White-Box Testing
(Part 1)

Dr. RAJIB MALL

Professor

Department Of Computer Science & Engineering

IIT Kharagpur.

Defect Reduction Techniques

- Review
- Testing
- Formal verification
- Development process
- Systematic methodologies

Why Test?

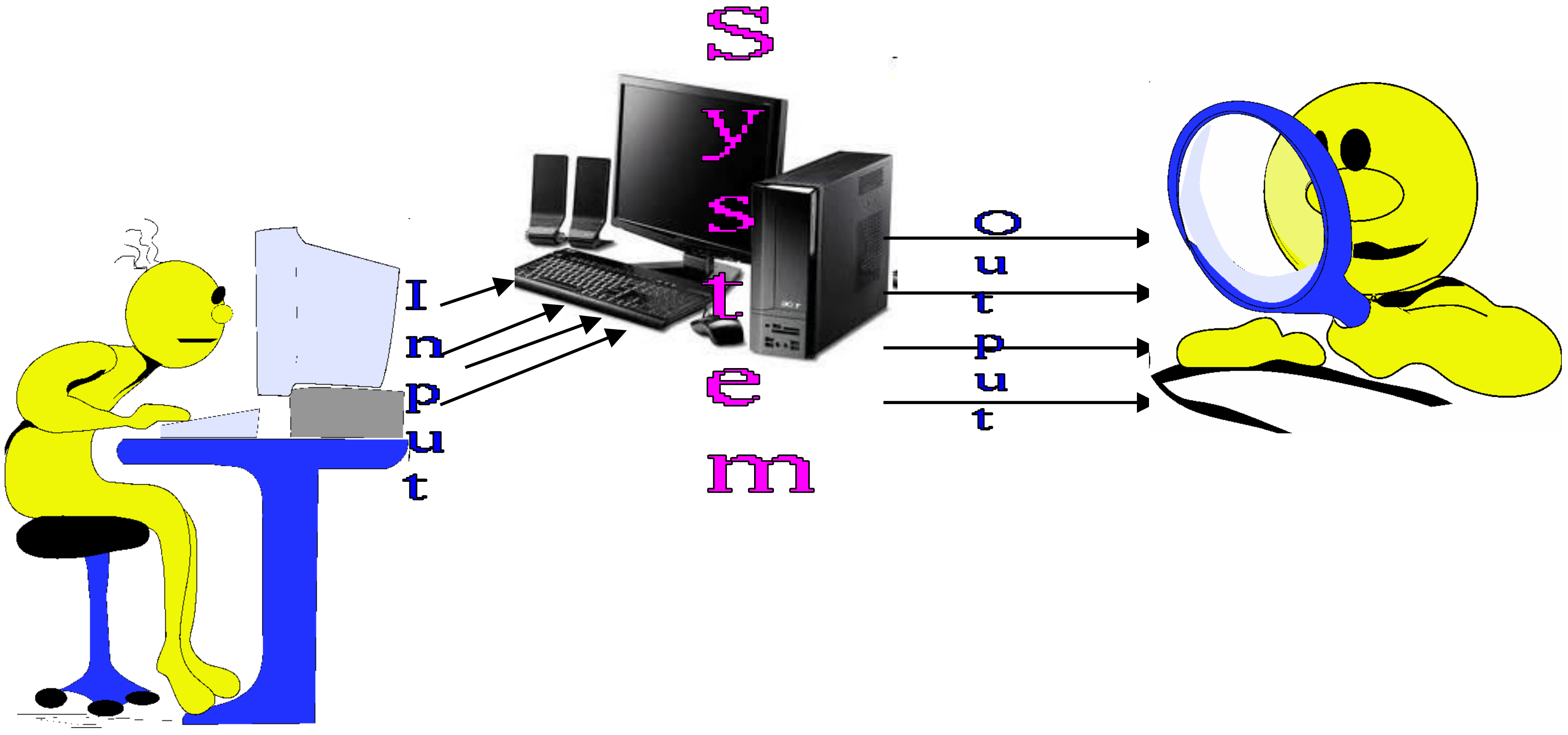


- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
  - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
    - The floating point number was larger than 32767
    - Efficiency considerations had led to the disabling of the exception handler.
- Total Cost: over $1 billion

How Do You Test a Program?

- Input test data to the program.
- Observe the output:
  - Check if the program behaved as expected.

How Do You Test a Program?



How Do You Test a Program?

- If the program does not behave as expected:
  - Note the conditions under which it failed.
  - Later debug and correct.

What's So Hard About Testing ?

- Consider `int proc1(int x, int y)`
- Assuming a 64 bit computer
  - Input space = 2^{128}
- Assuming it takes 10secs to key-in an integer pair
  - It would take about a billion years to enter all possible values!
  - Automatic testing has its own problems!

Testing Facts

- Consumes largest effort among all phases
  - Largest manpower among all other development roles
  - Implies more job opportunities
- About 50% development effort
  - But 10% of development time?
  - How?

Testing Facts

- Testing is getting more complex and sophisticated every year.
  - Larger and more complex programs
  - Newer programming paradigms

Overview of Testing Activities

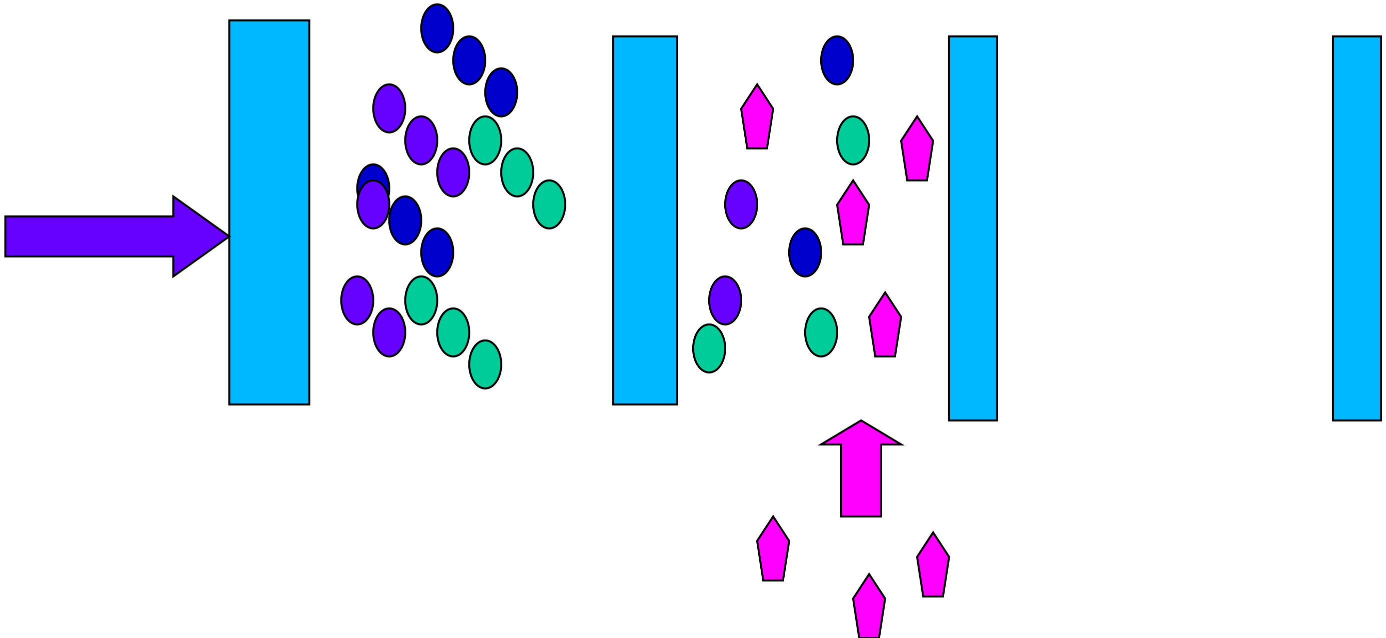
- Test Suite Design
- Run test cases and observe results to detect failures.
- Debug to locate errors
- Correct errors.

