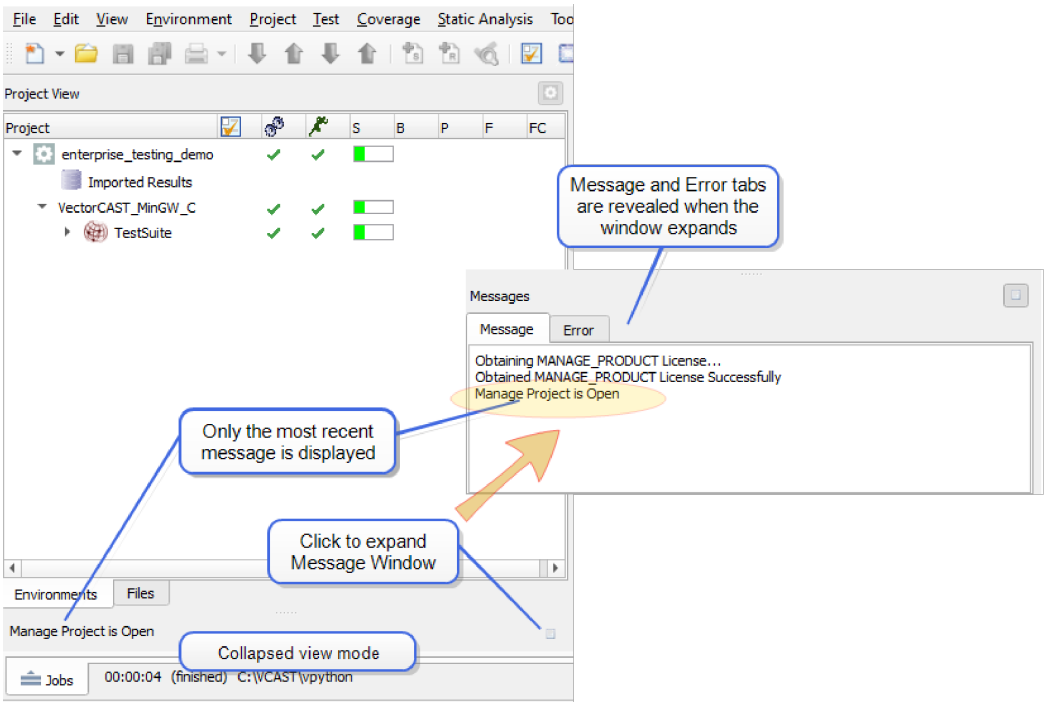
The Messages Window

The Messages window consists of the Message and Error tabs. The Message tab displays user status messages. The Error tab displays VectorCAST diagnostic messages.



The Message window is a docking window and can be moved within the main application window or float outside of the application 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.

To hide the Message window, select **View => Dock Control** from the Menu Bar and uncheck **Messages**. To make the Message window visible again, select **View => Dock Control** from the Menu Bar and check **Messages**, or alternately select **View => Default Layout** from the Menu Bar.

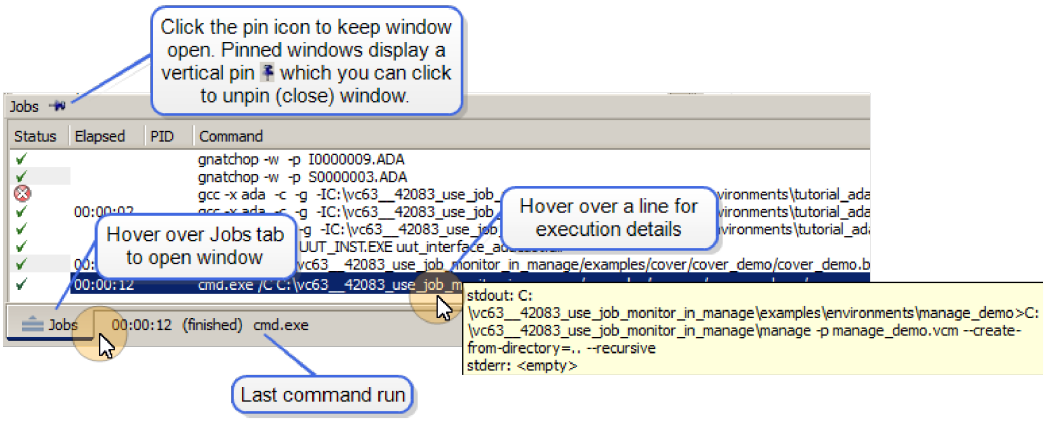
Use the right-click context menu in the Message window to **Clear, Copy** and **Select All** text in the window. Use the **Find** option to search within the messages and the **GoTo** option to navigate to a particular location. **Wordwrap** and **Line Numbers** can be toggled on and off via the right-click context menu.

Using the Jobs Window and Job 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.



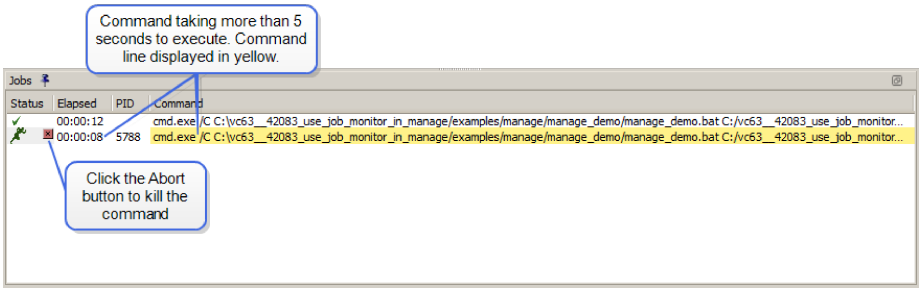
The Jobs window displays the full command line for every invocation of an executable by VectorCAST's back end. Such executables include the compilers, .bat files, python and any other commands called while building the environment.

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:

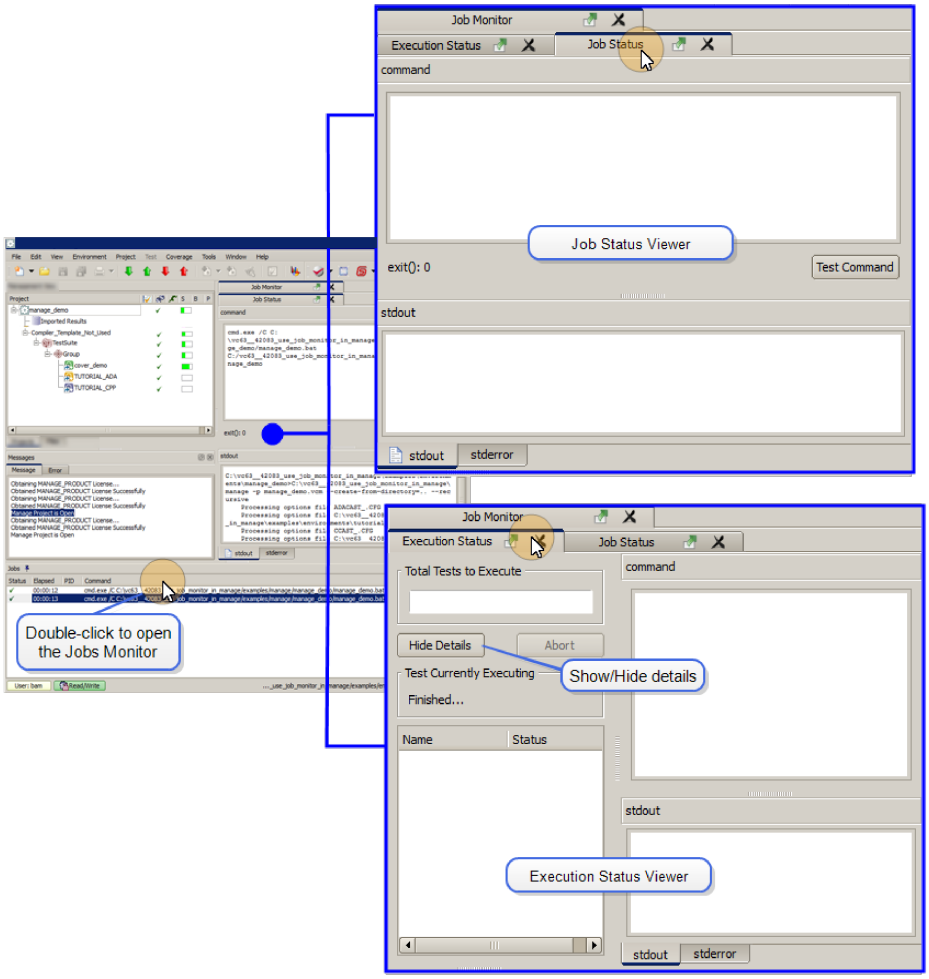
- = exit() code is 0
- = exit() code is non-zero
- = Click to abort job
- = Executing
- = Error
- = Monitored sub-process (used in Verbose mode)

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 the Job Status Viewer. The Job Status Viewer displays details about a command.

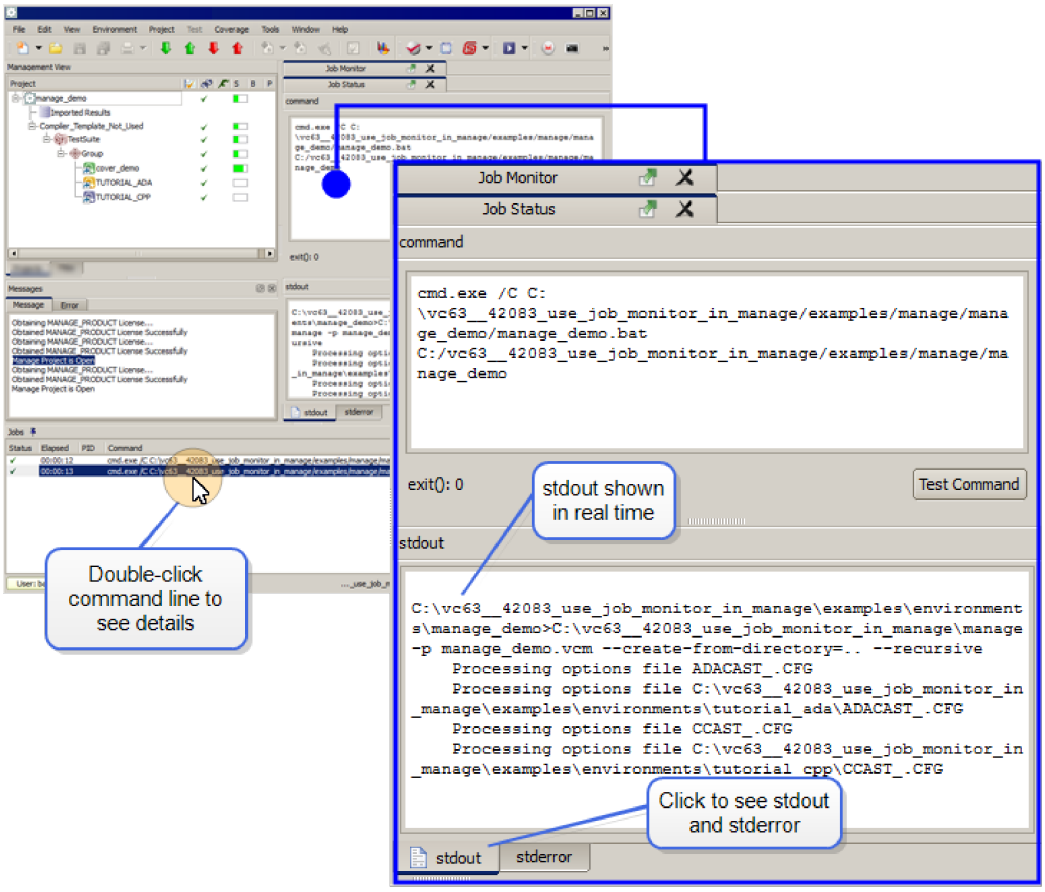


To open the Job Monitor and the Job Status viewer, double-click a command line in the Jobs window or right-click a command line and select **Job Status** from the context menu. The Job Status Viewer opens in the MDI window.

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.

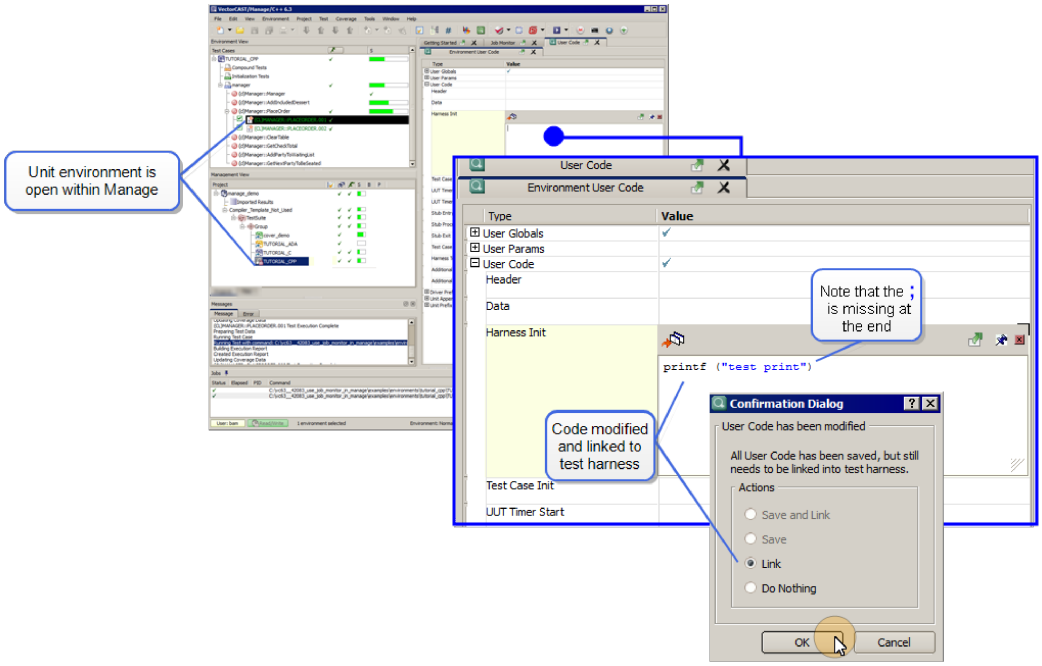
Debugging Using the Job Status Viewer

The Job Status viewer shows failed commands 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.

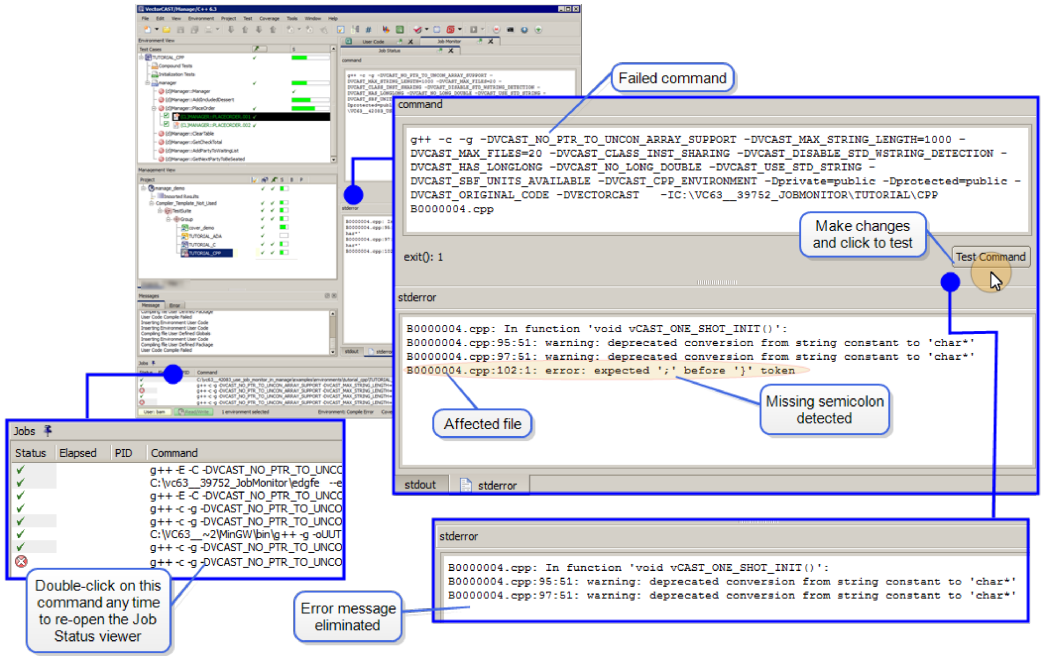
This debugging feature provides a way to diagnose failing commands. It does not affect the state of the environment or resolve errors caused by the original command failing to run.

Note: In the example shown below, the unit environment is open in Enterprise Testing. This puts the command in its correct context. Note that if Enterprise Testing is open but the unit environment is not open, the **Test Command** button will return an error because the command will not be in the correct context.

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.



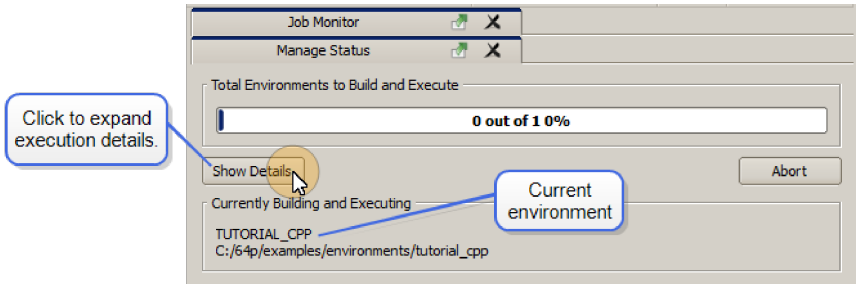
Note: If the command does not execute properly in the context of the current working environment, then it must be set up by the user in the necessary context. This may mean changing the working directory, setting environment variables, creating temporary files or opening an environment.

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.

Manage Status Viewer

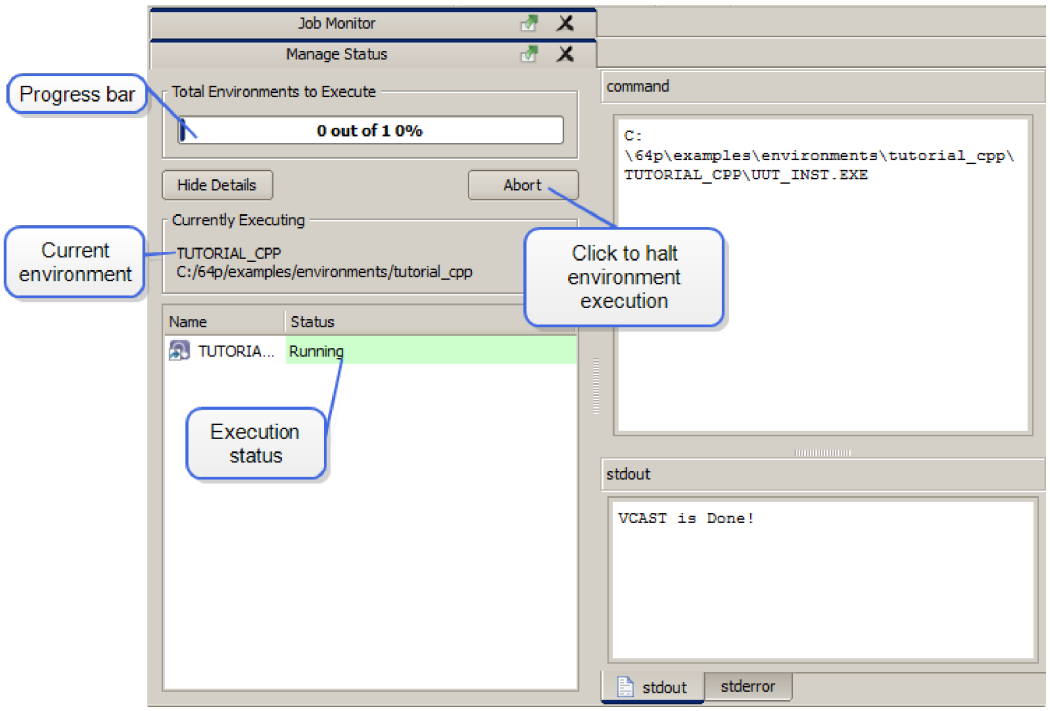
The Manage Status viewer is the counterpart to the Job Status viewer. The Manage Status viewer provides detailed real-time environment build and execution information, showing the progress, outcome and output for environments in a VectorCAST project.

The Manage Status viewer opens automatically when a Build, Execute or Build/Execute action is performed on an environment. Click the **Show Details** button to display the full details of the execution in real-time.

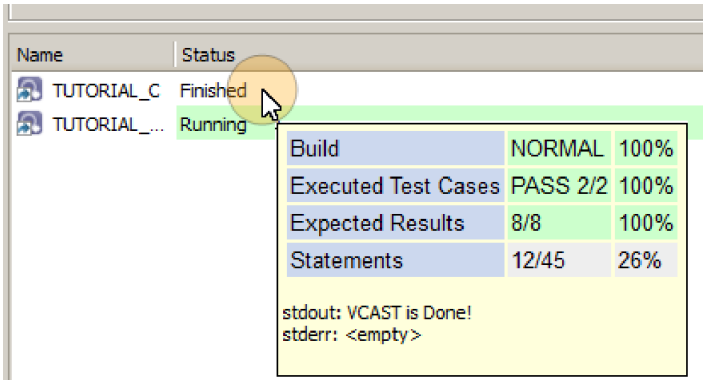


A progress bar is displayed on the top left showing the number of environments remaining to build

and/or execute. The name of the currently executing environment is displayed beneath. An **Abort** button is available to interrupt the currently running process if desired, and to return control to VectorCAST. Note that VectorCAST does not always have control over processes that are running, and using the **Abort** button could potentially leave the environment in an unstable state.

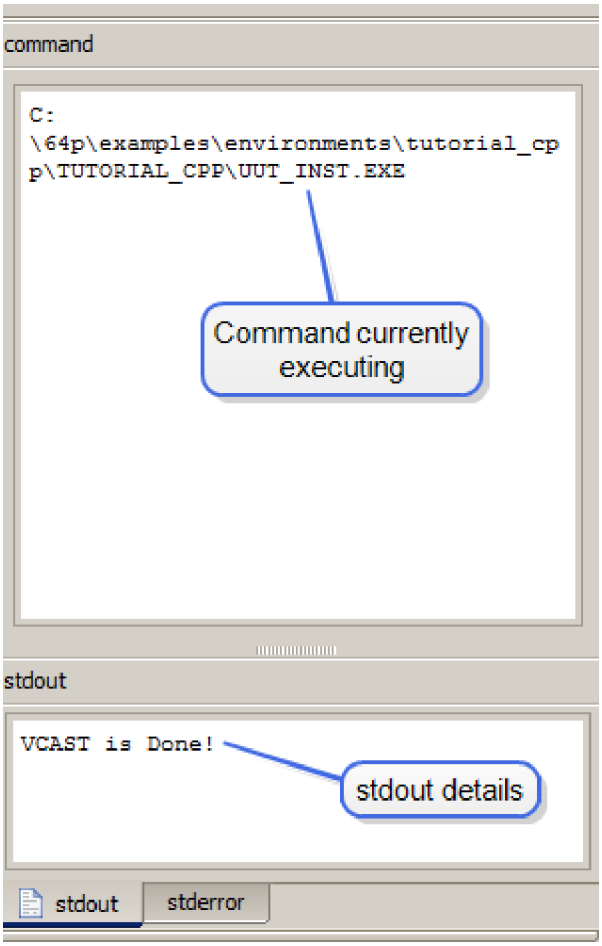


When viewing full execution details, as each test completes it is listed on the bottom left along with its status. Hover over the status of an environment to see details of the build and execution.



Double-clicking on an environment opens the environment from within VectorCAST.

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


command
C:
\64p\examples\environments\tutorial_cp
p\TUTORIAL_CPP\UUT_INST.EXE
Command currently executing
stdout
VCAST is Done!
stdout details
stdout
stderror

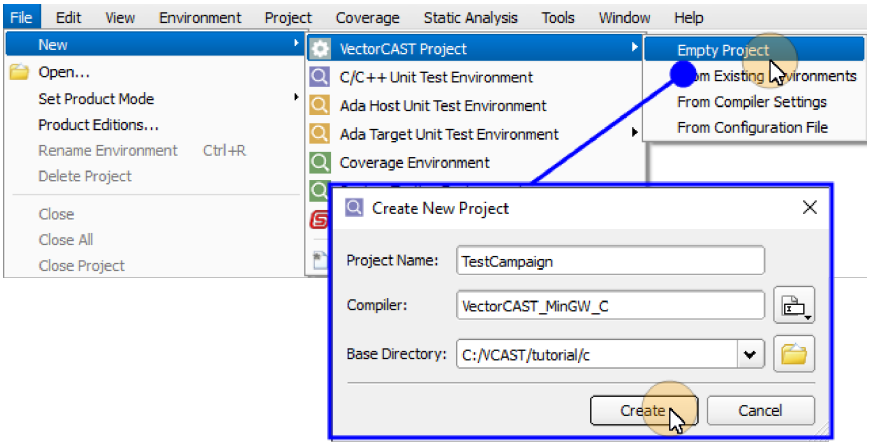
Project-Related Tasks

There are several tasks associated with creating and maintaining a typical Enterprise Testing project. Several of the most common tasks are discussed in detail below.

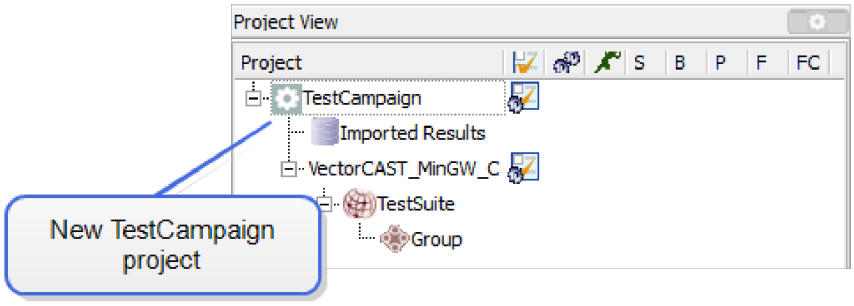
Creating an Empty VectorCAST Project

Enterprise Testing provides the ability to quickly create a new VectorCAST project and build test suites and environments.

From the Menu Bar, select **File => New => VectorCAST Project => Empty Project**. On the Create New Project dialog, enter the Project Name, select the compiler from the drop down menu on the right, and enter the path to the base directory for the project. Click the **Create** button.

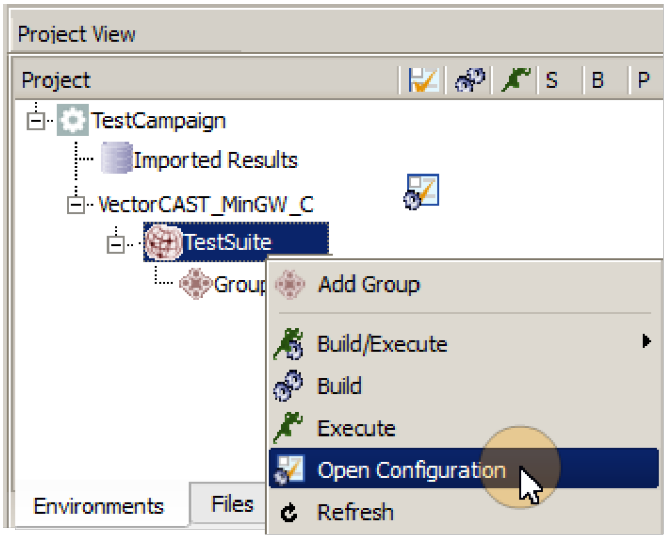


The new project is displayed in the Project Tree.

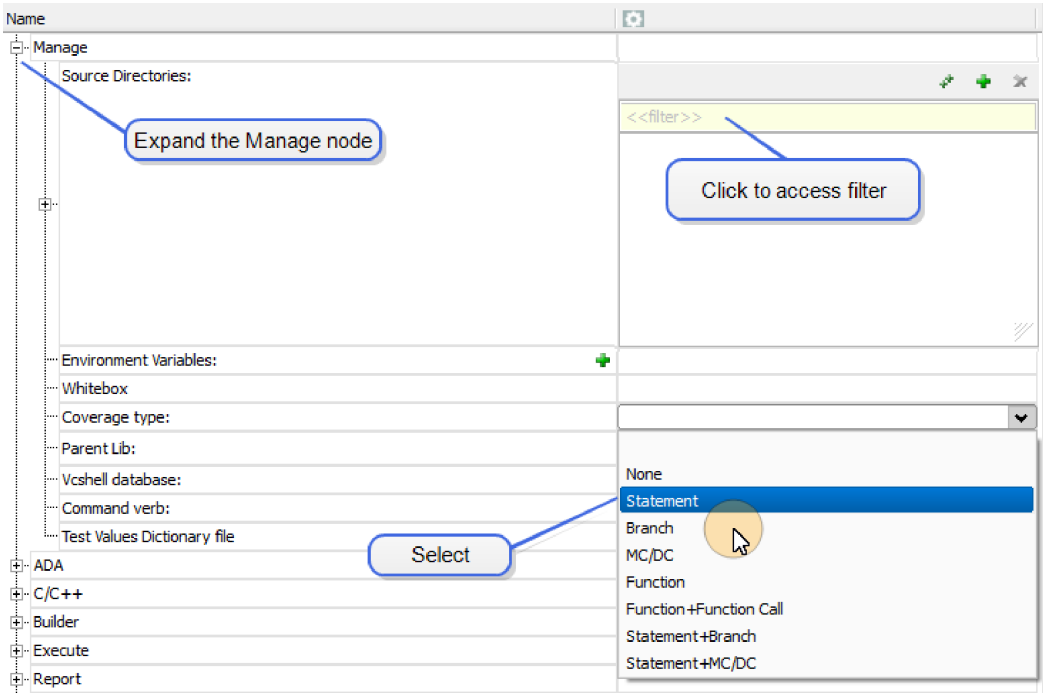


Setting Default Options for Your Project

Use the Configuration Editor to set default options for your project. Access the Configuration Editor by right-clicking on a node in the Project Tree and selecting **Open Configuration** from the context menu. In the example below, we will set the default coverage mode for the TestSuite node to Statement coverage.



To set coverage type, expand the Manage node on the Configuration Options Editor. Use the Coverage Type drop down menu to select the appropriate coverage instrumentation to perform. Save the changes. By setting the coverage mode at the TestSuite node level, all environments under that node will inherit the coverage mode setting.

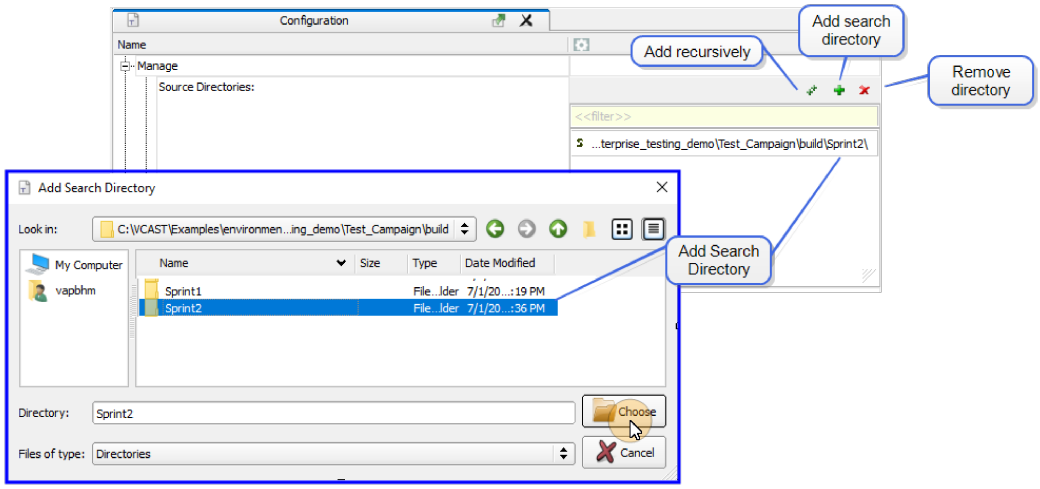


To set a source directory, click in the column to the right of the Source Directories node in the Configuration Options Editor to bring focus to the column.

A filter is provided at the top of the Source Directories pane making it easier to filter out some results when working with a large set of directories. 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.

Use this location to add or delete testable source directories, library include directories, or type-handled directories. Use the button options to add a search directory, add directories recursively, or remove a directory. Use the Add Search Directory or the Add Search Directory Recursively dialogs to browse to the source directory location and select the **Choose** button.



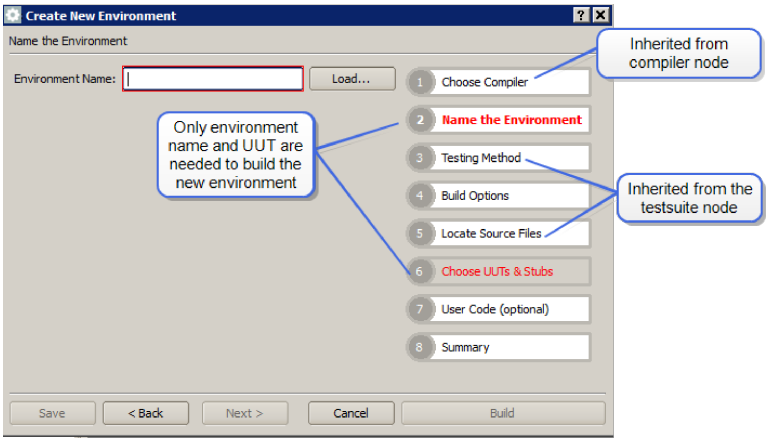
Creating a New Unit Test Environment

Unit Test environments can be created one of two ways: either interactively using the New Environment dialog or by using an existing environment script. Both methods are discussed below.

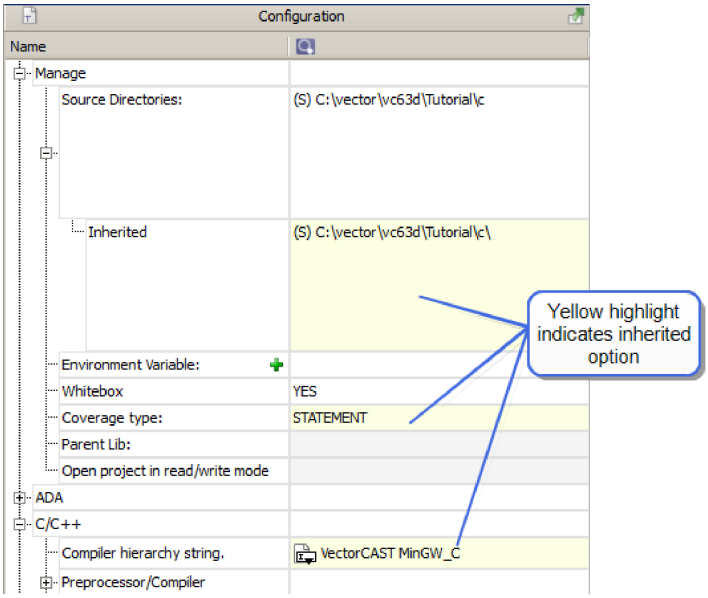
Create a New Environment Interactively

To create a new environment interactively, right click on the Group node and select **Create Unit Test Environment => Interactive** from the context menu. Notice in the New Environment dialog that several options are automatically filled in. This is because they are inherited from the parent nodes.

In our example below, the compiler is set in the VectorCAST_MinGW_C compiler node and the Statement Coverage and Source Files are set in the Test Suite node. In this instance, only the Environment Name and UUTs and Stubs need to be entered by the user.

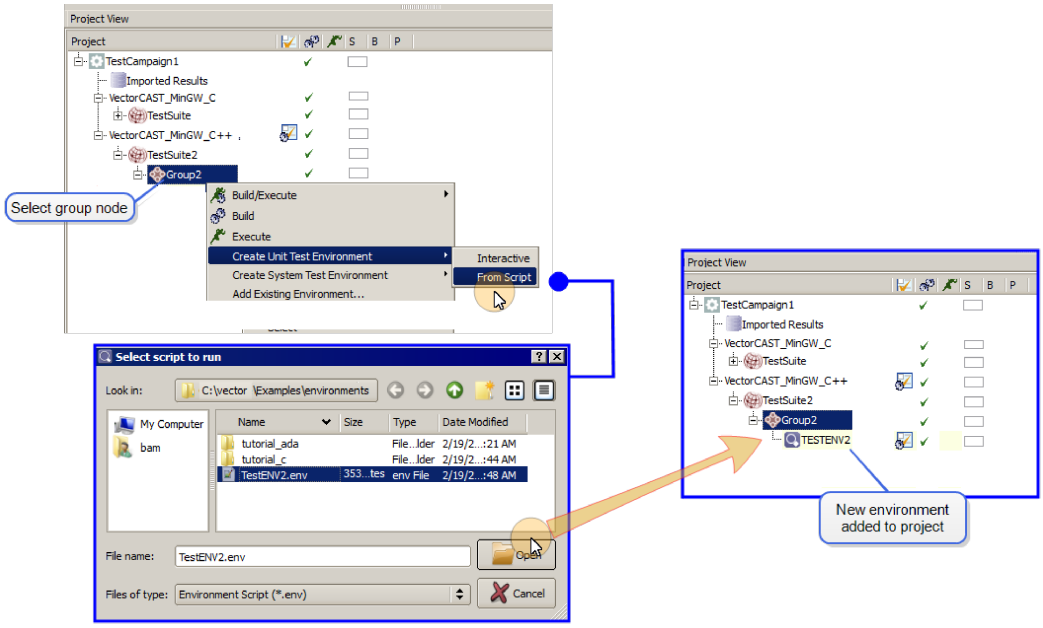


After clicking the **Build** button, the environment is created. Note that when viewing the newly created environment in the Configuration Options Editor, inherited options are highlighted in yellow.



Create a New Environment From Script

To create a new unit test environment from an existing environment script (.env), right-click on a Group node in the Project Tree and select **Create Unit Test Environment => From Script** from the context menu.



Browse to the location of the .env script and select the **Open** button. After clicking the **Open** button, the environment is created.

Note that environments must have unique names. If an environment already exists in the project that has the same name as the new one being created, an error message is produced and the environment is not built.

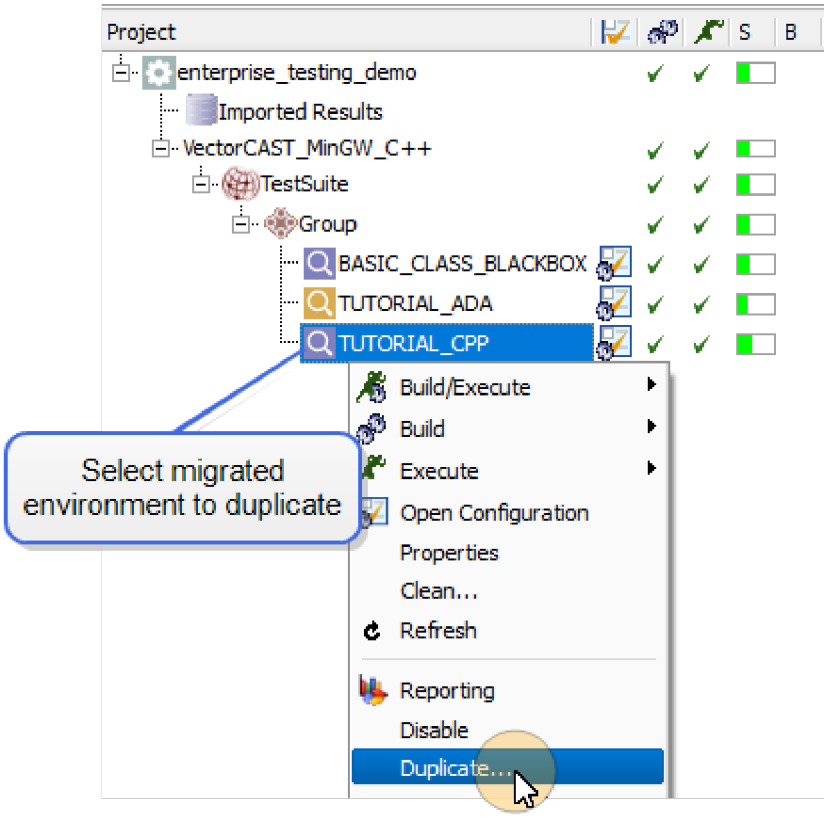
The new environment inherits its configuration from the nodes above the Group node, as well as any settings you configure in the .env script itself.

Duplicating a Unit Test Environment

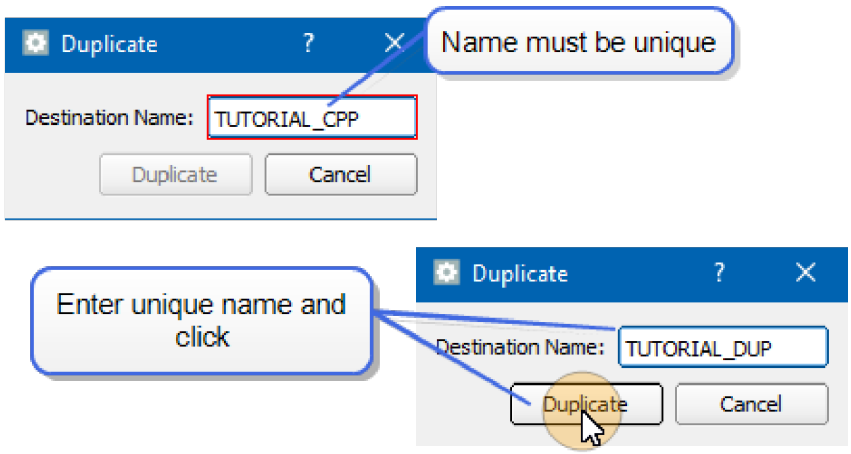
Unit Test environments can be created as a duplicate of an existing migrated Unit Test Environment.

Note: Only migrated Unit Test environments are available for duplication.

To duplicate an environment, right-click on a migrated environment and select **Duplicate...** from the context menu.

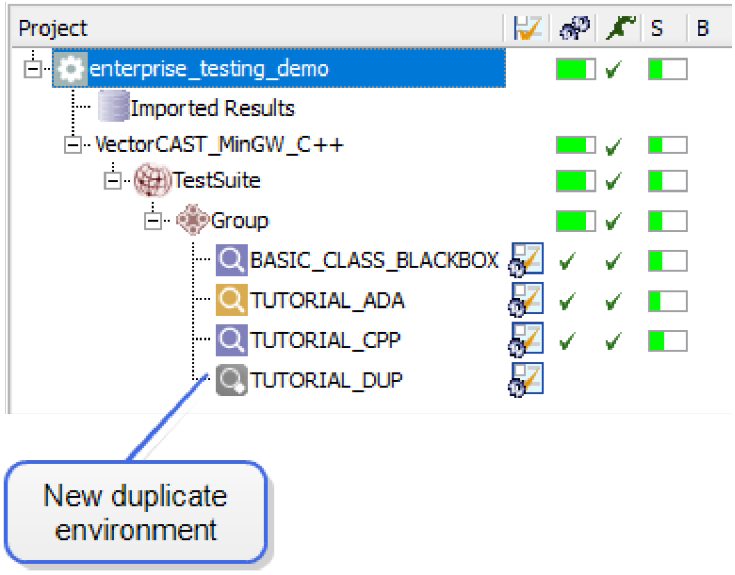


Enter a unique name for the environment in the Destination Name field of the Duplicate dialog.



Note that the environment name must be unique within the context of the project. Names that are not unique are bordered with a red highlight.

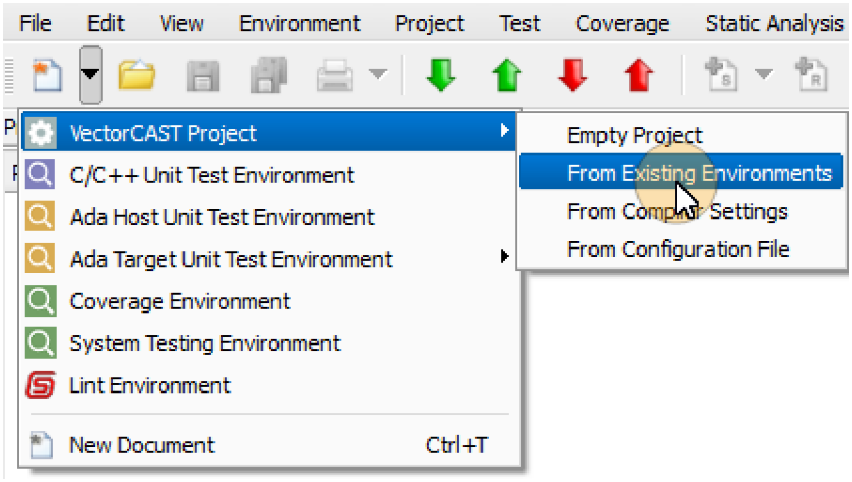
Click the **Duplicate** button. VectorCAST creates the new environment and adds it to the same group context as the original environment.



