Software Testing, Verification and Validation

November 9, 2022
Week #9— Lecture #6

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Last week, we introduced line/statement coverage and decision coverage (aka branch coverage) as part of our set of white-box techniques. This week we will introduce condition coverage, condition+decision coverage, path coverage, and mc/dc.

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

```
public class BlackJack {

    public int play(int left, int right) {
1         int ln = left;
2         int rn = right;
3         if (ln > 21)
4           ln = 0;
5         if (rn > 21)
6           rn = 0;
7         if (ln > rn)
8           return rn;
9         else
10          return ln;
    }

}
```

`play(30, 30)`

$$\frac{9}{10} \times 100\% = 90\% \text{ line coverage}$$

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)
          ln = 0;
        if (rn > 21)
          rn = 0;
        if (ln > rn)
          return rn;
        else
          return ln;
    }

}
```

```java
play(30, 30)
play(10, 9)
```

$$\frac{10}{10} \times 100\% = 100\% \text{ line coverage}$$

An interesting aspect of line coverage is that it is quite easy to visualize achieved coverage, in order to help developers improve the code coverage of their tests.

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21) ln = 0;
        if (rn > 21) rn = 0;
        if (ln > rn) return rn;
        else return ln;
    }

}
```

```
public class BlackJack {

    public int play(int left, int right) {
1       int ln = left;
2       int rn = right;
3       if (ln > 21) ln = 0;
4       if (rn > 21) rn = 0;
5       if (ln > rn) return rn;
6       else return ln;
    }

}
```

```
play(10, 9)
```

$$\frac{5}{6} \times 100\% = 83\% \text{ line coverage}$$

Using lines of code as a way to determine line coverage is a simple and straightforward idea. However, counting the covered lines is not always a good way of calculating the coverage. The number of lines in a piece of code depends on the decisions taken by the programmer who writes the code. Some coverage tools measure coverage at statement level. Statements are the unique instructions that your JVM, for example, executes. This is a bit better, as splitting one line of code in two would not make a difference.

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)  // True ??  False ??
          ln = 0;
        if (rn > 21)  // True ??  False ??
          rn = 0;
        if (ln > rn)  // True ??  False ??
          return rn;
        else
          return ln;
    }

}


play(30, 30)
```

```java
public class BlackJack {

    public int play(int left, int right) {
        int ln = left;
        int rn = right;
        if (ln > 21)   // True ✅   False ❌
          ln = 0;
        if (rn > 21)   // True ✅   False ❌
          rn = 0;
        if (ln > rn)   // True ❌   False ✅
          return rn;
        else
          return ln;
    }

}
```

```java
play(30, 30)
```

$$\frac{3}{6} \times 100\% = 50\% \text{ decision coverage}$$

Statement or Line coverage is generally seen as a weak criterion. Stronger criteria are often based on the control flow graph of the program under test, e.g., decision coverage. *Decision coverage* captures the notion of coverage of all edges in the control flow graph, which means that each if condition requires at least one test where it evaluates to `true`, and at least one test where it evaluates to `false`.

Condition Coverage

Condition Coverage

Decision coverage gives two branches for each decision, no matter how complicated or complex the decision is. When a decision gets complicated, i.e., it contains more than one conditions, e.g.,

```
if (a > 10 && b < 20 && c < 10)
    // code to test
```

decision coverage might not be enough to test all the possible outcomes of all these decisions.

```
if (a > 10 && b < 20 && c < 10) // True ??  False ??
    // code to test
```

```
if (a > 10 && b < 20 && c < 10) // True ✅  False ??
    // code to test
```

```
t1(a=20, b=10, c=5)
```

$$\frac{1}{2} \times 100\% = 50\% \text{ decision coverage}$$

```
if (a > 10 && b < 20 && c < 10) // True ✅  False ✅
    // code to test
```

```
t1(a=20, b=10, c=5)
t2(a=5,  b=10, c=5)
```

$$\frac{2}{2} \times 100\% = 100\% \text{ decision coverage}$$

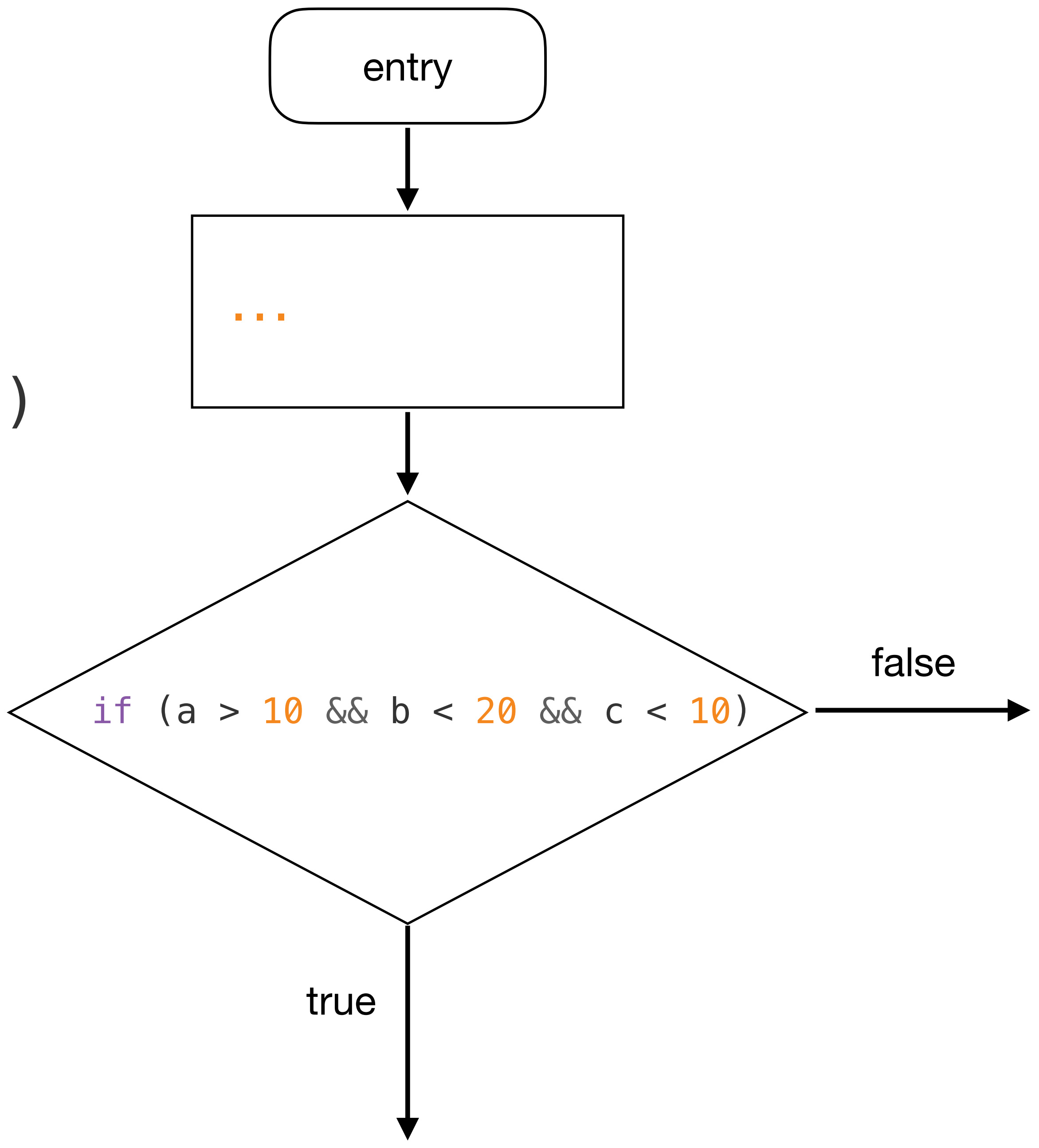
Although these two test cases fully cover this decision block, in terms of decision coverage, they do not cover all the possibilities/different combinations for this decision to be evaluated to `false`; e.g., t3(a=20, b=30, c=5), etc.

Condition Coverage

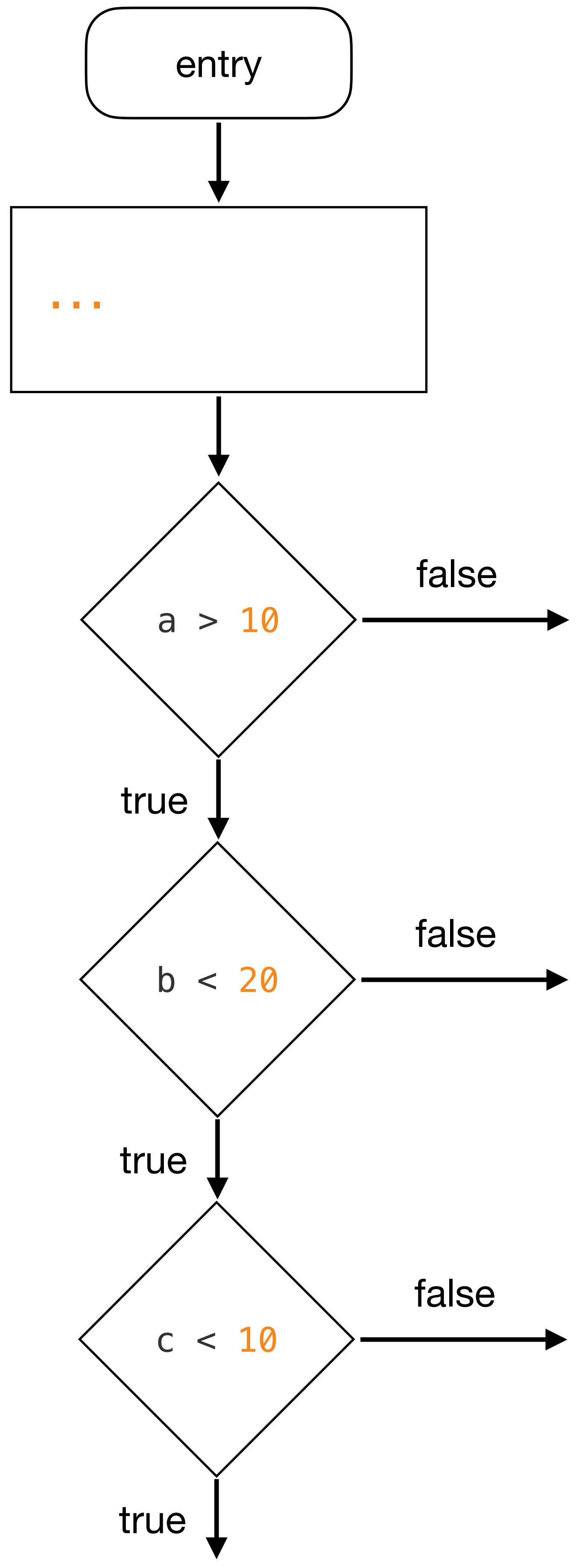
When using *condition coverage* as a criterion, we split each compound condition into multiple decision blocks. This means each of the conditions will be tested separately, and not only the "big decision block".

It is common to then re-design the CFG and make sure each decision block is now composed of a single condition. With the new CFG in hands (and with it new edges to explore), it works the same as decision coverage. The formula is basically the same, but now there are more decision outcomes to count.

