TESSY V5.1 Features

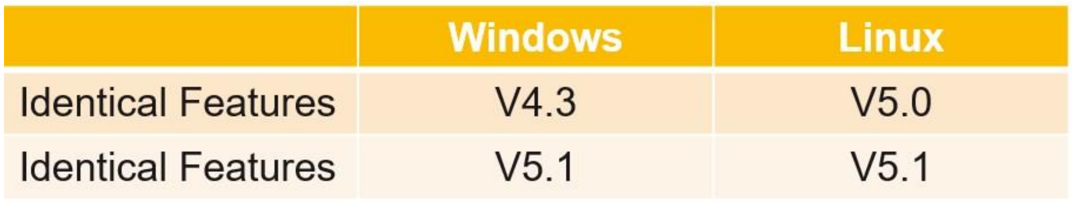
New features in TESSY V5.1 (compared to TESSY V4.3 / TESSY V5.0)

Frank Büchner, March 2023 --- 006

Contents

1 Preliminary Remark on Versions

TESSY V4.3 runs under Windows. TESSY V5.0 is the Linux variant of TESSY V4.3. With respect to features, TESSY V5.0 is identical to TESSY V4.3.



	Windows	Linux
Identical Features	V4.3	V5.0
Identical Features	V5.1	V5.1

Fig. 1: TESSY V5.1 is available on Windows and Linux

TESSY V5.1 is available on Windows and Linux with identical features. A license for TESSY is valid for Windows and Linux.

2 New Icons

All icons were re-designed. Especially the coverage icons now require less space, allowing for narrower columns in the respective views. Below are some examples.

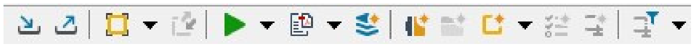


Fig. 2: Redesigned icons in the Test Project view

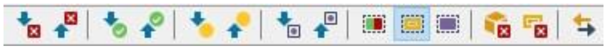


Fig. 3: Redesigned icons in the Test Data view

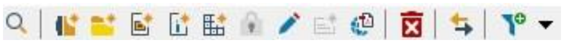
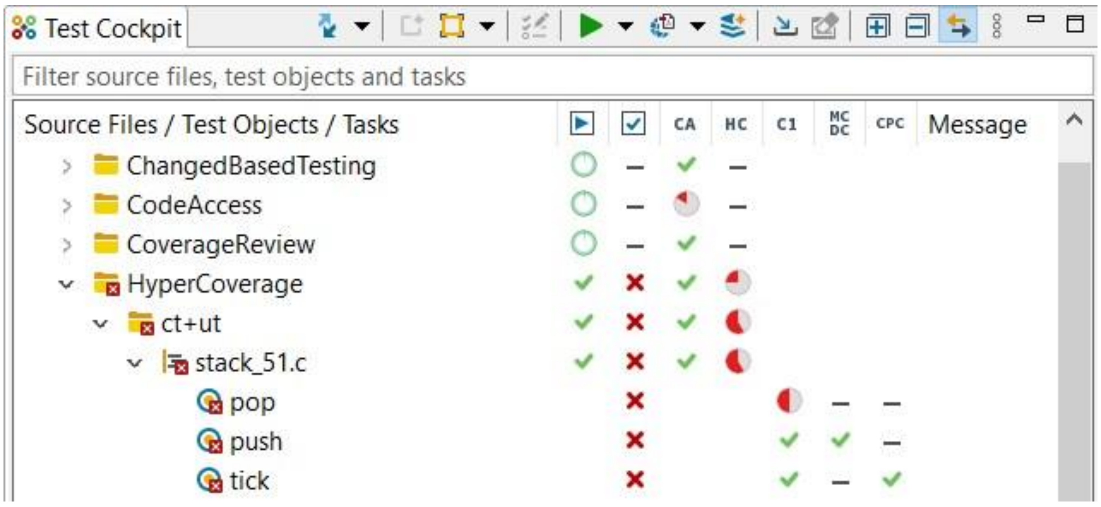


Fig. 4: Redesigned icons in the RQMT Explorer view

3 New View: Test Cockpit View

Based on the source files of the project, both the results of test execution as well as the achieved coverage are summarized.



Test Cockpit

Filter source files, test objects and tasks

Source Files / Test Objects / Tasks	▶	✓	CA	HC	C1	MC DC	CPC	Message
ChangedBasedTesting	○	–	✓	–				
CodeAccess	○	–	◔	–				
CoverageReview	○	–	✓	–				
HyperCoverage	✓	✗	✓	◔				
ct+ut	✓	✗	✓	◐				
stack_51.c	✓	✗	✓	◐				
pop		✗			◐	–	–	
push		✗			✓	✓	–	
tick		✗			✓	–	✓	

Fig. 5: The Test Cockpit view shows information related to source files

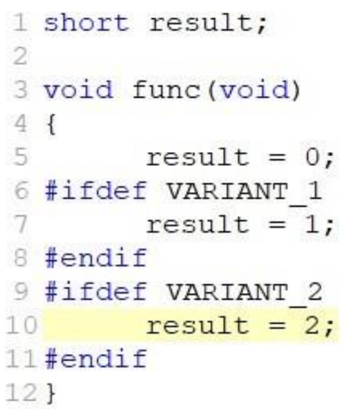
The Test Completion Rate column ▶ shows the relation of the number of test objects with test cases, that are not yet executed, to the total number of test objects with test cases of a source file. Test objects in that source file, which have no tests at all, are not considered in this calculation. Example: If a source file contains 4 test objects, and 2 of these test objects have executed test cases, and 1 test object has testcases, that were not executed, and 1 test object has no test cases, the test completion rate is 66%.

