SWE 4603

Software Testing and Quality Assurance

Lecture 4

Prepared By Maliha Noushin Raida, Lecturer, CSE
Islamic University of Technology

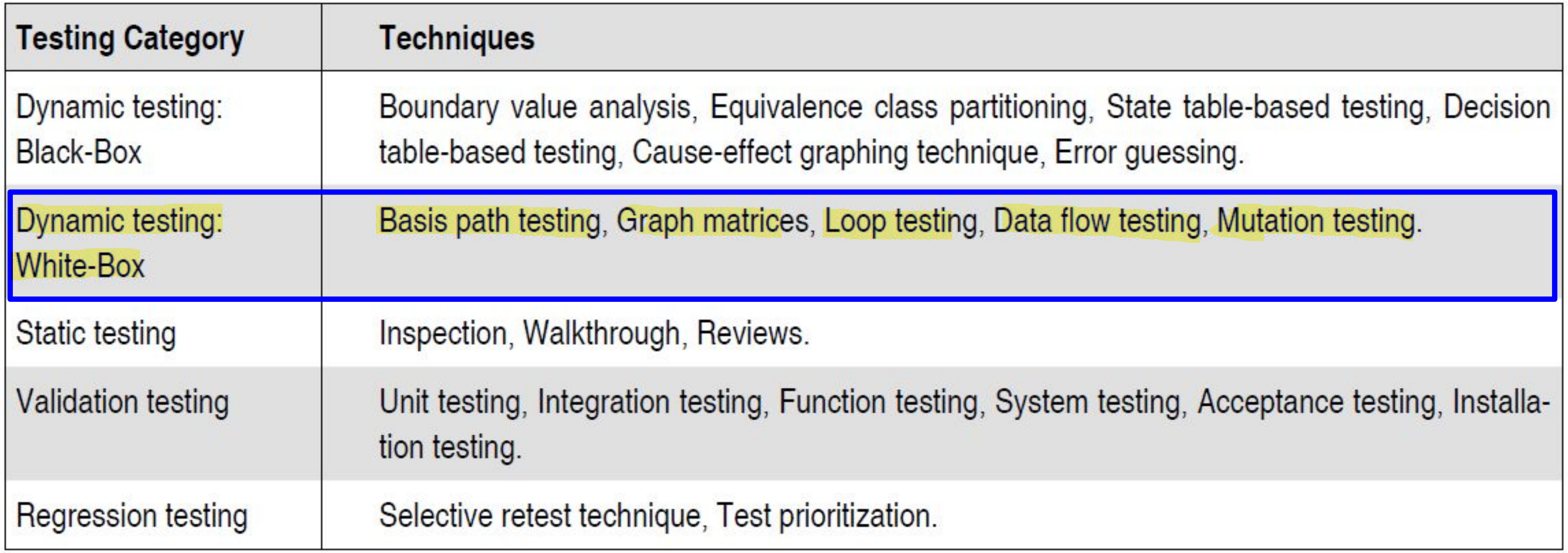
Lesson Outcome

- White-box testing demands complete understanding of the program logic/ structure
- Test case designing using white-box testing techniques
- Basis path testing method
- Building a path testing tool using graph matrices
- Loop testing
- Data flow testing method
- Mutation testing method

Week 4 On:

- Chapter 5: Dynamic Testing: White-Box Testing Techniques

Testing Categories & Techniques



Testing Category	Techniques
Dynamic testing: Black-Box	Boundary value analysis, Equivalence class partitioning, State table-based testing, Decision table-based testing, Cause-effect graphing technique, Error guessing.
Dynamic testing: White-Box	Basis path testing, Graph matrices, Loop testing, Data flow testing, Mutation testing.
Static testing	Inspection, Walkthrough, Reviews.
Validation testing	Unit testing, Integration testing, Function testing, System testing, Acceptance testing, Installation testing.
Regression testing	Selective retest technique, Test prioritization.

Dynamic Testing

White-Box Testing Techniques

White-Box Testing

- A “white-box” or “clear-box” mean that one can see the internals of the system, or in other words, see inside the box.
- So a test or a testing activity that is considered “white-box” is one where the tester has access to the code and uses that knowledge to write tests or conduct the testing activity.
- example of “white-box” test is unit test where developers write test cases, often automated, to verify each unit of code such as a function, method, or class.
- white-box testing ensures that the internal parts of the software are adequately tested.

White-Box Testing: Necessity

- white-box testing techniques are used for testing the module for initial stage testing.
- Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but not through black-box testing.
- Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
- Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques.
- White-box testing explores and confirms the testing of logical paths.

White-Box Testing: Logical Coverage Criteria

Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered.