```
if (a > 10 && b < 20 && c < 10)
    // code to test
```



```
if (a > 10)
    if (b < 20)
        if (c < 10)
            // code to test
```



$$\text{condition coverage} = \frac{\text{\# conditions outcomes covered}}{\text{\# conditions outcomes}} \times 100\%$$

We achieve 100% condition coverage when all of the outcomes of all the conditions in our program have been exercised. In other words, whenever all the conditions have been `true` and `false` at least once.

```
if (a > 10 && b < 20 && c < 10)
    // code to test
```

$$\texttt{if (}\overbrace{\texttt{a > 10}}^{\texttt{// True ?? False ??}}\texttt{ \&\& }\overbrace{\texttt{b < 20}}^{\texttt{// True ?? False ??}}\texttt{ \&\& }\overbrace{\texttt{c < 10}}^{\texttt{// True ?? False ??}}\texttt{)}$$

```
    // code to test
```

$$\texttt{if (}\overbrace{\texttt{a > 10}}^{\text{// True ✓ False ??}}\texttt{ \&\& }\overbrace{\texttt{b < 20}}^{\text{// True ✓ False ??}}\texttt{ \&\& }\overbrace{\texttt{c < 10}}^{\text{// True ✓ False ??}}\texttt{)}$$

```
if (a > 10 && b < 20 && c < 10)
    // code to test
```

```
t1(a=20, b=10, c=5)
```

$$\frac{3}{6} \times 100\% = 50\% \text{ condition coverage}$$

// True ✓ False ✓

// True ✓ False ??

// True ✓ False ??

```
if (a > 10 && b < 20 && c < 10)
    // code to test
```

```
t1(a=20, b=10, c=5)
t2(a=5,  b=10, c=5)
```

$$\frac{4}{6} \times 100\% = 66.7\% \text{ condition coverage}$$

$$\texttt{if (}\overbrace{\texttt{a > 10}}^{\text{// True ✓ False ✓}}\texttt{ \&\& }\overbrace{\texttt{b < 20}}^{\text{// True ✓ False ✓}}\texttt{ \&\& }\overbrace{\texttt{c < 10}}^{\text{// True ✓ False ??}}\texttt{)}$$

```
    // code to test
```

```
t1(a=20, b=10, c=5)
t2(a=5,  b=10, c=5)
t3(a=20, b=30, c=5)
```

$\frac{5}{6}$ x 100% = 83.3% condition coverage

Does 100% condition coverage imply in 100% decision coverage?

```
read x
read y
if (x == 0 || y > 0)
    y = y / x
else
    x = y + 2
print x + y
```

Does 100% condition coverage imply in 100% decision coverage?

```
read x
read y
if (x == 0 || y > 0)
    y = y / x
else
    x = y + 2
print x + y
```

$\left\{ \begin{array}{l} \text{// x == 0, True ?? False ??} \\ \text{// y > 0, True ?? False ??} \end{array} \right.$

Does 100% condition coverage imply in 100% decision coverage?

```
read x
read y
if (x == 0 || y > 0)
    y = y / x
else
    x = y + 2
print x + y
```

$\left\{ \begin{array}{l} \text{// x == 0, True ✅ False ??} \\ \text{// y > 0, True ?? False ✅} \end{array} \right.$

```
t1(x=0, y=-5)
```

$$\frac{2}{4} \times 100\% = 50\% \text{ condition coverage}$$

Does 100% condition coverage imply in 100% decision coverage?