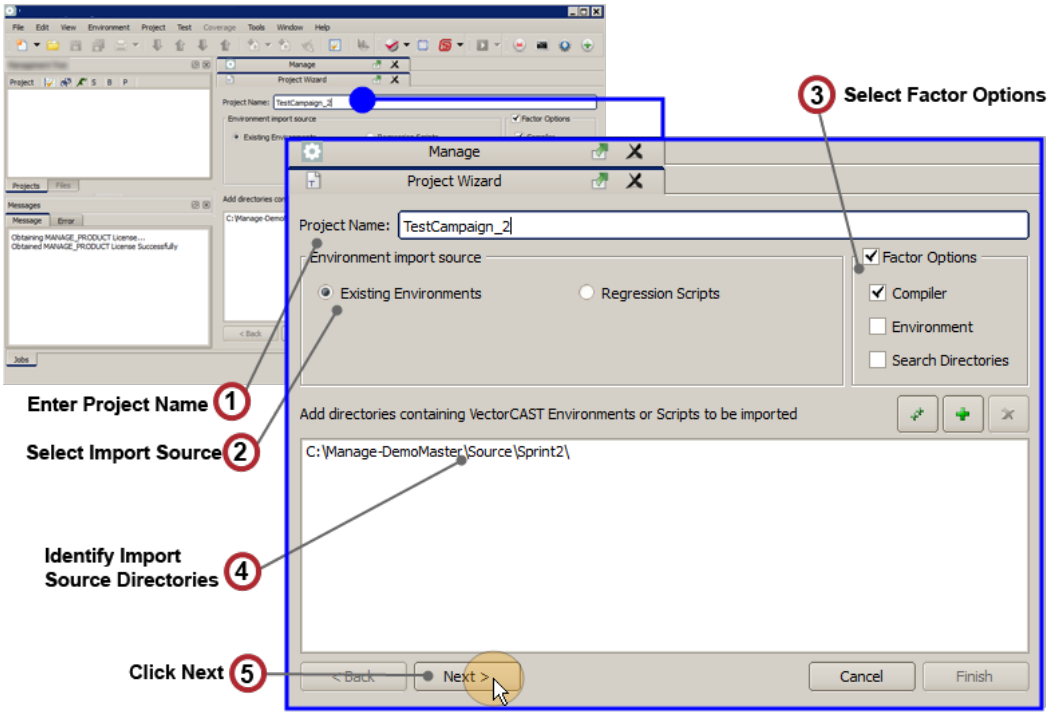
Note: Specialized test cases and specialized configurations are not applied to the new duplicate environment.

Creating a VectorCAST Project From Existing VectorCAST Environments

New VectorCAST Projects can be created using previously created environments. To begin, select **New => VectorCAST Project => From Existing Environments** from the Toolbar.



The Project Wizard opens in the MDI Window. Use the following steps to build a Project from existing VectorCAST Test Environments:



1. **Enter Project Name:** Give the project a unique name. Any space characters you type are converted to "_" characters. A directory and .vcm file are created with the project name and placed in the working directory. If a project already exists in the working directory with the name you chose, the name entry is outlined in red. The tool-tip asks you to choose a unique name for the project.
2. **Select Environment Import Source:** Specify whether to import existing environments or regression scripts. The import source is selected using the radio buttons in the Environment Input Source panel of the Wizard.
   - *Existing Environments:* The Wizard searches the directories for existing environments. It detects pairs of .vce files and directories with matching names. Environments that have a build error are not imported.
   - *Regression Scripts:* The Wizard searches the directories for sets of regression files: .bat/.csh and env. All option values are read from the .bat/.csh files.
3. **Select Factor Options:** Factoring options detects settings that are common to more than one environment and then groups those environments together, allowing you to easily reuse environments in multiple contexts in the VectorCAST project. Factoring options are set to None by default. See "Option Factoring in the Project Wizard" on page 84 to learn more.
4. **Identify Import Source Directories:** You must identify a set of directories where the wizard looks for VectorCAST environments or VectorCAST regression script files. You may add these directories one at a time or recursively.

    Use the Add Directory button to add a single directory as an import source. The wizard searches the directories in the order they are added.

    Use the Add Directory Recursively button to add multiple directories to be searched. Use

the directory navigation dialog to set the top most directory. This directory and all below it are searched for import sources.

By default, VectorCAST adds a maximum of 100 directories for import sources. When this limit is reached, a dialog appears which allows you to terminate or continue adding.

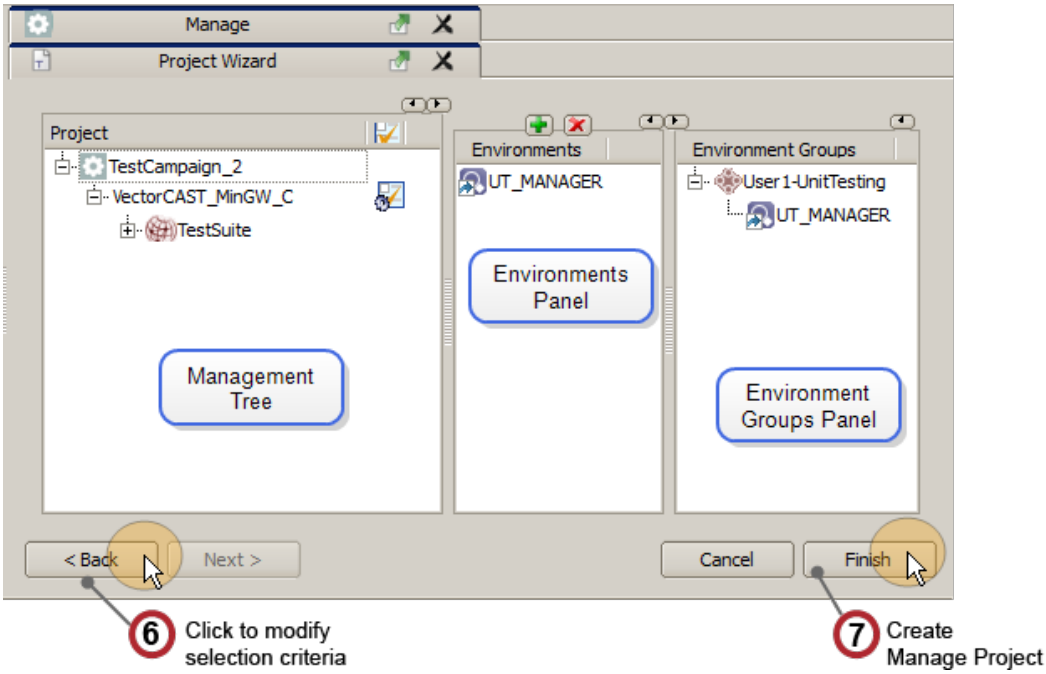
Only directories that contain VectorCAST environments or VectorCAST regression scripts are added to the list. While determining if a directory should be added to the list, the Wizard:

> disregards VectorCAST backup (.BAK) environments

> disregards VectorCAST project build directories

Remove Directory button: To remove a directory from the list, first select the directory, then click the **Remove Directory** button. If no directory is selected, the **Remove Directory** button is dim. Removing a directory does not affect the environments that have already been imported.

5. **Initiate Import**: Click the **Next** button. VectorCAST automatically imports environments using the criteria set on page one.

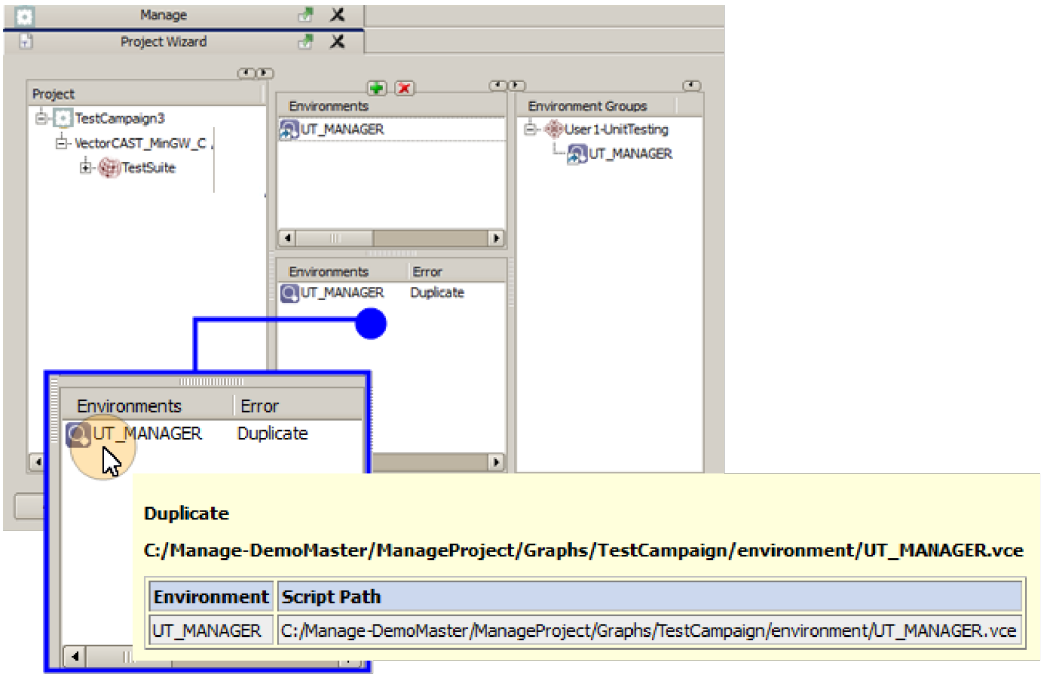


6. **Make Edits to the VectorCAST Project:** To modify the selected criteria, select the **Back** button. Make changes to factoring, import source or search directories as needed and click the **Next** button to initiate the import process again. See "Using the Project Update Editor" on page 120 for more information on making edits to the VectorCAST Project.

7. **Click Finish** to create the VectorCAST project. VectorCAST will automatically build the new Project Tree.

Wizard Import Errors

If an environment is found but cannot be imported for some reason, it is listed in a small panel at the bottom of the Environments panel. The Environment name and the error are listed. Hover over the error to see the directory where the environment was found in a tool-tip. You can select an error message and right-click to remove the error message from the list.



Error messages:

Duplicate

The Wizard has found a duplicate environment name.

Failed to import existing environment

The Wizard has found VectorCAST Test Environment file (.vce or .vcp) without a matching directory, or a Test Environment that has an abnormal build status.

Invalid regression script

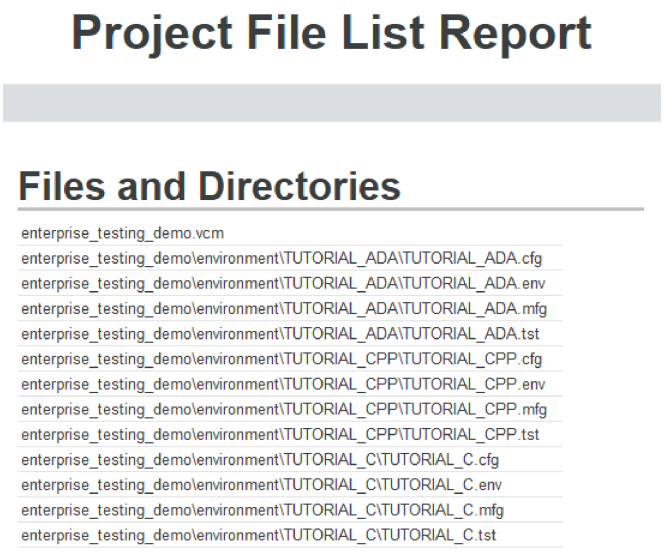
The Wizard has found an inconsistency in the regression scripts, such as mismatched names, or a missing compiler tag.

Opening a VectorCAST Project

To open an existing VectorCAST Project select **File => Open** from the Menu Bar and navigate to the location of the project's .vcm file. Select **Open**. Alternately, if the project has been opened recently, select **File => Recent Environments** and select the project from the list.

The Project File List

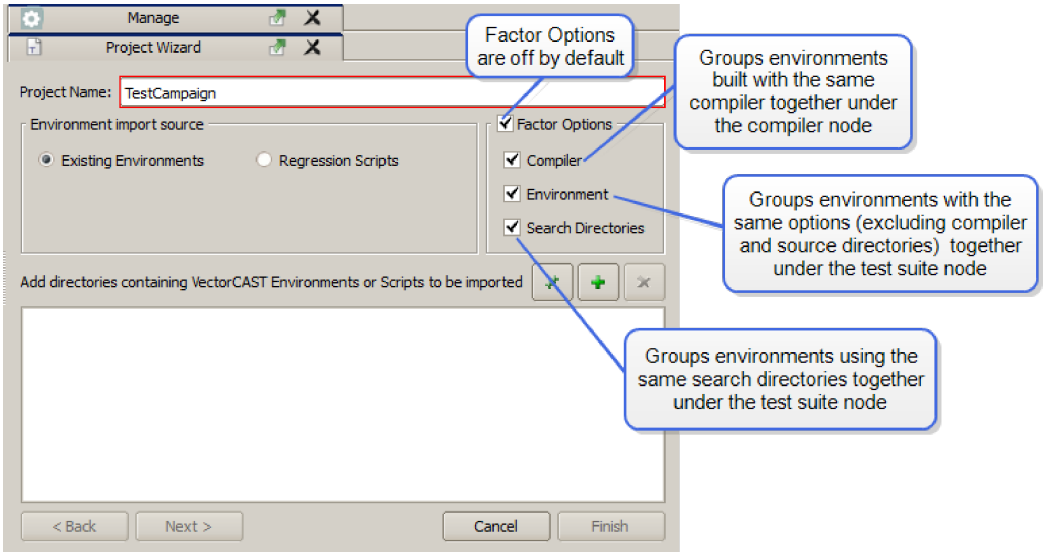
To open the Project File List Report, select **Project => Project File List** from the Menu Bar. The Project File List Report displays a list of project files and directories. For monitored environments, the `.vcm` file is listed. For migrated unit environments and migrated Cover environments, all files and directories are listed. The example below shows the Project File List for migrated unit test environments.



Note: The Project File List can be useful in identifying the regression files to save for projects with migrated environments.

Option Factoring in the Project Wizard

Factoring options detects settings that are common to more than one environment and then groups those environments together, allowing you to easily reuse environments in multiple contexts in the VectorCAST project. Factoring options are set to None by default.

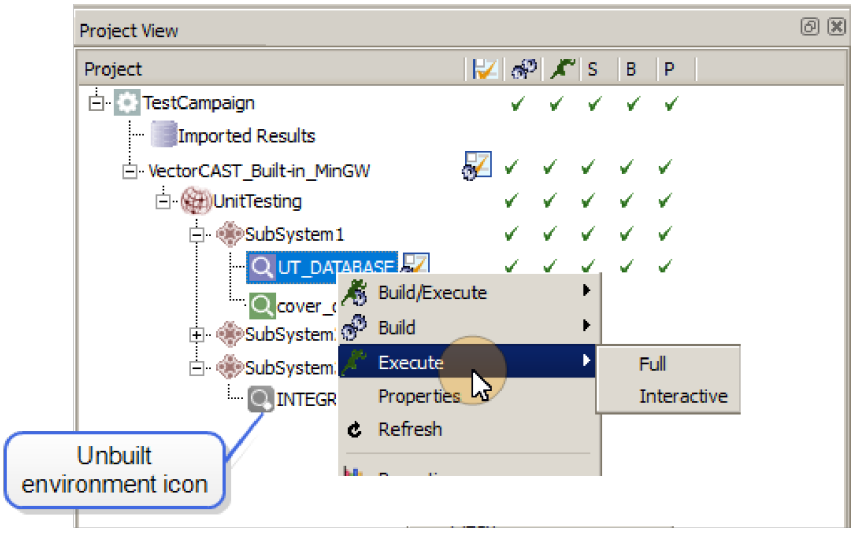


There are three types of factoring available:

- *Compiler*: If factor “Compiler options” is selected, the wizard factors the compiler-specific options to a compiler node. Library include directories are considered a compiler option. If two environments have been built with the same compiler, they are both placed under the same compiler node in the Project Tree. Factoring compiler options makes it easy to change a compiler option or library include directory and have the change affect all environments under that compiler node.
- *Environment*: Environment options are factored to the test suite level and include all options except compiler options and source directories. Two environments with the same environment options are placed in the same test suite. Factoring environment options makes it easy to change the coverage type, whitebox, or any other configuration option at the test suite level and have the change affect all environments in that test suite.
- *Search Directories*: An environment’s Search directories and Type-handled directories are factored to the Test Suite level. Factoring Search Directories makes it easy to change the source code baseline used to build the unit test environments.
- *None*: None is the default setting. If no factoring is performed, then all environments are imported and keep their compiler options, environment options, and search directory options at the environment node in the Project Tree.

Building and Executing Tests

Environment build and execution operations are accomplished by selecting either individual or multiple nodes on the Project Tree, and using the right-click context menu to select **Build/Execute**, **Build**, or **Execute** menu options.



Alternatively, you can select **Project => Build, Project => Execute,** or **Project => Build/Execute** from the Menu Bar. These menu selections all do full builds and executions.

A gray environment icon indicates that the environment has not yet been built and therefore has no status.

VectorCAST scans the code changes since the last test run and finds the sub-set of tests that are affected by the change and automatically re-runs only those tests. See "Change-Based Testing " on page 172 for more information.

Build/Execute Options

You can combine both the build and execute operations into one operation for environments by choosing **Build/Execute** from the right-click context menu. You have the option to perform either a Full Build or an Incremental Rebuild from the Build/Execute sub-menu. A Full Build builds the environment from scratch and then executes the tests. An Incremental Rebuild will perform only the work that needs to be done based on any changes.

Build Option - Full Build

A Full Build builds the environment from scratch and then executes the tests. To run a Full Build, highlight the desired nodes and select the **Build** option from the **Project => Build** menu or open the right-click context menu and select **Full** from the **Build** sub-menu. A Full Build is built with local and inherited options using the existing environment script and test cases that are part of the VectorCAST project.

Build Option - Interactive Build

When selecting a *single* environment for the build operation, use the right-click context menu and select **Interactive** from the **Build** sub-menu.

If you select an environment that has not yet been built, a *Create New Environment* wizard appears. If the environment has been previously built, the *Update Environment* wizard appears.

Make the necessary changes and click the **Build** (or **Update**) button on the wizard. When the build completes, the environment remains open allowing you to interactively add, modify, and execute tests, and the Status Panel is updated with the build status. If the build fails, the environment name is displayed in red.

Execute Option - Full Execute

Choosing **Execute => Full** executes all test cases in batch mode in all test environments for the selected branch of the VectorCAST project.

An execution history, as well as an execution report, is generated as a result of performing execute. If coverage was selected for the environment, coverage results and reports are generated as well. Execution Status is provided on the Status Panel in the Execution Status Column.

Execute Option - Interactive Execute

When selecting a single environment for the execute operation, use the right-click context menu and select Interactive from the Execute sub-menu.

After selecting **Interactive Execute**, the environment is opened, the Execution Status Viewer is opened and the test cases are automatically executed as a batch.

When the execution completes, the status of each test and the execution details are displayed in the Execution Status Viewer. In the Project Tree the *Execute Status* column is updated for the environment. Using the Environment View, you may interact with the environment to add, delete or modify environment tests.

Execute Option - Incremental Execute

An Incremental execute performs only the work that needs to be done, based on the current state of the project.

To automatically run an Incremental execute on an environment, right-click on the Environment node and select **Execute => Incremental** from the context menu. For more information, see "Incremental Build and Execute" on page 179.

Configuration-Based Test Cases

VectorCAST provides a "Specialized" test case attribute which is used to bind a test case to a specific configuration. This is helpful in managing test cases that are specific to a particular configuration.

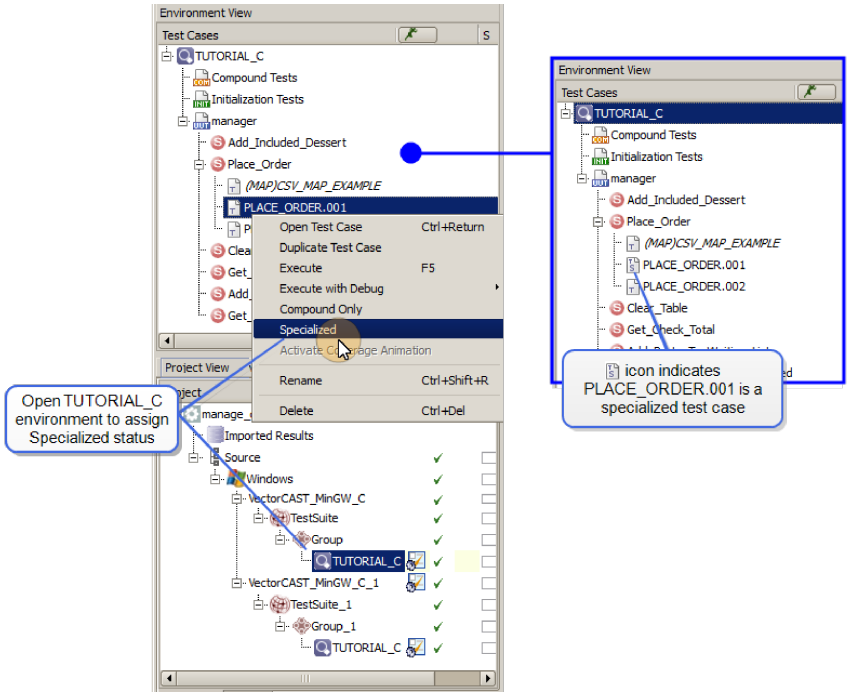
For example, a specialized test case may be used to test source code that contains conditional directives (such as #if or #ifdef). Conditional directives are used to selectively include or exclude blocks of code from a source file, based on the result of an expression or the value of an identifier.

To test source code containing conditional directives, the source file must be compiled with different configurations in order to compile the different line groups and then be put into the environment's translation unit. VectorCAST allows a single environment to be created for a source file and tested using the different configurations required to compile the different line groups.

Because the translation unit will vary based on the compiler's configuration, the test cases are customized, or specialized, to a single configuration by assigning the status Specialized to the test

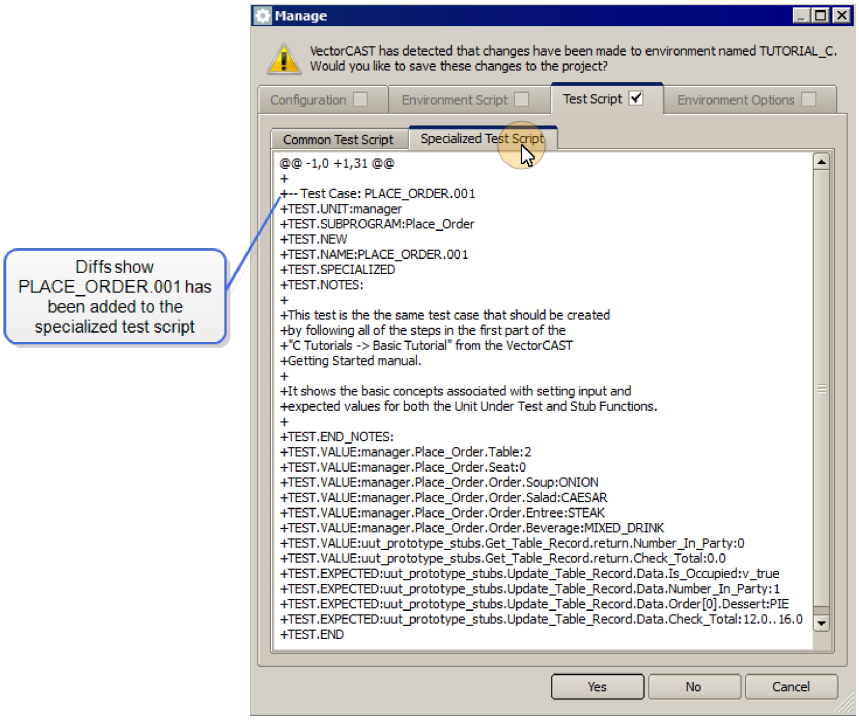
case.

To assign Specialized status to a test case in a migrated unit test environment in a VectorCAST project, open the environment from within VectorCAST, then right-click on the test case in the Test Case Tree and select **Specialized** from the context menu.

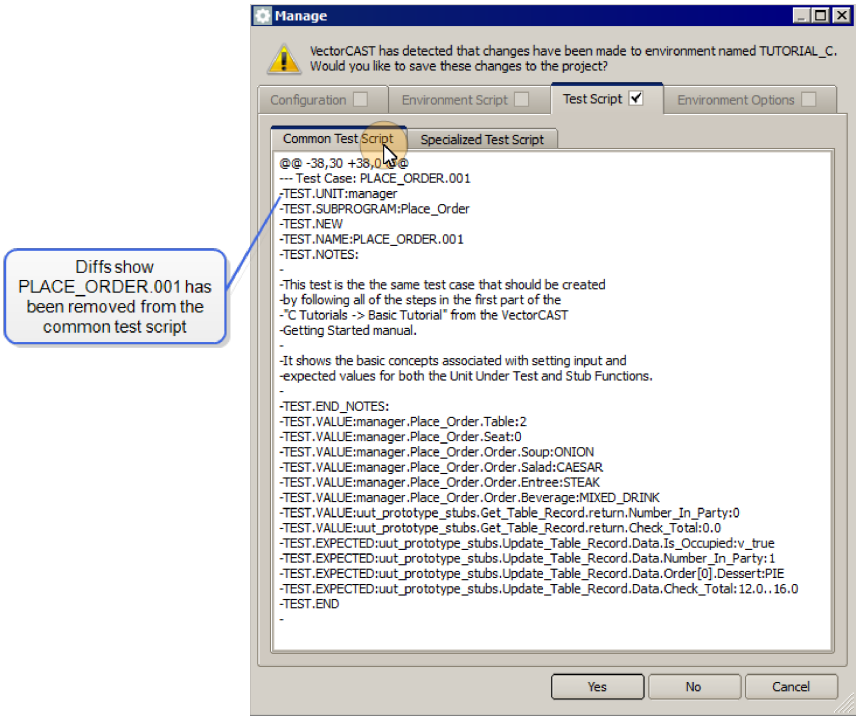


The specialized test case icon is displayed next to the test case name. A specialized test case is only visible in the environment in that specific configuration.

Once a test case is assigned Specialized status and the environment is closed, VectorCAST detects the changes to the environment and displays the Diff dialog. In our example below, the Test Script tab is open and the Specialized Test Script sub-tab is active, showing that test case PLACE_ORDER.001 has been added to the Specialized Test Script. If you normally commit the .tst files to your Source Code Management, you should commit this new Specialized test script as well.



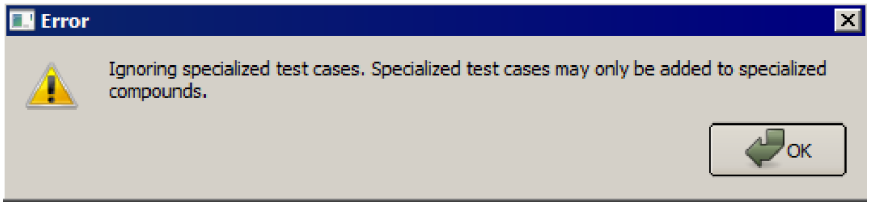
Click on the Common Test Script sub-tab and note that PLACE_ORDER.001 has been removed from the Common Test Script as a result of our assigning Specialized status to the test case.



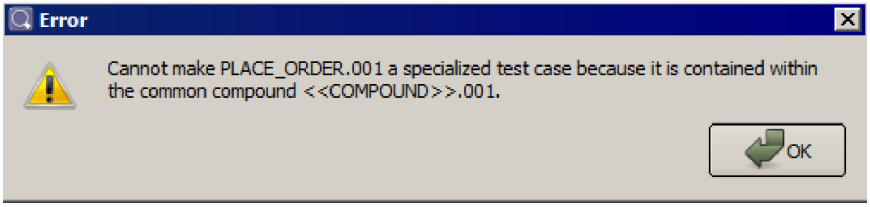
Specialized Compound Tests

Specialized test cases can only be added to specialized compound tests and cannot be added to a common compound test. In other words, because common compound test cases are shared across environment contexts they can only contain common tests; specialized compound test cases are not shared and can therefore contain both specialized and common tests.

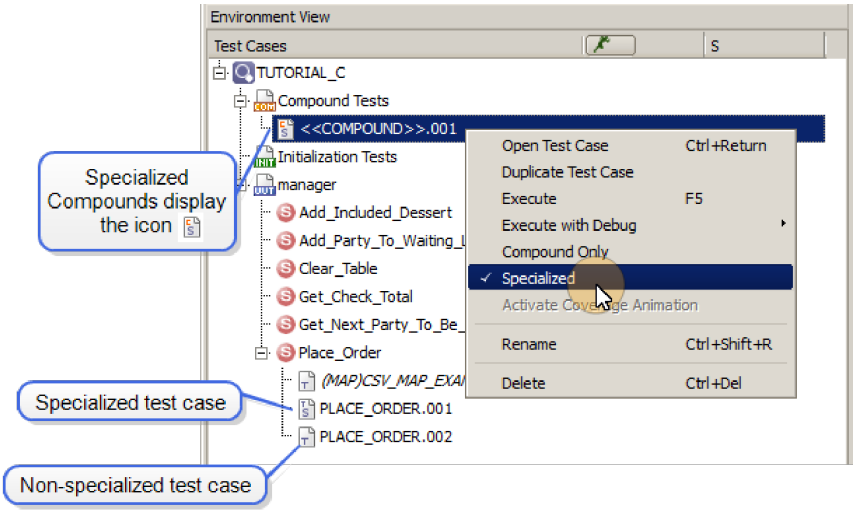
Attempting to add a specialized test case to a non-specialized compound test will generate the following error message:



An error message is also displayed when toggling the specialized status of a common test case that is a member of a common compound test.

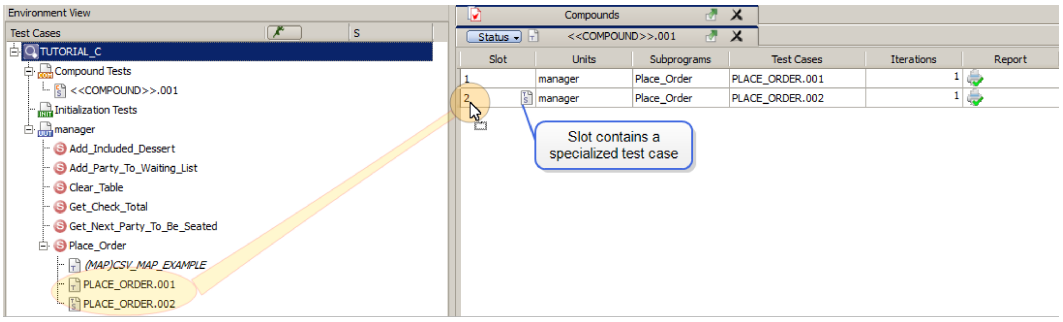


To add a specialized test case to a compound, you must create a specialized compound test by right-clicking on the compound test case node and selecting **Specialized** from the context menu.



The specialized compound test icon is displayed next to the compound test case name.

Add test cases to the Specialized compound test by dragging them from the Test Case Tree into the Compound Test Editor. Remember that specialized compound tests can contain both specialized and non-specialized test cases (but attempting to add a specialized test case to a non-specialized compound test will generate an error).



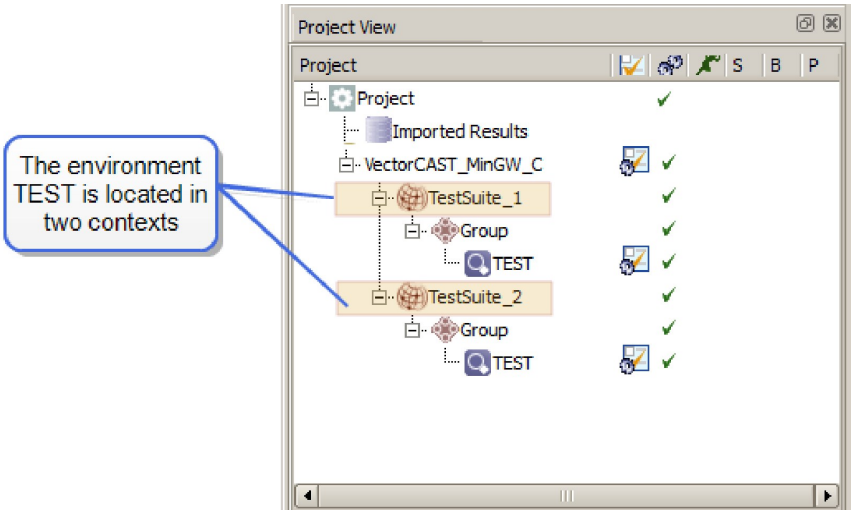
Creating Specialized Environment Configurations

The Specialized Environment Configuration allows configurations to be set for an environment in a single context. Users can modify the configuration options of a single environment without affecting other environments.

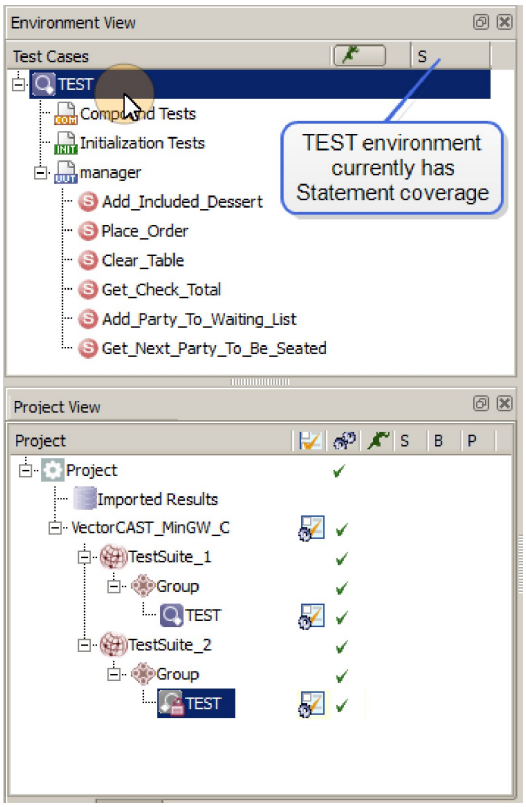
To Modify Configuration Options for a Single Environment

In the following example, the environment TEST appears in two contexts: TestSuite_1 and TestSuite_2.

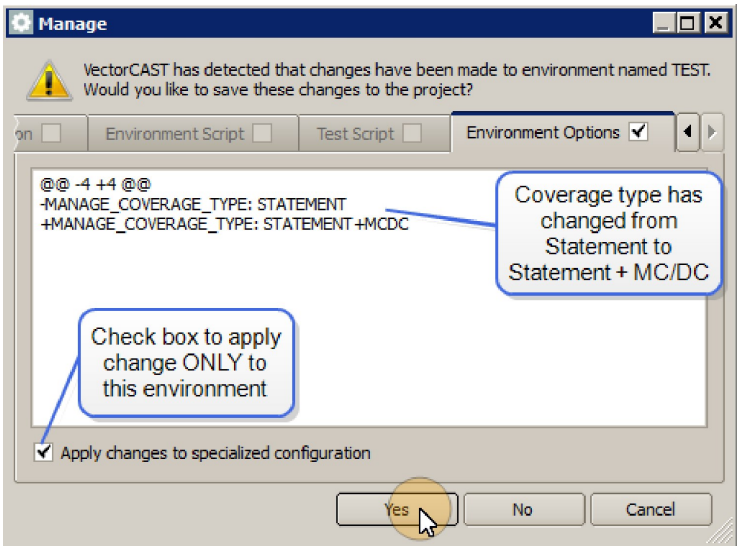
We are going to change the coverage type only for the environment in TestSuite_2.



1. Open the TEST environment for TestSuite_2.



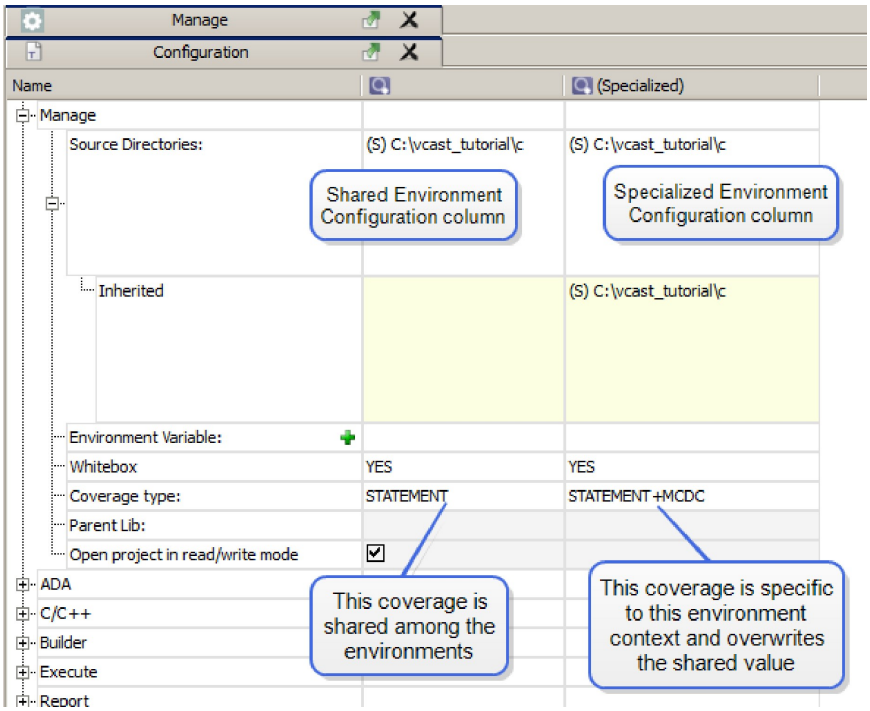
2. Select **Coverage => Initialize => Statement + MC/DC** from the Menu Bar to change the coverage type from 'Statement' to 'Statement + MC/DC'.
3. Close the environment.
4. Build the TEST environment for TestSuite_2. The Difference Checker displays a listing of the changes to the environment. Check the box **for Apply Changes to Specialized Configuration** and select the **Yes** button.



Note: To apply changes to all instances of the environment, leave the Apply Changes to Specialized Configuration box unchecked. The changes will then be shared by all environments.

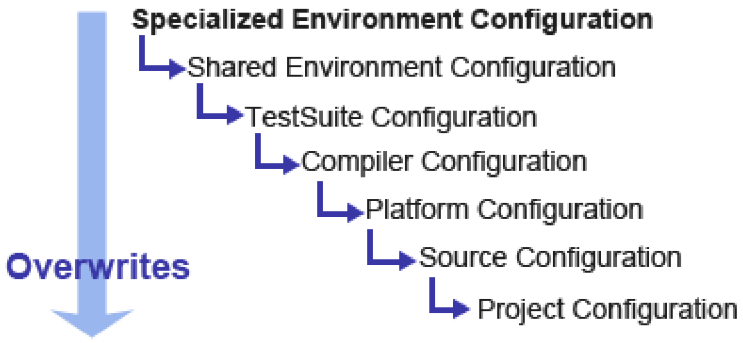
Viewing Specialized Configuration Options for an Environment

Specialized configuration options for an environment are viewable in the Configuration Editor. A Specialized Environment Configuration column titled (Specialized) is located to the right of the Shared Environment Configuration column.



Configuration values in the Specialized Environment column overwrite the values in the Shared column.

The .CFG file for an environment is generated by combining configuration options in the following order, with the options at the Specialized Environment level overwriting all other levels:



Copy/Paste Functionality and Specialized Environment Configurations

VectorCAST provides the ability to right-click on the Configuration node icon next to an environment and select **Copy** or **Paste** from the context menu.

It is important to note that any local configuration options set in the Specialized Environment will not be included in a Copy/Paste action. Only those options that are set at the Shared Environment level will be included in the Copy/Paste. Specialized configuration options are not transferable via the Copy/Paste context menu.

Using Test-Only Symbolic Constants

The Test-Only Symbolic Constants feature allows the user to supply test-only constants. This makes test cases more portable by replacing hard-coded scalar values with symbols that can be controlled independently from tests.

The benefits of this approach are:

- Improved test readability - replacing hard-coded values with symbolic names increases readability and understanding.
- Decreased test maintenance - values of symbolic names can be easily changed in one location. When a value is changed, all test cases are updated automatically.
- Better support for variant testing - symbolic names can be set to different values for different test contexts.

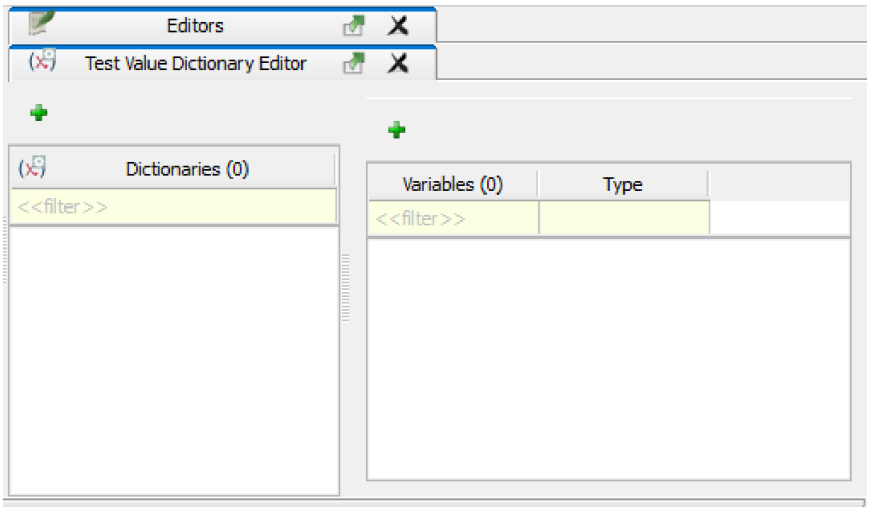
Test Value Dictionary

The Test Value Dictionary is an .xml file containing the variables for test values. Variables can be set to have either a a single value, a list of values, or a range of values. VectorCAST provides a Test Value Dictionary Editor to provide easy access for the creation and maintenance of Test Value Dictionaries and their variables.

Open the Test Value Dictionary Editor

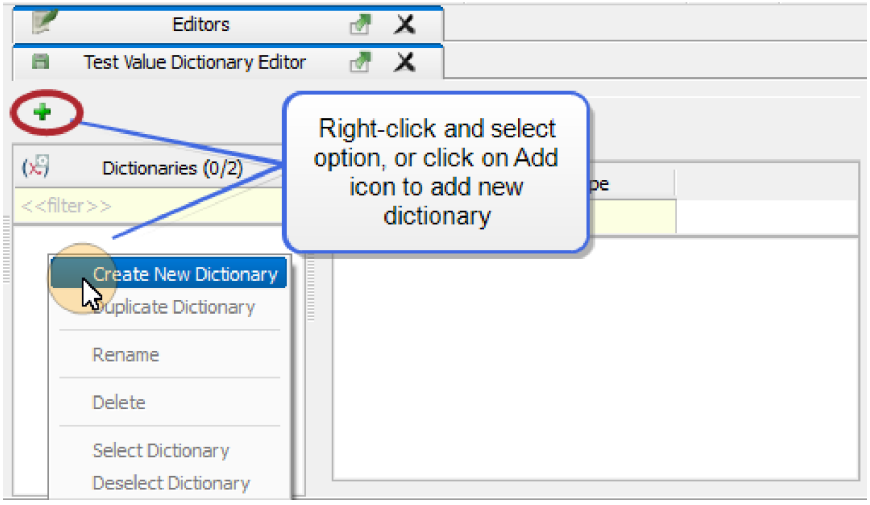
To open the Test Value Dictionary Editor, select the Edit Test Value Dictionary icon from the Toolbar. Alternatively, select **Project => Edit Test Value Dictionary** from the Menu Bar.

The Test Value Dictionary Editor opens.

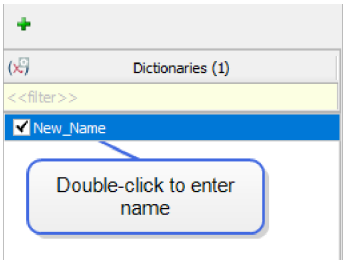


Create a New Test Value Dictionary

To create a new Test Value Dictionary, click on the plus icon above the Dictionaries pane. Alternatively, right-click within the Dictionaries pane and select **Create New Dictionary**.

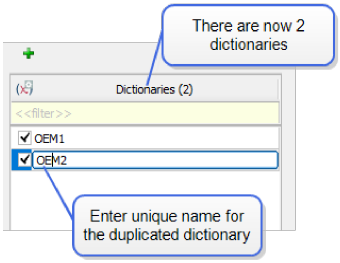


A new dictionary is added to the Editor. Double-click on the name field to open editing mode and enter a new name for the dictionary. Test Value Dictionary names must be unique. Duplicate Test Value Dictionaries are ignored during **Save**.



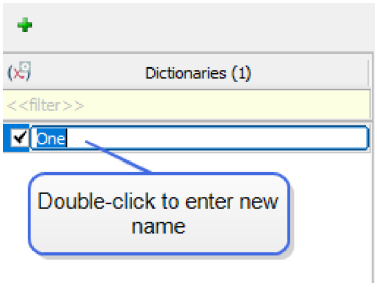
Duplicate a Test Value Dictionary

Once an initial dictionary has been created, additional dictionaries can be added using duplication.To duplicate an existing dictionary, right-click in the name field of the dictionary and select **Duplicate Dictionary** from the context menu. You must enter a unique name for the duplicated dictionary, otherwise the dictionary will be ignored during **Save**.



Rename a Test Value Dictionary

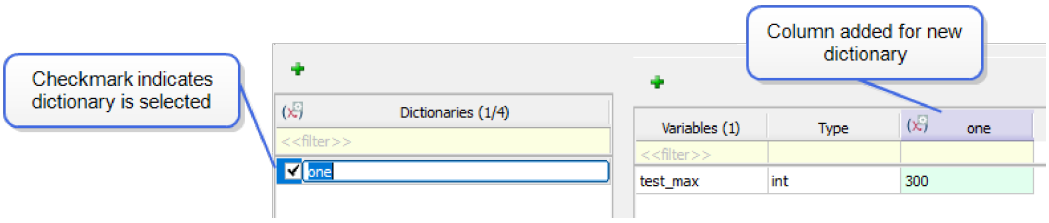
A dictionary can renamed by right-clicking on the directory name field and selecting **Rename** from the context menu. Enter the new name and select **Enter**.