4 Code Access

Is there a variant of the source code which is endangered not to be tested at all?

A source code module is assigned to a TESSY module and related to a TESSY module certain preprocessor constants are #defined (or not). During analysis of the source code assigned to a certain TESSY module, the #defined preprocessor constants are considered. So TESSY can determine which source code lines will be executed with respect to the #defined preprocessor constants for the TESSY module in question. The same source code module can be assigned to a different TESSY module, having different #defined preprocessor constants. During analysis of this TESSY module TESSY can also determine the source code lines that will be executed considering the #defined preprocessor constants of this TESSY module.

TESSY can combine the results of the analyses of the different TESSY modules for the same source file. So TESSY knows which source code line can be executed by the tests related to one or the other TESSY module. Consequently, TESSY also knows which executable source code line cannot be executed, because no TESSY module #defines an appropriate preprocessor constant. This is a very important feature, because it points to potentially executable source code lines that will not undergo testing because they cannot be executed.

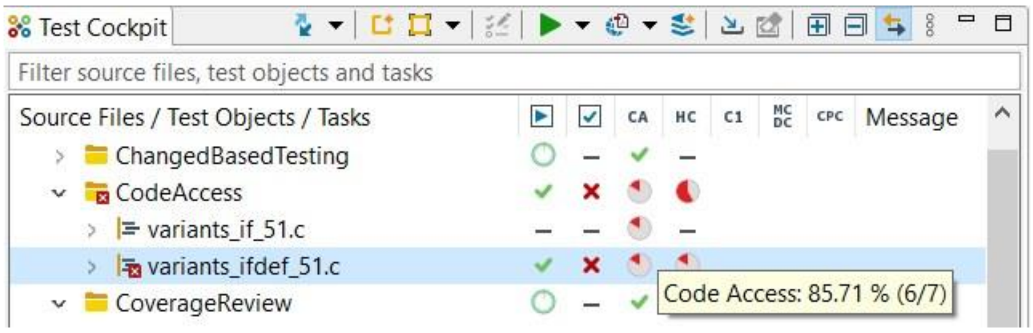


```
 1 short result;
 2
 3 void func(void)
 4 {
 5       result = 0;
 6 #ifdef VARIANT_1
 7       result = 1;
 8 #endif
 9 #ifdef VARIANT_2
10       result = 2;
11 #endif
12 }
```

Fig. 6: No analysis was done in which VARIANT_2 was defined

For instance, if in the example above this source code was not analyzed in a TESSY module for which VARIANT_2 was defined. Hence, the instruction “result = 2;” will not be tested. Therefore, TESSY highlights line 10.

This is reflected in the Code Access result by TESSY. The example above has seven lines with code (1, 3, 4, 5, 7, 10, 12), of which one line (line 10) cannot be executed. 6/7 = 0.85.



Test Cockpit

Filter source files, test objects and tasks

Source Files / Test Objects / Tasks	▶	☑	CA	HC	C1	MC DC	CPC	Message
ChangedBasedTesting	○	–	✓	–				
CodeAccess	✓	✗	◔	◑				
variants_if_51.c	–	–	◔	–				
variants_ifdef_51.c	✓	✗	◔	◔				
CoverageReview	○	–	✓					

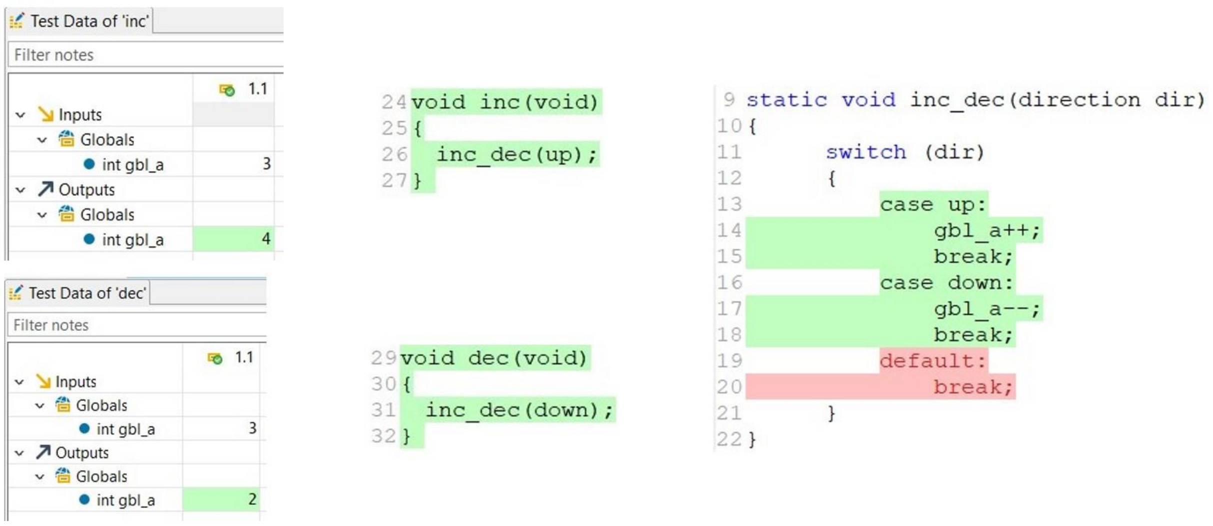
Code Access: 85.71 % (6/7)

Fig. 7: Code Access is not at 100%, because one code line cannot be executed

5 Hyper Coverage

Coverage from unit testing of different test objects can be added, as can coverage from component/integration testing and coverage from unit testing. The latter allows technically to start with component/integration testing and fill the gap to 100% by unit testing.