```
read x
read y
if (x == 0 || y > 0)
    y = y / x
else
    x = y + 2
print x + y
```

$\left\{ \begin{array}{l} \text{// x == 0, True ✅ False ✅} \\ \text{// y > 0, True ✅ False ✅} \end{array} \right.$

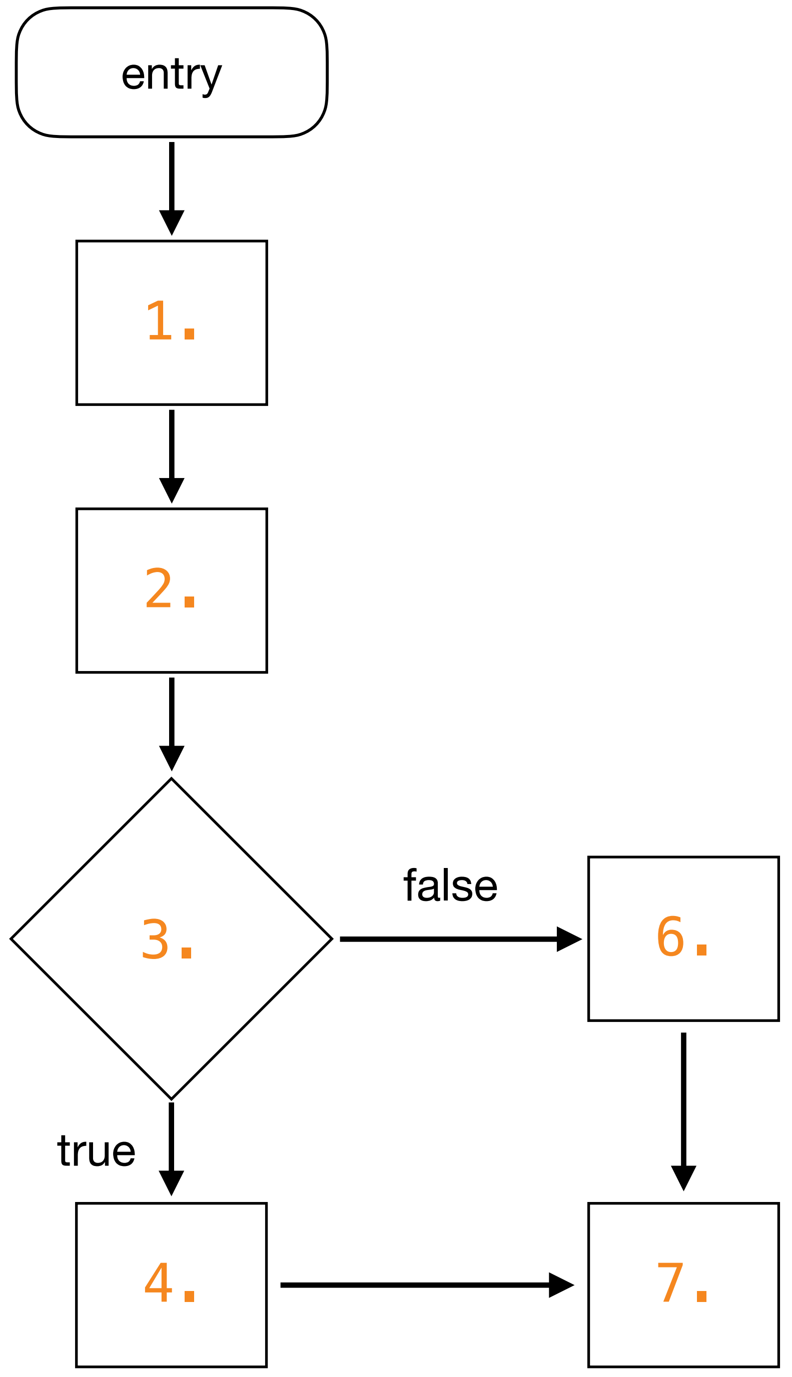
```
t1(x=0, y=-5)
t2(x=5, y=5)
```

$$\frac{4}{4} \times 100\% = 100\% \text{ condition coverage}$$

Does 100% condition coverage imply in 100% decision coverage?

```
1    read x
2    read y
3    if (x == 0 || y > 0)
4        y = y / x
5    else
6        x = y + 2
7    print x + y
```

```
t1(x=0, y=-5)
t2(x=5, y=5)
```

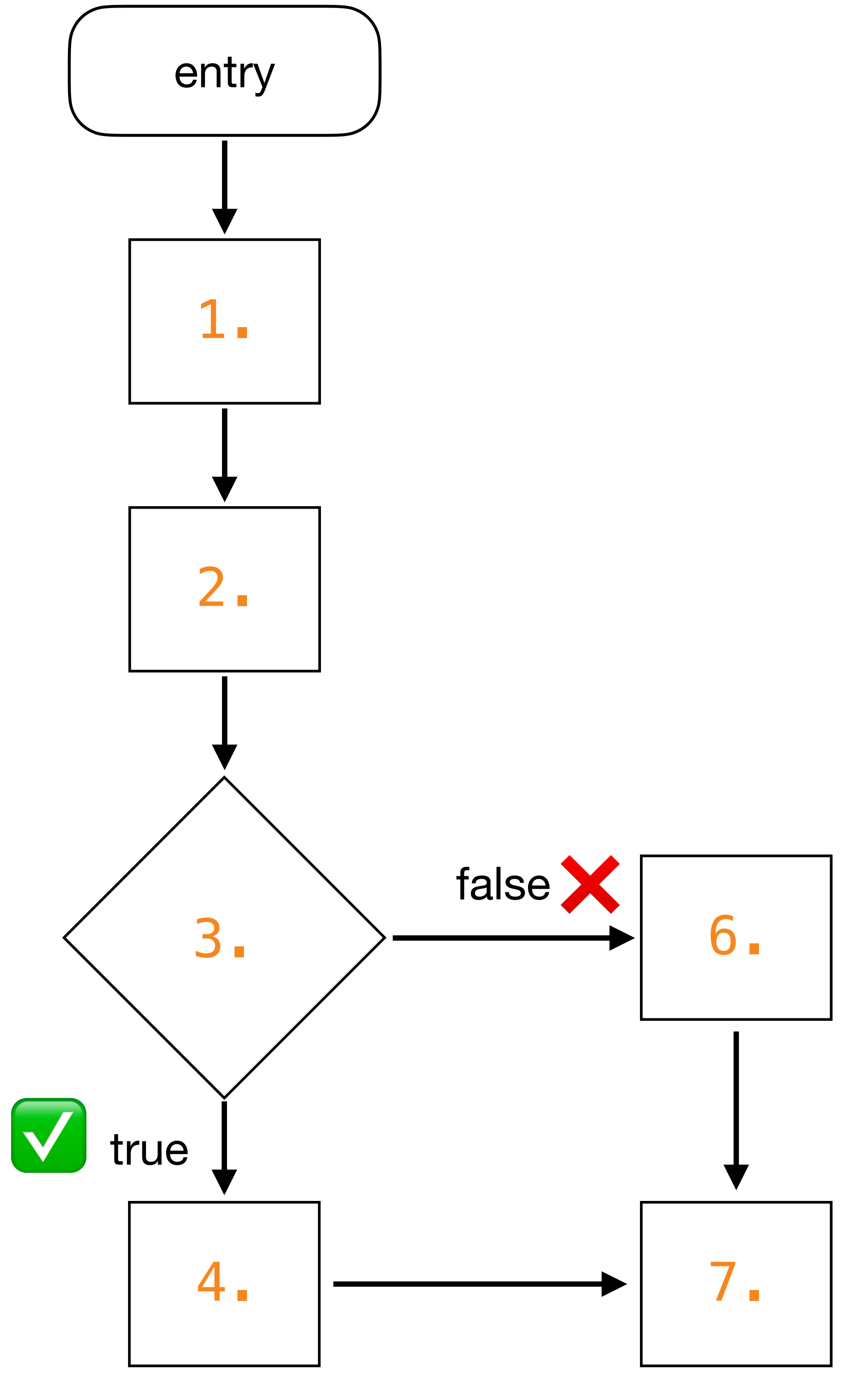


Does 100% condition coverage imply in 100% decision coverage?

