STRUCTURAL TESTING & TEST ADEQUACY – PART 1

How good are your tests?

Statement, branch, and condition-based criteria

One sentence definitions

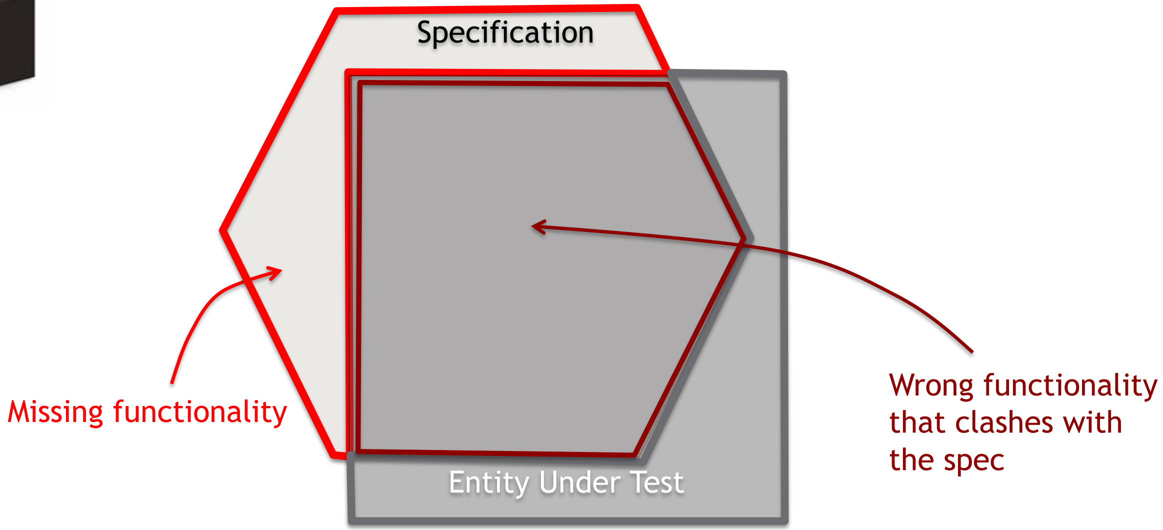
- **Specification-based (blackbox) testing** ignores the internal mechanism of a system or component and focuses solely on the outputs [side effects] generated in response to selected inputs and execution conditions (IEEE, 1990).

Structural (whitebox) testing takes into account the internal mechanism of a system or component (IEEE, 1990).

- Also known as open-box, clear-box or glass-box.
- Approaches include tracing data and control flow through a program
  - How do we trace? *One way to trace control flow is to compute coverage using existing tests...*

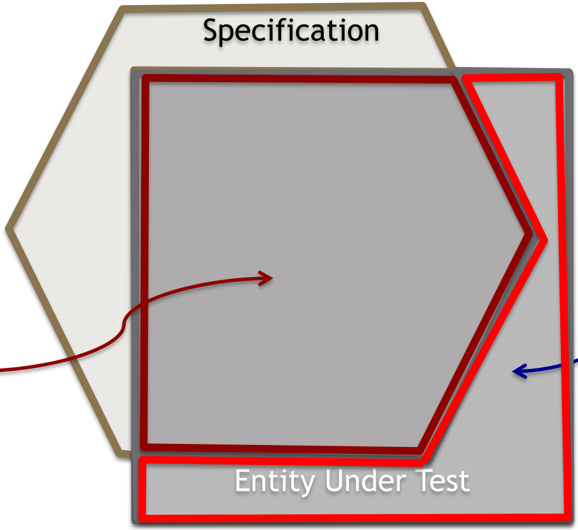
Courtesy of Jeff Jennari

So far with black-box testing, we have focused on the spec



We have been interested in this first!

Now we can use code-focused, structural strategies



Behavior in spec that we forgot or missed during black-box testing (because we optimized)

Unexpected behavior <u>not</u> detectable by black-box testing

Now it's time to be interested in this!

TESTING IS ALL ABOUT COVERAGE

Test adequacy?

Goodness of some notion of "coverage"

Do tests cover all requirements/specs?

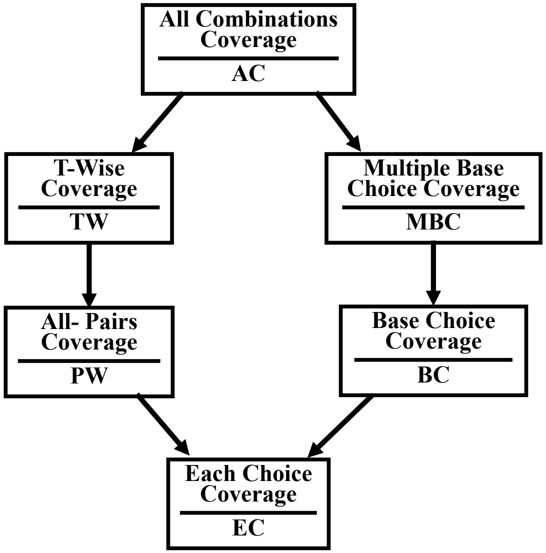
Do tests cover all of codebase's control-flow?

Image: https://www.easypolicy.com/life-insurance/term-insurance/articles/risk-coverage-in-life-insurance.html

We have seen some low-level spec-based adequacy criteria

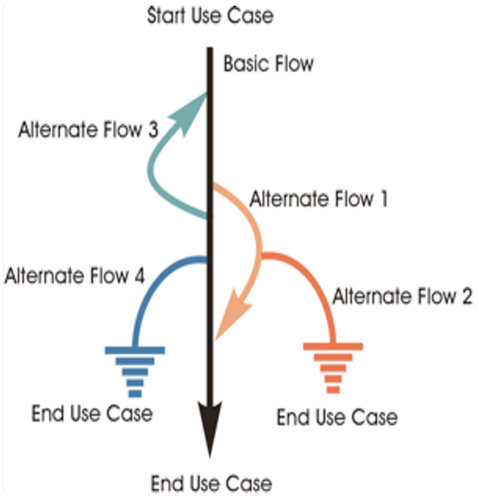
- Corresponding to different combinatorial testing strategies

- Answering the question "How well do the tests **cover** the input space or the spec?"



There is also high-level spec-based adequacy, such as requirements coverage

- Coverage of requirements
  - each use case (or user story) covered by a test case
- For each requirement:
  - each scenario (basic flow and alternative flows) are covered by a test case

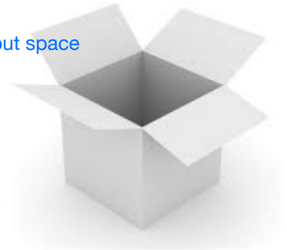


Structural test adequacy answers a different question…

"How well do the tests cover the program code?"

