

TESSY coverage measurements

Abstract

This application note describes the features and limits of the TESSY supported coverage measurements:

- Statement (C0) coverage
- Branch (C1) coverage
- Decision (DC) coverage
- MC/DC coverage
- MCC coverage
- Entry point coverage (EPC)
- Function coverage (FC)
- Call pair coverage (CPC)

Also the source file based measures are described:

- Code access (CA)
- Hyper coverage (HC)

Please note: The TESSY coverage instrumentation for MC/DC coverage has been extended in v3.1 to include measurement of switch statements as branch points which was previously covered only by the branch (C1) coverage.

Since TESSY v3.2 the handling of DC, MC/DC and MCC coverage measurement has been changed with respect to functions containing no decisions at all: Reaching the function entry point will now result in 100% coverage. Previously no DC, MC/DC or MCC coverage result was available for such functions.

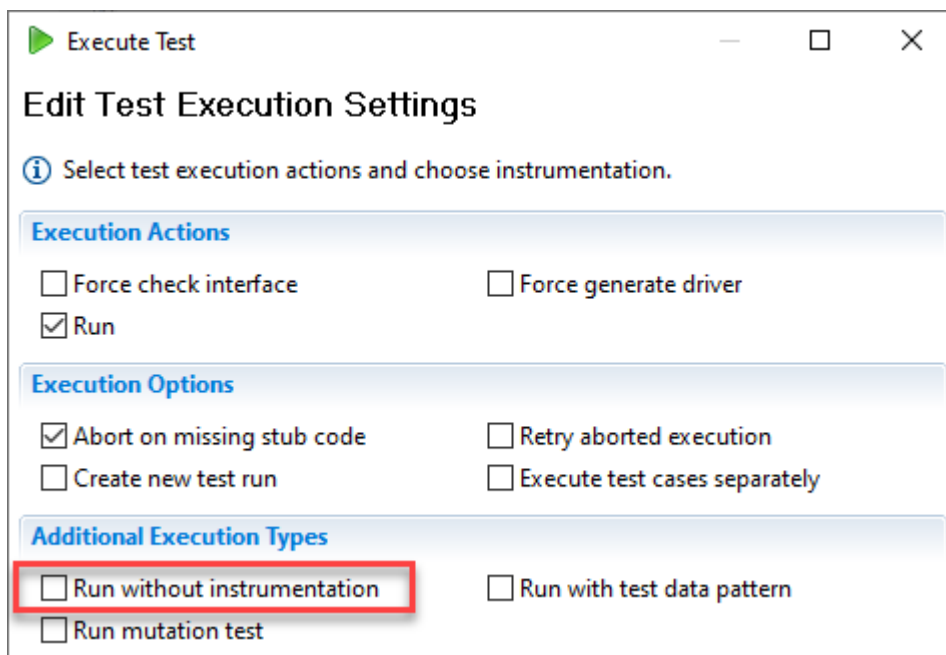
Table of Contents

Abstract	1
1 Introduction	3
2 Statement (C0) coverage	4
3 Branch (C1) coverage	4
3.1 If/else statement	4
3.2 Switch statement	5
3.2.1 Case without break	5
3.2.2 Default case	5
3.3 Called functions	5
3.4 The “?” operator	6
4 Decision (DC) coverage	6
4.1 Switch statement	6
4.1.1 Case without break	6
4.1.2 Default case	7
4.2 Boolean expressions within statements	7
4.3 The “?” operator	8
4.4 Using arithmetic operators to calculate boolean expressions	8
4.5 Functions without decisions	9
5 MC/DC coverage	10
5.1 Masking	10
5.2 Calculation of independence pairs	11
6 MCC coverage	11
7 Entry point (EPC) coverage	11
8 Function (FC) coverage	11
9 Call pair (CPC) coverage	11
10 Code access (CA)	12
11 Hyper coverage (HC)	12
12 Referenced documents	13

1 Introduction

The coverage measurements implemented by TESSY are based on the definition of the statement, branch, decision and condition/decision coverage measurements described within Liggesmeyer [1]. Following the discussions about DC and MC/DC coverage in CAST-10 [4], CAST-6 [5] and other position papers and comments with respect to RTCA DO178-B [2], the relevant details of the handling of statement, branch, decision and condition/decision coverage within TESSY have been documented within this document.