```
read x
read y
if (x == 0 || y > 0)
    y = y / x
else
    x = y + 2
print x + y
```

```
t1(x=0, y=-5)
t2(x=5, y=5)
```

100% condition coverage, but
50% decision coverage



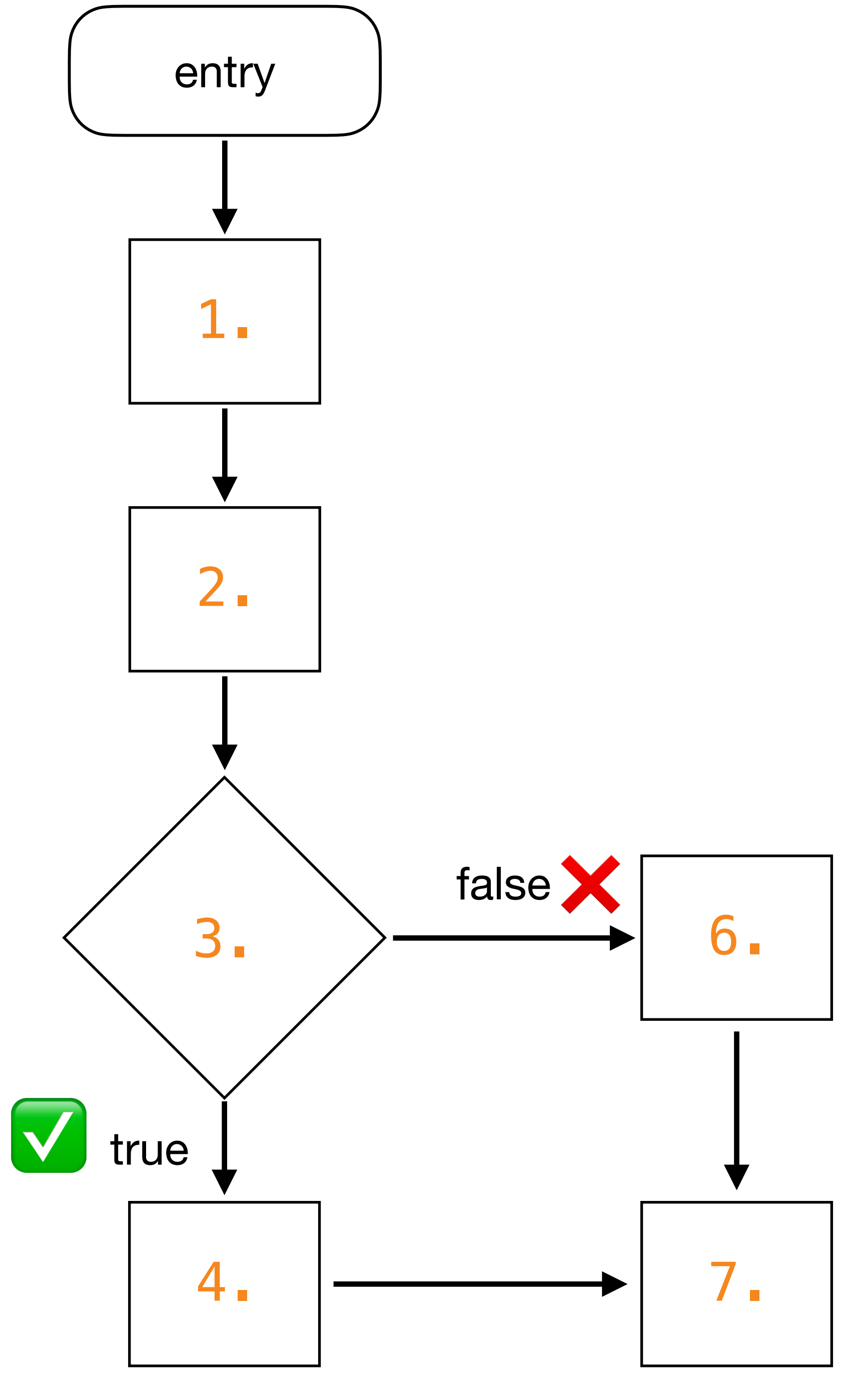
Condition Coverage + Decision Coverage

Condition + Decision Coverage

```
1     read x
2     read y
3     if (x == 0 || y > 0)
4         y = y / x
5     else
6         x = y + 2
7     print x + y
```

```
t1(x=0, y=-5)
t2(x=5, y=5)
```

100% condition coverage, but
50% decision coverage

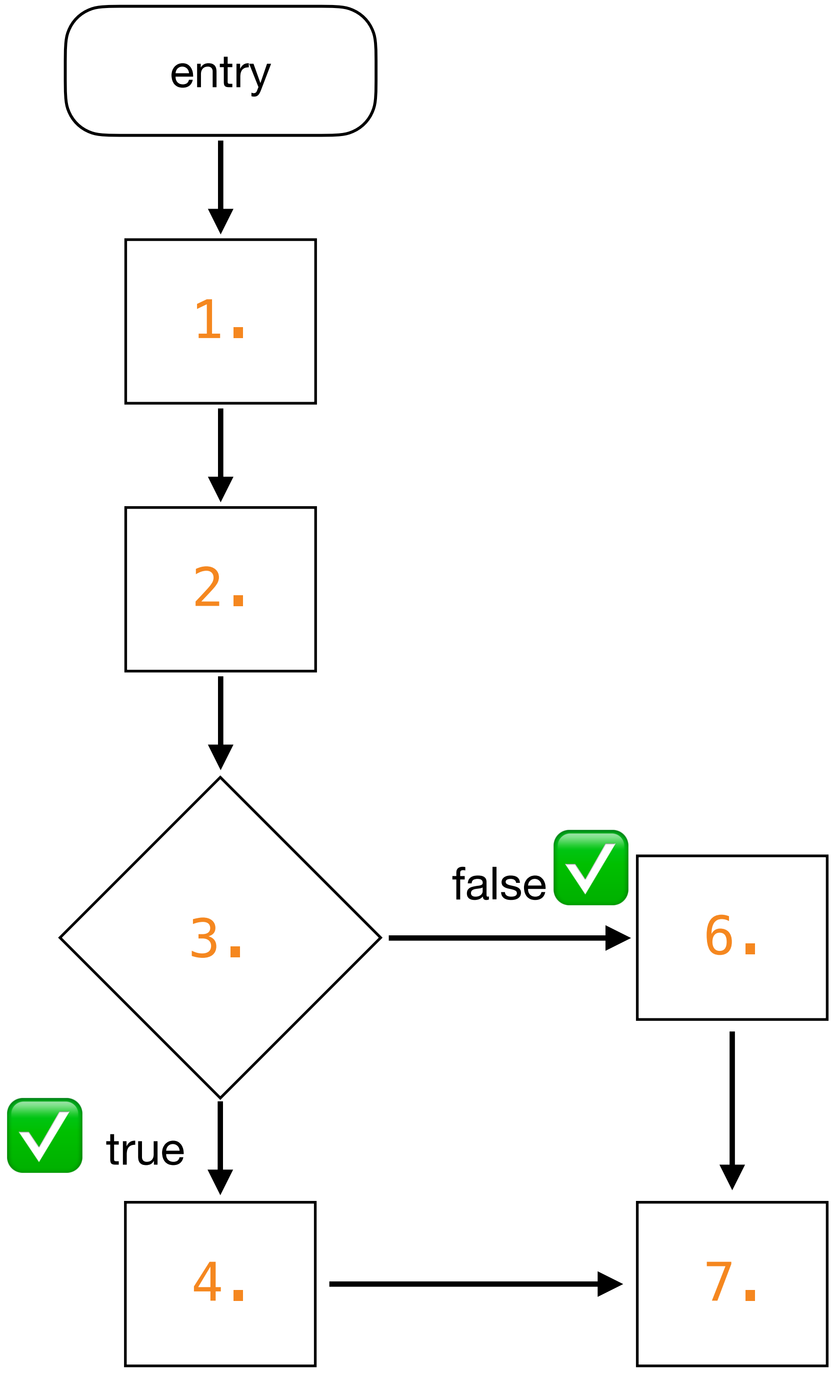


Condition + Decision Coverage

```
1     read x
2     read y
3     if (x == 0 || y > 0)
4         y = y / x
5     else
6         x = y + 2
7     print x + y
```