Basic forms of logic coverage:

- **Statement Coverage**
- **Decision or Branch Coverage**
- **Condition Coverage**

Statement Coverage

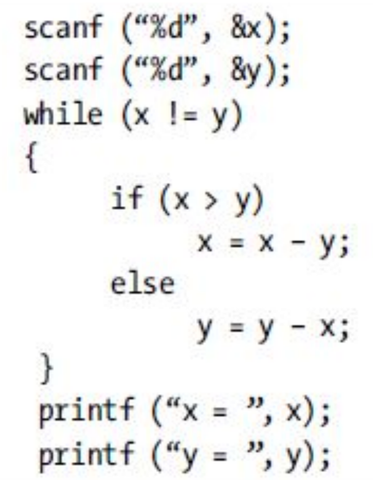
What is Statement Coverage?

It is a white box test design technique which involves execution of all the executable statements in the source code at least once.

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$$

The goal of Statement coverage is to cover all the possible statement in the code. Unless a statement is executed, we have no way of knowing if an error exists in that statement.

Example of Statement Coverage



```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

If we want to cover every statement of this code, then the following test cases must be designed:

Test case 1: x = y = n, where n is any number

Test case 2: x = n, y = n', where n and n' are different numbers.

Here, Test case 1, skips the loop, Test case 2, the loop is executed. However, every statement inside the loop might not execute.

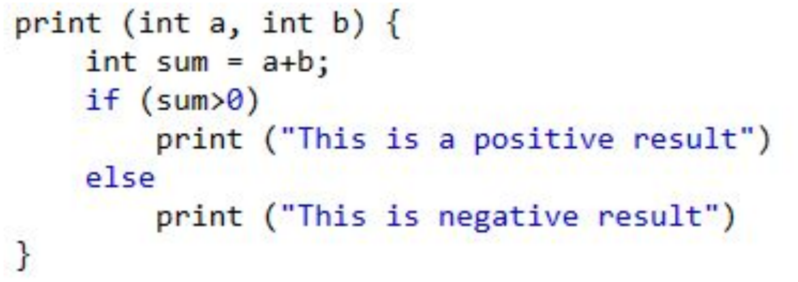
So, two more cases need to be designed to cover all statements: designed:

Test case 3: x > y

Test case 4: x < y

Test case 3 and 4 are sufficient to execute all the statements in the code. But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected. Statement coverage is a necessary but not a sufficient criteria for logic coverage.

Example of Statement Coverage



```
print (int a, int b) {
    int sum = a+b;
    if (sum>0)
        print ("This is a positive result")
    else
        print ("This is negative result")
}
```

Take input of two values of a and b

Find the sum of these two values.

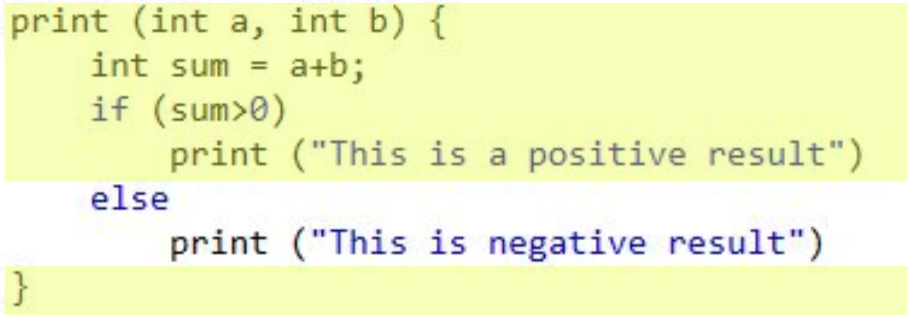
If the sum is greater than 0, then print "This is the positive result."

If the sum is less than 0, then print "This is the negative result."

Now, let's see the two different scenarios and calculation of the percentage of Statement Coverage for given source code.

Example of Statement Coverage

Scenario 1:
If a = 3, b = 9,



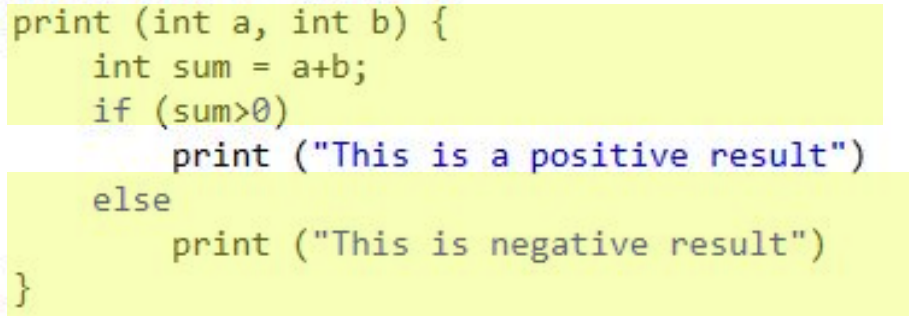
```
print (int a, int b) {
    int sum = a+b;
    if (sum>0)
        print ("This is a positive result")
    else
        print ("This is negative result")
}
```

The statements marked in yellow color are those which are executed as per the scenario.

Number of executed statements = 5, Total number of statements = 7
Statement Coverage: 5/7 = 71%

Example of Statement Coverage

Scenario 1:
If a = -3, b = -9,