Error, Faults, and Failures

- A failure is a manifestation of an error (also defect or bug).
  - Mere presence of an error may not lead to a failure.

Pesticide Effect

- Errors that escape a fault detection technique:
  - Can not be detected by further applications of that technique.



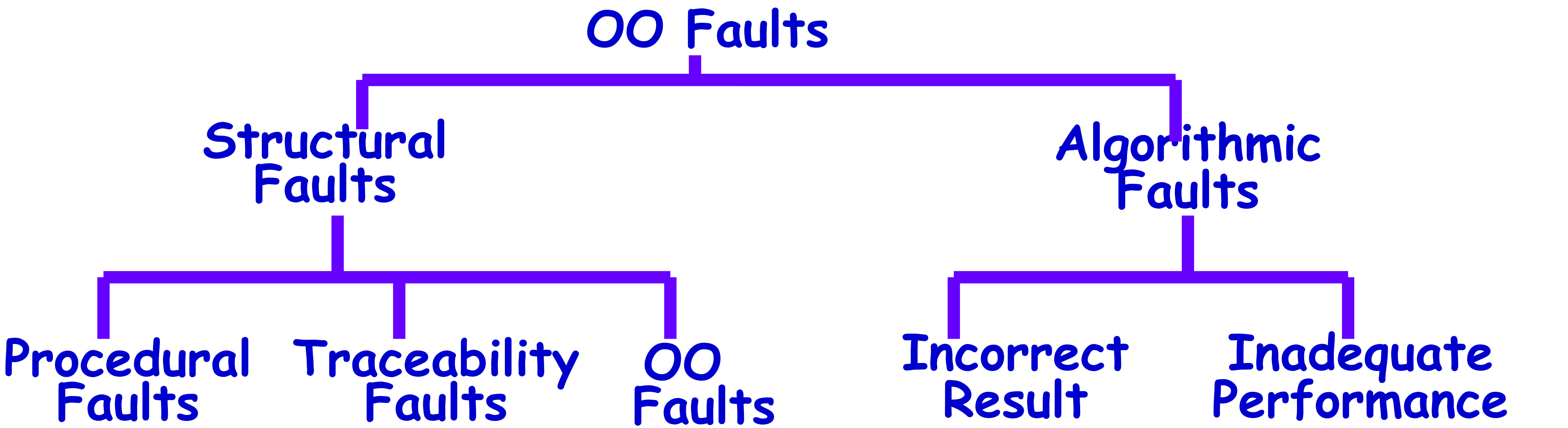
Pesticide Effect

- Assume we use 4 fault detection techniques and 1000 bugs:
  - Each detects only 70% bugs
  - How many bugs would remain
  - $1000*(0.3)^4=81$ bugs

Fault Model

- Types of faults possible in a program.
- Some types can be ruled out
  - Concurrency related-problems in a sequential program

Fault Model of an OO Program



Hardware Fault-Model

- Simple:
  - Stuck-at 0
  - Stuck-at 1
  - Open circuit
  - Short circuit
- Simple ways to test the presence of each
- Hardware testing is fault-based testing

Software Testing

- Each test case typically tries to establish correct working of some functionality
  - Executes (covers) some program elements
  - For restricted types of faults, fault-based testing exists.

Test Cases and Test Suites

- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the test suite

Test Cases and Test Suites

- A test case is a triplet [I,S,O]
  - I is the data to be input to the system,
  - S is the state of the system at which the data will be input,
  - O is the expected output of the system.

Verification versus Validation

- Verification is the process of determining:
  - Whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining:
  - Whether a fully developed system conforms to its SRS document.

Verification versus Validation

- Verification is concerned with phase containment of errors,
  - Whereas the aim of validation is that the final product be error free.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - Input data domain is extremely large.
- Design an optimal test suite:
  - Of reasonable size and
  - Uncovers as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
  - Many test cases would not contribute to the significance of the test suite,
  - Would not detect errors not already being detected by other test cases in the suite.
- Number of test cases in a randomly selected test suite:
  - Not an indication of effectiveness of testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - Does not mean that many errors in the system will be uncovered.
- Consider following example:
  - Find the maximum of two integers x and y.

Design of Test Cases

- The code has a simple programming error:
- If (x>y) max = x;
  else max = x;
- Test suite {(x=3,y=2);(x=2,y=3)} can detect the error,
- A larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error.

Design of Test Cases

- Systematic approaches are required to design an optimal test suite:
  - Each test case in the suite should detect different errors.

Design of Test Cases

- There are essentially three main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach
  - Grey-box testing

Black-Box Testing

- Test cases are designed using only functional specification of the software:
  - Without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as functional testing.

White-box Testing

- Designing white-box test cases:
  - Requires knowledge about the internal structure of software.
  - White-box testing is also called structural testing.
  - In this unit we will not study white-box testing.

White-Box Testing

- There exist several popular white-box testing methodologies:
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - MC/DC coverage
  - Mutation testing
  - Data flow-based testing

Why Both BB and WB Testing?

<u>Black-box</u>

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- Does not tell if extra functionality has been implemented.

<u>White-box</u>

- Does not address the question of whether or not a program matches the specification
- Does not tell you if all of the functionality has been implemented
- Does not discover missing program logic

Coverage-Based Testing Versus Fault-Based Testing

- Idea behind coverage-based testing:
  - Design test cases so that certain program elements are executed (or covered).
  - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing:
  - Design test cases that focus on discovering certain types of faults.
  - Example: Mutation testing.