```
t1(x=0, y=-5)
t2(x=5, y=5)
t3(x=5, y=-5)
```

100% condition coverage and
100% decision coverage



$$\text{c/d coverage} = \frac{\text{\# conditions outcomes covered} + \text{\# decisions outcome covered}}{\text{\# conditions outcomes} + \text{\# decisions outcome}} \times 100\%$$

In practice, we should perform condition + decision coverage. In other words, we should make sure that we achieve 100% condition coverage (i.e., all the outcomes of all conditions are exercised) and 100% decision coverage (all the outcomes of the compound decisions are exercised). The formula to calculate condition + decision coverage is as above. Note this formula gives us a clear differentiation between basic condition and decision+condition coverage.

Path Coverage

Path Coverage

With condition + decision coverage, we looked at each *condition* and *decision* individually. Such a criterion gives testers more branches to generate tests, especially when compared to the first criterion we discussed (line coverage).

However, although we are testing each condition to be evaluated as `true` and `false`, this does not ensure testing of all the paths that a program can have.

Path coverage considers the (full) combination of the conditions in a decision rather than considering the conditions individually. Each of these combinations is a path. This, however, is rarely a practical criterion given the enormous number of possible paths in a software system.

$$\text{path coverage} = \frac{\text{\# paths covered}}{\text{\# paths}} \times 100\%$$

```java
if (!Character.isLetter(str.charAt(i)) & (last == 's' | last == 'r')) {
    words++;
}
```

```
(         A                  & (   B    |    C   ))
```

```java
if (!Character.isLetter(str.charAt(i)) & (last == 's' | last == 'r')) {
    words++;
}
```

Note that this piece of code avoids lazy (short-circuit) operators (i.e., && and ||), on purpose, to make sure all conditions of the expression are evaluated.

```
  (           A                  & (  B     |    C   ))

if (!Character.isLetter(str.charAt(i)) & (last == 's' | last == 'r')) {
    words++;
}
```

To get 100% path coverage, we would have to test all the possible combinations of these three conditions.

```
     (          A                      & (   B      |    C    ))
if (!Character.isLetter(str.charAt(i)) & (last == 's' | last == 'r')) {
    words++;
}
```

Tests	A	B	C	Outcome

$$(\overbrace{\texttt{!Character.isLetter(str.charAt(i))}}^{A}) \;\&\; (\overbrace{\texttt{last == 's'}}^{B} \mid \overbrace{\texttt{last == 'r'}}^{C}))$$

```java
if (!Character.isLetter(str.charAt(i)) & (last == 's' | last == 'r')) {
    words++;
}
```

Tests	A	B	C	Outcome
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

$$\left(\overbrace{\texttt{!Character.isLetter(str.charAt(i))}}^{A}\ \&\ \left(\overbrace{\texttt{last == 's'}}^{B}\ |\ \overbrace{\texttt{last == 'r'}}^{C}\right)\right)$$

```
if (!Character.isLetter(str.charAt(i)) & (last == 's' | last == 'r')) {
    words++;
}
```

Tests	A	B	C	Outcome
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

This means that, for full path coverage, we would need 8 tests just to cover this `if` statement. It is a large number for just a single statement. By aiming at achieving path coverage of our program, testers can indeed come up with good tests. However, the main issue is that achieving 100% path coverage might not always be feasible or too costly. The number of tests needed for full path coverage will grow exponentially with the number of conditions in each decision.

Lazy vs Eager Operators in Path Coverage

```
   (         A                  & (   B      |     C    ))
```

```java
if (!Character.isLetter(str.charAt(i)) & (last == 's' | last == 'r')) {
     words++;
}
```

Note that this piece of code avoids lazy (short-circuit) operators (i.e., && and ||), on purpose, to make sure all conditions of the expression are evaluated. This allows us to devise test cases for each possible combination we saw in the decision table. However, that might not be the case if we use lazy operators.

Lazy vs Eager Operators in Path Coverage

```
   (           A                  && (   B      ||    C    ))
```

```
if(!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
    words++;
}
```

Note that this piece of code avoids lazy (short-circuit) operators (i.e., && and ||), on purpose, to make sure all conditions of the expression are evaluated. This allows us to devise test cases for each possible combination we saw in the decision table. However, that might not be the case if we use lazy operators. Let's take as an example the same expression, but now using lazy operators: (A && (B || C)).

```
  (        A                   && (  B    ||   C    ))

if(!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
    words++;
}
```

Tests	A	B	C	Outcome

```
  (            A                 && (   B     ||   C    ))
if(!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
    words++;
}
```

Tests	A	B	C	Outcome
t1	T	T	(don’t care)	T
t2	T	F	T	T
t3	T	F	F	F
t4	F	(don’t care)	(don’t care)	F

```
   (             A                    && (     B      ||     C     ))

if(!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
    words++;
}
```

Tests	A	B	C	Outcome
t1	T	T	(don’t care)	T
t2	T	F	T	T
t3	T	F	F	F
t4	F	(don’t care)	(don’t care)	F

For this particular example, if A is `false`, then the rest of the expression will be not evaluated because the result of the entire statement will be automatically `false`. Moreover, for the second part of the expression, if B is `true`, then the entire proposition (B || C) is already `true`, so we “don’t care” about the value of C. Generically speaking, it might be not possible to devise test cases for all the combinations. As a tester, you just have to take such constraints into consideration.

Loops in Path Coverage

In terms of coverage criteria, what to do when we have loops? When there is a loop, the block inside of the loop might be executed many times, making testing more complicated.

Think of a `while(true)` loop which can be non-terminating. If we wanted to be rigorous about it, we would have to test the program where the loop block is executed one time, two times, three times, etc. Imagine a `for(i = 0; i < 10; i++)` loop with a break inside of the body. We would have to test what happens if the loop body executes one time, two times, three times, ..., up to ten times. It might be impossible to exhaustively test all the combinations.

How can we handle long-lasting loops (a loop that runs for many iterations), or unbounded loops (where we do not know how many times it will be executed)?

Loops in Path Coverage

Given that exhaustive testing is impossible, testers often rely on the *loop boundary adequacy criterion* to decide when to stop testing a loop. A test suite satisfies this criterion if and only if for every loop:

- A test case exercises the loop zero times.
- A test case exercises the loop once.
- A test case exercises the loop multiple times.

The idea behind the criterion is to make sure the program is tested when the loop is never executed (does the program behave correctly when the loop is simply 'skipped'?), when it only iterates once (as we empirically know that algorithms may not handle single cases correctly), and many times.

Pragmatically speaking, the main challenge comes when devising the test case for the loop being executed multiple times. Should the test case force the loop to iterate for 2, 5, or 10 times? That requires a good understanding of the program/requirement itself. Our suggestion for testers is to rely on specification-based techniques. With an optimal understanding of the specs, one should be able to devise good tests for the particular loop.

Modified Condition + Decision Coverage (MC/DC)

MC/DC

Modified condition/decision coverage (MC/DC) looks at the combinations of conditions like path coverage does. However, instead of aiming at testing all the possible combinations, we follow a process in order to identify the "important" combinations. The goal of focusing on these important combinations is to manage the large number of test cases that one needs to devise when aiming for 100% path coverage.

The idea of MC/DC is to exercise each condition in a way that it can, independently of the other conditions, affect the outcome of the entire decision. In short, this means that every possible condition of each parameter must have influenced the outcome at least once.

MC/DC

Assuming the decision block: `A & (B | C)`, MC/DC dictates that:

- For condition A:
  - There must be one test case where A=`true` (say T1).
  - There must be one test case where A=`false` (say T2).
  - T1 and T2 (which we call independence pairs) must have different outcomes (e.g., T1 makes the entire decision to evaluate to `true`, and T2 makes the entire decision to evaluate to `false`, or the other way around).
  - In both test cases T1 and T2, variables B and C should be the same.

MC/DC

Assuming the decision block: `A & (B | C)`, MC/DC dictates that:

- For condition A:
    - There must be one test case where A=`true` (say T1).
    - There must be one test case where A=`false` (say T2).
    - T1 and T2 (which we call independence pairs) must have different outcomes (e.g., T1 makes the entire decision to evaluate to `true`, and T2 makes the entire decision to evaluate to `false`, or the other way around).
    - In both test cases T1 and T2, variables B and C should be the same.
- For condition B:
    - There must be one test case where B=`true` (say T3).
    - There must be one test case where B=`false` (say T4).
    - T3 and T4 have different outcomes.
    - In both test cases T3 and T4, variables A and C should be the same.

MC/DC

Assuming the decision block: `A & (B | C)`, MC/DC dictates that:

- For condition A:
    - There must be one test case where A=`true` (say T1).
    - There must be one test case where A=`false` (say T2).
    - T1 and T2 (which we call independence pairs) must have different outcomes (e.g., T1 makes the entire decision to evaluate to `true`, and T2 makes the entire decision to evaluate to `false`, or the other way around).
    - In both test cases T1 and T2, variables B and C should be the same.
- For condition B:
    - There must be one test case where B=`true` (say T3).
    - There must be one test case where B=`false` (say T4).
    - T3 and T4 have different outcomes.
    - In both test cases T3 and T4, variables A and C should be the same.
- For condition C:
    - There must be one test case where C=`true` (say T5).
    - There must be one test case where C=`false` (say T6).
    - T5 and T6 have different outcomes.
    - In both test cases T5 and T6, variables A and B should be the same.

MC/DC

In other words.

- Every *decision* in the program has taken all possible outcomes at least once (decision coverage).

- Every *condition* in a decision in the program has taken all possible outcomes at least once (condition coverage).

- Every condition in a decision has been shown to *independently affect* that decision’s outcome; a condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other possible conditions operators.

MC/DC

Cost-wise, a relevant characteristic of MC/DC coverage is that, supposing that conditions only have binary outcomes (i.e., true or false), the number of tests required to achieve 100% MC/DC coverage is, on average, N+1, where N is the number of conditions in the decision. Note that N+1 is definitely smaller than all the possible combinations (2^N)!

Again, to devise a test suite that achieves 100% MC/DC coverage, we should devise N+1 test cases that, when combined, exercise all the combinations independently from the others.

Then, the question is how to identify such test cases.

MC/DC

Imagine a program that decides whether an applicant should be admitted to the 'University of Porto':

