Code Coverage

Code Coverage

- Code coverage model calls out the parts of an implementation that must be exercised
  - It is a metric
- Coverage is an **adequacy criterion**
  - A test suite is **adequate** if all the elements to be exercised have been exercised.
- A coverage model x is said to **subsume** some other model y if all elements that y exercises are also exercised by x
  - **Branch coverage subsumes statement coverage**

Role of code coverage

- Represents a set of requirements that a test suite should fulfill

- Usually, there is more than one **adequate** test suite

- Two main goals:
  - Adequacy: Have I got enough tests?
  - Guidance: Where should I test more?

Role of code coverage - 2

- Do not use a coverage model as a test model!
  - Use coverage to analyse test suite adequacy

- Coverage reports can:
  - Point out a grossly inadequate test suite
  - Suggest the presence of surprises
  - Help to identify implementation constructs that may require implementation-based test design
- Usually, very hard to achieve more than 90%
- Must be computed by a tool

Code coverage example

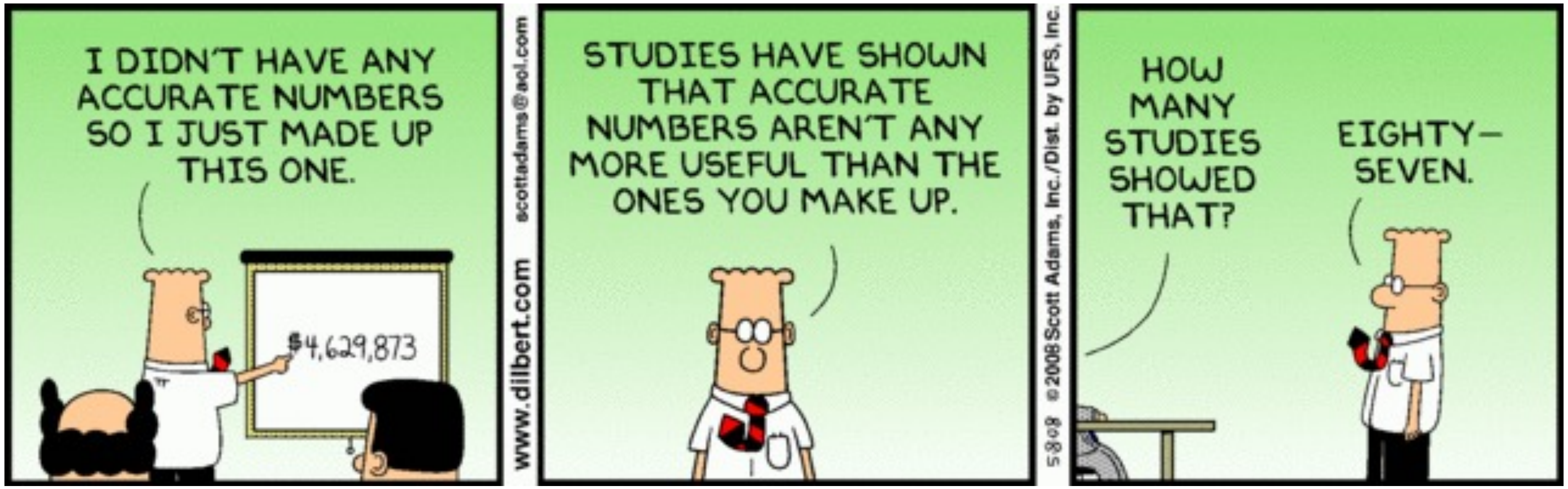
- Coverage achieved by functional testing only

System under test	Segment	Branch	p-use	c-use
TEX	85	72	53	48
AWK	70	59	48	55

- It proves how difficult it is to test a complex system without explicit feedback
- 100% of coverage unattainable due to infeasible paths, dead code, and exception handling – 10 to 15%

Coverage value

$$\text{Coverage value} = \frac{\#\ \text{Covered test requirements}}{\#\ \text{Total test requirements}}$$



Coverage value

$$\text{Coverage value} = \frac{\#\ \text{Covered test requirements}}{\#\ \text{Total test requirements}}$$

- Test suite achieves 100% statement coverage
- This means my program is bug-free?
- **No!**
  - Coverage measures what is *executed*, not what is *checked*

Coverage is dangerous

- Low coverage means code is not well tested
- **But** high coverage does not mean code is well tested
- Should not focus only on coverage metric
  - Developers write test only to satisfy coverage
  - Do not use coverage model as test model

Coverage is useful

- It always tells you the parts of code not exercised yet
- Testing everything a bit is better than not testing most of the program
  - unless you know where the faults are
- To improve, apply a coverage model with a strict criterion
  - Stricter criterion → more tests
  - More tests = more chances of hitting bugs

When can I stop testing?

- Coverage is achieved, but some tests do not pass
  - Non-critical bugs or include bug list with release documentation

- Coverage isn't achieved
  - And all tests pass
    - More tests should be done, unless:
      - infeasible paths exist or cost is prohibitive
  - And some tests do not pass
    - Some bugs can prevent coverage

- Coverage is achieved and all tests pass
  - Final rerun of the entire coverage achieving test suite on an uninstrumented implementation.
  - Stop testing if all tests pass on uninstrumented code

Method scope code coverage models

- Basic control flow
  - Statement
  - Branch
  - Multiple Condition
  - Path
  - Basis-Path Model
  - MCDC

Control flow graph - Concepts

- Condition
  - Each boolean operator in a predicate expression
- Predicate
  - Expression that contains a condition
  - Compound Predicate: a predicate with multiple conditions
- Segment
  - One or more lexically contiguous statements with no conditionally executed statements
  - Last statement must be the predicate of a conditional statement, a loop control, a *break*, a *goto* or method *exit()*
- Branch
  - Conditional transfer of control

Control flow graph - More concepts

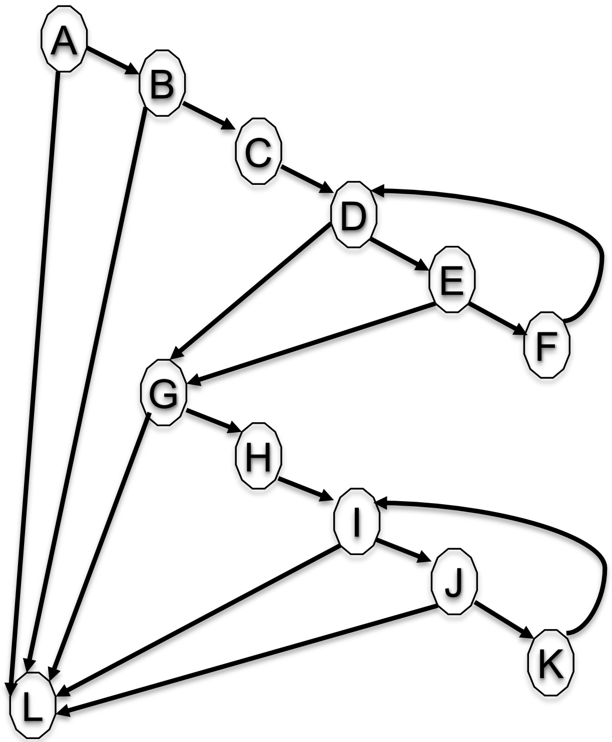
- Node: Represents a segment
- Branch: Represented as an outbound egde
- Entry-Node: Node with no inbound edges
- Exit-Node: Node with no outbound edges
- Path: composed of segments connected by arrows
- Entry-Exit Path: Path from an entry-node to an exit-node

- Representation of a compound predicate
  - Depends on the coverage model we want to apply

Example

1. Identify segments
2. Design CFG

```
class CircularBuffer {
   protected int lastMsg ;
   protected int msgCounter;
   private final int SIZE = 1000;
   // …..
   public int displayLastMsg(int nToPrint) {
      int np = 0;
      if ((msgCounter > 0)  && (nToPrint > 0)) {
         for (int j = lastMsg ; ( j != 0)  && (np < nToPrint) ; --j ) {
            System.out.println(messageBuffer[j]);
            np++;
         }
         if ( np > nToPrint) )
            for ( int j = SIZE ; ( j != 0 ) && (np < nToPrint) ) ; --j ) {
               System.out.println(messageBuffer[j]);
               np++;
            }
      }
      return np;
}
```

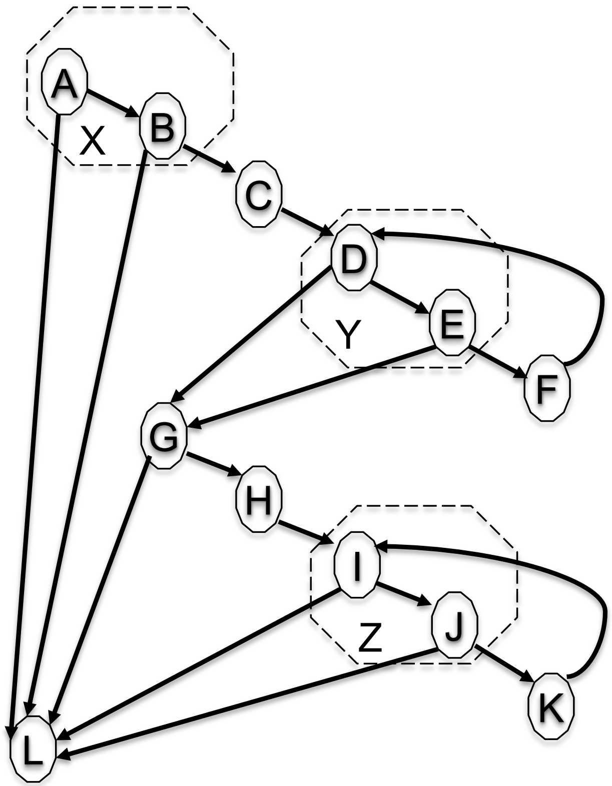


Impact of compound predicates

- Coverage models use a control flow graph
- Some coverage models do not distinguish simple predicates from compound predicates
  - E.g., Statement, segment and branch
  - Must use a single node to represent a compound predicate
  - Use X instead of A->B

    Y instead of D->E

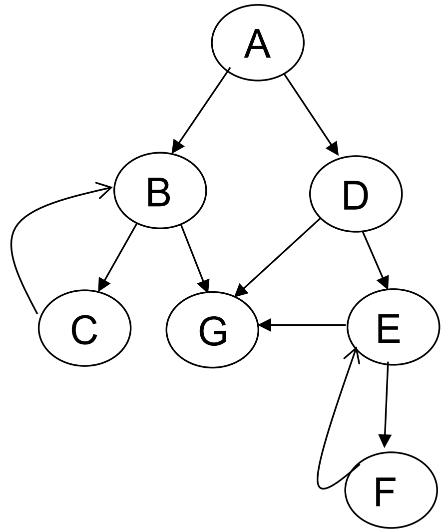
    Z instead of I->J



Example 2

```
public int mult(int a, int b) {
  int result = 0, int x = a;            A
  if ( x < 0 ) {
   while ( x != 0 ) {  B
     result -= b;
     x += 1;                 C
   }
  } else if ( x > 0 ) {     D
     while ( x != 0 ) {     E
      result += b;
      x -= 1;               F
   }
  }
  return result;    G
}
```

Control Flow Graph



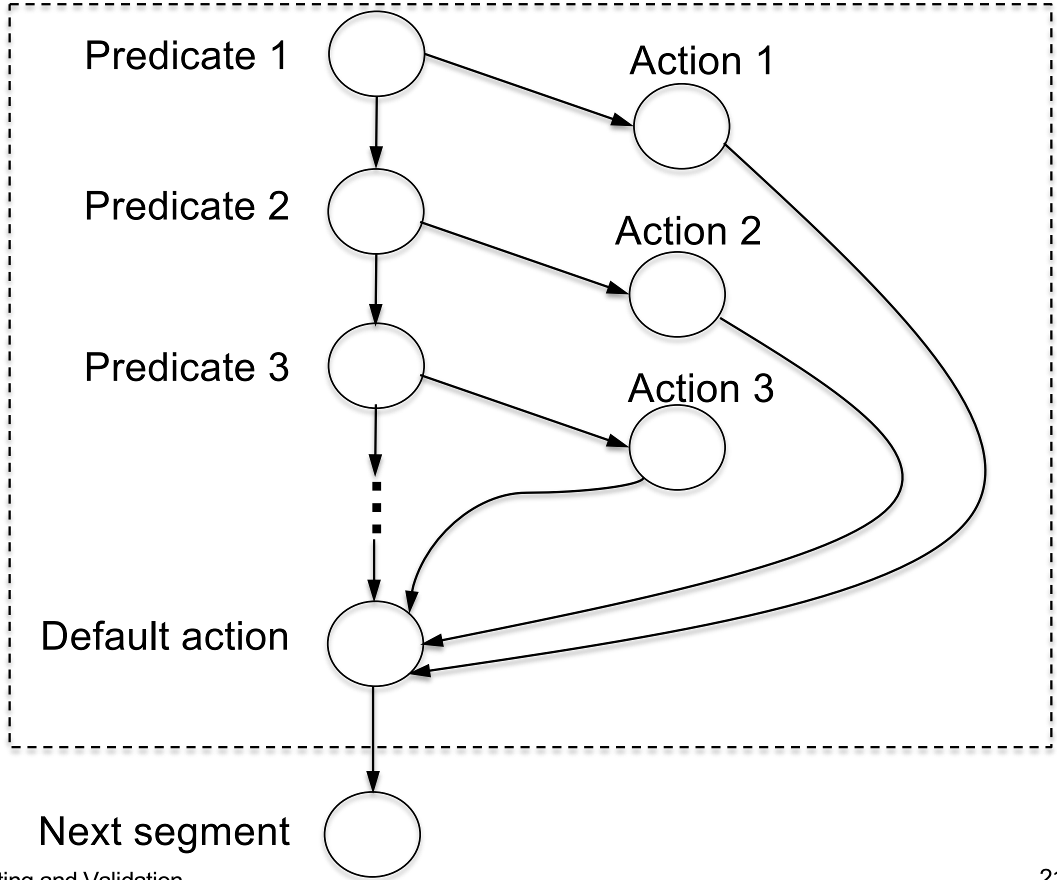
N-Way Branching

- N-Way branching statements:
  - Multiple if-else statement
  - switch statement
- How to represent N-Way branching?

N-Way branching – Multiple if-else statement

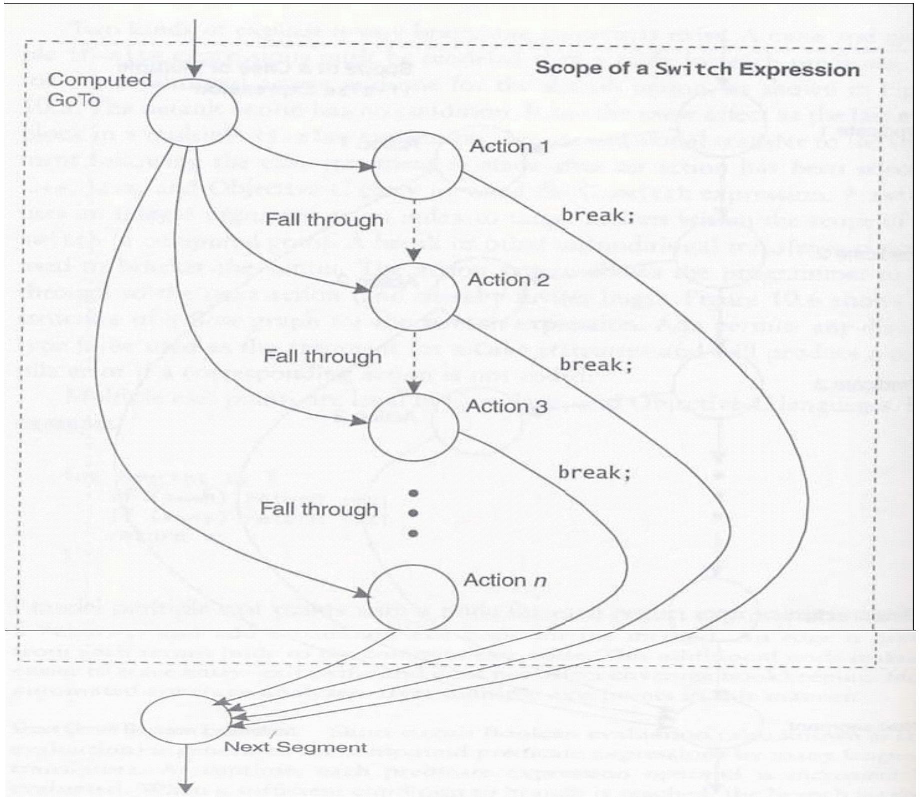
- How to represent multiple if-else statements?

```
if (Predicate 1)
   Action 1;
else if (Predicate 2)
   Action 2;
else if (Predicate 3)
   Action 3;
….
else
   Default action;

Next Segment;
```



N-Way branching – Switch statement

```
switch (expression) {
    case constant1:
        Action 1;
        break;
    case constant2:
        Action 2;
        break;
…
    case constantN-1:
        Action n -1;
        break;
     default:
        Action N;
        break;
}
Next Segment;
```



Statement coverage model

- Achieved when all statements in a method have been executed at least once
  - All nodes in CGF are exercised at least once
- Also known as C0 coverage, line coverage, basic block coverage
- $\text{Coverage} = \dfrac{\#\ \text{executed statements}}{\#\ \text{statements}}$

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

Segment coverage model

- Achieved when all segments in a method have been executed at least once
- No essential difference compared with statement coverage model
  - Difference in granularity, not in concept
  - 100% node coverage <-> 100% statement coverage
    - but levels will differ below 100%

Statement coverage blind spots

- Suppose the predicate in node B is incorrectly coded as *msgCounter => 0* instead of *msgCounter > 0*
  - Any test suite that did not force the message counter to be zero at least once would miss this error

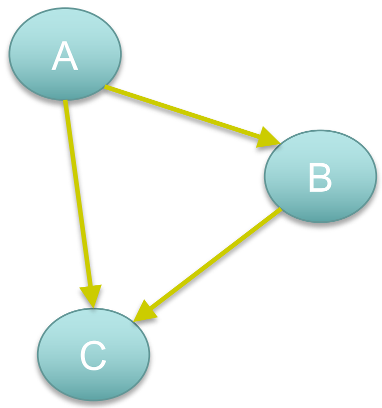
- When all statements in a loop can be reached with a single iteration

- Statement coverage can typically be achieved without exercising all true/false combinations of a simple predicate
  - May not require exercising all branches
  - Worse with compound predicate

Structures requiring branch coverage - Example

- Can hide bugs with a 100% statement coverage
- Statements:
  - Null else, do-while, switch without default, case without break
- Example:

```
void foo(int x, int y) {
    String str = null;
    if (x == y) {
      str = “Example”;
    }
    return str.length();
}
```



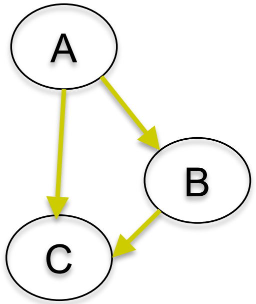
- How many paths to achieve statement coverage?
  - Statement coverage of **foo** can be achieved in a single test with x equal to y
  - Exercises ABC path
- Is **foo** bug-free?

NO!

Structures requiring branch coverage

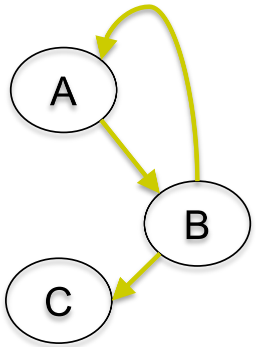
- Null else

```
if (condition) {
  doSomething()
}

nextSegment();
```



- Until loop (do-while)

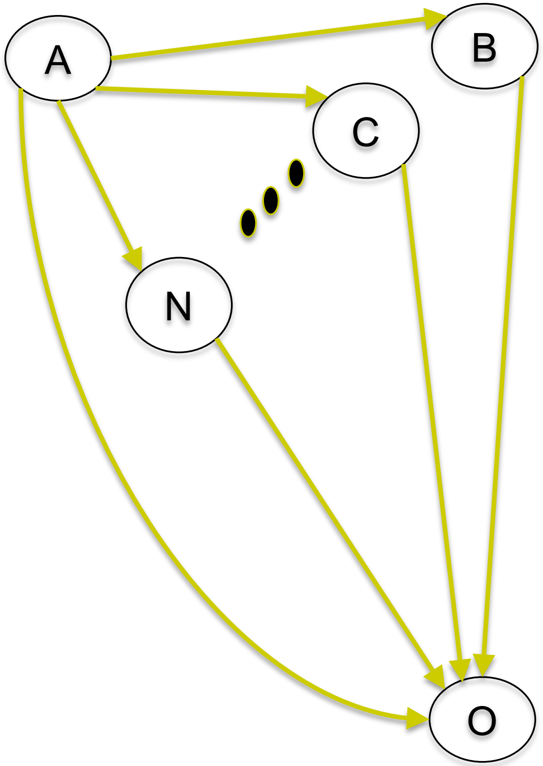
```
do {
  doSomething()
} while (condition)

nextSegment();
```



Structures requiring branch coverage - 2

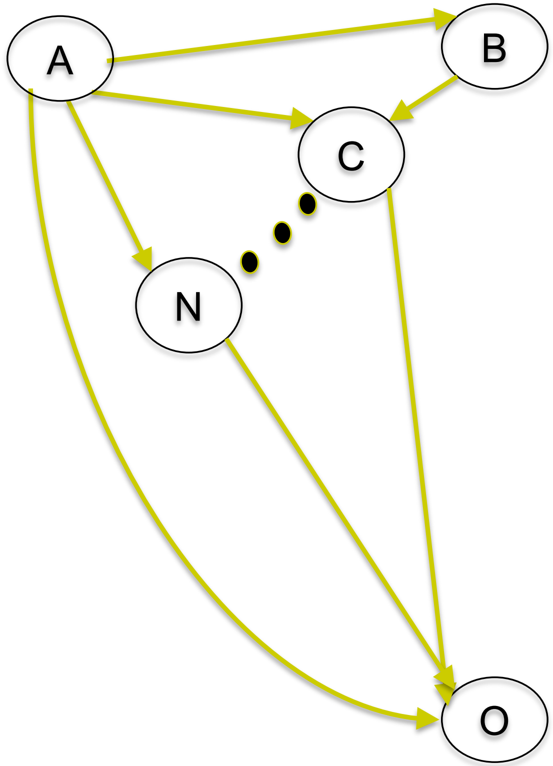
- Switch without default

```
switch (expression) {
  case constant1:
     doSomething1();
     break;

 case constant2:
     doSomething2();
     break;
 …
 case constantN:
     doSomethingN();
     break;
}
nextSegment();
```



- Switch without break

```
switch (expression) {
  case constant1:
     doSomething1();

 case constant2:
     doSomething2();
     break;
 …
 case constantN:
     doSomethingN();
     break;
}
nextSegment();
```



Branch coverage model

- Achieved when each branch in the code (edge in the CFG) is executed at least once
  - Improves on statement coverage by requiring that each branch is taken at least once
- Also known as decision coverage, all-edges coverage, or C1 coverage
- $\text{Coverage} = \dfrac{\text{\# executed braches}}{\text{\# branches}}$
- However, **treats** a compound predicate as a single predicate

Branch vs statement coverage

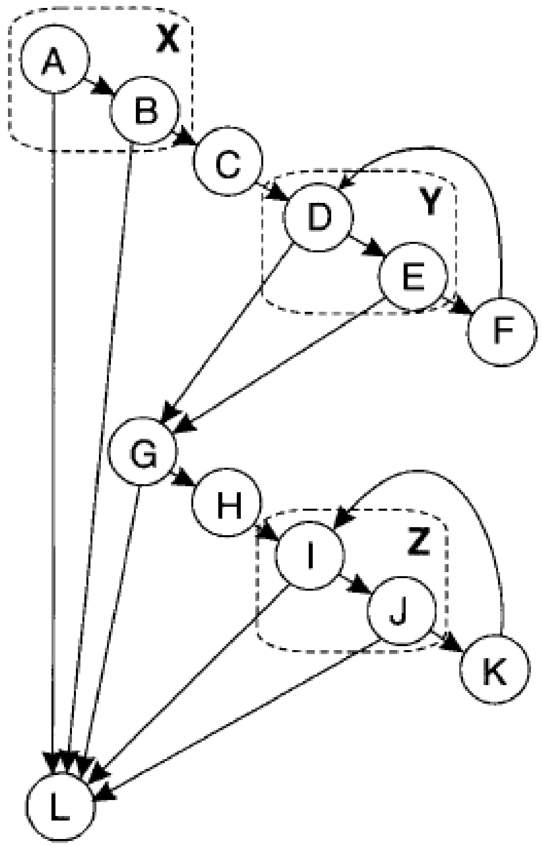
- Traversing all edges causes all nodes to be visited
  - 100% branch coverage implies 100% statement coverage
  - Branch coverage subsumes statement coverage
- The converse is not true
- Usually, statement coverage requires less test cases
- 100% statement coverage?

  XCYFYGHZKZL
- 100% branch coverage?

  XL

  XCYFYGL

  XCYFYGHZKZL



Short circuit boolean evaluation

- Is generated for compound expressions by many language translators
- Each predicate expression operand is incrementally evaluated
- Stops when sufficient condition to branch is reached

    - if ((x != 0) && (y/x > 0)) fred.foo(x);

Branch coverage blind spots

- Short circuit evaluation → some conditions not exercised
  - TC1 – (a = b, x, y)
  - TC2 – (a != b, x != y)
  - Branch coverage achieved
  - However, isEmpty() not executed

- Ignores the implicit paths that result from compound predicates
  - If n clauses, 2^n combinations, but only 2 are required
  - A compound predicate is treated as a single statement
    - (digit_high == 1 || !digit_low == -1)
    - Branch can be satisfied by varying only digit_high
    - The effect of digit_low is not testeed
    - Fault: missing operator may not be detected even with 100% branch

- Does not require that all entry-exit paths are exercised
  - How many entry-exit paths?
    - 4
  - Number of paths required to achieve branch coverage?
    - 2

```
int foo (int x) {
   if (a==b || (x==y && isEmpty()))
     ++x;
   else
     --x;
    return x;
}
```

```
int foo (int x) {
   if (a==b)
     ++x;
   if (z== y)
     --x;
   return x;
}
```

Multi-Condition Coverage

- Exercise each condition in a compound predicate or all true/false combinations of a compound predicate
- Three alternatives:
  - Condition Coverage
  - Branch/condition Coverage
  - Multiple condition Coverage

Condition coverage model

- This model considers only the conditions
  - Ignores predicates
  - Each condition is evaluated to true and false
- Achieved when all conditions are evaluated to true and false at least once
- $Condition\ Coverage\ = \dfrac{\#\ true\ conditions + \#\ false\ conditions}{2\ \#\ conditions}$

Condition coverage model – Blind spot

- Can achieve 100% condition coverage and a lower value for branch coverage
  - Does not subsume branch coverage model

```
public void example(int a, int b) {
 if ((a == 0) || (b > 0))
  do something;
 else
  do something else;

 System.out.println("end");
}
```

- With
  - TC1 (a = 0, b = -3)
  - TC2 (a = 10, b = 2)
- Condition coverage: 100%
- Branch coverage: 50%

- A condition may mask other conditions
  - ***if (a > 1 && b == 0)***
  - Do not distinguish from ***if (a > 1)***

TC1 (a = 2, b = 0) , TC2 (a = 0, b = 1)

100% condition and branch coverage

Branch/Condition coverage

- Condition coverage + each branch taken at least once
- It is computed by considering both branch and individual condition coverage measures

```
public void example(int a, int b) {
 if ((a == 0) || (b > 0))
  do something;
 else
  do something else;

 System.out.println("end");
}
```

- With
  - TC1 (a = 0, b = -3)
  - TC2 (a = 10, b = 2)
  - Condition coverage: 100%
  - **+ TC3 (a = -2, b = -5)**
    - Branch/Condition coverage: 100%
  - Still has mask of conditions

Multiple condition coverage

- Requires that all true-false combinations of simple conditions be exercised at least once

- Characteristics:
  - 2^n true-false combinations for a predicate with n simple conditions
    - Short-circuit evaluation reduces number of combinations
  - Usually requires **much more** test cases compared with statement or branch coverage
  - Relation with path coverage?

Predicates in real code

- Number of Boolean expressions with *n* conditions in avionic systems

Number of conditions									
1	2	3	4	5	6-10	11-15	16-20	21-35	36-76
16491	2262	685	391	131	219	35	36	4	2

```
public void example(int a, int b) {
 if ((a == 0) || (b > 0))
  do something;
}
```

- With TC1 (a = 0, b = 2), TC2 (a = 10, b = -1)
- Branch coverage: 100%
- But does not distinguish from
  - if (a== 0)
- Can apply multiple condition coverage
  - High number of test cases

Modified Condition/Decision coverage (MC/DC)

- Key idea: Decrease testing cost by ignoring the combinations of conditions that cannot be exercised
- Definition of MC/DC
  - Every condition in a decision in the program has taken all possible outcomes at least once (Decision coverage)
  - Every decision in the program has taken all possible outcomes at least once (Branch coverage)
  - Each 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
- Often required for safety-critical systems

MC/DC properties

- Subsumes condition/decision coverage
- 100% MC/DC guarantees each condition is not masked by the other conditions in the decision
  - if (A && B)
    - 100% for Branch and C/D coverage with TT and FF
    - Does not catch if (A) / if (B) / if (A || B)
- Usually, a decision with N conditions requires N + 1 test cases to achieve MC/DC
- Guarantees to detect
  - Missing condition
  - Wrong operator
    - && instead of ||, > instead of <, …

Compute minimal test suite for MC/DC

- For each condition c in the decision, determine two test cases where outcome of decision is different and only condition c has changed
- Examples

A and B

Outcome	1	0
	AB	AB
Focus A	11	01
Focus B	11	10

Test cases: {(11), (01), (10)}

A or B

Outcome	1	0
	AB	AB
Focus A	10	00
Focus B	01	00

Test cases: {(00), (01), (10)}

Another example

(A and B) or C

Outcome	1	0
	ABC	ABC
Focus A	1 1 0	0 1 0
Focus B	1 1 0	1 0 0
Focus C	1 0 1	1 0 0

Test cases: {(110), (101), (100), (010)}

Path coverage

- Focus on entry-exit paths of method
- Achieved when all entry-exit paths of a method have been exercised at least once
- Coverage = $\dfrac{\#\ \text{executed paths}}{\#\ \text{paths}}$
- Characteristics:
  - Loops?
    - May require an infinite number of paths
    - Consider a limited number of looping possibilities
      - Most used: zero iterations, one or more iterations.
  - Advantage: Very thorough testing
  - However
    - The number of paths is exponential with the number of branches
    - Many paths are impossible to exercise due to relationships of data

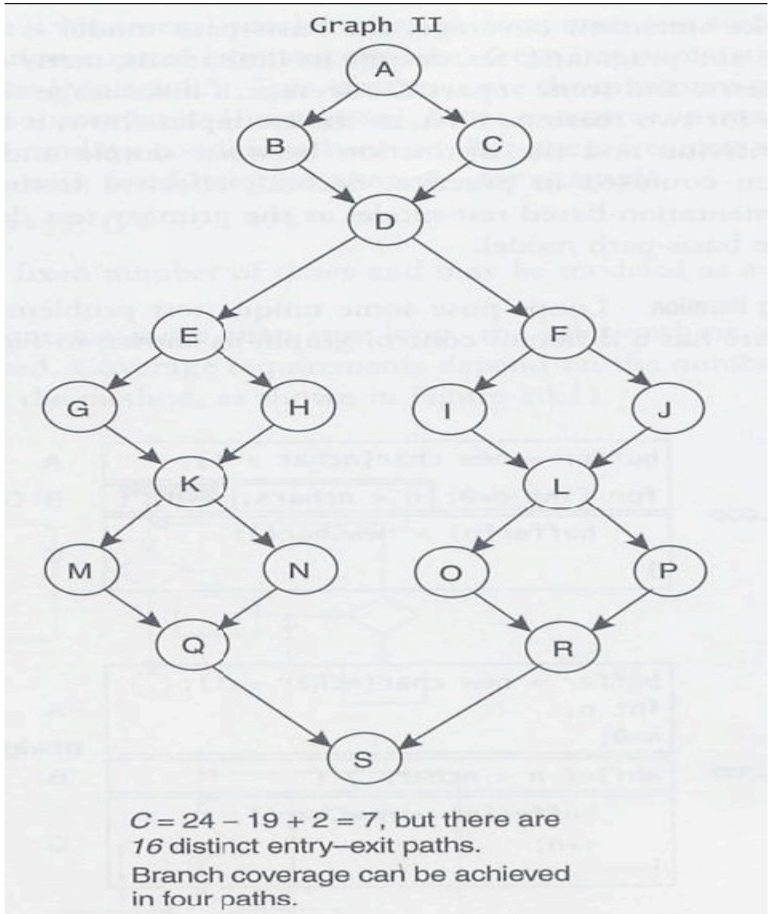
Basis-Path Model Coverage

- Also known as structured testing
- Focus on entry-exit paths
- Achieved when **C independent** entry-exit paths have been exercised
  - $C = e - n + 2$

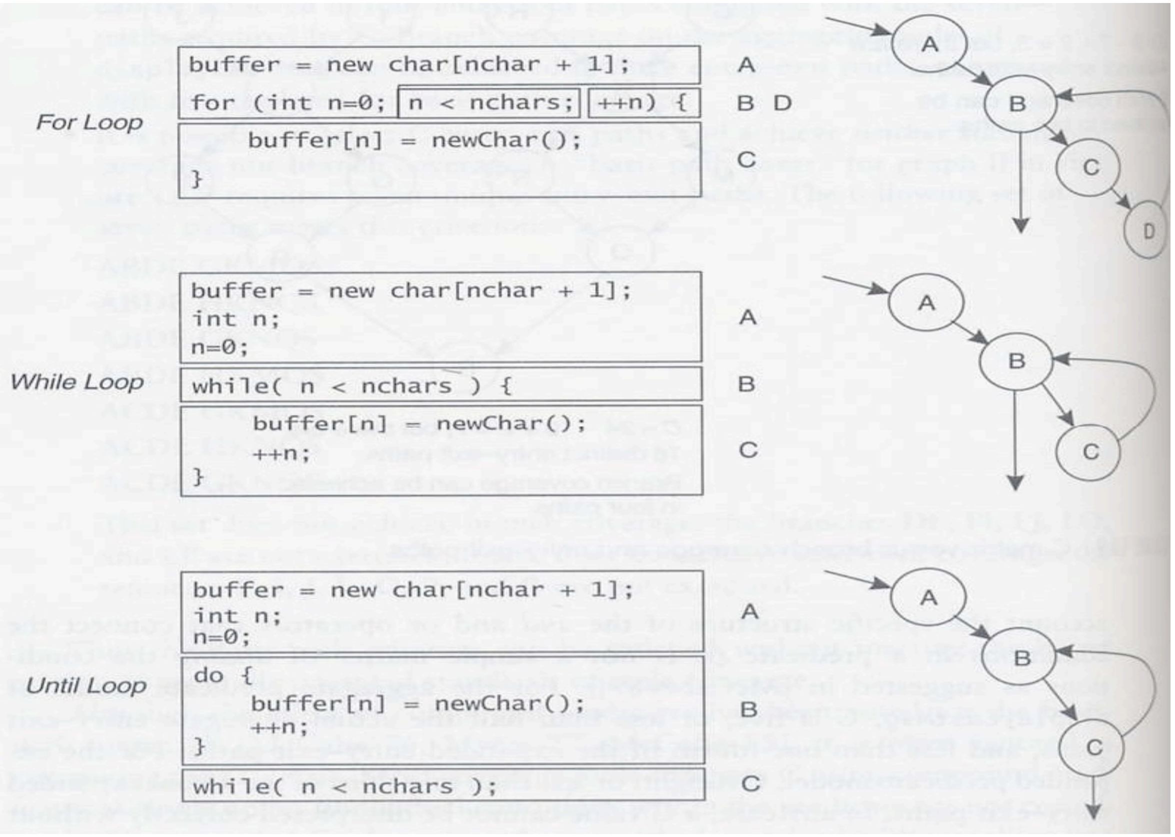
    e is number of edges, n is number of nodes
  - An independent path is any path that introduces at least one new edge
- C is the cyclomatic complexity metric.
- Available studies of OO code do not show significant correlation between defects and C

Basis-Path coverage example

- Branch coverage may be achieved with less C paths, in some methods
- It is possible to select C entry-exit paths and achieve neither statement nor branch coverage.
- But this set of paths is not a set of **independent** paths!
- Basis-path subsumes branch and statement
- A compromise between path and branch criteria



Canonical loop structures



- Each canonical loop has a different control graph

Simple loop coverage

- Fault model
  - Bug in control loop variables
- Exercise body of loop:
  - `0`
  - `1`
  - `2`
  - `typical`
  - `max` and `max+1` times
- Minimum test suite is 0, 1 and max
- Full test suite: 0,1, 2, typical, max, max +1

Loop coverage

- **Idea: For each loop, test 0, 1, or more than 1 consecutive iterations**
- Let n_k be the number of loops with exactly k consecutively executed iterations
- $cov_{loop} = \dfrac{n_0 + n_1 + n_{>1}}{\text{Total nb. of loops} * 3}$
- Typically combined with other criteria, such as statement or branch coverage

The Dark Side of code coverage

- No coverage model will find this bug.
- This bug could be found by a single test case that included revealing values of deposit and price

```
public void computeChange(int price) {
   int n_100, n_25, n_10, n_5;
   if (_deposit <= price)
     change_due = 0;
   else {
      change_due = _deposit – price;
      n_100 = change_due/100;
      change_due -= n_100 * 100;
      n_25 = change_due/25;
      change_due -= n_25*25;
      n_10 = change_due/10;
      change_due -= n_10*10;
      n_5 = change_due/10;
   }
   // ....
}
```

Infeasible Path

- Path that cannot be executed
- Possible causes:
  - Contradictory or mutually exclusive conditions:

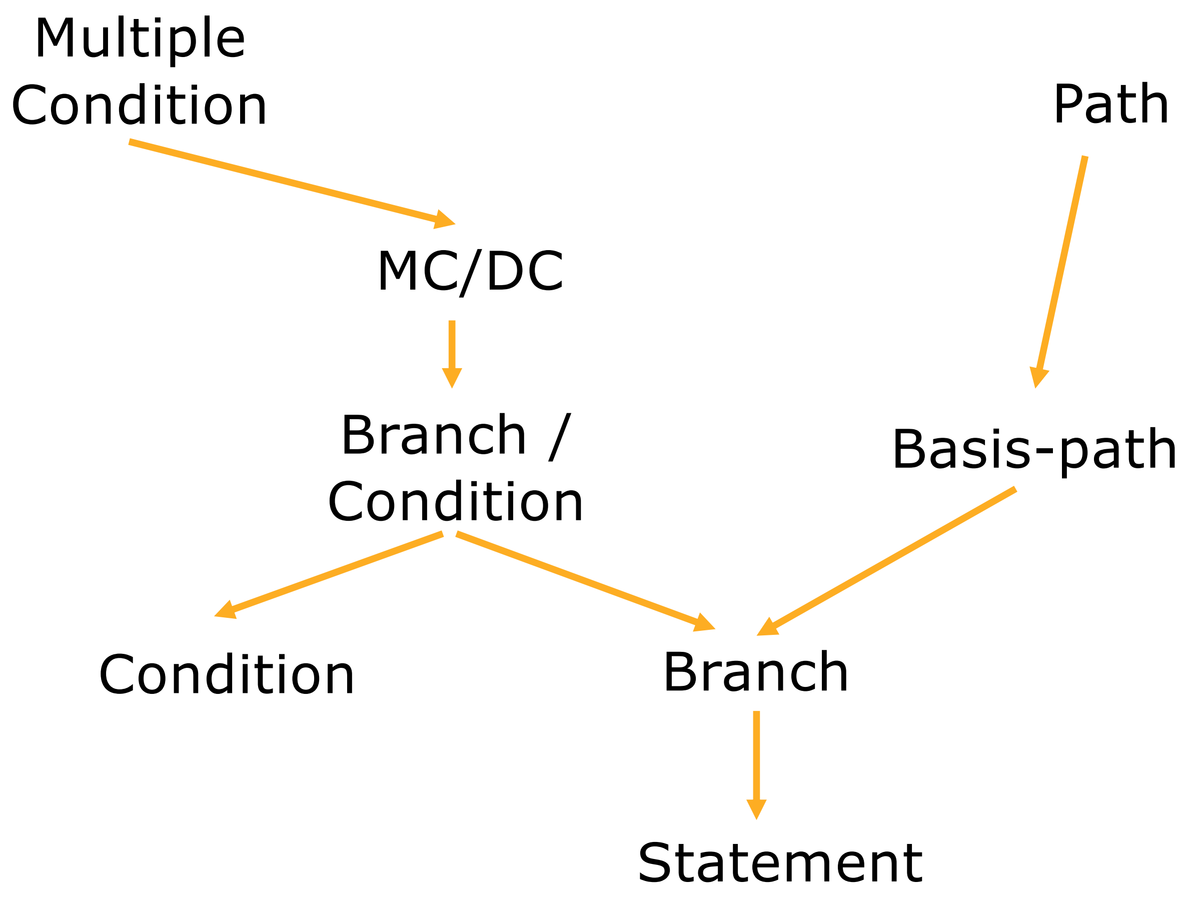
    if ((x > 2) || (x < 10)) { … }
    - the false-false branch of this predicate can never be taken
  - Mutually exclusive, redundant predicates:

    if (x == 0) oof.perpetual(); else oof.free();

    … // code follows that does not change the value of x

    If (x != 0) oof.motion(); else oof.lunch()l
    - Paths perpetual() … motion() and free() … lunch() are impossible
  - Dead code or code detours.
  - Exception handling code
  - “This should never happen”

Subsume Relationship



Code Coverage FAQ

- Is 100% coverage the same as exhaustive testing?
- Is branch coverage the same as path coverage?
- Is statement coverage the same as path coverage?
- Can path coverage be achieved?
- Is every path in a flow graph testable?

Code Coverage FAQ (cont)

- Is less than 100 percent coverage acceptable?
- Can I have high confidence in a test suite if I don't measure coverage?
- Does achieving 100% coverage for x and passing all tests mean that I have bug-free code?
- *Conclusion:*
  - *Code Coverage is a useful tool for finding code that a responsibility-based test suite hasn't touched, but achieving any coverage goal is never a guarantee of the absence of bugs*