Statement Coverage

- Statement coverage methodology:
  - Design test cases so that every statement in the program is executed at least once.

Statement Coverage

- The principal idea:
  - Unless a statement is executed,
  - We have no way of knowing if an error exists in that statement.

Statement Coverage Criterion

- Observing that a statement behaves properly for one input value:
  - No guarantee that it will behave correctly for all input values.

Statement Testing

- Coverage measurement:

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

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

Example

```
int f1(int x, int y){
while (x != y){
    if (x>y) then
        x=x-y;
    else y=y-x;
}
return x;        }
```

Euclid's GCD Algorithm

Euclid's GCD Algorithm

- By choosing the test set $\{(x=3,y=3),(x=4,y=3), (x=3,y=4)\}$
  - All statements are executed at least once.

Branch Coverage

- Test cases are designed such that:
  - Different branch conditions
    - Given true and false values in turn.

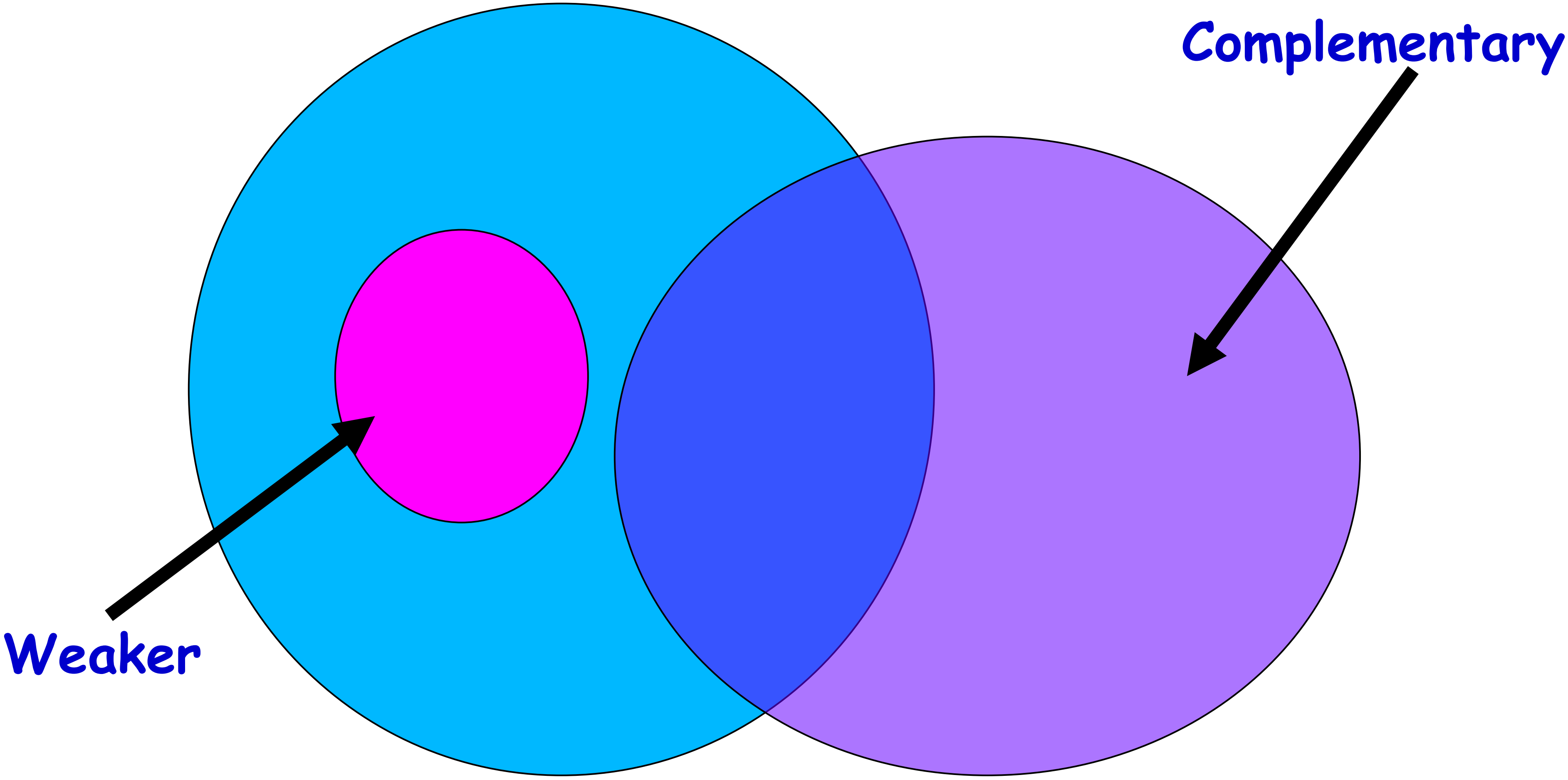
Branch Coverage

- Branch testing guarantees statement coverage:
  - A stronger testing compared to the statement coverage-based testing.

Stronger Testing

- Test cases are a superset of a weaker testing:
  - *A* stronger testing covers at least all the elements of the elements covered by a weaker testing.

Stronger, Weaker, and Complementary Testing



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; }

Example

- Test cases for branch coverage can be:
- {(x=3,y=3),(x=3,y=2), (x=4,y=3), (x=3,y=4)}

Branch Testing

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

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

Statements vs Branch Testing

- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

All Branches can still miss conditions

- Sample fault: missing operator (negation)

  digit_high == 1 || digit_low == -1

- Branch adequacy criterion can be satisfied by varying only digit_low
  - The faulty sub-expression might never determine the result
  - We might never really test the faulty condition, even though we tested both outcomes of the branch

Condition Coverage

- Test cases are designed such that:
  - Each component of a composite conditional expression
    - Given both true and false values.

Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once,
  - i.e. given true and false values.

Basic condition testing

- Adequacy criterion: each basic condition must be executed at least once
- Coverage:

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

Branch Testing

- Branch testing is the simplest condition testing strategy:
  - Compound conditions appearing in different branch statements
    - Are given true and false values.

Branch Testing

- Condition testing:
  - Stronger testing than branch testing.
- Branch testing:
  - Stronger than statement coverage testing.

Condition Coverage

- Consider a boolean expression having n components:
  - For condition coverage we require 2^n test cases.
- Condition coverage-based testing technique:
  - Practical only if n (the number of component conditions) is small.

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 shown to independently affect the outcome of each decision
- 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

What is MC/DC?

- MC/DC stands for **Modified Condition / Decision Coverage**
- A kind of Predicate Coverage technique
  - Condition: Leaf level Boolean expression.
  - Decision: Controls the program flow.
- Main idea: Each condition must be shown to independently affect the outcome of a decision, i.e. the outcome of a decision changes as a result of changing a single condition.