```
print (int a, int b) {
    int sum = a+b;
    if (sum>0)
        print ("This is a positive result")
    else
        print ("This is negative result")
}
```

The statements marked in yellow color are those which are executed as per the scenario.

Number of executed statements = 6, Total number of statements = 7
Statement Coverage: 6/7 = 85.2%

Example of Statement Coverage

But overall if we see, all the statements are being covered by these 2 scenario’s.

So we can conclude that overall statement coverage is 100% if our test scenario consist of these two test cases.

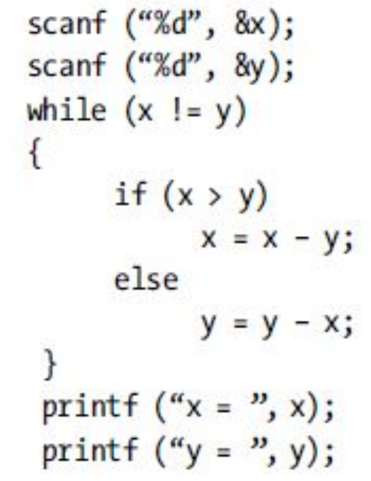
What is covered by Statement Coverage?

1. Unused Statements
2. Dead Code
3. Unused Branches
4. Missing Statements

Branch/Decision Coverage

- “Branch” in a programming language is like the “IF statements”. An IF statement has two branches: True and False.
- So in Branch coverage (also called Decision coverage), we validate whether each branch is executed at least once.
- In case of an “IF statement”, there will be two test conditions:
  - One to validate the true branch and,
  - Other to validate the false branch.
- Hence, in theory, Branch Coverage is a testing method which is when executed ensures that each and every branch from each decision point is executed.

Branch Coverage



```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

while and *if* statements have two outcomes: True and False. So test cases must be designed such that both outcomes for *while* and *if* statements are tested

Test case 1: x = y
Test case 2: x != y
Test case 3: x < y
Test case 4: x > y

Condition Coverage

Condition coverage states that each condition in a decision takes on all possible outcomes at least once.

Consider the following statement:

while ((I ≤5) && (J < COUNT))

In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes.

The following test cases are designed:

Test case 1: I ≤ 5, J < COUNT

Test case 2: I > 5, J > COUNT

Condition Coverage

Condition coverage in a decision does not mean that the decision has been covered. If the decision

if (A && B)

is being tested, the condition coverage would allow one to write two test cases:

Test case 1: A is True, B is False.

Test case 2: A is False, B is True.

But these test cases would not cause the THEN clause of the IF to execute .

Condition Coverage

Multiple Condition Coverage:
requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once.

Example:

Test case 1: A = True, B = True

Test case 2: A = True, B = False

Test case 3: A = False, B = True

Test case 4: A = False, B = False

Basic Path Testing

- ❖ Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing..
- ❖ It is a more general criterion as compared to other coverage criteria and useful for detecting more errors.
- ❖ The objective of basis path testing is to define the number of independent paths, so the number of test cases needed can be defined explicitly to maximize test coverage.
- ❖ Problem: programs that contain loops may have an infinite number of possible paths and it is not practical to test all the paths.

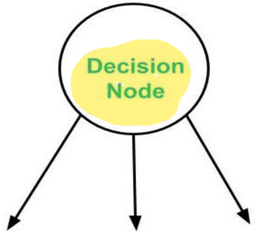
Basic Path Testing

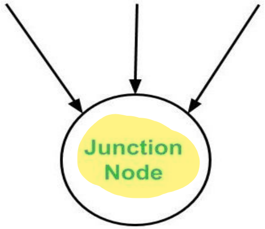
Steps for Basis Path testing

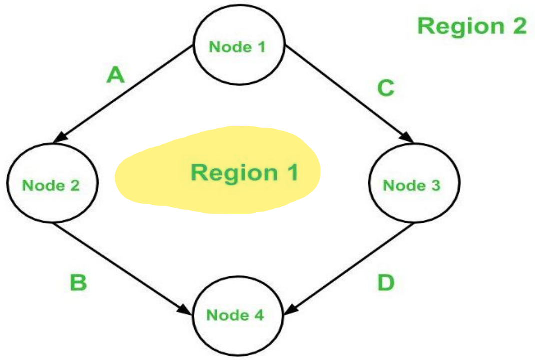
- ❖ Draw a **control flow graph** (to determine different program paths)
- ❖ Calculate **Cyclomatic complexity** (metrics to determine the number of independent paths)
- ❖ Find a basis set of paths (**independent path**)
- ❖ Generate test cases to exercise each path

Basic Path Testing: Control Flow Graph

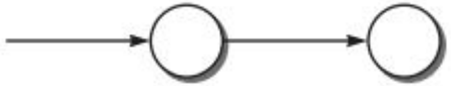
- A control flow graph (or simply, flow graph) is a directed graph which represents the control structure of a program or module.
- A control flow graph (V, E) has V number of **nodes/vertices, procedural statement** and E number of **edges, flow of control** in it. A control graph can also have :
    - **Junction Node** – a node with more than one arrow entering it.
    - **Decision Node** – a node with more than one arrow leaving it.
    - **Region** – area bounded by edges and nodes (area outside the graph is also counted as a region.).