Select / Deselect a Dictionary

By default, the selection check box next to the dictionary is checked when a new dictionary is created. The dictionary selection can be toggled on and off by either using the check box next to the dictionary name, or by highlighting the dictionary, right-clicking, and selecting the appropriate **Select Dictionary** or **Deselect Dictionary** option from the context menu. Selecting a dictionary will add a

column for that dictionary in the Variables pane.



Delete a Dictionary

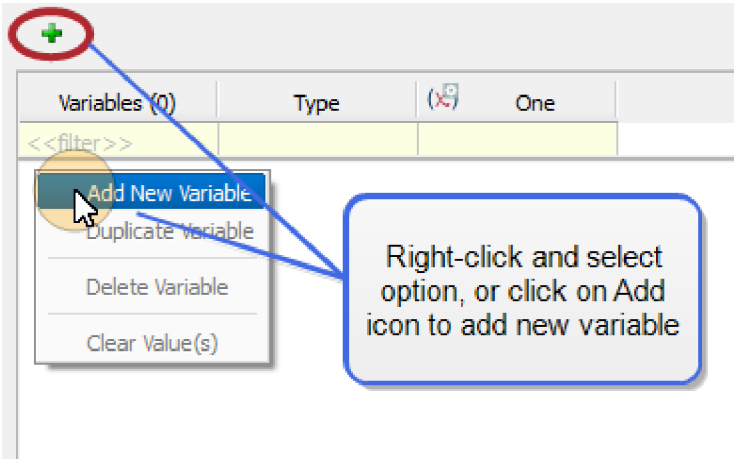
A dictionary may be deleted by highlighting the dictionary, right-clicking and selecting **Delete** from the context menu. A confirmation dialog will open and you will be prompted to confirm the deletion. Select **Yes** to permanently remove the dictionary. Select **No** to abort the deletion and return to the Editor.

Add a Variable

Once a dictionary is named and selected, enter the variables contained in the dictionary in the Variables pane. Note that the Variables pane now contains a column for the selected dictionary.

Note: If a Test Value Dictionary is empty, the dictionary will not be saved when the editor is closed.

To add a variable, click on the plus icon ✚ above the Variables pane. Alternatively, right-click within the Variables pane and select **Add New Variable**.



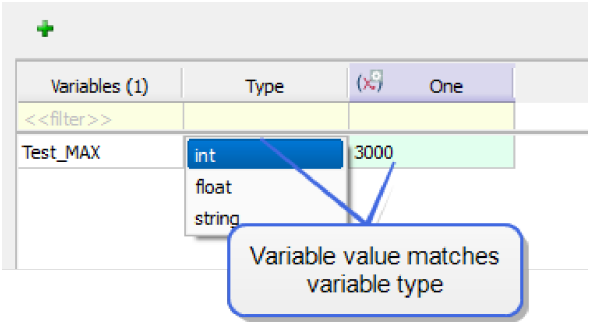
Enter a name for the variable by double-clicking on the field to enter editing mode. The variable name field cannot be empty. The following rules apply for constructing variable names:

- Underscore (_) is supported
- Uppercase letters (A-Z) are supported
- Lowercase letters (a-z) are supported

- Digits (0-9) are supported
- Spaces and commas are not allowed
- No special symbols (with the exception of the underscore mentioned above) are allowed.
- First character of the name should be a letter or underscore
- Variable name should not be a reserved word

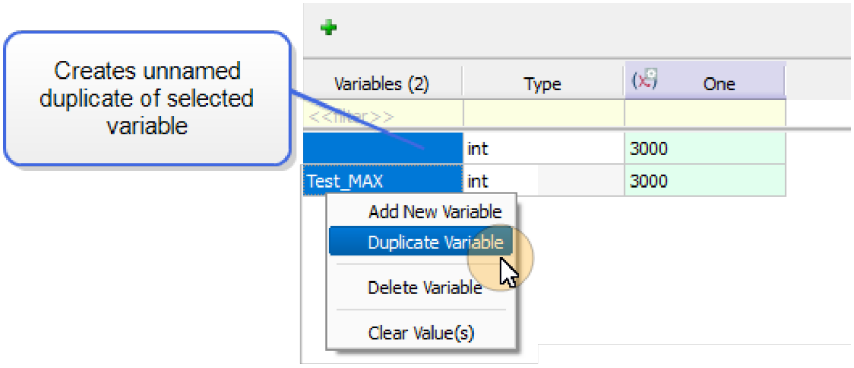
Enter the variable type. The default type is **`int`**. To select a different type, double-click on the variable field to open the drop-down menu and select the type. Alternatively, right-click on the type field and select **Edit** from the context menu. Supported types are **`int`**, **`float`** and **`string`**.

Enter a value for the variable by double-clicking in the dictionary column to enter editing mode. Alternatively, right-click on the value field and select **Edit** from the context menu.The value must match the variable type. Values that do not match the type are displayed with a red background. Values which match the type are displayed with a green background.



Duplicate a Variable

Once an initial variable has been added, additional variables can be added using duplication.To duplicate an existing variable, right-click in the name field of the variable and select **Duplicate Variable** from the context menu. This adds an unnamed variable containing the type and value of the selected variable to the dictionary.



Edit a Variable Value

To edit a variable, double-click on the value field and open Editing mode. Alternatively, right-click on the value field and select **Edit** from the context menu.

Clear Values for a Variable

To clear the values for a variable, right-click within the name field or the value field of the variable and select **Clear Value(s)** from the context menu. A confirmation dialog opens to confirm that you want to clear all selected entries. Select **Yes** to clear the selected entries. Select **No** to cancel the action and return to the Editor.

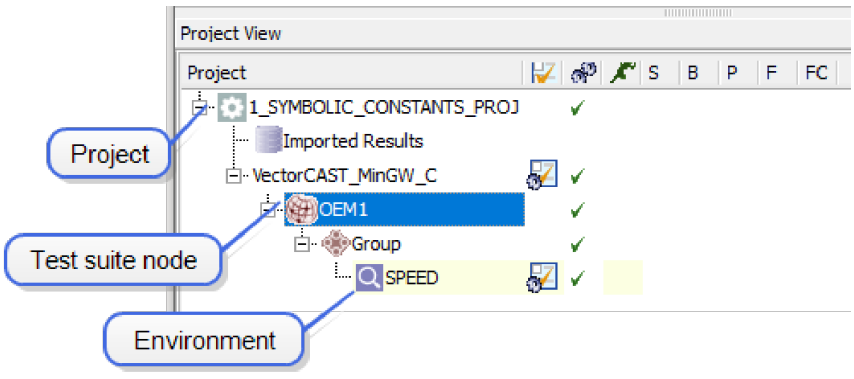
Delete a Variable

Delete an existing variable by right-clicking within the variable's line and selecting **Delete Variable** from the context menu. A confirmation dialog will open and you will be prompted to confirm the deletion. Select **Yes** to permanently the selected variable from all dictionaries. Select **No** to abort the deletion and return to the Editor.

Test-Only Symbolic Constants Example

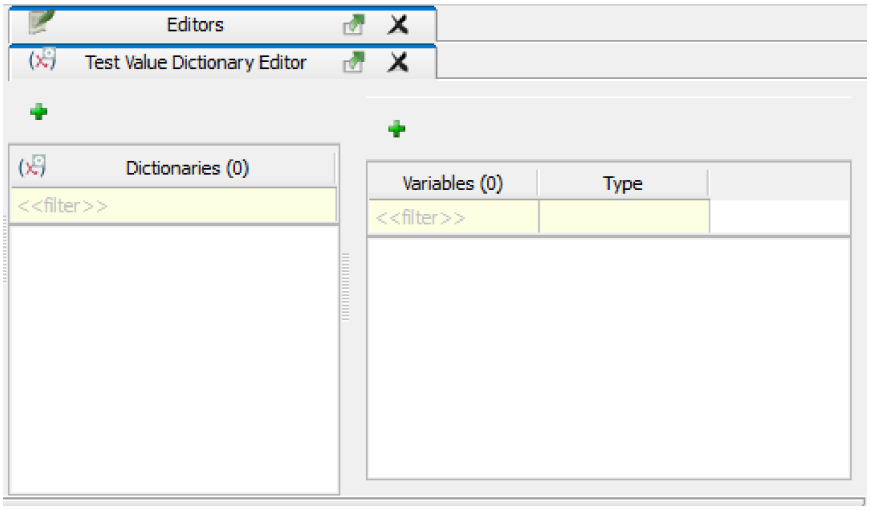
In the following example, we first create an empty VectorCAST project, **`1_SYMBOLIC_CONSTANTS_PROJ`** and set the compiler to VectorCAST_MinGW_C.

Next, rename the TestSuite node to **`OEM1`**. Under the **`OEM1`** test suite, create a unit test environment **`SPEED`**, using the example source code **`speed.c`**. This source code is included in the VectorCAST distribution and is located in **`%VECTORCAST_DIR%\Examples\symbolic_constants\speed.c`**.

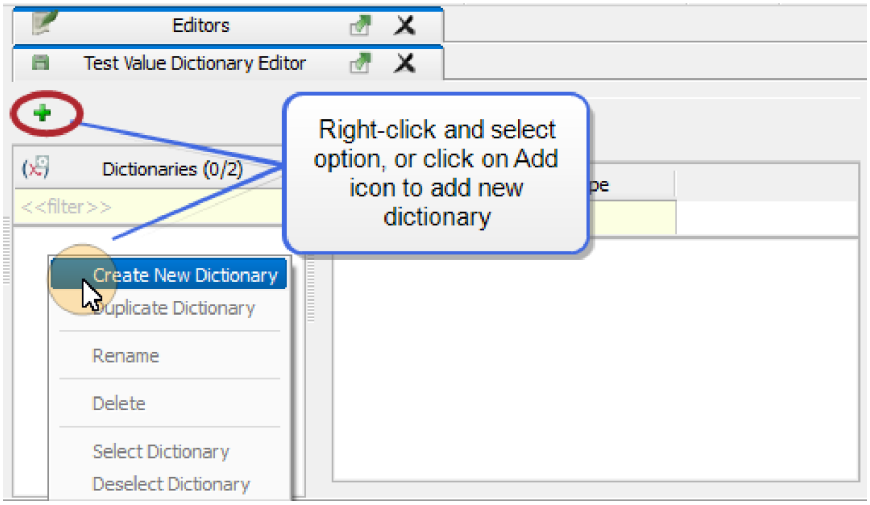


Close the **`SPEED`** environment if it has opened.

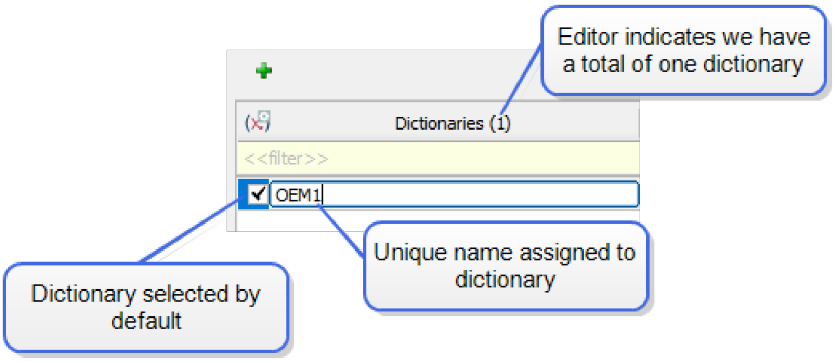
Next, open the Test Value Dictionary Editor in the VectorCAST project by selecting the Edit Test Value Dictionary icon from the Toolbar. Alternatively, select **Project => Edit Test Value Dictionary** from the Menu Bar. The Test Value Dictionary Editor opens.



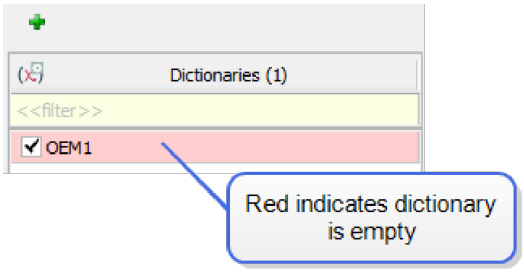
To create a new Test Value Dictionary, click on the plus icon ✚ above the Dictionaries pane. Alternatively, right-click within the Dictionaries pane and select **Create New Dictionary**.



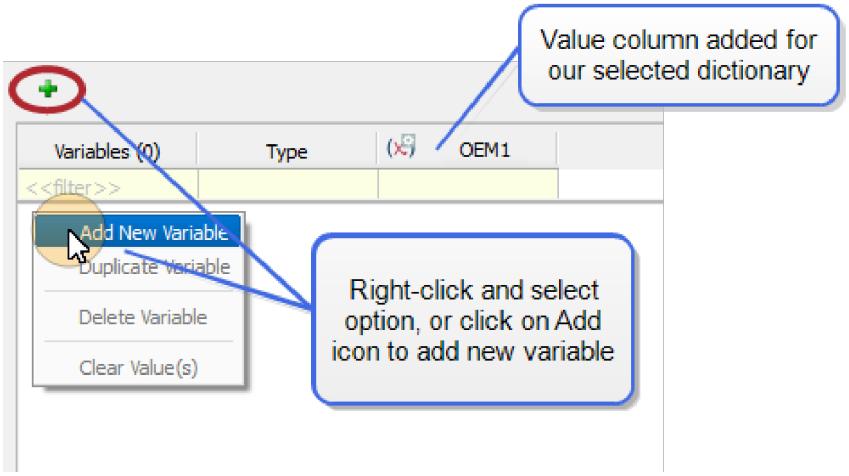
A new dictionary is added to the Editor. Double-click on the name field to open editing mode and enter a unique name for the new dictionary. For our example, we will name our new dictionary `OEM1`. Note that new dictionaries are selected by default when added.



Note that the dictionary background is displayed in red, to indicate that the dictionary is incomplete without variables and has not been saved.



Next, add variables to the dictionary. Note that there is now a value column added to the Variables pane for our selected dictionary. To add a variable, click on the plus icon ✚ above the Variables pane. Alternatively, right-click within the Variables pane and select **Add New Variable**.



For each variable, enter a unique name, type and value. Note that once an initial variable has been created, we can easily duplicate a second variable for our dictionary by right-clicking in the name field of the variable and selecting **Duplicate Variable** from the context menu. This adds an unnamed variable containing the same type and value of the selected variable to our dictionary. For more

details on how to enter variables see the "Add a Variable" section of "Using Test-Only Symbolic Constants" on page 94.

Enter the following variables for the **OEM1** dictionary:

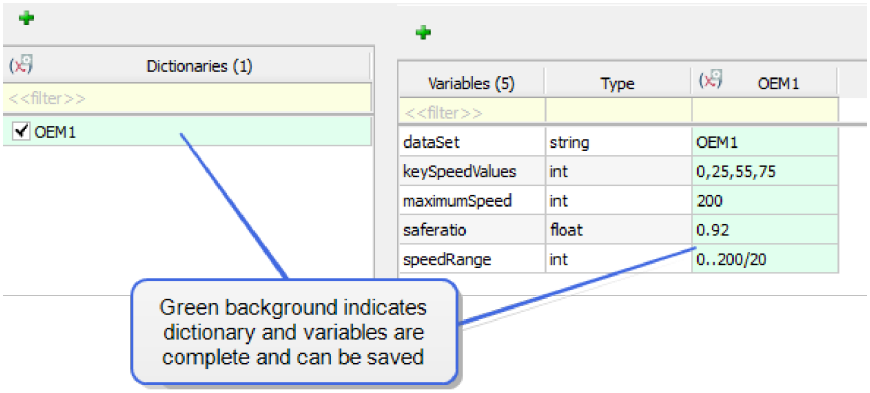
Name: dataSet
Type: string
Value: OEM1

Name: keySpeedValues
Type: int
Value: 0,25,55,75

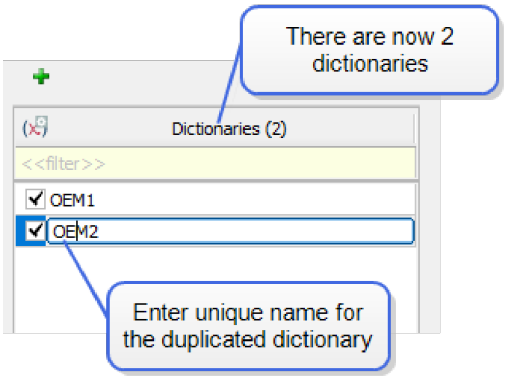
Name: maximumSpeed
Type: int
Value: 200

Name: speedRange
Type: int
Value: 0..200/20

Name: saferatio
Type: float
Value: 0.92



Now that an initial dictionary has been created, additional dictionaries can be added using duplication.To duplicate the **OEM1** dictionary, right-click in the name field of **OEM1** and select **Duplicate Dictionary** from the context menu. Enter a unique name for the duplicated dictionary, otherwise the dictionary will be ignored during **Save**. Enter the dictionary name **OEM2**



In the Variables pane, a column has been added for the new **OEM2** dictionary. Enter the following variables for the **OEM2** dictionary:

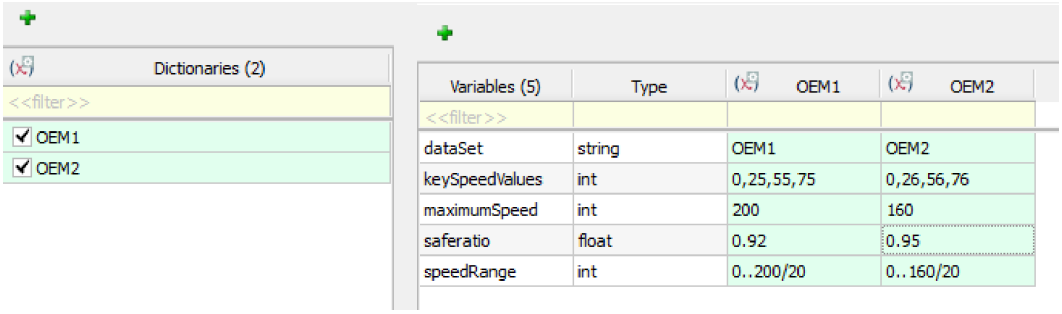
Name: dataSet
Type: string
Value: OEM2

Name: keySpeedValues
Type: int
Value: 0,26,56,76

Name: maximumSpeed
Type: int
Value: 160

Name: speedRange
Type: int
Value: 0..160/20

Name: saferatio
Type: float
Value: 0.95

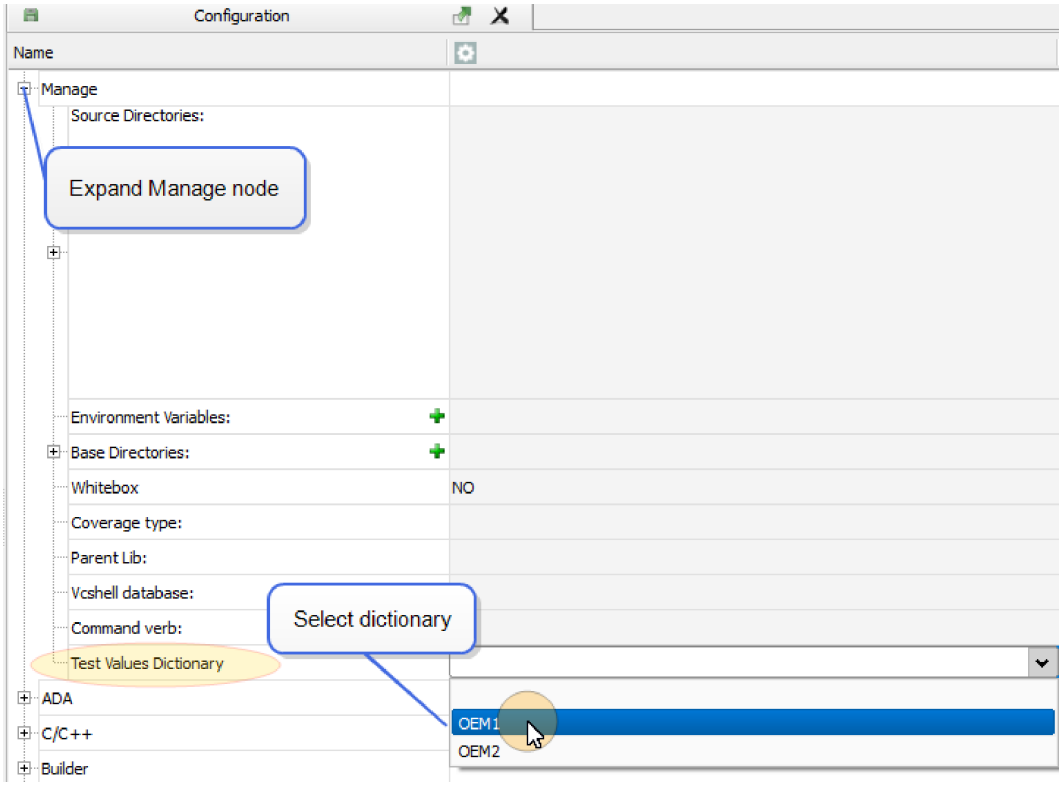


Dictionaries **OEM1** and **OEM2** are now complete, as indicated by the green background. Save the changes and close the Editor. Note that if the Test Value Dictionary Editor is closed without saving changes, you will be prompted to save your changes and continue.

Now add the **OEM1** dictionary to the **SPEED** environment by right-clicking on the **OEM1** Test Suite node

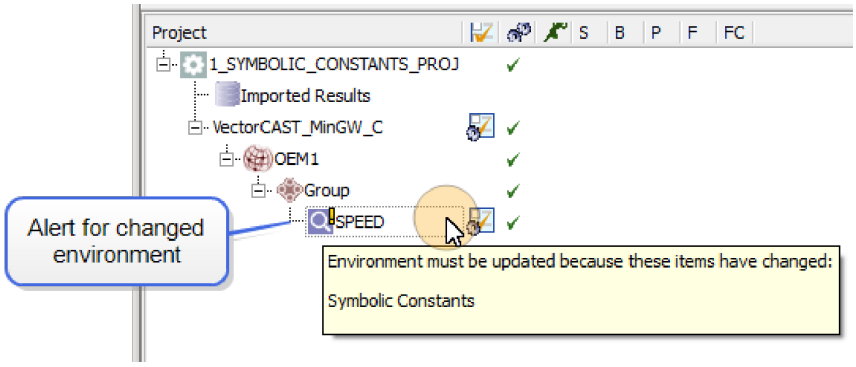
in the Project tree and selecting **Open Configuration** from the context menu. The Configuration Editor opens.

Expand the **Manage** node, and under the Test Values Dictionary field, use the drop-down menu to select the dictionary, **OEM1**, and add it to the configuration.



Save the changes and close the Configuration Editor.

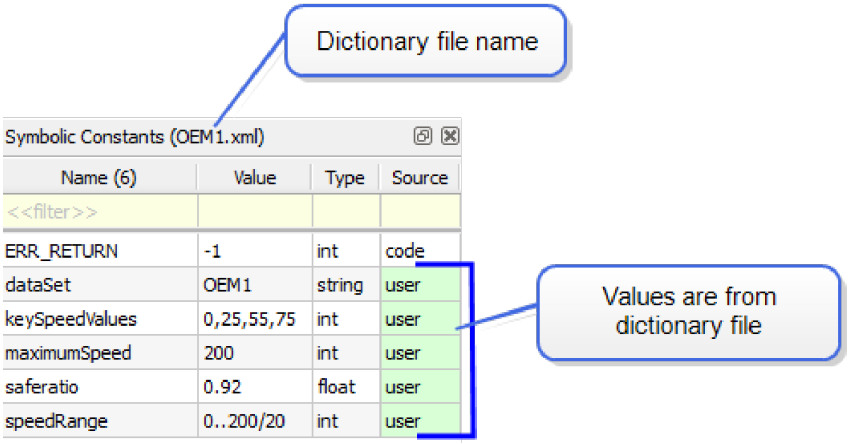
The environment node in the Project Tree now displays an alert indicating that there are changes to the environment and the environment must be updated.



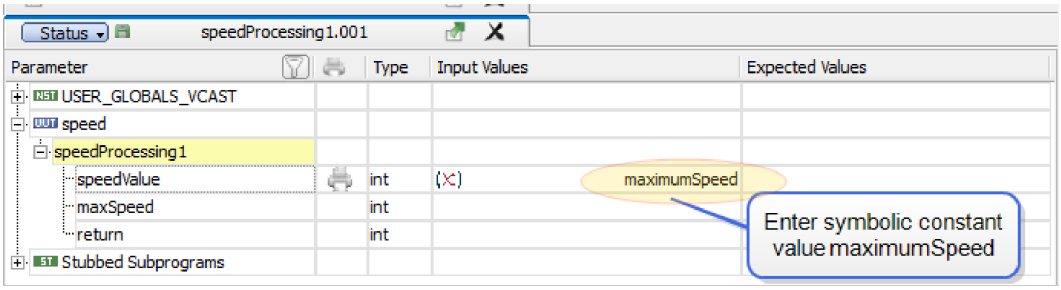
Perform a full build on the environment by right-clicking and selecting **Build => Full** from the context menu.

Open the **SPEED** environment and select the Symbolic Constants button

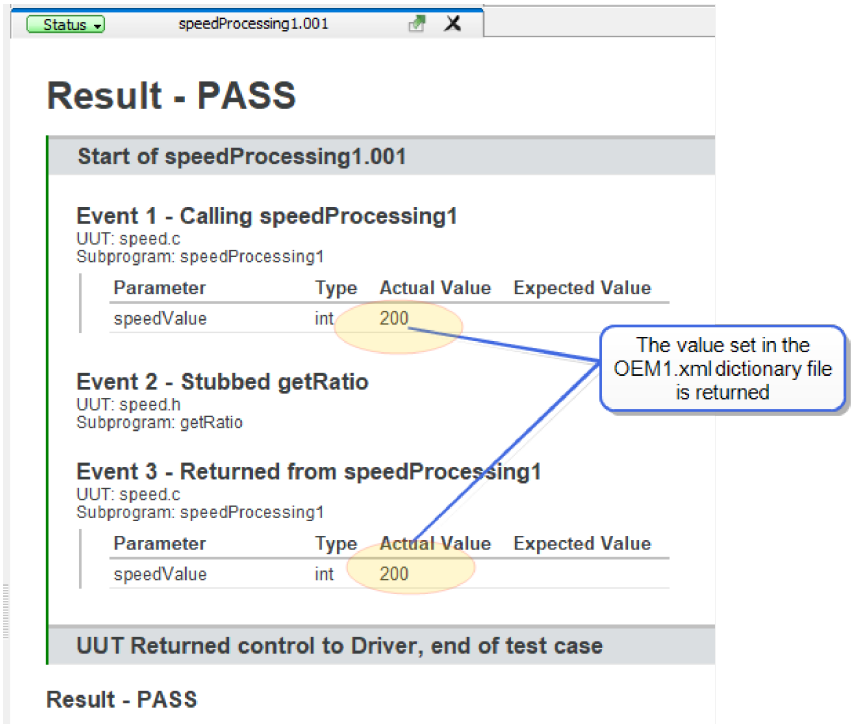
(x) on the Toolbar to view the values from the **OEM1** Test Value Dictionary file.



Next create a test case, **speedProcessing1.001**. In the Test Case Editor, enter an Input Value of **maximumSpeed** for the **speedValue** parameter. Note that in the case of the **OEM1** dictionary, the value will be 200.

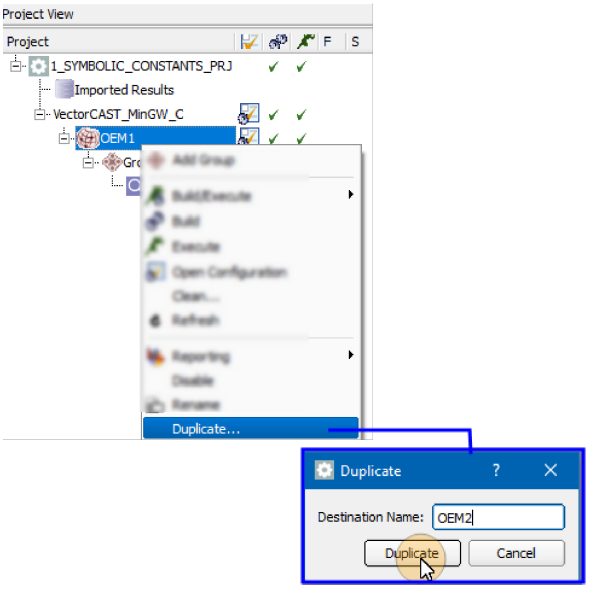


Save the changes and execute the test case **speedProcessing1.001**. Note that the actual value returned for the **speedValue** parameter is 200, the value set in the **OEM1**dictionary file.



Close the **SPEED** environment and save the changes to the project.

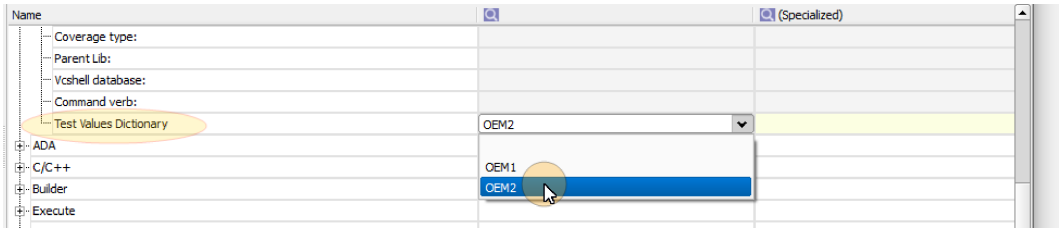
Now duplicate the **OEM1** Test Suite node and rename it **OEM2** by right-clicking and selecting **Duplicate...** from the context menu.



Perform a full build on the new environment.

Add the **OEM2** dictionary to the **SPEED** environment in the **OEM2** Test Suite by right-clicking on the **OEM2** Test Suite node in the Project tree and selecting **Open Configuration** from the context menu.

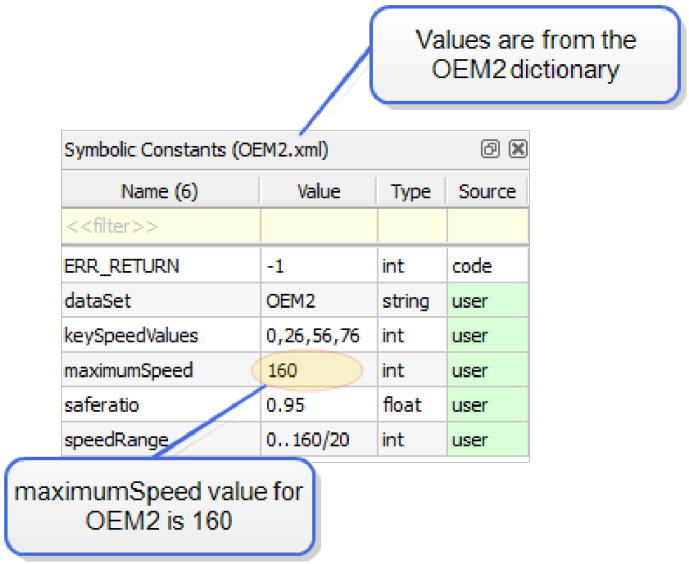
Expand the **Manage** node, and under the Test Values Dictionary field, use the drop-down menu to select the **OEM2** dictionary to add it to the configuration.



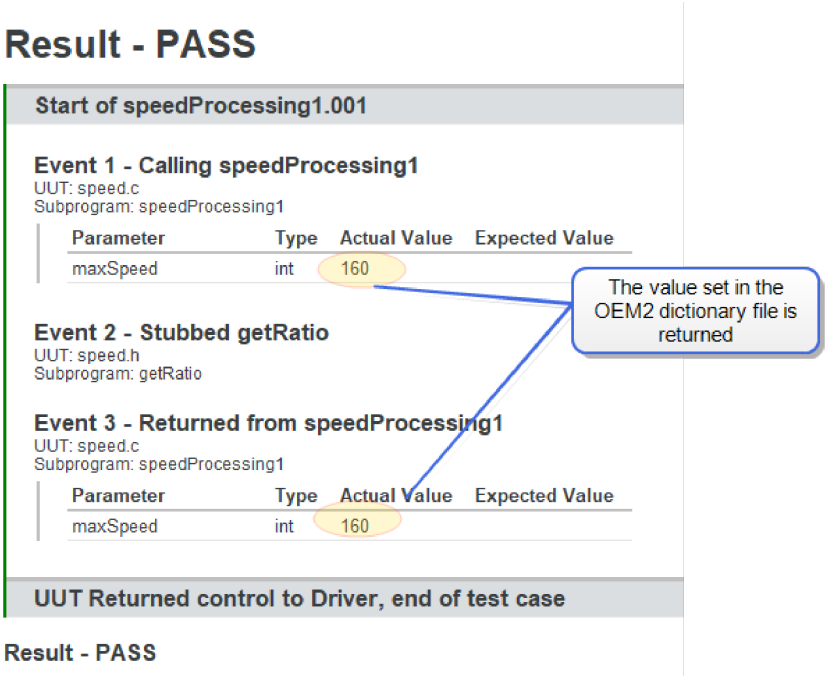
Save the changes and close the Configuration Editor.

The environment node in the Project Tree now displays an alert indicating that there are changes to the environment and the environment must be updated. Perform a full build on the **SPEED** environment.

Open the **SPEED** environment for the **OEM2** Test Suite. Open the Test Case Editor for the **speedProcessing1.001** test case. Note that the Symbolic Constants window shows the values from the **OEM2** dictionary loaded.



Note that when we execute the **speedProcessing1.001** test case in the **OEM2** Test Suite the actual value returned for the **speedValue** parameter is 160, the value set in the **OEM2** dictionary file.



As a result of using Test-Only symbolic constants stored in a portable .xml file, the user can replace hard-coded scalar values with symbols that are controlled independently of the tests.

Show Project Coverage for the Current Unit in the Coverage Viewer

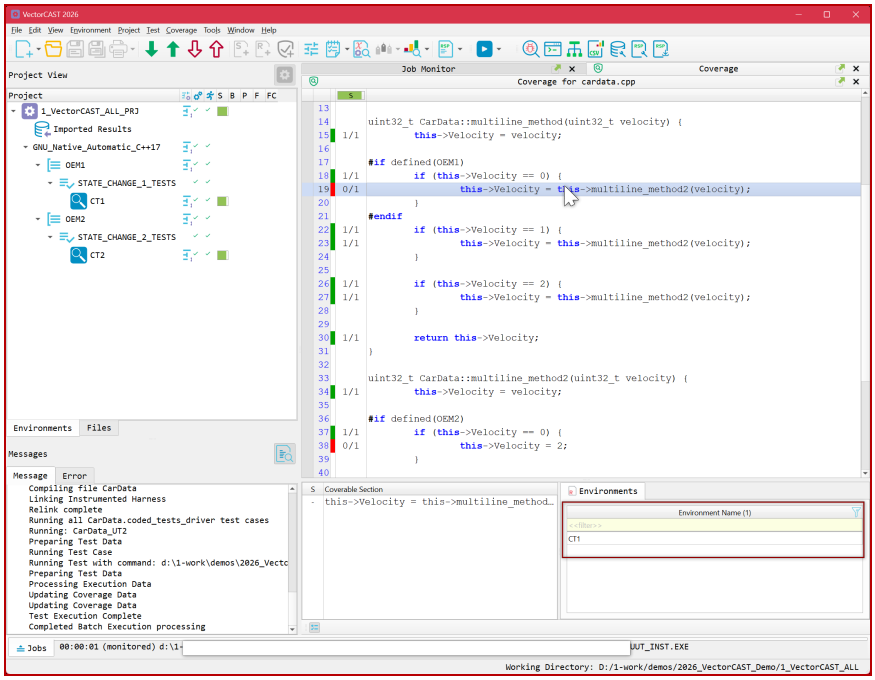
This feature can help when trying to add more tests to improve code coverage. Normally, the coverage shown for a source file comes only from the current environment. When you toggle the "Show Project Coverage" button to indicate "Show Environment+Project coverage," you can see coverage from other environments for this source file, allowing you to focus on code that is not covered by any other environment. This feature is only available in Source File Perspective (SFP).

Basic Use Case

For this use case, the following source code is used.


```
1  #include "CarData.h"
2  #include <stdint.h>
3
4  // Implementation of the constructor
5
6  CarData::CarData(uint32_t Velocity)
7      : Velocity(Velocity) {
8  }
9
10 // Implementation of the destructor
11 CarData::~CarData() {
12 }
13
14 uint32_t CarData::multiline_method(uint32_t velocity) {
15         this->Velocity = velocity;
16
17 #if defined(OEM1)
18         if (this->Velocity == 0) {
19                 this->Velocity = this->multiline_method2(velocity);
20         }
21 #endif
22         if (this->Velocity == 1) {
23                 this->Velocity = this->multiline_method2(velocity);
24         }
25
26         if (this->Velocity == 2) {
27                 this->Velocity = this->multiline_method2(velocity);
28         }
29
30         return this->Velocity;
31 }
32
33 uint32_t CarData::multiline_method2(uint32_t velocity) {
34         this->Velocity = velocity;
35
36 #if defined(OEM2)
37         if (this->Velocity == 0) {
38                 this->Velocity = 2;
39         }
40
41 #endif
42         if (this->Velocity == 1) {
43                 this->Velocity = 4;
44         }
45
46         if (this->Velocity == 2) {
47                 this->Velocity = 6;
48         }
49
50         return this->Velocity;
51 }
```

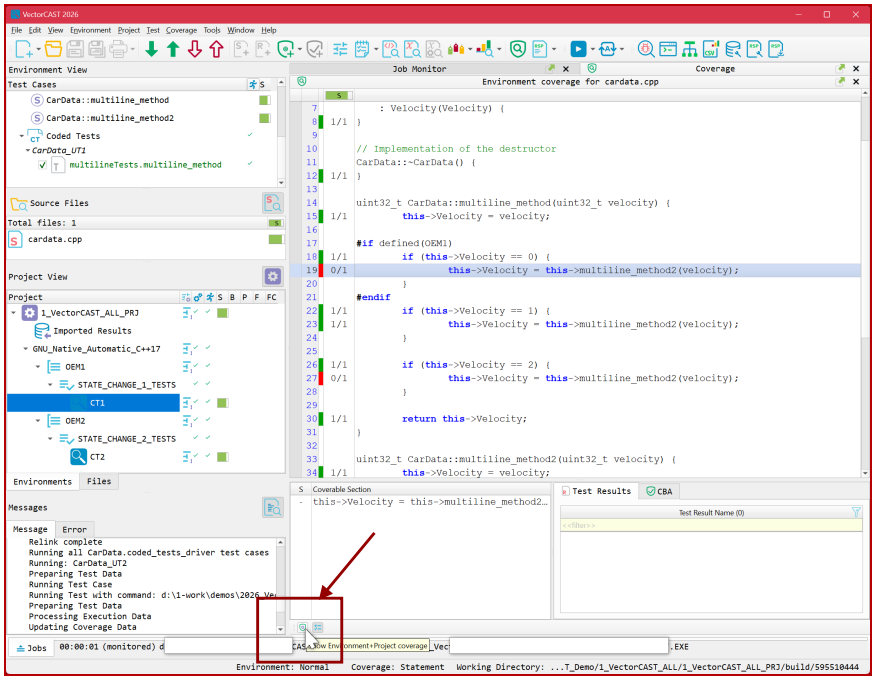
Here, two C++ Unit Test Environments have been created. Both test environments are configured for Statement coverage, and the code coverage results are shown below.



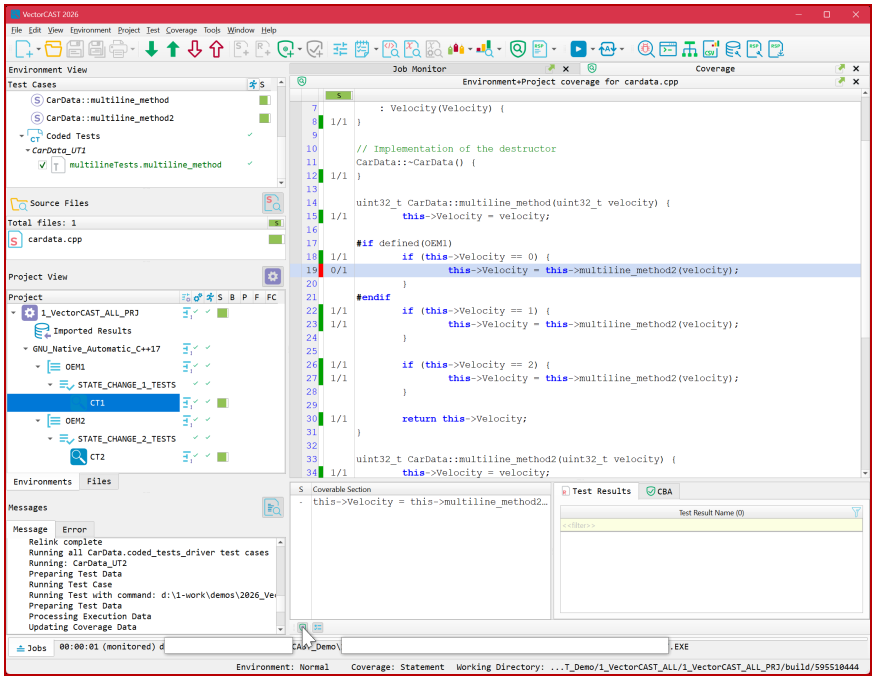
The Coverage Viewer shows two lines that are not covered, line 19 and line 38. Click on line 19, and VectorCAST shows the environment where that line is coverable. Here it is Environment CT1.

Open environment CT1 by double-clicking (or right-click and choose **Open Environment**) on CT1 in the Environment Name Panel (highlighted in the red box). This will open the environment on the selected line. The coverage in this view now shows an additional uncovered line, line 27. Normally the coverage shown is only the coverage from the open environment. Line 27 is not covered by any test in Environment CT1. To focus on lines not covered by any environment, the Coverage Viewer can display compatible coverage from all environments in the project.

To show coverage from all environments in the project, toggle the "Show Environment+Project coverage" button. The button is highlighted in the image below.



After Selection ...



The Coverage Viewer now displays coverage from the Environment as well as coverage from other Environments in the Project. Line 27 now shows as covered as the coverage from Environment CT2 is

now also shown.

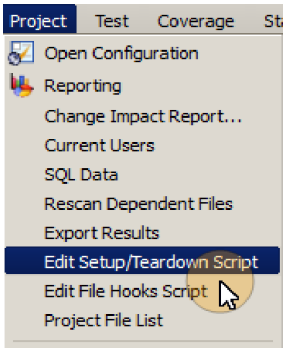
Target Setup and Teardown

Prior to running tests, there are often preparatory tasks (*setup*) to perform, such as starting a target server, setting environment variables, and starting simulators. There may be cleanup tasks (*teardown*) such as killing processes and cleaning up files that also must be completed.

Setup/Teardown functionality, using the customizable python script **`setup_teardown.py`**, allows the user to automate these compiler/target setup and teardown tasks from within the VectorCAST project. The script is run when a VectorCAST project is loaded and is used to map the compiler node names within a project to Python classes that contain setup and teardown members.

Whenever a new environment is created in an existing VectorCAST project which already has the setup and teardown members defined in the **`setup_teardown.py`** script, the script is automatically triggered, making it easier to add an existing setup from within the VectorCAST project.

To edit the **`setup_teardown.py`** script (which is located in the **`<PROJECT_DIRECTORY>/python`** directory), from the Menu Bar, select **Project => Edit Setup/Teardown Script.**



The **`setup_teardown.py`** script opens in a text editor.

To use the Setup/Teardown feature:

1. Create a class that implements the "setup" and "teardown" methods of the Target class.
2. Add the Target subclass along with the VectorCAST Project's Compiler Node Name as an entry to the compiler_name_to_target_mapping dictionary.
3. Save changes.

When the file is saved, VectorCAST automatically reloads the setup_teardown module.

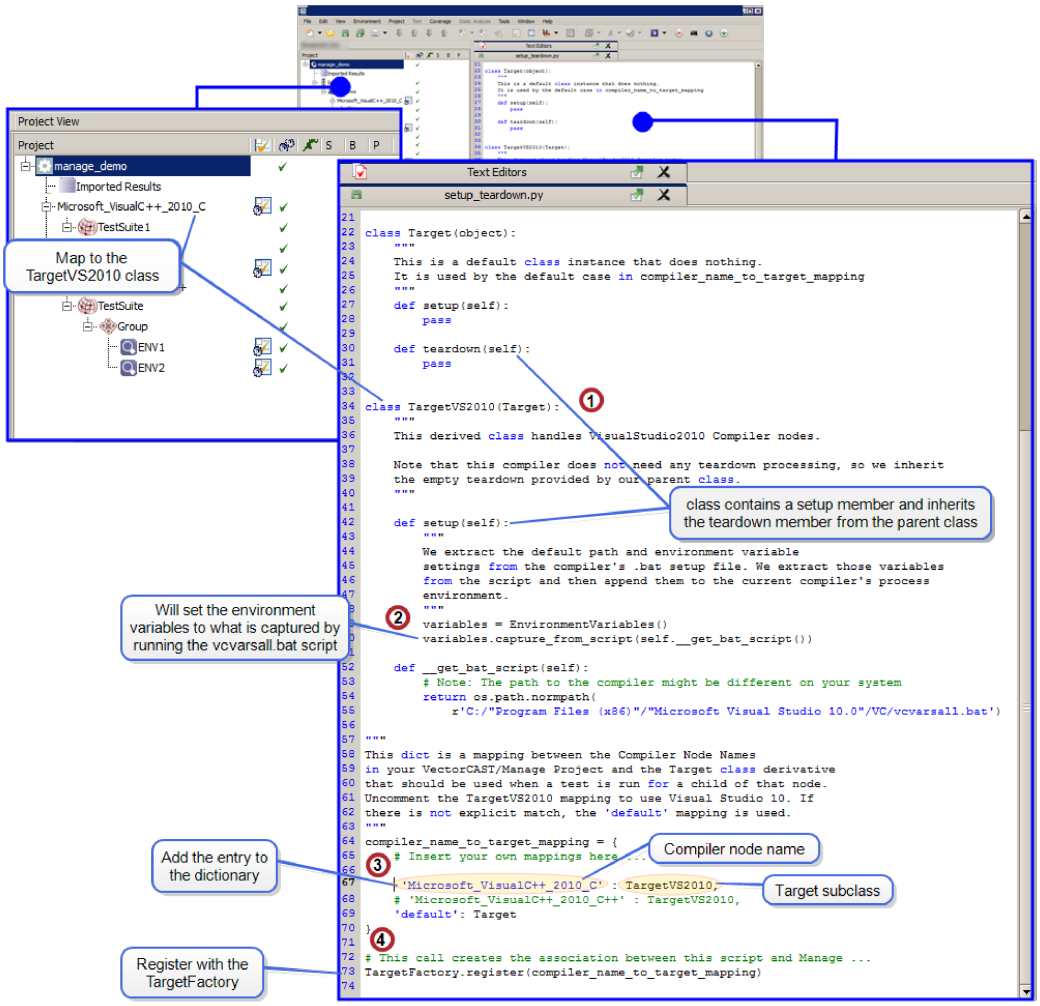
Note: If the **`setup_teardown.py`** script fails to reload (for example, due to a syntax error or runtime error), error messages are displayed in the Message window. Expand the Message window to view the Message and Error tabs.