Condition Coverage

- Every condition in the decision has taken all possible outcomes at least once.
- If ((a) && (b) || (c)) then…

  true false true false true false

MC/DC Coverage

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

```
If ((a)    &&    (b)    ||    (c)) then…
```

true *false* *true* *false*

Change the value of each condition individually while keeping all other conditions constant.

MC/DC: linear complexity

- N+1 test cases for N basic conditions

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

Test Case	a	b	c	d	e	outcome
(1)	<u>true</u>	--	<u>true</u>	--	<u>true</u>	true
(2)	false	<u>true</u>	true	--	true	true
(3)	true	--	false	<u>true</u>	true	true
(6)	true	--	true	--	<u>false</u>	false
(11)	true	--	<u>false</u>	<u>false</u>	--	false
(13)	<u>false</u>	<u>false</u>	--	false	--	false

- Underlined values independently affect the output of the decision
- Required by the RTCA/DO-178B standard

Comments on MC/DC

- MC/DC 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 compound conditions 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)

Creating MC/DC test cases

`If (A and B) then…`

- (1) create truth table for conditions.
- (2) Extend truth table so that it indicated which test cases can be used to show the independence of each condition.

A B	result
T T	T
T F	F
F T	F
F F	F

number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		

Creating test cases cont'd

number	A B	result	A	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
  - (T , T) + (T , F) + (F , T)

More advanced example

```
If (A and (B or C)) then…
```

number	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			

Note: 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}

Where does it fit in?

- The MC/DC criterion is much stronger than the condition/decision coverage criterion, but the number of test cases to achieve the MC/DC criterions still varies linearly with the number of conditions n in the decisions.
  - Much more complete coverage than condition/decision coverage, but
  - at the same time it is not terribly costly in terms of number of test cases.

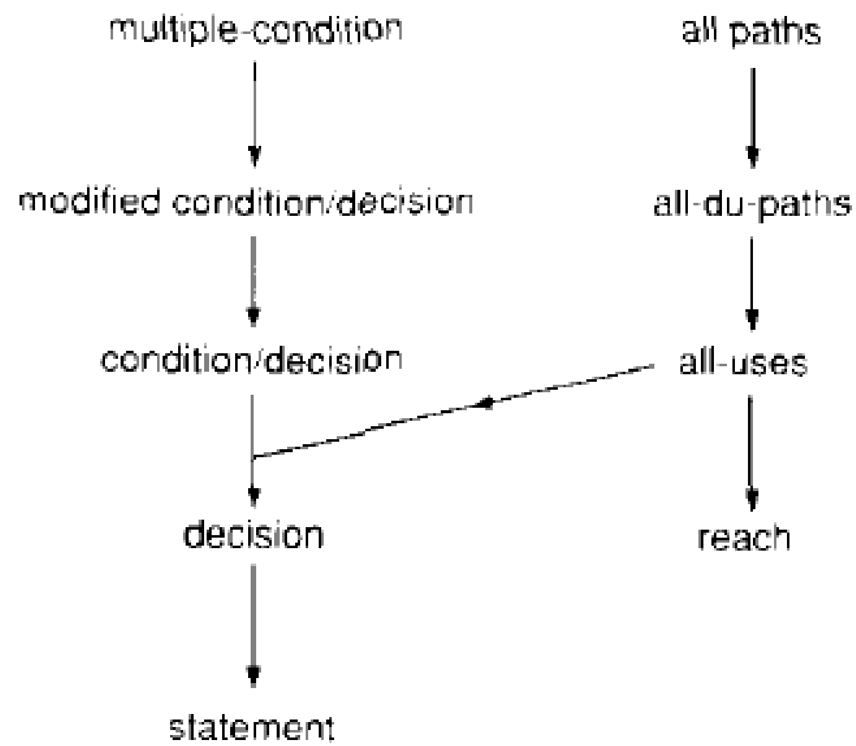


Fig. 2 Subsumption hierarchy

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.

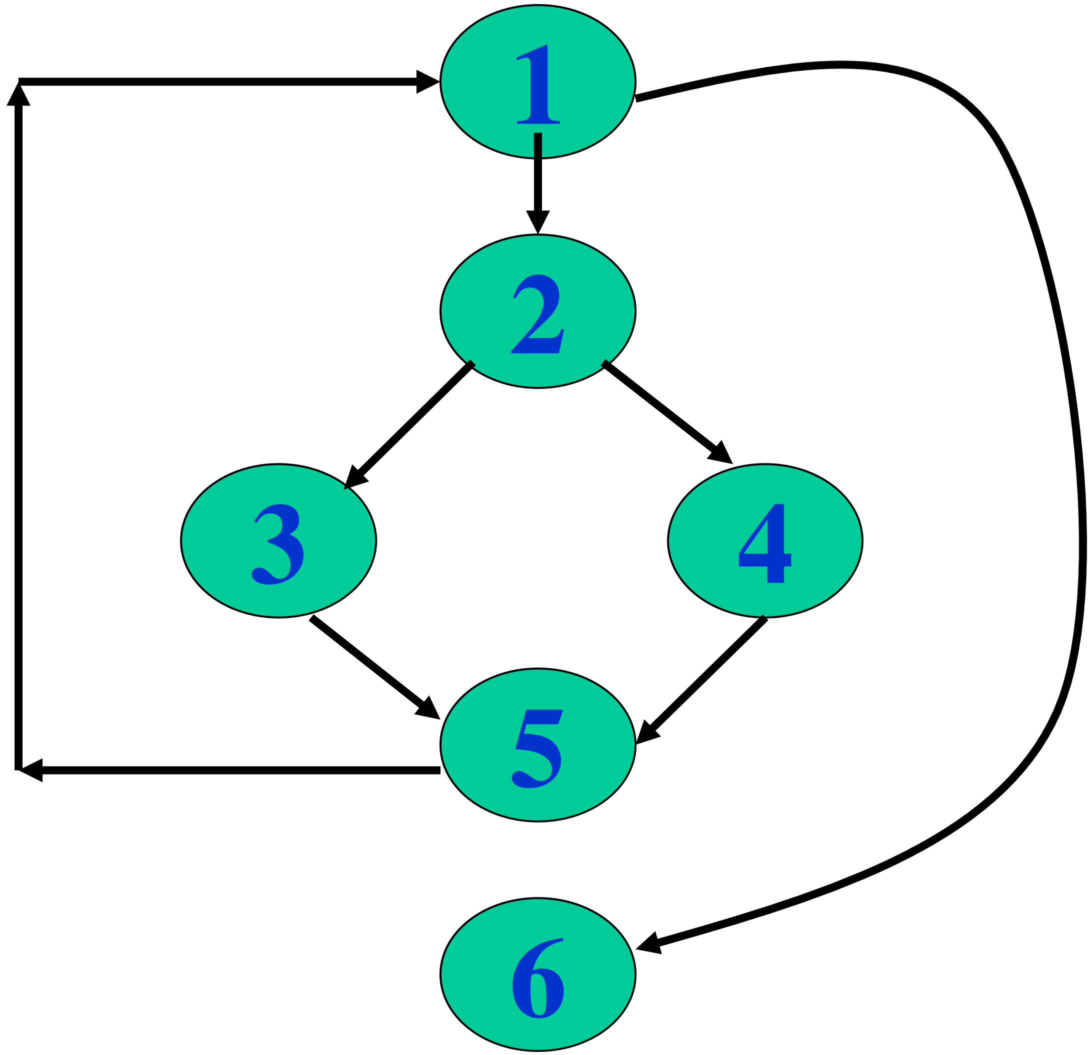
How to Draw Control Flow Graph?

