White-box coverage

Mark Timmer

This document describes the white-box coverage metrics used in the Testing Techniques course. In practice, but also in testing literature, coverage metrics are often defined in an ambiguous way. Their apparent simplicity is dangerous; without precise definitions, different users might obtain different results for the same situation.

We will not provide formal definitions, in order to keep things easy, but do try to be as precise as possible. Note that the choices made in this document are not the only possible choices.

1 Control-flow graphs

Coverage metrics can be computed based on control-flow graphs: graphs that depict the control-flow in a program. Each statement is represented by a rounded rectangle and each decision point (`if` or `while`) by a diamond box.

The main issue here is the question what a statement actually is. Although several definitions are around, we define a statement as the smallest standalone element of a programming language. More concretely, in the context of for instance Java, this corresponds to every line of code that ends with a semicolon. Basically, this boils down to the observation that statements are variable declarations, variable assignments and return statements.

Example 1. Consider the following pseudo-code function:

```
 1.  int someFunction(int a, int b) {
 2.     int result = 0;                           (*)
 3.     if (a < b) {                              (**)
 4.       System.exit(0);                         (*)
 5.     }
 6.     else {
 7.       int c = a + b;                          (*)
 8.       int i = 0;                              (*)
 9.       while (i < c) {                         (**)
10.         result = (result + a) / b;            (*)
11.         i++;                                  (*)
12.       }
13.     }
14.     return result;                            (*)
15.  }
```

Note that this piece of code contains 7 statements (indicated by asterisks) and 2 decision points (indicated by double asterisks). Figure 1 provides the control-flow graph for this program. □

Now, each test execution of the program can be seen as a traversal of this graph. Take for instance the test case $t = (a = 1, b = 1 \mid 1)$, were the values before the bar indicate the inputs, and the value after the bar indicates the expected result. It traverses the edges 1-2-5-6-7-8-9-10-8-9-10-11-12.

Remark 1. Here, we purposely used a `while`-loop instead of a `for`-loop, as the latter is a mixture of two statements and a decision. For clarity, we therefore only write down control-flow graphs and compute coverage for `while`-loops, and assume each `for`-loop to be unfolded to a `while`-loop before computing any coverage metric.

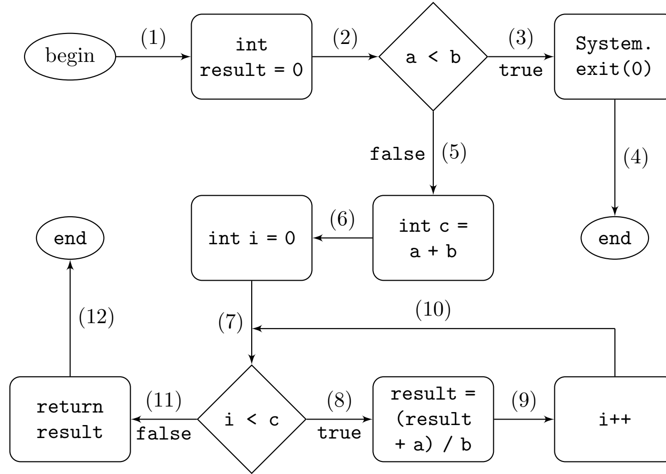


Figure 1: Control-flow graph for Example 1.

2 Statement coverage

Statement coverage is the most basic form of white-box coverage. Basically, a test suite T for a program P achieves complete statement coverage if it executes each statement in a program at least once. Consequently, every line of P's code has to be 'touched' during testing.

If no complete statement coverage is reached by T, then we can compute the *statement coverage percentage*: the percentage of statements that is touched by T.

Definition 1. *Given a program P with control-flow graph G and a test suite T for P, the* statement coverage percentage *of T is the percentage of statement nodes of G that are visited when executing T.*

Example 2. Consider the program introduced in Example 1, and consider the two test cases

- $t_1 = (a = 0, b = 1 \mid \text{error})$
- $t_2 = (a = 1, b = 1 \mid 1)$

To achieve complete statement coverage, we could for instance apply the test suite $T_1 = \{t_1, t_2\}$, as it traverses every statement node of the control-flow graph at least once. If we apply the test suite $T_2 = \{t_2\}$, the second statement (line 4; the rounded rectangle in the upper right of the control-flow graph) would not be executed anymore. So, as the traversal of test case T_2 visits 6 out of the 7 statements, the statement coverage percentage is $\frac{6}{7} \cdot 100\% = 85.7\%$. □

2.1 Unreachable code

It is easy to write down a program for which not all statements are reachable. For instance:

```
1.  void someFunction(int a, int b) {
2.     if (a > b && a < b) {
3.       System.exit(0);
4.     }
5.     else {
6.       System.out.println("Succeeded.");
7.     }
8.  }
```

Clearly, any test case for this function achieves 50% statement coverage: the statement on line 3 is unreachable and the statement on line 6 is always reached. We could argue that the unreachable code should be excluded from our computation, but in general it is undecidable which statements are reachable and which are not. Because of this, we will not take into account the reachability of code and just compute statement coverage under the assumption that all lines of code should be executed. The same holds for all other coverage metrics we will discuss.

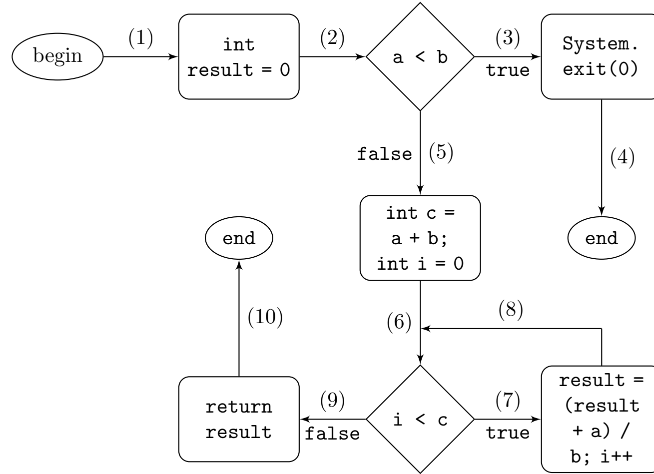


Figure 2: Adjusted control-flow graph for Example 3.

(We could even argue that unreachable code should not be present anyway; not being able to achieve complete statement coverage would then even be a positive thing, as it would motivate people to remove unreachable blocks of code.)

Remark 2. Note that program transformations may change coverage, even though they do not change a program's behaviour. For instance, an intelligent compiler could detect some unreachable code and change a program accordingly. Therefore, the example above might be transformed to

```
1.  void someFunction(int a, int b) {
2.      System.out.println("Succeeded.");
3.  }
```

Now, every test case suddenly has complete statement coverage, something that wasn't possible before the transformation.

2.2 Block coverage

Instead of computing the percentage of individual statements that are visited, it is also possible to compute the percentage of blocks of statements that are visited. Basically, this means that we merge adjacent rounded rectangles in the control-flow graph, and proceed as before.

Definition 2. *Given a program P with control-flow graph G and a test suite T for P. Let G' be the graph obtained from G by merging adjacent statement nodes into block nodes. Then, the* block coverage percentage *of T is the percentage of block nodes of G' that are visited when executing T.*

Example 3. Consider again the control-flow graph depicted in Figure 1. When merging the adjacent statement block, we obtain the graph depicted in Figure 2. Looking again at the test suite T_2 of Example 2, we now observe that it traverses 4 out of the 5 block nodes, so the block coverage is $\frac{4}{5} \cdot 100\%$.

3 Decision coverage

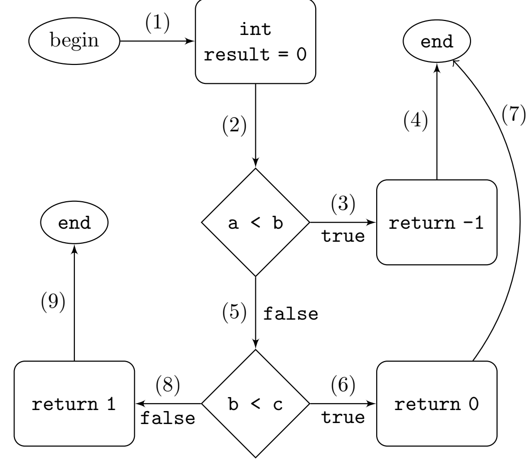
Decision coverage, also called branch coverage, is stronger than statement coverage. To completely achieve it, not only do we need to execute each statement, we also need to make sure that every decision in the program evaluates at least once to `true` and once to `false`. In that case, we say that every *branch* of the program is traversed at least once.

In this context, a decision is a boolean expression, used in for instance an `if`-statement or a `while`-loop. Note that we do not only require each decision to be `true` at least once and `false` at least once; we also require them to really be evaluated to these values during the testing process.

Definition 3. *Given a program P with control-flow graph G and a test suite T for P, the* decision coverage percentage *of T is the percentage of outgoing edges of decision nodes of G (called* branches*) that are visited when executing T.*

Example 4. Consider the following pseudo-code function and its control-flow graph:

```
 1.  int someFunction(int a, int b, int c) {
 2.    int result = 0;
 3.    if (a < b) {                 (**)
 4.      return -1;
 5.    }
 6.    else {
 7.      if (b < c)                 (**)
 8.        return 0;
 9.      else
10.        return 1;
11.    }
12.  }
```



The program consists of 2 decisions (again indicated by double asterisks), and therefore of 4 branches (corresponding to the edges 3, 5, 6 and 8 in the control-flow graph). Consider the following test cases:

- $t_1 = (a = 1, b = 2, c = 3 \mid -1)$
- $t_2 = (a = 3, b = 2, c = 1 \mid 1)$
- $t_3 = (a = 3, b = 1, c = 2 \mid 0)$

To cover each of the branches and achieve complete decision coverage, we could apply the test suite $T_1 = \{t_1, t_2, t_3\}$. Edge (3) is then traversed by test case t_1, and (5) is traversed by the other two test cases. Moreover, t_2 traverses (8), and t_3 traverses (6). That way, all branches have been traversed and indeed complete decision coverage is achieved.

Note that the test suite $T_2 = \{t_1, t_2\}$ *does not* achieve complete decision coverage. Although for the inputs provided by t_1 the second decision would evaluate the `true`, branch (6) is never traversed as the second decision is never reached during the executing of t_1. Therefore, as 3 out of 4 branches are executed by T_2, its decision coverage percentage is $\frac{3}{4} \cdot 100\% = 75\%$. □

4 Condition coverage

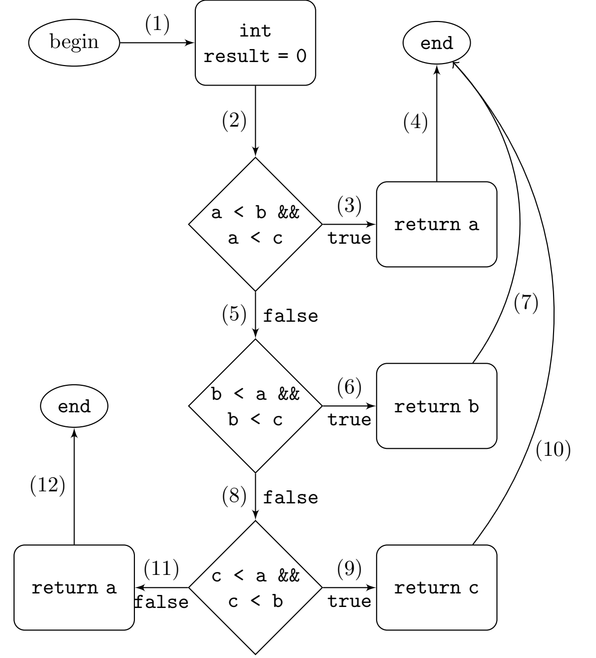
Just as for decision coverage, condition coverage also looks at the decisions that appear in a program. However, instead of requiring the entire boolean expression of each decision to evaluate at least once to `true` and once to `false`, for condition coverage we require every *condition* it consists of to evaluate at least once to `true` and once to `false`. A *condition* is just a boolean expression without any boolean operators. For instance, the decision `if(a > 10 && (b > c || x == 5))` consists of the conditions `a > 10`, `b > c` and `x == 5`.

Definition 4. *Given a program P with control-flow graph G and a test suite T for P. Let C be the total number of conditions in G (counting conditions that occur several times also really as the number of times they occur). Let $C_T^{\texttt{true}}$ be the number of these conditions for which it holds that the decision node containing the condition is reached by T at least once while at that times the condition is* `true`*. Symmetrically, we define $C_T^{\texttt{false}}$. Then, the* condition coverage percentage *of T is $\frac{C_T^{\texttt{true}} + C_T^{\texttt{false}}}{2C} \cdot 100\%$.*

Remark 3. We don't take into account the concept of lazy evaluation. Suppose for instance that we test a program containing a decision `if(b || c)`, choosing the inputs such that `b` and `c` both hold. Then, because of lazy evaluation, most languages would stop evaluation after evaluating `a`, as the disjunction cannot be `false` anyway anymore. We, however, take a more simple view, and do assume that all conditions in a decision are evaluated regardless of their ordering.

Example 5. Consider the following pseudo-code function:

```
1.  int min(int a, int b, int c) {
2.    int result = 0;
3.    if (a < b && a < c) {
4.      return a;
5.    }
6.    if (b < a && b < c) {
7.      return b;
8.    }
9.    if (c < a && c < b) {
10.     return c;
11.   }
12.   return a;
13. }
```



This program consists of 6 conditions, so $C = 6$. A test suite that wants to achieve complete condition coverage should make sure that each of the conditions is evaluated to `true` at least once and to `false` at least once. For this, consider the following test cases:

- $t_1 = (a = 1, b = 2, c = 3 \mid 1)$
- $t_2 = (a = 2, b = 1, c = 3 \mid 1)$
- $t_3 = (a = 3, b = 2, c = 1 \mid 1)$
- $t_4 = (a = 1, b = 1, c = 1 \mid 1)$

Applying the test suite $T_1 = \{t_1, t_2, t_3, t_4\}$ yields complete condition coverage. After all, for t_1 both conditions in the first decision evaluate to `true`, for t_2 both conditions in the second decision evaluate to `true` and for t_3 both conditions in the third decision evaluate to `true`. Finally, when running t_4 all conditions evaluate to `false`. Therefore, all conditions have been evaluated at least once to `true` and once to `false`, so $C_{T_1}^{\texttt{true}} = 6$, $C_{T_1}^{\texttt{false}} = 6$ and therefore the condition coverage percentage is $\frac{6+6}{2 \cdot 6} \cdot 100\% = 100\%$.

Note that the test suite $T_2 = \{t_1, t_2, t_3\}$ does not have complete condition coverage. Although all conditions at least once hold and at least once do not hold for the inputs provided by these test cases, not all of the decision nodes containing them are indeed reached. For T_2, we have $C_{T_2}^{\texttt{true}} = 6$ and $C_{T_2}^{\texttt{true}} = 3$, so the condition coverage percentage is $\frac{6+3}{2 \cdot 6} \cdot 100\% = 75\%$. □

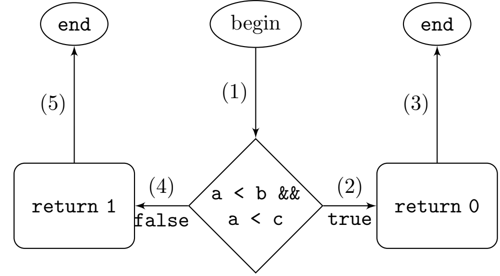
5 Decision / condition coverage

Decision / condition coverage combines the two metric introduced above. For it to be satisfied completely, all decision outcomes should occur at least once *and* all conditions should evaluate to `true` and to `false` at least once.

Definition 5. *Given a program P with control-flow graph G and a test suite T for P. Let D be the number of decision nodes of G, and let C be the total number of conditions in G (counting conditions that occur several times also really as the number of times they occur). Let $C_T^{\texttt{true}}$ be the number of these conditions for which it holds that the decision node containing the condition is reached by T at least once while at that times the condition is `true`. Symmetrically, we define $C_T^{\texttt{false}}$. Let D_T be the number of outgoing edges of decision nodes in G that are traversed when executing T. Then, the* decision / condition coverage percentage *of T is* $\dfrac{C_T^{\texttt{true}} + C_T^{\texttt{false}} + D_T}{2C + 2D} \cdot 100\%$.

Example 6. Consider the following pseudo-code function:

```
1.  int min(int a, int b, int c) {
2.    if (a < b && a < c) {
3.      return 0;
4.    }
5.    else {
6.      return 1;
7.    }
8.  }
```



This program consists of 2 conditions (so $C = 2$) and 1 decision (so $D = 1$). Now, consider the following test cases:

- $t_1 = (a = 2, b = 3, c = 1 \mid 1)$
- $t_2 = (a = 2, b = 1, c = 3 \mid 1)$

When applying the test suite $T = \{t_1, t_2\}$, both conditions evaluate once to `true` and once to `false`, so $C_T^{\texttt{true}} = 2$ and $C_T^{\texttt{false}} = 2$. However, only one outgoing edge of the decision node is traversed, so $D_T = 1$. Therefore, the decision / condition coverage percentage is $\frac{2+2+1}{2\cdot 2+2\cdot 1} \cdot 100\% = 83.3\%$. □

6 Multi-condition coverage

For multi-condition coverage, we do not only require every condition to be evaluated to `true` and to `false` at least once; we require every *combination* of such valuations of conditions within a decision to occur at least once. So, for a decision `if (a && b)` to be completely covered with respect to this metric, we should make sure that the decision is reached at least once while `a` is `true` and `b` is `true`, once while `a` is `true` and `b` is `false`, once while `a` is `false` and `b` is `true`, and once while `a` is `false` and `b` is `false`.

Definition 6. *Given a program P with control-flow graph G and a test suite T for P. For each decision node d of G, let C_d be the number of conditions in d. Then, $C = \sum_{d \in G} 2^{C_d}$ is the total number of combinations of conditions. Now, the* multi-condition coverage percentage *of T is the percentage of these combinations that indeed occur when executing T.*

Example 7. Consider the program of Example 5 again. Each of the three decisions consists of two conditions, and therefore contains $2^2 = 4$ combinations of conditions. This makes the total number of condition combinations $4 + 4 + 4 = 12$. The test case $T = \{t_1, t_2, t_3, t_4\}$ introduced in Example 5 had complete condition coverage, but does not have complete multi-condition coverage.

The following table provides the condition combinations that occur:

Test	`a < b`	`a < c`	`b < a`	`b < c`	`c < a`	`c < b`
t_1	`true`	`true`	-	-	-	-
t_2	`false`	`true`	`true`	`true`	-	-
t_3	`false`	`false`	`true`	`false`	`true`	`true`
t_4	`false`	`false`	`false`	`false`	`false`	`false`

Apparently, for the first decision the combination `true-false` is missing, for the second `false-true` is missing, and for the third both `true-false` and `false-true` are missing. So, we covered 8 out of the 12 combinations, yielding a multi-condition coverage of $\frac{8}{12} \cdot 100\% = 66.7\%$.

7 Path coverage

Path coverage is one of the strongest white-box coverage notions. For complete coverage, it requires every possible path through the control-flow graph to be traversed at least once.

Definition 7. *Given a program P with control-flow graph G and a test suite T for P. Let Π be the number of paths G, i.e., the number of ways in which we can travel from the* `begin` *node to an* `end`

node. Then, the path coverage percentage *of T is the percentage of these paths that are traversed when executing T.*

Remark 4. For any program containing at least one `while`-loop, the number of paths is infinite, and any finite test suite therefore has a path coverage percentage of 0%.

Remark 5. Every test case traverses exactly one path of a control-flow graph.

Example 8. Consider the function of Example 5 again. There are 4 possible ways to go from the `begin` node to the `end` node. Each of the test cases mentioned in the example traverses one of these paths. So, if for instance $T = \{t_1, t_2, t_3\}$, three of the four paths are traversed and the path coverage percentage is 75%.