Note: If you have multiple test environments under a specific compiler node, the setup and teardown will only be invoked once for that group of tests.

In our example below, the environment under the Microsoft_VisualC++_2010_C compiler node

needs certain environment variables set before it can be built. Previously, this was accomplished by running Visual Studio's **`vcvarsall.bat`** script which sets the environment variables in the shell that invoked it. An instance of VectorCAST is then spawned from that shell, inheriting the shell's environment variables. The environments under the Microsoft_VisualC++_2010_C compiler can then be built.

Setup/Teardown functionality allows the user to do the compiler setup directly within the VectorCAST project using the **`setup_teardown.py`** script.



In our example:

1. The user first modifies the **`setup_teardown.py`** script to include a **`TargetVS2010`** class for the Microsoft_VisualC++_2010_C compiler.
2. Note that the EnvironmentVariables **`variables.capture_from_script`** call in the setup will set the compiler node's environment variables to whatever process environment is captured by running the **`vcvarsall.bat`** script.
3. The user then adds the **`'Microsoft_VisualC++_2010_C': TargetVS2010`** entry to the

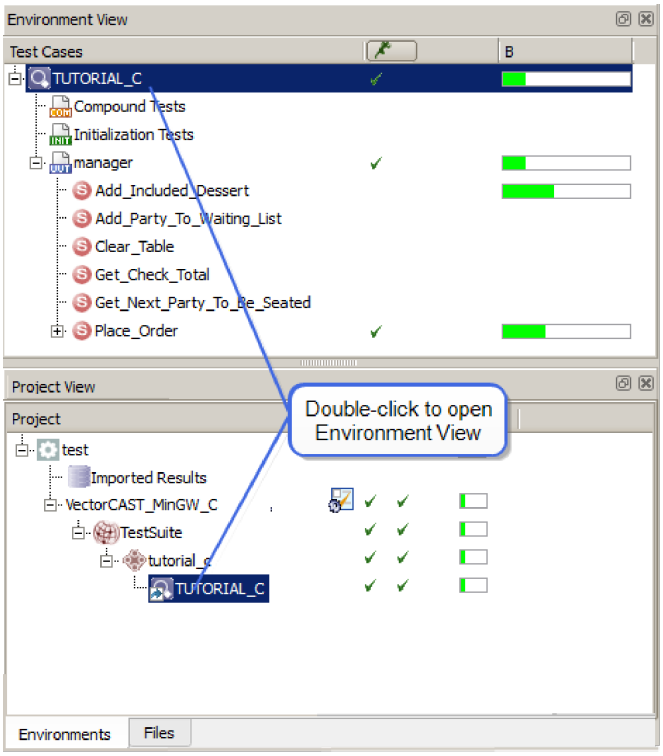
`compiler_name_to_target_mapping` dictionary.

4. The dictionary registers with the `TargetFactory` on line 73.

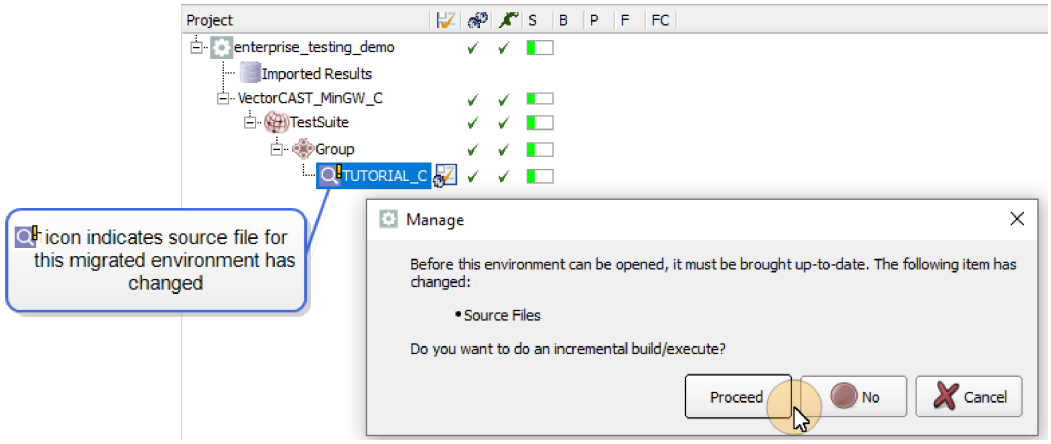
The setup task associated with the compiler has now been automated from within the VectorCAST project.

Opening a Test Environment

Double-clicking a previously-built environment on the Project Tree opens that environment.



Upon opening a migrated environment, if VectorCAST detects a dependency change (source code change, test script change, environment script change, etc.,) a dialog appears:



The user can select one of three options from the dialog:

- **Proceed** - Syncs the environment with its dependencies before opening the environment.
- **No** - Declines syncing the environment with its dependencies. The environment is opened and any changes to the dependencies are not evident. Note that if you do not rebuild the environment while it is open, when the environment is closed if there are changes to test scripts and environment scripts the Diff dialog opens, and if there are changes to the source, the Project tree will continue to display the source change icon.
- **Cancel** - Cancels the action and closes the dialog.

Automating Test Script Maintenance

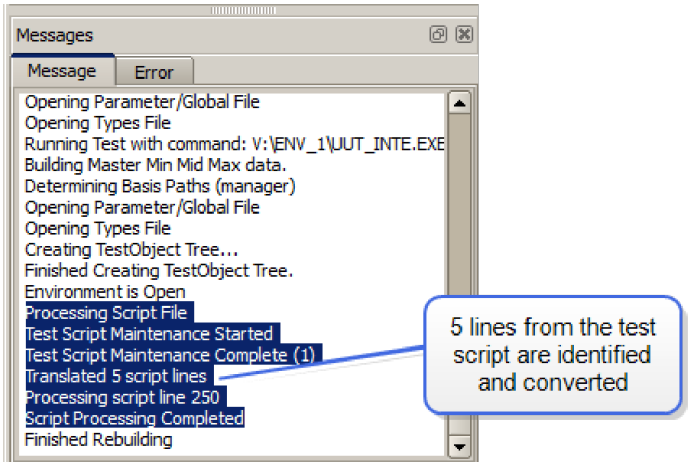
VectorCAST provides a Test Script Conversion utility that is automatically invoked when rebuilding an environment to maintain existing test scripts across source code changes. At the beginning of environment rebuild, the utility makes a copy of the original test script in the environment directory, naming 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.

Using the Test Script Converter

In a VectorCAST project, in instances where the environment has already been built, the Test Script Converter is invoked automatically during environment rebuild. The following example shows the Test Script Converter doing an automatic conversion where only one field differs and there is no ambiguity concerning which conversion to choose.

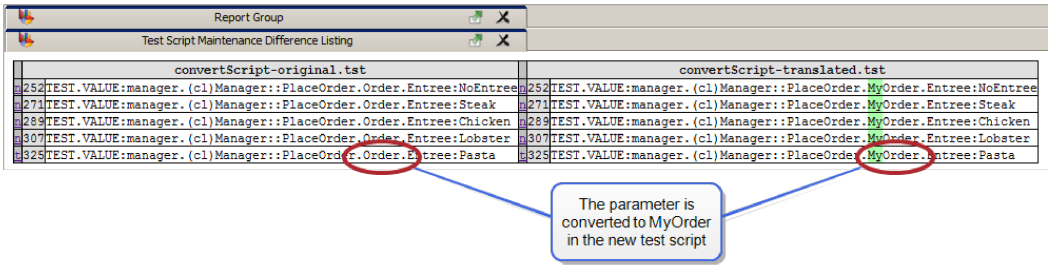
To automatically convert test scripts, you must first have an existing environment that has been previously built and which has test cases. Next, upload a new version of the source code and select **Environment => Rebuild Environment** from the Menu Bar.

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 five 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`**.

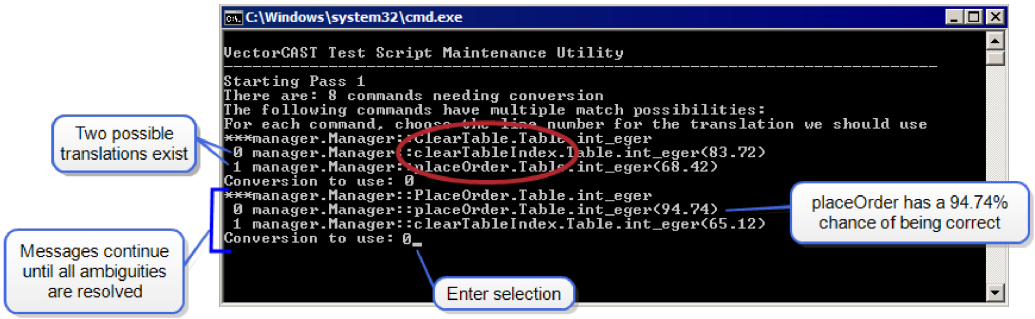
Using the Test Script Converter With Multiple Solutions

If the comparison of the old and new files reveals multiple solutions to a conversion, the converter prompts the user to choose the translation. For example, if the comparison reveals the following lines:

```
OLD: manager.Manager::ClearTable.Table.int_eger

NEW: manage.Manager::clearTableIndex.Table.int_eger

NEW: manage.Manager::placeOrder.Table.int_eger
```

The Test Script Converter recognizes that there are two possible translations, **`clearTableIndex.Table.int_eger`** or **`placeOrder.Table.int_eger`**, but does not know which one to use.

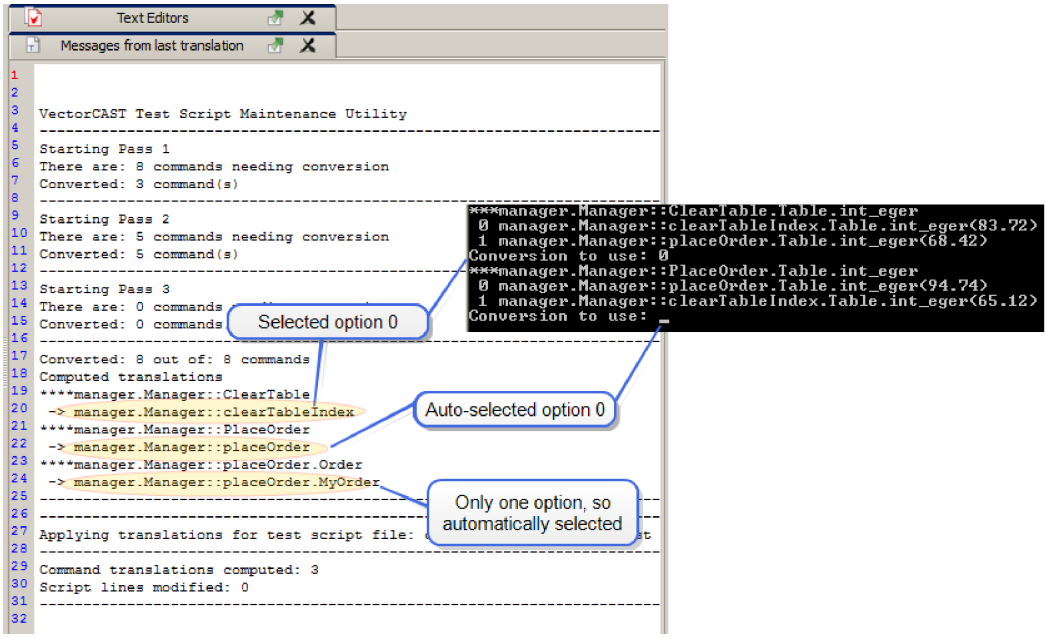
When using the Test Script Converter in the GUI, a prompt is displayed allowing the user to resolve the ambiguity and select the translation. Each choice is displayed along with a score indicating the likelihood of the choice being correct. The list is sorted with the highest score first.



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

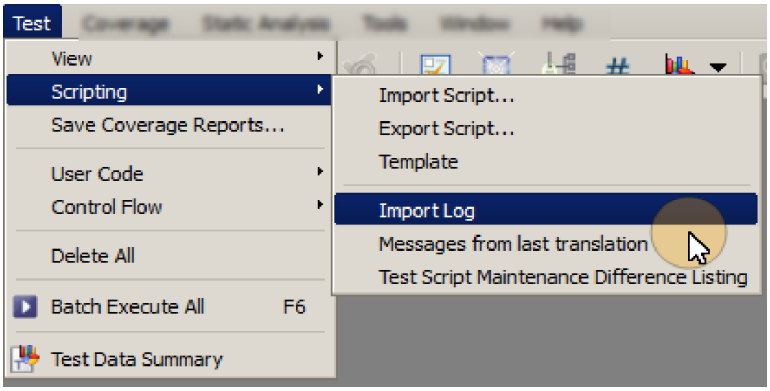
To delete a test script translation and revert to the original test script, perform the following steps:

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

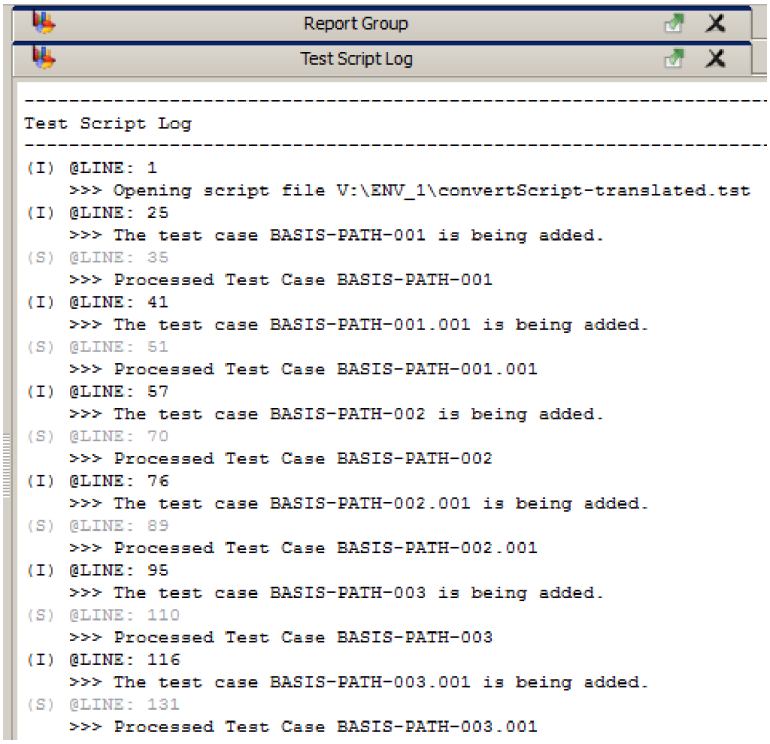
Data on the test script conversion is available from the Menu Bar by selecting **Test => Scripting** and selecting one of the report types:

- **Import Log**
- **Messages from last translation**
- **Test Script Maintenance Difference Listing**

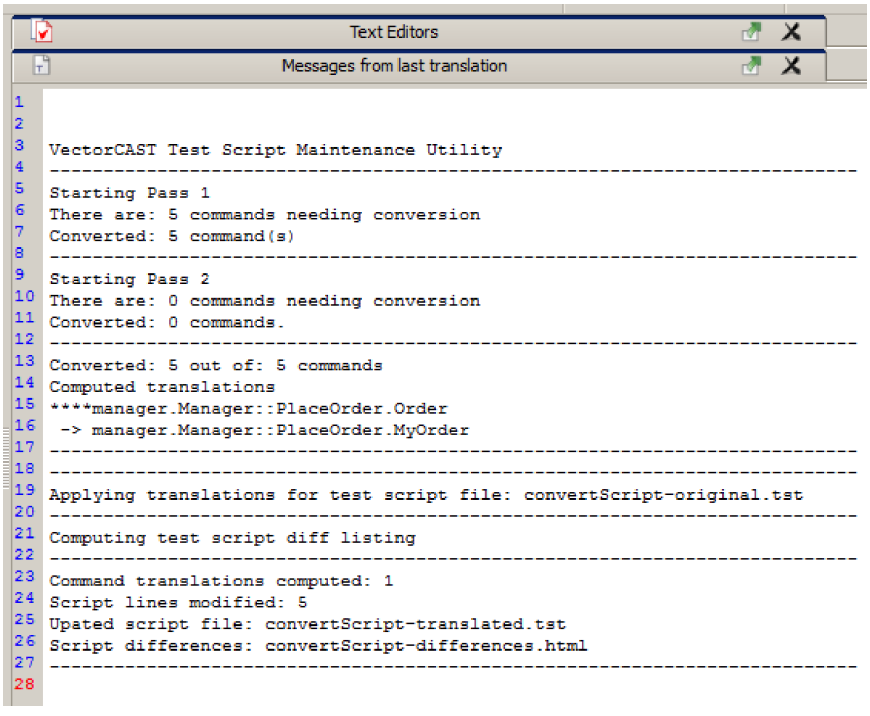


The reports are described below.

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.

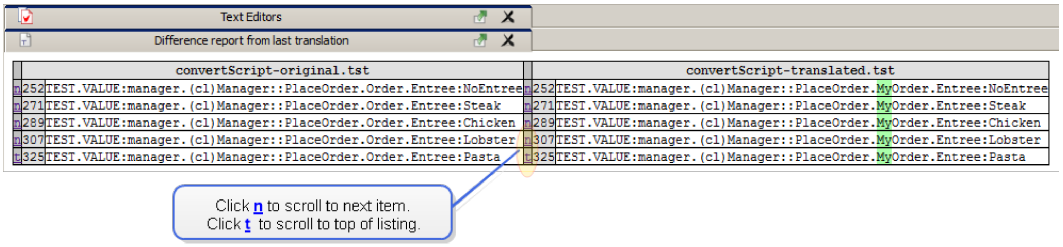


Selecting **Messages from last translation** opens the Messages From Last Translation report. This report provides status messages for the Test Script Converter utility.



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

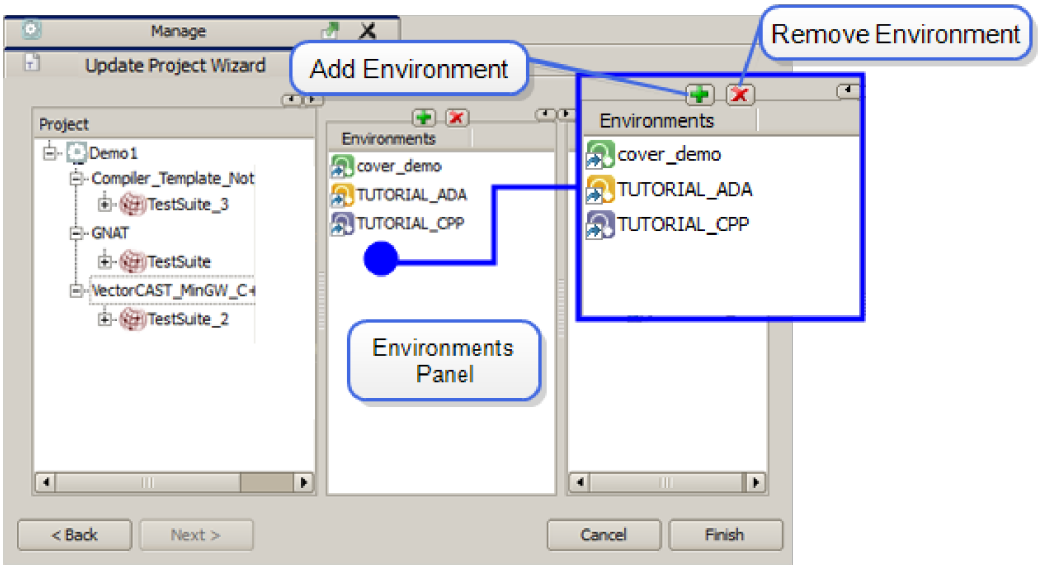
Test script conversion requires that a backup environment directory (.BAK) exists in order to begin processing. The Test Script Converter first reads the parse data in the **`param.xml`** and **`types.xml`** files in the .BAK version of the environment, and compares it to the newly-rebuilt environment 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 Converter can be found at **Test => Scripting => Messages** from last translation, or in the file **`convertScript-log.txt`**.

Using the Project Update Editor

Access the Project Editor by selecting **Project => Project Editor...** from the Menu bar or from the New Project Wizard. You can make modifications to any item contained in the panels of the Project Editor by right-clicking and then selecting an operation from the context menu. You can add additional items, delete, or rename an item as well as create environment groups and add them to the Project Tree.

The Environments Panel

The center panel is named the Environments Panel. It alphabetically lists the names of all environments found in the Search. Ada host and target environments are colored orange and C and C++ environments are colored blue. Cover environments are colored green.



Newly added environments are displayed in the Environments Panel list. The user must then include the new environment by dragging and dropping into the Environment Groups and Test Suites as desired.

To Add an Environment Manually

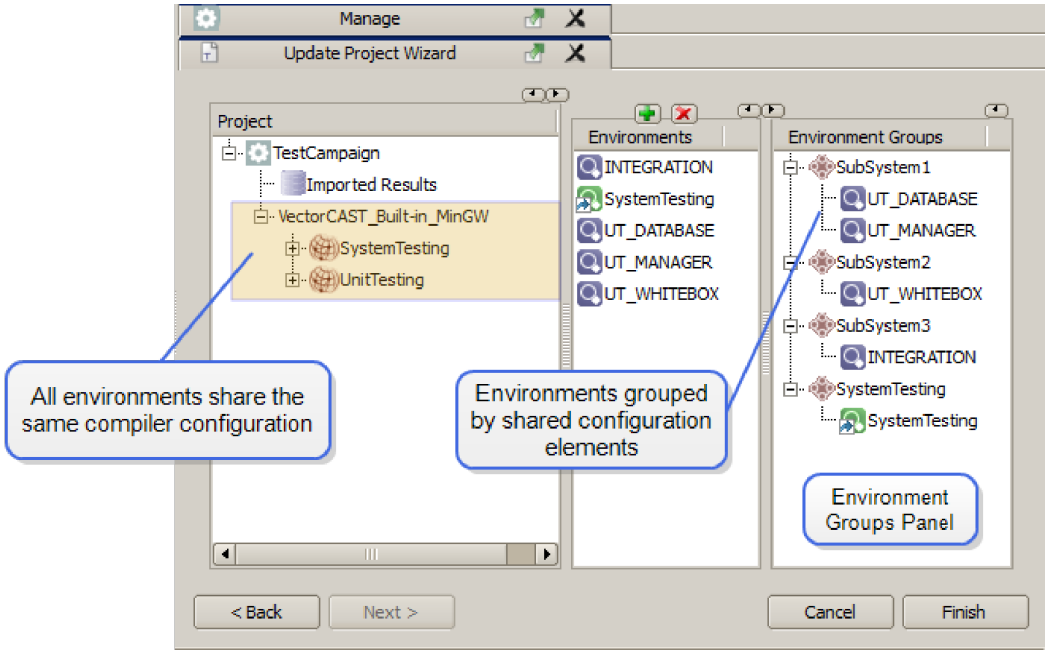
To add an environment manually, click the **Add** button ( ) or right-click in the panel and choose **Import Environments** from the context menu. The standard Add Environment dialog appears. You have a choice of searching for Environment Files (.vce or .vcp), Script Files (.env), Environment and Script Files or All Files. Click **Open** to import.

To Remove an Environment

To remove an environment, select the environment in the Environments Panel, and then click the **Remove** button along the top ( ) or right-click on the environment name in the panel and choose **Remove Environment** from the context menu.

The Environment Groups Panel

The right-most panel is the Environment Groups Panel. It allows you to organize your test environments into collections that have similar characteristics, or that need to be tested under similar configurations. In the example below, all environments share the same configuration and there are two test suites with four environment groups.



To Add an Environment Group

To add an Environment Group, click the **Add** button ( ) or right-click within the Environment Groups Panel and select **Add Environment Group**. Enter the name of the new Environment Group and click the **Add** button. The new group is displayed in the Environment Groups Panel. Add environments to the group by dragging environments from the Environments Panel and dropping them into the new group.

To Remove an Environment Group

To remove an Environment Group, select the group in the Environment Group Panel, and then click the **Remove** button along the top ( ) or right-click on the group name in the panel and choose Remove from the context menu. You are prompted that you will permanently remove all of the selected entries and their sub-components. Click **Yes** to continue.

Deleting Nodes in the Project Tree Panel

A deletion removes a node from the `project.vcm` file. The user can multi-select several nodes, right-click on one, and select **Delete** from the context menu. The type of node right-clicked is the type of node that is deleted from the selection. This functionality supports the selection of Compiler and Test Suite nodes.

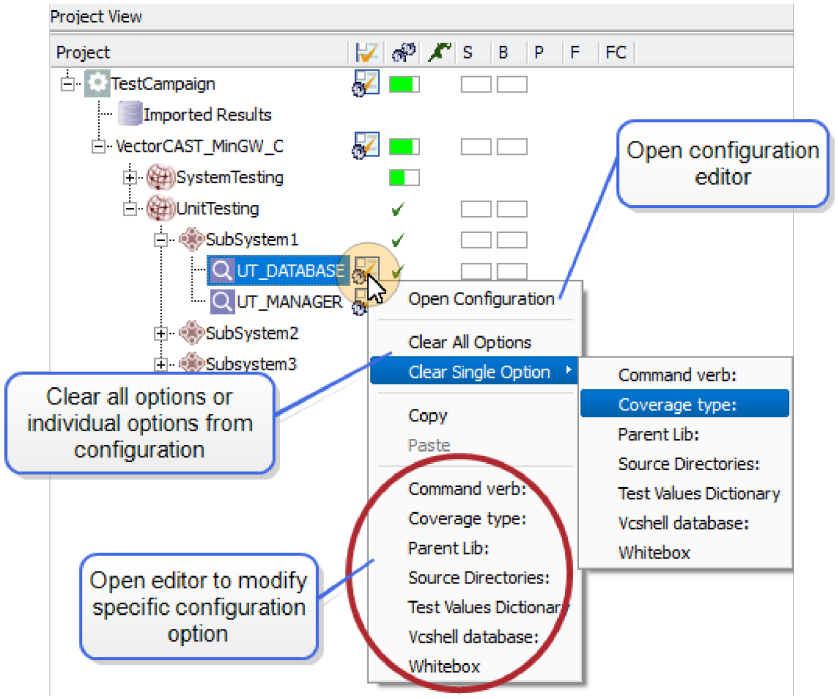
Deleting a node does not remove the environment from the flat list in the Environments Panel. Removing an environment from the Environments Panel causes VectorCAST to delete the underlying regression script files and to remove the environment from the Project Tree, the Environments Panel, and Environment Groups Panel.

Editing and Modifying Configurations

VectorCAST allows you to quickly adjust your configuration options to ensure tests are run effectively in appropriate environments.

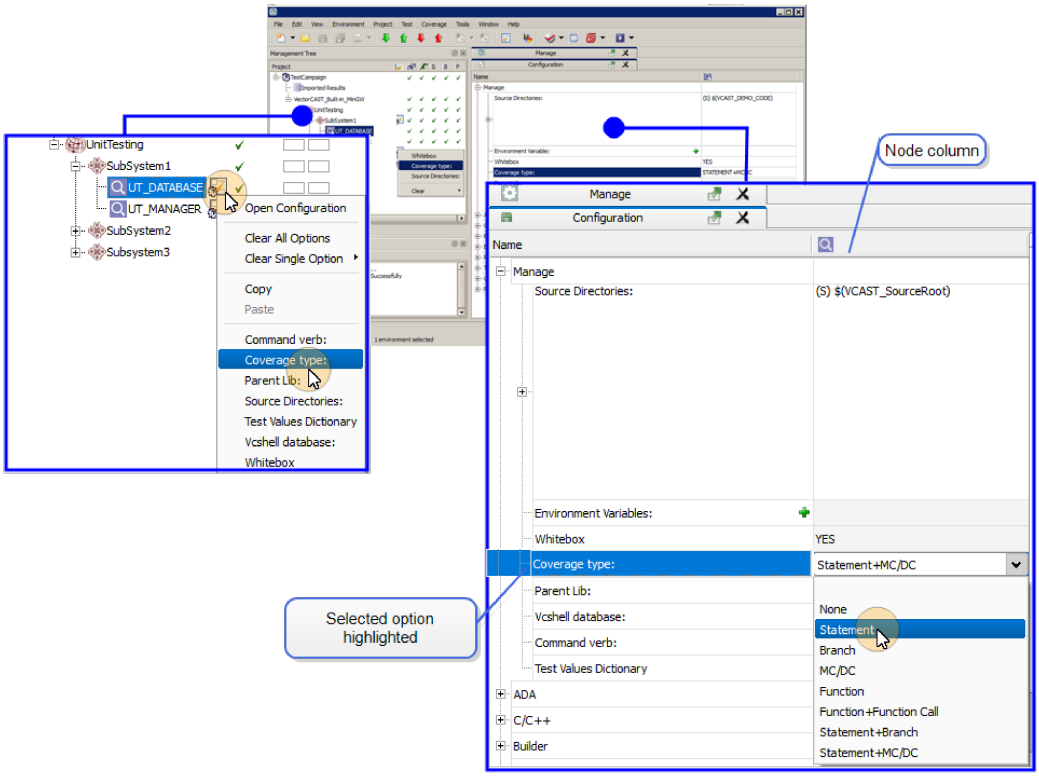
Configuration Context

To edit configuration options for a node, select and right-click the node's configuration options icon  in the Project Tree. A context menu is displayed that allows you to edit or clear a single configuration option, to clear all configuration options for the node, or to open the Confguration Editor.



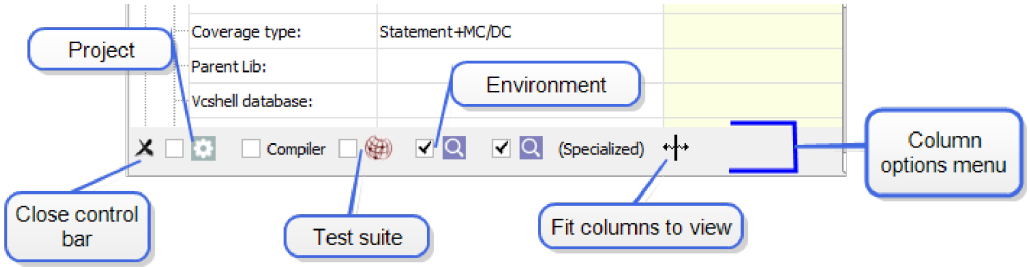
Modifying Configuration Options

When a configuration option is selected from the context menu, the Configuration editor is opened and the name of the selected option is highlighted in the editor. The option's value can be modified by clicking its current value and then making the desired changes. For the example below, **Coverage Type:** was selected from the context menu and it is highlighted in the editor.



If you right-click a value in the node column, a context menu is displayed that allows you to:

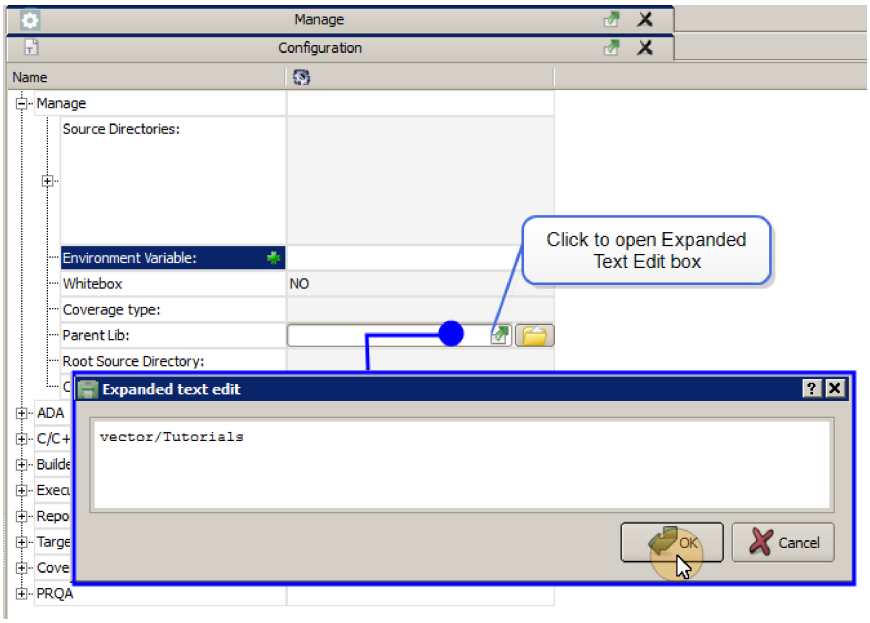
- **Load Default** – sets the option to the default value.
- **Cut** – removes the option value
- **Copy** – copies data and stores on clipboard
- **Paste** – inserts data from clipboard
- **Clear** – clears the option value.
- **Column Options** – exposes the Columns Options menu at the bottom of the editor window if not already exposed. This menu allows you to add or remove columns to the configuration options editor for each of the node levels in the Project Tree by checking the box. Click the X icon to hide the menu. Drag the Re-size bar at the top of the column to re-size.



One example for using the Column Options feature is to promote a configuration option currently set in several lower level nodes (i.e. environment nodes), to a single higher level node (i.e. compiler) in

the Project Tree, by constraining the option to the higher level and factoring the option out of the lower levels. To implement this change, a column for the compiler level would be added to the editor by selecting it on the Column Options Menu. The desired option value is selected and set in the compiler column and then each environment is selected, and its configuration option value is cleared in the environment column.

An Expanded Text Edit box is available to make it easier to enter the text of selected configuration options. To open the box, click on the green arrow icon within the text field. Type in the text and select the **OK** button when complete. The text field populates with the text entry.



Detecting Configuration Changes

The environment variable VCAST_CBT_CFG_CHANGES is used to detect changes in a project's configuration.

Note: The VCAST_CBT_CFG_CHANGES environment variable is only applicable to migrated unit test environments in VectorCAST projects and is especially useful for those working in the continuous integration or change-based testing workflows.

If this variable is set either in the shell or at the project level, any change made to the project configuration at any level is detected. The affected environments are marked as needing a full rebuild and their icons are flagged to indicate they are outdated. Selecting **Properties** from the right-click menu on an affected environment opens the Properties dialog, which lists the configuration option(s) that have changed.

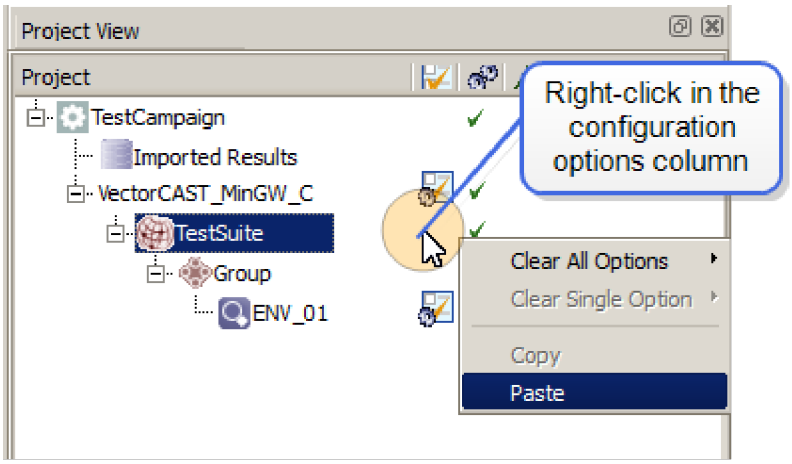
```
manage -p project --dependency-changes
```

Issuing an incremental build-execute using either the command line or the GUI while this environment variable is set causes VectorCAST to perform a *full* rebuild and test execution.

```
manage -p <project> --build-execute --incremental
```

Copying and Pasting Configuration Options

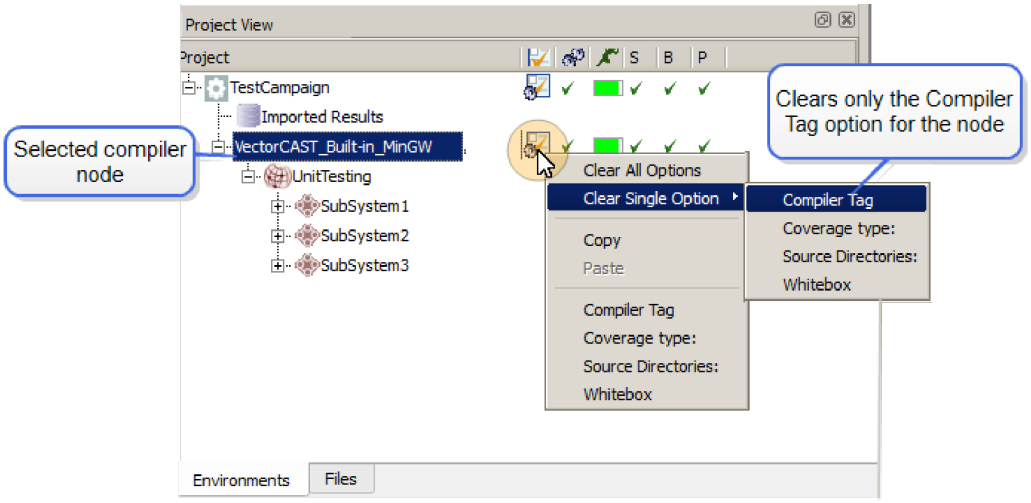
An entire configuration for a node can be copied and pasted to another node. To copy and paste configuration options, right-click on the Configuration node icon next to the desired node and select **Copy** from the context menu. Right-click in the Configuration Options column next to the second node and select **Paste** from the context menu. A Configuration node icon appears in the column and the new configuration is applied.



For example, you might want to elevate the configuration options from a test environment up to the testsuite node level. This would ensure that all new environments in that testsuite would share the same set of options. Also, any changes to those options could be controlled in the single testsuite node, rather than in each individual environment.

Clearing Configuration Options

To clear one option of a single node or multi-selected nodes, right-click one of the selected nodes and choose **Clear Single Option** from the context menu. A cascade menu is displayed listing the options. Use this menu to clear the value of one or more options from the Project Tree.



Right-clicking a single node's configuration options icon in the Project Tree and selecting **Clear All Options** from the context menu permanently removes all configuration items for the selected node.

When multiple nodes are selected, right-clicking a configuration options icon in the Project Tree and selecting **Clear All Options** from the the context menu provides a cascade menu. Select:

- **Clear All Options => Selected Node(s)** to clear all options of the selected nodes.
- **Clear All Options => All Children** to clear all options of only the child nodes of the selected nodes.

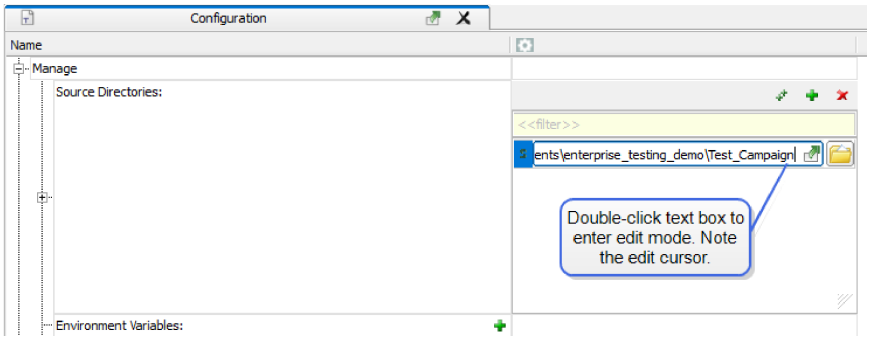
Editing Paths

It is possible to edit paths in two places in VectorCAST:

- On the first page of the Wizard, when you specify the paths to directories in which VectorCAST should look for environments
- In the Source Directories option in the Manage Configuration window, which VectorCAST uses to find the source files for the environments

Note: To allow file system paths to be made portable across source code repository checkouts, it is best practice to use the Base Directory configuration option. See Using Base Directories for more information on this option.

To edit a path, double-click the path to put it in "edit mode." The edit box shows an outline around it and there is an edit cursor. When finished editing, press **Enter** or click the mouse elsewhere to take the path out of edit mode.



In addition, it is possible to add environment variables to these paths. To add an environment variable in a path, use the syntax: $(ENV_VAR)

For example, suppose you want to specify that the environments in the VectorCAST project use SOURCE_BASE_1 or SOURCE_BASE_2 when building, depending on the value of an environment variable. You define the environment variable SRC to be SOURCE_BASE_1:

```
set SRC=SOURCE_BASE_1
```

Then, edit the Configuration option "Source Directories," which currently specifies:

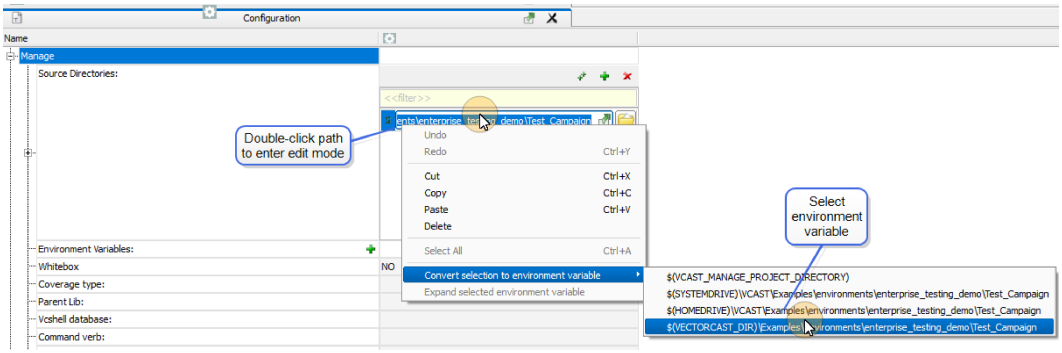
```
C:\testing\environments\SOURCE_BASE_1
```

to use the environment variable:

```
C:\testing\environments\$(SRC)
```

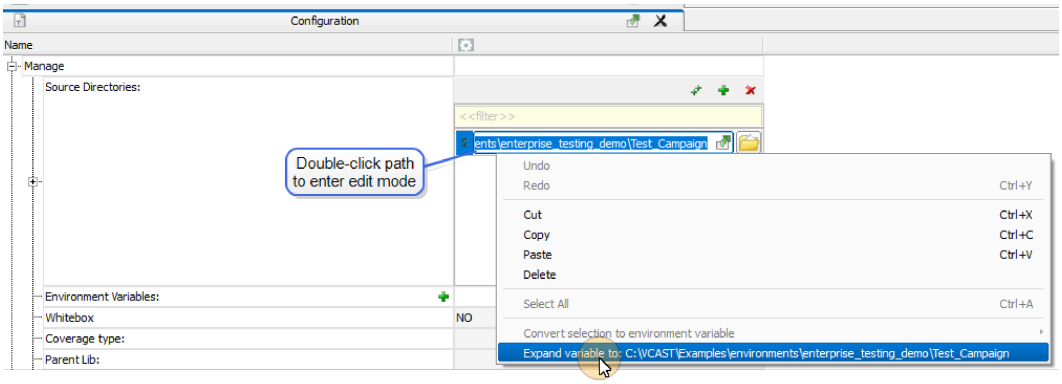
If you change the value of SRC and build the environments again, they use the new location for source files, as specified by the environment variable.

A convenient way to convert a path to use an environment variable is to double-click to select the path and enter edit mode, then right-click and choose **Convert selection to environment variable** and select the environment variable from the context menu.



VectorCAST detects if the value of a known environment variable matches a portion of a path using a string comparison.

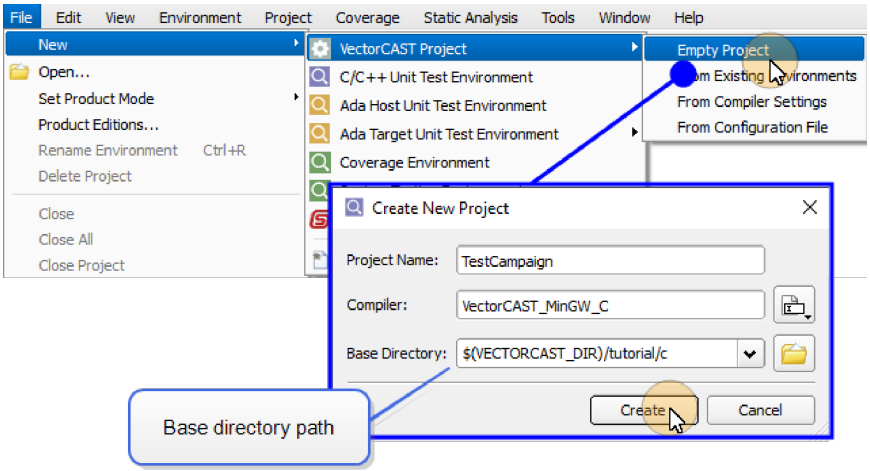
To expand an environment variable back to a path, double-click the variable to enter edit mode. Right-click and choose **Expand variable to: <path>** from the context menu.