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

Example

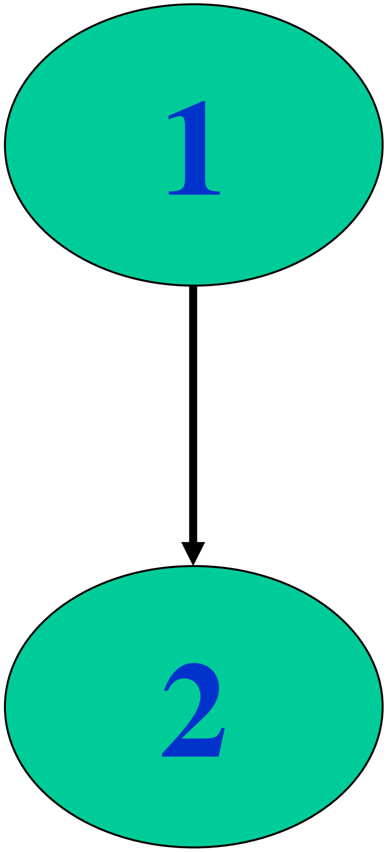
- int f1(int x,int y){
- 1 while (x != y){
- 2   if (x>y) then
- 3   x=x-y;
- 4   else y=y-x;
- 5 }
- 6 return x;   }

Example Control Flow Graph



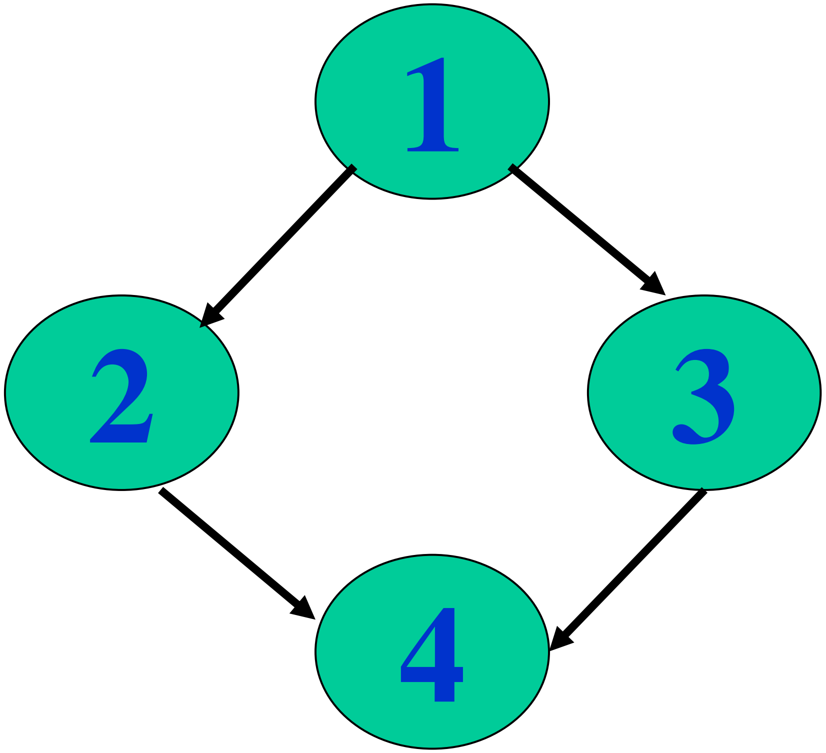
How to Draw Control flow Graph?

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



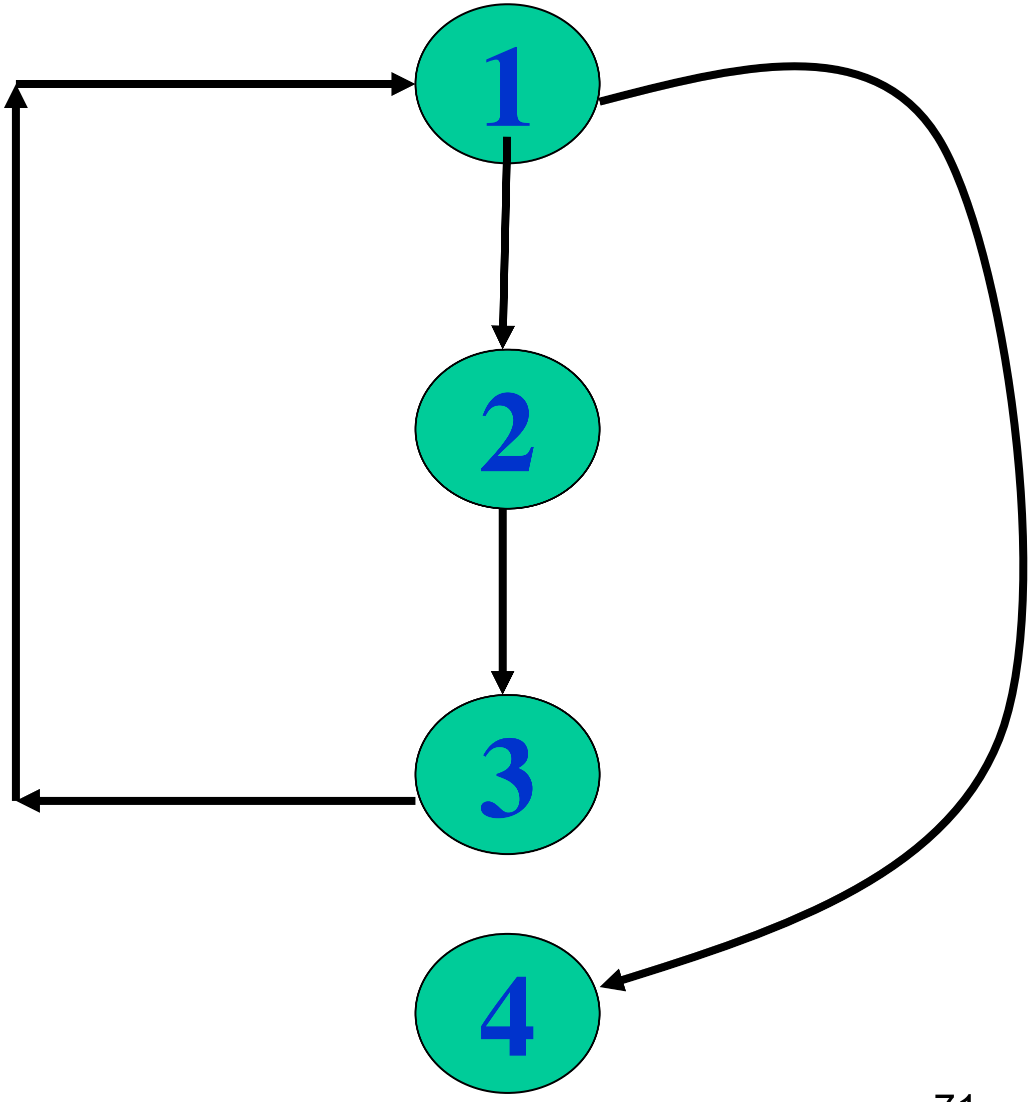
How to Draw Control Flow Graph?