Code coverage measurement with TESSY is achieved by instrumenting the source code in question. I.e. test runs used to measure code coverage do not use the original, unchanged source code of the application. However, functional behavior should be identical, but this is not guaranteed. Therefore, you should separate the functional test (using unchanged source code) from the code coverage measurement tests (using instrumented source code). This can easily be achieved using the additional test execution type “Run without instrumentation” as shown below:



With this option selected, all given test cases will be executed once without instrumentation and once with the chosen coverage measurements. The actual results of both test executions will be checked against the expected results and both must yield a passed result.

Please note that the following features require a basic instrumentation to be applied in any case:

- Using static local variables
- Using the call trace feature
- Using the fault injection feature
- Stubbing of local functions/methods

2 Statement (C0) coverage

The statement coverage measured by TESSY is based on the branch (C1) coverage instrumentation without taking into account missing “else” branches of “if” statements.

For the coverage calculation, only control statements and branches will be counted and not individual statements or subsequent blocks. The statements at the beginning of a function until the first branch point will be left out of scope because the control flow guarantees that they will be reached. In the example shown below there will be two statements in total that need to be reached (i.e. the “if” control statement and the branch inside the “if” statement).

```
1 void test_function()
2 {
3     int a,b,c;
4
5     // Not counted statement block
6     a = 1;
7     b = 2;
8     c = 3;
9
10    // Statement 1
11    if (a == 1) {
12        // Branch (Statement 2)
13        b++;
14        c++;
15    }
16
17    // Not counted statement block
18    a--;
19    b--;
20    c--;
21 }
```

3 Branch (C1) coverage

3.1 If/else statement

Every “if” statement within a program will always cause two branches being recognized for code coverage measurement. In case of an omitted “else” statement, the missing “else” branch will nevertheless be counted as a separate branch for coverage calculation. The following code fragment has two branches:

```
22 if (A) {
23     // first branch
24 }
25 // second branch (missing else)
26
```

3.2 Switch statement

3.2.1 Case without break

If a switch contains a case label (e.g. case 1) without break statement before the next case label (e.g. case 2), then executing the respective case (i.e. case 1) would result in falling through to the next case label (i.e. case 2, and maybe further case labels). This execution of a fall-through case will be recognized by TESSY and it will be treated as follows:

- The coverage count of the originally targeted case label branch will be incremented.
- The coverage count of the fall-through case labels will not be changed.

This means that you need at least **three (3)** test cases to fully cover the example shown below (with two explicit branches and the default branch):

```
1 switch (x) {
2   case 1:
3     // first case without break
4   case 2:
5     // second case with break
6     break;
7   default:
8     // third (default) case
9     break;
10 }
```

3.2.2 Default case

The default case of a switch statement is always counted as separate branch, even if it is omitted in the code.

This means that you need at least **three (3)** test cases to fully cover the example shown below (with two explicit branches and the missing default branch):

```
1 switch (x) {
2   case 1:
3     // first case
4     break;
5   case 2:
6     // second case
7     break;
8 }
```

3.3 Called functions

The entry point into a function is only treated as a separate branch, if the function contains no other branches.

This helps in finding dead code like shown in the example below:

```
1 void called_function()
2 {
3     // third branch
4 }
5
6 void testobject()
7 {
8     if (x) {
9         // first branch
10        return;
11        called_function();
12    }
13    else {
14        // second branch
15        x++;
16    }
17 }
```

TESSY will find **three (3)** branches in this example with one of them (the third branch) uncovered.

3.4 The “?” operator

TESSY treats the expressions following the “?” operator as code branches. The behavior is the same as for the if/else statement, resulting in two branches for each “?” operator. The function within the following code fragment has two branches:

```
3 int is_less_than_five(int A)
4 {
5     return (A < 5 ? 1 /* first branch */ : 0 /* second branch */);
6 }
```

4 Decision (DC) coverage

4.1 Switch statement

The switch statement is treated as a branch point of the program with the case labels counted as possible outcomes (i.e. branches). All case labels need to be reached to achieve full decision coverage.

4.1.1 Case without break

If a switch contains a case label (e.g. case 1) without break statement before the next case label (e.g. case 2), then executing the respective case (i.e. case 1) would result in falling through to the next case label (i.e. case 2, and maybe further case labels). This execution of a fall-through case will be recognized by TESSY and it will be treated as follows:

- The coverage of the originally targeted case label branch will be marked as being covered.
- The coverage of the fall-through case labels will not be marked as being covered.

TESSY coverage measurements

This means that you need at least **three (3)** test cases to achieve full decision coverage for the example shown below (with two explicit branches and the default branch):

```
1 switch (x) {
2   case 1:
3     // first case without break
4   case 2:
5     // second case with break
6     break;
7   default:
8     // third (default) case
9     break;
10 }
```

4.1.2 Default case

The default case of a switch statement is always counted as separate branch, even if it is omitted in the code.