Using Base Directories

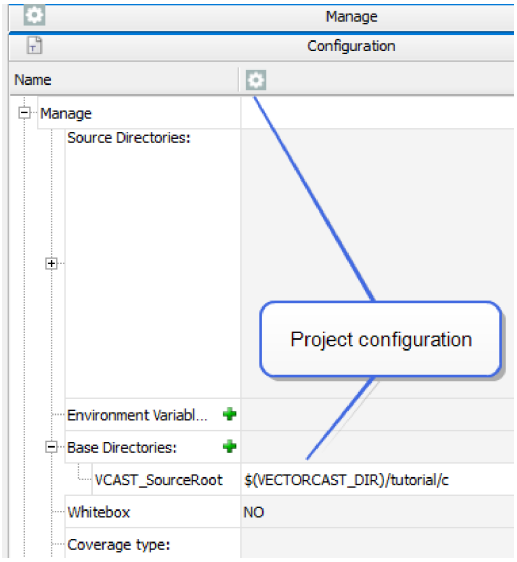
The Base Directory configuration option allows file system paths to be made portable across source code repository checkouts.

A base directory is a NAME=PATH pair, where NAME is an identifier for a base directory and PATH is a directory that points to the root of the source code repository under test. Once set, any newly created C/C++ unit test or Coverage environments will use the base directories from the project to produce source code references that are relative to the base directory, ensuring that the VectorCAST project is portable.



To specify an environment variable, use the syntax **`$(env_var)`**, where **`env_var`** is the name of an existing environment variable. It is recommended that the base directory path be set to an environment variable or relative path to ensure portability.

Base directories can be added to existing VectorCAST Projects by adding them to the project's configuration level. To configure a base directory, select the project node in the Project tree, right-click, and select **Open Configuration** from the context menu. Expand the Manage node in the Configuration Editor and click the Add button ✚ located next to the Base Directories field to open the Add Base Directory dialog. Enter a name and the path to the base directory and click the **Add** button.



Note: VectorCAST Projects can still be created without base directories. Scripting that is making use of the manage command line to create VectorCAST Projects will continue to work without issue. Also, creating a VectorCAST Project from existing environments or .CFG files does not require a base directory.

VectorCAST Projects without base directories are not changed in any way, and will continue to function normally.

```
manage -p <project> --base-dir NAME=PATH

    Set the source code's Base Directory for the project. PATH must
    resolve to a valid directory. Use this command to update the path for
    a specific Base Directory.
```

```
manage -p <project> --unset-base-dir NAME

    Remove a Base Directory from the project.
```

Note: The Base Directory may not be set using the **`--config`** argument.

The following example incorrectly uses the **`--config`** argument, generating an error message:

```
manage -p Main_Project --level GNU_CPP_X/SUITE --config BASE_
DIRECTORY=src
```

The correct syntax is:

```
manage -p Main_Project --base-dir SourceRoot=src
```

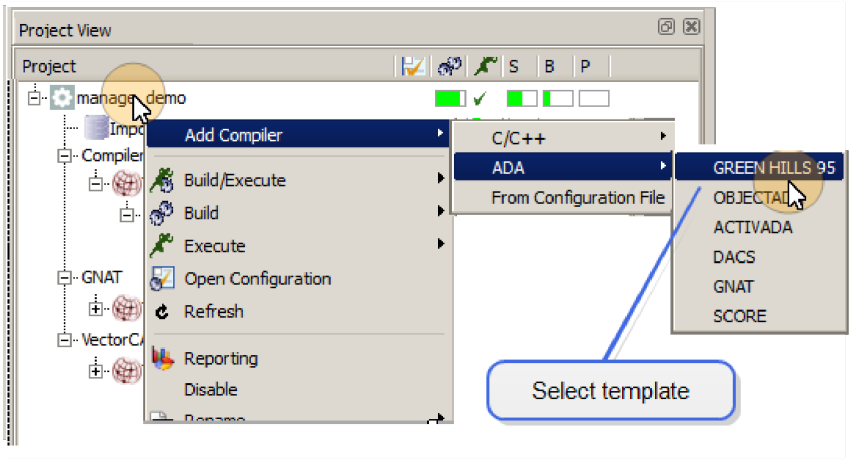
Adding a Compiler

Compilers can be added by using the supplied templates or by importing an existing .CFG file. Once a compiler has been added, test suites can then be added to the node.

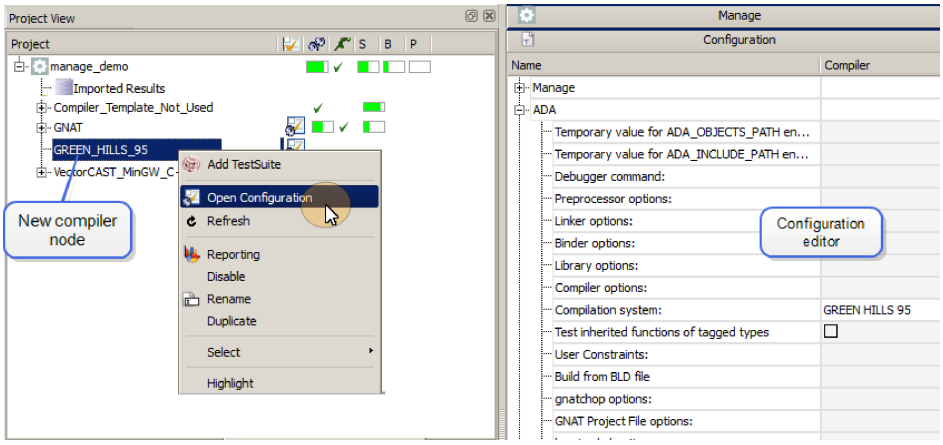
Note: Existing test suites can be dragged and dropped from one compiler context to another using **Ctrl+click** or **Shift+click** to select the test suites of interest.

To Add a Compiler From a Template

Right-click on the Project root node and select **Add Compiler**. Select the desired template from the context menu list of supported compilers and their templates.



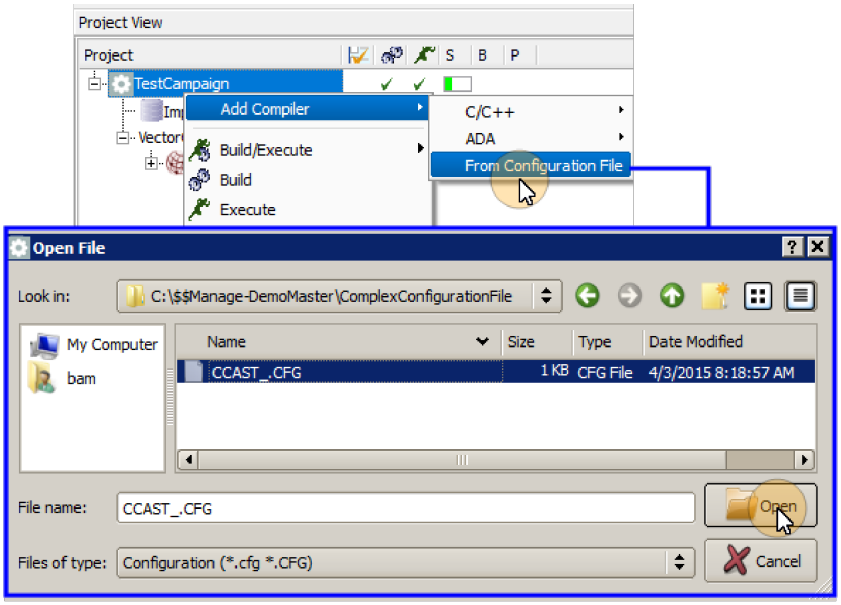
A new Compiler node is created in the Project Tree. Right-click on the Compiler node and select **Open Configuration** from the context menu. The Configuration Editor opens allowing you to set compiler options as needed.



To Add a Compiler From a .CFG File

VectorCAST provides a feature allowing the import of customized .CFG files to create new Compiler nodes. This option eliminates the need to modify the supplied template settings in the Configuration Editor and is useful in projects with complex compiler settings.

Right-click on the Project root node and select **Add Compiler => From Configuration File** from the context menu. Navigate to the location of the .cfg file and select the **Open** button.




```
manage -p PROJECT --cfg-to-compiler=<CFG_FILE>

Creates a compiler from a given CFG file.
```

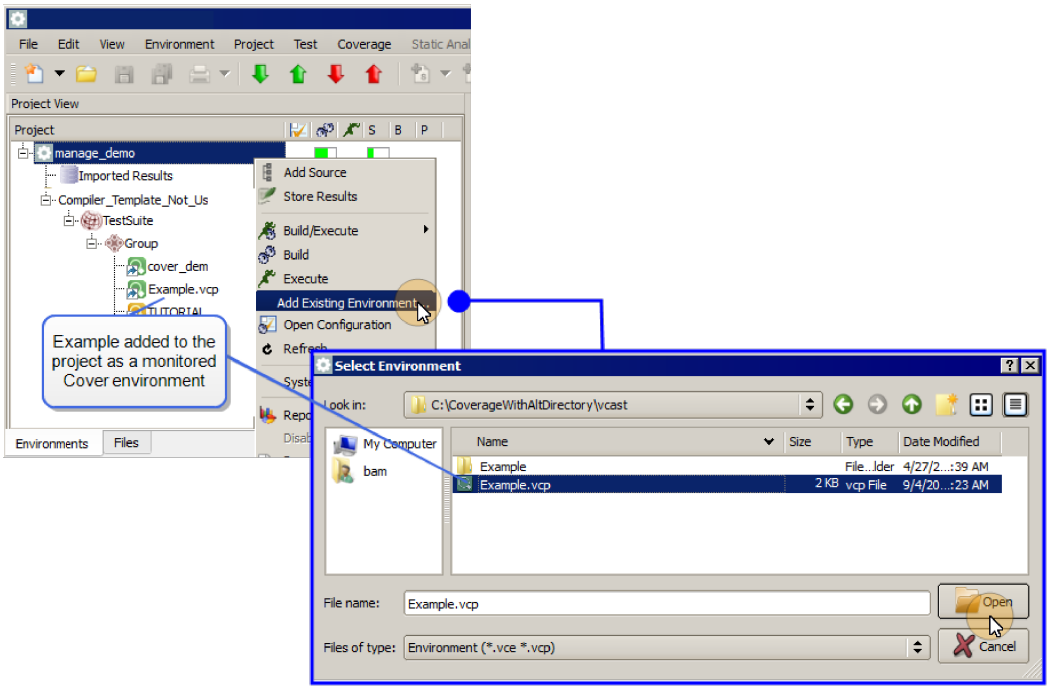
For more information about adding compilers using a .CFG file see the work flow "Creating a Compiler Node Using an Existing .CFG File" on page 41.

Add an Environment to the Project Tree

Unit Test or Cover environments can be added directly into the VectorCAST Project Tree using one of two methods:

- Via a Project or Group node level's right-click context menu item, **Add Existing Environment** or,
- Via drag and drop functionality.

To add a Unit Test or Cover Environment using the context menu, right-click on either a Project or Group node and select **Add Existing Environment...** from the context menu. From the file browser dialog, select either a .vce (Unit Test environment) or .vcp (Cover environment) file to create a monitored environment within VectorCAST.



When adding an environment at the Project node level, VectorCAST will either identify an existing location or create a new location for the environment based on the environment's configuration. Use the Group node level to add an environment to a specific group in the project.

Dragging and Dropping Environments

The drag and drop functionality in the Project Tree and in the Update Project Wizard is useful for adding and organizing environments. When dragging environment files from a Windows Explorer window into a Project or Group node in the Project Tree, the following unit environment types are created:

- Drag and drop of an .env file onto a Group or Project node creates a migrated unit test environment within VectorCAST.
- Drag and drop of a .vce file onto a Group or Project node creates a monitored unit test environment within VectorCAST.
- Drag and drop of a .vcp file onto a Group or Project node creates a Cover environment within VectorCAST.

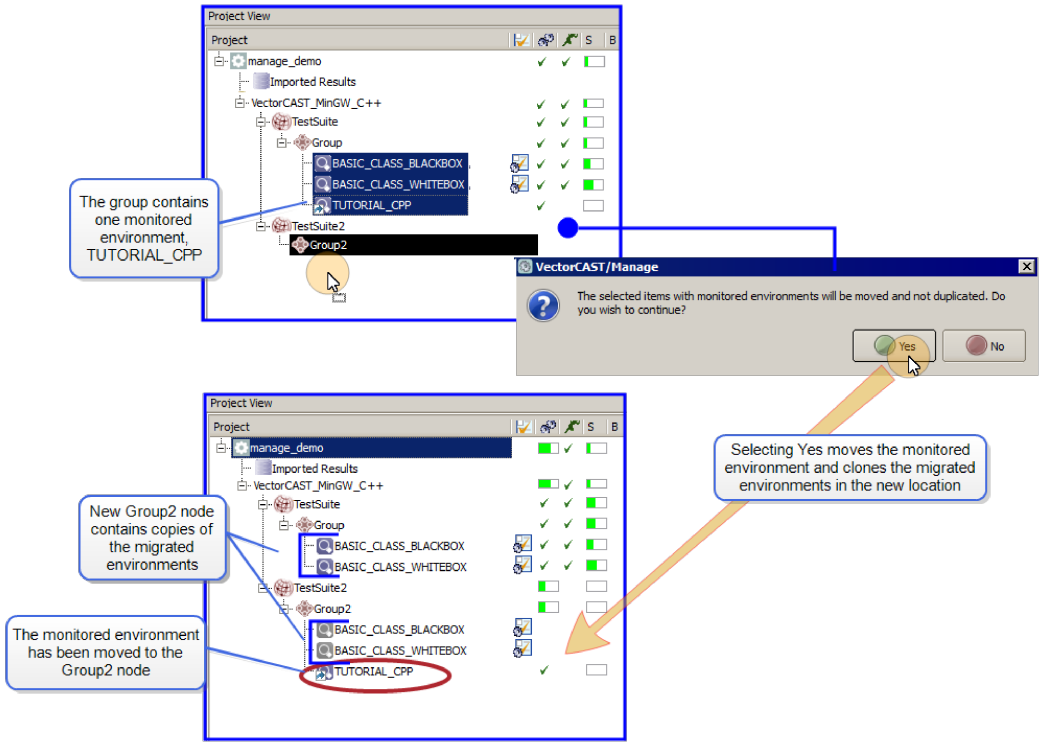
When a migrated unit test environment is dragged and dropped within the Project Tree, the environment is cloned in the new location.

When a migrated Cover environment is dragged and dropped within the Project Tree, the Cover environment is moved to the new location. Cover environments can only exist in a single context and cannot be cloned.

Note that monitored environments for both unit test environments and Cover environments behave differently than do migrated environments when dragged and dropped. Because monitored environments can only exist in a single context, when a monitored environment is dragged and dropped, the environment is moved to the new location (and is not cloned). See "Using Monitored

Environments in a VectorCAST Project" on page 137 to learn more about Monitored environments.

Note in the example below that the selected group contains one monitored unit test environment and two migrated unit test environments. When the group is dragged and dropped in the new location, the monitored unit environment is moved to the new location (and is no longer in the original location) and the migrated environments are cloned in the new location (and exist in both the original and new locations).



Add Multiple Environments to the Project Tree

Multiple Unit Test environments can be added at the same time directly into an existing VectorCAST project using the CLICAST command:

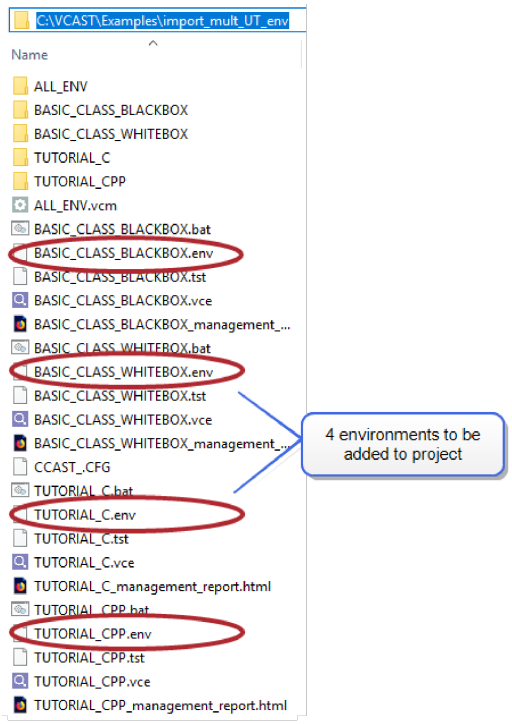
```
--import-all <path to directory containing multiple .env scripts>
```

The command takes as an argument a directory containing multiple environment scripts (.env), and is not recursive. All the environments to be imported must reside in the same directory.

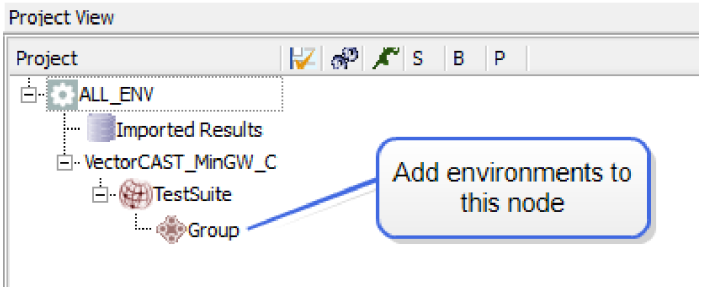
The **`--import-all`** command can also be used with the following arguments:

`[--level=<compiler>/<testsuite>]`	Adds environments to the given level
`[--group=<group>]`	Adds environments to the given group
`[--force]`	Creates any missing nodes in the specified level
`[--migrate]`	Migrates the environments to the given testsuite when they are imported

For example, we have an existing VectorCAST project called **ALL_ENV** and we wish to add four environments into the project. The four environments reside in the same directory, **import_mult_UT_env**.

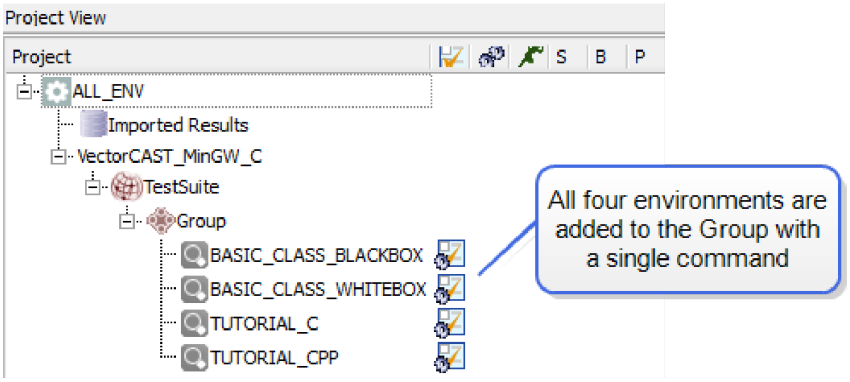


We want to add these environments to the Group node in our project.



We can import the environments one at a time, or we can use the following command to quickly and easily import them all at the same time:

```
%VECTORCAST_DIR%\manage -p ALL_ENV --level=TestSuite --group=Group --
migrate --import-all
```

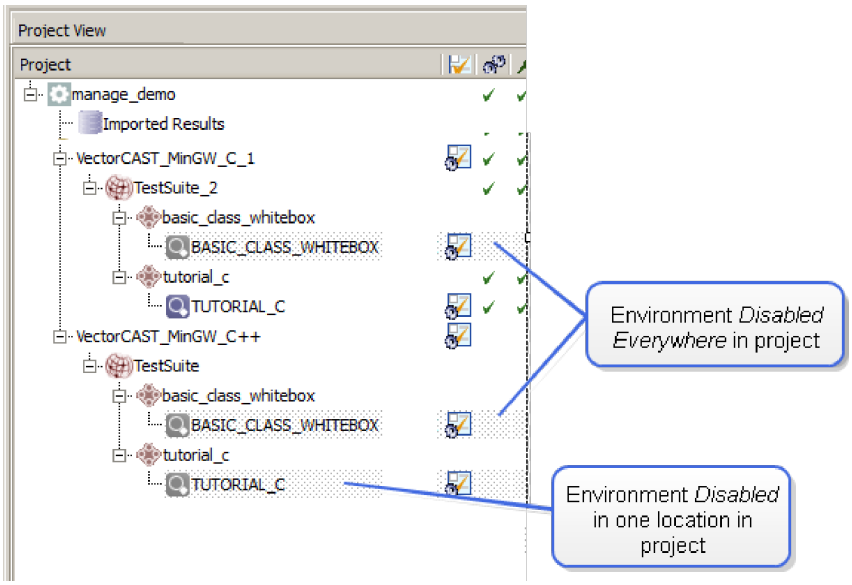


Disabling/Enabling an Environment

To disable an environment, right-click the name of an environment in the Project Tree. When an environment is disabled, it cannot be built or executed and the metrics for that environment will not be included in the aggregate metrics for the project. A disabled environment is displayed with a light gray background. There are two options for disabling an environment: **Disable** or **Disable Everywhere**.

To disable an environment in every location where it appears in the project, select **Disable Everywhere**.

To disable an environment only in a single selected location, select **Disable**.



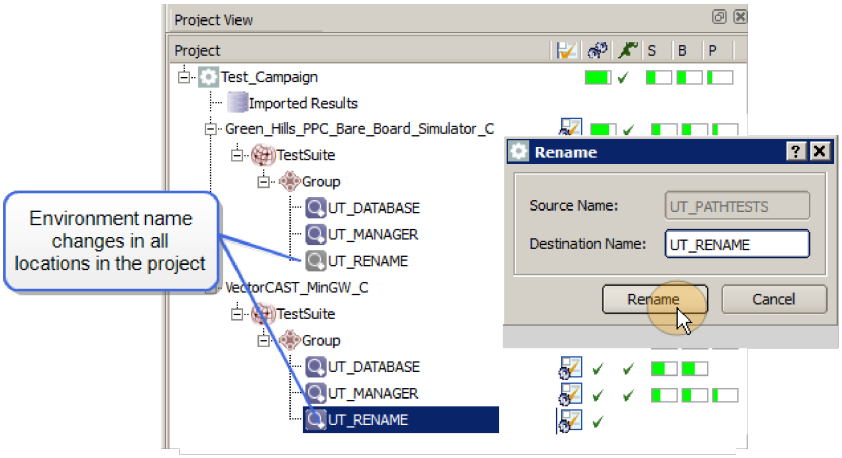
To re-enable an environment in all locations where it appears, select **Enable Everywhere**. To re-enable an environment in a single location, toggle the **Disable** option from the right-click context menu.

Renaming an Environment

Only migrated unit test environments can be renamed. To rename a migrated unit test environment,

right-click on the environment and select **Rename** from the context menu. Environments can be renamed from the Project Tree and from the Project Wizard's Environments and Environment Groups panels.

Enter a new, unique environment name. Duplicate names are not allowed within a project. Select the **Rename** button. All environments within the project are updated to use the new name.

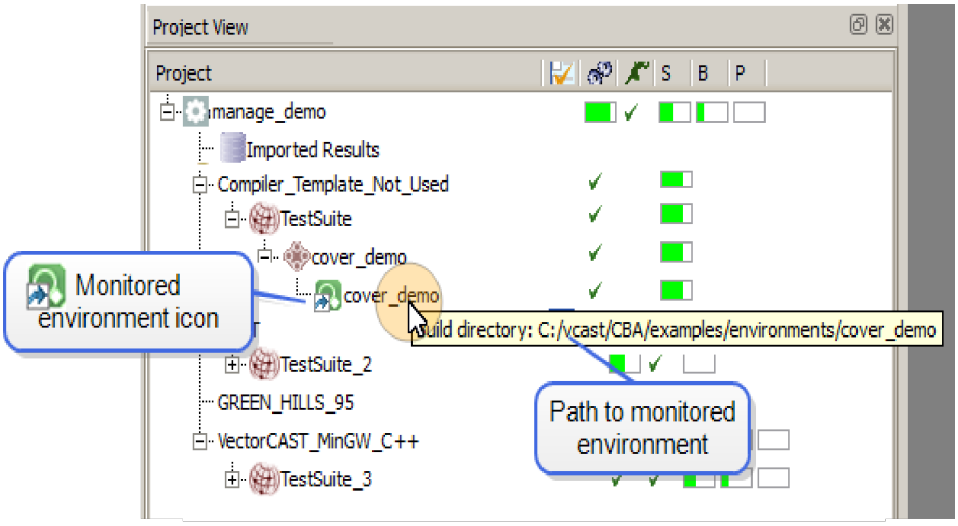


Note: Monitored unit test and Cover environments and migrated Cover environments cannot be renamed. Only migrated unit test environments can be renamed.

Using Monitored Environments in a VectorCAST Project

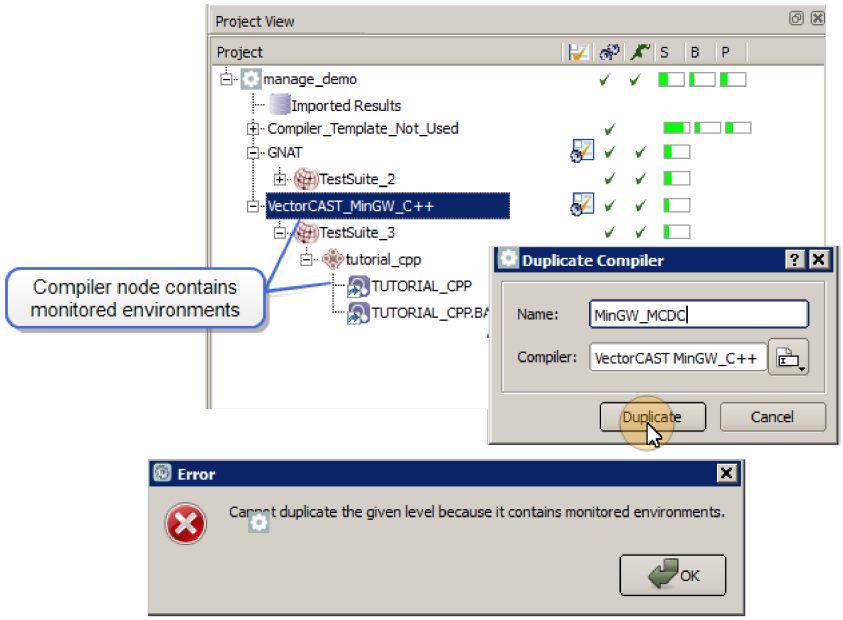
Monitored environments are links to environments which reside in their original location and not within the Project directory. In the Project Tree, monitored environments are identified by an icon with a small arrow. Hovering over the monitored environment will display a Tooltip showing the path to the original environment.

Note: Using VectorCAST to monitor external test environments, while supported, is not the recommended work flow. Many of the advanced testing features, such as Change-Based Testing and Environment Cloning, are not supported for monitored environments.



A monitored unit test environment cannot be in two different contexts until it is migrated. Only migrated unit test environments can be duplicated. Therefore, if you drag a monitored environment into another group or test suite, it will be moved to the new location. Also, you cannot duplicate any level in the tree that contains monitored environments.

Note: Migrated Cover environments behave differently from migrated unit test environments. Migrated Cover environments cannot be in two different contexts. Dragging a migrated Cover environment will move the environment to the new location and it will not be duplicated.



Because VectorCAST does not maintain any monitored environment files in the Project directory, there is no notification to the user if a monitored environment is opened, changed, and closed. If a monitored environment is changed outside the VectorCAST project, right-click on the environment

node and select **Refresh** from the context menu. The environment will update and provide the current build, execution, and coverage status to the VectorCASTproject.

A monitored environment that was imported as an existing environment cannot be cleaned. Cleaning would delete the original environment and it could not be rebuilt.

A monitored environment that was imported from its regression scripts can be cleaned. It can be built by running the regression scripts. Selecting the **Build** command runs the regression scripts the first time. Subsequent builds rebuild the environment.

If you import environments as regression scripts and keep them as monitored environments, you can delete the <project> directory and save only the <project>.vcm file and the regression scripts under Source Code Management (SCM). This is possible because opening the <project>.vcm file recreates the entire VectorCAST project, assuming it can find the regression scripts in the same locations as they were before.

Opening the <project>.vcm file will not recreate the environments if they have been migrated. This feature only works if they were still monitored.

File Hooks Scripting

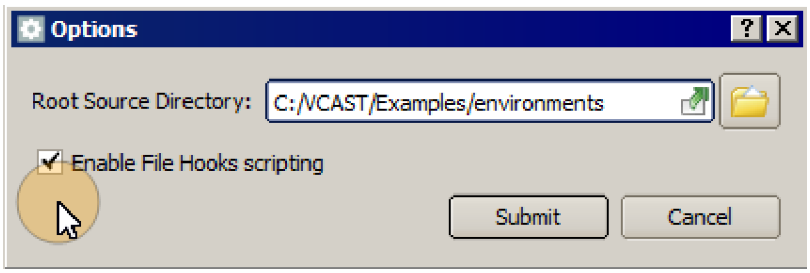
File hook processing provides an open python interface to interact with the VectorCAST project and migrated environment files that would normally be involved in source code management (SCM). This feature allows the user to perform actions, such as modifying file permissions, line endings, or other processing, before and after VectorCAST modifies them.

The following files for migrated environments are audited by the File Hooks Script:

- project.vcm file
- project.db file
- *.mfg files
- *.cfg files
- *.env files
- *.tst files, including specialized test case scripts

Enable/Disable File Hooks Scripting

To enable or disable the functionality, from the Menu Bar, select **Project => Options**. On the Options dialog, check **Enable File Hooks scripting** to enable the functionality. Un-check the box to disable the functionality. Click the **Submit** button.



```
%VECTORCAST_DIR%\manage -p <PROJECT> --enable-file-hooks-scripting

Enables the file hooks scripting support.
```

```
%VECTORCAST_DIR%\manage -p <PROJECT> --disable-file-hooks-scripting

Disables the file hooks scripting support.
```

Running the File Hooks Script

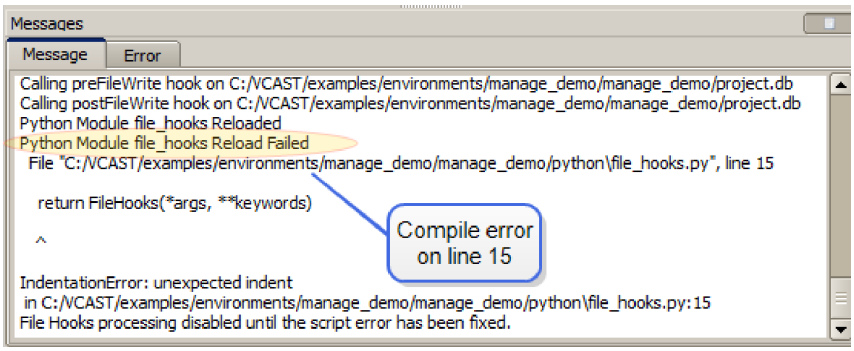
If enabled, the following actions trigger the File Hooks Script to run:

- Migrating an environment
- Building and executing an environment
- Modifying an environment
- Modifying the project (i.e. adding or removing an environment)
- Storing results

Editing the File Hooks Script

To edit the python script (**`file_hooks.py`**), from the Menu Bar, select **Project => Edit File Hooks Script**. The file opens in the Text Editor in edit mode and changes can be made as required.

When saved, the python file is reloaded. In the case of a compile error, VectorCAST alerts the user of the failure in the Message Window and if the python file is still open in the editor, scrolls to the approximate location of the error.



Understanding Workspaces

A *workspace* is a disk directory that contains the actual tests that are executed by VectorCAST. Workspaces may exist on the same machine as the VectorCAST Project, or on alternate machines. The two main reasons to use multiple workspaces are:

- Project testing can be spread across multiple machines. Each machine will have its own workspace. The status from these workspaces can be “pushed” to the VectorCAST Project or imported to the VectorCAST Project. The results are included in the Status Panel and Regression Reports.

- Individual developers can debug their tests without affecting the project results. Each developer can have their own “local” workspace where they can work on creating new tests and debugging existing tests without affecting the master data.

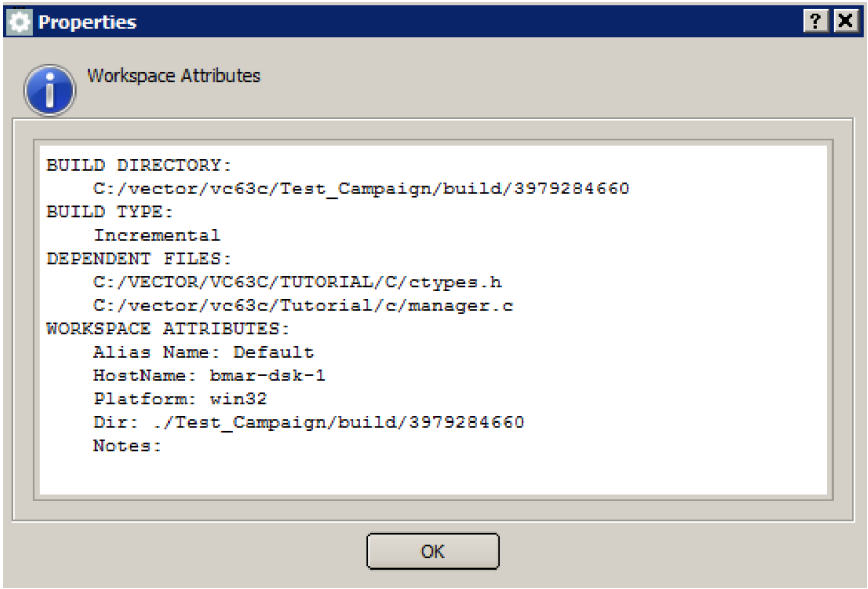
The following sections describe the controls available when working with workspaces.

The Default Workspace Location

By default, the workspace is set to the *project/build* directory when a VectorCAST Project is first created and when the Project is opened, unless the workspace has been changed.

To Determine the Current Workspace

If you want to know which workspace is being used for an environment, built or not, right-click the environment and choose **Properties**.



BUILD DIRECTORY: The directory in which the test environment will be or has been built in the current work area.

BUILD TYPE: Full or Incremental build.

DEPENDENT FILES: The source files used by this test environment. Empty if the environment has not been built.

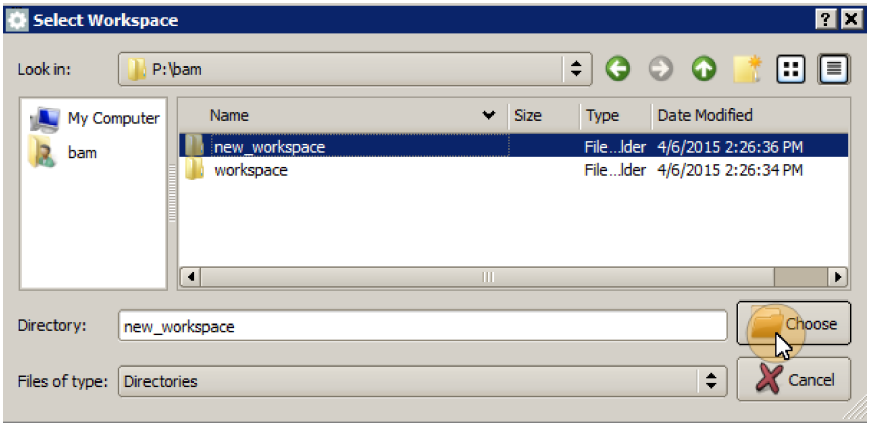
WORKSPACE ATTRIBUTES: Empty if the test environment has not been built. Otherwise, indicates the attributes of the current workspace:

- *Alias Name:* The Alias name, or nickname. The alias can only be assigned when the workspace results are imported.
- *HostName:* The hostname of the machine on which the workspace directory resides.
- *Platform:* The platform of the machine on which the VectorCAST Project resides when that workspace was selected or imported. Choices are: win32, linux, solaris.

- *Dir:* The location of the current workspace. If relative, it is relative to the working directory of the VectorCAST Project.
- *Notes:* Notes written when the workspace results are imported.

To Change the Location of the Workspace

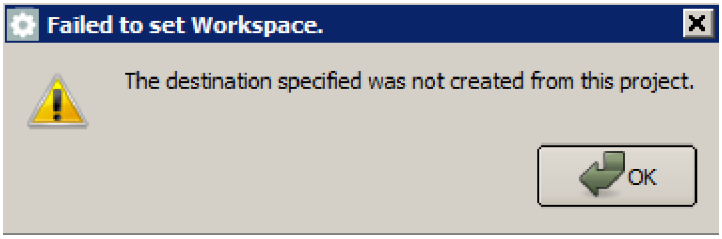
When you build or execute a test environment in the VectorCAST Project, the current workspace is used to store the environment directory etc. To change the workspace to a different location, choose **Project => Select Workspace...**. Navigate to a different directory on your local drive, a mapped drive, or a networked drive. Click **Choose**.



You can change the workspace to an empty directory for the purpose of building/executing test environments there. This workspace's results can then be imported by the VectorCAST Project even if it is on a different machine.

You can change the workspace to one which already has been used (and therefore it has compatible data), then the Status Panel shows the build, execute, and coverage data as soon as you click Choose.

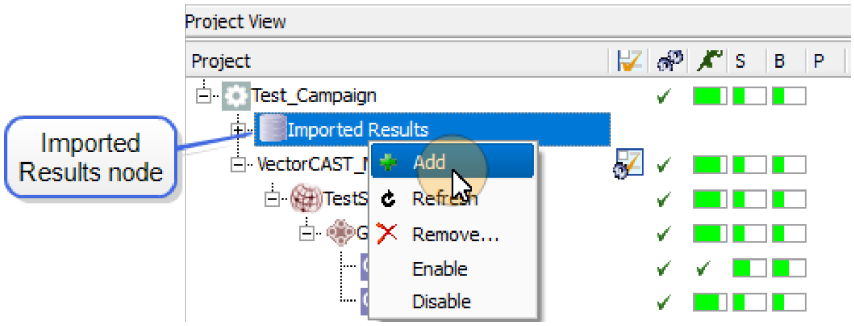
If you navigate to a workspace that was used for another VectorCAST Project and click Choose, then you see the error message "Failed to set Workspace."



Working With Imported Results

Users can import results (build, execution, and coverage results) from a clone of a VectorCAST project using the Imported Results node located at the top of the Project Tree. This action is like taking a snapshot of the data in that workspace at that moment. The Imported Results node provides

a right-click menu to add, remove, refresh, enable, and disable imported results.



If changes occur to the data in the workspace after you import, they are not reflected in the Status Panel unless you refresh the imported results.

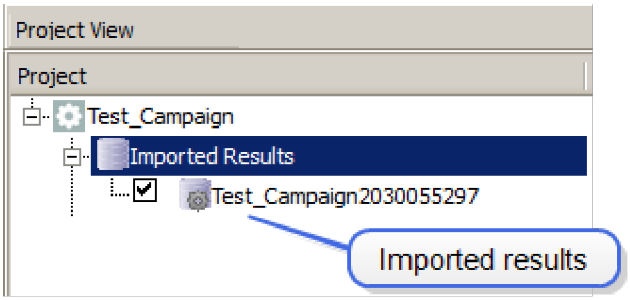
Additionally, you can export a snapshot of the clone's results in a zipped results file (*.vcr). With the project open whose results are to be exported, choose **Project => Export Results**. In the dialog, navigate to the destination for the file, enter a name, and click **Save**. The file extension is .vcr (for VectorCAST Results).

Import Results From a Clone Project's Current Workspace

1. Right-click the Imported Results node in the "original" project (the one receiving the imported results), and choose **Add** from the context menu.

2. In the dialog, navigate to the cloned project's directory. Here, you can select the project's .vcm file itself. The default alias is "<name of cloned project><project ID>".

3. Click **Import**. The build, execution, and coverage results from the clone project's default workspace (build) are imported to the project.



Results imported by the clone project's .vcm file are automatically refreshed on Project open or they can be manually refreshed by right-clicking and choosing **Refresh** from the context menu.

Import Results From Any of a Clone's Workspaces

1. Right-click the Imported Results node in the "original" project, and choose **Add**.

2. In the dialog, navigate to location of the cloned project's workspace and click the <workspace>.vcw

file. The new file extension .vcw stands for "VectorCAST workspace." The default alias is "<name of workspace directory>".

3. Click **Import**. The build, execution, and coverage results from the clone project's workspace are imported to the project.

Results imported by the clone project's workspace (.vcw) file are automatically refreshed on Project open or they can be manually refreshed by right-clicking and choosing Refresh.

Import a Previously-exported, Zipped File of Results

1. Right-click the Imported Results node in the "original" project, and choose **Add**.

2. In the dialog, navigate to the zipped file and click **Open**. The default alias is the name of the file.

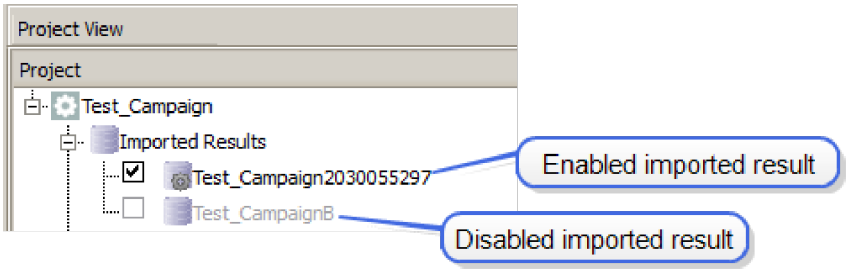
3. Click **Import**. The build, execution, and coverage results from the zipped file of results are imported to the project.

Results imported by a zipped file (.vcr) file cannot be refreshed. They are a snapshot of the results at the time they were exported. If the zipped file (.vcr) is overwritten with a newer file, the results are "refreshed" on Project open.

Enable/Disable Imported Results

By default, a workspace is enabled when imported. When a data source is enabled, its data is available to be used in the Status panel. Enabled imported results are indicated by a check mark and displayed in normal black text.

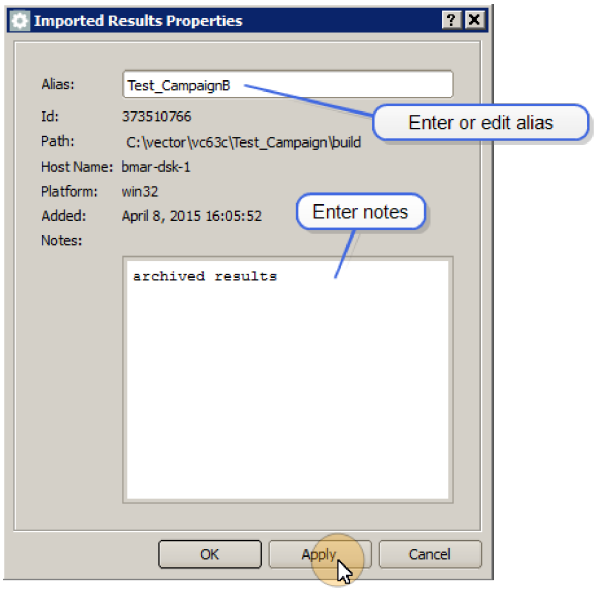
To disable a selected imported result, uncheck the box to the left of the imported result. Disabled results are displayed in gray, or dimmed, text.



To enable or disable all imported results simultaneously, right click on the Imported Results node and select **Enable** or **Disable** from the context menu.

Give an Imported Result an Alias

An *alias* is a nickname for the Imported Result to make it convenient for you to identify it in the list of Imported Results. It is not required. If an Imported Result does not have an alias then it is identified as "*…/last directory*" in the list. Spaces and any special characters may be used. To change an Imported Result's alias in the Imported Results window, right-click it and choose **Properties**. Change the alias and click **Apply**. Click **OK** to close the Imported Results Properties dialog.



Notes can also be entered on the Imported Results Properties dialog and associated with the selected Imported Results.

Remove Imported Results

To delete an selected imported result, right-click a result and select **Remove** from the context menu. A warning prompt will appear to confirm that you want to remove the selected items. Click the **Yes** button to confirm. The imported result is removed from the list and its data is removed from the Status Panel.

To delete all imported results, right-click on the Imported Results node and select **Remove** from the context menu. A warning prompt will appear to confirm that you want to remove the selected items. Click the **Yes** button to confirm. All imported results are removed from the list and their data is removed from the Status Panel.

Refreshing Imported Results

Each time the Project is opened, the data in each imported result is refreshed. That is, if there has been any change to the data in the workspace caused by another instance of the VectorCAST Project, then upon re-opening, that data is now present in the Status Panel.

For example, suppose you have one instance of the Project open, and there are many Imported Results listed in the Imported Results window. Each of these workspaces is on a different machine and has data from an instance of the VectorCAST Project running tests in a distributed manner. As these machines progress through building and executing their test environments, more and more data is available. You want to see the new information in real-time.

To refresh a selected Imported Result, right-click and select **Refresh** from the context menu. You can multi-select two or more. Each selected Imported result is then re-imported and any new or deleted results are now visible.

To refresh all Imported Results, right-click on the Imported Results node and select **Refresh** from the

context menu. Each Imported Result is then re-imported (i.e. a fresh snapshot is taken of each workspace), so any new or deleted results are now visible.

