Coverage Testing

Barbara Russo
SwSE - Software and Systems Engineering research group

Coverage

- One way to judge a test suite is to ask how thoroughly it exercises the program
- This is called coverage

Example text

```
1  int foo (int a, int b, int c, int d, float e) {
2      if (a == 0) {
3          return 0;
4      }
5      int x = 0;
6      if ((a==b) || ((c == d) && bug(a) )) {
7           x=1;
8      }
9      e = 1/x;
10    return e;
11 }
```

bug(): if a=1 it returns true

and false if a!=1.

Coverage

- Coverage is a measure of the completeness of the set of test cases
  - Method coverage
  - Statement coverage
  - Branch coverage
  - Condition coverage

Method coverage

- Measure: percentage of methods that have been executed at least once by test cases
- Tests should call 100% of the methods
- It seems irresponsible to deliver methods in the product when testing never used these methods
  - you need to ensure you have 100% method coverage

Test Case 1

- There is only one method
- int foo (int a, int b, int c, int d, float e)
- for a=0 foo returns 0 no matter the values of the other parameters
- calling foo with input (0,0,0,0,0) we attain 100% method coverage in our example

Statement coverage

- Measure: percentage of statements that have been executed by test cases
  - Achieve 100% statement coverage. Count the number of statements and cover all of them with a test

Example

- With Test Case 1, we executed the program statements on lines 1-4 out of 11 lines of code
- As a result, we had 42% (5/12) statement coverage from Test Case 1

Example

- We can attain 100% statement coverage by one additional test case,
- Test Case 2: foo(1, 1, 1, 1,1), expected return value of 1.
- we have now executed the program statements on lines 5-11

Branch Coverage

- Measure: percentage of the decision points have been evaluated as both true and false in test cases.
- Two decision points – one on line 2 and the other on line 6
- 2 if (a == 0) {}
- 6 if ((a==b) OR ((c == d) AND bug(a))) {}

- For decision/branch coverage, we evaluate an entire Boolean expression as one true-or-false predicate
- We need to ensure that each of these predicates (compound or single) is tested as both true and false

Line #	Predicate	True	False
3	(a == 0)	Test Case 1 foo(0, 0, 0, 0, 0) return 0	Test Case 2 foo(1, 1, 1, 1, 1) return 1
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2 foo(1, 1, 1, 1, 1) return 1	

TestCase3

- With TewstCase1 and TestCase2 we have executed three of the four necessary conditions
  - we have achieved 75% branch coverage so far
- TestCase3 foo(1, 2, 1, 2, 1) return ??
- in calculating the output, we discover a division by 0 that can cause future failures!
  - That was due to a local variable that we could not control before!

Line #	Predicate	True	False
3	(a == 0)	Test Case 1 foo(0, 0, 0, 0, 0) return 0	Test Case 2 foo(1, 1, 1, 1, 1) return 1
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2 foo(1, 1, 1, 1, 1) return 1	TestCase3 foo(1, 2, 1, 2, 1) division by zero!

Condition Coverage

- Measure: percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome in test cases
  - applies to compound predicates
- Condition coverage measures the outcome of each of these sub-expressions independently of each other

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, 1,1, 1) return 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)		Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)		

Only the 50% coverage!

TestCase4

- We examine our available information on the bug method and determine that is should return true when a=1
- foo(1, 2, 1, 1, 1), expected return value 1

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, 1,1, 1) return 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)	Test Case 4 foo(1, 2, 1,1, 1) return 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)	Test Case 4 foo(1, 2, 1,1, 1) return 1	

TestCase5

- To finalize our condition coverage, we must force bug(a) to be false
- We again examine our bug() method, which informs us that it should return a false value if fed any integer a different from 1
- So we create Test Case 5, foo(3, 2, 1, 1, 1), expected return value “division by zero”.

Note

- We could have (2,2,1,1,1). The input would have been fine but we would never reach the AND condition. Thus, we must make the (a==b) false to be sure to test the AND condition for FALSE.
- The same applies for (c==d): we need to have it TRUE to be sure that FALSE is due to the bug(a) condition.

Traceability matrix

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, 1,1, 1) return 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)	Test Case 4 foo(1, 2, 1,1, 1) return 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)	Test Case 4 foo(1, 2, 1,1, 1) return 1	Test Case 5 foo(3, 2, 1,1, 1) division by zero!

Path coverage

- Path coverage is every possible combination of branches — every path through the program — taken by some test case
- McCabe complexity is used to determine how many complete execution paths (i.e. test built on them) a tester need to consider
- As with code coverage this is a measure that approximates exhaustiveness

JaCoCo

- It is an Eclipse plug-in
- In the node build and sub-node plugins of the POM file include

```
<plugin>
   <groupId>org.jacoco</groupId>
   <artifactId>jacoco-maven-plugin</artifactId>
   <version>0.8.2</version>
   <executions>
      <execution>
         <goals>
            <goal>prepare-agent</goal>
         </goals>
      </execution>
<!-- attached to Maven test phase -->
      <execution>
            <id>report</id>
            <phase>test</phase>
            <goals>
                  <goal>report</goal>
            </goals>
      </execution>
   </executions>
</plugin>
```

Exercise

```
public class Hailstone {
  public static void main(String[] args) {
    int n = 3;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

- Run this class with JaCoCo code coverage highlighting turned on, by choosing Run → Coverage As → Java Application.
- By changing the initial value of n, you can observe how JaCoCo highlights different lines of code differently.

When n=3 initially, what color is the line n = n/2 after execution?

When n=16 initially, what color is the line n = 3 * n + 1 after execution?

What initial value of n would make the line while (n != 1) yellow after execution?