This means that you need at least **three (3)** test cases to achieve full decision coverage for the example shown below (with two explicit branches and the missing default branch):

```
1 switch (x) {
2   case 1:
3     // first case
4     break;
5   case 2:
6     // second case
7     break;
8 }
```

4.2 Boolean expressions within statements

Boolean expressions within statements will be instrumented by TESSY and they will be evaluated for coverage if they contain at least one boolean operator. The code below shows an example of a boolean expression in a statement which results in a decision with two atomic conditions. The decision coverage requires that the whole decision evaluates both to true and false.

```
3 C = A || B;
4
5 if (C) {
6   // some code
7 }
8 else {
9   // some code
10 }
```

The following code contains a relational expression without boolean operators in line three and a branch point with a decision that uses the result of the relational

expression in line five. The relational expression in line three will not be instrumented by TESSY but both possible results of this expression will be evaluated at its usage location within the decision in line five.

```
3 A = x > 0;
4
5 if (A) {
6     // some code
7 }
8 else {
9     // some code
10 }
```

The code below shows another relational expression within a return statement which will also not be instrumented by TESSY. The truth value being returned here may either be used within a branch point somewhere else within the software where the full decision coverage requirements also apply or it may not be used at all.

```
9 int is_valid(int x)
10 {
11     return (x > 0);
12 }
13
```

It is good practice to test this function with at least two different test cases to cover both possible outcomes. But this is more a functional requirement than a structural coverage requirement.

There is only one test case necessary to reach full decision coverage here because there is no branch point within this function.

4.3 The “?” operator

The decision of the “?” operator will be instrumented and evaluated for coverage. The function within the following code fragment has one decision with two atomic conditions: The decision coverage requires that the whole decision evaluates both to true and false.

```
9 int is_valid(int x)
10 {
11     return ((x > 0) && (x < 10) ? 1 : 0);
12 }
13
```

4.4 Using arithmetic operators to calculate boolean expressions

The C syntax allows using arithmetic operations to calculate a result that may itself be used as a decision.

Such code is seen as bad practice in programming. Coding rules shall prohibit the usage of such constructs and any violations should be detected during code reviews. TESSY will not instrument such code constructs and they will not be evaluated for coverage. Below is an example of such code where the arithmetic multiplication operator is used instead of the logical 'and' operator:

```
3 C = A * B;  
4  
5 if (C) {  
6     // some code  
7 }  
8 else {  
9     // some code  
10 }
```

4.5 Functions without decisions

Functions that contain no decisions at all will result in 100% coverage if the function entry point has been reached (i.e. if at least one test case has been executed).

5 MC/DC coverage

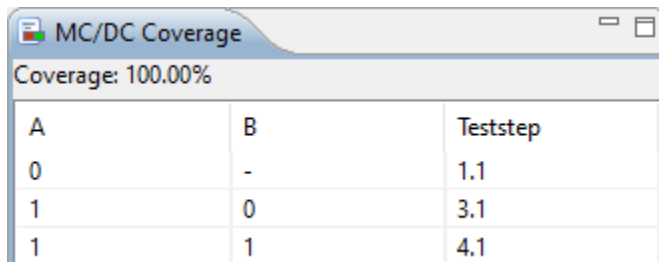
The above described rules for decision (DC) coverage apply as well for MC/DC coverage. Compared to decision coverage, the MC/DC coverage additionally requires evaluation of the atomic logical conditions of each decision. It must be shown that each condition independently affects the decision's outcome. Two test cases that show the independent effect of a condition within a decision are referred to as an independence pair [5].

There are two different approaches to show the independent effect: Unique cause and masking. TESSY implements the masking approach which is outlined below.

For technical reasons, the absolute number of atomic conditions of each decision is restricted to 31. Decisions with more than this number of atomic conditions will have no coverage information available.

5.1 Masking

Masking refers to the concept that specific inputs to a logic construct can hide the effect of other inputs to the construct [5]. Due to the short-circuit evaluation semantics of the C language, not all boolean conditions of a decision need to be evaluated when evaluating the decision outcome. Therefore, TESSY calculates the truth table for each decision by analysis of the internal logic of the decision and marks any conditions that will not be evaluated with a hyphen symbol. Below is the truth table for the decision “(A && B)” containing all possible inputs:



A	B	Teststep
0	-	1.1
1	0	3.1
1	1	4.1

Because B will not be evaluated any more if A is zero, B can have an arbitrary value (i.e. the effect of B is masked). The masking approach to MC/DC allows more than one input to change in an independence pair, as long as the condition of interest is shown to be the **only** condition that affects the value of the decision outcome [5].