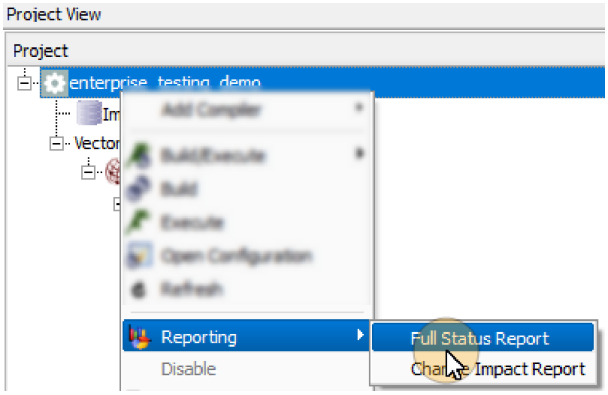
Enterprise Testing Reports

The ability to display reports of stored data result sets allows you to identify testing trends over time. Enterprise Testing provides Summary Reports to assist in the evaluation of trends.

Summary Report

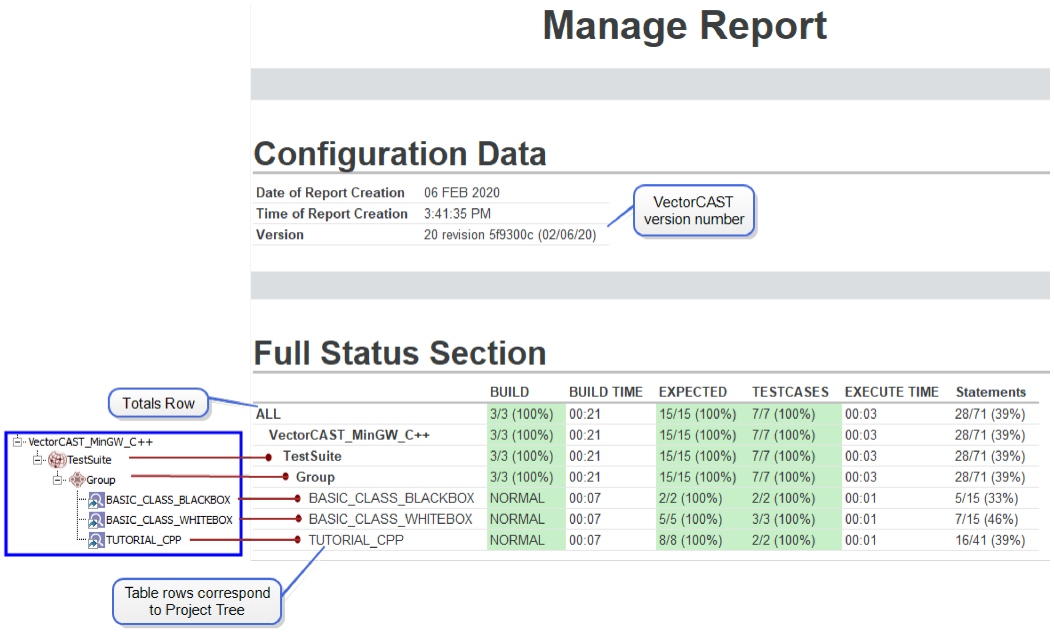
VectorCAST generates a *summary report*, titled Manage Report, which provides build, execution, and coverage results in table form. Any node may be selected for summary report generation and the steps to generate a report for any node level are the same.

To generate a Manage Report, select a node in the Project Tree and right-click. Choose the **Reporting => Full Status Report** option from the context menu.



A typical Manage Report is shown below. The report contains two sections:

- The Configuration Data section providing the time and date the report was generated and the VectorCAST version used, and
- The Full Status section providing build, execution and coverage metrics. The report includes the parents of the level generating the report.



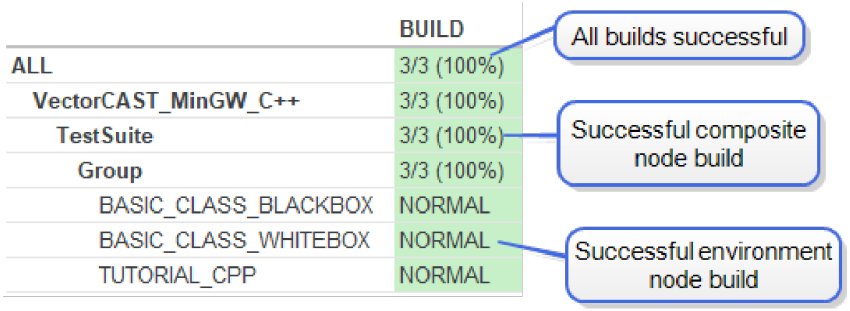
When selecting group, test suite, compiler, platform, source or project nodes, these nodes may have other nodes nested within them. They correspond to the composite node rows in the Summary report.

Each test environment has its own row in the Summary report. If the environment name begins with the characters “ENV_”, these characters are left off the name in the Summary report. Disabled environments are not displayed in the report.

Table Columns

BUILD STATUS

The BUILD column indicates the environment build status for a node.

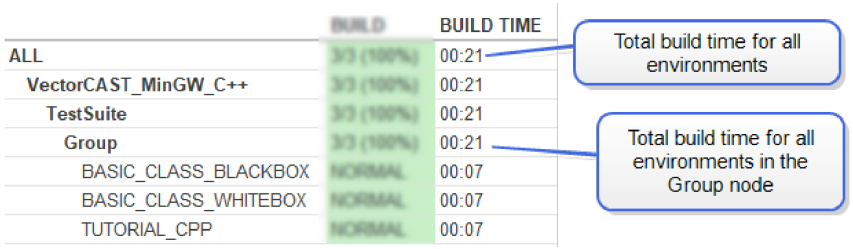


> ALL: Displays the percentage of successful environment builds by comparing the total number of builds attempted to the actual number of successful builds. 100% successful builds have the background set to green. Builds that are less than 100% successful have the background set to yellow.

- Composite Node: Displays the composite build statistics for the node's branch of the tree. 100% successful builds have the background set to green. Builds that are less than 100% successful have the background set to yellow.
- Environment Node: A successful build is indicated by the word *NORMAL* and its background color is set to green. Other indicators are NOT_LINKED, NOT_COMPILED, NOT_SET (which indicates the build failed) and a dash which means the environment was not requested to be built.

BUILD TIME

The BUILD TIME column indicates total build time for a node.

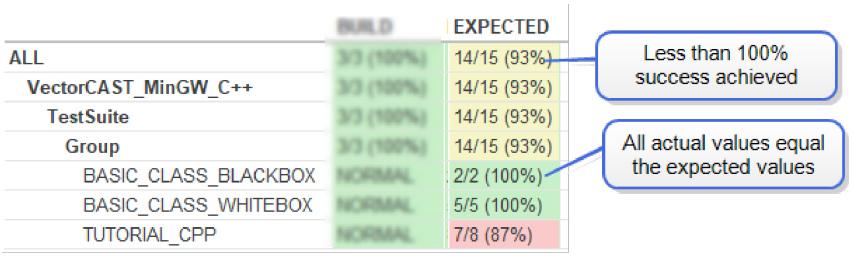


- ALL: the total amount of build time for all environments within the report.
- Composite Node: the composite amount of build time for all environments within the node's branch of the tree.
- Environment Node: the total amount of build time for each individual environment.

There is no color associated with this column.

EXPECTED VALUES

The EXPECTED column reports on the total number of expected values that have been checked and the percentage that match for a node.



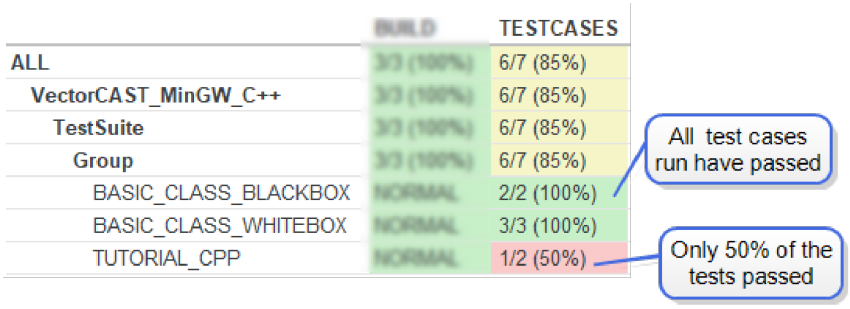
This column compares the number of comparisons that have passed (actual value equals the expected value) to the total number of comparisons made.

- ALL: If 100% success is achieved, the background set to green. Less than 100% success sets the background to yellow.
- Composite Node:If 100% success is achieved, the background set to green. Less than 100% success sets the background to yellow.

> Environment Node: If 100% success is achieved, the background set to green. Less than 100% success sets the sets the background to red.

TESTCASE STATUS

The TESTCASES column shows the number of test cases that have passed or failed for a node.



This column compares the number of passing test cases to the total number of test cases run.

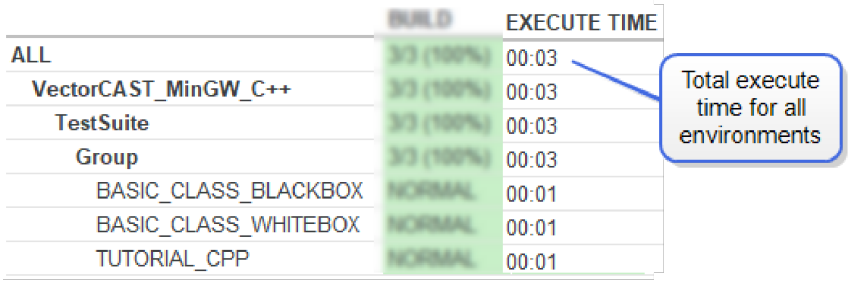
> ALL: If 100% success is achieved, the background set to green. Less than 100% success sets the background to yellow.

> Composite Node: If 100% success is achieved, the background set to green. Less than 100% success sets the background to yellow.

> Environment Node: Background color is green when all tests pass, red when 50% or less of the tests fail, and yellow for some tests pass.

TEST EXECUTION TIME

The EXECUTE TIME column displays total test execution time for a node.



> ALL: the total amount of execution time for all environments within the report.

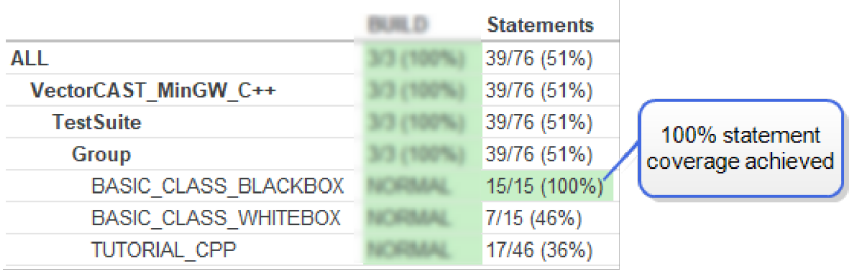
> Composite Node: the composite amount of execution time for all environments within the node's branch.

> Environment Node: the total amount of execution time for each individual environment.

There is no color associated with this column.

STATEMENT COVERAGE

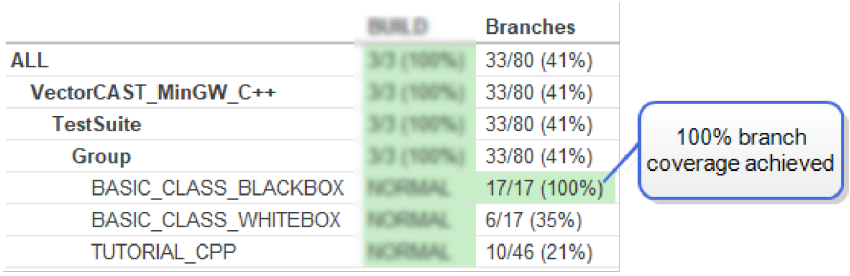
The STATEMENTS column reports on Statement Coverage for a node.



For all row types, the number to the left of the slash indicates the amount of statement coverage achieved and the number to the right is the total number of statements to be covered. The percentage of coverage is shown in parentheses. Background color is green when coverage is fully achieved.

BRANCH COVERAGE

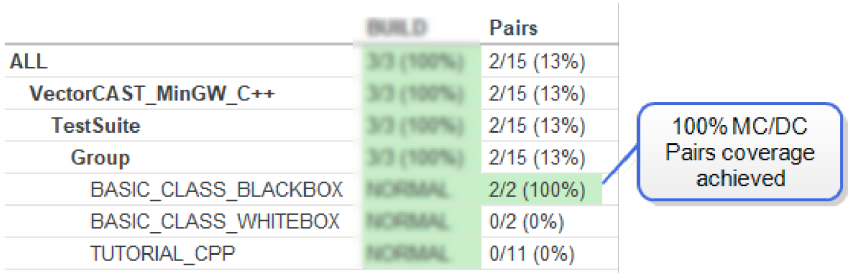
The BRANCHES column reports on Branch Coverage for a node.



The number to the left of the slash indicates the amount of branch coverage achieved and the number to the right is the total number of branches to be covered. If MC/DC or DO-178B coverage is enabled on the test environment, *BRANCHES* represents the number of MC/DC branches. The percentage of coverage is shown in parentheses. Background color is green when coverage is fully achieved.

MC/DC PAIR COVERAGE

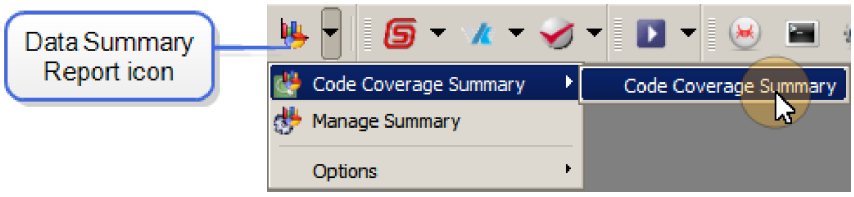
The PAIRS column reports on MC/DC Equivalence Pair Coverage for a node.



The number to the left of the slash indicates the amount of MC/DC Pairs coverage achieved and the number to the right is the total number of MC/DC Pairs to be covered. The percentage of coverage is shown in parentheses. Background color is green when coverage is fully achieved.

Code Coverage Summary

The Code Coverage Summary table allows the user to easily view coverage information for all the environments of a VectorCAST project. To open the summary from the Toolbar, select **Code Coverage Summary** from the Data Summary Report icon drop-down menu.

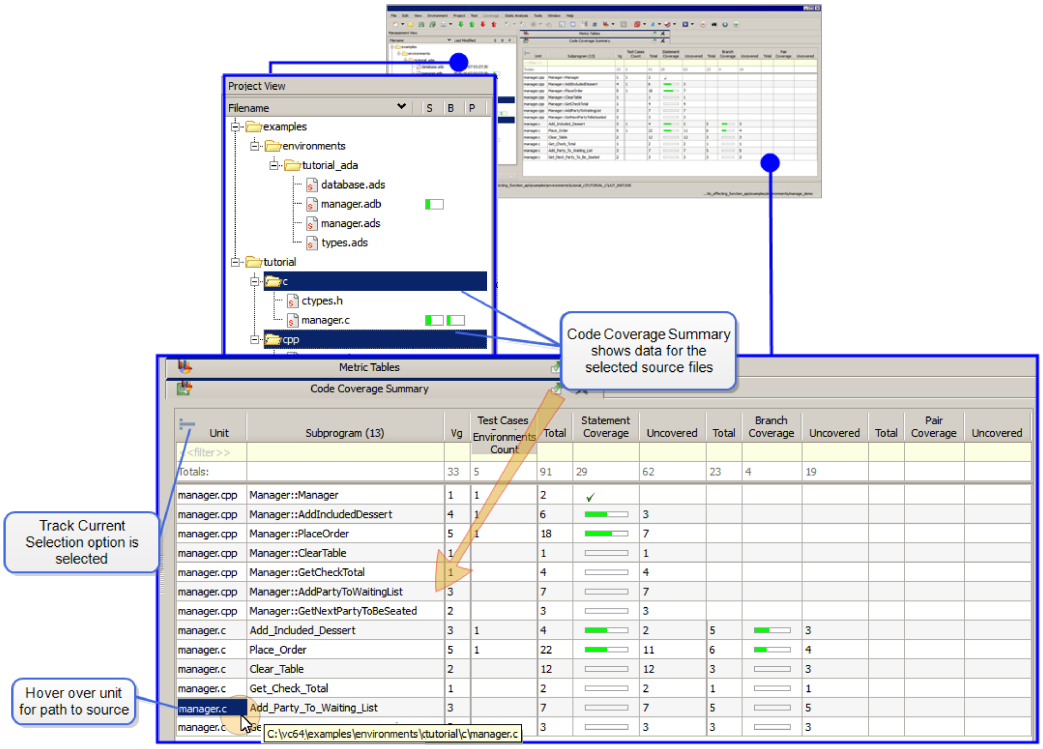


Alternatively, from the Menu Bar, select **Coverage => Code Coverage Summary => Code Coverage Summary**.

Viewing Code Coverage

On initial open, the Code Coverage Summary table displays coverage data for all environments in the VectorCAST project. Use the Files tab to select data of interest to be displayed in the summary. By default, the contents of the Code Coverage Summary reflect the source files selected in the Files tab. A tracking icon is displayed at the top of the Unit column indicating that the Code Coverage Summary is currently tracking selected source files.

To override tracking of selected source files, 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 Files tab.

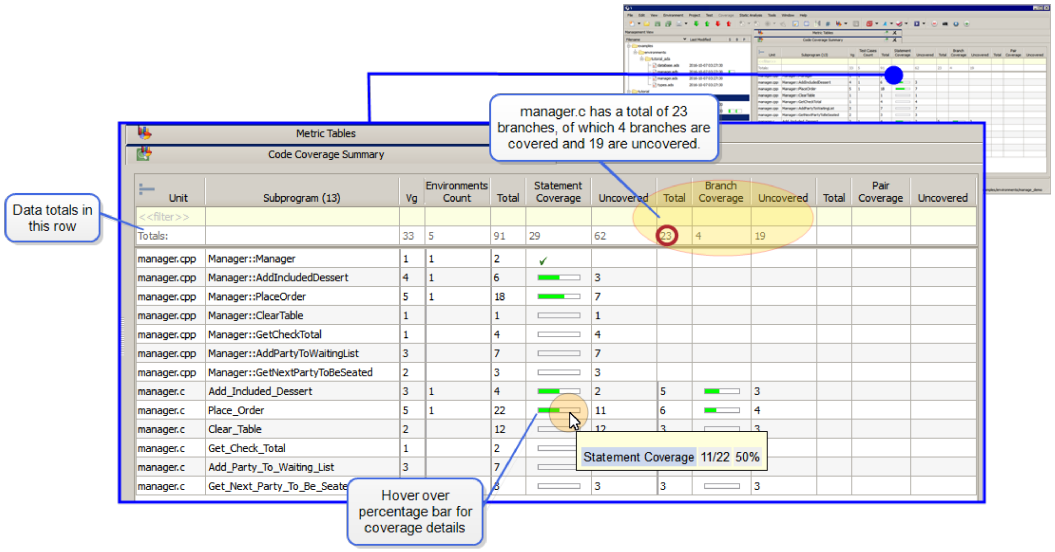


Unit	Subprogram (13)	Vg	Test Cases Environments Count	Total	Statement Coverage	Uncovered	Total	Branch Coverage	Uncovered	Total	Pair Coverage	Uncovered
<<filter>>												
Totals:		33	5	91	29	62	23	4	19			
manager.cpp	Manager::Manager	1	1	2	✓							
manager.cpp	Manager::AddIncludedDessert	4	1	6		3						
manager.cpp	Manager::PlaceOrder	5	1	18		7						
manager.cpp	Manager::ClearTable	1		1		1						
manager.cpp	Manager::GetCheckTotal	1		4		4						
manager.cpp	Manager::AddPartyToWaitingList	3		7		7						
manager.cpp	Manager::GetNextPartyToBeSeated	2		3		3						
manager.c	Add_Included_Dessert	3	1	4		2	5		3			
manager.c	Place_Order	5	1	22		11	6		4			
manager.c	Clear_Table	2		12		12	3		3			
manager.c	Get_Check_Total	1		2		2	1		1			
manager.c	Add_Party_To_Waiting_List	3		7		7	5		5			
manager.c	Ge			3		3	3		3			

The Code Coverage Summary table is dynamic. When the **Track Current Selection** option is enabled, as units are selected and deselected in the Files tab, 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 Environments Count (showing the number of environments contributing coverage to a subprogram), 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.



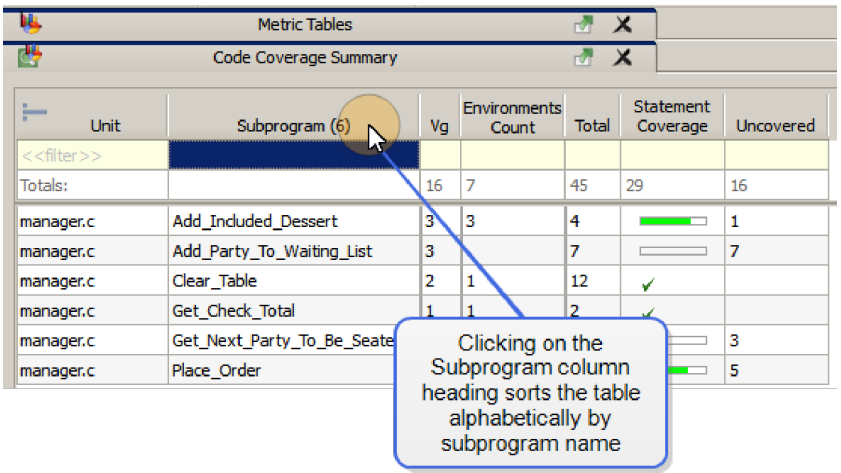
The Totals row at the top of the table displays the totals for each data column. In the example above, note that in the unit manager.c we have a total of 23 branches, of which 4 branches are covered and 19 are uncovered.

The Summary table updates whenever coverage data is updated. For example, the table refreshes when coverage is initialized, or following test execution.

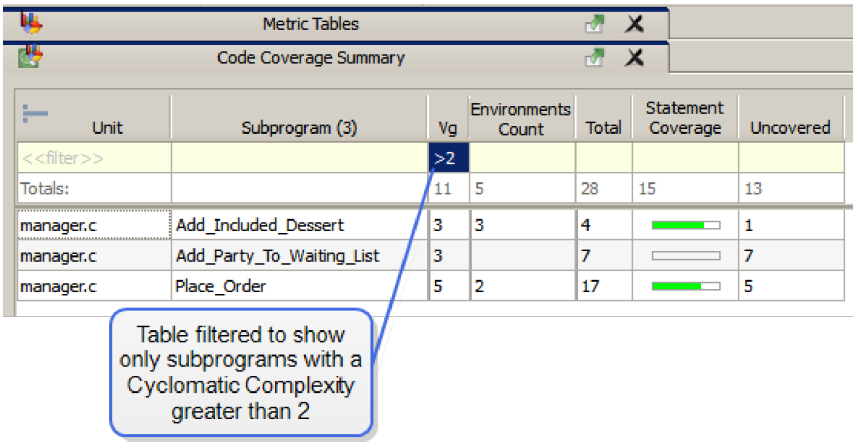
Double-clicking on a line in the Code Coverage Summary opens the corresponding UUT in the Coverage Viewer.

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 data for test cases with Cyclomatic Complexity greater than 2.



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. Selecting **File => Save** from the Menu Bar enables you to save the contents of the Code Coverage Summary currently open in the MDI window. The Save As dialog opens, allowing you to give the file a name and choose the output format.

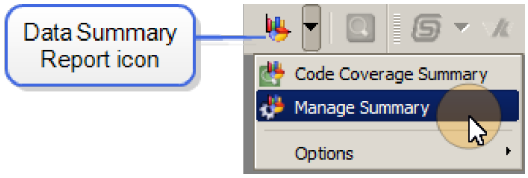
The **Save** command may also be invoked by clicking on the **Save** button on the Toolbar  .

Printing the contents of the Code Coverage Summary is supported. Select **File => Print** From the Menu Bar or click on the **Print** button on the toolbar  to print the contents of the Code Coverage Summary currently open in the MDI window.

Manage Data Summary

The Manage Data Summary table allows the user to easily view environment execution results and metrics for all the environments of a VectorCAST project. To open the summary from the Toolbar,

select **Manage Summary** from the Data Summary Report icon drop-down menu.

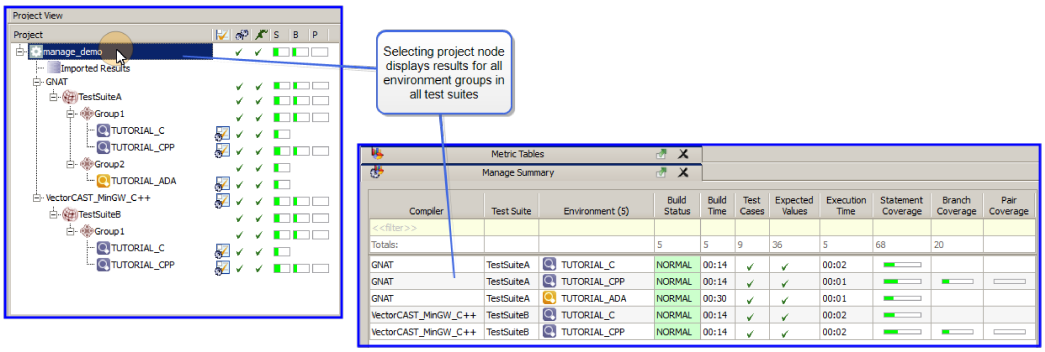


Alternatively, from the Menu Bar, select **Project => Manage Summary**.

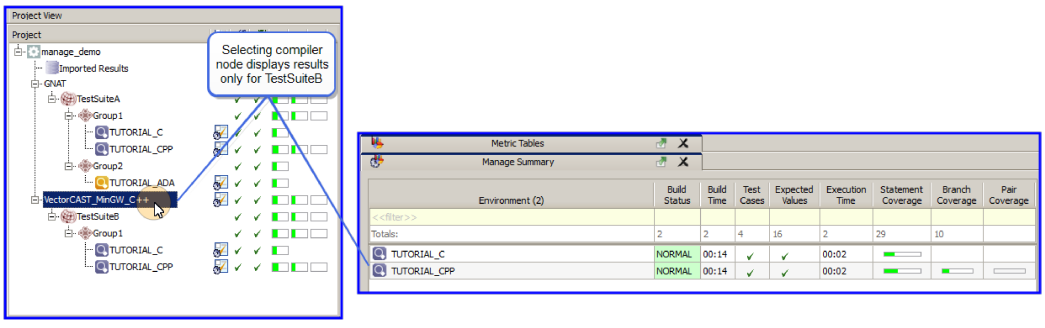
Viewing Manage Data Summary

On initial open, The Manage Data Summary displays all environment execution results and metrics. By default, the **Track Current Selection** option is enabled and the Manage Data Summary displays environment data and metrics based on the current selection in the Project Tree. The Manage Data Summary is dynamic and as units are selected and deselected in the Project Tree, the Manage Summary table updates in real time reflecting the selections.

In the example below, the Project node, manage_demo, is selected and the Manage Summary displays data for all the test suites in the project.

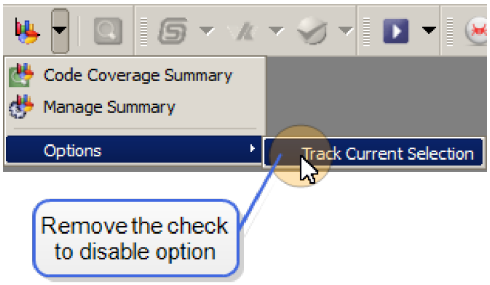


In the following example, the MinGW_C++ compiler node is selected in the Project Tree. Note that the Manage Summary displays data only for TestSuiteB.

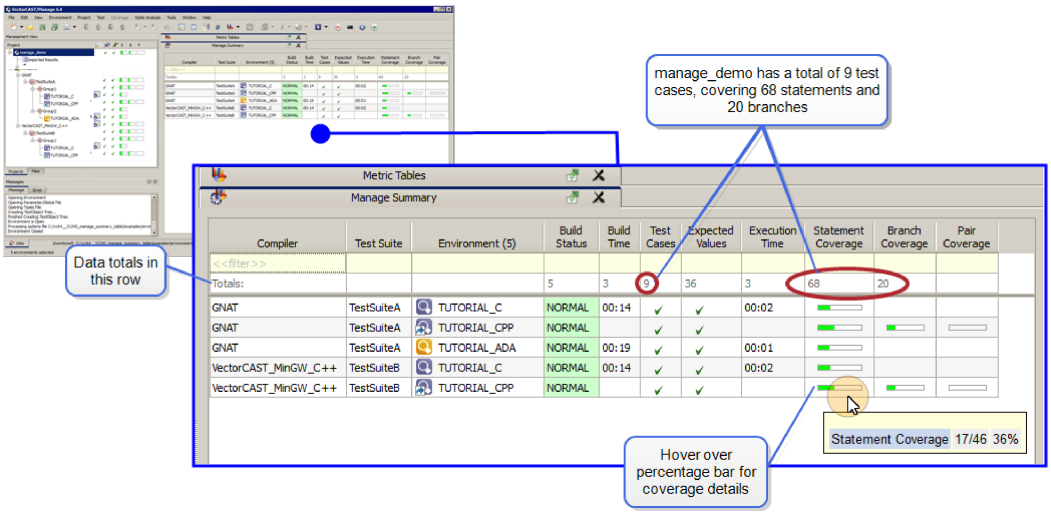


To disable the **Track Current Selection** option, from the Data Summary Report icon drop-down menu, select **Options =>Track Current Selection** and uncheck the menu item. All items in the

project are tracked.



The data displayed in the Manage Data Summary includes the Environment name, Build Status, Build Time, number of Test Cases, number of Expected Values, Execution Time and achieved coverage for each coverage type. When the display is based on a selected Project node, additional columns for Compiler name and Test Suite name are included.

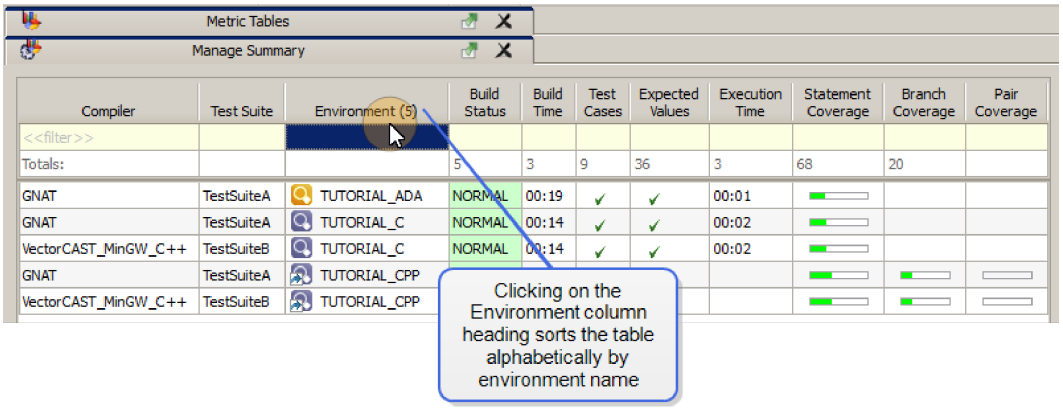


The Totals row at the top of the table displays the totals for each data column. In the example above, note that in the manage_demo project we have a total of 9 test cases, covering 68 statements and 20 branches.

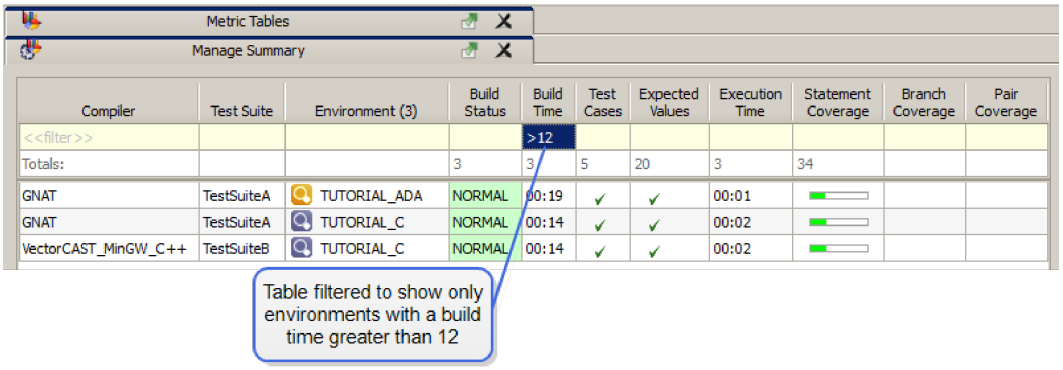
Double-clicking on a line in the Manage Summary opens the corresponding environment in the Environment View.

Sorting and Filtering Manage 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.



Manage data can be accessed all the way down to the individual environment 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 environments with Build Times greater than 12 seconds.



Filtering supports the following symbols: <, >, =. For example, the summary can be filtered to show only environments with a Build Time greater than 12 seconds. Other examples of filtering inputs are:

10 - lists environments matching the specific value of "10" in the selected column

>50 - lists environments greater than the value of "50" in the selected column

<90 - lists environments less than the value of "90" in the selected column

=100 - lists environments matching the specific value of "100" in the selected column

1:15 - lists environments matching the specific value of "1:15" in the selected Build Status or Execution Time columns. Acceptable formats are hh:mm:ss, mm:ss, and ss.

< - lists environments with empty values in the selected column

> - lists environments with non-empty values in the selected column

= - lists environments with non-empty values in the selected column

Saving and Printing Manage Data Summary

Saving the Manage Data Summary in .html or .csv format is supported. Selecting **File => Save** from

the Menu Bar enables you to save the contents of the Code Coverage Summary currently open in the MDI window. The Save As dialog opens, allowing you to give the file a name and choose the output format.

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

Printing the contents of the Manage Data Summary is supported. Select **File => Print** From the Menu Bar or click on the **Print** button on the toolbar  to print the contents of the Code Coverage Summary currently open in the MDI window.

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.

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 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_section":[<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_section - 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_section": [<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_section` - 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

Follow the following steps to configure VectorCAST to use the Custom Templates in the Configuration Directory.

From the GUI:

In a VectorCAST Project, right-click on the project node and select **Open Configuration** from the menu.

Expand the Report node in the Configuration Editor, and enter the path to the Configuration Directory in the "Reports custom configuration" field. VectorCAST supports the use of an environment variable, for example, **$(PROJ_DIR)\vcast\custom_rep**. Note that a full build on the environments is needed to force this setting to be applied to the individual environments.

From the command line:

```
clicast 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
```

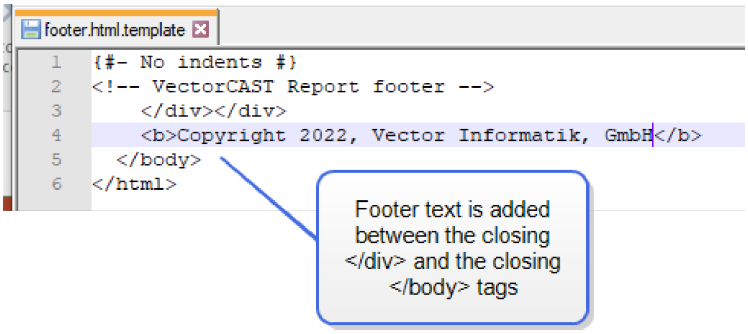
2. Next, configure VectorCAST to use the new configuration directory, **`CustomDirectory`**.

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

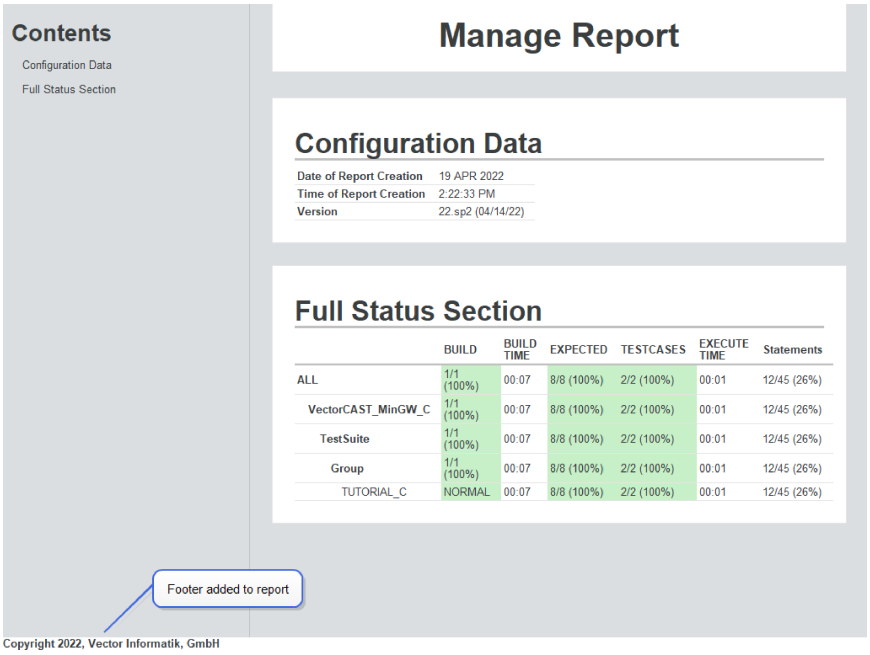
To customize reports, locate the report template in the configuration directory, remove the **`.orig`** extension and modify the file as needed.

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.

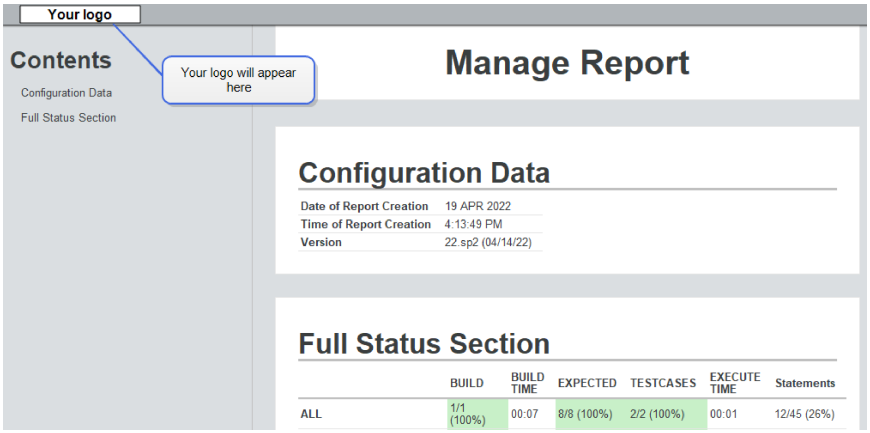


Add Your Logo to Reports

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.

Save changes and generate a report. Your logo is displayed in the title bar of the report.



Customizing Report Style

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`**.

The styles contained in this style sheet are embedded into the **`<style>`** section of the HTML VectorCAST reports when they are generated.

Creating a Custom Style Sheet

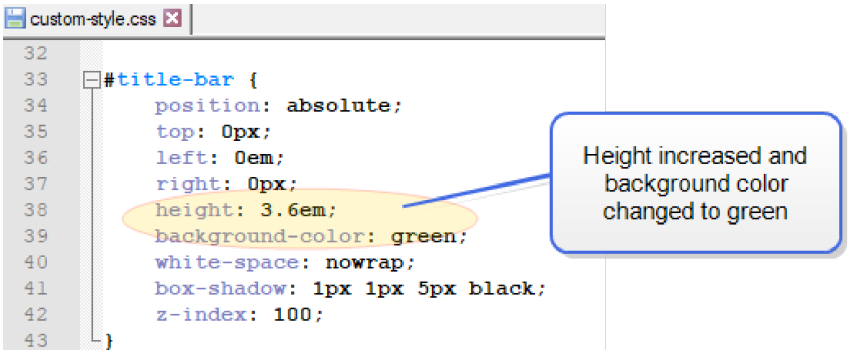
While you can modify the default stylesheet, best practice is to make a copy of the default stylesheet, store it in your configuration directory (you can rename the stylesheet), and customize that file as needed.

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.



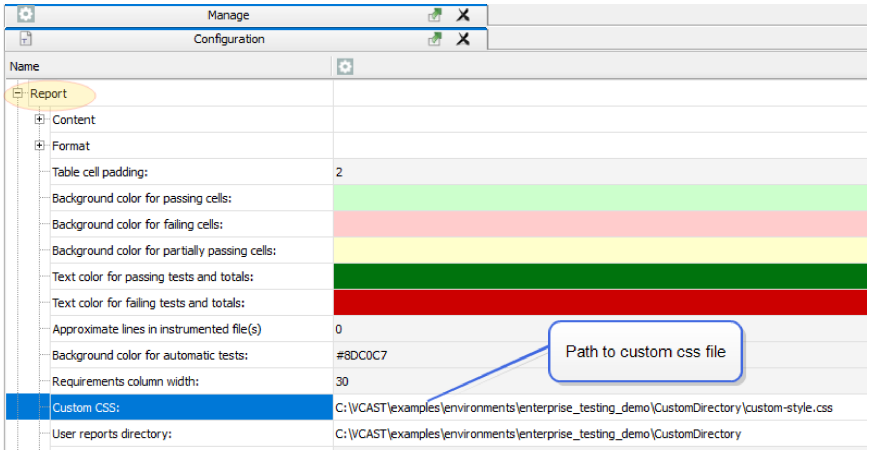
Next we add our custom style sheet to VectorCAST.

Adding a Custom Style Sheet

From the GUI:

In a VectorCAST Project, right-click on the project node and select **Open Configuration** from the menu.

Expand the Report node in the Configuration Editor, and enter the path to the Configuration Directory in the "Custom CSS" field. VectorCAST supports the use of an environment variable, for example, **`$(PROJ_DIR)\vcast\custom_css`**. Note that a full build on the environments is needed to force this setting to be applied to the individual environments.

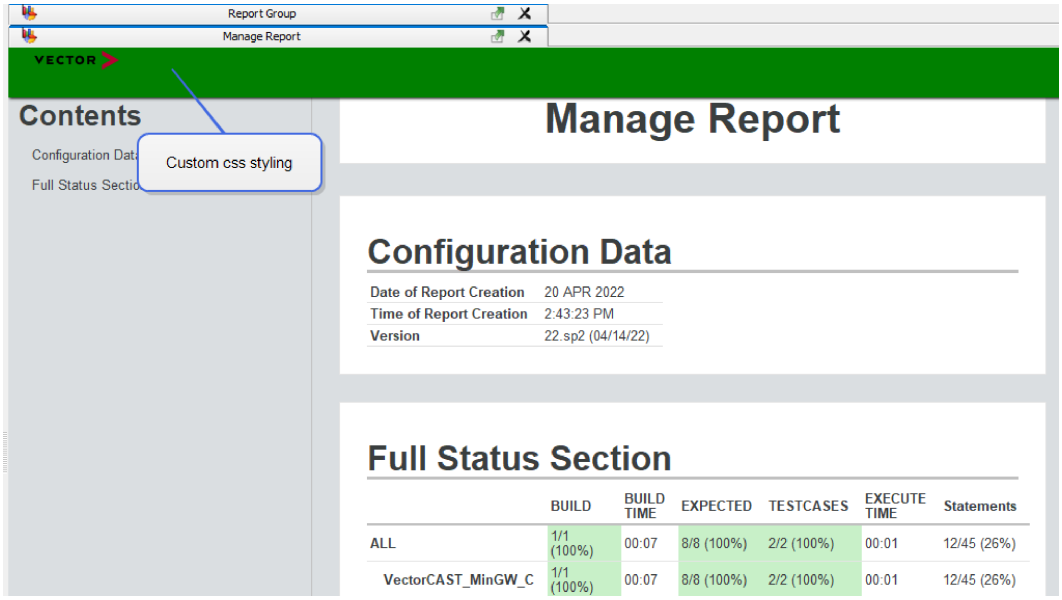


From the command line:

```
clicast 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 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:

In a VectorCAST project, enter the following from the command line:

```
manage -p <project> -c <compiler> -e <environment> --clicast-args report
user <report_name>
```

For example:

manage -p <project> -c <compiler> -e <environment> --clicast-args report user my_report

This command will generate MANAGER_user_report.html in the environment directory for the selected environment. The compiler and environments oprions can be omitted to generate the report for all environments, which appear in each of the subdirectories in the build directory. You can also specifiy a file name on the command line.

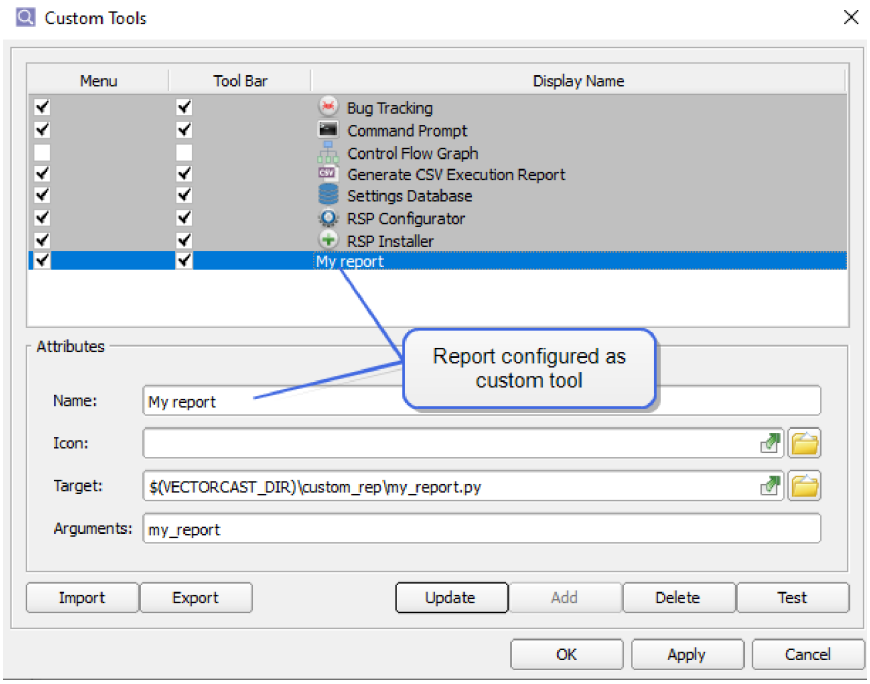
From the GUI:

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

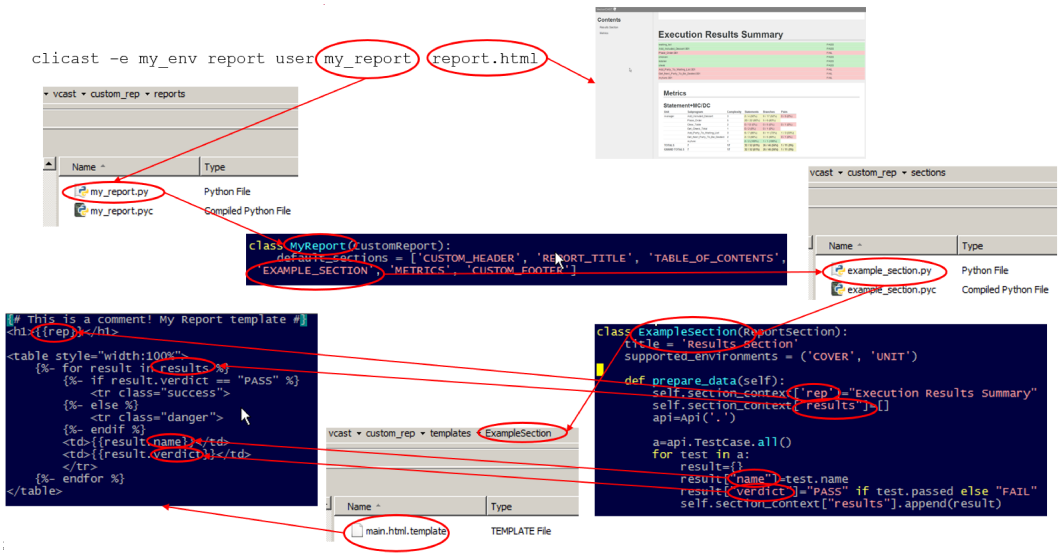
Create a helper script which will do the following:

- 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.



Change-Based Testing

Change-Based Testing (CBT) 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.

To perform an incremental build, right-click on a node in the Project Tree and select **Build/Execute => Incremental** from the context menu. Upon completion, the Manage Incremental Rebuild Report is produced and displayed in the MDI window.

Change-Based Testing will:

- Build the environment if unbuilt
- Build incrementally for function scope changes, or
- Build full for file scope changes, and
- Execute any new tests, or tests affected by the source change.

Performing **Build/Execute => Incremental** on a migrated Cover environment will:

- Create the environment if it does not exist in the workspace,
- Update the sources from the base directories,
- Do an incremental instrument on the source files and remove any affected results/testcases, and
- Execute any removed tests.

Performing Build/Execute Incremental on migrated environments brings test scripts up-to-date prior to executing. This is especially useful in projects having migrated environments in multiple configurations.

For example, a single unit test environment might be located under both a Host compiler node and a Target compiler node. Say the environment is opened under the Host compiler node and modifications to the tests are made and then saved. Later, when the environment is selected under the Target compiler node and an Incremental Build/Execute performed, VectorCAST will first update the test script to match the most recent one in the <project>/environment/ENV directory before executing. By performing Build/Execute Incremental on environments, you are assured that each instance of the environment is using the same, updated test script.

After an incremental rebuild the Manage Incremental Rebuild Report is produced to summarize the results. The report provides the status from the Incremental Rebuild, the number of tests preserved for that environment, and the number of tests that required re-execution.

Manage Incremental Rebuild

Environments Affected

Environment	Rebuild Status	Unaffected Tests	Affected Tests	Total Tests
VectorCAST_MinGW_C++ / TestSuite / BASIC_CLASS_BLACKBOX	Full Rebuild Succeeded	0	2	2
VectorCAST_MinGW_C++ / TestSuite / BASIC_CLASS_WHITEBOX	Full Rebuild Succeeded	0	3	3
VectorCAST_MinGW_C++ / TestSuite / TUTORIAL_CPP	Rebuild Unnecessary	2	0	2
Totals	**2 / 2 (100%)**	**2**	**5**	**7**

VectorCAST/QA - System Test Automation

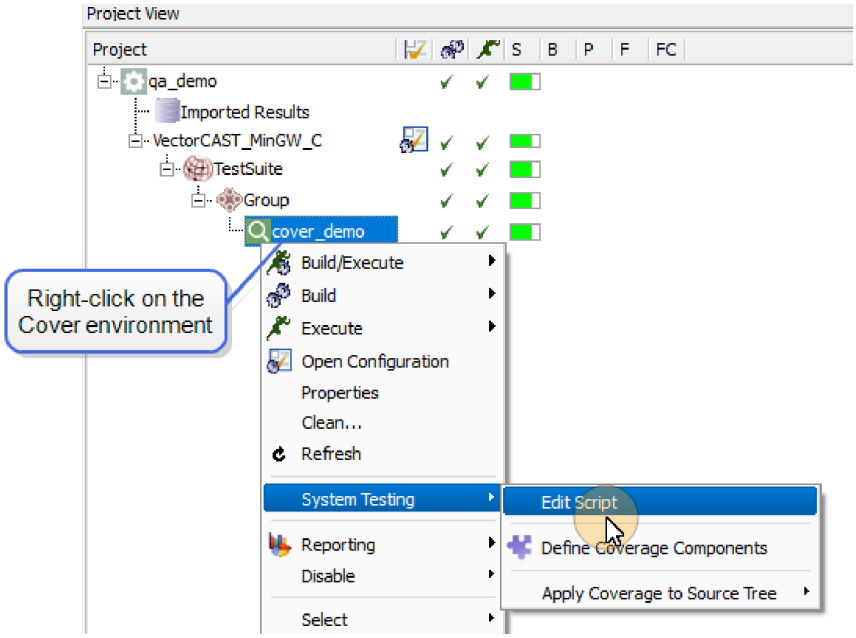
VectorCAST/QA provides a single point of control and reporting for all test activities. By combining VectorCAST's Code Coverage, Change-Based Testing (CBT) and System Test functionalities into a single tool, VectorCAST/QA automates system testing and enables faster release cycles.

VectorCAST/QA uses a simple user-defined Python script to define the application build commands, list of test cases and test execution commands for an application. VectorCAST/Cover supports the maintenance of the Cover environment (for example, code changes) and automatically captures test execution coverage data. CBT automatically identifies the minimum tests that must be run for each code change and automatically re-runs only those tests. For more information on CBT, see "Change-Based Testing " on page 172.

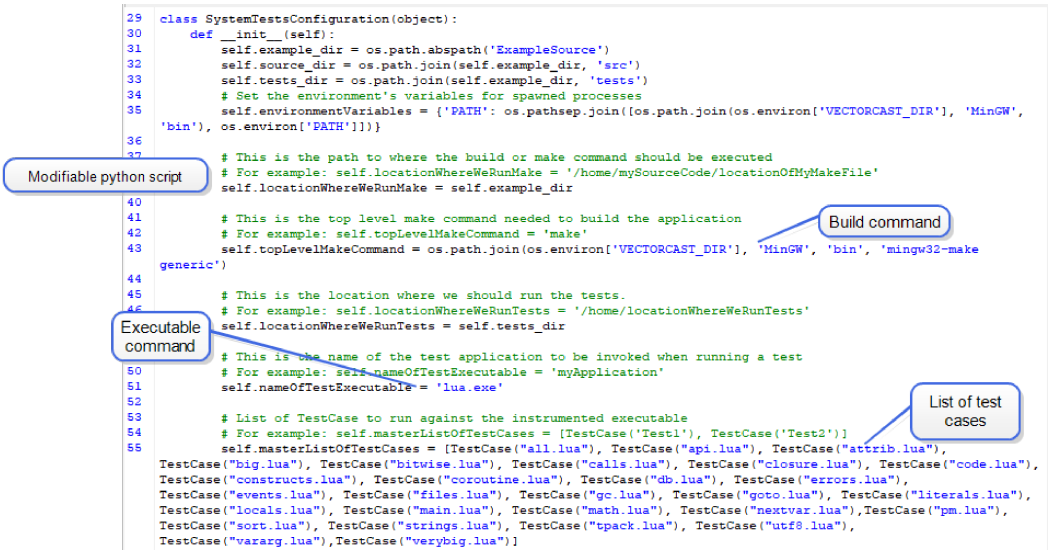
The Python Configuration Script

The Python Configuration Script contains functions that are invoked by VectorCAST to provide custom build and execute commands. Each time a new system test environment is created within a project, its own **`<environment name>_system_tests.py`** script is created, which defines its own build and execution commands. The configuration script is fully customizable and allows the user to define the application build commands, list of test cases, and test execution commands for an application.

To open and edit the Configuration Script, right-click on a Cover Environment node and select **System Testing => Edit Script** from the context menu.



The Python Configuration Script opens in the Script Editor. Modify the script as required and save changes.

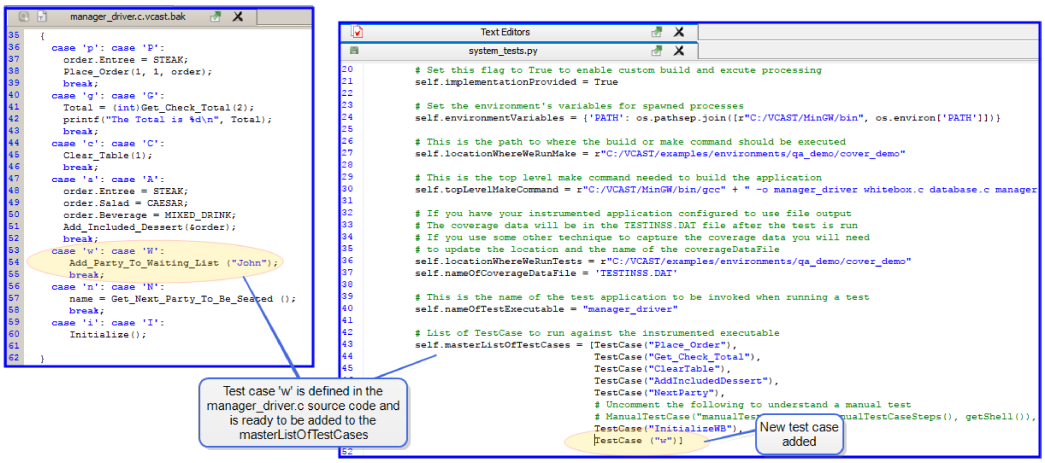



```
class SystemTestsConfiguration(object):
    def __init__(self):
        self.example_dir = os.path.abspath('ExampleSource')
        self.source_dir = os.path.join(self.example_dir, 'src')
        self.tests_dir = os.path.join(self.example_dir, 'tests')
        # Set the environment's variables for spawned processes
        self.environmentVariables = {'PATH': os.pathsep.join([os.path.join(os.environ['VECTORCAST_DIR'], 'MinGW',
'bin'), os.environ['PATH']])}

        # This is the path to where the build or make command should be executed
        # For example: self.locationWhereWeRunMake = '/home/mySourceCode/locationOfMyMakeFile'
        self.locationWhereWeRunMake = self.example_dir

        # This is the top level make command needed to build the application
        # For example: self.topLevelMakeCommand = 'make'
        self.topLevelMakeCommand = os.path.join(os.environ['VECTORCAST_DIR'], 'MinGW', 'bin', 'mingw32-make
generic')

        # This is the location where we should run the tests.
        # For example: self.locationWhereWeRunTests = '/home/locationWhereWeRunTests'
        self.locationWhereWeRunTests = self.tests_dir

        # This is the name of the test application to be invoked when running a test
        # For example: self.nameOfTestExecutable = 'myApplication'
        self.nameOfTestExecutable = 'lua.exe'

        # List of TestCase to run against the instrumented executable
        # For example: self.masterListOfTestCases = [TestCase('Test1'), TestCase('Test2')]
        self.masterListOfTestCases = [TestCase("all.lua"), TestCase("api.lua"), TestCase("attrib.lua"),
TestCase("big.lua"), TestCase("bitwise.lua"), TestCase("calls.lua"), TestCase("closure.lua"), TestCase("code.lua"),
TestCase("constructs.lua"), TestCase("coroutine.lua"), TestCase("db.lua"), TestCase("errors.lua"),
TestCase("events.lua"), TestCase("files.lua"), TestCase("gc.lua"), TestCase("goto.lua"), TestCase("literals.lua"),
TestCase("locals.lua"), TestCase("main.lua"), TestCase("math.lua"), TestCase("nextvar.lua"),TestCase("pm.lua"),
TestCase("sort.lua"), TestCase("strings.lua"), TestCase("tpack.lua"), TestCase("utf8.lua"),
TestCase("vararg.lua"),TestCase("verybig.lua")]
```

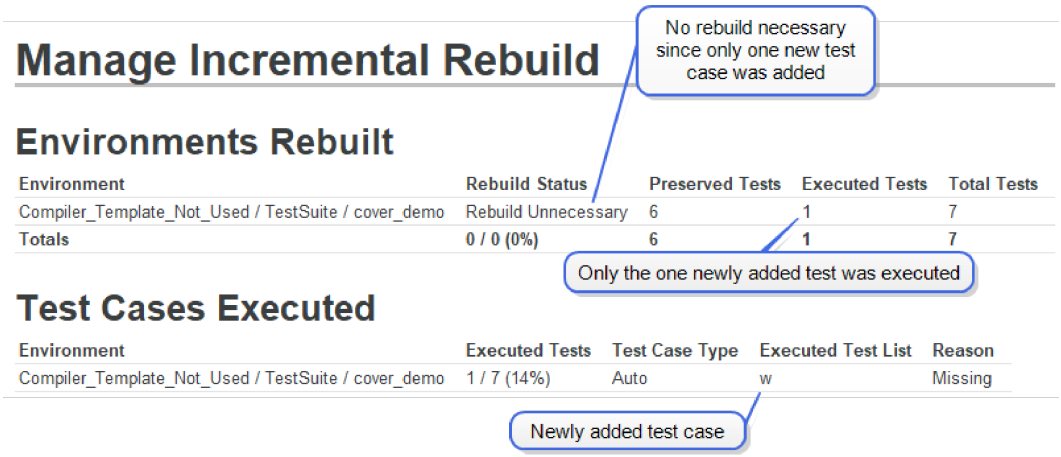
Add a Test Case Using Python Configuration Script

To add a new test case that is already defined in the source code, first open the Python Configuration Script by right-clicking on the environment node and selecting **System Testing => Edit Script** from the context menu.

The configuration script opens in the Script Editor. Scroll down to the list of system test case names and add the new test name to the end of the Python list. In our example, we add the test case "w".



Save the script changes. Right-click on the environment node and select **Execute => Full** from the context menu. VectorCAST will execute only the newly added test.



Add a Manual Test

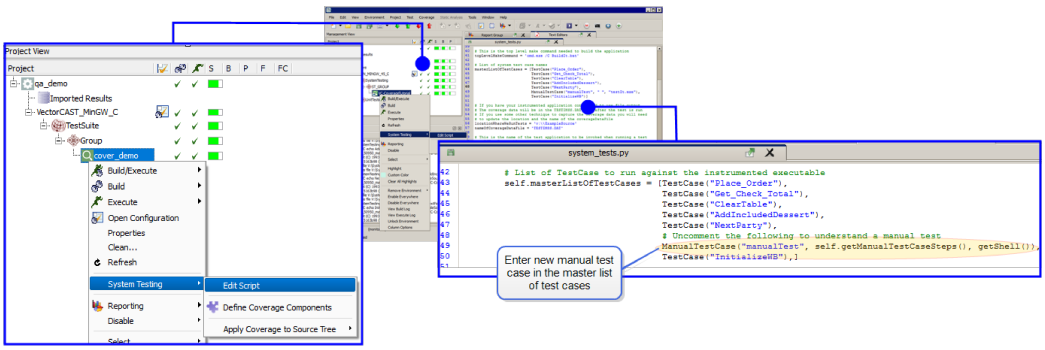
VectorCAST supports manual tests in addition to automated test cases. Adding the test case class **`ManualTestCase`** to the Python Configuration Script invokes a dialog allowing the user to implement a manual test. Manual test results are added to the Cover environment and included in the total number of tests.

A manual test case is implemented by right-clicking on the environment node and selecting **System Testing => Edit Script** from the context menu.

The Python Configuration Script opens in the Script Editor. Scroll down to the list of system test case names and add the manual test entry to the Python list using the following syntax:

```
ManualTestCase('<TestName>','[<TestSteps>]','<NameOfTestExecutable>')

where the parameters required for the ManualTestCase class are <TestName>
and <NameOfTestExecutable>. The parameter <TestSteps> is optional.
```

For example:
`ManualTestCase ("manualTest", self.getManualTestCaseSteps(), getShell())`



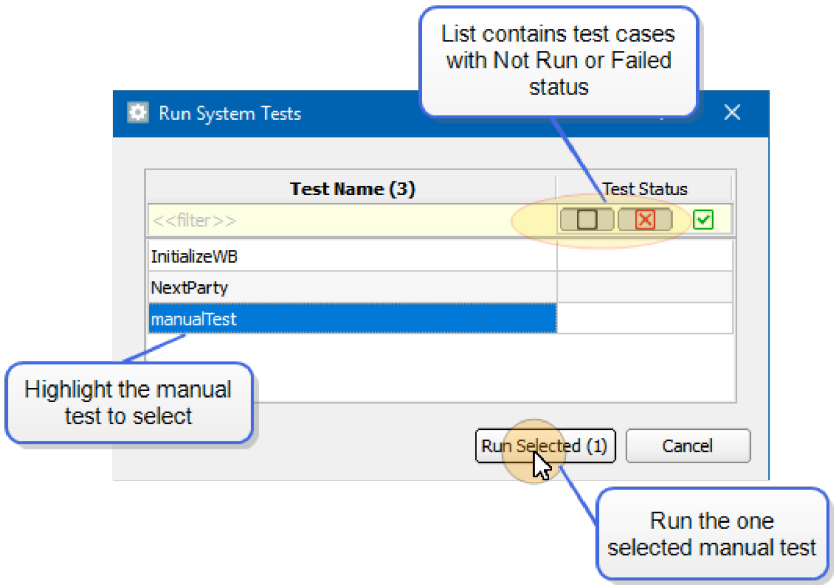
On Windows platforms, test case names are case insensitive. To avoid unexpected results, be sure to give each test case a unique name that does not rely on case to differentiate. For example, use

`manualTest1, manualTest2`, instead of **`ManualTest, manualTest`**.

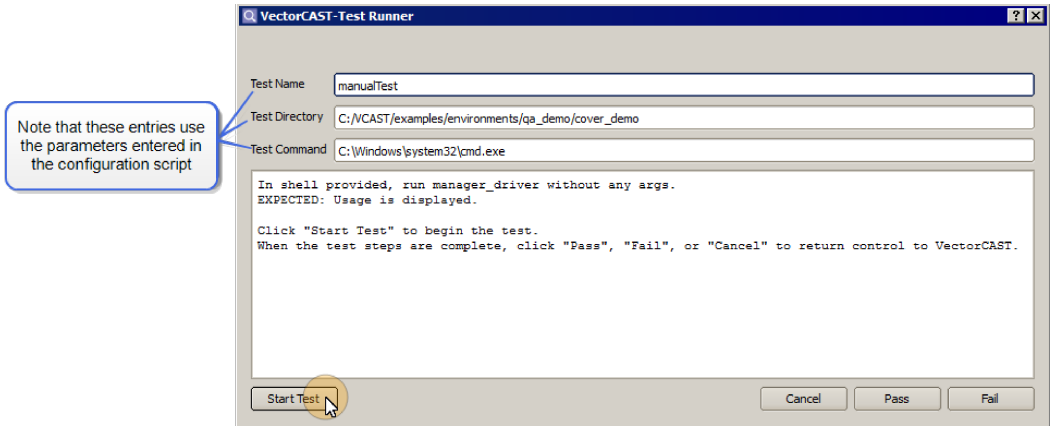
Save the script changes. To run a manual test case, right-click on the environment node and select **Execute => Interactive** from the context menu. The Run System Tests dialog opens.

You can filter the list by entering the Test Name. You can also refine the list by enabling the Test Status (**Not Run**, **Failed**, or **Passed**). The selection of multiple statuses is supported. By default, the Failed and Not Run test statuses are enabled.

For our example, we will run only the **`manualTest`** test case. Our test case has not been run, so it appears in the listing because the Not Run Test Status is enabled. To run the test, highlight the Test Name, **`manualTest`**, and select the **Run Selected** button.

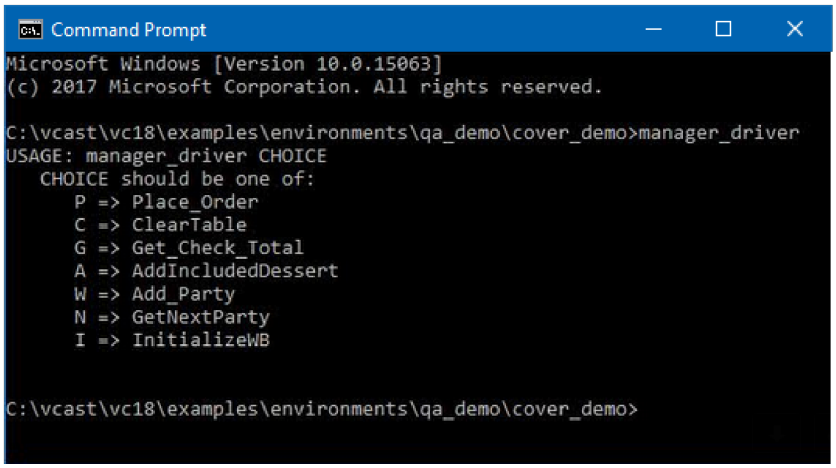


VectorCAST opens the Test Runner dialog for the manual test.



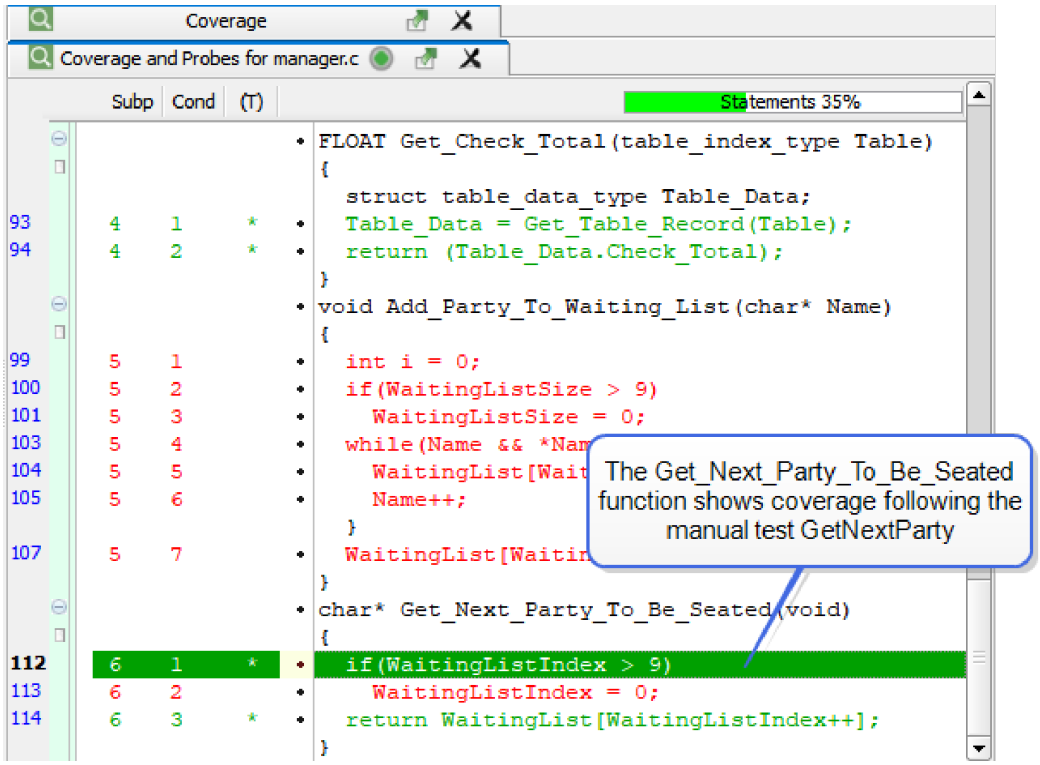
The fields displayed in the Test Runner dialog match the parameters passed to the ManualTestCase class. Click the **Start Test** button to run the test executable application.

A DOS command window opens displaying an input prompt. Select the test to be run. In our example, we enter "**N**" to run the **GetNextParty** test.



When the manual test is completed, record the results by clicking the appropriate **Pass** or **Fail** button on the Test Runner dialog. The dialog closes and the test results are added to the Cover environment. Clicking the **Cancel** button closes the dialog and does not add the test results to the total.

In our example we selected the **Pass** button at the conclusion of our manual test. Open the Coverage Viewer for **manager.c** and note that coverage is now shown for the **Get_Next_Party_To_Be_Seated** function.



Incremental Build and Execute

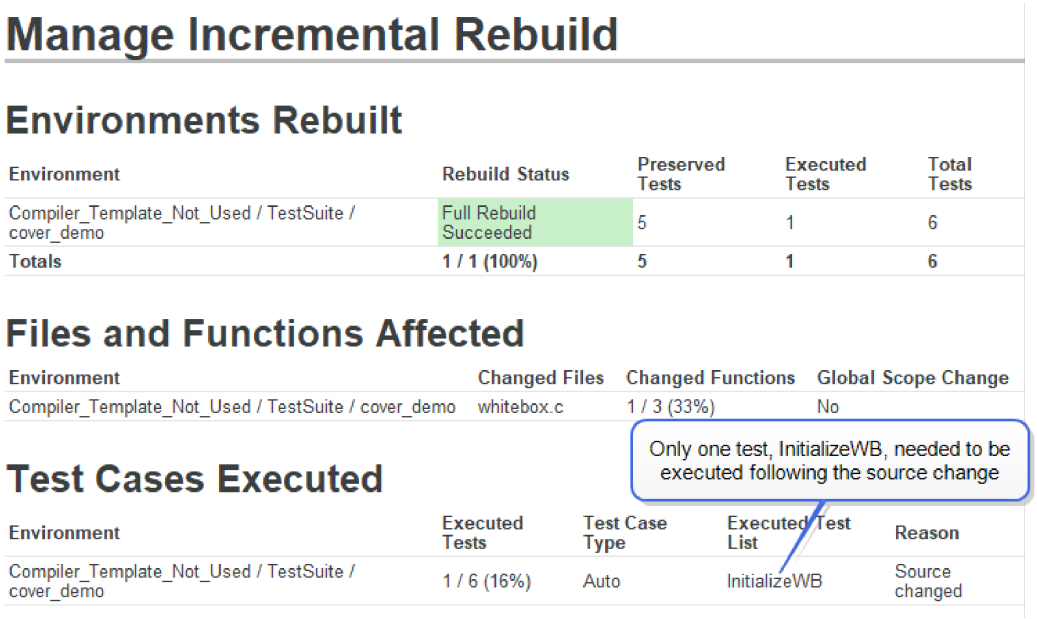
Once the Python configuration has been set up, the incremental build and execute functionality performs only the work that needs to be done, based on the current state of the project.

For example, if a change is made to the project's source files, performing an incremental build and execute will perform only the work that needs to be done based on the source change.

Run Incremental Build/Execute Automatically

To automatically run an incremental Build/Execute, right-click on the Environment node and select **Build/Execute => Incremental** from the context menu. For unit environments and migrated Cover environments, VectorCAST will fully build or rebuild environments and execute all affected or missing tests.

For example, the example report below shows that one test (InitializeWB) was affected due to a source file change in **`whitebox.c`**.



Manage Incremental Rebuild

Environments Rebuilt

Environment	Rebuild Status	Preserved Tests	Executed Tests	Total Tests
Compiler_Template_Not_Used / TestSuite / cover_demo	Full Rebuild Succeeded	5	1	6
Totals	1 / 1 (100%)	5	1	6

Files and Functions Affected

Environment	Changed Files	Changed Functions	Global Scope Change
Compiler_Template_Not_Used / TestSuite / cover_demo	whitebox.c	1 / 3 (33%)	No

Test Cases Executed

Environment	Executed Tests	Test Case Type	Executed Test List	Reason
Compiler_Template_Not_Used / TestSuite / cover_demo	1 / 6 (16%)	Auto	InitializeWB	Source changed

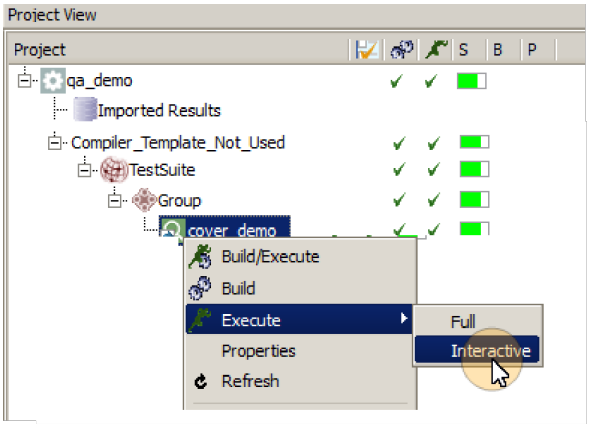
Only one test, InitializeWB, needed to be executed following the source change

The Manage Incremental Rebuild Report shows that only one file was instrumented and only one test, InitializeWB, was run as a result of the source code change.

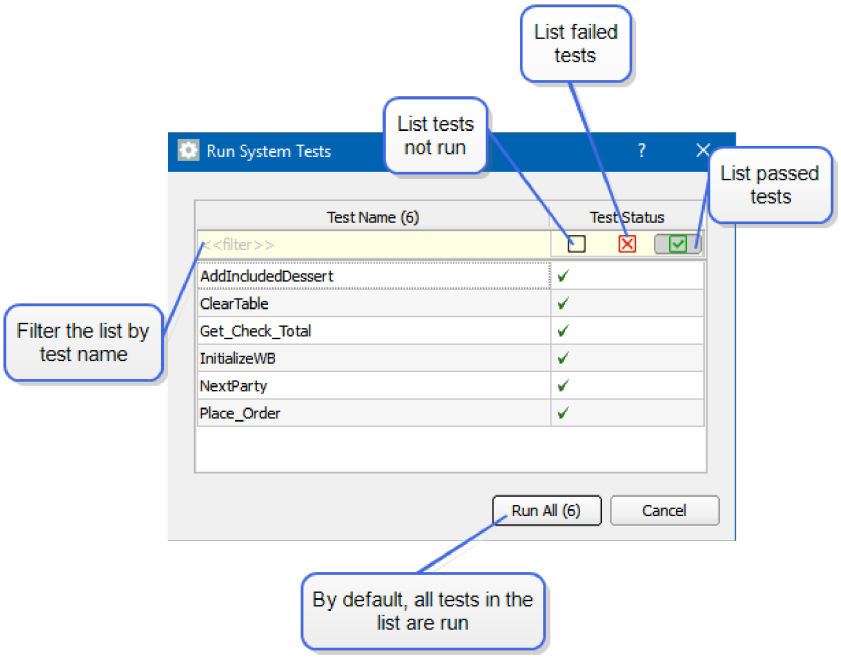
Interactive Execution of Tests

VectorCAST provides the ability to manually select a single test or a set of tests for execution in a System Test environment. Tests can be selected by Test Type and Test Name, allowing the user to run all or any subset of tests for the environment.

To select tests, right-click on a System Test environment in the Project Tree and select **Execute => Interactive** from the context menu.

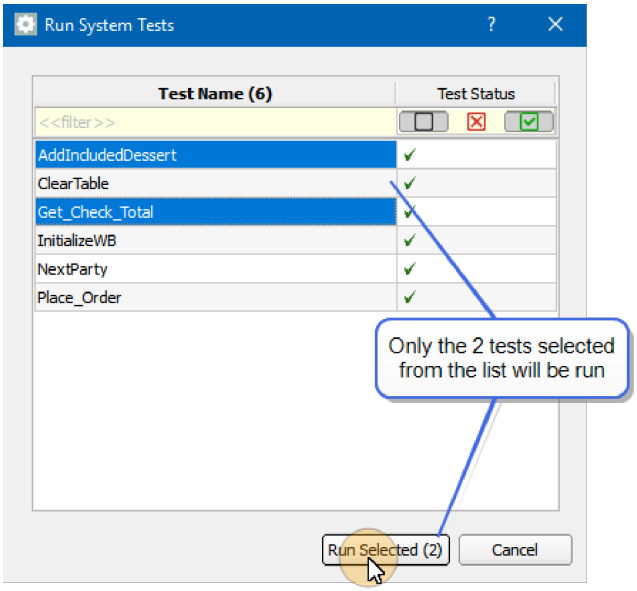


The Run System Tests dialog opens. You can filter the list by Test Name and Test Status (Not Run, Failed, or Passed), allowing you to quickly select a specific set of tests to run. The selection of multiple statuses is supported. By default, the Failed and Not Run test statuses are enabled.

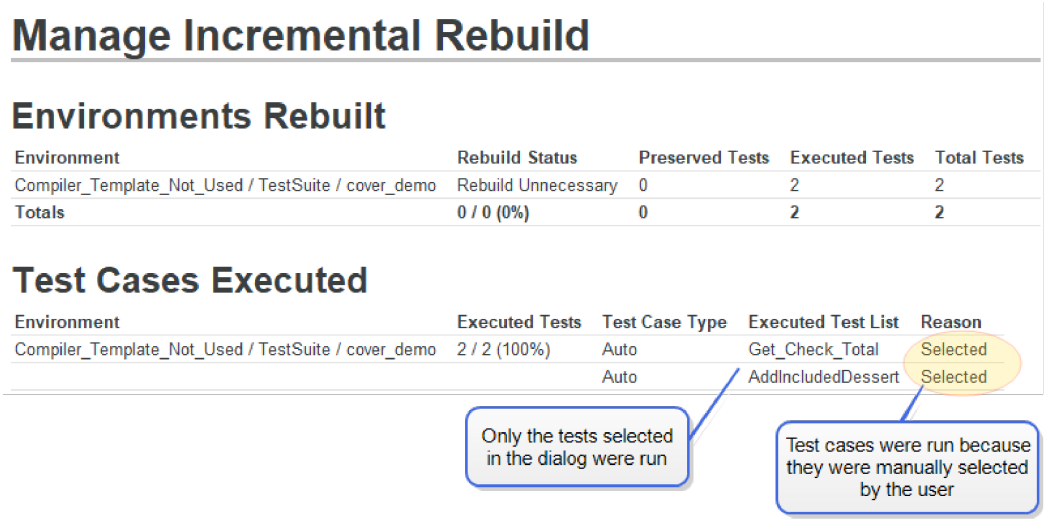


You can further refine the set of tests to be run by highlighting the Test Name(s) and selecting the **Run Selected** button. The button will display the total number of tests being selected to run. By default, VectorCAST runs all of the tests listed when no selections are made.

In our example, we will only run two tests, **`AddIncludedDessert`** and **`Get_Check_Total`**.



Select the **Run Selected** button to execute the list of selected test cases. The Manage Incremental Rebuild Report is displayed. Note that only our two manually selected test cases are executed.




```
manage -p <project> --level <level> -e <system test env> --system-
tests-status[=<filename>]

Lists the system tests and their execution status for the project or
the specified --level and environment. (default='')
```

```
manage -p <project> --level <level> -e <system test env> --execute-
system-tests

Executes all system tests for the given environment. To specify a sub-
```

```
set of all tests, use one or more of the optional flags:
--automated
--manual
--not-run
--failed
--passed
--testcase-name=<TEST_CASE_NAME> (May provide multiple test case
names.)

Note: this command differs from the global --build-execute or --
execute commands which cause only the not-run, automated tests to run.
```

Component Coverage

VectorCAST/QA provides support for component coverage. Component coverage is often used when an entire instrumented image is too large to fit on a target. When component coverage is active, the instrumentation of the application is performed for one component at a time, and the full set of tests is run for each component. Using component coverage reduces the amount of memory required to run tests since only a portion of the application is instrumented at any time.

Enabling Component Coverage

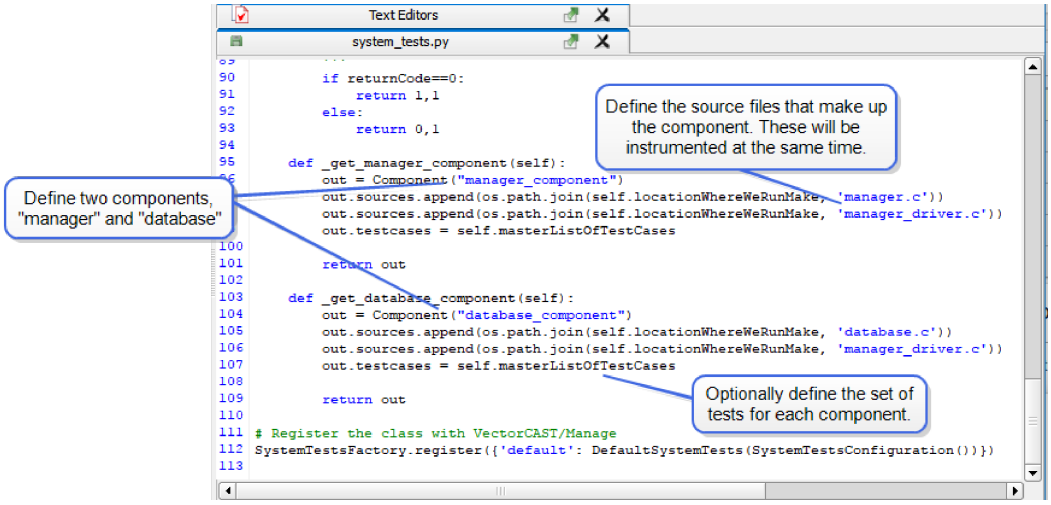
Component coverage is disabled by default. To enable component coverage, you must modify the python configuration script, **`system_tests.py`**. To open and edit the configuration script, right-click on a Cover Environment node and select **System Testing => Edit Script** from the context menu.

The **`system_tests.py`** script opens in the Script Editor. Modify the script as required and save changes. For our example, we first uncomment the following line of code to enable component coverage for two components, **`manager_component`** and **`database_component`**, and save the change:

```
self.components = [self._get_manager_component(), self._get_database_
component()]
```

`self.components` is a list of VectorCAST components. Each component is a sub-set of the files in the application. In our example, the components are built in the **`_get_manager_component`** and **`_get_database_component`** functions defined toward the bottom of our **`system_tests.py`** file.

Component names must be unique. Note that the **`system_tests.py`** script lists the source files that make up each component.



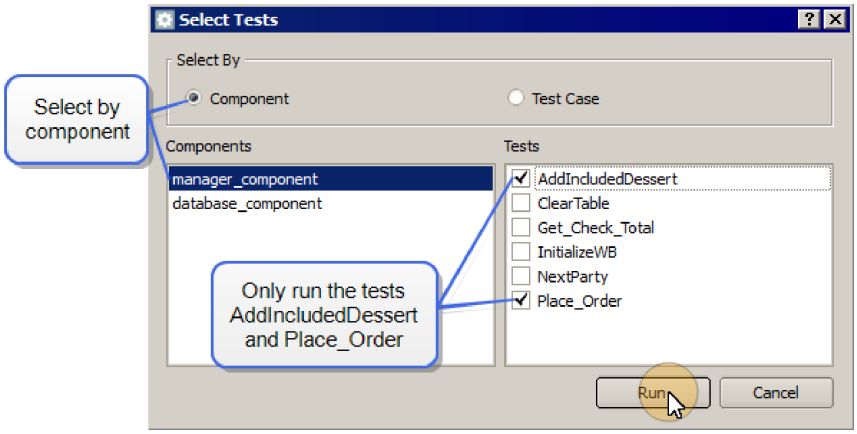
In the example above **`manager_component`** is defined on line 95. The source files that make up **`manager_component`** are listed. This set of files will be instrumented at the same time. Source files can be added or removed from a component by editing the configuration script and saving the changes.

Building and Executing on Components

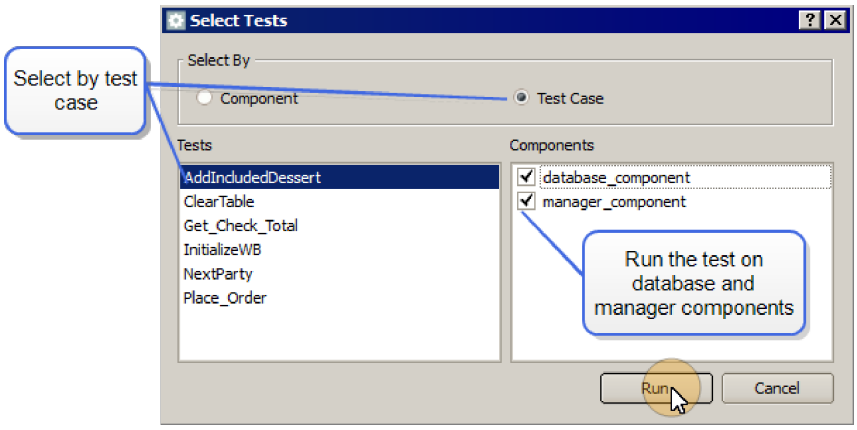
Performing a full Build/Execute action on a Cover environment will run all the tests for each component. Optionally, an interactive Build/Execute can be performed allowing you to select specific test cases to be run for a component. For example, you may only want to run a single test for all components, or you may want to run just one test on one component.

When component coverage is active, clicking on a Cover environment in the Project Tree and selecting **Build/Execute => Interactive** from the context menu opens the Select Tests dialog. Select the tests to execute based on Component or based on Test Case.

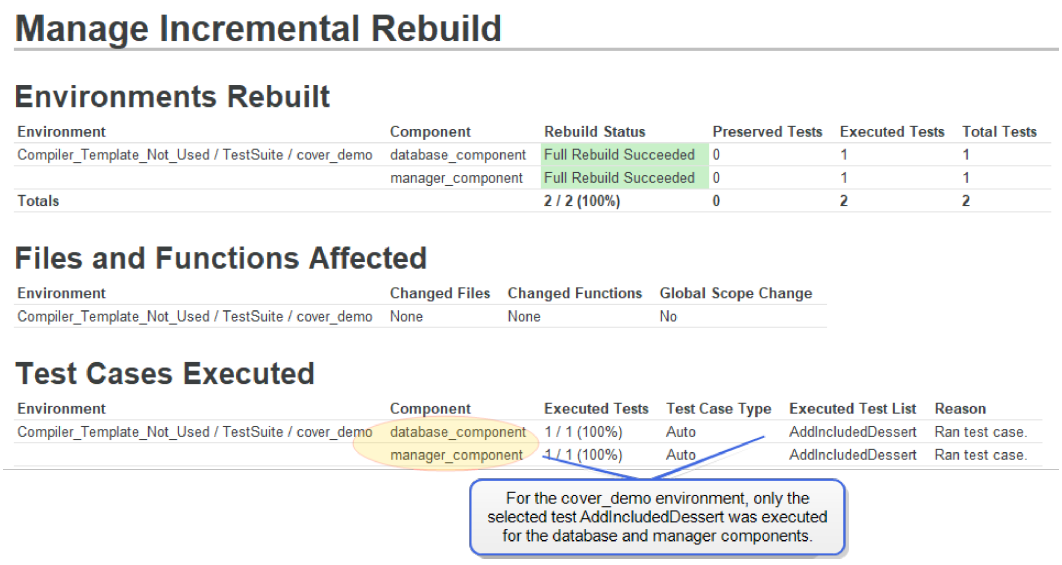
To select by Component, select the **Component** radio button. A list of available components for the environment is shown in the left pane. Highlight a component and the associated tests are listed in the right pane. Check the desired test(s) and select the **Run** button.



To select by Test Case, select the **Test Case** radio button. A list of available tests is shown in the left pane. Highlight a test case and the associated components are listed in the right pane. Check the desired component(s) and select the **Run** button.



In our example, we ran the test **`AddIncludedDessert`** on both **`database_component`** and **`manager_component`**, and the results are provided in the Incremental Build Report. Running a test only on the components it touches saves time and resources.



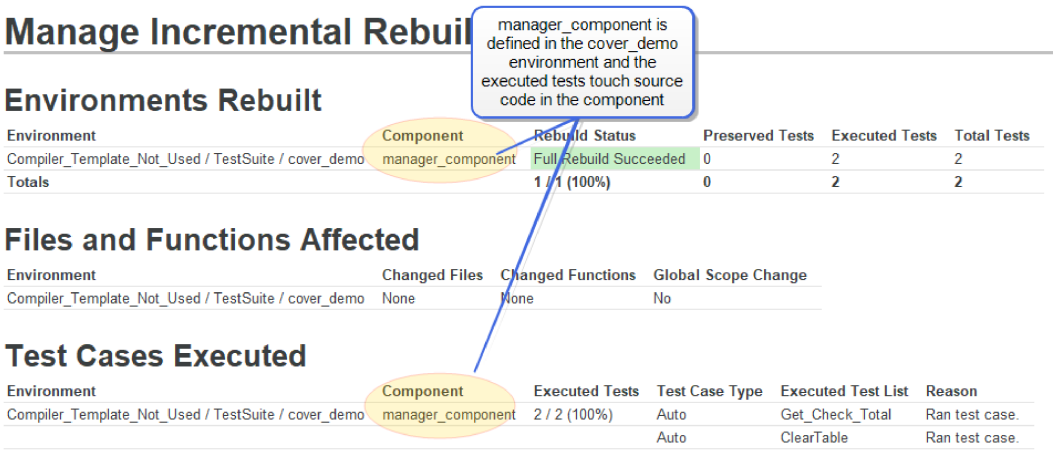

```
manage -p <project> --level <level> -e <system test env> --component-
name=<component_name> --testcase-name=<test_case_name> --execute-
system-tests

Executes system tests for components in the given environment. To
specify a component, use the optional flag --component name.