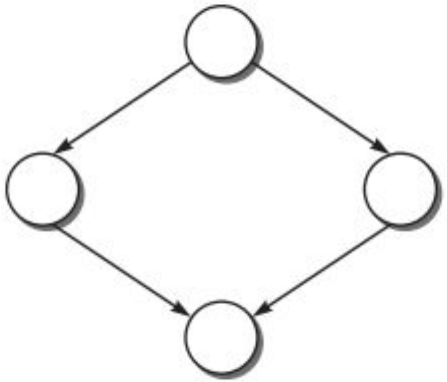


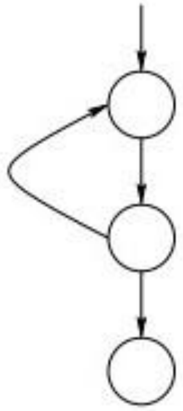


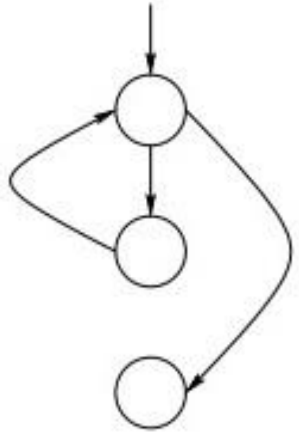


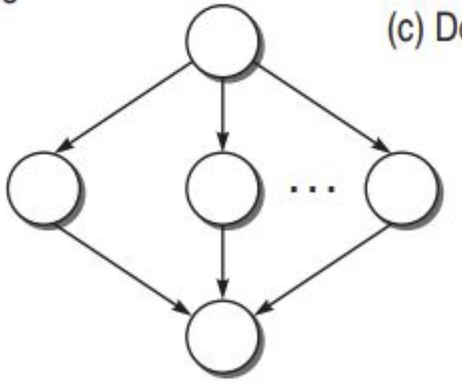
Flow Graph Notations


(a) Sequence


(b) If-Then-Else


(c) Do-While


(d) While-Do


(e) Switch-Case

Basic Path Testing: Cyclomatic Complexity

❖ The cyclomatic complexity V(G) is said to be a measure of the logical complexity of a program. It can be calculated using three different formulae :

Formula based on edges and nodes

V(G) = e - n + 2*P

Where,
e is number of edges,
n is number of vertices,
P is number of connected components.

Formula based on Decision Nodes

V(G) = d + P

where,
d is number of decision nodes,
P is number of connected nodes.

Formula based on Regions

V(G) = # of regions (R) in the graph

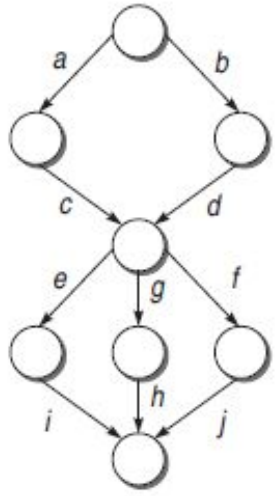
Calculating the number of decision nodes for Switch-Case/Multiple If-Else

If a decision node has exactly two arrows leaving it, then it is counted as one decision node.
However, if there are more than 2 arrows leaving a decision node, it is computed :

d = k – 1, where k is the number of arrows leaving the node.

Basic Path Testing: Independent Path

- An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined
- cyclomatic complexity gives the number of independent paths present in a flow graph.



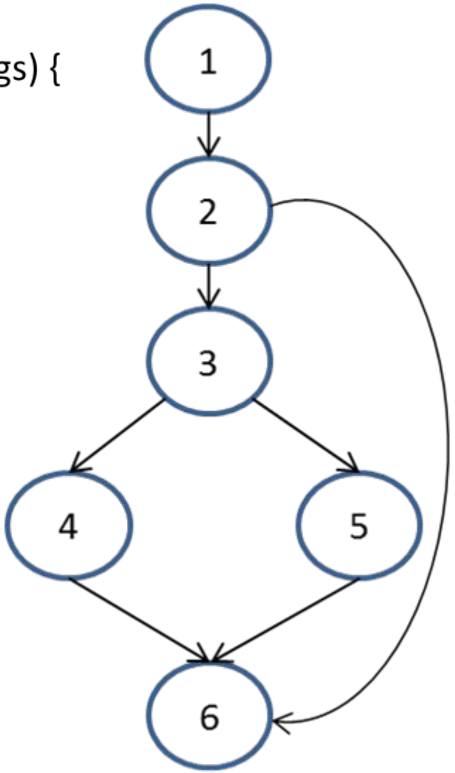
In the graph, there are six possible paths: *acei, acgh, acfj, bdei, bdgh, bdfj.*

In this case, of the **six** possible paths, only **four** are independent, as the other two are always a linear combination of the other four paths.

If we calculate the cyclomatic complexity, it will be same.