```
void admission(boolean degree, boolean experience, boolean character) {
    if (character & (degree | experience)) {
        System.out.println("Admitted");
    } else {
        System.out.println("Rejected");
    }
}
```

The program takes three booleans as input, which, generically speaking, is the same as the `A & (B | C)`

- Whether the applicant has a good character (`true` or `false`),
- Whether the applicant has a degree (`true` or `false`),
- Whether the applicant has experience in a field of work (`true` or `false`).

If the applicant has good character and either a degree or experience in the field, he/she will be admitted. In any other case the applicant will be rejected.

To test this program, we first use the truth table for `A & (B | C)` to see all the combinations and their outcomes. In this case, we have 3 decisions and 2^3 is 8, therefore we have tests that go from 1 to 8:

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

To test this program, we first use the truth table for `A & (B | C)` to see all the combinations and their outcomes. In this case, we have 3 decisions and 2^3 is 8, therefore we have tests that go from 1 to 8:

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Our goal will be to apply the MC/DC criterion to these test cases, and select N+1, in this case 3+1=4, tests. In this case, the 4 four tests that satisfy that MC/DC coverage is {t2, t3, t4, t6}.

How did we find them? We go test by test, condition by condition.

In t1, we see that Character, Degree, and Experience are all `true` and the Decision (i.e., the outcome of the entire boolean expression) is also `true`. We now look for another test in this table, where only the value of Character is the opposite of the value in t1, but the others (Degree and Experience) are still the same.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

In t1, we see that Character, Degree, and Experience are all `true` and the Decision (i.e., the outcome of the entire boolean expression) is also `true`. We now look for another test in this table, where only the value of Character is the opposite of the value in t1, but the others (Degree and Experience) are still the same. This means we have to look for a test where Character = `false`, Degree = `true`, Experience = `true`, and Decision = `false`. This combination appears in t5.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Thus, we just found a pair of tests (again, called independence pairs), t1 and t5, where Character is the only parameter which changed and the outcome (Decision) changed as well. In other words, a pair of tests where the Character independently influences the outcome (Decision). Let's keep the pair {t1,t5} in our list of test cases.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

We could have stopped here and moved to the next variable. After all, we already found an independence pair for Character. However, finding them all might help us in reducing the number of test cases at the end, as you will see. So let us continue and we look at the next test. In t2, Character = `true`, Degree = `true`, Experience = `false`, and Decision = `true`. We repeat the process and search for a test where Character is the opposite of the value in t2, but Degree and Experience remain the same (Degree = `true`, Experience = `false`). This set appears in t6.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

We could have stopped here and moved to the next variable. After all, we already found an independence pair for Character. However, finding them all might help us in reducing the number of test cases at the end, as you will see. So let us continue and we look at the next test. In t2, Character = `true`, Degree = `true`, Experience = `false`, and Decision = `true`. We repeat the process and search for a test where Character is the opposite of the value in t2, but Degree and Experience remain the same (Degree = `true`, Experience = `false`). This set appears in t6.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

This means we just found another pair of tests, t2 and t6, where Character is the only parameter which changed and the outcome (Decision) changed as well.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Again, we repeat the process for t3 (Character = `true`, Degree = `false`, Experience = `true`) and find that the Character parameter in t7 (Character = `false`, Degree = `false`, Experience = `true`) is the opposite of the value in t3 and changes the outcome (Decision).

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Again, we repeat the process for t3 (Character = `true`, Degree = `false`, Experience = `true`) and find that the Character parameter in t7 (Character = `false`, Degree = `false`, Experience = `true`) is the opposite of the value in t3 and changes the outcome (Decision).

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

For t4 (Character = `true`, Degree = `false`, Experience = `false`). Its pair is t8 (Character = `false`, Degree = `false`, Experience = `false`). Now, the outcome of both tests is the same (Decision = `false`). This means that the pair {t4, t8} does not show how Character can independently affect the overall outcome; after all, Character is the only thing that changes in these two tests, but the outcome is still the same.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