To specify a sub-set of tests, use the optional flag --testcase-name.

Note: the --component-name and --testcase-name options may be provided
```

```
    multiple times.
```

Viewing Component Coverage Data

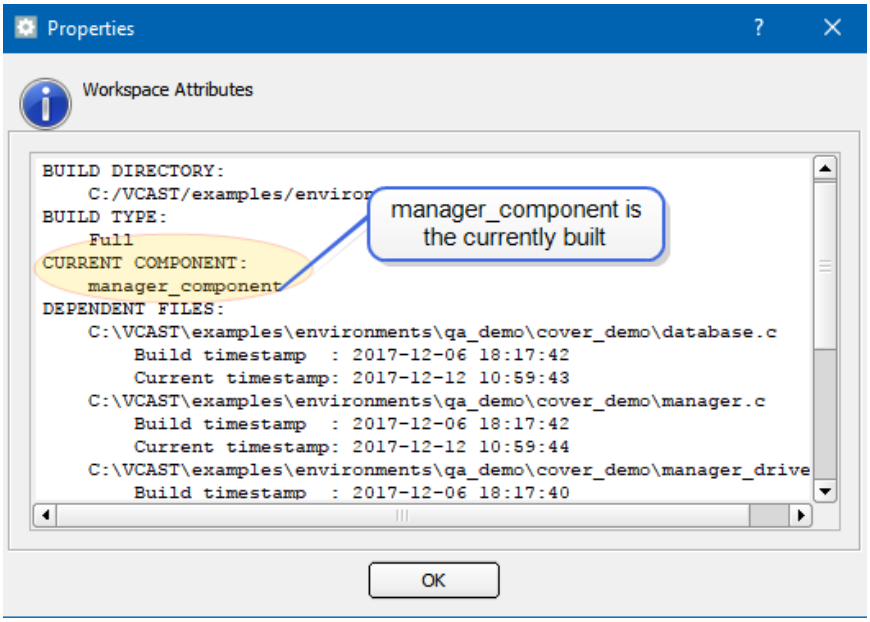
Manage Incremental Rebuild Report

When performing a Build/Execute action with Component Coverage active, the Incremental Rebuild Report includes a Component column in the Environments Rebuilt section. This column identifies the component(s) which are affected when an environment is rebuilt. The report also includes a Components column in the Test Cases Executed section which indicates the component that is affected when a test case is executed that touches source files that make up the component.

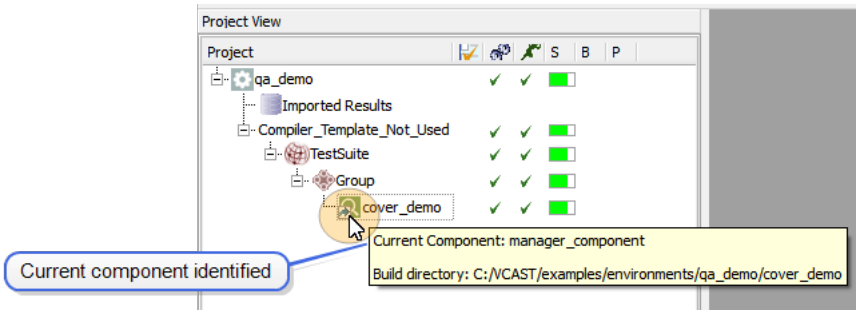


Properties Dialog

Right-clicking on an environment node in the Project Tree and selecting **Properties** from the context menu opens the Properties Dialog for the Workspace Attributes. The currently built component is listed.



The current component can also be quickly identified by hovering over the environment in the Project Tree and displaying the associated tool-tip.

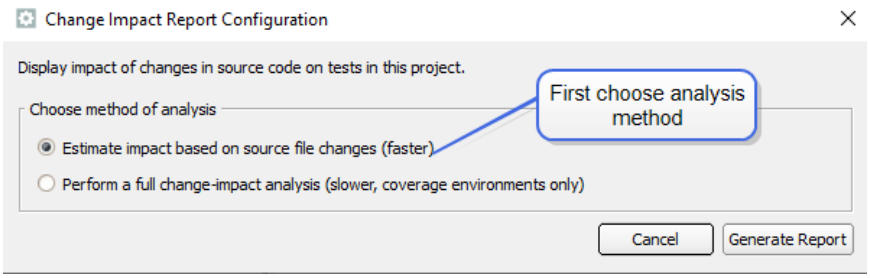


Change Impact Report

VectorCAST/QA provides the ability to evaluate the effect of source code changes on test cases using the Change Impact Report. This feature evaluates "what-if" situations.

For example, you may be considering re-factoring a widely-used function. The Change Impact Report shows that the planned changes will trigger the need to re-run 85% of the existing tests. Knowing this in advance, you might choose to postpone the planned changes until schedule pressure eases.

To generate a Change Impact Report, select **Project => Change Impact Report...** from the Menu Bar. The Change Impact Report Configuration dialog opens.



The dialog provides access to both the estimated Change Impact Report for unit test and Coverage environments and the Build/Execute Dry Run report for Coverage environments.

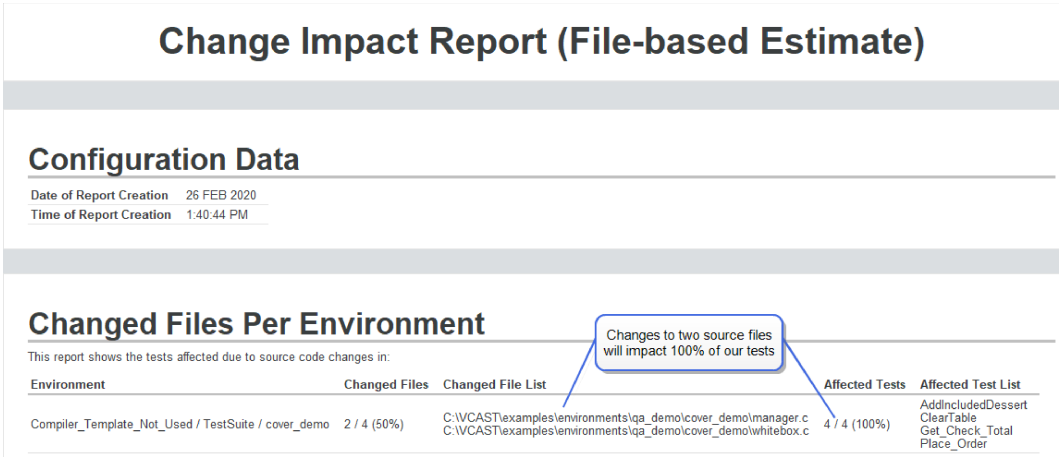
Estimated Change Impact Report

The Estimated Change Impact Report method can be used for unit test and Coverage environments. This method also supports displaying the impact from changes in an alternative root source directory on tests in the Project. Select the **Estimate impact based on source file changes** radio button.

Note: All tests that touch a file are considered affected by any change to that source file. This may differ from the actual action taken by incremental build/execute.

Select the **Generate Report** button. The title of report is "Change Impact Report (File-based Estimate)." When generated in the GUI, the report is saved automatically in the **`<project>/build/vcast_data`** directory.

```
manage -p <project> [--level=<:level>] --change-impact-report

The default filename is <project>_change_impact_report.html in the working
directory.
```



Full Change Impact Report

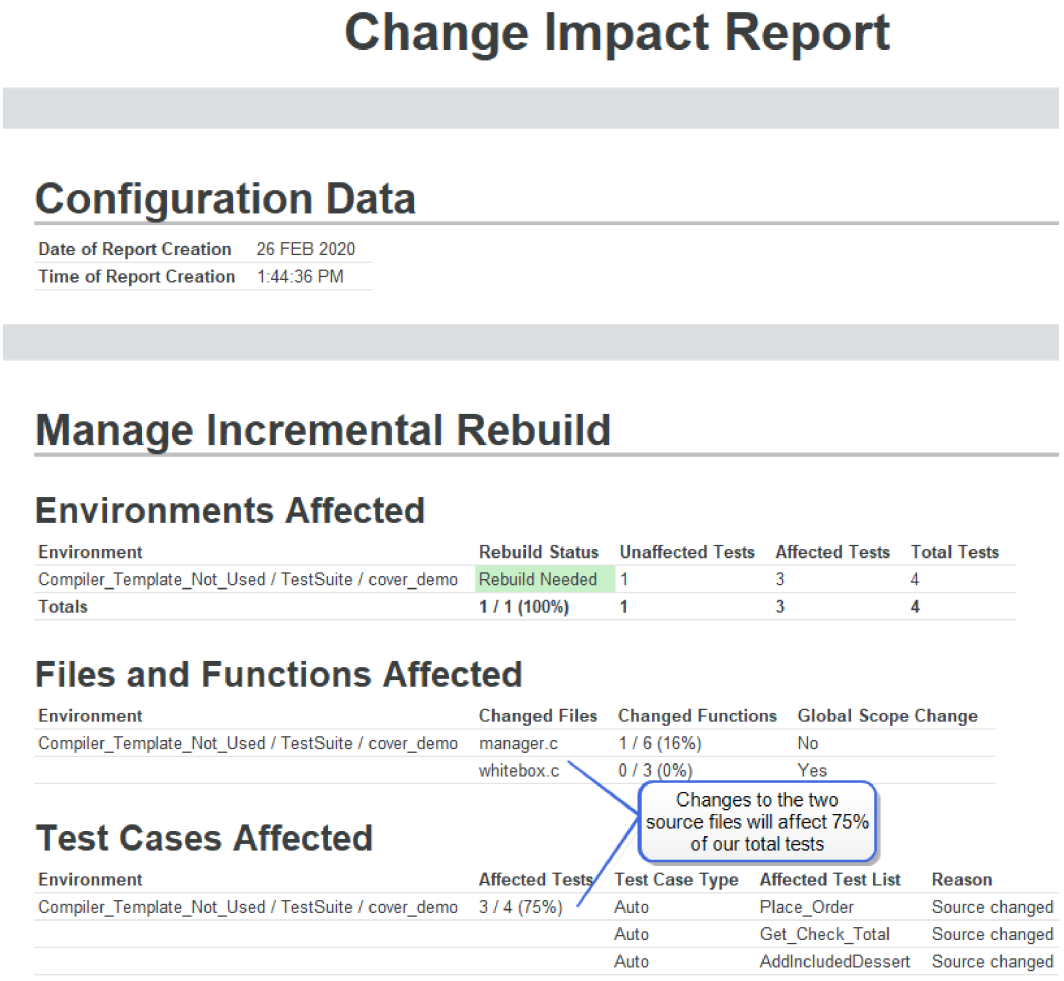
The Full Change Impact Report method is only used for Coverage environments that have a system

test implementation defined in the **`system_tests.py`** file. Select the **Perform a full change-impact analysis** radio button.

Note that only those tests that touch a changed function are considered affected. This method is slower due to preprocessing, but produces output matching the actual actions taken by incremental build/execute.

Select the **Generate Report** button.The title of report is "Change Impact Report." When generated in the GUI, the report is saved automatically in the **`<project>/build/vcast_data`** directory.

```
manage -p <project> [--level=<:level>] --change-impact-report --full-
analysis

The default filename is <project>_manage_incremental_rebuild_report.html in
the working directory.
```



Change Impact Report

Configuration Data

Date of Report Creation	26 FEB 2020
Time of Report Creation	1:44:36 PM

Manage Incremental Rebuild

Environments Affected

Environment	Rebuild Status	Unaffected Tests	Affected Tests	Total Tests
Compiler_Template_Not_Used / TestSuite / cover_demo	Rebuild Needed	1	3	4
Totals	**1 / 1 (100%)**	**1**	**3**	**4**

Files and Functions Affected

Environment	Changed Files	Changed Functions	Global Scope Change
Compiler_Template_Not_Used / TestSuite / cover_demo	manager.c	1 / 6 (16%)	No
	whitebox.c	0 / 3 (0%)	Yes

Changes to the two source files will affect 75% of our total tests

Test Cases Affected

Environment	Affected Tests	Test Case Type	Affected Test List	Reason
Compiler_Template_Not_Used / TestSuite / cover_demo	3 / 4 (75%)	Auto	Place_Order	Source changed
		Auto	Get_Check_Total	Source changed
		Auto	AddIncludedDessert	Source changed

Enterprise Testing and Jenkins Integration

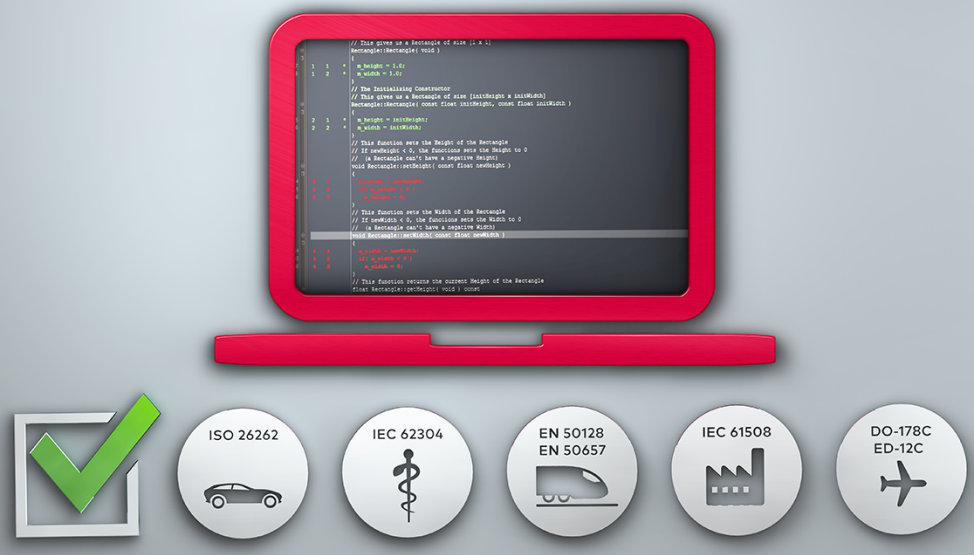
Enterprise Testing and Jenkins integration enables users to implement a Continuous Test process, where each code change is completely tested against all existing test cases prior to integration. This results in the shortest possible feedback loop for the developer, allowing bugs to be identified within minutes of code changes, rather than weeks.

Jenkins is an extensible, open source automation server for continuous integration and continuous delivery for any application. Two plugins are available from the Jenkins website for use with VectorCAST projects: VectorCAST Execution and VectorCAST Coverage.

The VectorCAST Execution plugin allows the user to create, delete and update jobs to build and run VectorCAST projects. Jobs can be created as a single job or split into multiple jobs for a VectorCAST project, with one job for each environment and an overall job to combine the results. The plugin adds a new top-level menu item to the Jenkins sidebar that provides job control for VectorCAST projects. To learn more about the Execution plugin, visit the Jenkins website at https://plugins.jenkins.io/vectorcast-execution.

The VectorCAST Coverage plugin allows the user to capture code coverage reports from VectorCAST projects. The VectorCAST Coverage plugin is added as a dependency to the VectorCAST Execution plugin and is automatically used to display coverage data. Jenkins automatically generates the trend report of coverage and displays a coverage trend graph at the top page of a job. To learn more about the Coverage plugin, visit the Jenkins website at https://plugins.jenkins.io/vectorcast-coverage.

General information on plugin installation is provided on the Jenkins' wiki at https://wiki.jenkins.io/display/JENKINS/Plugins.



Command Line Interface

Introduction

The Manage Command Line Interface (CLI) enables you to further automate testing by creating scripts to automate frequently used commands, or to integrate Enterprise Testing with a Continuous Integration process.

To see a summary of all the available manage commands enter:

```
%VECTORCAST_DIR%\manage --help
```

Running Manage CLI in Parallel

The Manage CLI supports adding a **--jobs** argument to specify the number of jobs to run an action across environments simultaneously. The flag may be used in combination with:

- **--build**
- **--execute**
- **--build-execute**
- **--build-execute --incremental**
- **--clicast-args <CLICAST-ARGS>**
- **--batch-env-commands**

For example, to build and execute all environments across the project using up to 32 simultaneous jobs:

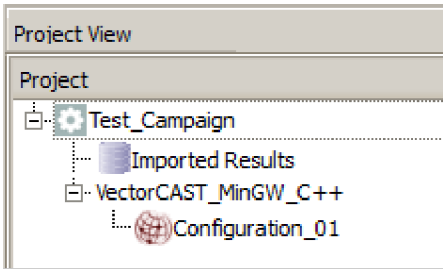
```
%VECTORCAST_DIR%\manage
   --project Project
   --build-execute
   --jobs=32
```

By default, the output of the jobs is not output to the screen, instead a status line is output which provides progress updates. To view each individual jobs output, add the **--verbose** argument to the command line.

Running a Script

In general, the Manage CLI requires two pieces of information: the name of the VectorCAST project and the command to execute. Also, it is possible to limit the command to a sub-section of the project by providing the name of a compiler or test suite node.

For example, given the project structure shown below:



The command to create a new test suite named “Configuration_02” under the VectorCAST_MinGW_C++ node would be:

```
%VECTORCAST_DIR%\manage
    --project=Test_Campaign
    --compiler=VectorCAST_MinGW_C++
    --testsuite=Configuration_02
    --create
```

For purposes of clarity, each argument and its associated parameter is shown on a separate line. In reality, the command is written as a single line.

Let’s take a quick look at the example above line by line.

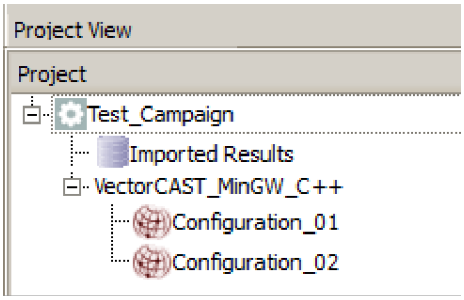
```
%VECTORCAST_DIR%\manage          Invokes the manage executable
--project=Test_Campaign          The project name is specified
--compiler=VectorCAST_MinGW_C++  The compiler name is specified
--testsuite=Configuration_02     The new test suite name is specified
--create                         The create command is specified
```

An alternative command that accomplishes the same result is:

```
%VECTORCAST_DIR%\manage
    --project=Test_Campaign
    --level=VectorCAST_MinGW_C++/Configuration_02
    --create
```

In this example, the level command allows all of the levels to be concatenated into a single argument. Notice that the levels are separated with a forward slash (“/”). That is because the level parameter is a delimited string and not a Windows path. The “/” is used for both Windows and Unix environments.

After executing the command above, the Project Tree now appears as:



Running a Manage Script with Multiple Commands

VectorCAST can also run a script containing multiple commands. One advantage of invoking multiple commands from a single command line is the resulting processing speed improvements. Another advantage to a multiple command script is the ease with which frequently used combinations of commands can be stored in scripts and reused, saving time, ensuring consistency, and streamlining the testing process. This automation is especially useful in a Continuous Integration environment.

Note: In order to run a script containing multiple commands, the VectorCAST project must already exist.

The VectorCAST project is loaded when the script is invoked. If an error is encountered during execution, the script processing aborts unless the command **`--continue-on-error`** is given.

The usage is:

```
manage -p <PROJECT> --script <SCRIPT> [--continue-on-error]
```

For example, we can create the following script file named **`Build1.bat`**:

```
--level GNU_C_41/SUITE --create
--group GROUP --create
--level GNU_C_41/SUITE --add GROUP
--import ENV
--group GROUP --add ENV
--migrate-to-workspace
--build-execute
--full-status
```

First, we will create our VectorCAST project with this command:

```
%VECTORCAST_DIR%\manage --project=Test_Campaign --create
```

Now, we will invoke our **`Build1.bat`** script on our existing VectorCAST project, Test_Campaign, using the following command:

```
%VECTORCAST_DIR%\manage --project=Test_Campaign --script=Build1.bat [--
continue-on-error]
```

Let's take a look at what happens with the script line by line:

`--level GNU_C_41/SUITE --create`	**Creates the compiler and test suite nodes: GNU_C_41 (compiler), SUITE (testsuite)**
`--group GROUP --create`	**Creates the environment group, GROUP**
`--level GNU_C_41/SUITE --add GROUP`	**Adds the environment group, GROUP, to the testsuite node, SUITE**
`--import ENV`	**Imports the ENV environment to the project**
`--group GROUP --add ENV`	**Adds the ENV environment to the environment group, GROUP**
`--migrate-to-workspace`	**Migrates the ENV environment into the Workspace**
`--build-execute`	**Fully builds and executes all the environments and then executes all the tests**
`--full-status`	**Gets the full status of each environment in the project**

Running a Script with a Pipeline of Environment Commands

It is possible to run a set of Manage CLI commands across multiple environments by use of the **`--batch-env-commands <SCRIPT>`** argument. This argument accepts the name of a script file. The file contents are a list of Manage CLI environment action commands separated by line, which include only the following commands: **`--build, --execute, --build-execute, --build-execute --incremental`**, and **`--clicast-args <CLICAST ARGS>`**.

The benefit of using such a script is that a single invocation of Manage CLI is capable of running multiple commands, thereby improving the speed of execution. The **`--batch-env-commands`** argument may also be combined with the **`--jobs`** argument to run the script simultaneously across environments.

For example, the following script, **`build_and_report.msh`**, will perform a build-execute incremental action, followed by a clicast command to generate a report.

File: **`build_and_report.msh`**

```
--build-execute --incremental
--clicast-args report custom full
```

To run these actions for each environment within the project, issue the following:

```
manage -p Project --batch-env-commands build_and_report.msh
```

An Example for Using the Command Line Interface

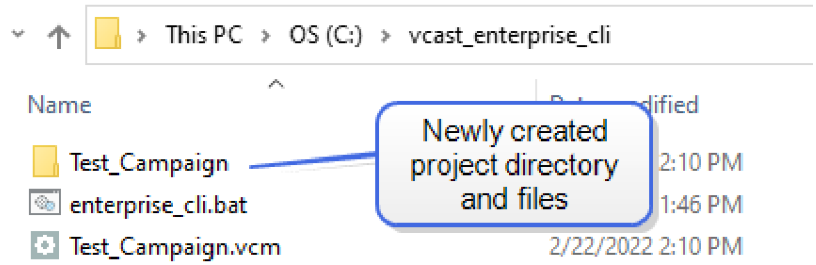
In this section, the Manage CLI is used to create a VectorCAST project named Test_Campaign. First, for the examples in this section, the ENV_DATABASE and ENV_MANAGER environments are built in the **`C:\vcast_enterprise_cli\enterprise_tutorial\ENTERPRISE\unit_test_envs`** working directory.

Next, the Manage CLI script is developed. Each line of the script is described below and the results of executing each line is illustrated by invoking the VectorCAST GUI to graphically show the results. The full script is provided at the end of this example.

Creating the Project

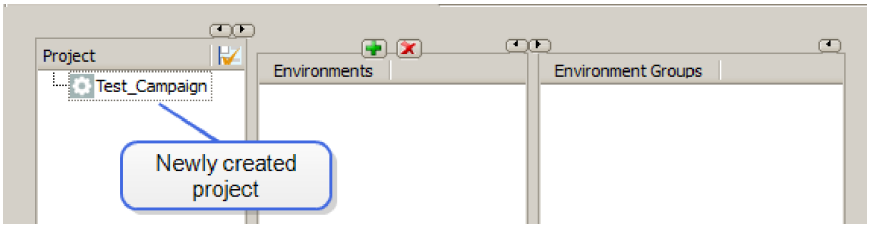
The first step is to create the VectorCAST Project. The script file, named "**`enterprise_cli.bat`**" is placed and run in a newly created and empty directory called **`C:\vcast_enterprise_cli`**.

```
%VECTORCAST_DIR%\manage
  --project=Test_Campaign
  --create
```



Note that the first argument for any manage command is always the *--project* argument. The project must be identified for each command.

Executing this command will create the Project directories and files.



Creating an Environment Group

Next we create the environment group called "**`unit_test_envs`**". The following CLI command line creates the environment group.

```
%VECTORCAST_DIR%\manage
  --project=Test_Campaign
  --group=unit_test_envs
  --create
```

The command line invokes the manage executable. It specifies the project name, the environment group name to be created, and the **--create** action.

A graphical view of the result is shown below.

Creating Compiler and Test Suite Nodes

Next, we create the desired nodes below the Project node in the Project Tree hierarchy.

When creating a node, the **--level** command argument must be used. To specify the level, a "/" delimited list of node names is provided as the parameter to the level argument. Note that the delimiter is the same for both Windows and UNIX. In the example below, all of the nodes in the list are created. If a node in the list has been previously created, it is ignored by the **--create** command and no error or warning is displayed.

```
%VECTORCAST_DIR%\manage
    --project=Test_Campaign
    --level=VectorCAST_MinGW_C++/Configuration_01
    --create
```

This command creates the VectorCAST_MinGW_C++ node and its nested Configuration_01 node.

After executing, the VectorCAST Project looks as follows:

Importing Test Environments

Now that the basic VectorCAST Project framework has been created for Configuration_01, test

environments are imported. The first test environment imported is the ENV_MANAGER environment.

```
%VECTORCAST_DIR%\manage
    --project=Test_Campaign
    --group=unit_test_envs
    --import .\unit_test_envs\ENV_MANAGER.vce
```

To import a test environment, the **--group** argument is used to specify the *environment group* name as the *destination* for the imported environment. In this example, it is the group "**unit_test_envs**" that was created earlier.

The **--import** command action identifies the operation to perform.

Next the importation *source* is specified, including its physical path. You may specify:

- a .vce file,
- an environment directory,
- a .env, .tst and a .CFG file,
- or a .env, .tst and a regression script (.csh/.bat) file.

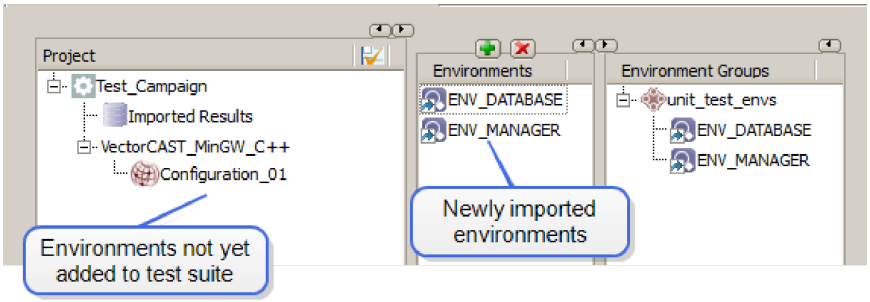
How the Compiler is Determined

When using the manage command line to import environments, no factoring of the compiler is performed, as is done in the GUI. When an environment is imported using the **--import** command, the files in the *source* directory for the environments are searched for information on the compiler used to build that environment. VectorCAST searches the .CFG files and any .bat (.csh) files for "**C_COMPILER_TAG**" and "**COMPILATION_SYSTEM**." If found, the compiler is set at the environment level in the project.

To illustrate **--import** using an environment directory instead of a .vce file, the following command uses the **ENV_DATABASE** directory as the importation source instead of its .vce file.

```
%VECTORCAST_DIR%\manage
  --project=Test_Campaign
  --group=unit_test_envs
  --import=.\unit_test_envs\ENV_DATABASE
```

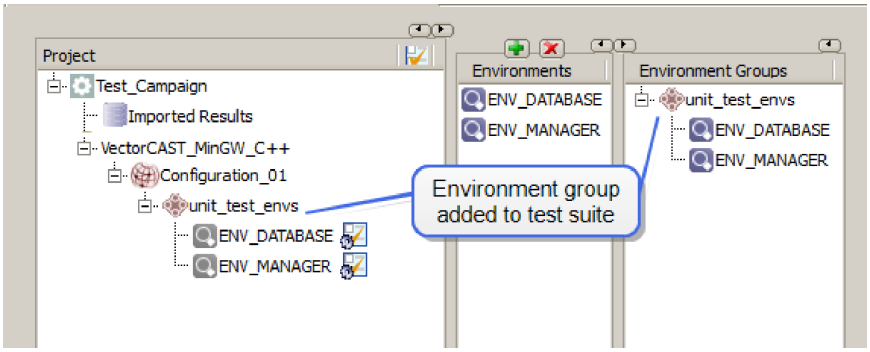
Here is the result of executing the import commands. Both environments have been added to the unit_test_envs group, but notice that the group has not yet been added to the project's test suite. This will be accomplished next.



Adding an Environment Group to a Test Suite

To add an environment group to a test suite, both the environment group and the level to which it is being added to must be specified. In this example we are adding the *unit_test_envs* group to the *Configuration_01* test suite, and the standard output is shown.

```
%VECTORCAST_DIR%\manage
  --project=Test_Campaign
  --level=VectorCAST_MinGW_C++/Configuration_01
  --add=unit_test_envs
```



Building and Executing the Project

The project is now ready to be built and executed using the CLI.

The following command is used to both build and execute the entire project.

```
%VECTORCAST_DIR%\manage
   --project=Test_Campaign
   --build-execute
```

The **--build-execute** command combines both building and execution in one command. To build and not execute, substitute **--build-execute** with **--build**, and to execute but not build, substitute **--build-execute** with **--execute**.

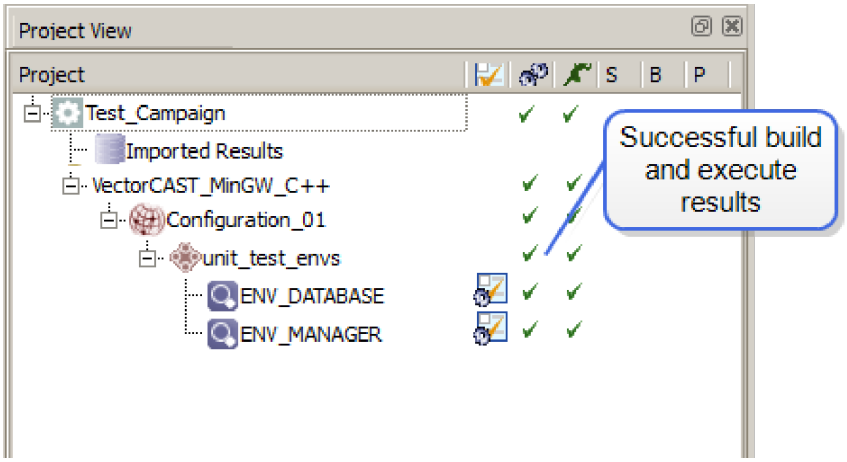
Add the flag **--incremental** to the **--build-execute** command to perform a rebuild and execute incrementally, as necessary. For example, if the source code has not changed, then a rebuild is not

necessary, so proceed to incremental execution. Or if an environment has not been built or if the incremental rebuild detects a global change, then fully build the environment. Note that Ada environments are only rebuilt if their source code has changed.

For all three commands, **--build-execute, --build**, and **--execute**, the command requires a build destination. Note that the **--workspace** argument is optional for these commands and is only needed if you want to build the environments in a user-defined location. **--workspace** requires a path parameter to the build destination directory, which can be specified as relative to the working directory.

```
--workspace=.\Test_Campaign\build
```

After executing this command, the GUI indicates that Configuration_01 was built and executed without errors.



Generating Reports

To create a full project status report, the **--full-status** command is used. If no file name is specified as an argument, the report appears in the console. If a filename is specified with an .html extension, an html report is created, otherwise a .txt report is created. A report may be generated for any level in the project by using the **--level** argument as part of the command.

```
%VECTORCAST_DIR%\manage
  --project=Test_Campaign
  --full-status=full_status.html
```

Manage Report

Configuration Data

Date of Report Creation	13 MAR 2020
Time of Report Creation	11:47:27 AM
Version	20 revision 9019158 (03/03/20)

Full Status Section

	BUILD	BUILD TIME	EXPECTED	TESTCASES	EXECUTE TIME	Statements	Branches	Pairs
ALL	1/1 (100%)	00:11	8/8 (100%)	2/2 (100%)	00:01	13/50 (26%)	7/45 (15%)	0/11 (0%)
VectorCAST_MinGW_C	1/1 (100%)	00:11	8/8 (100%)	2/2 (100%)	00:01	13/50 (26%)	7/45 (15%)	0/11 (0%)
TestSuite	1/1 (100%)	00:11	8/8 (100%)	2/2 (100%)	00:01	13/50 (26%)	7/45 (15%)	0/11 (0%)
Group	1/1 (100%)	00:11	8/8 (100%)	2/2 (100%)	00:01	13/50 (26%)	7/45 (15%)	0/11 (0%)
TUTORIAL_C	NORMAL	00:11	8/8 (100%)	2/2 (100%)	00:01	13/50 (26%)	7/45 (15%)	0/11 (0%)

To generate a *status summary* for a test suite, environment or project, use the **--status** command instead of **--full-status**.

Both *aggregate coverage* and *metrics* reports can be generated by using the **--create-report** command.

The syntax for the create-report command is:

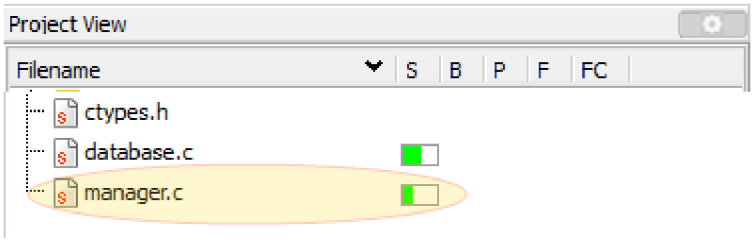
```
manage -p <project> --create-report=<aggregate|csv_metrics|metrics> [--
output=<PATH>] [--path=<PATH>]
```

where:

<create-report> is the report type: Aggregate, CSV_Metrics or Metrics.

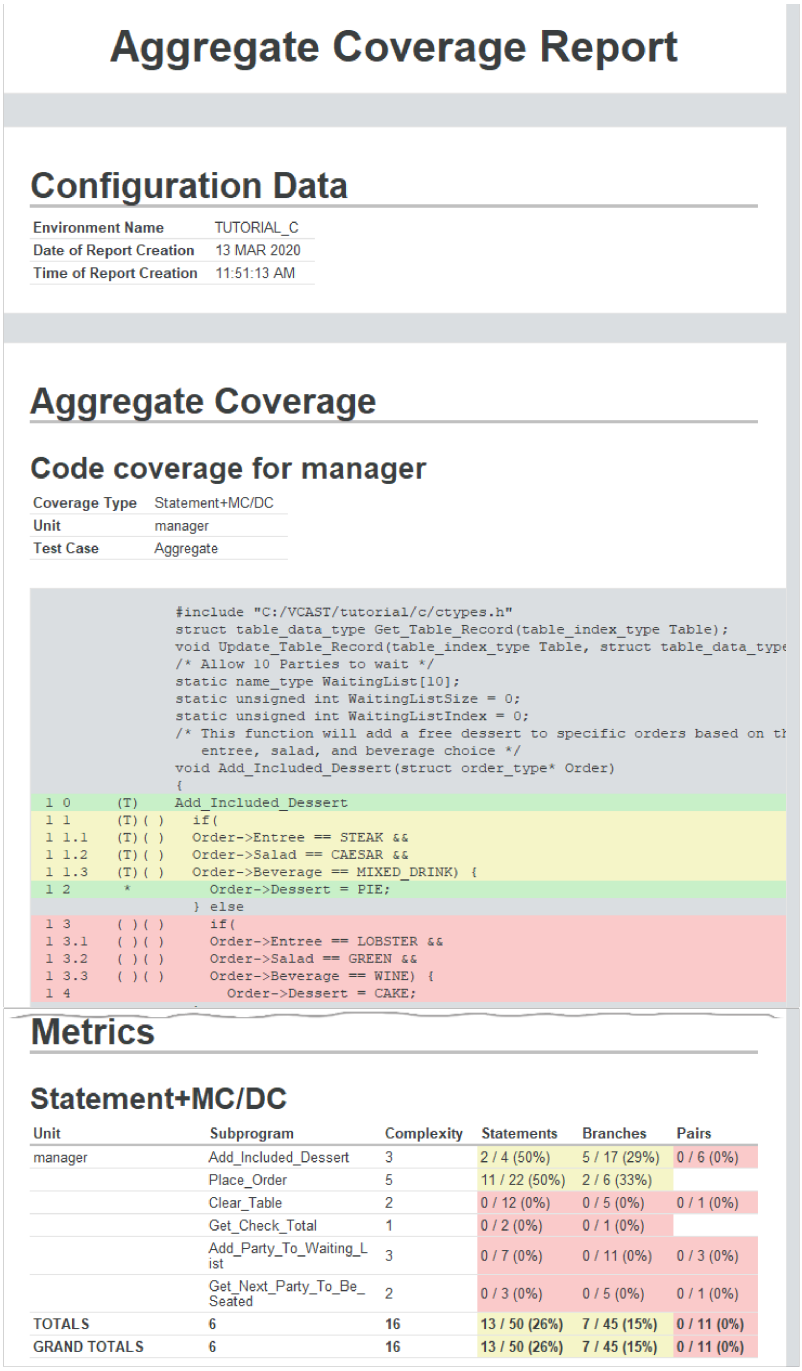
<output> is the location to write the report.

<path> is the filesystem path used to specify the sources to include in the report. For the aggregate and metrics reports, the path may be a file or directory and can be provided multiple times on the command line.



For example, to generate an aggregate coverage report for manager.c in baseline1, the following command is used:

```
%VECTORCAST_DIR%\manage
    --project=TEST_CAMPAIGN
    --create-report=aggregate
    --output=baseline_1_aggr_manager.c.html
    --path=c:\vcast_tutorial\manage_tutorial\MANAGE\baseline1\manager.c
```



Aggregate Coverage Report

Configuration Data

Environment Name	TUTORIAL_C
Date of Report Creation	13 MAR 2020
Time of Report Creation	11:51:13 AM

Aggregate Coverage

Code coverage for manager

Coverage Type	Statement+MC/DC
Unit	manager
Test Case	Aggregate

```
                #include "C:/VCAST/tutorial/c/ctypes.h"
                struct table_data_type Get_Table_Record(table_index_type Table);
                void Update_Table_Record(table_index_type Table, struct table_data_type
                /* Allow 10 Parties to wait */
                static name_type WaitingList[10];
                static unsigned int WaitingListSize = 0;
                static unsigned int WaitingListIndex = 0;
                /* This function will add a free dessert to specific orders based on th
                   entree, salad, and beverage choice */
                void Add_Included_Dessert(struct order_type* Order)
                {
1 0      (T)     Add_Included_Dessert
1 1      (T)( )    if(
1 1.1    (T)( )    Order->Entree == STEAK &&
1 1.2    (T)( )    Order->Salad == CAESAR &&
1 1.3    (T)( )    Order->Beverage == MIXED_DRINK) {
1 2       *          Order->Dessert = PIE;
                   } else
1 3      ( )( )     if(
1 3.1    ( )( )     Order->Entree == LOBSTER &&
1 3.2    ( )( )     Order->Salad == GREEN &&
1 3.3    ( )( )     Order->Beverage == WINE) {
1 4                   Order->Dessert = CAKE;
```

Metrics

Statement+MC/DC

Unit	Subprogram	Complexity	Statements	Branches	Pairs
manager	Add_Included_Dessert	3	2 / 4 (50%)	5 / 17 (29%)	0 / 6 (0%)
	Place_Order	5	11 / 22 (50%)	2 / 6 (33%)	
	Clear_Table	2	0 / 12 (0%)	0 / 5 (0%)	0 / 1 (0%)
	Get_Check_Total	1	0 / 2 (0%)	0 / 1 (0%)	
	Add_Party_To_Waiting_List	3	0 / 7 (0%)	0 / 11 (0%)	0 / 3 (0%)
	Get_Next_Party_To_Be_Seated	2	0 / 3 (0%)	0 / 5 (0%)	0 / 1 (0%)
TOTALS	**6**	**16**	**13 / 50 (26%)**	**7 / 45 (15%)**	**0 / 11 (0%)**
GRAND TOTALS	**6**	**16**	**13 / 50 (26%)**	**7 / 45 (15%)**	**0 / 11 (0%)**

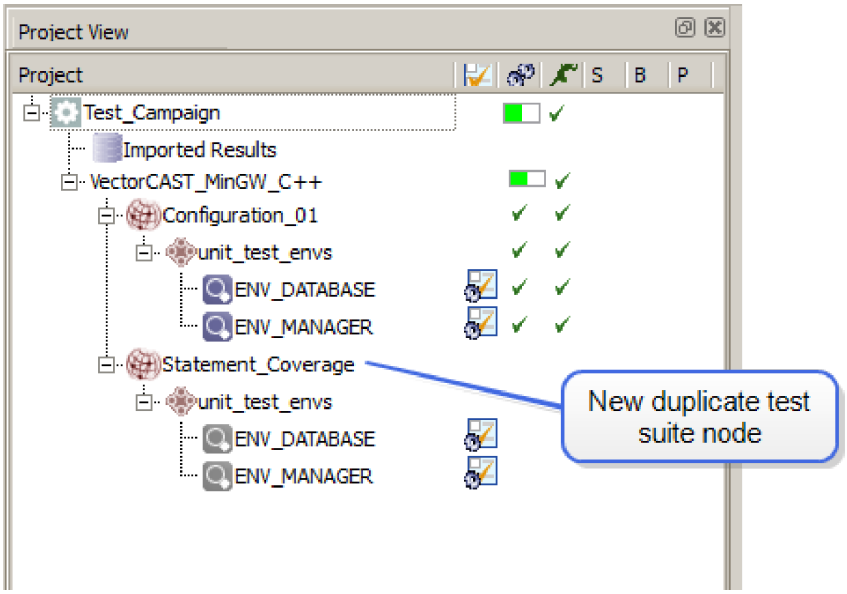
Duplicating a Test Suite

Next, an additional test suite, called *Statement_Coverage* is added to the project by cloning *Configuration_01*. To create the clone, the **--duplicate** command is used.

```
%VECTORCAST_DIR%\manage
   --project=Test_Campaign
   --level=VectorCAST_MinGW_C++/Configuration_01
   --duplicate=Source_1/Windows/VectorCAST_MinGW_C++/Statement_Coverage
```

The level is set to the level of the test suite that is being duplicated, and the **--duplicate** command specifies the location and name of the new test suite. The syntax for the **--duplicate** command is similar to that of the **--level** command.

Here is the result of executing this command:



Setting Coverage Type on a Test Suite

Next, statement coverage is applied to the *Statement_Coverage* test suite. To apply the coverage, the **--coverage-type** command is used.

The level is set to the *Statement_Coverage* test suite and *coverage-type* is set to STATEMENT.

```
%VECTORCAST_DIR%\manage
   --project=Test_Campaign
   --level=VectorCAST_MinGW_C++/Statement_Coverage
   --coverage-type=STATEMENT
```

The following is a list of valid coverage types that may be applied to **--coverage-type** (assuming that the Industry Mode is set to Default. See the VectorCAST User's guide for C++ or Ada for more information on Industry Mode).:

- NONE
- STATEMENT
- BRANCH
- MCDC
- FUNCTION
- FUNCTION+FUNCTION_CALL
- STATEMENT+MC/DC
- STATEMENT+BRANCH
- BASIS_PATHS

Now that coverage has been applied to *Statement_Coverage*, the test suite needs to be built and executed.

To build and execute the *Statement_Coverage* test suite, the **--build-execute** command is used:

```
%VECTORCAST_DIR%\manage
    --project=Test_Campaign
    --level=VectorCAST_MinGW_C++/Statement_Coverage
    --build-execute
```

In this example, the **--level** command is used to restrict the build and execution to the *Statement_Coverage* test suite. Only it will be built and executed.

Example Script (.bat)

```
REM This batch file demonstrates the Manage CLI commands for the Enterprise
tutorial.

REM This batch file assumes that the C_MANAGER and C_DATABASE environments have been
built in the unit_test_envs directory, and their test scripts imported. It also
assumes that there is a CCAST_.CFG file in the same directory, which the --import
command will use to detect your compiler.

REM The comments below delineate the sections of the tutorial and the commands used
in each section.

REM %%%%%%%%%%%%%%%%%%%%%%%%
REM Creating The Project
REM %%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --create

REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
REM Creating an Environment Group
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --group=unit_test_envs --create
```

```
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
REM Creating Compiler and Test Suite Nodes
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --level=VectorCAST_MinGW_
C++/Configuration_01 --create

REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
REM Importing Test Environments
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --group=unit_test_envs --import
.\unit_test_envs\ENV_MANAGER.vce

%VECTORCAST_DIR%\manage --project=Test_Campaign --group=unit_test_envs --import
.\unit_test_envs\ENV_DATABASE

REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
REM Adding an Environment Group to a Test Suite
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --level=VectorCAST_MinGW_
C++/Configuration_01 --add=unit_test_envs

REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
REM Building and Executing the Project
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --workspace=.\Test_Campaign\build --
build-execute

REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
REM Generating Reports
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --full-status=full_status.html

REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
REM Duplicating a Test Suite
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --level=VectorCAST_MinGW_
C++/Configuration_01 --duplicate=VectorCAST_MinGW_C++/Statement_Coverage

REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
REM Setting Coverage Type on a Test Suite
REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%VECTORCAST_DIR%\manage --project=Test_Campaign --level=VectorCAST_MinGW_
C++/Statement_Coverage --coverage-type=STATEMENT
```

```
%VECTORCAST_DIR%\manage --project=Test_Campaign --level=VectorCAST_MinGW_
C++/Statement_Coverage --workspace=.\Test_Campaign\build --build-execute

%VECTORCAST_DIR%\manage --project=Test_Campaign --full-status=full_status.html
```

Index