Is 100% Code Coverage Enough?

Nomenclature, measures and values of code coverage

> TESSY White Paper
  Author: Frank Büchner

Preface

In order to keep you up-to-date with the latest developments on our products, we provide White Papers containing additional topics, special hints, examples and detailed procedures etc.

For more information on the current software and hardware revisions as well as our update service, please visit www.hitex.de, www.hitex.co.uk or www.hitex.com.

Topic:	Version:
The objective of this paper is to define some commonly used code coverage measures and to discuss their strengths and weaknesses. Also the relation between the measures will be discussed. Small examples are used to illustrate some measures and to indicate common traps and pitfalls. Eventually, we will be able to rate the value of code coverage in general and what it means if we have reached 100% of a certain coverage measure.	2012-08-003

Author:

Frank Büchner

Dipl.-Inform.

Principal Engineer Software Quality

Hitex GmbH

Greschbachstr. 12

D-76229 Karlsruhe

Tel.: +49-721-9628-125

frank.buechner@hitex.de

Contents

1. Introduction

1.1. Motivation

Software quality improvement, customer demand, or standards like [DO-178B] or [IEC 61508] require or recommend the measurement of coverage based on the code structure.

Unfortunately, some confusion exists with respect to the nomenclature. Furthermore, the measures are sometimes hard to understand and sometimes the difference between two measures split a very fine hair.

Even if you have reached 100% by appropriate means, there may be still some flaws in the tested software, because code coverage is insensitive to some kind of problems.

1.1.1. Confusing Nomenclature

Confusion with respect to the nomenclature relates even to the name of the topic we want to discuss in the following. It is sometimes simply called "code coverage", but also "control-flow-based coverage" and "structure-based coverage" or "structure coverage". (We will adhere to the term "code coverage" in the following).

Sometimes people also take the term "white-box testing" synonymous to "determining code coverage", what at least neglects data coverage.

Confusion with respect to the nomenclature is continued with other terms related to code coverage. Different terms are used to denote exactly the same measure, e.g. the same measure is called "multiple condition coverage" in [LIGGESMEYER] and "branch condiiton combination testing" in [SPILLNER]. On the other hand, terms are ambiguous: E.g., statement coverage sometimes is denoted "C1" [BEIZER] and sometimes "C0" [SPILLNER].

1.1.2. Complex Measures

Most of the measures discussed have rather complicated definitions. Therefore, it is often hard to understand what a certain percentage of a measure exactly testifies and how different measures are related to one another.

1.1.3. The Hunt for a Percentage

In practice, an objective to reach a certain percentage of code coverage is sometimes achieved by inappropriate means. Sometimes, tests using random test data are executed, without verifying the outcome of such tests against an expected result. Furthermore, test data is created primarily based on the structure of the code under test and not on the specification of the functionality.

Even if you have reached 100% code coverage, can you rely on that being enough?

1.2. Objective of this Paper

The objective of this paper is to define some commonly used code coverage measures and to discuss their strengths and weaknesses. Also the relation between the measures will be discussed.

Small examples are used to illustrate some measures and to indicate common traps and pitfalls. Academic comprehensiveness is not the ambition. Exactness may be sacrificed for understandability and brevity.

Eventually, we will be able to rate the value of code coverage in general and what it means if we have reached 100% of a certain coverage measure.

2. Code Coverage Measures

Code coverage is based on the control-structure of a piece of software respectively the flow of control achieved by the execution of a test case for that piece of software. Code coverage counts the execution of items and relates this number to the total number of items in the piece of software in question. Such items are typically statements, branches, conditions, paths. These will be discussed in the following.
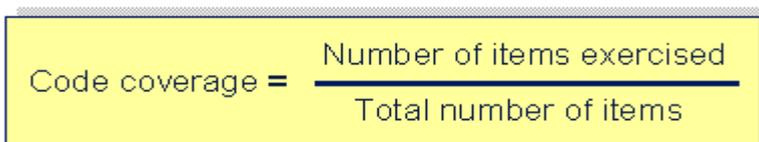
$$\text{Code coverage} = \frac{\text{Number of items exercised}}{\text{Total number of items}}$$

Fig. 1: Code coverage is a relation and is usually expressed in percent

2.1. Statement Coverage

Statement coverage ("Anweisungsüberdeckung") reports about execution of (executable) statements. Statements can be assembler statements or statements of the C programming language, etc.

Statement coverage is related to line coverage ("Zeilenüberdeckung"). In case each line holds only one executable statement (what is automatically the case with assembler programs), the equivalence is obvious. To have equivalence also if a source code line can hold several statements requires that all statements in that line are executed.

Statement coverage sometimes is denoted "C1" (e.g. in [CAST-10] and sometimes "C0" (e.g. in [THALLER] and in [SPILLNER]). Therefore, please use these identifiers with caution.

2.1.1. Example

Statement coverage is a weak measure. To illustrate this, we consider the following example:

```
√   int a = 0;
√   if (decision)
    {
√       a = 1;
    }
√   a = 1 / a;
```

Fig. 2: Code excerpt to illustrate the weakness of statement coverage

In the example above, we assume that all executable statements (respective lines) of the code excerpt were executed. (This shall be indicated by the tick marks in front of the executable statements.) Hence we have 100% statement coverage. The value of the variable "decision" obviously was not 0 during the execution in question, because the statement "a = 1;" was executed. This prevented a division by zero in the last line.

However, if a value of 0 would have been used for "decision", a division by zero would have been occurred in the last statement.

2.1.2. Appraisal of Statement Coverage

As seen in the example, even if 100% statement coverage is achieved, severe bugs still can be present. Therefore, please keep in mind what 100% statement coverage actually means: All statements were executed during the tests at least one time. Not more and not less.

We can assume, that 100% statement coverage is "better" (i.e. results in a higher software quality, whatever this may be) than, say 70% statement coverage, but the software quality achieved by 100% statement coverage is certainly not sufficient for a safety-critical project.

If you consider to apply statement coverage, to reach 100% is the minimal objective.
Statement coverage can be useful for detecting "dead code", i.e. code that cannot be executed at all.
Statement coverage can reveal missing test cases.

2.2. Branch Coverage

The term "branch coverage" ("Zweigüberdeckung") relates to the decisions in a program, where program execution can take one out of two possible branches. This is equivalent to the decision evaluating to both true and false.

The simplest example is an if-instruction which has a "then" branch and an "else" branch.
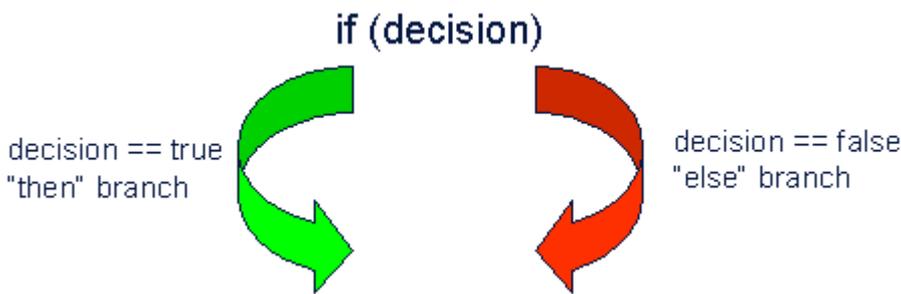
Fig. 3: A decision in an if-statement causes two branches

Branch coverage is often abbreviated as C1 respectively C1, e.g. in [LIGGESMEYER], [SPILLNER] and [TESSY], but other nomenclature may be used also, e.g. C2 in [BEIZER]. Therefore, please use these identifiers with caution.

2.2.1. An if- Statement Has Always Two Branches

Some people are astonished that the else-branch exists even if it is not represented in the source code, as shown in the example for statement coverage above (see Fig. 2 on p. 6).

We perform the same test as described above for the code excerpt depicted in Fig. 2 on p. 6, but measure branch coverage coverage instead of statement coverage.


```
1 |
  |    int a = 0;
  |    if (decision)
  |    {
1 |
  |        a = 1;
  |    }
****** 0 |
  |    a = 1 / a;
  |  }
```
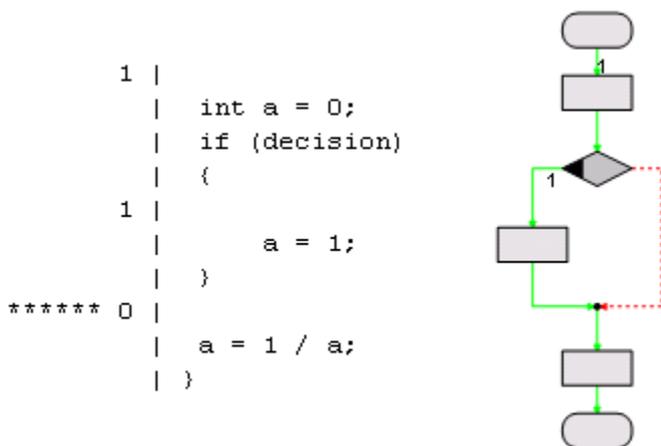
Fig. 4: The invisible else-branch is taken into account for branch coverage

The figure above depicts the branch coverage result of Tessy, a tool for module/unit testing of embedded software. More on Tessy can be found under [TESSY]. On the left hand side of the figure above, the printable branch coverage result is displayed. The numbers are each related to a branch and indicate the number of executions of that branch. The branch with zero executions is marked by asterisks. The right hand side of the figure above gives the flow-chart representation of the coverage information. The else-branch with zero executions is in dotted format and marked in red. Executed branches are in green. The branch coverage for the code excerpt is 50%.

It becomes obvious from the figure above that the "invisible" else-branch was not executed. This makes branch coverage a more valuable measure than statement coverage.

In the example, it becomes obvious that at least a second test case needs to be executed to achieve 100% branch coverage.

2.2.2. A Pathological Situation

In the C programming language, the decision in a switch statement selects one of several possible branches. This can be considered to be equivalent to a sequence of if-statements. However, sometimes astonishing things happen. Consider the following code snippet.

```c
int i;
for (i = 0; i < 2; i++)
{
    switch (i)
    {
        case 0:
            a = 600;
            break;
        case 1:
            a = 700;
            break;
    }
}
```

Fig. 5: Can 100% branch coverage be achieved?

Obviously, the for-loop is executed exactly two times: The first time the variable i has the value 0 and the code starting at the label "case 0" is executed; the second time the variable i has the value 1 and the code starting at the label "case 1" is executed. This should yield 100% branch coverage. But actually, 100% branch coverage cannot be achieved for this code snippet. The reason is the label "default", which is considered as an additional branch, even if it is not implemented explicitly.
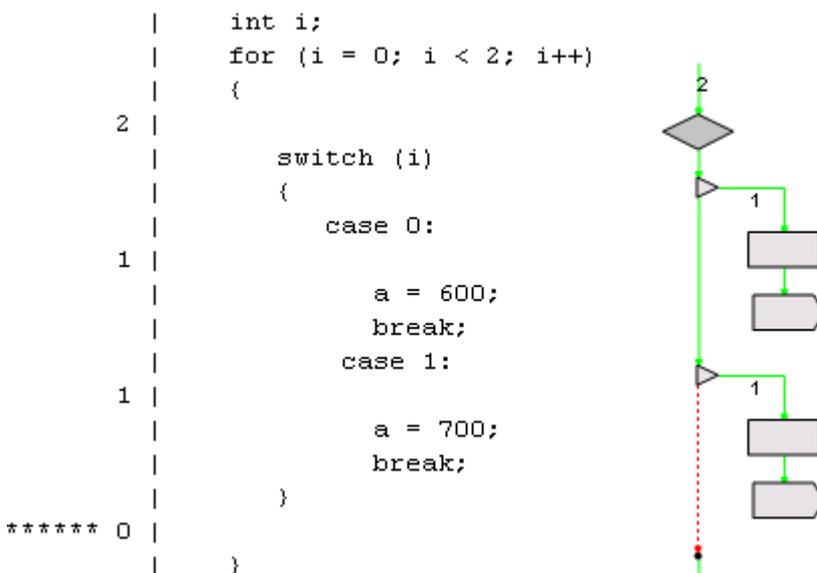

Fig. 6: The implicit default branch will never be excuted

```
|       int i;
|       for (i = 0; i < 2; i++)
|       {
2 |
|           switch (i)
|           {
|               case 0:
1 |
|                   a = 600;
|                   break;
|               case 1:
1 |
|                   a = 700;
|                   break;
|               default:
****** 0 |
|                   a = 800;
|                   break;
|           }
|       }
```


Fig. 7: Also the explicitly programmed default branch will never be executed

This example shows that you might not be able to achieve 100% branch coverage for a reasonable looking piece of code. The problem can be overcome be rearranging the code, e.g. by making the statement "a = 700;" the default case. But this might reduce comprehensibility of the code. If rearrangement is not possible or not desired, you might want to state why at this specific point a branch cannot be executed. This is called a justification. A justification is the "correct" solution to the problem.

2.2.3. Consecutive Swith Labels

```
switch (i)
{
    case 0:
    case 1:
    case 2:
        a = 700;
        break;
    default:
        a = 800;
        break;
}
```

Fig. 8: Which branch coverage is achieved executing a test case where (i==1)?

Does each label in a switch statement represent a branch, even if this branch does not contain any statements? In other words: , ,If a test case where (i==1) is executed: Is the branch belonging to the label "case 2" considered executed (what would result in 50% branch coverage) or not (what would result in 25% branch coverage)?

```
       |    switch (i)
       |    {
       |        case 0:
****** O |
       |        case 1:
     1 |
       |        case 2:
****** O |
       |            a = 700;
       |            break;
       |        default:
****** O |
       |            a = 800;
       |            break;
       |    }
```
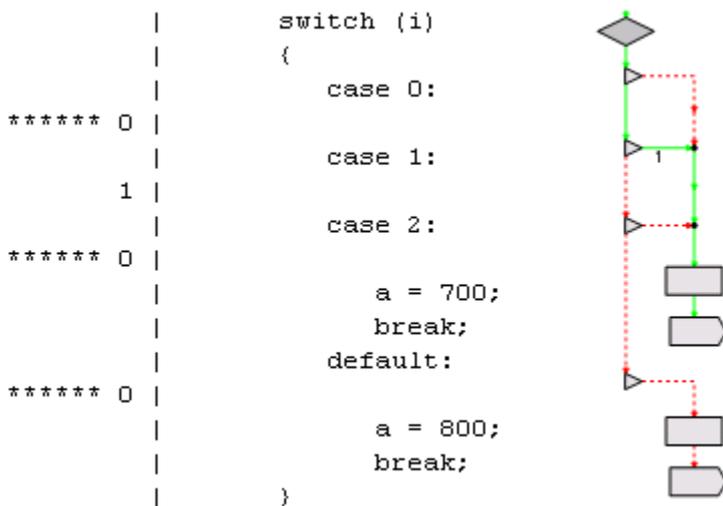
Fig. 9: A test case with (i==1) was executed

The figure above depicts the branch coverage reached by executing a single test case with (i==1). Tessy does not consider the branch belonging to the label "case 2" as executed. Therefore, the branch coverage is 25%.

If the branch belonging to the label "case 2" would be considered executed in the example above, 100% branch coverage can be reached without executing a test case with (i==2). However, such a test case could reveal an error.

```
switch (i)
{
    case 0:
    case 1:
    case 2:
        a = 1/(i-2);
        break;
    default:
        a = 800;
        break;
}
```

Fig. 10: A test case with (i==2) would reveal an error

2.2.4. Relation of Branch Coverage To Statement Coverage

If you have reached 100% branch coverage, you can deduct that you have also reached 100% statement coverage. I.e. branch coverage "includes" statement coverage. (The technical term for this is "to subsume").

However, if you have reached a certain amount of branch coverage, you cannot deduct that you have the same amount of statement coverage also, as one might assume at a first glance. E.g. from 50% branch coverage one cannot deduct to have reached also 50% statement coverage.

Consider the following code excerpt:

```
int a = 0;
if (decision)
     a++;
else
{
     a++;
     a++;
}
```

Fig. 11: Any single test case yields different results for branch coverage and statement coverage

If we execute a single test case, either with (decision == 1) or with (decision == 0), we will get 50% branch coverage in both cases. However, it is obvious, that in the first case (decision == 1) we will get lower statement coverage than in the second case (decision == 0).

2.2.5. Appraisal of Branch Coverage

Branch coverage subsumes statement coverage, i.e. branch coverage is more valuable than statement coverage. Therefore, branch coverage should be preferred over statement coverage whenever it is feasible. The initial goal should be to reach 100% branch coverage. If this goal is achieved, 100% statement coverage is achieved also automatically.

However, as we have seen, it is not always possible to reach 100% branch coverage.

The shortcoming of branch coverage is that it is insensitive on the structure of a decision. This shortcoming is remedied by condition coverage.

2.3. Decision Coverage

The definition of decision coverage ("Entscheidungsüberdeckung") in [CAST-10] is:

• Every point of entry and exit in the program has been invoked at least once, and

• every control statement (i.e. branchpoint) in the program has taken on all possible outcomes (i.e.branches) at least once, and

• every non-constant Boolean expression in the program has evaluated to both a True and a False result.

Because of the second item in the definition, decision coverage looks very similar to branch coverage. Actually, from 100% decision coverage you can deduct 100% branch coverage. But the opposite is not true, i.e. branch coverage does not subsume decision coverage. This is caused by the third item in the definition.

Branch coverage != Decision coverage

Fig. 12: You cannot deduct 100% decision coverage from 100% branch coverage

2.3.1. Example

Let's have a look on the following code snippet:

```
C = A && B;
E = C || D;
If (E)
      ret_val = 1;
else
      ret_val = 0;
```

Fig. 13: Code snippet to demonstrate the difference between branch coverage and decision coverage

We consider three input combinations for the above snippet.

Fig. 14: Input combinations for the above snippet

With input combination 1 and 2, we get E to evaluate to both true and false. This is equivalent to the fact that both the 'then' and the 'else' branch of the if statement is executed. Hence we have 100% branch coverage for the code snippet with input combination 1 and 2. However, we do have C evaluated only to false with input combination 1 and 2. Because C is a non-constant Boolean expression that did not evaluate to true with input combination 1 and 2, we do not have a 100% decision coverage with input combination 1 and 2. To achieve a 100% decision coverage, we additionally need input combination 3 which effects that C also evaluates to true.

2.4.   Condition Coverage

A decision ("Entscheidung") can be made up of one or more conditions ("Bedingungen"), which are combined by logical (i.e. Boolean) operators (AND, OR, NOT). A condition is a Boolean expression containing no Boolean operator. Conditions are also called atomic decisions.
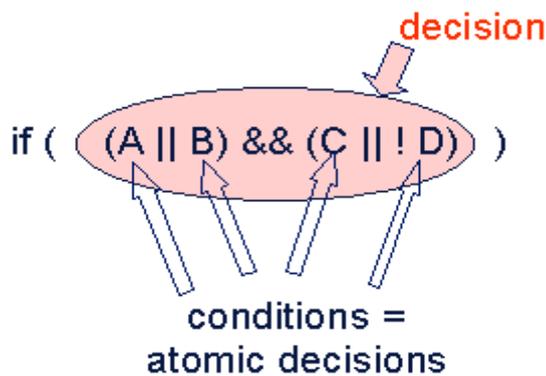
Fig. 15: A decision can be made up of conditions

Condition coverage measures to what extent conditions (i.e. atomic decisions) affect the outcome of a decision. A lot of different condition coverage measures exist.

2.4.1. Complete and Incomplete Evaluation

Decisions consisting of logical combined conditions raise the problem of complete and incomplete evaluation. Assume evaluation from left to right and as an example for a decision the decision "(A || B)". For a test case, in which A evaluates to true, it is not necessary to also evaluate B, because the outcome of the decision is true, regardless of the value of B.

In some programming languages, the compiler is allowed to stop evaluating a decision as soon as the outcome of the decision is known. This is called incomplete evaluation. Logical operators of such languages are sometimes attributed as short-circuit operators.

Languages, where the compiler is not allowed to stop evaluation, have complete evaluation. C, C++, and Java normally have incomplete evaluation. Pascal and Visual Basic are examples for languages with complete evaluation. Caution: Maybe this feature can be controlled by a compiler option.

2.4.2. Overview of Condition Coverage Measures

Several condition coverage ("Bedingungsüberdeckung") measures exist. We follow the nomenclature of [LIGGESMEYER].

2.4.2.1 Simple Condition Coverage

Simple condition coverage ("einfache Bedingungsüberdeckung") reports, if all conditions are evaluated at least one time to true and one time to false.

Simple condition coverage is called "branch condition testing" in [SPILLNER].

But 100% simple condition coverage does not necessarily yield 100% branch coverage. This is especially the case for languages with complete evaluation. To illustrate this, we use a code snippet in Pascal, a language featuring complete evaluation.

```
√   IF A AND B THEN
0        ...
    ELSE
√        ...
```

	A	B	A and B	
1	f	f	F	
2	f	t	F	√
3	t	f	F	√
4	t	t	T	

Fig. 16: A code snippet in Pascal and the four possible test cases (complete evaluation)

On the left hand side of Fig. 16, a code snippet in Pascal is depicted, where the decision (A and B) controls if the then-branch or the else branch is taken. On the right hand side of Fig. 16 the four (!) possible test cases are listed. If we execute test case 2 and test case 3, A gets both the value true (in test case 2) and false (in test case 3). Also B gets both the value true (in test case 2) and false (in test case 3). This results in 100% simple condition coverage. However, the overall outcome of the decision (A and B) is false both for test case 2 and for test case 3. Therefore, both test cases execute the else-branch; the then-branch is not executed. This results in only 50% branch coverage. Hence the example proves that in languages with complete evaluation you cannot deduce from 100% simple condition coverage to have 100% branch coverage also.

In languages with incomplete evaluation, e.g. in C, only three test cases are possible.

```
√  if (A && B)
   {
√       ...
   }
   else
   {
√       ...
   }
```

	A	B	A && B	
I	f	-	F	√
II	t	f	F	√
III	t	t	T	√

Fig. 17: A code snippet in C and the three possible test cases (incomplete evaluation)

We now assume a language with incomplete evaluation like C, and assume evaluation from left to right. We consider the decision (A && B) from the figure above. As soon as it is known that A evaluates to false, it is also known that (A && B) evaluates to false also, regardless of the value of B. Therefore, test case 1 and test case 2 from Fig. 16 have to be combined to form test case I in the Fig. 17. B is not evaluated, i.e. you cannot assume a certain value for B. This is indicated by the '-' sign in test case I. Test case II respectively III from Fig. 17 is identical to test case 3 respectively 4 from Fig. 16. You need test case II and III to have true and false for B. But A is true both in test case II and III. Therefore, you need test case I to have false for A. All in all, you need all three test cases to get 100% simple condition coverage for (A && B). The three test cases execute both branches; hence we have 100% branch coverage.

2.4.2.2 Condition / Decision Coverage
Because 100% simple condition coverage does not yield 100% branch coverage in any case, an additional measure is introduced, which achieves this goal by definition. 100% condition / decision

coverage ("Bedingungs-/Entscheidungsüberdeckung") is reached if both 100% simple condition coverage and 100% branch coverage is reached.

This is only relevant for languages with complete evaluation.

2.4.2.3 Minimal Multiple Condition Coverage

Minimal multiple condition coverage ("Mehrfach-Bedingungsüberdeckungstest") requires that

(1)   all atomic conditions are evaluated to both true and false, and

(2)   all compound conditions are evaluated to both true and false, and

(3)   the whole decision evaluates to both true and false.
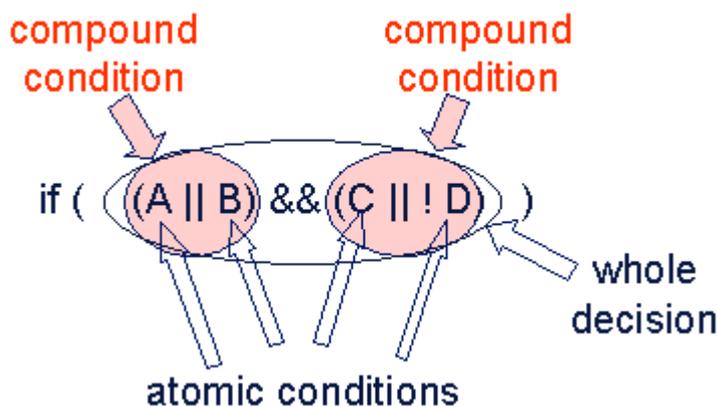
Fig. 18: A decision can be made up of atomic and compound conditions

Obviously minimal multiple condition coverage tests more intensive than simple condition coverage, because simple condition coverage requires only (1) of the above list. However, more test cases are required to reach 100% minimal multiple condition coverage than for 100% simple condition coverage.

Minimal multiple condition coverage subsumes condition / decision coverage, because (1) of the list above subsumes simple condition coverage and (3) from the list above subsumes decision coverage.

Especially with complete evaluation, minimal multiple condition coverage does not always take the structure of a decision into account. Languages with incomplete evaluation are better suited for minimal multiple condition coverage.

Minimal multiple condition coverage is called "modified branch condition decision testing" in [SPILLNER].

2.4.2.4 Modified Condition / Decision Coverage

One of the best known measures of condition coverage is modified condition / decision coverage ("Modifizierte Bedingungs-/Entscheidungsüberdeckung"), often abbreviated to "MC/DC" or MCDC.

MC/DC is defined as follows:

Each condition (= atomic decision) of a decision requires a pair of test cases, that

(1)   differs in the Boolean value for that condition, and

(2)   has the same Boolean value for all other conditions, and

(3)   produces true and false in the outcome of the decision.

The definition of MC/DC in [DO-178B] is:
- Every point of entry and exit in the program has been invoked at least once,
- every condition in a decision in the program has taken all possible outcomes at least once,
- every decision in the program has taken all possible outcomes at least once, and
- each condition in a decision has been shown to independently affect that decision's outcome.
- A condition is shown to independently affect a decisions outcome by varying just that condition while holding fixed all other possible conditions.

MC/DC is required by [DO-178B] or software on level A of criticality.

If the decision is made up of n conditions, a set of test cases with pairs of test cases for each condition consists of at least (n+1) test cases. All test cases of such a set need to be executed to achieve 100% MC/DC. It is possible that different sets of test cases exist, that yield 100% MC/DC when executed.

As a first example, we consider the decision (A || B) && (C || D), for a language with complete evaluation. There are 16 combinations for A, B, C, and D.

	A	B	C	D	A or B	C or D	(A or B) and (C or D)
1	f	f	f	f			
2	f	f	f	t	f	t	f
3	f	f	t	f			
4	f	f	t	t			
5	f	t	f	f			
6	f	t	f	t	t	t	t
7	f	t	t	f			
8	f	t	t	t			
9	t	f	f	f	t	f	f
10	t	f	f	t	t	t	t
11	t	f	t	f	t	t	t
12	t	f	t	t			
13	t	t	f	f			
14	t	t	f	t			
15	t	t	t	f			
16	t	t	t	t			

A: 2+10
B: 2+6
C: 9+11
D: 9+10

Fig. 19: The truth table and a set of pairs for 100% M/DC (complete evaluation)

T E.g., the pair for condition D is made up by test case 9 and 10. Test case 9 and 10 build a valid pair because

(1)   they differ in the value for the condition in question (D is false in test case 9 and D is true in test case 10); furthermore,

(2)   the value of the other three conditions (A, B, and C) are identical for both test cases; and

(3)   the overall outcome for the decision differs, i.e. ((A || B) && (C || D)) is false in test case 9 and true in test case 10.

As a second example, we take the same decision, but with incomplete evaluation.he figure above shows the pairs of test cases for each condition.

	A	B	C	D	A || B	C || D	(A || B) && (C || D)
I	f	f	-	-	f		f
II	f	t	f	f	t	f	f
III	f	t	f	t	t	t	t
IV	f	t	t	-	t	t	t
V	t	-	f	f	t	f	f
VI	t	-	f	t	t	t	t
VII	t	-	t	-	t	t	t

A: I + VII
B: I + IV
C: II + IV
D: II + III

Fig. 20: The truth table and a set of pairs for 100% M/DC (incomplete evaluation)

With incomplete evaluation, the number of possible test cases is reduced. For instance, the first four test cases from Fig. 19 are combined to the first test case in Fig. 20. If a condition is not evaluated, this is indicated by the minus sign in the truth table.

Still we need a pair of test cases for each condition. If a condition is not evaluated, it can be used as both true and false when building pairs. E.g. in test case IV, the value for D is not evaluated. With respect to D, this allows the combination of test case IV with any other test case to form a pair for one of the other three conditions.

A pair of test cases for each condition is shown in Fig. 20. Actually, test case IV is used in the pair for B and in the pair for C.

The set of pairs shown in the figure above is not the only set of test cases that yield 100% MC/DC. The figure below shows a different set of pairs.

	A	B	C	D	A || B	C || D	(A || B) && (C || D)
I	f	f	-	-	f		f
II	f	t	f	f	t	f	f
III	f	t	f	t	t	t	t
IV	f	t	t	-	t	t	t
V	t	-	f	f	t	f	f
VI	t	-	f	t	t	t	t
VII	t	-	t	-	t	t	t

A: I + VI
B: I + III
C: V + VII
D: V + VI

Fig. 21: Another set of pairs of test cases that yield 100% MC/DC for the decision

MCC-Coverage / MC/DC-Coverage				
Coverage: 100.00%				
A	B	C	D	Teststep
0	0	-	-	1.1
0	1	0	0	2.1
0	1	0	1	3.1
0	1	1	-	4.1
1	-	1	-	7.1

MCC-Coverage / MC/DC-Coverage				
Coverage: 100.00%				
A	B	C	D	Teststep
0	0	-	-	1.1
0	1	0	1	3.1
1	-	0	0	5.1
1	-	0	1	6.1
1	-	1	-	7.1

Fig. 22: Tessy determines 100% MC/DC for both sets

It is practically not possible to determine manually, if a set of test cases yield 100% MC/DC. Therefore, appropriate tools should be used. It is a plus if such a tool determines the lacking test cases for 100% MC/DC, if 100% were not achieved by the already executed test cases.

MCC-Coverage / MC/DC-Coverage				
Coverage: 60.00%				
A	B	C	D	Teststep
0	0	-	-	1.1
0	1	0	1	(red)
1	-	0	0	5.1
1	-	0	1	6.1
1	-	1	-	(red)

Fig. 23: Tessy shows which input combination was tested by which test case and reveals the combinations that still need to be exercised to get 100% MC/DC

2.4.2.5 Multiple Condition Coverage

Multiple Condition Coverage (MCC, "Mehrfach-Bedingungsüberdeckung") measures, if all possible combinations of true and false for all conditions in a decision were tested.

In other words, the test cases are given by the truth table of the conditions.

MCC is called "branch condition combination testing" in [SPILLNER].

For a decision with n conditions, 2n test cases are required to get 100% MCC in case of complete evaluation. In case of complete evaluation, the 16 combinations / test cases for the decision "(A || B) && (C || D)" are given by Fig. 19.

However, as discussed above, in case of incomplete evaluation, only 7 out of the 16 combinations exist. These are listed in Fig. 20 resp. Fig. 21. All 7 combinations need to be tested to get 100% MCC. These are two combinations more than for 100% MC/DC.

Fig. 24: Tessy marks which input combinations were already tested and indicates the combinations that still need to be exercised to get 100% MCC

2.4.3. More Examples of Condition Coverage

2.4.3.1 Oder Matters!

Sometimes it is important how a decision is written. Below are two decisions that are logically equivalent, but differ in the order of the conditions.

D1	D2
((a==4) || (a==5)) && b	b && ((a==4) || (a==5))

Fig. 25 What is the difference between these two decisions with respect to code coverage?

D1		D2	

D1

	A (a==4)	B (a==5)	C b	(A || B) && C ((a==4) || (a==5)) && b
1	f	f	f	
2	f	f	t	
3	f	t	f	
4	f	t	t	
5	t	f	f	
6	t	f	t	
7	t	t	f	
8	t	t	t	

D2

	C b	A (a==4)	B (a==5)	C && (A || B) b && ((a==4) || (a==5))
1	f	f	f	
2	f	f	t	
3	f	t	f	
4	f	t	t	
5	t	f	f	
6	t	f	t	
7	t	t	f	
8	t	t	t	

Fig. 26: All values for input combinations

Regarding D1:

- Input combination 7 and 8 are both logically not possible, because both A and B can not be true at the same time.
- Due to incomplete evaluation, input combination 1 and 2 can be combined, because if A is false and if B is also false, the outcome of the decision is false, regardless of the value of C.
- Due to incomplete evaluation, in input combination 5 and 6, B is not evaluated, because A is true for both input combinations, and therefore (A \|\| B) is true, regardless of the value of B.

- The remaining 5 actually possible input combinations are shown in the Fig. 27 (left hand side).

Regarding D2:
- Input combination 4 and 8 are both logically not possible, because both A and B can not be true at the same time.
- Due to incomplete evaluation, input combination 1 to 4 (respectively 1 to 3) can be combined, because if C is false, the outcome of the decision is false, regardless of the values of A and B.
- Due to incomplete evaluation, in input combination 7, B is not evaluated, because A is true, and therefore (A || B) is true, regardless of the value of B.
- Only 4 input combinations are possible! The remaining input combinations are shown in Fig. 27 (right hand side).

	A (a==4)	B (a==5)	C b	(A || B) && C ((a==4) || (a==5)) && b
I	f	f	-	f
II	f	t	f	f
III	f	t	t	t
IV	t	-	f	f
V	t	-	t	t

	C b	A (a==4)	B (a==5)	C&& (A || B) b && ((a==4) || (a==5))
I	f	-	-	f
II	t	f	f	f
III	t	f	t	t
IV	t	t	-	t

Fig. 27: The actually possible input combinations

Considering MC/DC

Both decisions consist of three atomic conditions (A, B, and C). Therefore, a set of test cases from which the necessary pairs for MC/DC can be built, consists of 4 test cases.

For D1, even two sets can be built. For D2, the (one and only) set is made up of all the four actually possible test cases.

D1	D2

Set 1:   A: I + V
 	
            B: I + III
 	
            C: IV + V
 	
Set 2:   A: I + V
 	
            B: I + III
 	
            C: II + III
	Set 1: A: I + III or I + IV
	
	            B: II + IV
	
	            C: II + III

Fig. 28: The pairs of test cases for each condition

For D2, two different pairs are possible for A. But regardless which pair is used, all four possible test cases are needed for the set.

Fig. 29

D1

Set 1:

MC/DC-Coverage | MCC-Coverage

Coverage: 100.00%

(a==4)	(a==5)	b	Teststep
0	0	-	1.1
0	1	1	3.1
1	-	0	4.1
1	-	1	5.1

Set 2:

MC/DC-Coverage | MCC-Coverage

Coverage: 100.00%

(a==4)	(a==5)	b	Teststep
0	0	-	1.1
0	1	0	2.1
0	1	1	3.1
1	-	1	5.1

D2

Set 1:

MC/DC-Coverage | MCC-Coverage

Coverage: 100.00%

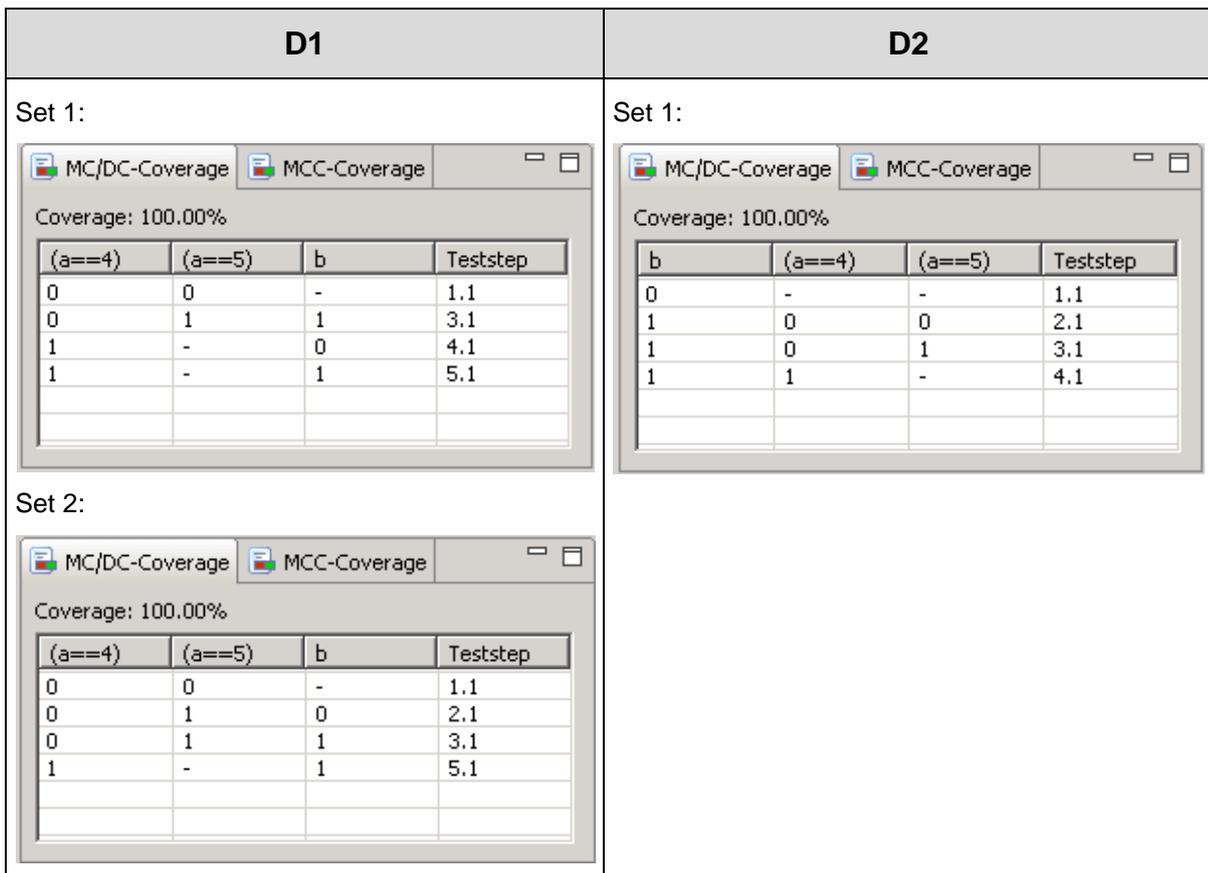| b	(a==4)	(a==5)	Teststep
0	-	-	1.1
1	0	0	2.1
1	0	1	3.1
1	1	-	4.1

Fig. 29: The test cases needed for 100% MC/DC

Considering MCC

To achieve 100% MCC, all actually possible test cases must be executed. Hence, D1 requires 5 test case executions, whereas D2 requires only 4 test cases (because there are only 4 actually possible test cases).

D1

MC/DC-Coverage | **MCC-Coverage**

Coverage: 80.00%

(a==4)	(a==5)	b	Reached
0	0	-	X
0	1	0	X
0	1	1	X
1	-	0	*(red)*
1	-	1	X

D2

MC/DC-Coverage | **MCC-Coverage**

Coverage: 75.00%

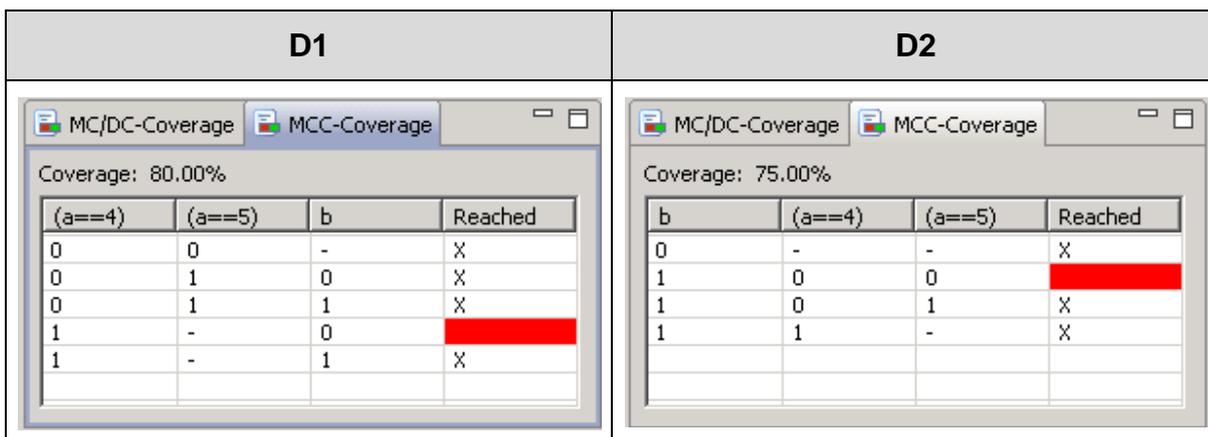| b	(a==4)	(a==5)	Reached
0	-	-	X
1	0	0	*(red)*
1	0	1	X
1	1	-	X

Fig. 30: Different number of test cases needed for 100% MCC

The figure above shows that 4 test cases are not enough to reach 100% MCC for D1; and 3 test cases are not enough to reach 100% MCC for D2. The test case needed in addition to reach 100% for D1 respectively D2 is marked in red.

Conclusion

The example at hand shows how the order in which a decision is written affects the number of actually possible test cases, what in turn not only influences MC/DC and MCC, but also the (average) execution speed. If one assumes equal distribution of the input values, the compiler needs to evaluate fewer conditions for D2 compared to D1.

2.4.3.2 Style Matters!

Sometimes it is important how a decision is programmed, e.g. how many instructions are used. Below are two code snippets that are functionally equivalent, but differ in the programming style.

S1	S2
| ```
if ((A || B) && (C || D))
    r = 1;
else
    r = 0;
``` | ```
if (A)
    if (C)
        r = 1;
    else if (D)
        r = 1;
    else
        r = 0;
else if (B)
    if (C)
        r = 1;
    else if (D)
        r = 1;
    else
        r = 0;
else
    r = 0;
``` |

Fig. 31: What is the difference between these two code snippets with respect to code coverage?

S2 resembles the implementation of the decision ((A || B) && (C || D) in most assembler languages and hence also in the binary object format. (It goes without saying that some braces could be added to S2 to improve understandability.)

However, for all possible input combinations, S1 and S2 assign the same value to r, i.e. the code snippets are functionally equivalent.

We know from above, that in case of incomplete evaluation, only 7 test cases are possible. For your convenience, these are again depicted in the following figure:

	A	B	C	D	A || B	C || D	(A || B) && (C || D)
I	f	f	-	-	f		f
II	f	t	f	f	t	f	f
III	f	t	f	t	t	t	t
IV	f	t	t	-	t	t	t
V	t	-	f	f	t	f	f
VI	t	-	f	t	t	t	t
VII	t	-	t	-	t	t	t

Fig. 32: The 7 actually possible test cases for ((A || B) && (C || D))

By exercising only two of the test cases from Fig. 32, we get 100% branch coverage. We need a pair of test cases with different outcome for the whole decision, e.g. test case II and III.
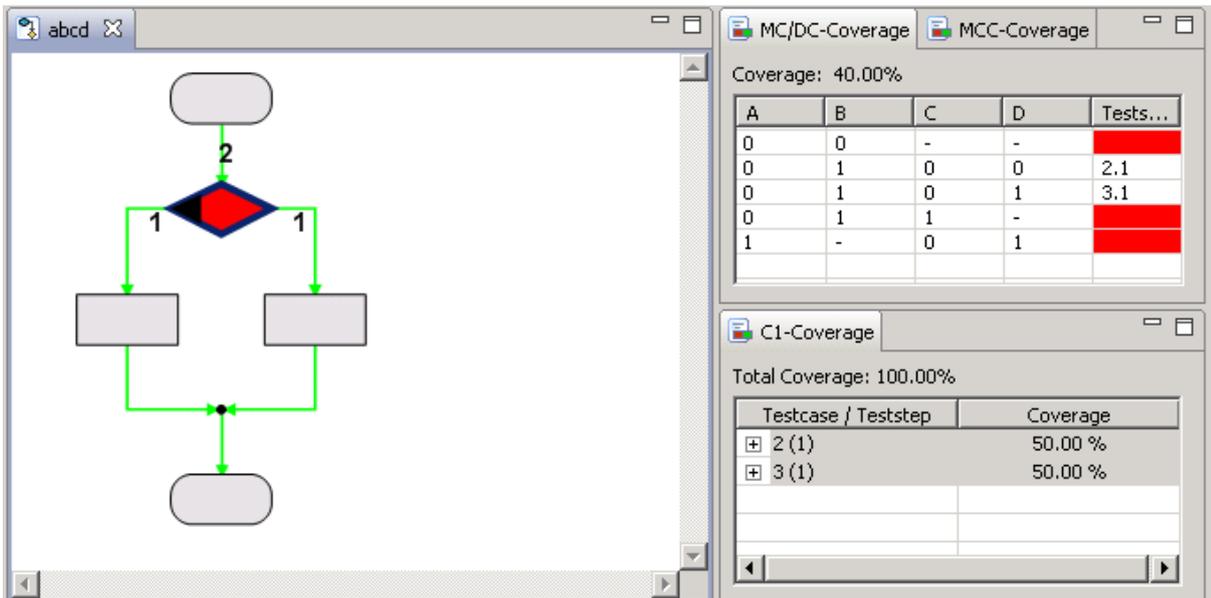

Fig. 33: Test cases II and III result in 100% branch coverage for S1

However, the two test cases neither do reach 100% MC/DC nor 100% MCC. According to the discussions above, at least 5 test cases are needed for 100% MC/DC, because S1 consists of 4 (atomic) conditions. To reach 100% MCC, all 7 possible test cases need to be executed.

To reach 100% branch coverage for S2, it turns out that all 7 test cases need to be executed. The pair above (II + III), which results in 100% branch coverage for S1, does only result in 41% branch coverage for S2.
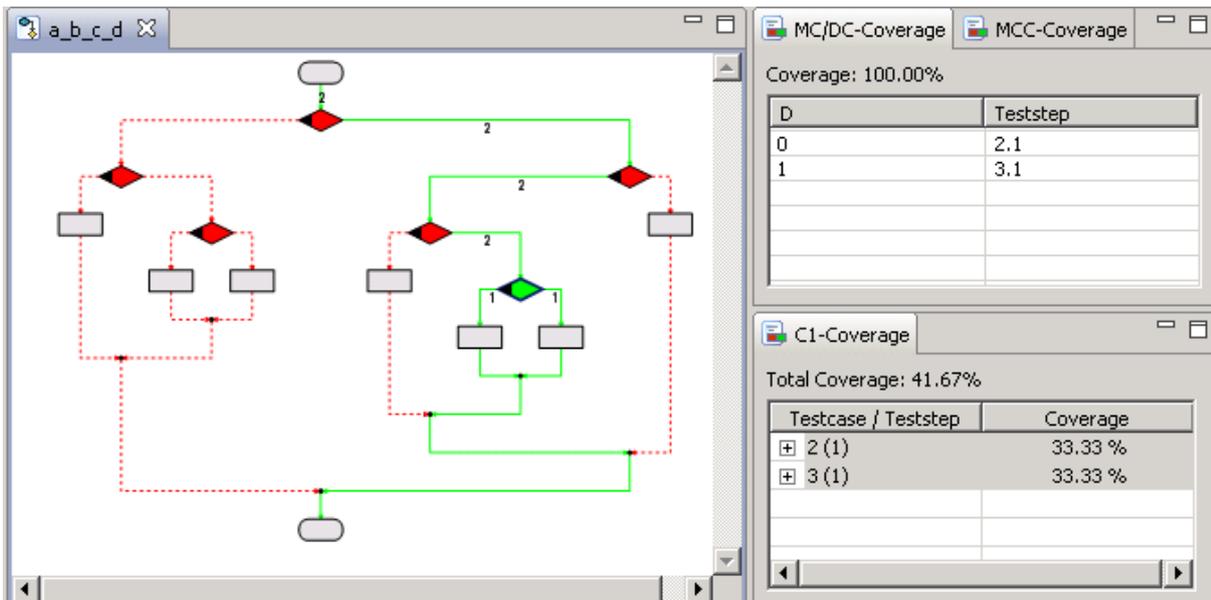
Fig. 34: Test cases II and III do not result in 100% branch coverage for S2

Since the decision in each if-instruction is an atomic condition, if 100% branch coverage is reached for S2, also 100% MC/DC coverage and also 100% MCC coverage is reached.
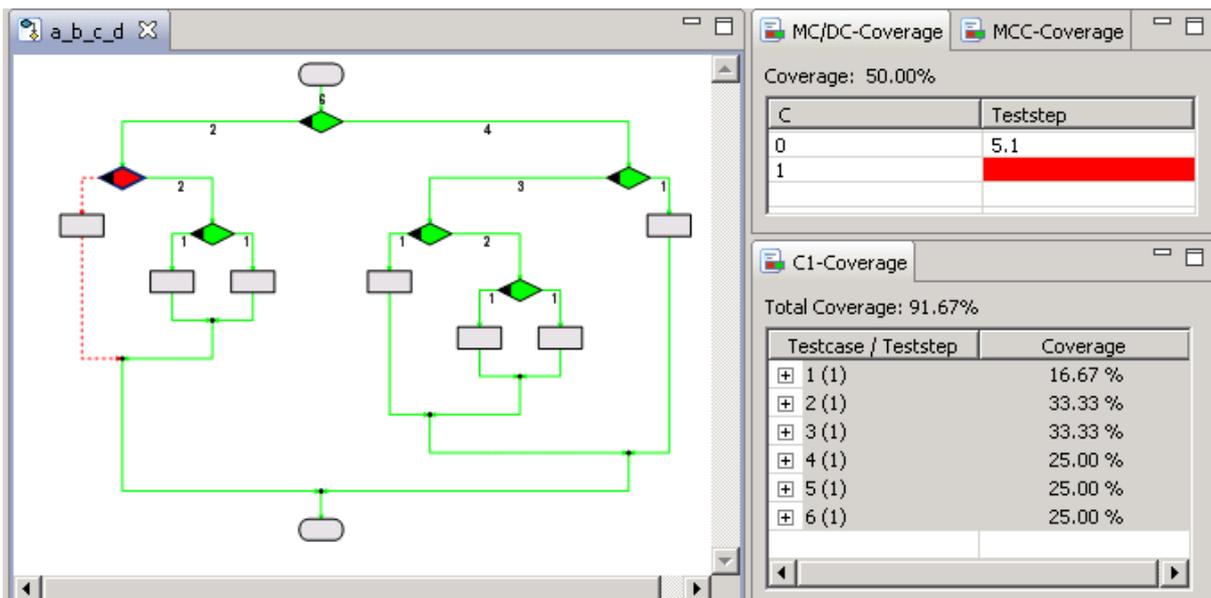

Fig. 35: Even six test cases do not result in 100% branch coverage for S2

Conclusion

- The way a decision is programmed is relevant for the effort to reach a certain percentage of a certain coverage measure.
- The example at hand points to a method (i.e. a programming style) in which MC/DC respectively MCC is subsumed by branch coverage. This may be useful if it is not possible to determine MC/DC respectively MCC, e.g. due to missing tools.

2.4.4. Appraisal of Condition Coverage

Especially if the test object contains a lot of decisions, and if the decisions are made up of conditions, condition coverage is more valuable than e.g. branch coverage. This is accomplished by a greater number of tests cases than for branch coverage. The increased amount of test cases naturally results both in higher testing intenseness and in higher testing effort. Sometimes it may be impossible to actually execute all required test cases for a decision, due to the logic of the program (especially for MCC).

Simple condition coverage and minimal multiple condition coverage should only be used if incomplete evaluation is given.

MCC requires more test cases than MC/DC and is seldom used in practice.

2.5.   Path Coverage

A path is a sequence of branches taken during execution of a test case for the test object. Path coverage ("Pfadüberdeckung") measures how many of the possible paths are executed during the tests.

The number of paths grows exponentially with the number of branches. Furthermore, normally not all paths can be executed due to the logic of the program.

Especially when loops are involved, normally a very large number of possible paths exist. The problem stems from the potentially "infinite" number of loop executions, e.g. the loop "(for i=0; i < n; i++)" will be executed n times, and n could be a 32 bit integer variable or even bigger. Therefore, it is practically impossible to achieve 100% path coverage. To cope with loops measures were introduced that classify the paths through a loop by the number of loop executions. Such a measure is boundary interior coverage.

2.5.1. Boundary Interior Coverage

Boundary interior coverage ("Grenze-Inneres-Überdeckung") classifies paths through a loop by the number of loop executions. Normally, three classes are defined:
(1)   A class for all paths that do not execute the loop.
(2)   A class for all paths that execute the loop exactly once.
(3)   A class for paths that execute the loop at least two times.

It should be clear, that for a do-while-loop, the class according to (1) is empty. Variations of this definition exist. The effort to reach 100% boundary interior coverage is high in every case.

3. The Value of Code Coverage

3.1.   Essential Step / Reveals Untested Code

It is an essential step in the testing process to determine the code coverage that was reached during testing. Code coverage measurement is important with respect to software quality, because it would reveal if e.g. a part of the code was never executed during testing. This would be a hint for "dead code" or insufficient test cases.

The value of code coverage may be increased if not only the reached percentage is considered, but also single test cases are investigated. Did the test case execute the expected path?

3.2.   Can Indicate Project Progress

If the total coverage of a project is monitored over time, this can be an indication for the progress of the project.

For instance:

(1)   If the percentage of the total coverage grows, this could mean testing is catching up.

(2)   If the percentage of the total coverage is getting smaller, this could mean new code is added without testing it appropriately.

However, this discussion is beyond the topic of the paper at hand.

3.3.   Two Weaknesses of Code Coverage

However, even if you have reached the desired percentage, you must keep two weaknesses of code coverage in mind:

(1)   Code coverage measurement cannot detect omissions, e.g. missing or incomplete code.

(2)   Code coverage measurement is insensitive to calculations.

3.3.1. Cannot Detect Omissions

For instance, the specification for a function could state:

"The maximum input value the algorithm can handle is 500. If the input value is greater than 500, it shall be set to 500."

If the implementation of the function misses to check the input value, and therefore also misses to take an appropriate action if the value is greater 500, code coverage measurement will not detect the missing check, even if a test case with the input value 501 is executed.

3.3.2. Insensitive to Calculations

Consider the following function.

```
long double sin(long double x_deg)
{
    int i;
    long double temp, x_rad;
    int sign = -1;

    x_rad = x_deg / 180 * pi ;
    temp = x_rad;

    for(i=3; i<=(MAX_FAC-2); i+=2)
    {
        temp += sign * pot(x_rad,i) / fac(i);
        sign *= -1;
    }
    return(temp);
}
```

Fig. 36: A function that calculates the sinus value of its input

A single test case with the input x_deg == 0 (i.e. an angle of zero degrees) returns 0, what is the correct and expected result. Code coverage measurement of this single test case yields 100% branch coverage, 100% MC/DC, and 100% MCC. Although you have reached 100% coverage for all measures, you have no evidence that the sinus calculation is correct. (A completely different function could be implemented, e.g. signum(). It would also return 0 for an input of 0).

This is an example where 100% code coverage is not enough.

4. Recommendations

4.1. Should Be Reached by Functional Test Cases

However, code coverage can be reached by inappropriate means. E.g., test cases can be derived from the structure of the code, hereby inappropriately assuming the code is correct. Furthermore, code coverage could be achieved by simple executing test cases with randomly generated test data until the desired code coverage is reached. This approach becomes worse if the actual results of the test cases are not evaluated against expected results.

Therefore, it is important to specify the test cases from a functional point of view (i.e. from the specification). The actual result of the test cases have to be compared with the expected results, to create the "pass/failed" verdict of the test. A functional test takes the test object as a black box, i.e. the structure of the test object is not considered. (A recommended method for the specification of functional test cases is the Classification Tree Method. See [TESSY]).

Therefore, the functional test cases should be specified prior to the measurement of the code coverage. After all functional tests have passed (i.e. yielded the expected result), code coverage should be determined. If you did not reach the required percentage, you should determine the reason. Maybe you have to change your test cases or you have to add some new. Maybe you have found unreachable code and need to do a justification.

4.2.   Should Be Done During Module/Unit Testing

Unit/module testing is a good phase in the testing process to determine the coverage. During unit/module testing, more input combinations are feasible than e.g. during system testing. You may be able to provoke and check some error conditions which you are unable to provoke during system testing (e.g. due to defensive programming). Unit/module testing tools like Tessy allow to determine several code coverage measures without added effort.

4.3.   Select the Right Measure

The code coverage measures to use and the percentage to achieve might be predetermined, e.g. by a customer or by a certification authority.

If this is not the case, spend some time to select the code coverage measure that is appropriate for the code of your project. This may be based on the criticality of the code, or the complexity of the code, or the frequency of usage of the code. Not every piece of code must be covered by the same level. E.g. the software monitoring a safety-critical system should be tested more rigorously by trying to achieve a certain percentage of MC/DC than the monitored code, where branch coverage may be sufficient.

Try to get the right relationship between effort for reaching a certain percentage of the desired code coverage measure and the information you get out of it. For instance, it is generally not a good practice to reach 100% MC/DC coverage for a whole project with a huge effort, and neglecting other types/approaches towards software quality (e.g. static analysis, reviews) completely.

5. Conclusion

- Be aware of the confusing nomenclature.

- Don't rate the quality of your code too high if you have reached 100%.

6. Literature

[TESSY]   http://www.hitex.com/tessy: More about Tessy and the Classification Tree Method.

[SPILLNER]   Spillner, A., Linz, T.: Basiswissen Softwaretest, Heidelberg, 2003. dpunkt-Verlag.

[LIGGESMEYER]   Liggesmeyer, Peter: Software-Qualität: Testen, Analysieren und Verifizieren von Software. Heidelberg, Berlin, 2002. Spektrum Akademischer Verlag.

[THALLER]   Thaller, Georg Erwin.: Softwaretest: Verifikation und Validation, Hannover, 2000. Heise-Verlag.

[DO-178B]   Software Considerations In Airborne Systems And Equipment Certification, RTCA, 1992.

[CAST-10]   Certification Authorities Software Team, Position paper, 2002.

[BEIZER]   Beizer, Boris: Software Testing Techniques, 2nd edition, New York, 1990.

[IEC 61508]   Functional safety of electrical/electronic/programmable electronic safety-related systems, part 7, IEC, 2000, www.iec.ch.

Hitex Head Office, Germany

Hitex GmbH
Greschbachstr. 12
76227 Karlsruhe
Germany

Phone: +49-721-9628-0
Fax: +49-721-9628-149
Email: info@hitex.de

Hitex UK

Hitex (UK) Ltd
Millburn Hill Road
University of Warwick Science Park
Coventry CV4 7 HS
United Kingdom

Phone: +49-721-9628-0
Fax: +49-721-9628-149
Email: info@hitex.de

Consulting   Engineering   Testing   Training   Tools   Software Components   System Manufacturing