based on a *white-box approach*

Spec test: how well do the tests cover the input space



We’ve already seen one kind of structural adequacy criterion

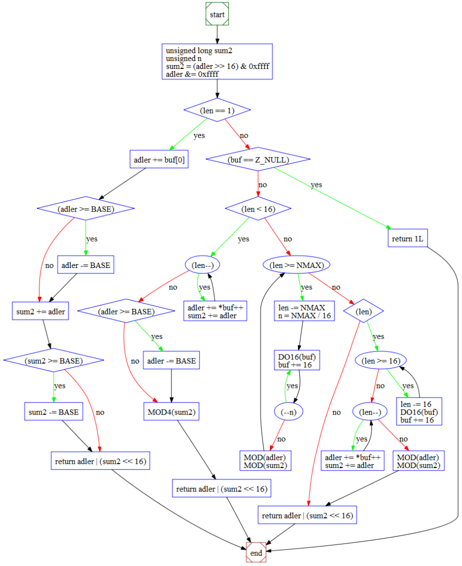
Mutation-based adequacy

Percent mutants killed – *a coverage metric applied with mutation testing*

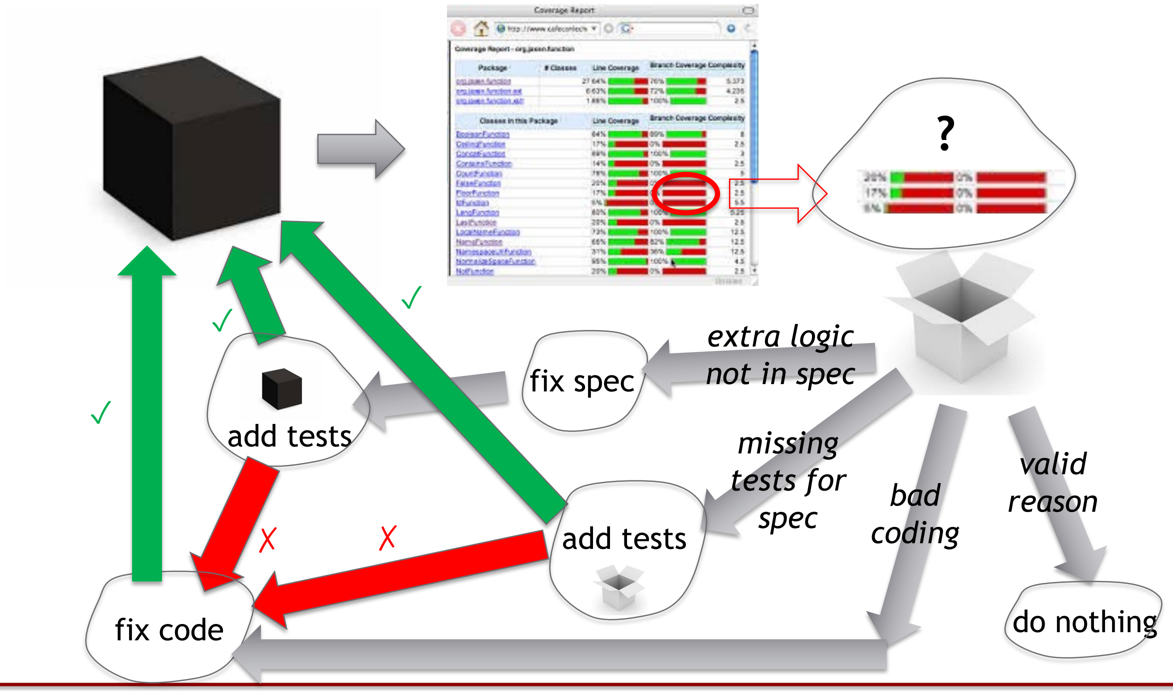
Remaining structural testing approaches try to cover a graph of the program

… our test suites are **good** (**adequate**) to the extent they cover that graph

- Which graph?
  - Control Flow Graph (CFG)
- How?
  - Which elements of the graph do tests touch
    - Nodes and edges

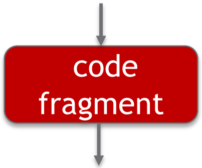


"Behavior-first, coverage-next" flow
("Spec-first, structure-next" flow)

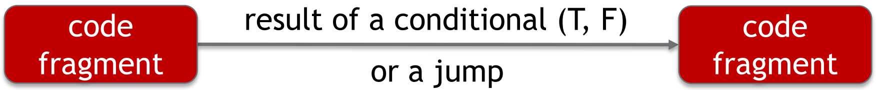


Control Flow Graph (CFG) consists of nodes and directed edges

- nodes = regions of source code
  - maximal program region with a single entry and single exit point (basic block, vertex)



- directed edges = possibility that program execution proceeds from the end of one region directly to the beginning of another (control flow)



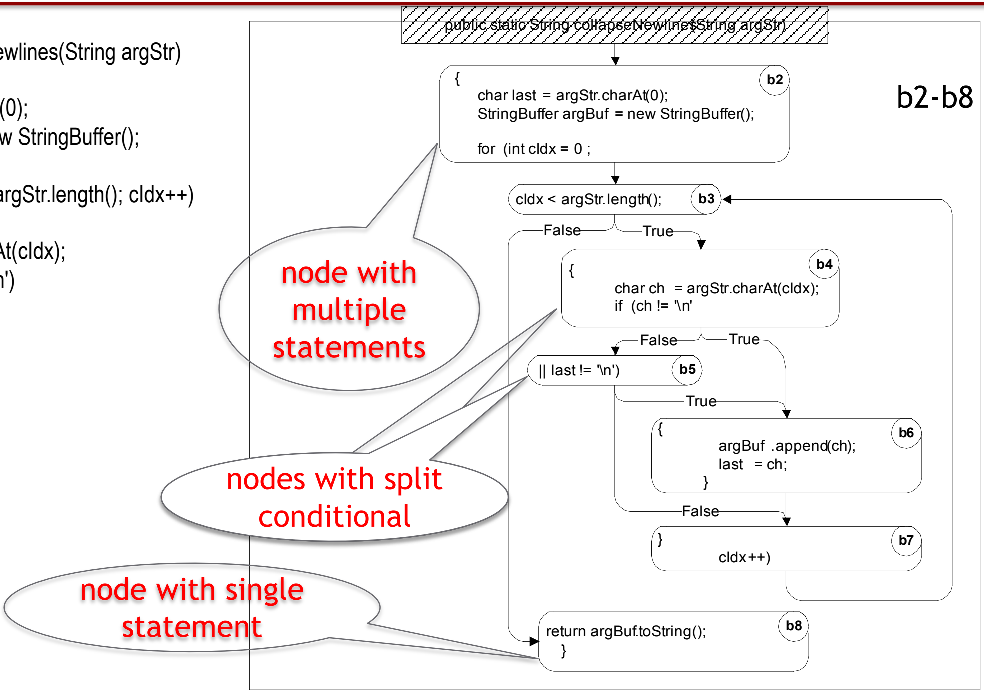
Nodes in a CFG

```
public static String collapseNewlines(String argStr)
  {
     char last = argStr.charAt(0);
     StringBuffer argBuf = new StringBuffer();

     for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
     {
        char ch = argStr.charAt(cIdx);
        if (ch != '\n' || last != '\n')
        {
           argBuf.append(ch);
           last = ch;
        }
     }

     return argBuf.toString();
  }
```

C snippet

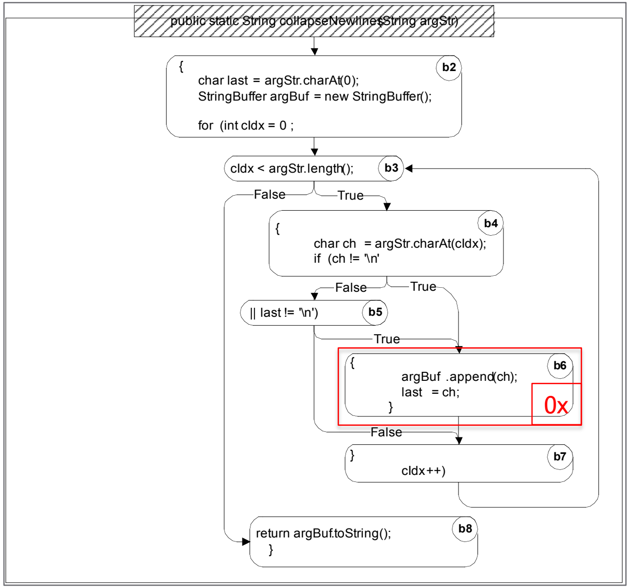


b2-b8 are nodes

Structural testing is using CFG coverage criteria to *assess adequacy of tests* and *enhance the test suite*

One way of answering the question “What is *missing* in our test suite?”

If part of a CFG is not executed (covered) by any test case, faults in that part cannot be exposed



More fine-grained than nodes: Statements vs. Lines

- Statement: elementary programming language construct, often terminated by ‘;’

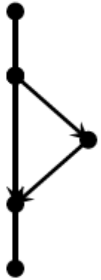
```
char a = '*';
```

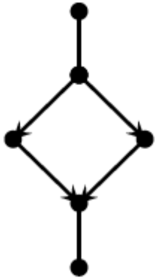
- Line: single line of program code, possibly containing multiple statements

```
a = b + 1; b++;
```

CFG patterns


sequence

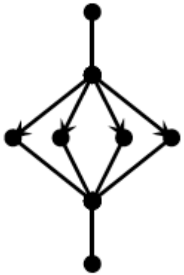

If .. then


If .. then .. else

What about try-catch?


Do .. While
also Repeat.. Until


While .. Do
also For loop


Switch
aka Case

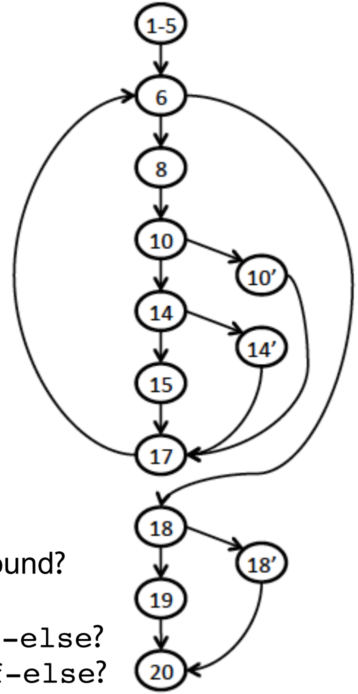
Claire Le Goues (2014)

Binary search CFG

```
1. public int binarySearch(int[] list, int len, int searchItem)
2. {
3.   int first=0, last = len – 1, mid;
4.   boolean found = false;
5.   // Loop until found or end of list.
6.   while(first <= last &&!found) {
7.   // Find the middle.
8.     mid = (first + last) /2;
9.   // Compare the middle item to the search item.
10.    if(list[mid] == searchItem) found = true;
11.    else {
12.     // Not found, readjust search parameters,
13.     // halving the size & start over.
14.      if(list[mid] > searchItem) last = mid -1;
15.      else first = mid + 1;
16.     }
17.    }
18.    if(found) return mid;
19.    else return(-1);
20.}
```

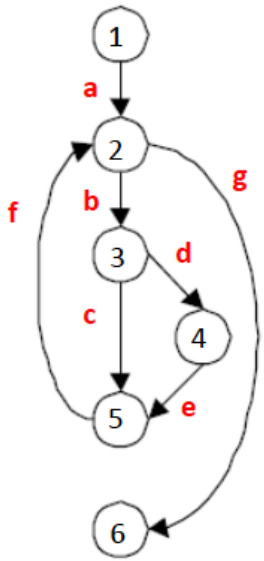
Claire Le Goues (2014)

Where is the initialization found?
Where is the `while`?
Where is the loop's inner `if-else`?
Where is the loop's outer `if-else`?



A CFG may have a small number of nodes and branches, but it may have lots of paths…

n: # loop iterations



n		Path via nodes
0		1,2,6
1		1,2,3,5,2,6
1		1,2,3,4,5,2,6
2		1,2,3,5,2,3,4,5,2,6
2		1,2,3,4,5,2,3,5,2,6
2		1,2,3,5,2,3,5,2,6
2		1,2,3,4,5,2,3,4,5,2,6
20		1,2,3,4,5,2,3,4,5,2,3 ,4,5,2,……….,2,6
20		
	⋮	
	Total number of paths	2,097,151

(n <= 20)

(Still) No guarantees

- Executing all control flow elements does not guarantee finding all faults
  - Even for small programs, we may not be able to execute all paths (can be too many, practically infinite)
  - Execution of a faulty statement may not always result in a failure
    - The state may not be corrupted when the statement is executed with some data values
    - Corrupt state may not propagate through execution to eventually lead to failure
    - Check the “white-box example” from previous lectures!

Structural adequacy
is **not** a definitive measure of
test suite goodness, but it’s a reasonable indicator

Structural adequacy is volatile

What happens when we change the code/implementation without changing the spec?

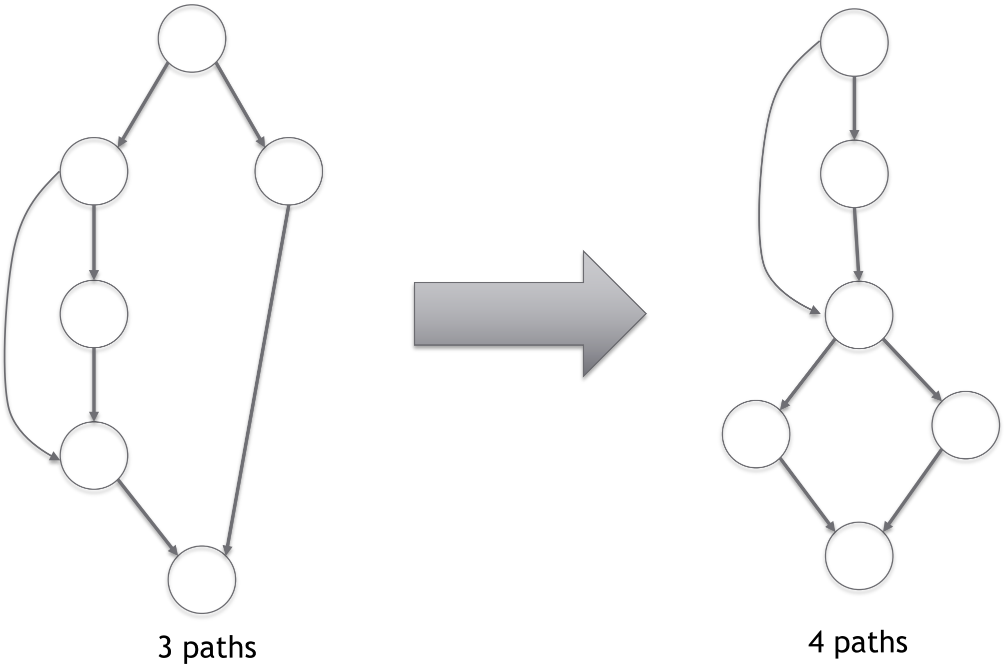
```
if (a >= b) {
   if (a == b) {
     ...
   }
    ...
} else {
    ...
}
```

CFG? How many paths?

```
if (a == b) {
    ...
}
if (a > b) {
    ...
} else {
    ...
}
```

CFG? How many paths?

Structural adequacy



Structural adequacy is volatile

- What happens when we change the code/implementation?

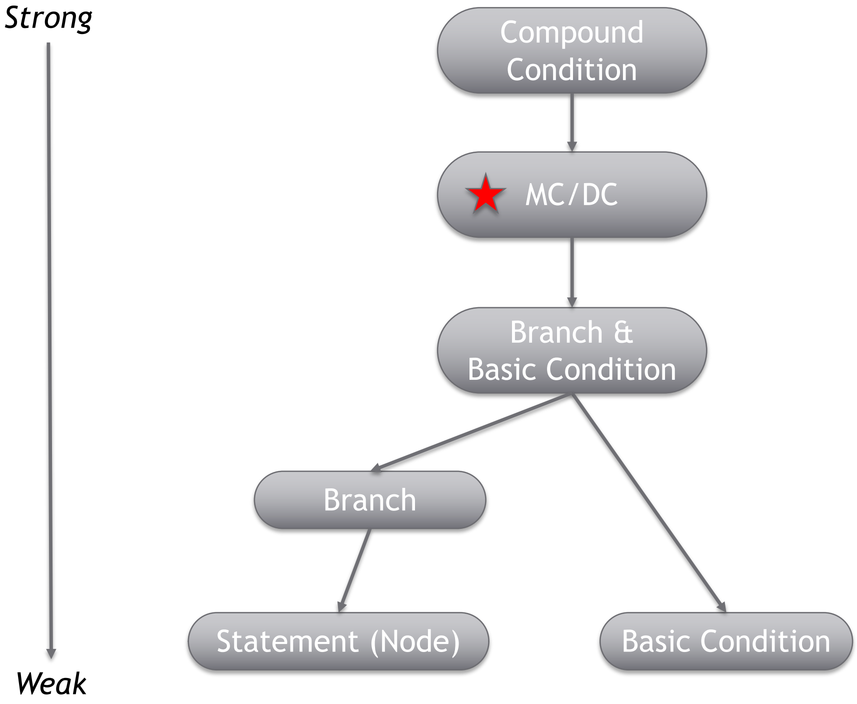
- *Structural test cases may no longer be valid*
  - structural adequacy need to be re-evaluated
  - had 3 paths to cover (3 test cases) before the change, but now have 4 paths to cover (4 test cases)

Structural Coverage Criteria

statements → branches → conditions → paths

and variations in between

CFG Coverage Criteria



*Error in diagram in P&Y 2008

Statement testing: trying to cover all statements

- Adequacy criterion: each statement must be executed at least once

- Coverage measure:

$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

- Rationale: a fault in a statement can only be revealed by executing the faulty statement

Example (url_decode)

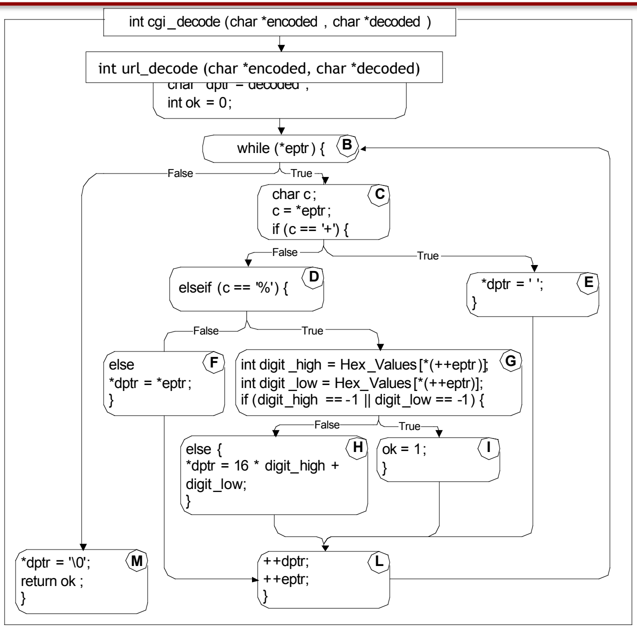
20 statements

S_0 =
{ “”, “test+case%1Dadequacy” }
19/20 = 95% Stmt. Cov.

S_1 =
{ “adequate+test%0Dexecution%7U” }
20/20 = 100% Stmt. Cov.

S_2 =
{ “%3D” , “%A” , “a+b” , “test” }
20/20 = 100% Stmt. Cov.

What’s the statement coverage for **red** & **blue** test cases in S_0?



decodes a string encoded for transmitting in a URL to obtain the original string (string transformation)

Example (url_decode)

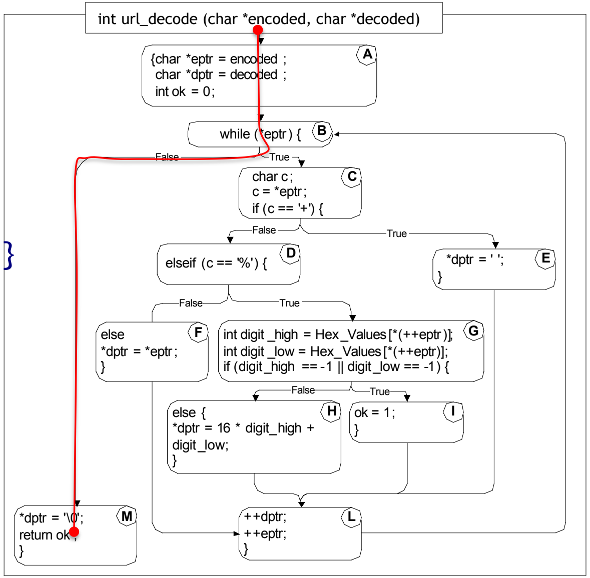
20 statements

S_0 =
{ “”, “test+case%1Dadequacy” }
19/20 = 95% Stmt. Cov.

6/20 = 30% Stmt. Cov.

S_1 =
{ “adequate+test%0Dexecution%7U” }
20/20 = 100% Stmt. Cov.

S_2 =
{ “%3D” , “%A” , “a+b” , “test” }
20/20 = 100% Stmt. Cov.



A, B, M

Example (url_decode)

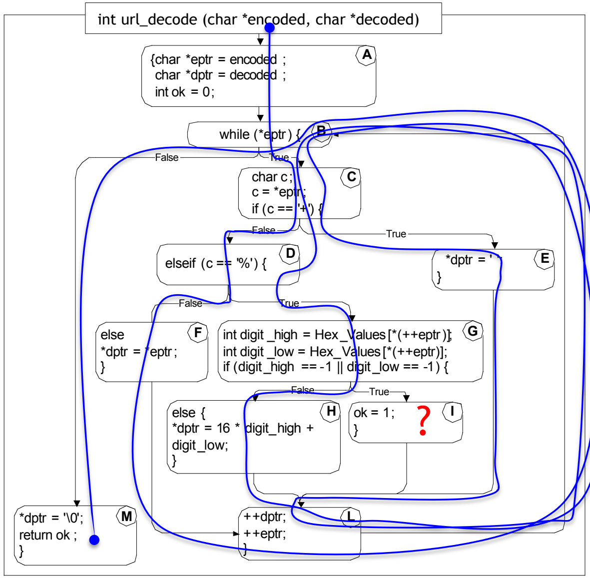
20 statements

S_0 =
{ “” , “test+case%1Dadequacy” }
19/20 = 95% Stmt. Cov.

19/20 = 95% Stmt. Cov.

S_1 = { “adequate+test%0Dexecution%7U” }
20/20 = 100% Stmt. Cov.

S_2 =
{ “%3D” , “%A” , “a+b, “test” }
20/20 = 100% Stmt. Cov.



A, B, C, D, F, L, B, C, E, L, B, C, D, G, H, L, B, M

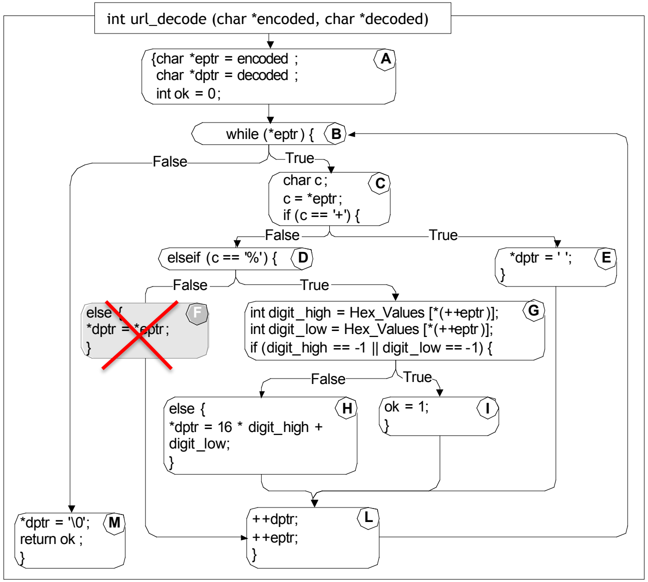
1 2 3 4

Coverage is *not* "test suite size"

- Coverage does not depend on the number of test cases
  - S_0 vs. S_1 : $S_1 >_{coverage} S_0$, *but* $|S_1| < |S_0|$
  - S_1 vs. S_2 : $S_2 =_{coverage} S_1$, *but* $|S_2| > |S_1|$
- From black-box testing, we know that having more tests just for the sake of more tests is not the goal, and may not be effective at all
- In white-box testing, minimizing test suite size is not the ultimate goal either
  - remember: "tests should have a single reason to fail"
  - small test cases make failure diagnosis easier
  - a failing test case in S_2 (many small tests) gives more information for fault localization than a failing test case in S_1 (one large test)

“All statements” can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose programmer made a mistake
  - Forgot block F



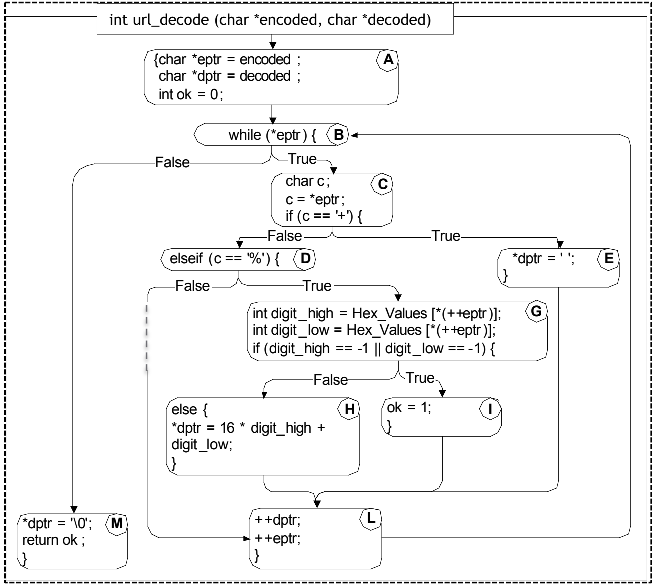
“All statements” can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose block F were missing
  - Statement adequacy would not require *False* branch from D to L

S_3 = { “” , “+%0D+%4J” }

100% Stmt. Cov.

But no *False* branch from D to L

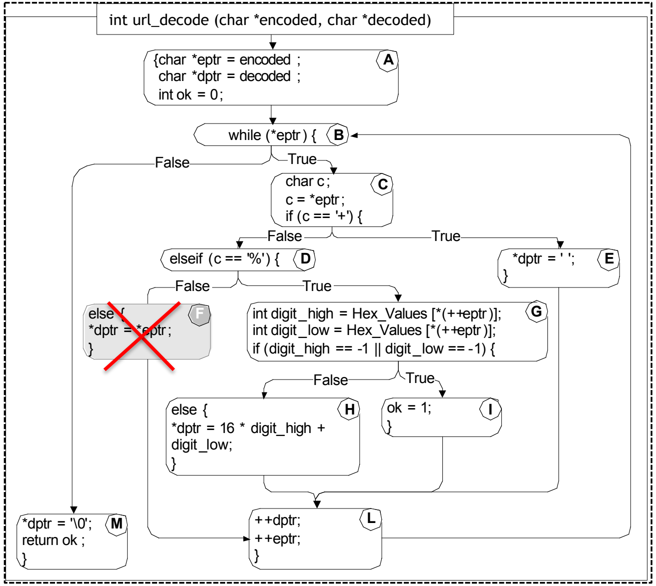


“All statements” can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose block F were missing
  - Statement adequacy would not require *False* branch from D to L

S_3 = { “” , “+%0D+%4J” }
100% Stmt. Cov.

Statement coverage would not detect missing logic here!



Branch (edge or decision) testing

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage measure:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

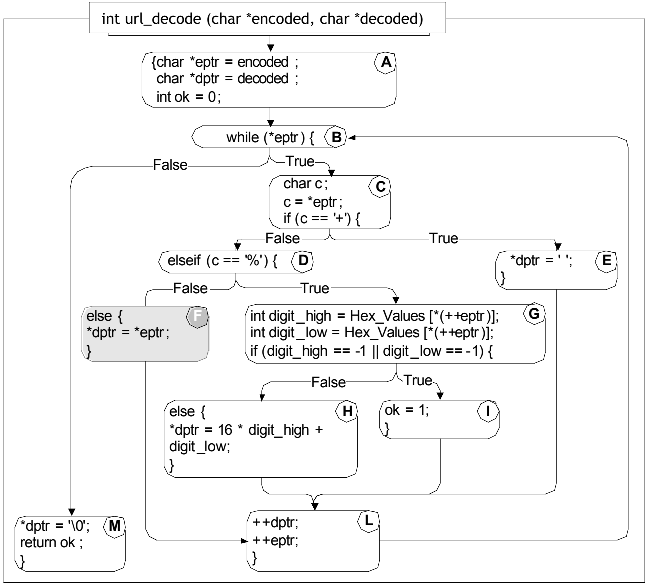
S_3 = { “” , “+%0D+%4J” }

100% Stmt. Cov., 88% Branch Cov. (7/8 branches)

S_2 = { “%3D” , “%A” , “a+b” , “test” }

100% Stmt. Cov., 100% Branch Cov. (8/8 branches)

When counting branches, we only count the forks



(*count only forks, do not count joins or single edges*)

8 branches

Statements are weaker than branches

- Traversing all edges of a graph causes all nodes to be visited

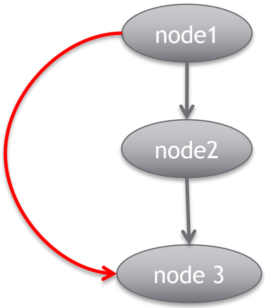
100% Branch coverage	$\rightarrow$	100% Statement coverage
Branch adequacy		Statement adequacy

- The converse is not true (see S_3)

100% Statement coverage	$\not\rightarrow$	100% Branch coverage
Branch adequacy		Statement adequacy

Branches vs. statements

The difference stems from control structures that have a “jump” to an exit node following an entry node:



If (node1) then {

node2

}
node3

How many branches according to branch counting rule?

2 branches

Can have full node coverage without full branch coverage!

"All branches" can still miss conditions that harbor faults

- Sample fault: missing unary negation operator
- if (digit_high == 1 || digit_low == -1)
- Branch adequacy criterion can be satisfied by varying only digit_low (fix digit_high = 0)
  - Two test cases T1, T2 with full branch coverage:
    - T1: digit_high = 0, digit_low = -1
    - T2: digit_high = 0, digit_low = 0

Correct Program

```
bool err(int digit_high, int digit_low) {
  if (digit_high == -1 || digit_low == -1) {
    abort_flight();
    return true;
  }
  return false;
}
```

100% branch coverage -- all good!

- T1: `assertTrue(err(0, -1))` pass
- T2: `assertFalse(err(0, 0))` pass

Incorrect Program

```
bool err(int digit_high, int digit_low) {
  if (digit_high == 1 || digit_low == -1) {
    abort_flight();
    return true;
  }
  return false;
}
```

100% branch coverage, but cannot detect fault!

- T1: `assertTrue(err(0, -1))` pass
- T2: `assertFalse(err(0, 0))` pass

Missing:

- T3: `assertTrue(err(-1, 0))` fail

Compound conditions require more than branch coverage to be fully covered

(digit_high == 1 || digit_low == -1) is a ***compound*** (***complex***) condition

basic condition ***basic*** condition

Decision
= Outcome of a Condition
= Branch

We could cover all combinations of all basic conditions in each decision

- **Compound condition adequacy**
  – Covers all possible evaluations of compound conditions within each decision
  – Covers all branches of underlying decision tree

Similar to **All-Choice** in black-box testing

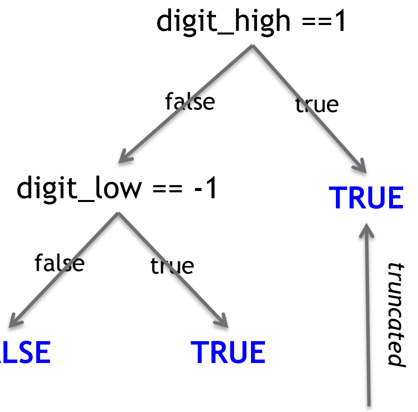
Decision tree for compound condition

(digit_high == 1 || digit_low == -1)

Evaluated from left to right

3 combinations (one for each leaf):

- T, don't care
- F, T
- F, F



Short-circuit evaluation implies we don't need all combinations

Compound conditions have exponential complexity (worst case)

```
(((a || b) && c) || d) && e
```

Test Case	a	b	c	d	e
(1)	T	–	T	–	T
(2)	F	T	T	–	T
(3)	T	–	F	T	T
(4)	F	T	F	T	T
(5)	F	F	–	T	T
(6)	T	–	T	–	F
(7)	F	T	T	–	F
(8)	T	–	F	T	F
(9)	F	T	F	T	F
(10)	F	F	–	T	F
(11)	T	–	F	F	–
(12)	F	T	F	F	–
(13)	F	F	–	F	–

Worst case:
n conditions → 2^n combinations

(here: worst case is $2^5 = 32$ combinations)

short-circuit evaluation creates "don't care" entries which often reduce complexity to a more manageable number, but this is not guaranteed

We could make covering conditionals weak: Basic condition testing

- Adequacy criterion: each basic condition must be executed with both outcomes at least once in some test case

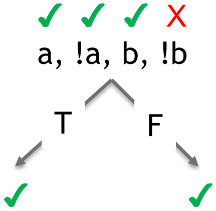
- Coverage measure:

$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$

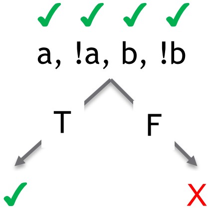
Similar to **Each-Choice** in combinatorial testing

Basic condition vs. branch coverage

- Condition: a && b
  - Test cases: (a, b) and (!a, b)
  - Full branch coverage, but not full basic condition coverage (basic condition value !b is not covered)



- Condition: a || b
  - Test cases (a, !b) and (!a, b)
  - Full basic condition coverage, but not full branch coverage (branch for !(a || b) is not covered)

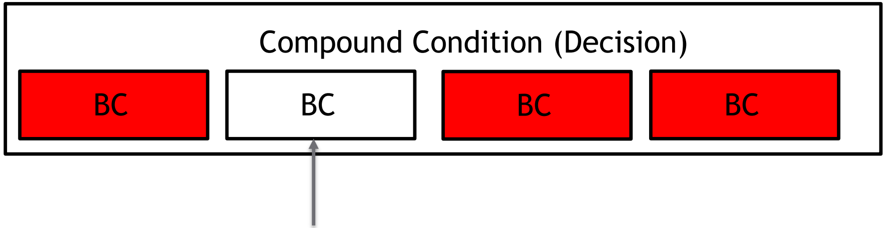


Modified condition/decision coverage (MC/DC)

- Motivation: Effectively test potentially *critical combinations* of conditions, without exponential blowup in test suite size
  - “critical” means: each basic condition shown to independently affect the outcome of each decision (branch)
- Requires:
  - for *each* basic condition C: two test cases
    - values of all *evaluated* conditions except C are kept *constant*
    - test case 1: compound condition evaluates to *true* for one truth value of C
    - test case 2: compound condition evaluates to *false* for the other truth value of C

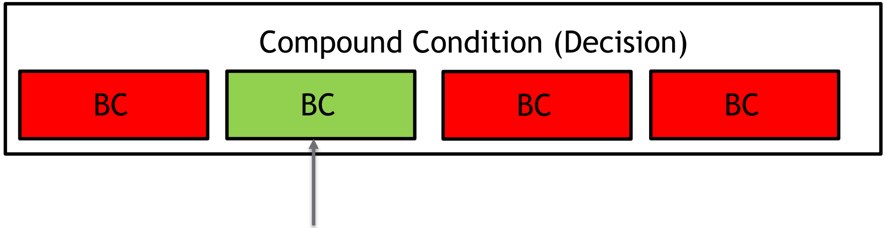
Modified condition/decision (MC/DC)

- Pick one BC
- Fix truth values of all other BCs



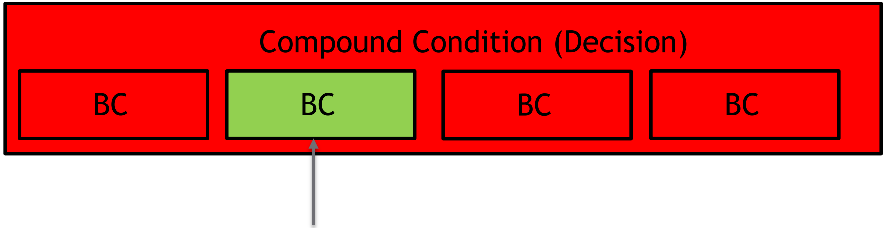
Modified condition/decision (MC/DC)

- Set BC to True



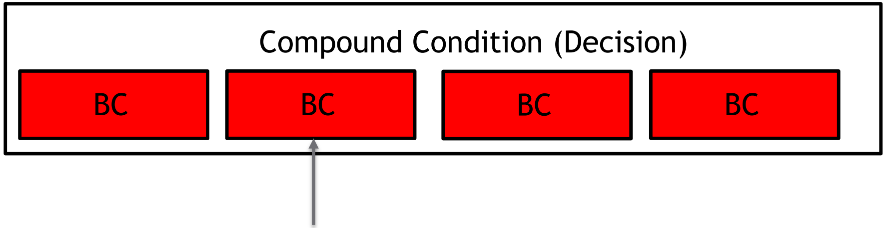
Modified condition/decision (MC/DC)

- Evaluate CC



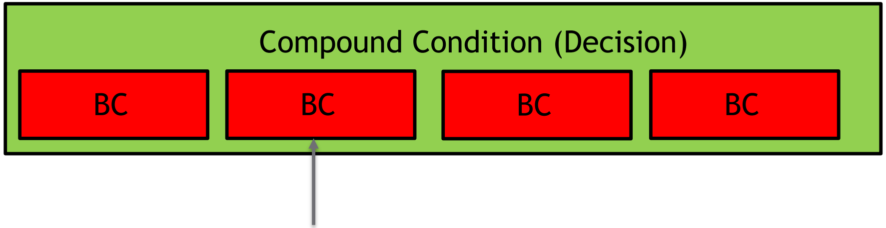
Modified condition/decision (MC/DC)

- Flip truth value of BC to False



Modified condition/decision (MC/DC)

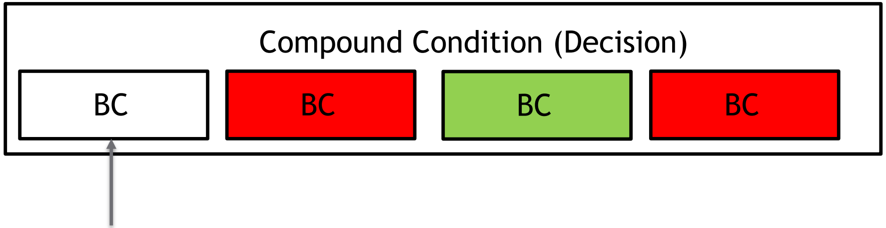
- Evaluate CC: it should switch truth value



It worked! Move on the next BC!

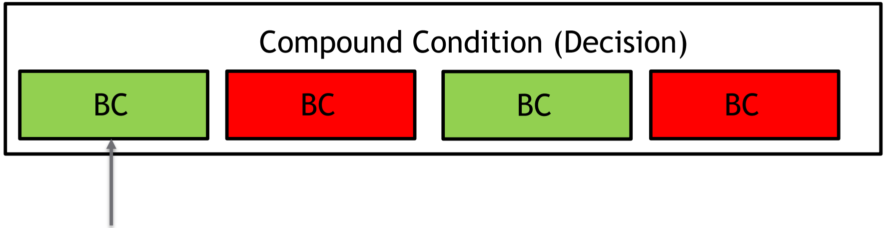
Modified condition/decision (MC/DC)

- Pick another BC
- Fix truth values of all other BCs



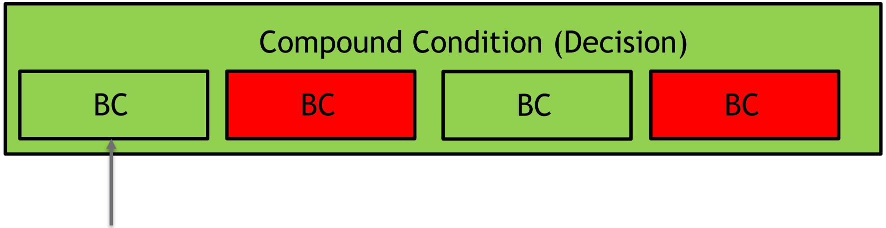
Modified condition/decision (MC/DC)

- Set BC to True



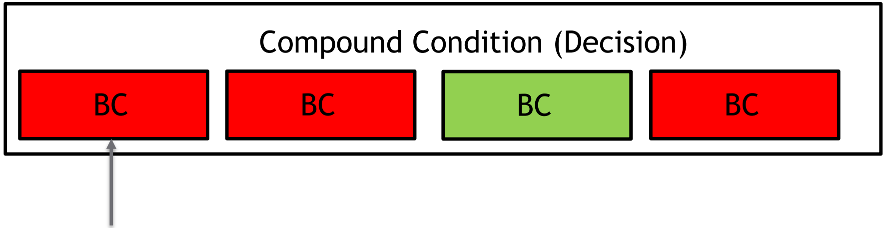
Modified condition/decision (MC/DC)

- Evaluate CC



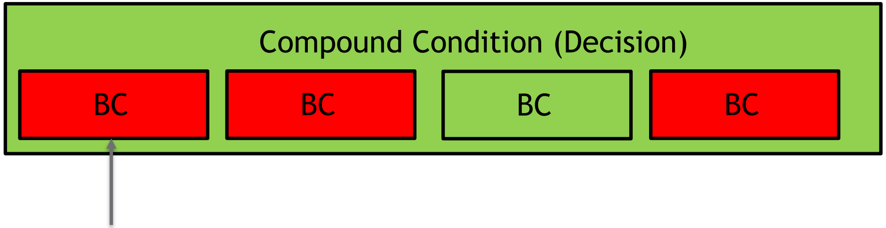
Modified condition/decision (MC/DC)

- Flip truth value of BC to False



Modified condition/decision (MC/DC)

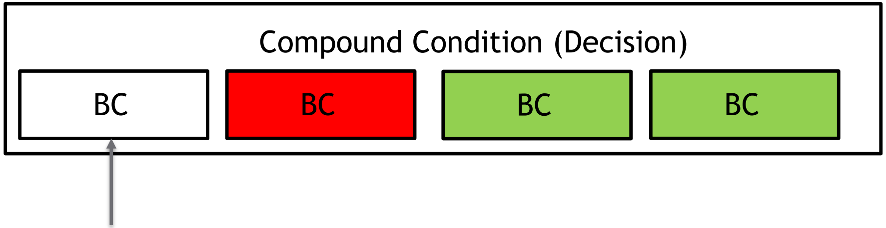
- Evaluate CC: it should switch truth value



Oh no, it didn't work!
I must try other values for the other conditions!

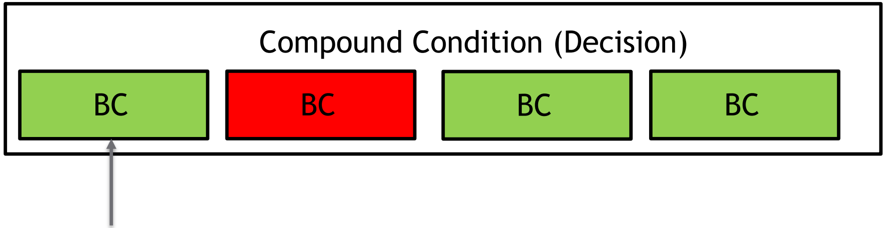
Modified condition/decision (MC/DC)

- Change fixed truth values of all other BCs and try again



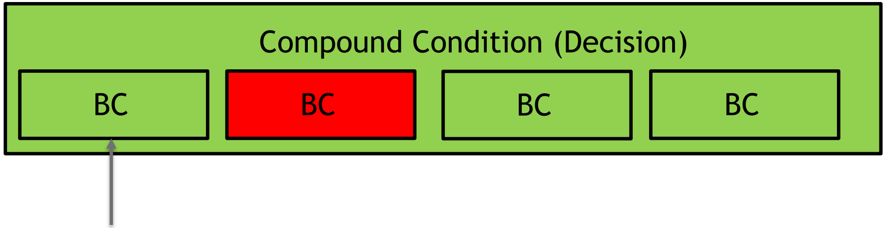
Modified condition/decision (MC/DC)

- Set BC to True



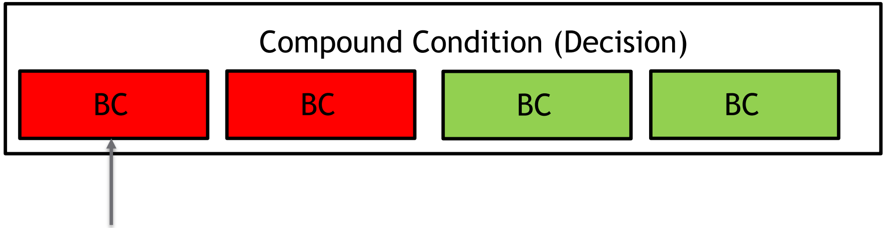
Modified condition/decision (MC/DC)

- Evaluate CC



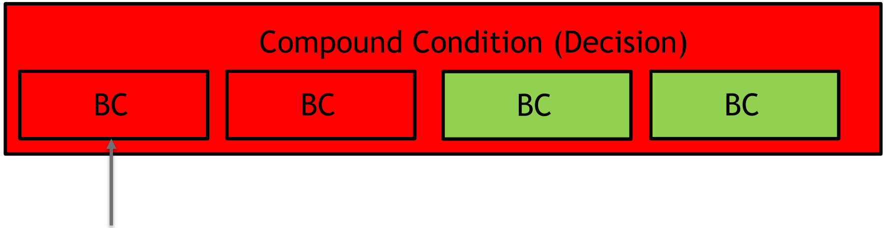
Modified condition/decision (MC/DC)

- Flip truth value of BC to False



Modified condition/decision (MC/DC)

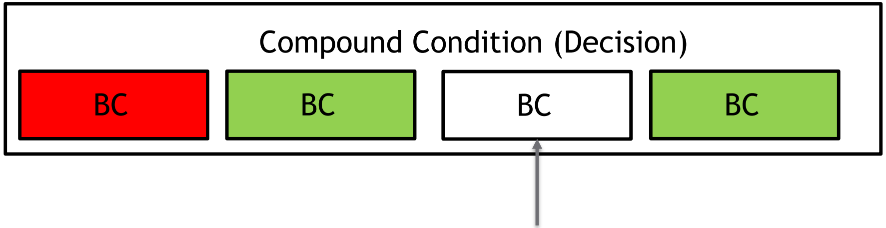
- Evaluate CC: it should switch truth value



It worked! Move on the next BC!

Modified condition/decision (MC/DC)

- Pick another BC
- Fix truth values of all other BCs



- Repeat same process until all BCs covered in this way

MC/DC: has linear complexity

N + 1 test cases for compound condition with *N* basic conditions

```
(((a || b) && c) || d) && e
```

Braces are important!
Never forget them!

Test Case	a	b	c	d	e	decision
(1)	T	–	T	–	T	T
(2)	F	T	T	–	T	T
(3)	T	–	F	T	T	T
(6)	T	–	T	–	F	F
(11)	T	–	F	F	–	F
(13)	F	F	–	F	–	F

6

b, c, d, e constant or – (don't care)

Dashed lines:
MC/DC satisfaction for condition **a**

Underlined values independently affect the output of the decision when other values are constant

Homework

- *N* + 1 test cases for compound condition with *N* basic conditions

```
(((a || b) && c) || d) && e
```

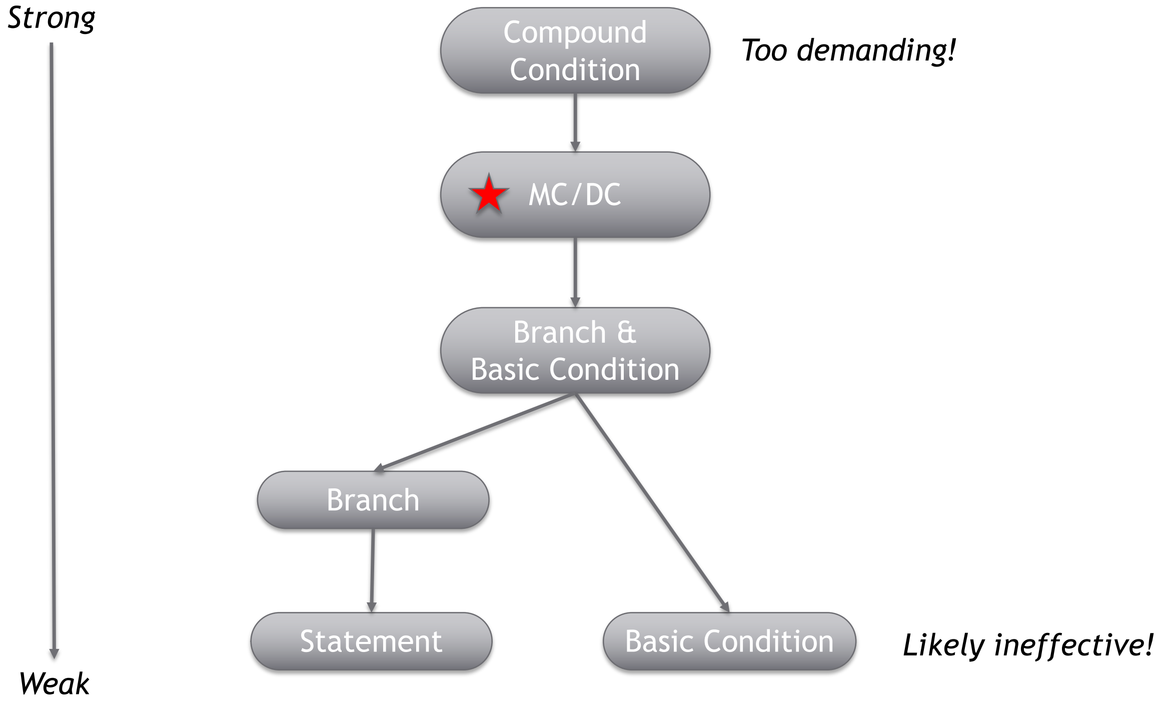
Test Case	a	b	c	d	e	decision
(1)	T	F	T	F	T	T
(2)	F	T	T	F	T	T
(3)	T	F	F	T	T	T
(6)	T	F	T	F	F	F
(11)	T	F	F	F	T	F
(13)	F	F	T	F	T	F

Check MC/DC satisfaction for conditions
`b, c, d, e`

MC/DC
"basic condition coverage done smartly"

- MC/DC implies
  - basic condition coverage (C)
  - branch coverage (DC)
  - plus one additional condition (M):
    every condition must *independently affect* the decision's output
- It is *weaker than* compound condition but *stronger than* all other criteria discussed so far
  - stronger than statement, branch, and basic condition coverage

CFG coverage so far



*Error in diagram in P&Y 2008

MC/DC is a good balance between effectiveness and test suite size

…similar to **All-Pairs** in combinatorial testing

…widely used in mission/safety-critical software)

…required by the RTCA/DO-178B* standard (US) and European equivalent

*Airborne systems software certification

How branch coverage is treated in Java coverage tools

?

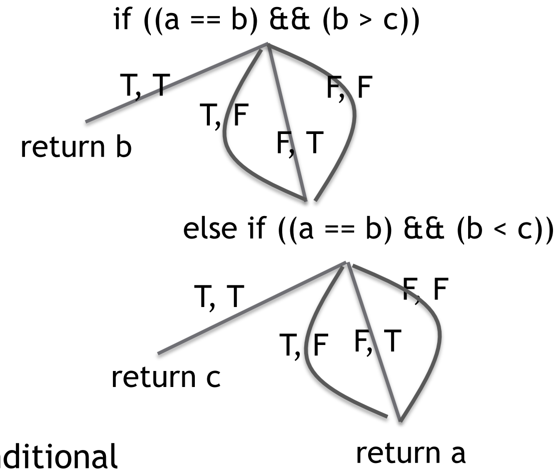
1 of 4 branches missed

```java
public static int foo(int a, int b, int c) {
    if ((a == b) && (b > c)) {
        return b;
    } else if ((a == b) && (b < c)) {
        return c;
    }
    else { return a;}
}
}
```

1 of 4 branches missed

How branch coverage is treated in Java coverage tools

```java
    public static int foo(int a, int b, int c) {
        if ((a == b) && (b > c)) {
            return b;
        } else if ((a == b) && (b < c)) {
            return c;
        }
        else { return a;}
    }
}
```



Reporting compound condition coverage for each conditional structure individually!

Branch coverage %: calculated in different (and sometimes mysterious) ways in different tools!
Not always clear how overall number of branches are calculated.
But always: 100% tool branch coverage => 100% real branch coverage

How many MC/DC test cases for this conditional?

?

(a == b || a == c) && (c > d))

How many MC/DC test cases for this conditional?

(a == b || a == c) && (c > d))

3 + 1 = 4

Homework: Derive them!

MC/DC example

(a && b) || (c && d)

#	a	b	c	d	Decision	covered
1	T	T	-	-	T	
2	F	-	F	-	F	a (1)
3	T	F	F	-	F	b (1)
4	T	F	T	T	T	c (3)
5	T	F	T	F	F	d (4)

This solution is not unique: there can be multiple MC/DC solutions for the same compound condition, or there could be no full MC/DC solution (not possible to MC/DC-cover all basic conditions)!

How many MC/DC test cases for a given program?

How many test cases does a method with two loops and two if-then blocks require for full MC/DC coverage?