In [5] it is also stated that “Both the unique-cause and masking approaches to MC/DC provide the same minimum tests of a logical operator in a decision. These minimum tests confirm that each condition independently affects the decision’s outcome”.

The TESSY implementation of MC/DC coverage measurement is applicable up to the highest security levels because “... research has shown that masking MC/DC also meets the intent of the MC/DC objective. Therefore, it is proposed that masking MC/DC be considered an acceptable method for meeting MC/DC by applicants striving to meet the objectives of DO-178B, Level A” [5].

5.2 Calculation of independence pairs

TESSY calculates all possible sets of independence pairs and finds out the best fit with respect to the executed test cases/steps. This ensures that you will always get the best possible coverage results for your given test cases.

A downside of this solution is that the required calculation of the best fit may take some time. For this reason, the calculation to find the best fit of all independence pairs is restricted to a number of 6 atomic conditions of each decision within your program. For decisions with more than this number of atomic conditions, the first calculated set of independence pairs will be used.

6 MCC coverage

All the above described rules for MC/DC coverage apply also for the MCC coverage. Only the last section 5.2 doesn't apply, because there is no independence pair calculation necessary for MCC coverage.

7 Entry point (EPC) coverage

This measurement applies to unit testing only and requires all functions of a module being executed at least once. If all test objects of a module are executed, you will reach full entry point coverage.

8 Function (FC) coverage

This measurement applies only to component testing and requires all called functions contained within a module being executed at least once.

9 Call pair (CPC) coverage

This measurement requires all call locations of functions or methods being reached at least once. Functions called via function pointers are ignored for the CPC coverage.

Other restrictions apply for C++ source code being tested:

- Class constructor calls are ignored
- Operators are ignored
- Overloaded method calls are collected and displayed under the same method name without taking the method signature into account.

10 Code access (CA)

The code access feature automatically detects hidden or untested code in all variants in the source code under test. The CA percentage is calculated on source file level. It requires analyzing the modules being defined for testing the source files. Therefore, the CA percentage can already be calculated after creating all needed modules for the planned tests but before any actual test is written.

After analysis of all modules, TESSY automatically detects any source code lines that are not accessed within any of the source code variations being tested with the existing modules (i.e. due to preprocessor directives hiding them within the preprocessed code). The CA percentage is the relation between all source code lines containing any code and the respective code lines being accessed within the preprocessed code by any of the available modules. Lines containing comments, defines or other preprocessor instructions are left out of scope.

11 Hyper coverage (HC)

The hyper coverage feature provides accumulation of coverage results across different tests and testing levels. The hyper coverage applies the normal coverage measurements (e.g. branch or MC/DC coverage) to determine which lines of the source code are fully covered. Each code line with any failed coverage result will be rated as failed.

The HC is calculated on source file level. The steps to calculate HC from the coverage measurement results are as follows:

1. All lines in the original source file that contain code, i.e. are not comments, defines or other preprocessor instructions, are determined.
2. The coverage results are used to determine all lines with coverage = 100%
=> Passed lines
3. The coverage results are used to determine all lines with coverage < 100%
=> Failed lines
4. If a function/method has < 100% coverage, then all remaining lines of this function/method for which there is no coverage information available are added to the failed lines (i.e. exactly the beginning of the function up to the first branch/condition and the last lines of a function after the last branch/condition).
5. All failed lines are now subtracted from the passed lines.

The HC percentage is the relation of the number of passed lines to the total number of code lines within the source file.

The green/red code coloring of the functions/methods in the code view of the coverage viewer also reflects the passed and failed lines. As long as a function/method has less than 100%, the lines not known by branch/condition positions and the first and last line of a function/method will also be marked as failed.

12 Referenced documents

- [1] Liggesmeyer P., “Software Qualität: Testen, Analysieren und Verifizieren von Software”, Heidelberg/Berlin, Spektrum Akademischer Verlag GmbH, 2002, ISBN 3-8274-1118-1.
- [2] RTCA, Inc. document DO-178B and EUROCAE document ED-12B, “Software Considerations in Airborne Systems and Equipment Certification”, dated December 1, 1992.
- [3] CAST-5 Paper, “Guidelines for Proposing Alternate Means of Compliance to DO-178B”, Completed June, 2000
- [4] CAST-10 Paper, “What is a “Decision” in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?”, Completed June, 2002
- [5] CAST-6 Paper, “Rationale for Accepting Masking MC/DC in Certification Projects”, Completed August, 2001