And we now move on from the Character parameter to the Degree parameter. We repeat the same process, but now we search for the opposite value of parameter Degree whilst Character and Experience stay the same.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

For t1 (Charater = `true`, Degree = `true`, Experience = `true`), we search for a test where (Charater = `true`, Degree = `false`, Experience = `true`). This appears to be the case in t3. However, the outcome for both test cases stay the same. Therefore, {t1, t3} does not show how the Degree parameter can independently affect the outcome.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

After repeating all the steps for the other tests we find only {t2, t4} have different values for the Degree parameter where the outcome also changes.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Finally we move to the Experience parameter. As with the Degree parameter, there is only one pair of combinations that will work, which is {t3, t4}.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Finally we move to the Experience parameter. As with the Degree parameter, there is only one pair of combinations that will work, which is {t3, t4}.

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

We highly recommend carrying out the entire process yourself to get a feeling of how the process works!

We now have all the pairs for each of the parameters:

- Character: {t1, t5}, {t2, t6}, {t3, t7}
- Degree: {t2, t4}
- Experience: {t3, t4}

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

We now have all the pairs for each of the parameters:

- Character: {t1, t5}, {t2, t6}, {t3, t7}
- Degree: {t2, t4}
- Experience: {t3, t4}

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

As we can see from the set of tests above, t4 is shared by Degree and Experience (and evaluates the Decision to false). t2 and t3 evaluate the Decision to true and covers Degree and Experience respectively. This means that we have the following combination {t2, t3, t4}.

We now have all the pairs for each of the parameters:

- Character: {t1, t5}, {t2, t6}, {t3, t7}
- Degree: {t2, t4}
- Experience: {t3, t4}

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

As we can see from the set of tests above, t4 is shared by Degree and Experience (and evaluates the Decision to false). t2 and t3 evaluate the Decision to true and covers Degree and Experience respectively. This means that we have the following combination {t2, t3, t4}.

We now have all the pairs for each of the parameters:

- Character: {t1, t5}, {t2, t6}, {t3, t7}
- Degree: {t2, t4}
- Experience: {t3, t4}

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Then, we need to find the appropriate pair for Character. Note that any of them would fit. However, we want to reduce the total amount of tests in the test suite (and again, we know we only need 4 in this case).

We now have all the pairs for each of the parameters:

- Character: {t1, t5}, {t2, t6}, {t3, t7}
- Degree: {t2, t4}
- Experience: {t3, t4}

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

If we were to pick either t1 or t5 we would have to include both t5 and t1, as they are their opposites, but that would unnecessarily increasing our number of tests from 3 to 5. In order to keep our test cases in accordance to N+1, i.e., 3+1=4, we could either include t6 or t7, as their opposites (t2 and t3, respectively) are already included in our test cases. Randomly, we pick t6.

We now have all the pairs for each of the parameters:

- Character: {t1, t5}, {t2, t6}, {t3, t7}
- Degree: {t2, t4}
- Experience: {t3, t4}

Tests	Character	Degree	Experience	Decision
t1	T	T	T	T
t2	T	T	F	T
t3	T	F	T	T
t4	T	F	F	F
t5	F	T	T	F
t6	F	T	F	F
t7	F	F	T	F
t8	F	F	F	F

Therefore, the tests that we need for 100% MC/DC coverage are {t2, t3, t4, t6}. These are the only 4 tests we need. This is indeed cheaper when compare to the 8 tests we would need for path coverage.

MC/DC

Some extra details about the MC/DC coverage:

- We have applied what we call unique-cause MC/DC criteria. We identify an independence pair (T1, T2), where only a single condition changes between T1 and T2, as well as the final outcome. That might not be possible in all cases. For example, `(A and B) or (A and C)`. Ideally, we would demonstrate the independence of the first A, B, the second A, and C. It is however impossible to change the first A and not change the second A. Thus, we can not demonstrate the independence of each A in the expression. In such cases, we then allow A to vary, but we still fix all other variables (this is what is called masked MC/DC).

- It might not be possible to achieve MC/DC coverage in some expressions, e.g., `(A and B) or (A and not B)`. While the independence pairs (TT, FT) would show the independence of A, there are no pairs that show the independence of B. While logically possible, in such cases, we recommend the developer to revisit the (degenerative) expression as it might had been poorly designed. In our example, the expression could be reformulated to simply `A`.

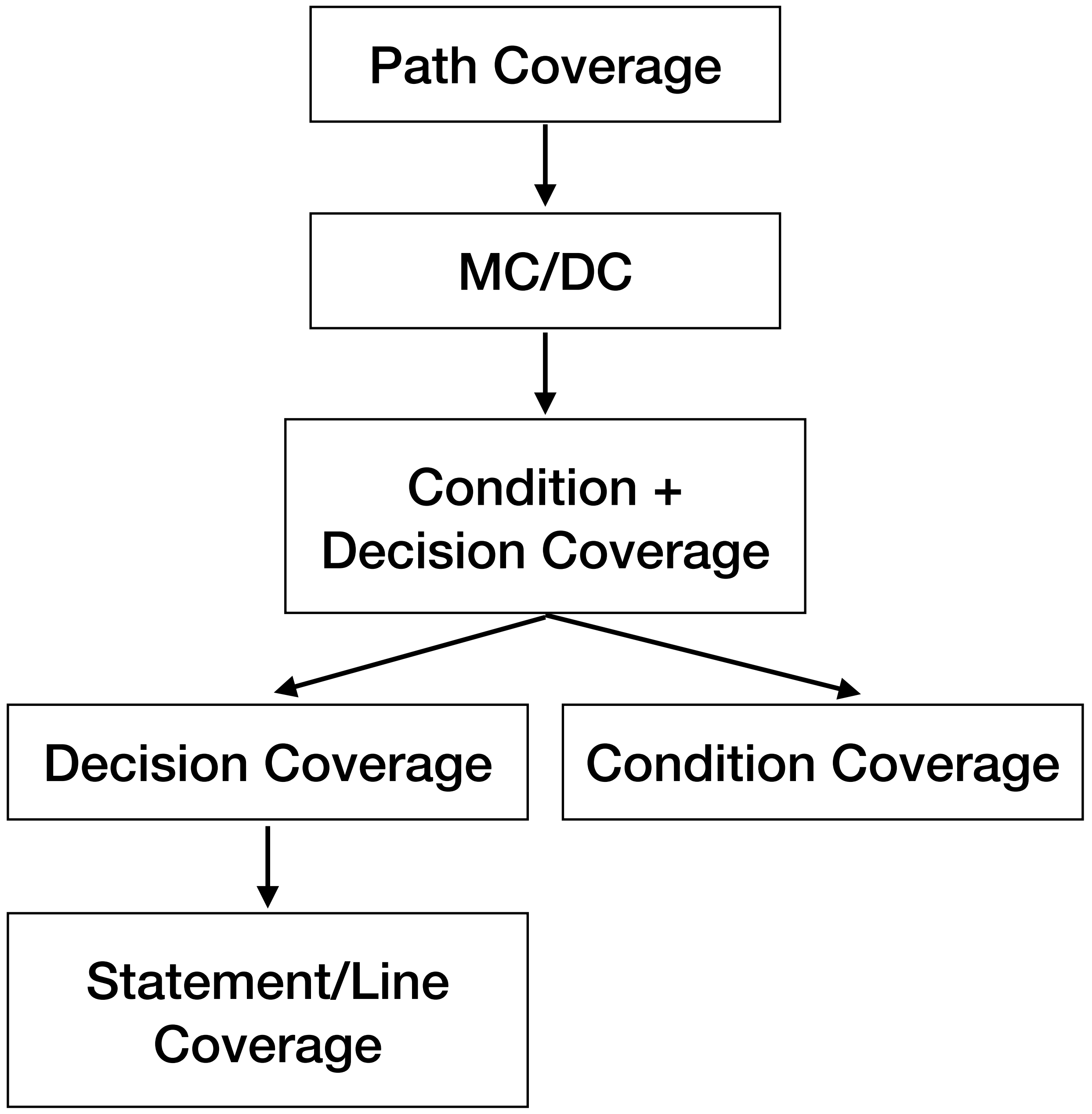
- Mathematically speaking, N+1 is the minimum number of tests required for MC/DC coverage (and 2^N the theoretical upper bound). However, empirical studies have shown that N+1 is often the required number of tests.

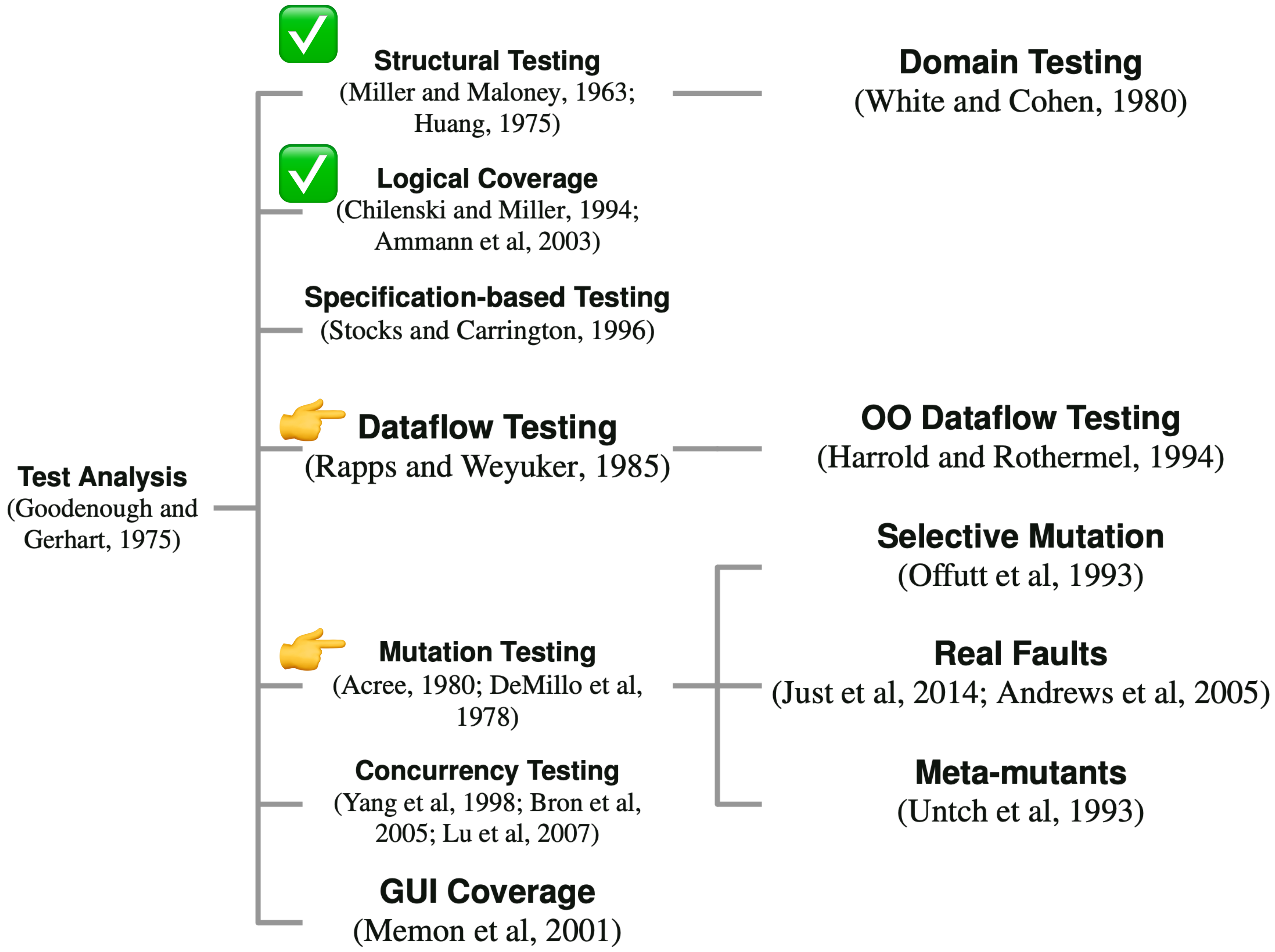
Criteria subsumption