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



How to Draw Control Flow Graph?

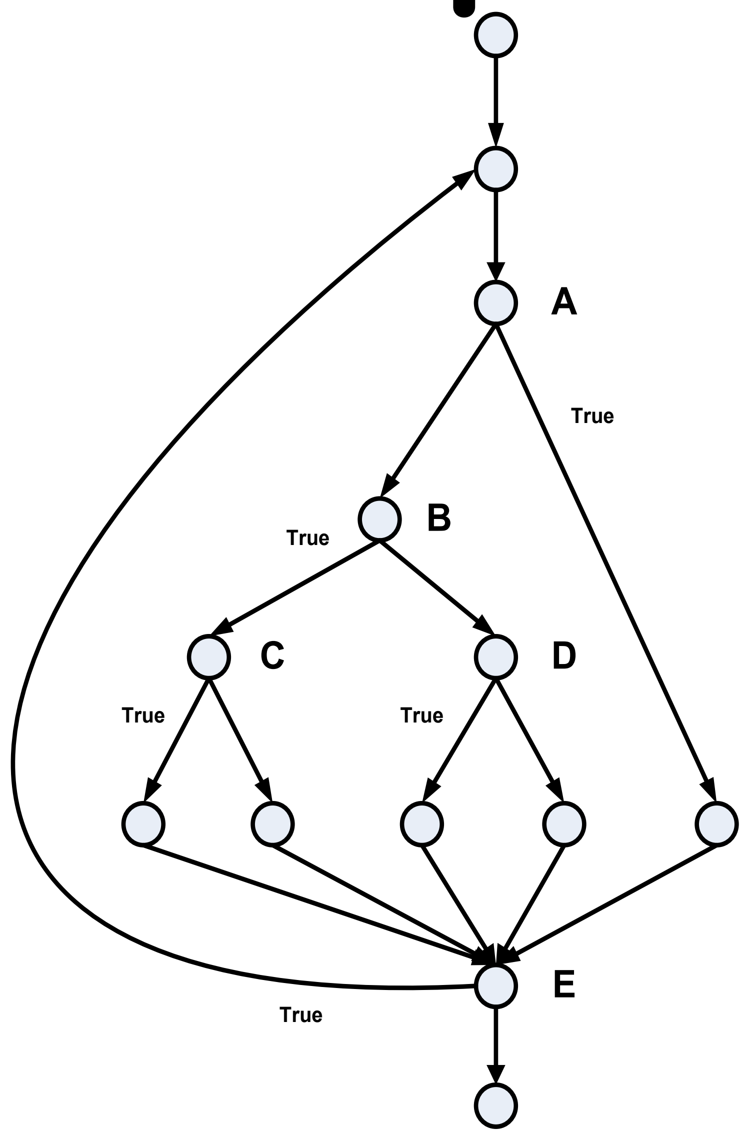
- Iteration:
  - 1 while(a>b){
  - 2 b=b*a;
  - 3 b=b-1;}
  - 4 c=b+d;



Example Code Fragment

```
Do
{
  if (A) then {...};
  else {
    if (B) then {
            if (C) then {...};
            else {…}
    }
    else if (D) then {...};
         else {...};
  }
}
While (E);
```

Example Control Flow Graph



Source: The Art of Software Testing – Glenford Myers

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.

Linearly Independent Path

- Any path through the program:
  - 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.