5.1 Adding Coverage from Unit Testing of Different Test Objects



Test Data of 'inc'

Filter notes

	1.1
Inputs	
Globals	
int gbl_a	3
Outputs	
Globals	
int gbl_a	4

Test Data of 'dec'

Filter notes

	1.1
Inputs	
Globals	
int gbl_a	3
Outputs	
Globals	
int gbl_a	2

```
24 void inc(void)
25 {
26   inc_dec(up);
27 }

29 void dec(void)
30 {
31   inc_dec(down);
32 }
```

```
 9 static void inc_dec(direction dir)
10 {
11        switch (dir)
12        {
13            case up:
14                gbl_a++;
15                break;
16            case down:
17                gbl_a--;
18                break;
19            default:
20                break;
21        }
22 }
```

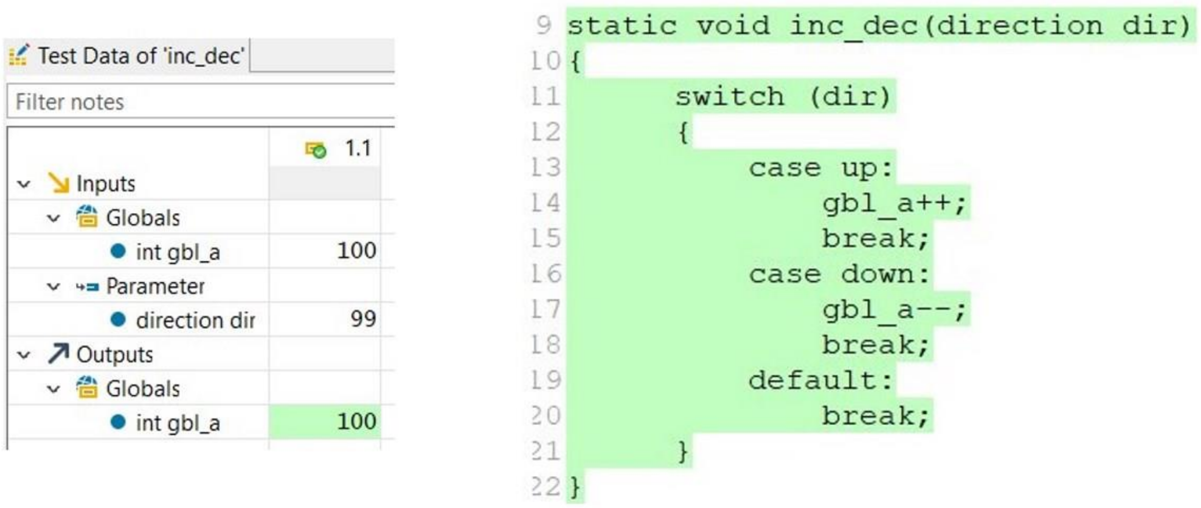
Fig. 8: Two test cases cover parts of the called unit inc_dec()

In the figure above one test case was executed for the test object inc(). This test case covers the test object inc() to 100% and also covers the case-label “up” (lines 13 to 15) in the switch instruction of the called test object inc_dec().

In the figure above another test case was executed for the test object dec(). This test case covers the test object dec() to 100% and also covers the case-label “down” (lines 16 to 18) in the switch instruction of the called test object inc_dec().

The label “default” (lines 19 and 20) in the switch instruction of the test object inc_dec() is not covered. It can neither be covered by a call from inc() nor by a call from dec().

Therefore, we need a third test case for the test object inc_dec(). This test case calls inc_dec() with an illegal parameter value, e.g. 99. This covers the label “default”.


Test Data of 'inc_dec'

Filter notes

	1.1
Inputs	
Globals	
int gbl_a	100
Parameter	
direction dir	99
Outputs	
Globals	
int gbl_a	100

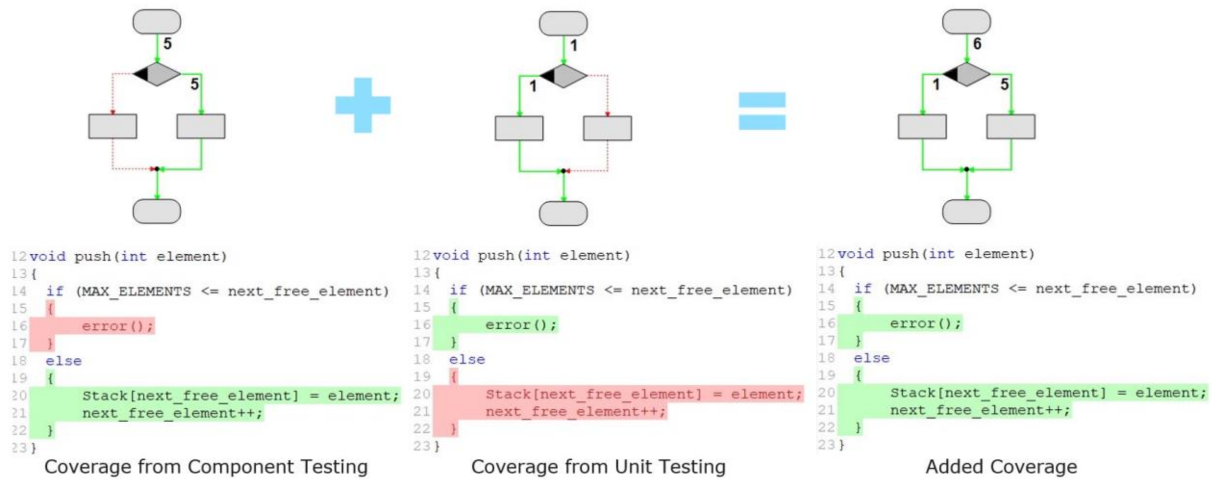
```
 9 static void inc_dec(direction dir)
10 {
11         switch (dir)
12         {
13             case up:
14                 gbl_a++;
15                 break;
16             case down:
17                 gbl_a--;
18                 break;
19             default:
20                 break;
21         }
22 }
```

Fig. 9: A third test case calls inc_dec() and covers the label “default”

It is not necessary to execute three test cases for inc_dec() to reach 100% branch coverage for inc_dec(), as it would have been the case without Hyper Coverage.

5.2 Adding Coverage from Component Testing and Unit Testing

It is also possible to add code coverage from component testing and unit testing. For instance, one might start testing by component testing and this tests only the normal behavior of the component under test, but not behavior under error conditions, e.g. the defensive code that might be present in the component. I.e. one does not reach 100% coverage for the component under test. The missing coverage can be “added” by unit testing one or more units in the component, thereby executing the hitherto untested code. So, 100% coverage for the component can be achieved. Also 100% coverage for the unit(s) can be achieved without having to execute the parts in the unit during unit testing that were already covered during component testing.



```
void push(int element)
{
  if (MAX_ELEMENTS <= next_free_element)
  {
      error();
  }
  else
  {
      Stack[next_free_element] = element;
      next_free_element++;
  }
}
```

```
void push(int element)
{
  if (MAX_ELEMENTS <= next_free_element)
  {
      error();
  }
  else
  {
      Stack[next_free_element] = element;
      next_free_element++;
  }
}
```

```
void push(int element)
{
  if (MAX_ELEMENTS <= next_free_element)
  {
      error();
  }
  else
  {
      Stack[next_free_element] = element;
      next_free_element++;
  }
}
```

Fig. 10: *Adding coverage from component/integration testing and unit testing*

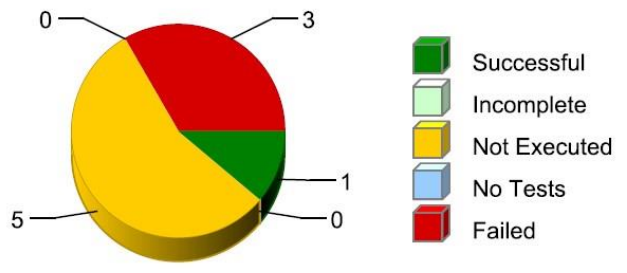
6 New Report: Test Summary Report

This report provides the current state of the test project based on source files and tasks.

Summary

Total Source Files:	9
Successful:	1
Incomplete:	0
Not Executed:	5
No Tests:	0
Failed:	3

Results per Source File



Total Test Objects:	19
Successful:	5
Incomplete:	0
Not Executed:	10
Failed:	4

Results per Test Object

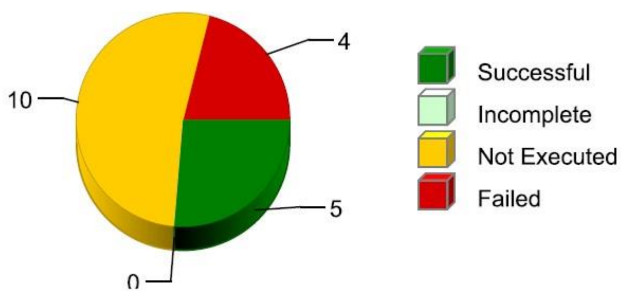
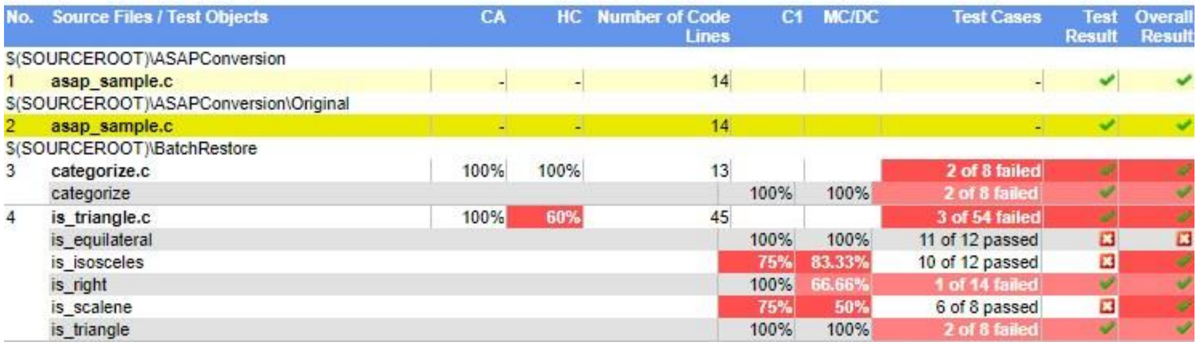
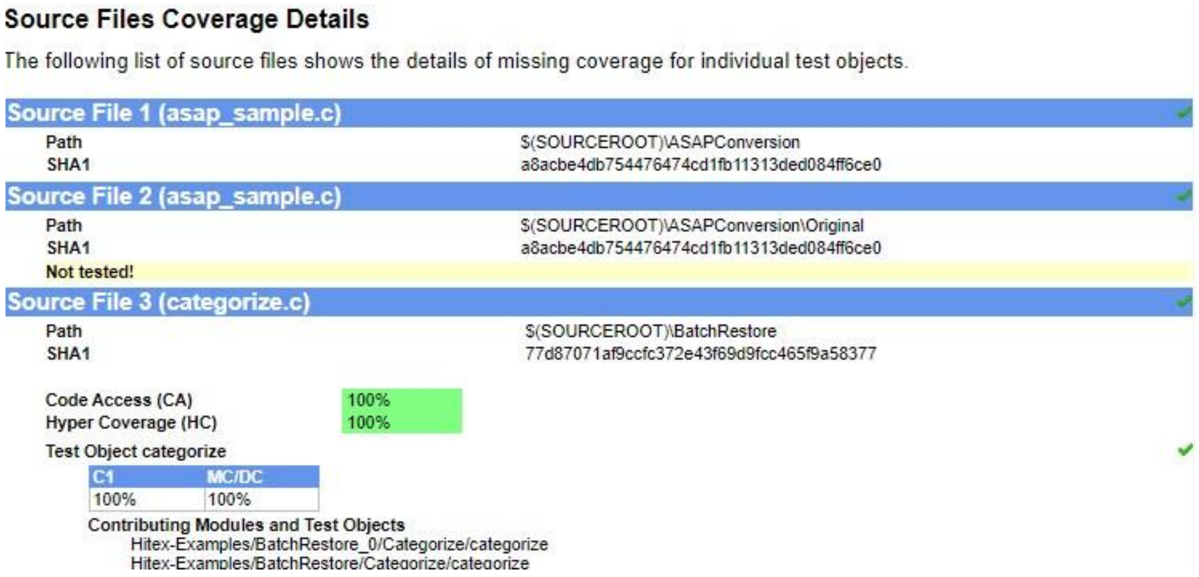


Fig. 11: *Excerpt form a Test Summary report*



No.	Source Files / Test Objects	CA	HC	Number of Code Lines	C1	MC/DC	Test Cases	Test Result	Overall Result
$(SOURCEROOT)\ASAPConversion									
1	**asap_sample.c**	-	-	14			-	✓	✓
$(SOURCEROOT)\ASAPConversion\Original									
2	**asap_sample.c**	-	-	14			-	✓	✓
$(SOURCEROOT)\BatchRestore									
3	categorize.c	100%	100%	13			2 of 8 failed	✓	✓
	categorize				100%	100%	2 of 8 failed	✓	✓
4	is_triangle.c	100%	60%	45			3 of 54 failed	✓	✓
	is_equilateral				100%	100%	11 of 12 passed	☒	☒
	is_isosceles				75%	83.33%	10 of 12 passed	☒	✓
	is_right				100%	66.66%	1 of 14 failed	✓	✓
	is_scalene				75%	50%	6 of 8 passed	☒	✓
	is_triangle				100%	100%	2 of 8 failed	✓	✓

Fig. 12: *Excerpt form a Test Summary report*



Source Files Coverage Details

The following list of source files shows the details of missing coverage for individual test objects.

Source File 1 (asap_sample.c) ✓

Path	$(SOURCEROOT)\ASAPConversion
SHA1	a8acbe4db754476474cd1fb11313ded084ff6ce0

Source File 2 (asap_sample.c) ✓

Path	$(SOURCEROOT)\ASAPConversion\Original
SHA1	a8acbe4db754476474cd1fb11313ded084ff6ce0

Not tested!

