Software Testing

Rajib Mall
CSE Department
IIT KHARAGPUR

MC/DC Testing

Modified Condition/Decision Coverage (MC/DC)

- **Motivation:** Effectively test important combinations of conditions, without exponential blowup to test suite size:
  - **"Important" combinations means:** Each basic condition should independently affect the outcome of each decision
- **Requires:**

  If((A==0) $\vee$ (B>5) $\wedge$ (C<100)) ….

  - **For each basic condition c, Compound condition as a whole evaluates to true or false as ac becomes T or F**

Condition/Decision Coverage

– Condition: true, false.

– Decision: true, false.

Multiple Condition coverage (MCC)

– all possible combinations of condition outcomes in a decision

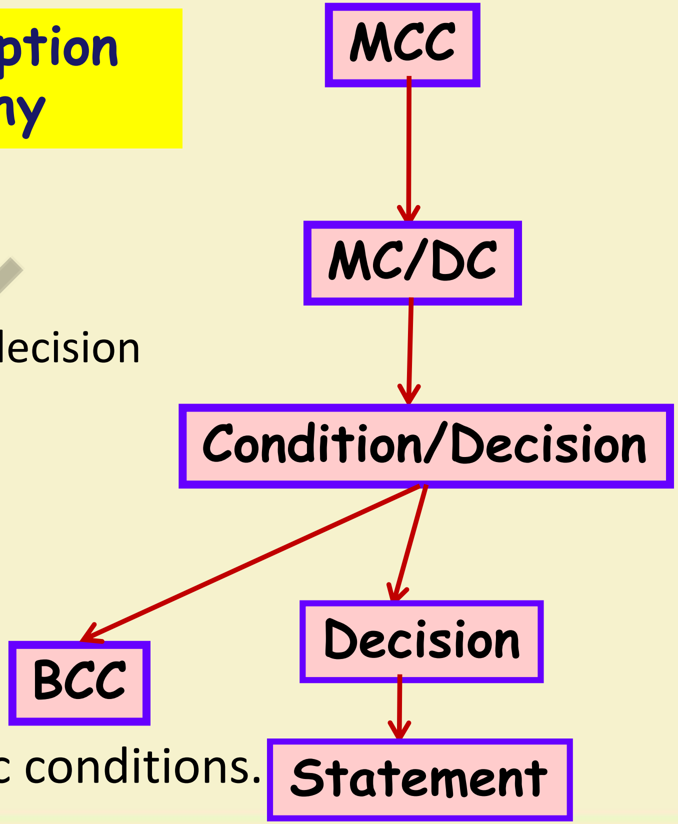
– for a decision with **n** conditions

2^n test cases are required

Modified Condition/Decision coverage (MC/DC)

- Bug-detection effectiveness almost similar to MCC
- Number of test cases linear in the number of basic conditions.

Subsumption hierarchy



- MC/DC stands for **M**odified **C**ondition / **D**ecision **C**overage

What is MC/DC?

- It is a condition coverage technique
  - **Condition:** Atomic conditions in expression.
  - **Decision:** Controls the program flow.
- **Main idea:** Each condition must be shown to independently affect the outcome of a decision.
  - **The outcome of a decision changes as a result of changing a single condition.**

Three Requirements for MC/DC

Requirement 1:

- Every decision in a program must take T/F values.

Requirement 2:

- Every condition in each decision must take T/F values.

Requirement 3:

- Each condition in a decision should independently affect the decision's outcome.

MC/DC Requirement 1

- **The decision is made to take both T/F values.**

If $\boxed{((\ a>10\ )\ \&\&\ ((\ b<50\ )\ \&\&\ (\ c==0\ )))\ \text{then}}$

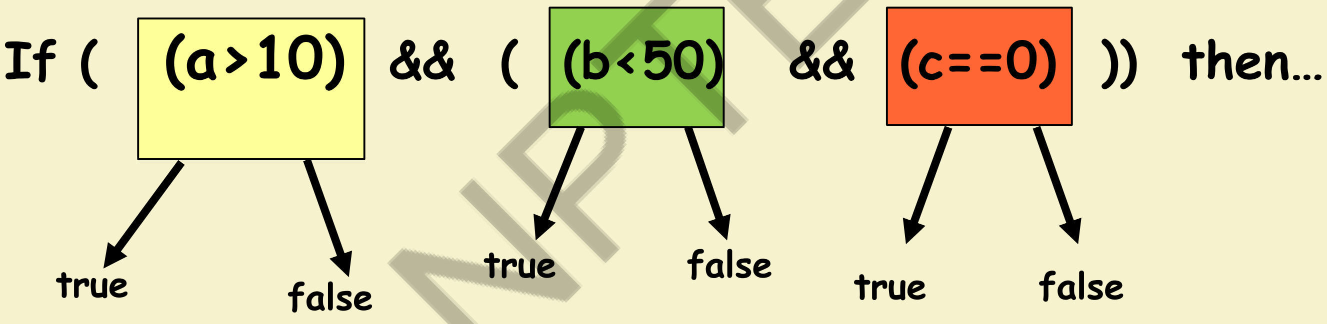
true

false

- This is as in Branch coverage.

MC/DC Requirement 2

- Test cases make every condition in the decision to evaluate to both T and F at least once.



MC/DC Requirement 3

- Every condition in the decision independently affects the decision's outcome.

If ((a>10) && ((b<50) || (c==0))) then…

true false true false

If ((a>50) && (b<50) || (c==0)) then…

true true false false

If ((a>50) && ((b<50) || (c==0))) then…

true false true false

MC/DC: An Example

- N+1 test cases required for N basic conditions
- Example:

(((a>10 || b<50) && c==0) || d<5) && e==10)

Test Case	a>10	b<50	c==0	d<5	e==10	outcome
(1)	<u>true</u>	false	<u>true</u>	false	<u>true</u>	true
(2)	false	<u>true</u>	true	false	true	true
(3)	true	false	false	<u>true</u>	true	true
(6)	true	false	true	false	<u>false</u>	false
(11)	true	false	<u>false</u>	<u>false</u>	true	false
(13)	<u>false</u>	<u>false</u>	true	false	true	false

- Underlined values independently affect the output of the decision

Creating MC/DC test cases

- Create truth table for conditions.
- Extend the truth table to represent test case pair that lead to show the independence influence of each condition.

Example : If (A and B) then . . .

Test Case Number	A	B	Decision	Test case pair for A	Test case pair for B
1	T	T	T	3	2
2	T	F	F		1
3	F	T	F	1	
4	F	F	F		

- Show independence of A :
  - Take 1 + 3
- Show independence of B :
  - Take 1 + 2
- Resulting test cases are
  - 1 + 2 + 3

If((A $\vee$ B) $\wedge$ C) ....

	A	B	C	Result	A	B	C	MC/DC
1	1	1	1	1			*	*
2	1	1	0	0			*	*
3	1	0	1	1	*			*
4	0	1	1	1		*		*
5	1	0	0	0				
6	0	1	0	0				
7	0	0	1	0	*	*		*
8	0	0	0	0				

Another Example

Minimal Set Example

If (A and (B or C)) then…

TC#	ABC	Result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

We want to determine the MINIMAL set of test cases

Here:

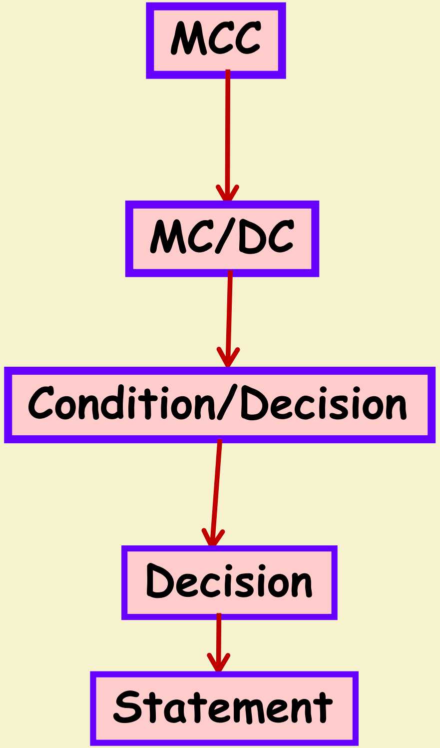
- {2,3,4,6}
- {2,3,4,7}

Non-minimal set is:

- {1,2,3,4,5}

Observation

- MC/DC criterion is stronger than condition/decision coverage criterion,
  - **but the number of test cases to achieve the MC/DC still linear in the number of conditions n in the decisions.**



MC/DC: Summary

- MC/DC essentially is :
  - basic condition coverage (C)
  - branch coverage (DC)
  - plus one additional condition (M):

    every condition must *independently affect* the decision's output
- It is subsumed by MCC and subsumes all other criteria discussed so far
  - stronger than statement and branch coverage
- **A good balance of thoroughness and test size and therefore widely used…**

Path Testing

Path Coverage

- Design test cases such that:
  - **All linearly independent paths in the program are executed at least once.**
- Defined in terms of
  - Control flow graph (CFG) of a program.

Path Coverage-Based Testing

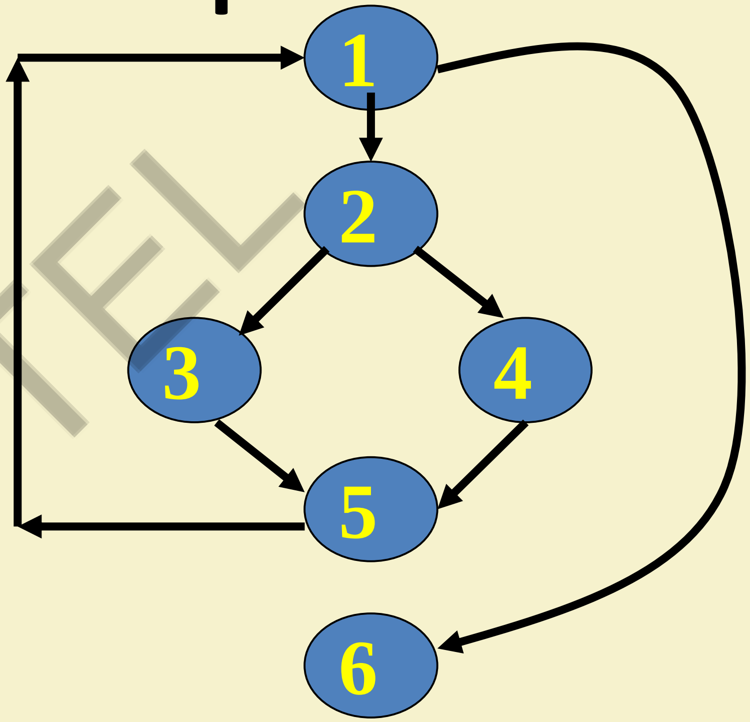
- To understand the path coverage-based testing:
  - We need to learn how to draw control flow graph of a program.
- A control flow graph (CFG) describes:
  - The sequence in which different instructions of a program get executed.
  - The way control flows through the program.

How to Draw Control Flow Graph?

- **Number all statements of a program.**
- Numbered statements:
  - Represent nodes of control flow graph.
- Draw an edge from one node to another node:
  - **If execution of the statement representing the first node can result in transfer of control to the other node.**

Example

```
int f1(int x,int y){
1 while (x != y){
2    if (x>y) then
3        x=x-y;
4    else y=y-x;
5  }
6 return x;        }
```



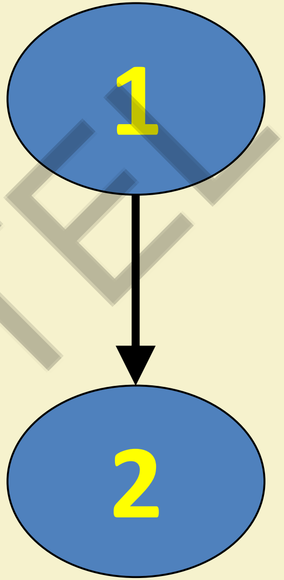
- Every program is composed of:
  - **Sequence**
  - **Selection**
  - **Iteration**

How to Draw Control flow Graph?

- If we know how to draw CFG corresponding these basic statements:
  - We can draw CFG for any program.

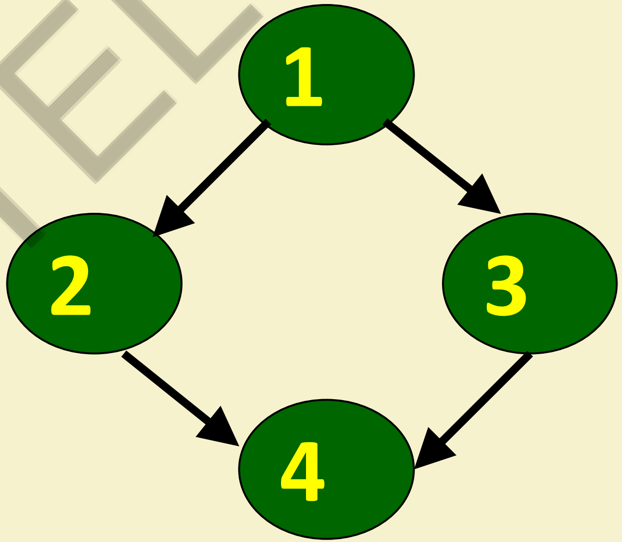
How to Draw Control flow Graph?

- **Sequence:**
  - 1 a=5;
  - 2 b=a*b-1;



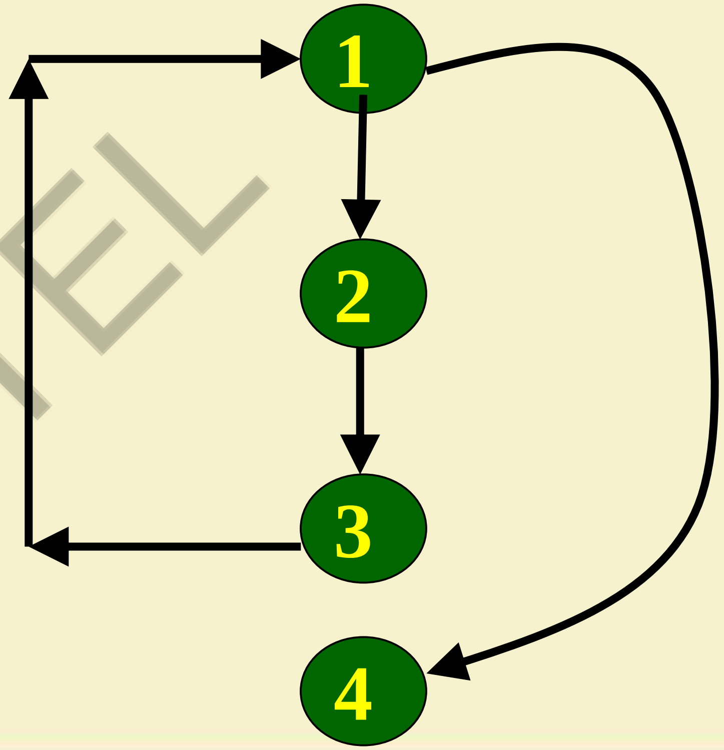
How to Draw Control Flow Graph?

- **Selection:**
  - 1 if(a>b) then
  - 2 c=3;
  - 3 else c=5;
  - 4 c=c*c;



How to Draw Control Flow Graph?

- **Iteration:**
  - 1 while(a>b){
  - 2 b=b*a;
  - 3 b=b-1;}
  - 4 c=b+d;

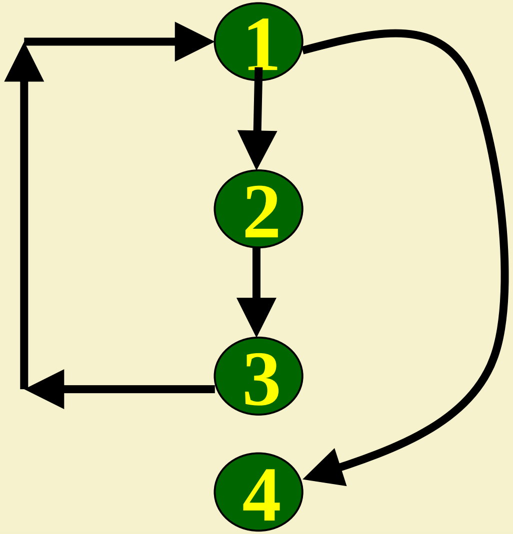


Path

- A path through a program:
  - A node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

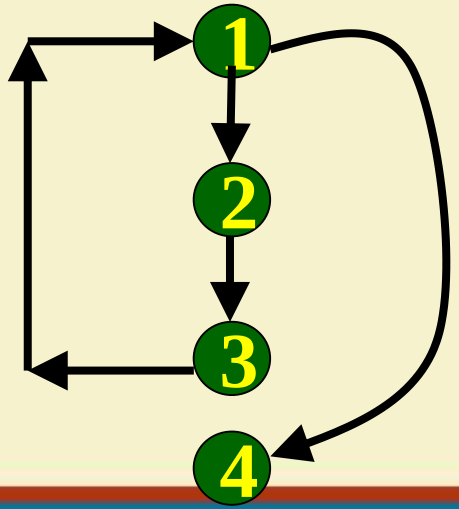
All Path Criterion

- In the presence of loops, the number paths can become extremely large:
  - This makes all path testing impractical



Linearly Independent Path

- Any path through the program that:
  - Introduces at least one new edge:
    - Not included in any other independent paths.



Independent path

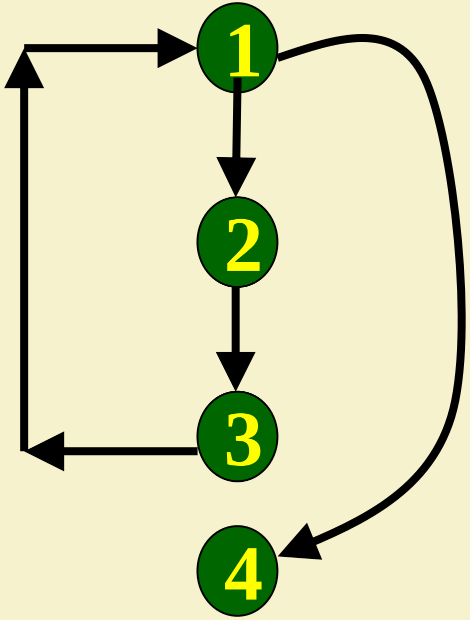
- It is straight forward:
  - To identify linearly independent paths of simple programs.
- For complicated programs:
  - It is not easy to determine the number of independent paths.

McCabe's Cyclomatic Metric

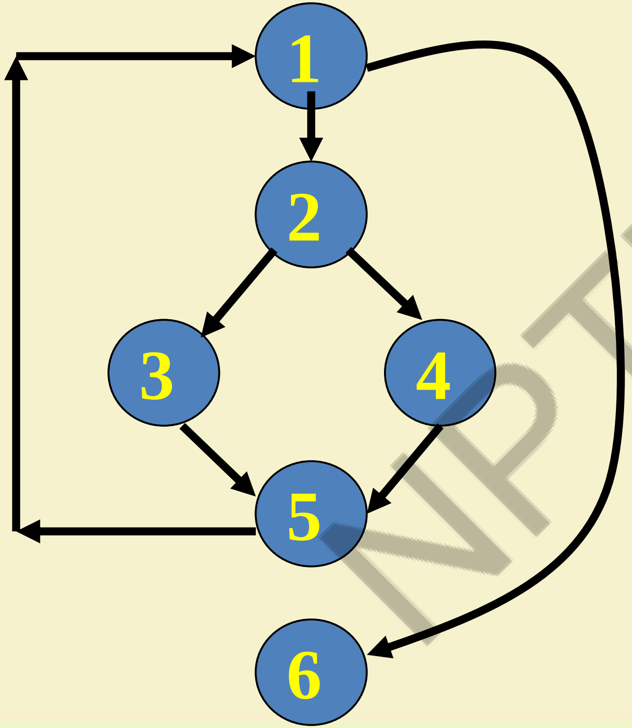
- An upper bound:
  - For the number of linearly independent paths of a program
- Provides a practical way of determining:
  - The maximum number of test cases required for basis path testing.

McCabe's Cyclomatic Metric

- Given a control flow graph G, cyclomatic complexity V(G):
  - **V(G)= E-N+2**
    - N is the number of nodes in G
    - E is the number of edges in G



Example Control Flow Graph

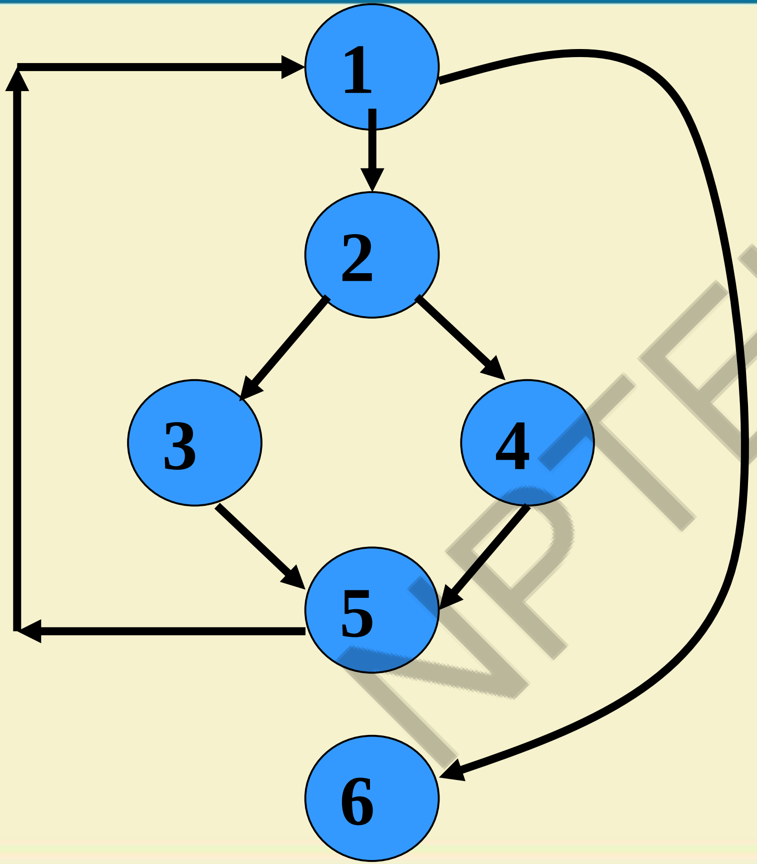


Cyclomatic complexity = 7-6+2 = 3.

Cyclomatic Complexity

- Another way of computing cyclomatic complexity:
  - –inspect control flow graph
  - –determine number of bounded areas in the graph
- V(G) = Total number of bounded areas + 1
  - –Any region enclosed by a nodes and edge sequence.

Example Control Flow Graph



Example

- From a visual examination of the CFG:
  - –Number of bounded areas is 2.
  - –Cyclomatic complexity = 2+1=3.

Cyclomatic Complexity

- McCabe's metric provides:
    - **A quantitative measure of testing difficulty and the reliability**
- Intuitively,
    - Number of bounded areas increases with the number of decision nodes and loops.

Cyclomatic Complexity

- The first method of computing V(G) is amenable to automation:
  - You can write a program which determines the number of nodes and edges of a graph
  - Applies the formula to find V(G).

Cyclomatic Complexity

- The cyclomatic complexity of a program provides:
  - A lower bound on the number of test cases to be designed
  - To guarantee coverage of all linearly independent paths.

Cyclomatic Complexity

- Knowing the number of test cases required:
  - Does not make it any easier to derive the test cases,
  - Only gives an indication of the minimum number of test cases required.

Practical Path Testing

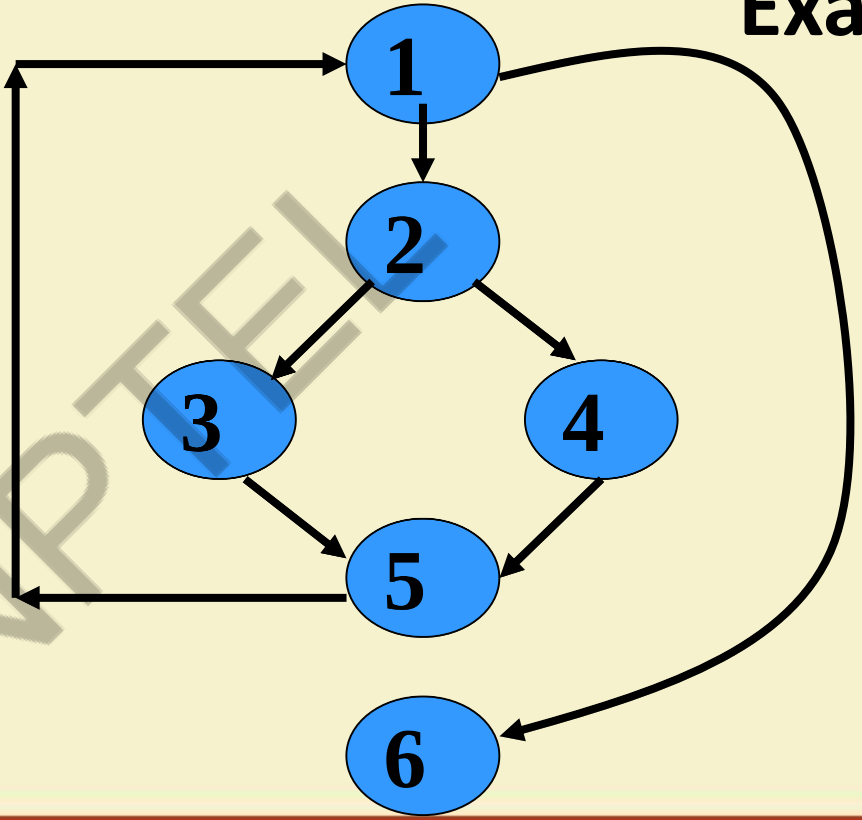
- The tester proposes initial set of test data :
  - –Using his experience and judgment.
- A dynamic program analyzer used:
  - –Measures which parts of the program have been tested
  - –Result used to determine when to stop testing.

Derivation of Test Cases

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - Force execution along each path.
  - Not practical for larger programs.

Example

```
int f1(int x,int y){
1 while (x != y){
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5  }
6 return x;        }
```



Derivation of Test Cases

- Number of independent paths: 3
  - 1,6 test case (x=1, y=1)
  - 1,2,3,5,1,6 test case(x=1, y=2)
  - 1,2,4,5,1,6 test case(x=2, y=1)

An Interesting Application of Cyclomatic Complexity

- Relationship exists between:
  - McCabe's metric
  - The number of errors existing in the code,
  - Time required to correct the errors.
  - Time required to understand the program

Cyclomatic Complexity

- Cyclomatic complexity of a program:
  - **Indicates the psychological complexity of a program.**
  - Difficulty level of understanding the program.

Cyclomatic Complexity

- From maintenance perspective,
  - Limit cyclomatic complexity of modules
    - To some reasonable value.
  - Good software development organizations:
    - Restrict cyclomatic complexity of functions to a maximum of ten or so.

Dataflow and Mutation Testing

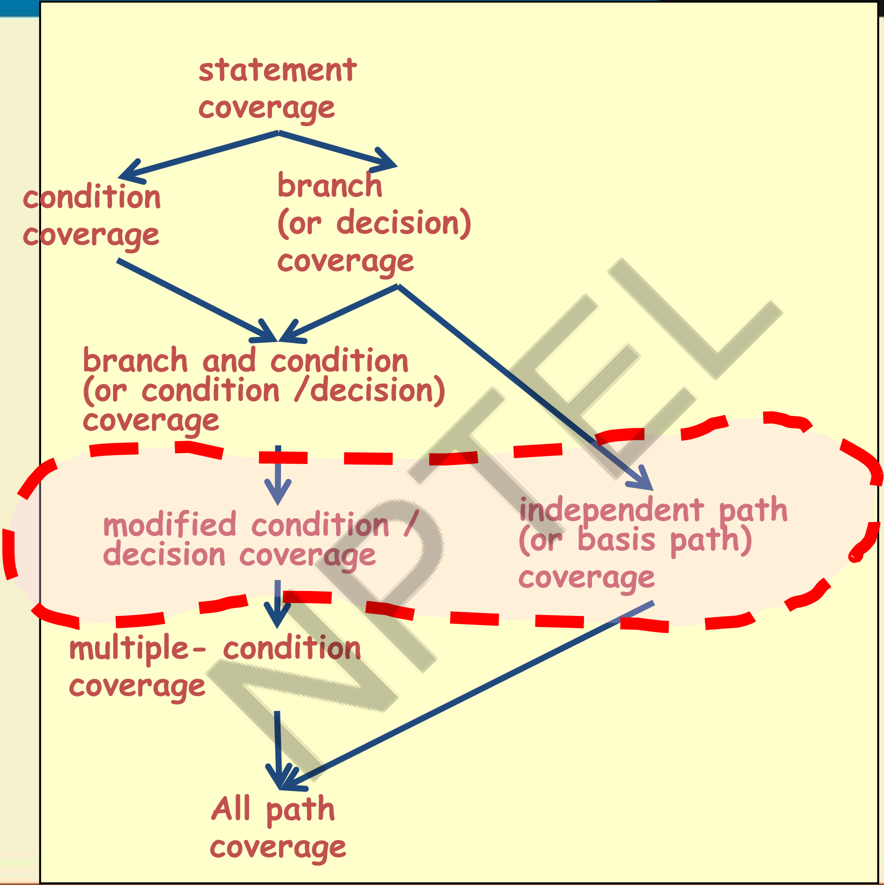
White Box Testing: Quiz

1. What do you mean by coverage-based testing?
2. What are the different types of coverage based testing?
3. How is a specific coverage-based testing carried out?
4. What do you understand by fault-based testing?
5. Give an example of fault-based testing?

weakest

Practically important coverage techniques

strongest



White-Box Testing

Data flow Testing

Data Flow-Based Testing

- Selects test paths of a program:
  - –According to the locations of
    - Definitions and uses of different variables in a program.

```
1 X(){
2       int a=5; /* Defines variable a */
          ....
3  While(c>5) {
4     if (d<50)
5              b=a*a;   /*Uses variable a */
6              a=a-1; /* Defines as well uses variable a */
...
7    }
8   print(a); } /*Uses variable a */
```

Data Flow-Based Testing

- For a statement numbered S,
  - –DEF(S) = {X/statement S contains a definition of X}
  - –USES(S)= {X/statement S contains a use of X}
  - –Example: **1: a=b;** DEF(1)={a}, USES(1)={b}.
  - –Example: 2: **a=a+b;** DEF(1)={a}, USES(1)={a,b}.

Data Flow-Based Testing

- A variable X is said to be live at statement S1, if
  - X is defined at a statement S:
  - There exists a path from S to S1 not containing any definition of X.

DU Chain Example

```
X(){
 int a=5; /* Defines variable a */
 While(c>5) {
    if (d<50)
          b=a*a;   /*Uses variable a */
          a=a-1; /* Defines variable a */
    }
  print(a); } /*Uses variable a */
```

Definition-use chain (DU chain)

- [X,S,S1],
  - –S and S1 are statement numbers,
  - –X in DEF(S)
  - –X in USES(S1), and
  - –the definition of X in the statement S is live at statement S1.

Data Flow-Based Testing

- One simple data flow testing strategy:
  - **Every DU chain in a program be covered at least once.**
- Data flow testing strategies:
  - Useful for selecting test paths of a program containing nested if and loop statements.

```
X(){
 B1;       /* Defines variable a */
 While(C1) {
    if (C2)
          if(C4) B4; /*Uses variable a */
       else B5;
       else  if (C3) B2;
       else B3;     }
 B6 }
```

Data Flow-Based Testing

Data Flow-Based Testing

- [a,1,5]: a DU chain.
- Assume:
  - DEF(X) = {B1, B2, B3, B4, B5}
  - USES(X) = {B2, B3, B4, B5, B6}
  - There are 25 DU chains.
- However only 5 paths are needed to cover these chains.

Mutation Testing

Mutation Testing

- In this, software is first tested:
  - Using an initial test suite designed using white-box strategies we already discussed.
- After the initial testing is complete,
  - Mutation testing is taken up.
- The idea behind mutation testing:
  - **Make a few arbitrary small changes to a program at a time.**

Main Idea

- Insert faults into a program:
  - Check whether the test suite is able to detect these.
  - This either validates or invalidates the test suite.

Mutation Testing Terminology

- Each time the program is changed:
  - –It is called a **mutated program**
  - –The change is called a **mutant.**

Mutation Testing

- A mutated program:
  - Tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - A mutant gives an incorrect result,
  - **Then the mutant is said to be dead.**

Mutation Testing

- If a mutant remains alive ---even after all test cases have been exhausted,
  - **The test suite is enhanced to kill the mutant.**
- The process of generation and killing of mutants:
  - **Can be automated by predefining a set of primitive changes that can be applied to the program.**

Mutation Testing

- Example primitive changes to a program:
  - **Deleting a statement**
  - **Altering an arithmetic operator,**
  - **Changing the value of a constant,**
  - **Changing a data type, etc.**

Traditional Mutation Operators

- **Deletion of a statement**
- Boolean:
  - **Replacement of a statement with another**

    eg. **==** and **>=, <** and **<=**
  - Replacement of **boolean expressions** with *true* or *false* eg. **a || b** with ***true***
- **Replacement of arithmetic operator**

  eg. * and +, / and -
- **Replacement of a variable** (ensuring same scope/type)

Underlying Hypotheses

- Mutation testing is based on the following two hypotheses:
  - **The Competent Programmer Hypothesis**
  - **The Coupling Effect**

Both of these were proposed by DeMillo *et al.*,1978

The Competent Programmer Hypothesis

- **Programmers create programs that are close to being correct:**
  - Differ from the correct program by some simple errors.

The Coupling Effect

- **Complex errors are caused due to several simple errors.**

- It therefore suffices to check for the presence of the simple errors

The Mutation Process