# SENG3320/6320:
# Software Verification and Validation

School of Electrical Engineering and Computing

Semester 1, 2020

# Basis Path Coverage

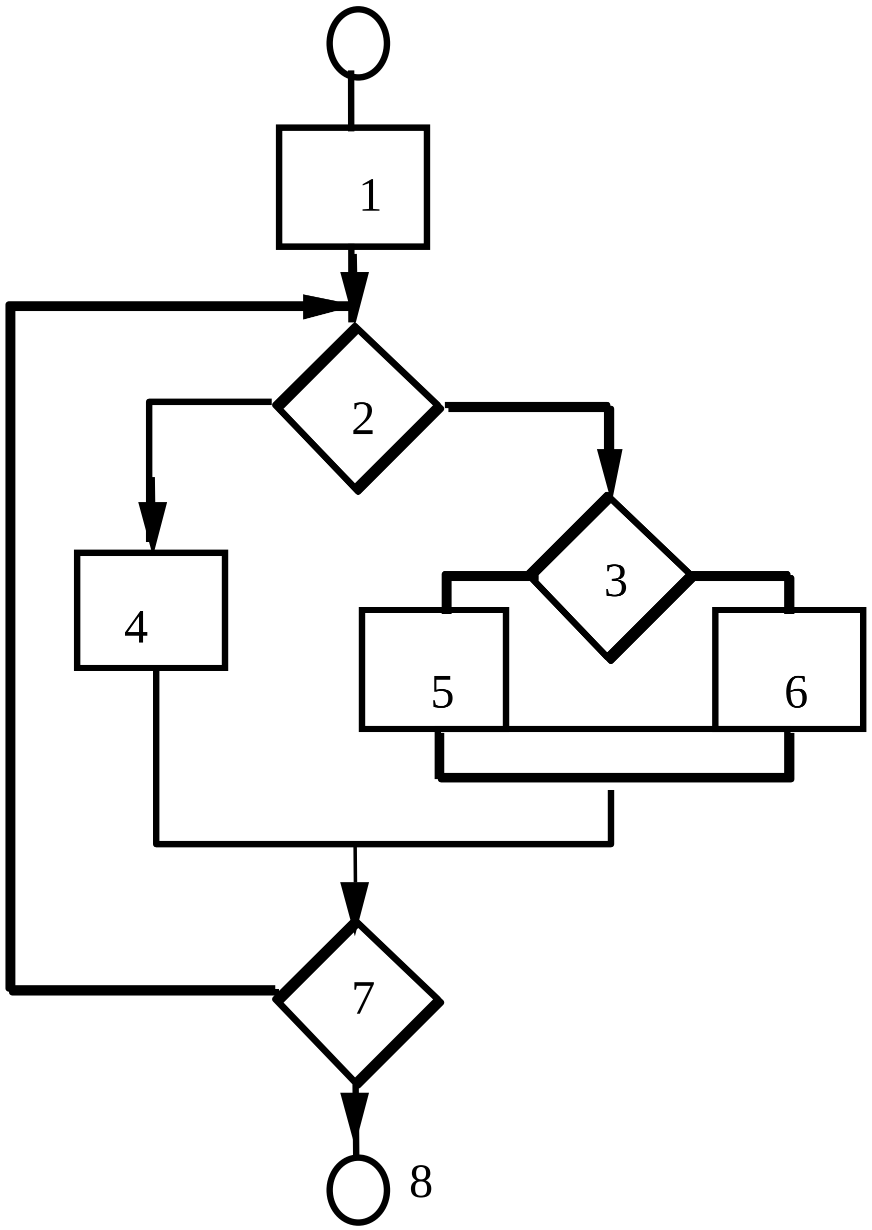
# Basis Path Testing

- A testing mechanism proposed by McCabe.
- A testing criterion between branch coverage and all path coverage.
- Basis Path testing is a software testing that fulfills the requirements of branch testing & also tests all of the **independent paths** that could be used to construct **any arbitrary path** through the computer program. [NIST]

Arthur H. Watson and Thomas J. McCabe (1996). "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric". NIST Special Publication 500-235.

Thomas J. McCabe: A Complexity Measure. IEEE Trans. Software Eng. 2(4): 308-320 (1976)

# Independent Path




A path through the system is independent from other paths only if it includes some vertices or edges that are not covered in the other path.

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8

# Cyclomatic Complexity

- A program's complexity can be measured by the cyclomatic number of the program flowgraph
- For a program with the program flowgraph *G*, the cyclomatic complexity *v(G)* is measured as:

$$\boldsymbol{v(G) = e - n + 2p}$$

  - *e* : number of edges
    - Representing branches and cycles
  - *n* : number of nodes
    - Representing block of sequential code
  - *p* : number of connected components
    - For a single component, *p=1*

# Simplified Complexity Calculation

- For a program with the program flowgraph $G$, the cyclomatic complexity $v(G)$ is measured as:

  $$v(G) = 1 + d$$

  - $d$ : number of predicate nodes (i.e., nodes with out-degree other than 1)
    - $d$ represents number of loops in the graph

      or number of decision points in the program

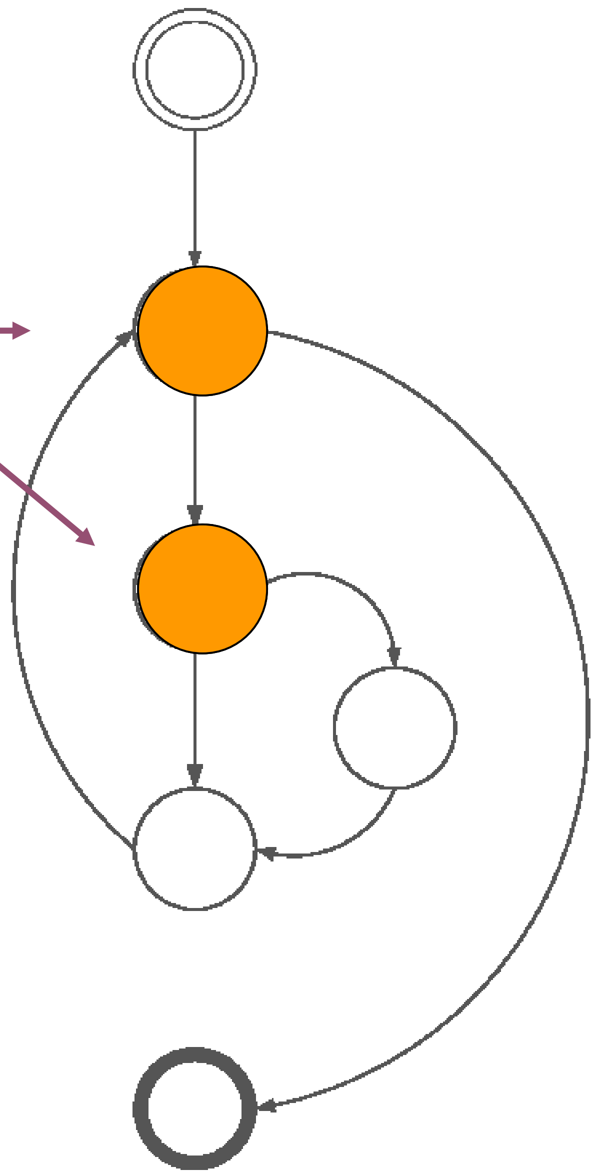
# Example 1

$$v(G) = e - n + 2p$$
$$= 7 - 6 + 2 \times 1$$
$$= 3$$

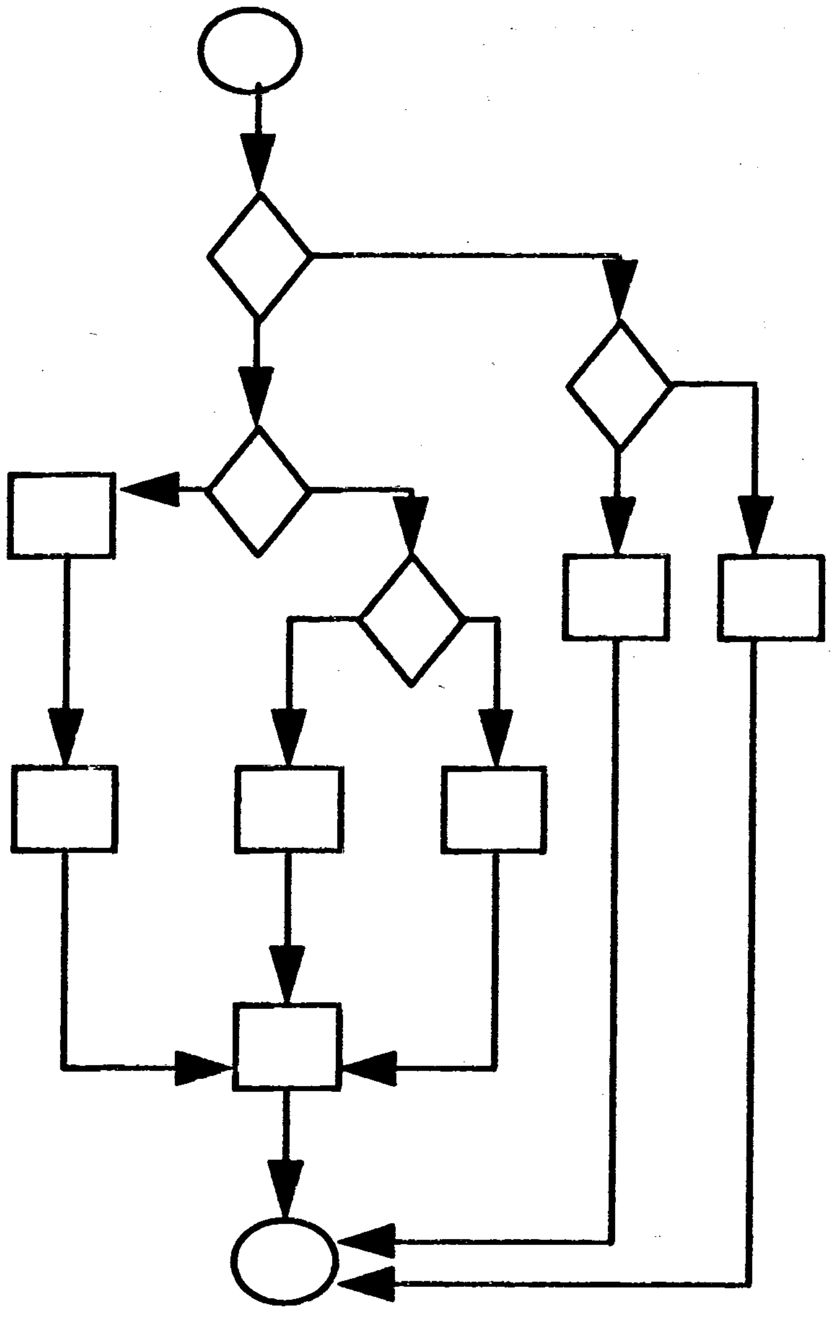
**Or**

$$v(G) = 1 + d$$
$$= 1 + 2 = 3$$

Predicate nodes (decision points)



# Example 2



***v(G) = 16 – 13 + 2 = 5***

***or***

***v(G) = 4 +1 = 5***

# Cyclomatic Complexity: Usage

- **A useful indicator of software quality**
  - McCabe (1976) suggested that a module may be problematic if *v(G)* exceeds 10.
  - Grady (1994) concluded that 15 should be maximum value of *v(G)*
  - Bennett (1994) suggested that a modules be rejected if its *v(G)* exceeds 20.
- **Useful in white-box testing**
  - It provides an upper bound on the number of test cases that will be required to guarantee coverage of all program statements.

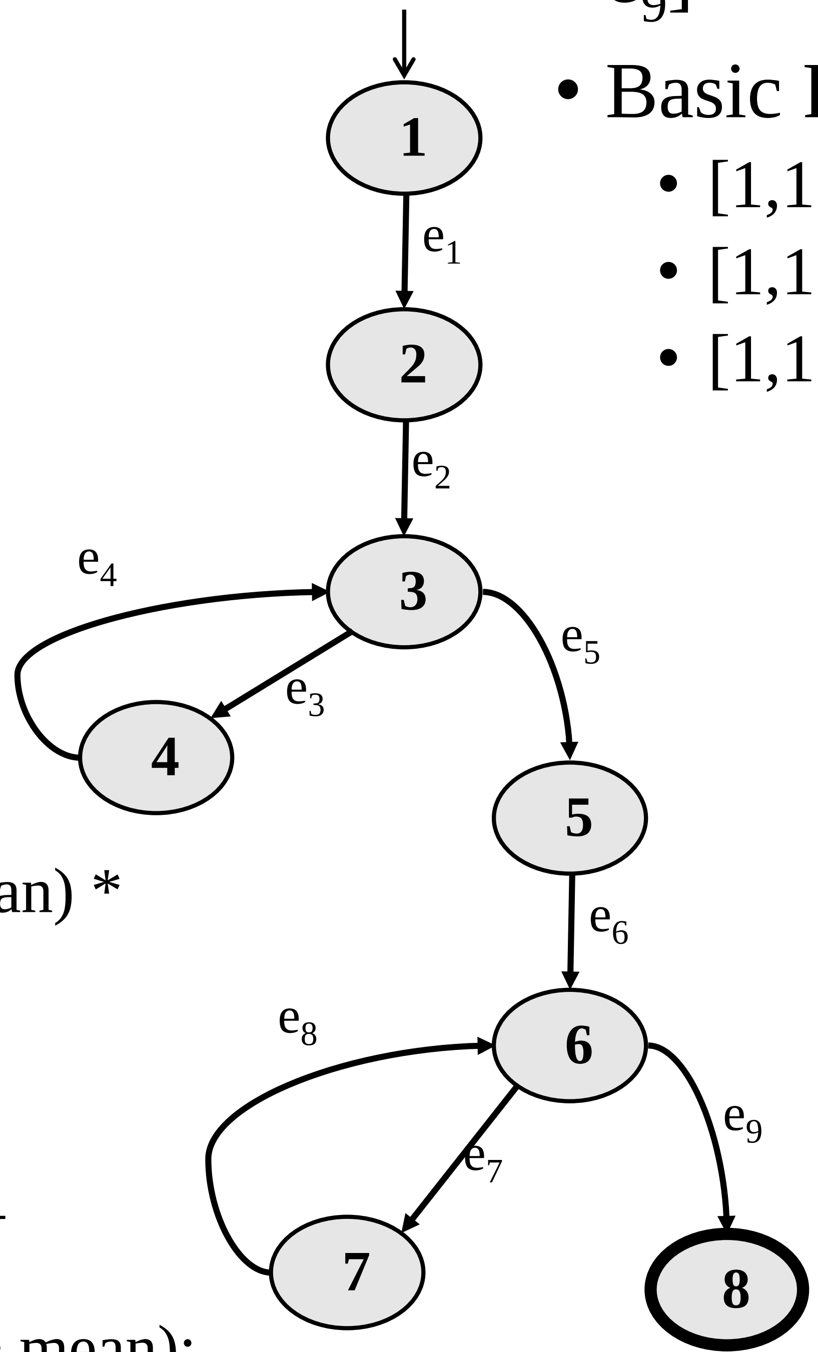
# **McCabe’s** Basis Path Testing

1. Generate control flow graph
2. Compute cyclomatic complexity
3. Determine a basis set of linearly independent paths
4. Generate tests for the basis paths

*All Path Testing>=Basis Path Testing>=Branch Testing*

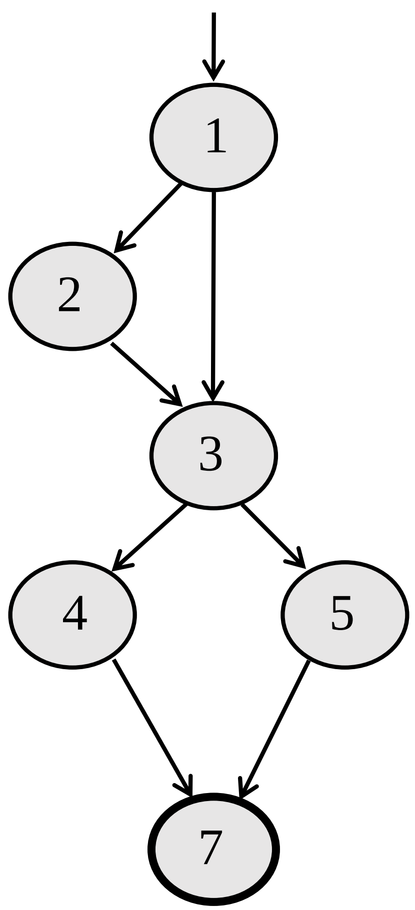
# Basis Path Testing

```
public static void CSta (int [ ] numbers)
{
    int length = numbers.length;
    double var, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    mean = sum / (double) length;

    varsum = 0.0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [i] - mean) * 
(numbers [i] - mean));
    }
    var = varsum / ( length - 1.0 );

    System.out.println ("length:                    " + 
length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("variance:                  " + var);
}
```

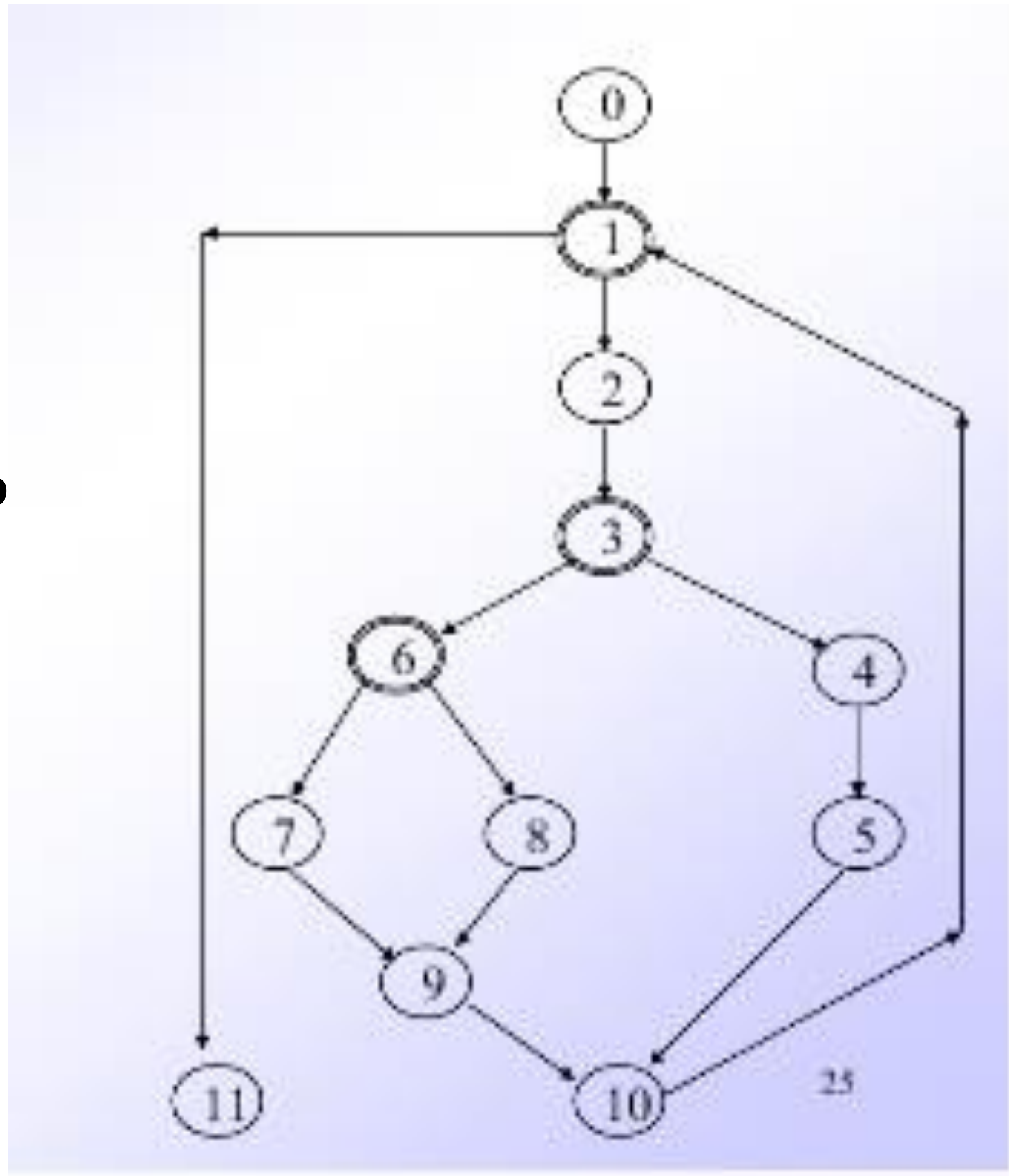



- V(g) = 9-8+2=3
- [$e_1$, $e_2$, $e_3$, $e_4$, $e_5$, $e_6$, $e_7$, $e_8$, $e_9$]
- Basic Path Set
  - [1,1,0,0,1,1,0,0,1]
  - [1,1,1,1,1,1,0,0,1]
  - [1,1,0,0,1,1,1,1,1]

# Quiz




Basis Path Set ?



# Quiz

```
#include <stdio.h>
main()
{
int a ;
scanf (“%d”, &a);
if ( a >= 10 )
 if ( a < 20 )
    printf ("10 < a< 20 %d\n" , a);
 else
    printf ("a >= 20    %d\n" , a);
else
  printf ("a <= 10    %d\n" , a);
}
```

Basis Path Set ?

# Decision and Condition Coverage

# Logic in Program

((x>5) && (y>0))

Decision

Condition

# Decision Coverage

- Decision Coverage (DC):
  Executing *true* and *false* of decision.
- Also called Branch Decision Coverage

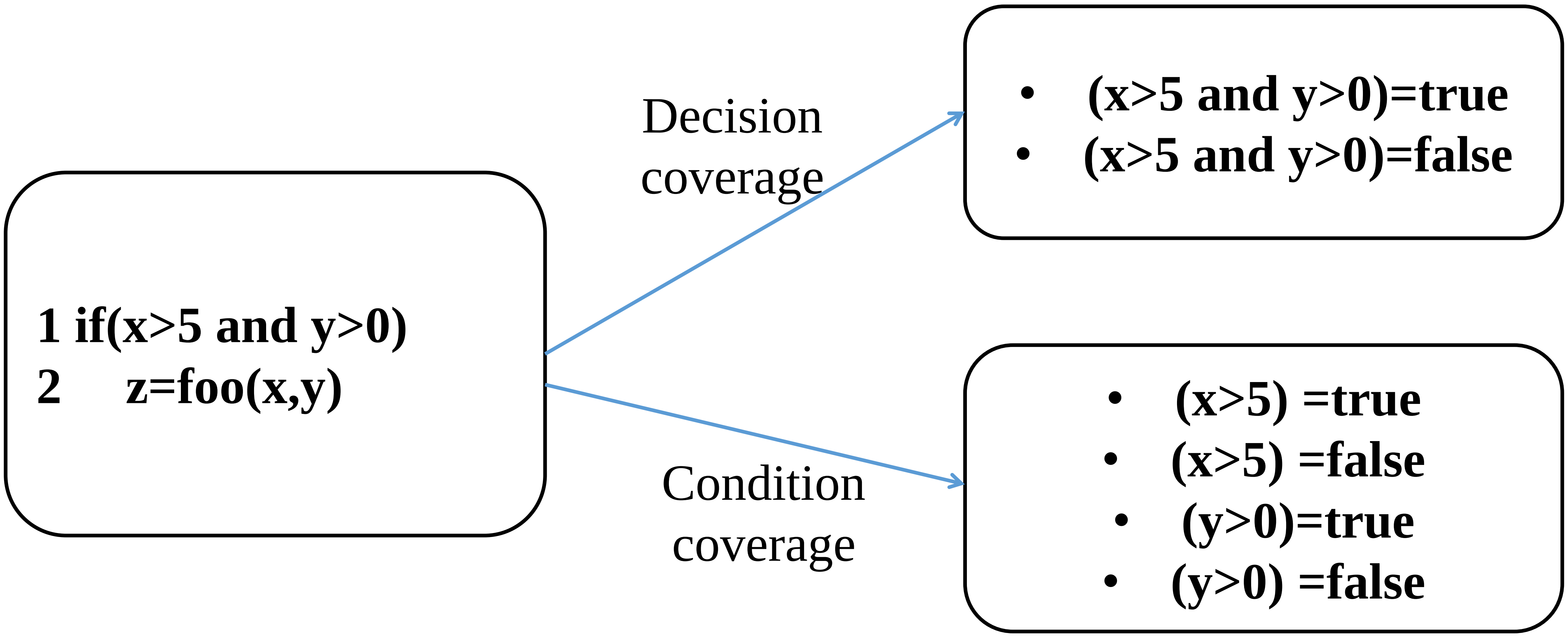
Example: ((x>5) && (y>0)): *true* and *false*

```
// {(6, 1), (1,1)} --- 100% Decision Coverage
int foo(int x, int y) {
        int z = y;
        if ((x>5) && (y>0)) {
                z = x;
        }
        return x*z;
}
```

# Condition Coverage

- Condition Coverage (CC):
  Executing *true* and *false* of each condition
- (x>5)： *true* and *false*
- (y>0)： *true* and *false*

```
// {(6, 0), (0,1)} --- 100% Condition Coverage
int foo(int x, int y) {
        int z = y;
        if ((x>5) && (y>0)) {
                z = x;
        }
        return x*z;
}
```


1 if(x>5 and y>0)
2 z=foo(x,y)
Decision coverage
• (x>5 and y>0)=true
• (x>5 and y>0)=false
Condition coverage
• (x>5) =true
• (x>5) =false
• (y>0)=true
• (y>0) =false

# An Example

```
begin
    int x,y,z;
    input(x,y);
    if(x<0 and y<0)
      z=foo1(x,y)
    else
      z=foo2(x,y);
    output(z);
end
```

**$T=\{t_1:<x=-3,y=-2>,\ \ t_2:<x=-4,y=2>\}$**

- **Decision coverage?**
- **Condition coverage?**

- **100% Condition coverage does not mean 100% Decision coverage**

```
begin
    int x,y,z;
    input(x,y);
    if(x<0 and y<0)
      z=foo1(x,y)
    else
      z=foo2(x,y);
    output(z);
end
```

**$T=\{t_1:<x=-3,y=2>,\ \ t_2:<x=4,y=-2>\}$**

- **Decision coverage?**
- **Condition coverage = 100%**

- **100% Decision coverage does not mean 100% Condition coverage**

```
begin
   int x,y,z;
   input(x,y);
   if(x<0 and y<0)
     z=foo1(x,y)
   else
      z=foo2(x,y);
   output(z);
end
```

$T=\{t_1:<x=-3,y=-2>,t_2:<x=-4,y=2>\}$

- **Decision coverage = 100%**
- **Condition coverage?**

# Condition/Decision Coverage

- Condition/Decision Coverage (C/DC): Combing DC and CC.
- Overcomes the limitation of Decision Coverage (DC) and Condition Coverage (CC)
  - C/DC $\geq$ CC
  - C/DC $\geq$ DC

```
begin
    int x,y,z;
    input(x,y);
    if(x<0 and y<0)
      z=foo1(x,y)
    else
       z=foo2(x,y);
    output(z);
end
```

$T=\{t_1:<x=-3,y=-2>,t_2:<x=4,y=2>\}$

- **Decision coverage = 100%**
- **Condition coverage = 100%**

# Multiple Condition Coverage

- Multiple condition coverage (MCC) reports whether every possible combination of Boolean sub-expressions occurs.
- The test cases required for full multiple condition coverage of a condition are essentially given by the logical operator truth table for the condition.

# D=(A<B) or (A>C)

| | A<B | A>C | D |
|---|---|---|---|
| 1 | true | true | true |
| 2 | true | false | true |
| 3 | false | true | true |
| 4 | false | false | false |

**T={t1:<A=2,B=3,C=1>, t2:<A=2,B=1,C=3>}**

- **Covers all decisions and conditions**
- **Covers only two combinations of Boolean sub-expressions.**

# An Example

- Requirements for a program:

*Given three integers A, B, and C, produce S according to the following table:*

| | A<B | A>C | S |
|---|---|---|---|
| **1** | **true** | **true** | **f1(A,B,C)** |
| **2** | **true** | **false** | **f2(A,B,C)** |
| **3** | **false** | **true** | **f3(A,B,C)** |
| **4** | **false** | **false** | **f4(A,B,C)** |

```
1 begin
2     int A,B,C,S=0;
3     input(A,B,C);
4     if(A<B and A>C)
      S=f1(A,B,C);
5     if(A<B and A≤C)
      S=f2(A,B,C);
6     if(A≥B and A≤C)
      S=f4(A,B,C);
7     output(S);
9 end
```

| | A<B | A>C | S |
|---|---|---|---|
| 1 | true | true | f1(A,B,C) |
| 2 | true | false | f2(A,B,C) |
| 3 | false | true | f3(A,B,C) |
| 4 | false | false | f4(A,B,C) |

A buggy implementation！

```
1 begin
2     int A,B,C,S=0;
3     input(A,B,C);
4     if(A<B and A>C)
      S=f1(A,B,C);
5     if(A<B and A≤C)
      S=f2(A,B,C);
6     if(A≥B and A≤C)
      S=f4(A,B,C);
7     output(S);
9 end
```

| | A<B | A>C | S |
|---|---|---|---|
| 1 | true | true | f1(A,B,C) |
| 2 | true | false | f2(A,B,C) |
| 3 | false | true | f3(A,B,C) |
| 4 | false | false | f4(A,B,C) |

$T=\{t_1:<A=2,B=3,C=1>, t_2:<A=2,B=1,C=3>\}$

- 100% condition coverage
- Bug not found

```
begin
    int A,B,C,S=0;
    input(A,B,C);
    if(A<B and A>C) S=f1(A,B,C);
    if(A<B and A≤C) S=f2(A,B,C);
    if(A≥B and A≤C) S=f4(A,B,C);
    output(S);
end
```

| | A<B | A>C | S |
|---|---|---|---|
| 1 | true | true | f1(A,B,C) |
| 2 | true | false | f2(A,B,C) |
| 3 | false | true | f3(A,B,C) |
| 4 | false | false | f4(A,B,C) |

**T={$t_1$:<A=2,B=3,C=1>,**
**$t_2$:<A=2,B=1,C=3>,**
**$t_3$:<A=2,B=3,C=5>}**

- **100% condition coverage**
- **100% decision coverage**
- **Bug not found**

```
begin
    int A,B,C,S=0;
    input(A,B,C);
    if(A<B and A>C) S=f1(A,B,C);
    if(A<B and A≤C) S=f2(A,B,C);
    if(A≥B and A≤C) S=f4(A,B,C);
    output(S);
end
```

**T={$t_1$:<A=2,B=3,C=1>,**
**$t_2$:<A=2,B=1,C=3>,**
**$t_3$:<A=2,B=3,C=5>,**
**$t_4$:<A=2,B=1,C=1>}**

- 100% Multiple Condition Coverage
- **Bug found**

| | A<B | A>C | T | A<B | A≤C | T | A≥B | A≤C | T |
|---|---|---|---|---|---|---|---|---|---|
| 1 | true | true | t1 | true | true | t3 | true | true | t2 |
| 2 | true | false | t3 | true | false | t1 | true | false | t4 |
| 3 | false | true | t4 | false | true | t2 | false | true | t3 |
| 4 | false | false | t2 | false | false | t4 | false | false | t1 |

# Quiz – C/DC

How to achieve 100% condition/decision coverage?

```
// ??
int foo(int x, int y) {
        int z = y;
        if ((x>5) && (y>0)) {
                z = x; }
        return x*z;
}
```

# Quiz- MCC

How to achieve 100% multiple condition coverage?

```
// ??
int foo(int x, int y) {
         int z = y;
        if ((x>5) && (y>0)) {
                 z = x; }
        return x*z;
}
```

# Limitation of MCC

- Assuming n conditions, $2^n$ test cases are required.
- Assuming each test case needs 1ms to execute:

| Conditions<br>n | Test cases<br>$2^n$ | Test case execution time |
|---|---|---|
| 1 | 2 | 2ms |
| 4 | 16 | 16ms |
| 8 | 256 | 256ms |
| 16 | 65536 | 65.5s |
| 32 | 4294967296 | 49.5 days |

# Modified condition/decision (MC/DC)

- Motivation: Effectively test important combinations of conditions, without exponential blowup in test suite size
  - "Important" combinations means: Each basic condition independently affects the outcome of each decision
  - MC/DC $\geq$ C/DC
- Requires:
  - For each basic condition C, two test cases,
  - values of all evaluated conditions except C are the same
  - compound condition as a whole evaluates to true for one and false for the other

# Modified condition/decision (MC/DC)

- **MC/DC coverage:**
  - Each entry and exit point is invoked
  - Each decision takes every possible outcome
  - Each condition in a decision takes every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision.
  - Independence of a condition is shown by proving that only one condition changes at a time.
- MC/DC is used in avionics software development guidance DO-178B and DO-178C to ensure adequate testing of the most critical (Level A) software.

https://en.wikipedia.org/wiki/Modified_condition/decision_coverage

# MC/DC

- ((x>5) && (y>0)) Decision

| | | |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

```
// ??
int foo(int x, int y) {
        int z = y;
        if ((x>5) && (y>0)) {
                z = x; }
        return x*z;
}
```

# Comparison on the number of test cases

| | |
|---|---|
| • DC | 2 |
| • CC | 2**n* |
| • C/DC | 2*n+1 |
| • MC/DC | 2*n+1 |
| • MCC | $2^n$ |

- To test `if (A xor B)`

| | | | |
|---|---|---|---|
| A: | T | T | F |
| B: | T | F | T |

Table 1. Types of Structural Coverage

| Coverage Criteria | Statement Coverage | Decision Coverage | Condition Coverage | Condition/ Decision Coverage | MC/DC | Multiple Condition Coverage |
|---|---|---|---|---|---|---|
| Every point of entry and exit in the program has been invoked at least once | | • | • | • | • | • |
| Every statement in the program has been invoked at least once | • | | | | | |
| Every decision in the program has taken all possible outcomes at least once | | • | | • | • | • |
| Every condition in a decision in the program has taken all possible outcomes at least once | | | • | • | • | • |
| Every condition in a decision has been shown to independently affect that decision's outcome | | | | | • | •[8] |
| Every combination of condition outcomes within a decision has been invoked at least once | | | | | | • |

# References


- J.J. Chilenski and S.P. Miller, "**Applicability of Modified Condition/Decision Coverage to Software Testing**," *Software Eng. J., vol. 9, no. 5, pp. 193-200, 1994.*
- D. Richard Kuhn. "**Fault classes and error detection capability of specification-based testing**" *ACM Transactions on Software Engineering and Methodology, 8(4):411--424, October 1999.*
- Jones, J. and Harrold, M. "**Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage**", *Proceedings of the IEEE International Conference on Software Maintenance ( ICSM'01), Florence, Italy, 7-9 November 2001, pp. 92--101.*
- Dupuy, A. and Leveson, N. "**An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software**", *Proceedings of the Digital Aviation Systems Conference (DASC), Philadelphia, USA, October 2000.*
- Lecture Materials from Dan Hao, Zhenyu Chen, Mauro Pezzè & Michal Young.
- Pressman, R. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2005.
- Hayhurst, Kelly; Veerhusen, Dan; Chilenski, John; Rierson, Leanna, "A Practical Tutorial on Modified Condition/ Decision Coverage". NASA, May 2001.

Thanks!


Testing
Software
development
test
quality
articles
management
design
risk
good
code
People
Project
information
web
use
Agile
usability
system
article
check
time
team
bug
testers
Ruby
Tools
life