Basic Path Testing: Example

```
public class gcd {
  public static void main(String[] args) {
    float x = Float.valueOf(args[0]);
    float y = Float.valueOf(args[1]);
    while (x != y) {
      if (x > y)
        x = x - y;
      else y = y - x;
    }
    System.out.println("GCD: " + x);
  }
}
```

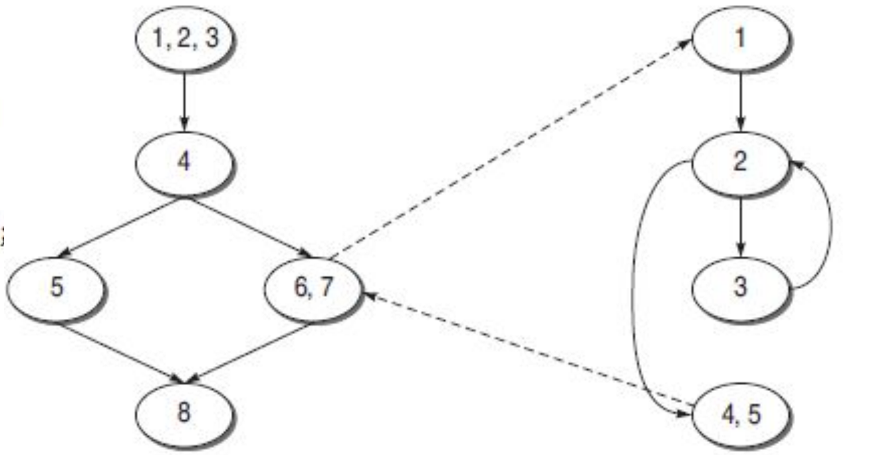


Here in this example
e=7 n=6 d=2 R=3

Now Calculate the cyclomatic complexity

Basic Path Testing: Example

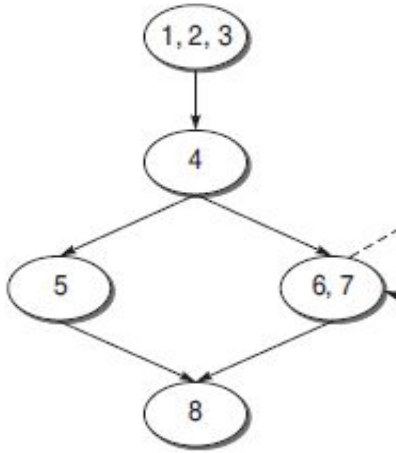
```
        main()
        {
            int number;
            int fact();
1.          clrscr();
2.          printf(“Enter the number whose factorial is to be found out”);
3.          scanf(“%d”, &number);
4.          if(number <0)
5.              printf(“Facorial cannot be defined for this number);
6.          else
7.              printf(“Factorial is %d”, fact(number));
8.      }

        int fact(int number)
        {
            int index;
1.          int product =1;
2.          for(index=1; index<=number; index++)
3.              product = product * index;
4.          return(product);
5.      }
```



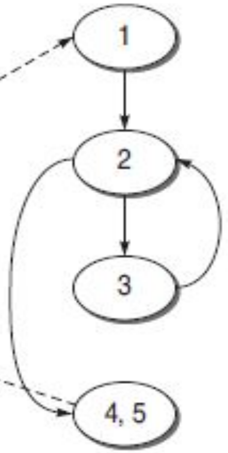
(a) Flow graph for main ()

(b) Flow graph for fact ()

Basic Path Testing: Example



(a) Flow graph for main ()



(b) Flow graph for fact ()

Cyclomatic complexity of main()

(a) $V(M) = e - n + 2p = 5 - 5 + 2$
$= 2$

(b) $V(M) = d + p = 1 + 1$
$= 2$

(c) $V(M)$ = Number of regions = 2

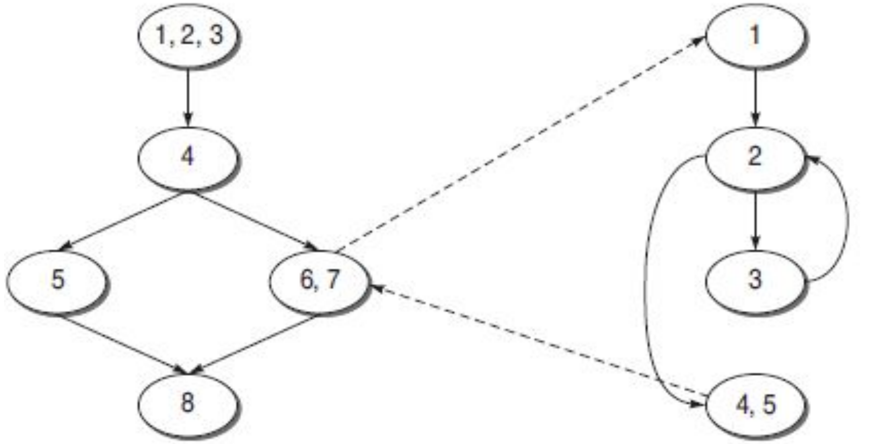
Cyclomatic complexity of fact()