You might have noticed that the criteria we studied became more rigorous and demanding throughout this course. We started our discussion with line coverage. Then we discussed branch coverage, and we noticed that we could generate more tests if we focused on branches. Then, we discussed branch + condition coverage, and we noticed that we could generate even more tests, if we also focused on the conditions.

There is a relationship between these criteria. Some strategies subsume other strategies. Formally, a strategy X subsumes strategy Y if all elements that Y exercises are also exercised by X. You can see in the figure the relationship between the coverage criteria we have studied.

For example, in the figure, one can see that branch coverage subsumes line coverage. This means that 100% of decision coverage always implies 100% line coverage. However, 100% line coverage does not imply 100% decision coverage. Moreover, 100% of decision + condition coverage always implies 100% decision coverage and 100% line coverage.




✅ Structural Testing (Miller and Maloney, 1963; Huang, 1975)
Domain Testing (White and Cohen, 1980)
✅ Logical Coverage (Chilenski and Miller, 1994; Ammann et al, 2003)
Specification-based Testing (Stocks and Carrington, 1996)
👉 Dataflow Testing (Rapps and Weyuker, 1985)
OO Dataflow Testing (Harrold and Rothermel, 1994)
Test Analysis (Goodenough and Gerhart, 1975)
Selective Mutation (Offutt et al, 1993)
👉 Mutation Testing (Acree, 1980; DeMillo et al, 1978)
Real Faults (Just et al, 2014; Andrews et al, 2005)
Concurrency Testing (Yang et al, 1998; Bron et al, 2005; Lu et al, 2007)
Meta-mutants (Untch et al, 1993)
GUI Coverage (Memon et al, 2001)

Tools

- VerifySoft Tech, https://www.verifysoft.com/en_ctc_java_csharp.html
- CodeCover, http://codecover.org
- (for C code) COEMS MC/DC, https://www.coems.eu/mc-dc/
- RapitaSystems, https://www.rapitasystems.com/mcdc-coverage
- EclEmma: Java Code Coverage for Eclipse
  https://www.eclemma.org
- JaCoCo
  https://www.jacoco.org/jacoco
- Cobertura: A code coverage utility for Java
  http://cobertura.github.io/cobertura/

The effectiveness of Structural Testing

A common question among practitioners is whether structural testing or, in their words, test coverage, matters. While researchers have not yet found a magical coverage number that one should aim for, they have been finding interesting evidence pointing towards the benefits of performing structural testing.

We quote two of these studies:

- Hutchins et al.: "Within the limited domain of our experiments, test sets achieving coverage levels over 90% usually showed significantly better fault detection than randomly chosen test sets of the same size. In addition, significant improvements in the effectiveness of coverage-based tests usually occurred as coverage increased from 90% to 100%. However, the results also indicate that 100% code coverage alone is not a reliable indicator of the effectiveness of a test set."
- Namin and Andrews: "Our experiments indicate that coverage is sometimes correlated with effectiveness when [test suite] size is controlled for, and that using both size and coverage yields a more accurate prediction of effectiveness than [test suite] size alone. This in turn suggests that both size and coverage are important to test suite effectiveness."

For interested readers, an extensive literature review on the topic can be found in Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys (csur), 29(4), 366-427.

Structural Testing vs Structural Coverage

A common misconception among practitioners to confuse structural testing with structural coverage.

Structural testing means leveraging the structure of the source code to systematically exercise the system under test. When compared to specification-based testing, we note that structural testing is:

- More objective. In other words, it does not depend on the opinions and experience of the tester. While different testers might come up with different specification-based tests, they would come with similar structural tests.
- Implementation-aware. Implementations can vary from the specifications. After all, there are so many ways one can implement a program. Structural testing enables testers to explore the precise implementation.

On the other hand, structural testing is a check and balance (as Chilenski puts it) on the specification-based tests. Structural testing confirms and complements the tests that we derived before.

It is common to see developers running their coverage tools and writing tests for the outputs they observe. Developers that are mostly focused on (simply) achieving high structural coverage are missing the main point of structural testing.

Again, structural testing should complement your requirements-based testing. As Chilenski suggests (see Figure 3 in his paper), the first step of a tester should be to derive test cases out of any requirements-based technique. Once requirements are fully covered, testers then perform structural testing to cover what is missing from the structural-point of view. Any divergences should be brought back to the requirements-based testing phase: why did we not find this class/partition before? Once requirements and structure are covered, one can consider the testing phase done.

Therefore, do not aim at 100% coverage. Use structural testing to complement your specification-based tests.

References

- Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6.
- Chapter 12 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
- Chapter 5 of the Practical software testing a process-oriented approach. Ilene Burnstein, 2002.
- Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys.