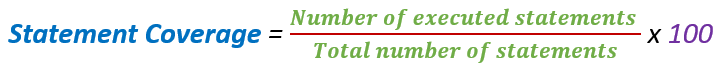
Statement coverage

*Statement coverage* is a code coverage metric that tells you whether the flow of control reached every executable statement of source code at least once.



Software developers and testers commonly use statement coverage because of its simplicity and availability in object code instrumentation technology. Of all the structural coverage criteria, statement coverage is the weakest, indicating the fewest number of test cases. Bugs can easily occur in the cases that statement coverage cannot see.

In particular, statement coverage does not call for testing the following:

* [Simple if statements](https://www.bullseye.com/statementCoverage.html#a1)
* [Logical operators](https://www.bullseye.com/statementCoverage.html#a2) (&&, ||, and ?:)
* [Loop termination decisions](https://www.bullseye.com/statementCoverage.html#a4)
* [Do-while loops](https://www.bullseye.com/statementCoverage.html#a5)

Specific Issues

### Simple If-Statements

Statement coverage does not call for testing simple if statements. A simple if statement has no else-clause. To attain full statement coverage requires testing with the controlling decision true, but not with a false outcome. No source code exists for the false outcome, so statement coverage cannot measure it.

int\* p = NULL;if (condition) { p = &variable; \*p = 1;}\*p = 0; // Oops, possible null pointer dereference

Without a test case that causes condition to evaluate false, statement coverage declares this code fully covered. In fact, if condition ever evaluates false, this code dereferences a null pointer.

### Logical Operators

Statement coverage cannot distinguish the code separated by logical operators from the rest of the statement. Executing any part of the code in a statement causes statement coverage to declare the whole statement fully covered.

void function(const char\* string1, const char\* string2 = NULL);...void function(const char\* string1, const char\* string2){ if (condition || strcmp(string1, string2) == 0) // Oops, possible null pointer passed to strcmp ...}

Statement coverage declares this code fragment fully covered when condition is true. With condition false, the call to strcmp gets an invalid argument, a null pointer.

### Loop Termination Decision

The C++ function below copies a string from one buffer to another.

char output[100];for (int i = 0; i <= sizeof(output); i++) { // Oops, buffer overrun; comparison should be < output[i] = input[i]; if (input[i] == '\0') { break; }}

The main loop termination decision, i <= sizeof(output), intends to prevent overflowing the output buffer. You can achieve full statement coverage without testing this condition. The overflow decision correctly ought to use operator < rather than operator <=. You get full statement coverage of this code with any input string of length less than 100, without exposing the bug.

### Do-While Loop Example

Consider the C++ function below, which initializes a string buffer with an optional input string.

void initString(char\* output, const char\* input = ""){ int i = 0; do { output[i] = input[i]; } while (input[i] != '\0'); // Oops, loop variable not incremented}

You can achieve full statement coverage without repeating this loop. Testing with a zero-length input string is sufficient for statement coverage. The problem is the programmer forgot to increment the index. Any non-zero length input string causes an infinite loop.

Main Idea

\*\*The fundamental assumption of code coverage is that to expose bugs, you should exercise as many *paths* through your code as possible. The more paths you exercise, the more likely your testing is to expose bugs. A path is a sequence of branches (decisions), or conditions (logical predicates). Bugs are often sensitive to branches and conditions. For example, incorrectly writing a condition such as i<=n rather than i<n may cause a boundary error bug.

Condition coverage

Condition coverage measures the proportion of conditions within decision expressions that have been evaluated to both true and false.

**Condition Coverage** or expression coverage is a testing method used to test and evaluate the variables or sub-expressions in the conditional statement.

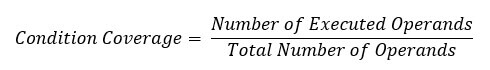
The goal of condition coverage is to check individual outcomes for each logical condition.

Condition coverage offers better sensitivity to the control flow than decision coverage.

In this coverage, expressions with logical operands are only considered.For example, if an expression has Boolean operations like AND, OR, XOR, which indicates total possibilities.

Condition coverage does not give a guarantee about full decision coverage.Note that 100% condition coverage does not guarantee 100% decision coverage. For example, “if (A || B) {do something} else {do something else}” is tested with [0 1], [1 0], then A and B will both have been evaluated to 0 and 1, but the *else* branch will not be taken because neither test leaves both A and B false.

The formula to calculate Condition Coverage:



Example:



For the above expression, we have 4 possible combinations

* TT
* FF
* TF
* FT

Consider the following input

|  |  |  |  |
| --- | --- | --- | --- |
| X=3  Y=4 | (x<y) | TRUE | Condition Coverage is ¼ = 25% |
| A=3  B=4 | (a>b) | FALSE |

Working Formula and Definition

Predicate :

A **path predicate expression** is a set of boolean **expressions**, all of which must be satisfied to achieve the selected **path**. Any set of input values that satisfy all of the conditions of the **path predicate expression** will force the routine to the **path**.

Atomic Condition:

A **condition** that cannot be decomposed, i.e., a **condition** that does not contain two or more single **conditions** joined by a logical operator (AND, OR, XOR).

e.g.

if(((a1551570219 == 32 && (( cf==1 && a1944816302 == 32) && (input == 2))) && a1041640432 == 10))

In our Code we defined several properties of a predicate object.

1. Rank
2. STarget
3. SObserved
4. Fitness
5. AtomCount
6. OperatorScore
7. ReachabilityScore
8. LineNo line # of predicate p in c program
9. Text content of predicate p

AtomCount(p):= # of atomic conditions in predicate p.

OperatorScore(p):= 10\*mixOpCount+5\*andOpCount+5\*orOpCount

* MixOpCount(p) takes val as 0(absence of mix) or 1(mix operator &&,|| present)
* andOpCount(p): count of AND(&&) operator in pth predicate
* OrOpCount(p): count of OR(||) operator in pth predicate

ReachabilityScore(p):= ΣrScore(i), where i varies from 1..atomCount

rScore(i)=3 if both true and false branch are satisfied or

2 if either is satisfied

1 if none.

STarget(TargetSuspiciousScore)(p):=

(atomCount÷3.0)+(operatorScore÷2.0)+((3\*atomCount)÷1.0)

SObserved(ObservedSuspiciousScore)(p):=

(atomCount÷3.0)+(operatorScore÷2.0)+(reachabilityScore)÷1.0

Fitness(p):= SObserved÷STarget