Source File 3 (categorize.c) ✓

Path	$(SOURCEROOT)\BatchRestore
SHA1	77d87071af9ccfc372e43f69d9fcc465f9a58377

Code Access (CA)	100%
Hyper Coverage (HC)	100%

Test Object categorize ✓

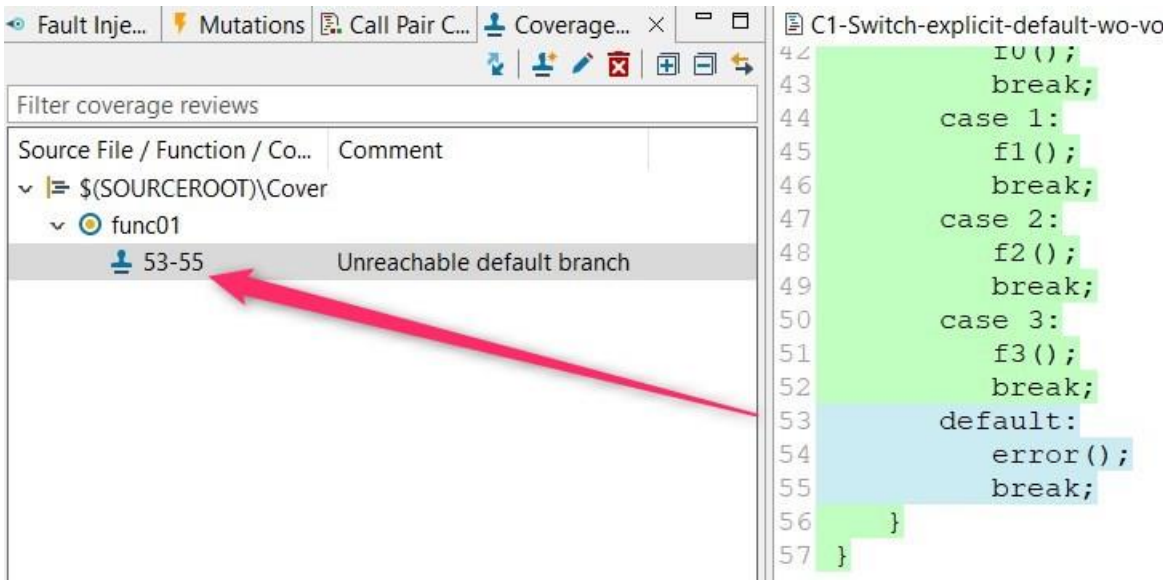
C1	MC/DC
100%	100%

Contributing Modules and Test Objects
Hitex-Examples/BatchRestore_0/Categorize/categorize
Hitex-Examples/BatchRestore/Categorize/categorize

Fig. 13: *Excerpt form a Test Summary report*

7 Coverage Reviews

If source code lines are not executed / covered during testing, those lines can be marked by comments (predefined or individual).

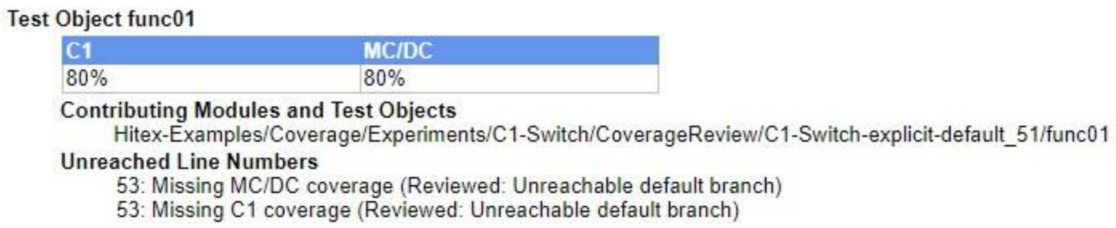


Source File / Function / Co...	Comment
$(SOURCEROOT)\Cover	
func01	
53-55	Unreachable default branch

```
42                 f0();
43                 break;
44             case 1:
45                 f1();
46                 break;
47             case 2:
48                 f2();
49                 break;
50             case 3:
51                 f3();
52                 break;
53             default:
54                 error();
55                 break;
56         }
57 }
```

Fig. 14: *An unreachable default label was reviewed and marked manually*

This information is transferred to the Test Summary report.



Test Object func01

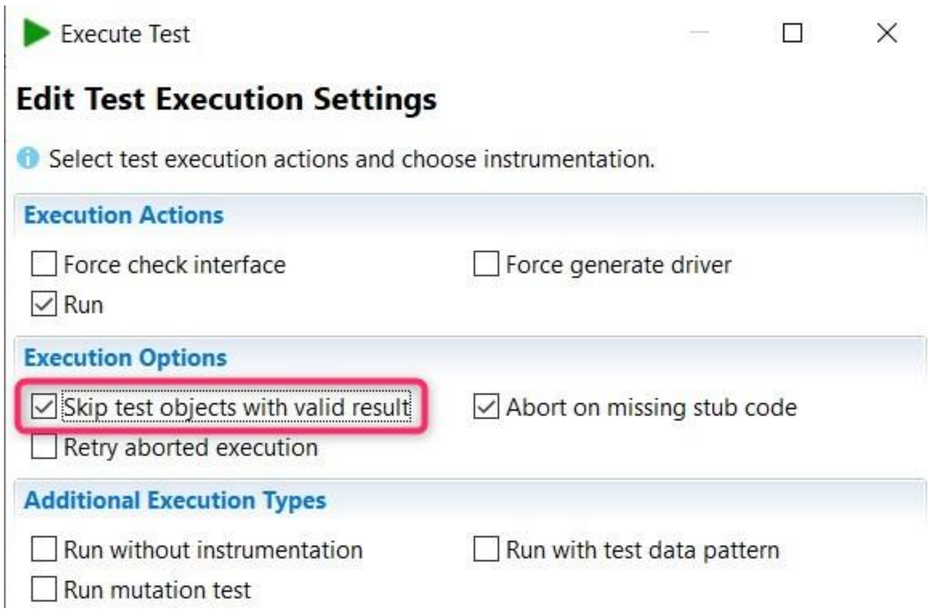
C1	MC/DC
80%	80%

Contributing Modules and Test Objects
Hitex-Examples/Coverage/Experiments/C1-Switch/CoverageReview/C1-Switch-explicit-default_51/func01
Unreached Line Numbers
53: Missing MC/DC coverage (Reviewed: Unreachable default branch)
53: Missing C1 coverage (Reviewed: Unreachable default branch)

Fig. 15: *Resulting effect in the Test Summary report*

8 Changed-based Testing

If a change in a source file (with several test objects in it) only affects a single test object, TESSY will only execute the test cases for the affected test object. This intelligent re-testing saves test execution time.



Execute Test

Edit Test Execution Settings

Select test execution actions and choose instrumentation.

Execution Actions

- [ ] Force check interface
- [ ] Force generate driver
- [x] Run

Execution Options

- [x] Skip test objects with valid result
- [x] Abort on missing stub code
- [ ] Retry aborted execution

Additional Execution Types

- [ ] Run without instrumentation
- [ ] Run with test data pattern
- [ ] Run mutation test

Fig. 16: *This execution setting executed only tests for changed test objects*

9 Improved Assignment of Test Data

An automatic reuse of test data for test objects with changed interface will now be done if:

- Variables were only added to or only removed from the interface.
- Parameters were only added to or only removed from the interface.
- The return type was changed from any type to void or vice versa.
- The scope of variables was changed.
- Extern function calls, which were not stubbed, were changed.

This feature is available since TESSY V5.1.8.

10 New Command: Debug Test

In the Test Project view:

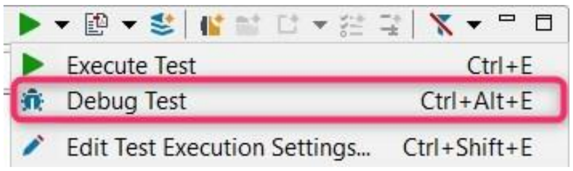


Fig. 17: The new command "Debug Test"

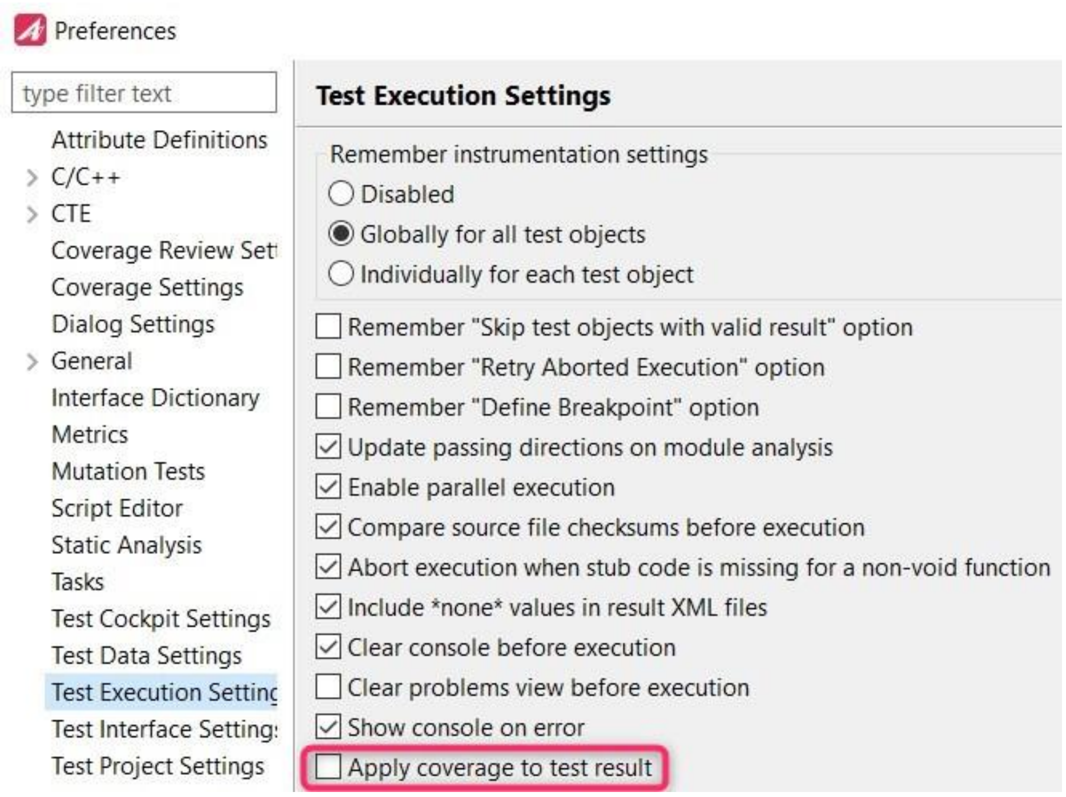
This is an abbreviation for executing the test with the test execution setting "Instrumentation" disabled and "Define breakpoint at test object" enabled.

This feature is available since TESSY V4.3.15.

11 Changed Behavior

11.1 Coverage in the Test Project view

The default for the treatment of the coverage result in the Test Project view was changed in TESSY V5.1. The coverage results will no longer be applied to the status icons of test collections, modules and test objects.



Preferences

type filter text

- Attribute Definitions
- C/C++
- CTE
- Coverage Review Set
- Coverage Settings
- Dialog Settings
- General
- Interface Dictionary
- Metrics
- Mutation Tests
- Script Editor
- Static Analysis
- Tasks
- Test Cockpit Settings
- Test Data Settings
- Test Execution Settinc
- Test Interface Setting:
- Test Project Settings

Test Execution Settings

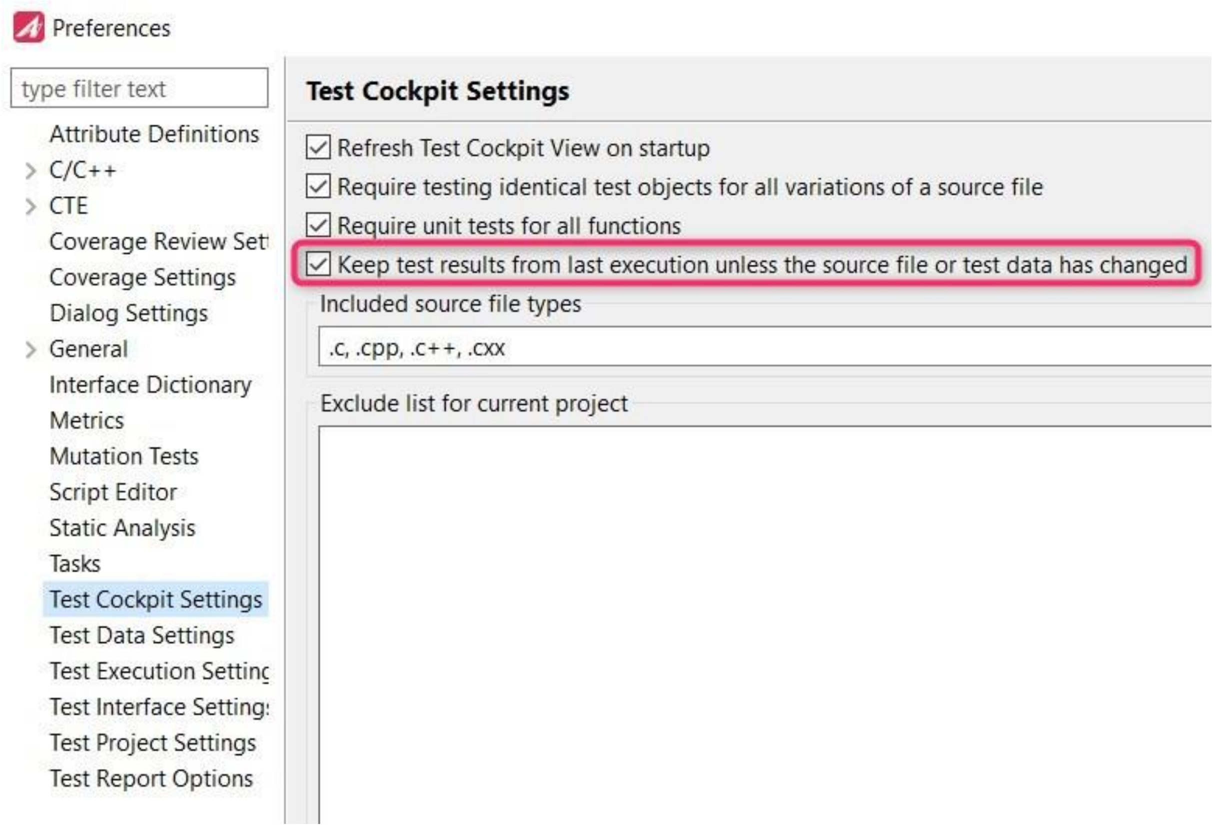
Remember instrumentation settings
- ○ Disabled
- ● Globally for all test objects
- ○ Individually for each test object

- ☐ Remember "Skip test objects with valid result" option
- ☐ Remember "Retry Aborted Execution" option
- ☐ Remember "Define Breakpoint" option
- ☑ Update passing directions on module analysis
- ☑ Enable parallel execution
- ☑ Compare source file checksums before execution
- ☑ Abort execution when stub code is missing for a non-void function
- ☑ Include *none* values in result XML files
- ☑ Clear console before execution
- ☐ Clear problems view before execution
- ☑ Show console on error
- ☐ Apply coverage to test result

Fig. 18: You can revert to the pre-V5.1 behavior in the Windows preferences

11.2 Effect of Module Analysis

Results for unchanged test objects with unchanged test data will be visible in the Test Cockpit view, even after a module analysis.



Preferences

type filter text

- Attribute Definitions
- C/C++
- CTE
- Coverage Review Sett
- Coverage Settings
- Dialog Settings
- General
- Interface Dictionary
- Metrics
- Mutation Tests
- Script Editor
- Static Analysis
- Tasks
- Test Cockpit Settings
- Test Data Settings
- Test Execution Setting
- Test Interface Setting:
- Test Project Settings
- Test Report Options

Test Cockpit Settings

- [✓] Refresh Test Cockpit View on startup
- [✓] Require testing identical test objects for all variations of a source file
- [✓] Require unit tests for all functions
- [✓] Keep test results from last execution unless the source file or test data has changed

Included source file types: .c, .cpp, .c++, .cxx

Exclude list for current project

Fig. 19 You can revert this behavior in the Windows preferences

12 The Author

Frank Büchner, Hitex GmbH, frank.buechner@hitex.de

Any comments or questions to this document are welcome.