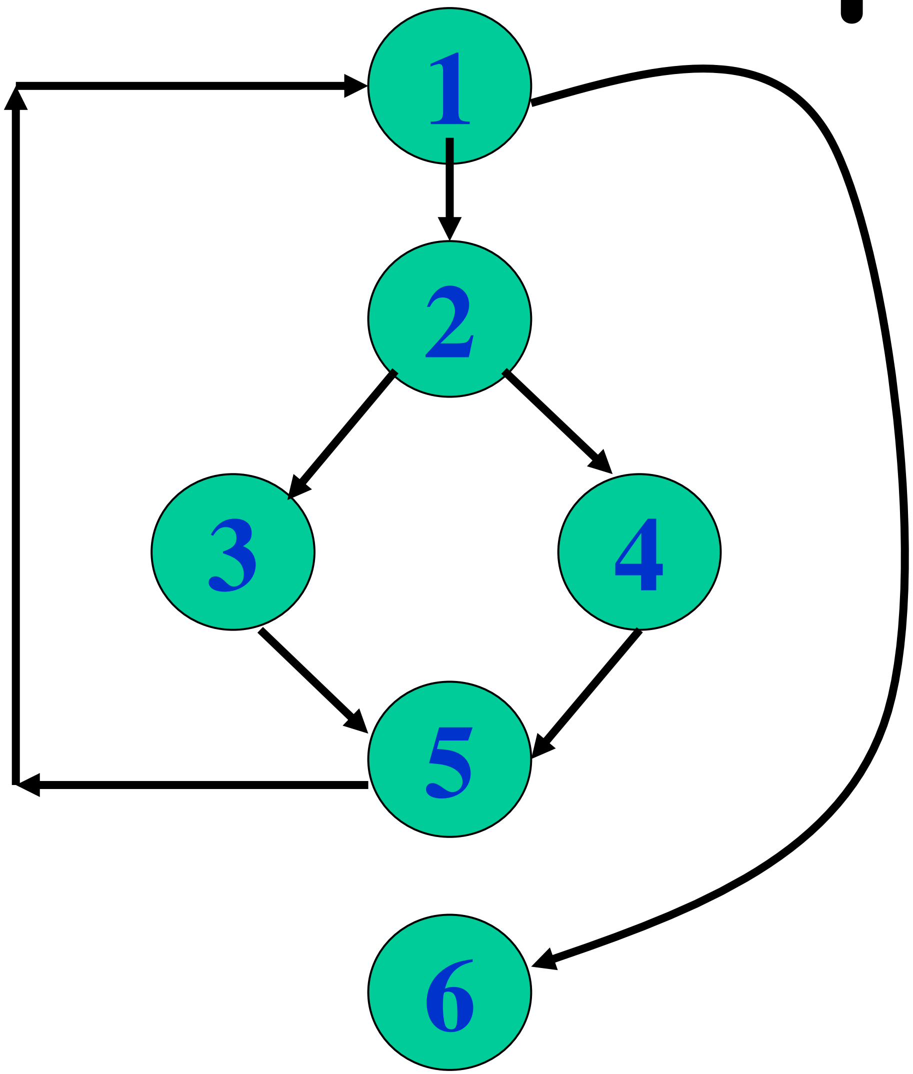
McCabe's Cyclomatic Metric

- An upper bound:
  - For the number of linearly independent paths of a program
- Provides a practical way of determining:
  - The maximum number of linearly independent paths in a program.

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

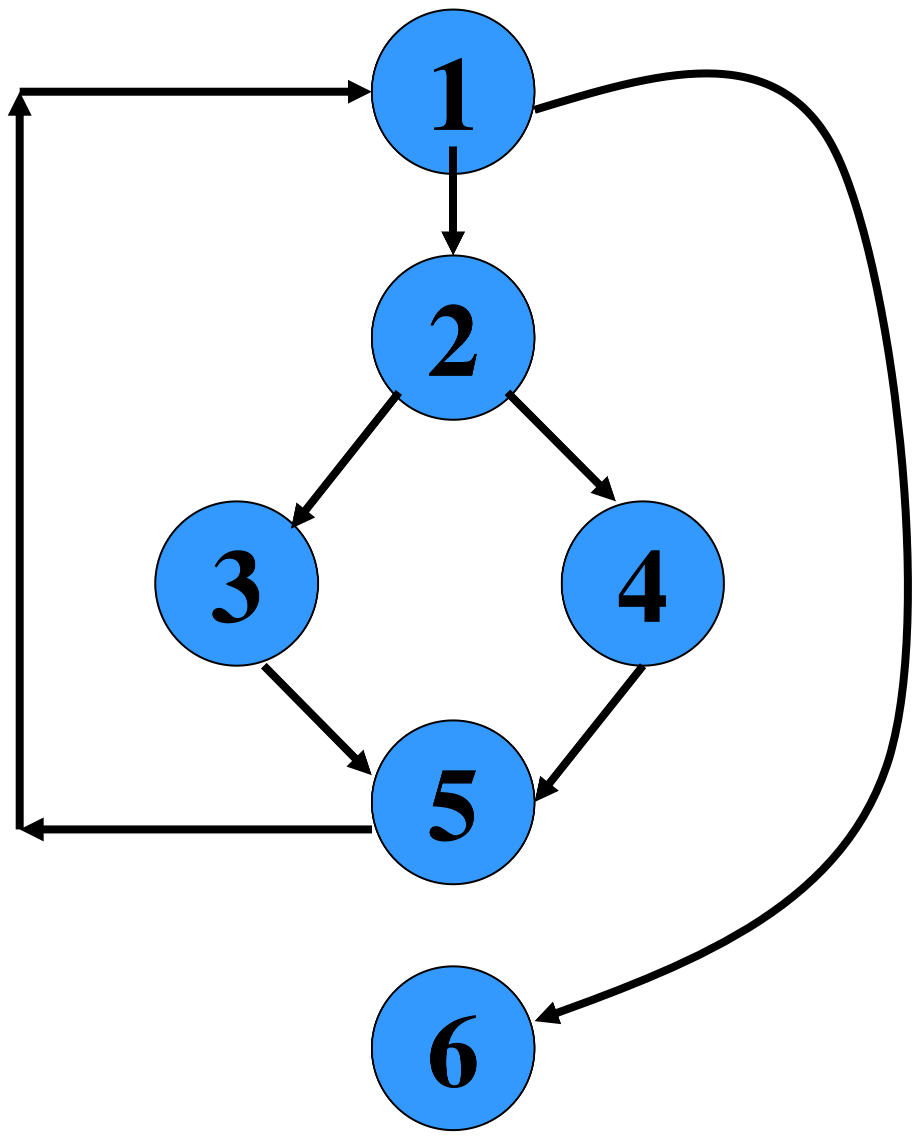


Cyclomatic complexity = 7-6+2 = 3.

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 ultimate 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

- A measure of the number of independent paths in a program.
- Provides a lower bound:
  - for the number of test cases for path coverage.

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 judgement.
- A dynamic program analyzer used:
  - Measures which parts of the program have been tested
  - Result used to determine when to stop testing.

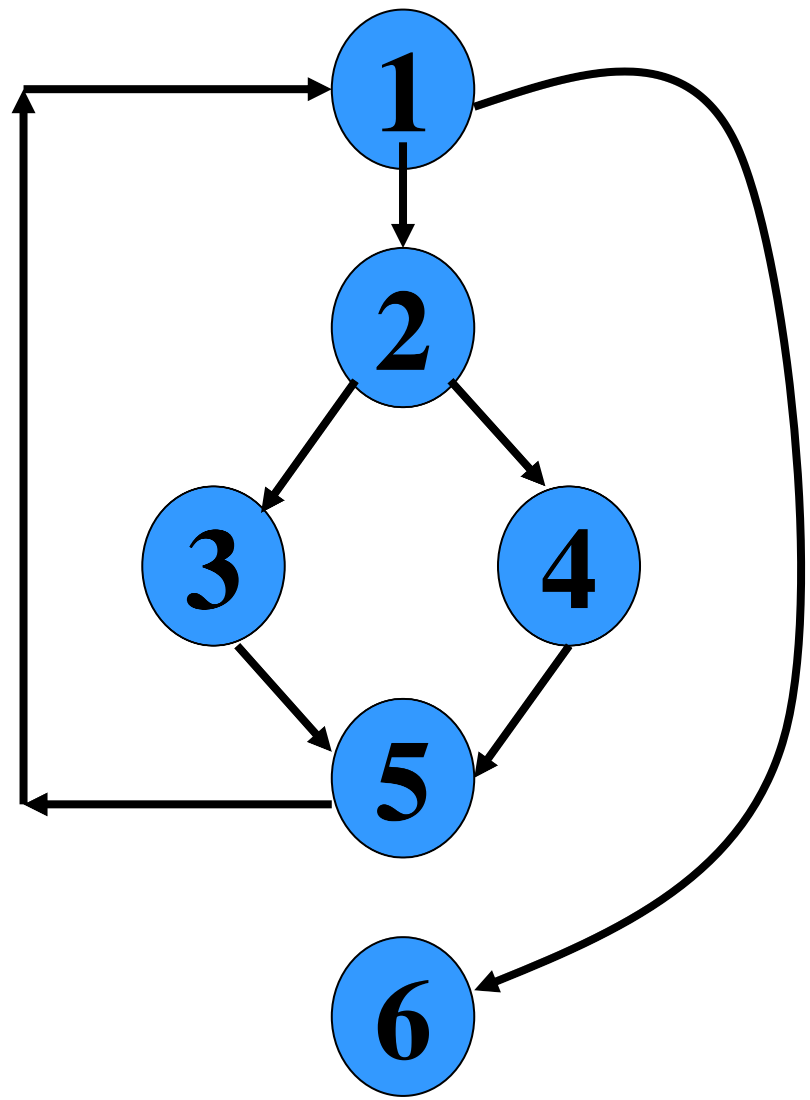
Derivation of Test Cases

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

Example

```
int f1(int x,int y){
  while (x != y){
      if (x>y) then
          x=x-y;
      else y=y-x;
  }
  return x;        }
```

Example Control Flow Diagram



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,
  - The time required to find and correct the errors.

Cyclomatic Complexity

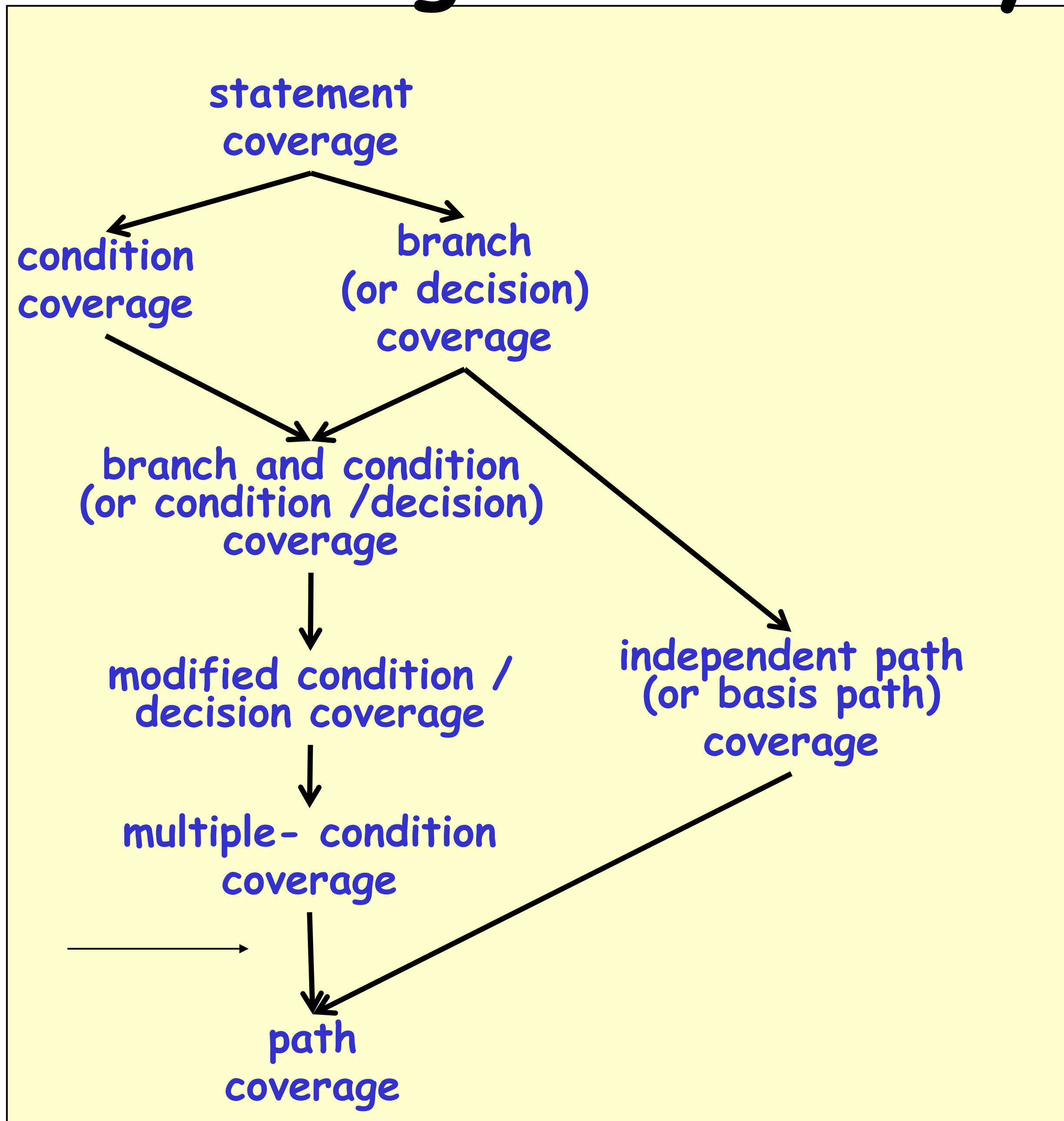
- Cyclomatic complexity of a program:
  - Also indicates the psychological complexity of a program.
  - Difficulty level of understanding the program.

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.

White-Box Testing : Summary

weakest



only if paths across composite conditions are distinguished

strongest

Data Flow-Based Testing

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

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(){
 a=5; /* Defines variable a */
 While(C1) {
   if (C2)
        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.

Data Flow-Based Testing

```
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

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

Mutation Testing

- The software is first tested:
  - using an initial testing method based on 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.

Mutation Testing

- 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

- The primitive changes can be:
  - altering an arithmetic operator,
  - changing the value of a constant,
  - changing a data type, etc.

Mutation Testing

- A major disadvantage of mutation testing:
  - computationally very expensive,
  - a large number of possible mutants can be generated.

Summary

- Exhaustive testing of non-trivial systems is impractical:
  - We need to design an optimal set of test cases
    - Should expose as many errors as possible.
- If we select test cases randomly:
  - many of the selected test cases do not add to the significance of the test set.

Summary

- There are two approaches to testing:
  - black-box testing and
  - white-box testing.
- Designing test cases for black box testing:
  - does not require any knowledge of how the functions have been designed and implemented.
  - Test cases can be designed by examining only SRS document.

Summary

- White box testing:
  - Requires knowledge about internals of the software.
  - Design and code is required.
- We have discussed a few white-box test strategies.
  - Statement coverage
  - branch coverage
  - condition coverage
  - path coverage

Summary

- A stronger testing strategy:
  - Provides more number of significant test cases than a weaker one.
  - Condition coverage is strongest among strategies we discussed.
- We discussed McCabe's Cyclomatic complexity metric:
  - Provides an upper bound for linearly independent paths
  - Correlates with understanding, testing, and debugging difficulty of a program.