(a) $V(R) = e - n + 2p = 4 - 4 + 2$
$= 2$

(b) $V(R)$ = Number of predicate nodes + 1
$= 1 + 1$
$= 2$

(c) $V(R)$ = Number of regions = 2

Basic Path Testing: Example



(a) Flow graph for main () (b) Flow graph for fact ()

Cyclomatic complexity of the whole graph considering the full program

(a) $V(G) = e - n + 2p$
$= 9 - 9 + 2 * 2$
$= 4$
$= V(M) + V(R)$

(b) $V(G) = d + p$
$= 2 + 2$
$= 4$
$= V(M) + V(R)$

(c) $V(G)$ = Number of regions
$= 4$
$= V(M) + V(R)$

Basic Path Testing: More Example

Page 151 ~163

SOFTWARE TESTING Principles and Practices

By Naresh Chauhan

Basic Path Testing: Application

More Coverage –basis path set provides us the number of test cases to be covered which determines the number of test cases that must be executed for full coverage..

Maintenance Testing – When a software is modified, it is still necessary to test the changes made in the software which as a result, requires path testing.

Unit Testing – When a developer writes the code, he or she tests the structure of the program or module themselves first.

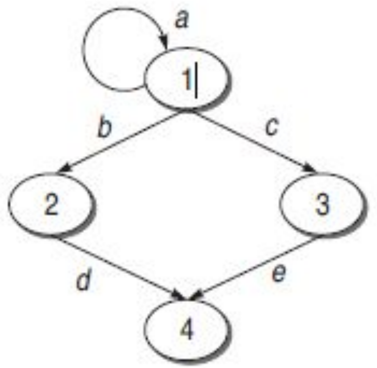
Integration Testing – When one module calls other modules, there are high chances of Interface errors. In order to avoid the case of such errors, path testing is performed to test all the paths on the interfaces of the modules.

Testing Effort – Since the basis path testing technique takes into account the complexity of the software (i.e., program or module), therefore it is intuitive to note that testing effort in case of basis path testing is directly proportional to the complexity of the software or program.

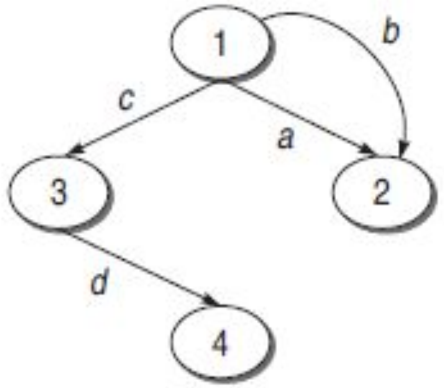
Graph Matrix

- A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph
- The following points describe a graph matrix:
    - Each cell in the matrix can be a direct connection or link between one node to another node
    - If there is a connection from node ‘a’ to node ‘b’, then it does not mean that there is connection from node ‘b’ to node ‘a’.
    - Conventionally, to represent a graph matrix, digits are used for nodes and letter symbols for edges or connections.

Graph Matrix: Example



	1	2	3	4
1	a	b	c	
2				d
3				e
4				



	1	2	3	4
1		a+b	c	
2				
3				d
4				

Graph Matrix

Use of Graph Matrix to find set of all paths

- ❖ There may be of interest in path tracing to find *k-link* paths from one node. For example, how many 2-link paths are there from one node to another node?
- ❖ The power operation on matrix expresses the relation between each pair of nodes via intermediate nodes
- ❖ For example, the square of matrix represents path segments that are 2-links long. Similarly, the cube power of matrix represents path segments that are 3-links long.

Graph Matrix

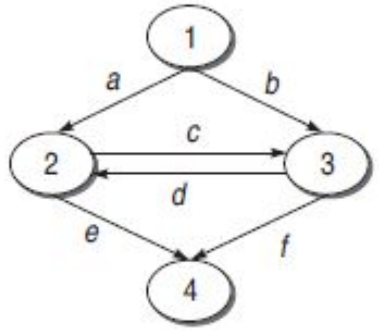
Use of Graph Matrix to find set of all paths:

- There may be of interest in path tracing to find *k-link* paths from one node. For example, how many 2-link paths are there from one node to another node?
- The power operation on matrix expresses the relation between each pair of nodes via intermediate nodes
- For example, the square of matrix represents path segments that are 2-links long. Similarly, the cube power of matrix represents path segments that are 3-links long.

	1	2	3	4
1	a	b	c	
2				d
3				e
4				

$$\begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} a^2 & ab & ac & bd + ce \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

More Example



Derive its graph matrix and find 2-link and 3-link set of paths.

The graph matrix is :

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

2-link set of paths:

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

3-link set of paths:

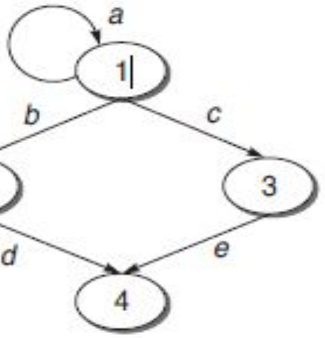
$$\begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & acd & bdc & bde+acf \\ 0 & 0 & cdc & cde \\ 0 & dcd & 0 & dcf \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

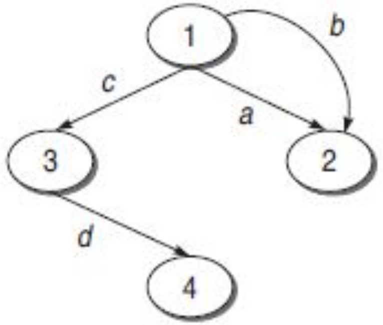
Connection Matrix

- ❖ Previously the matrix is just a tabular representation and does not provide any useful information.
- ❖ A matrix defined with link weights is called a connection matrix.
- ❖ In the simplest form, when the connection exists, then the link weight is 1, otherwise 0 (But 0 is not entered in the cell entry of matrix to reduce the complexity).

Connection Matrix

Example:





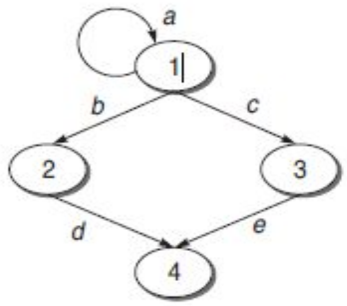
	1	2	3	4
1	1	1	1	
2				1
3				1
4				

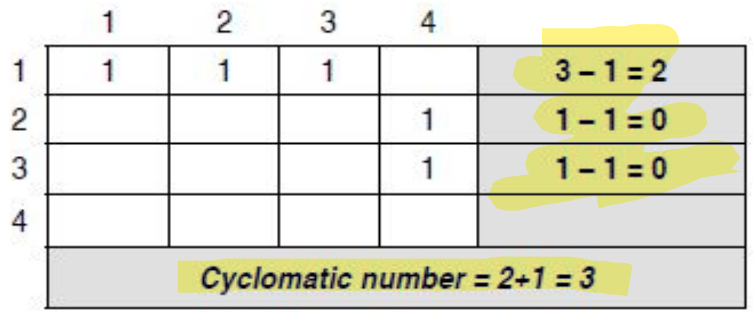
	1	2	3	4
1		1	1	
2				
3				1
4				

Connection Matrix

Use of Connection Matrix to find set of all paths:

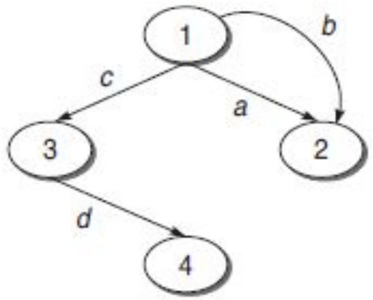
- ❖ Connection matrix is used to see the control flow of the program
- ❖ It is used to find the cyclomatic complexity number of the flow graph.
- ❖ procedure to find the cyclomatic number from the connection matrix:
  - ➢ Step 1: For each row, count the number of 1s and write it in front of that row.
  - ➢ Step 2: Subtract 1 from that count. Ignore the blank rows, if any.
  - ➢ Step 3: Add the final count of each row.
  - ➢ Step 4: Add 1 to the sum calculated in Step 3.
  - ➢ Step 5: The final sum in Step 4 is the cyclomatic number of the graph.





	1	2	3	4	
1	1	1	1		3 – 1 = 2
2				1	1 – 1 = 0
3				1	1 – 1 = 0
4					
	Cyclomatic number = 2+1 = 3				

Example



	1	2	3	4	
1		1	1		**2 – 1 = 1**
2					
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 1+1 = 2				

Loop Testing

- ❖ Loop Testing is a type of software testing type that is performed to validate the loops.
- ❖ Loop testing can be viewed as an extension to branch coverage.
- ❖ If loops are not tested properly, bugs can go undetected.
- ❖ Sufficient test cases should be designed to test every loop thoroughly.

Objectives of Loop Testing:

- ❖ To fix the infinite loop repetition problem.
- ❖ To know the performance.
- ❖ To identify the loop initialization problems.
- ❖ To determine the uninitialized variables.

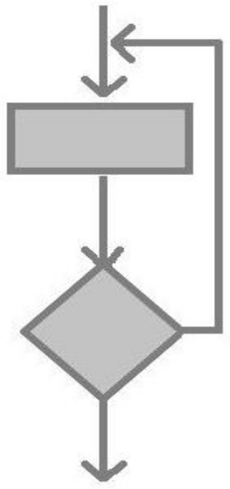
Loop Testing

- **Types of Loop:**
  - Simple Loop
  - Nested Loops
  - Concatenated Loops
  - Unstructured Loops

Loop Testing: Simple Loop

Simple loop is basically a normal “for”, “while” or “do-while” in which a condition is given and loop runs and terminates according to true and false occurrence of the condition respectively.

Example:

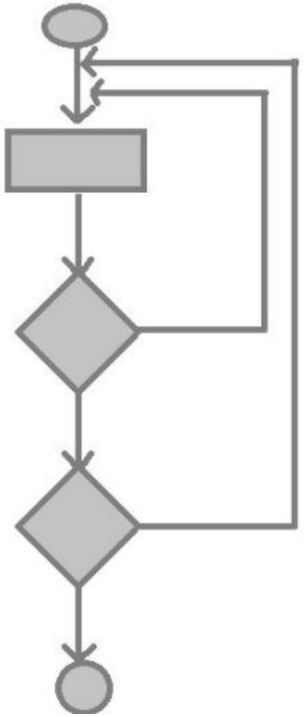


A simple loop is tested in the following way:

1. Skip the entire loop.
2. Make 1 pass through the loop.
3. Make 2 passes through the loop.
4. Make x passes through the loop where x<y, y is the maximum number of passes through the loop.
5. Make "y","y-1","y+1" passes through the loop where "y" is the maximum number of allowable passes through the loop.

Loop Testing: Nested Loop

Nested loop is basically one loop inside the another loop.
Example:



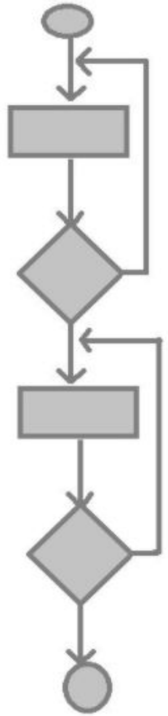
A nested loop is tested in the following way:

1. A nested loop starts at the innermost loop.
2. For the innermost loop, conduct a simple loop test.
3. Work outward.
4. Continue until the outermost loop has been tested.

Loop Testing: Concatenated Loop

Concatenated loops are loops after the loop. It is a series of loops.

Example:



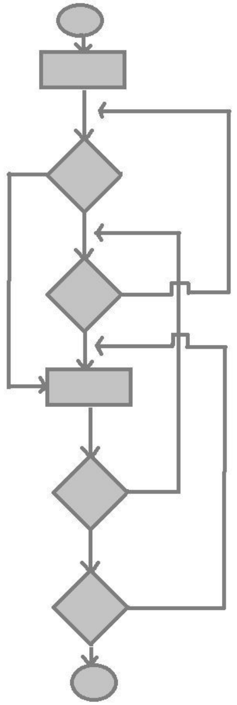
A Concatenated loop is tested in the following way:

1. If the loops are independent then test them as simple loops or else test them as nested loops.

Loop Testing: Unstructured Loop

Unstructured loop is the combination of nested and concatenated loops. It is basically a group of loops that are in no order.

Example:



A Unstructured loop is tested in the following way:

1. Don't test - redesign..

Data Flow Testing

- Data-flow testing uses the control flow graph to explore the unreasonable things that can happen to data (i.e., anomalies).
- Data-flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- E.g., Pick enough paths to assure that:
    - Every data object has been initialized prior to its use.
    - All defined objects have been used at least once.

State of Data Object

- (d) Defined, Created, Initialized
- (k) Killed, Undefined, Released
- (u) Used:
  - (c) Used in a calculation
  - (p) Used in a predicate

(d) Define Object

An object (e.g., variable) is defined when it:

- appears in a data declaration
- is assigned a new value
- is a file that has been opened
- is dynamically allocated

(u) Used Object

- ❖ An object is used when it is part of a computation or a predicate.
- ❖ A variable is used for a computation (c-use) when it appears on the RHS (sometimes even the LHS in case of array indices) of an assignment statement.
- ❖ A variable is used in a predicate (p-use) when it appears directly in that predicate.

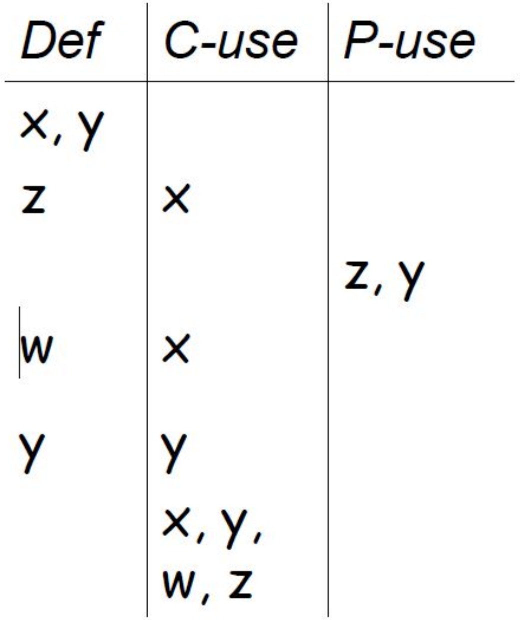
(k) Killed/Undefined

When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting the loop or memory is released dynamically or a file has been closed.

Example: Definition & Usage

What are the definitions and uses for the program below?

```
1. read (x, y);
2. z = x + 2;
3. if (z < y)
4 w = x + 1;
else
5. y = y + 1;
6. print (x, y, w, z);
```

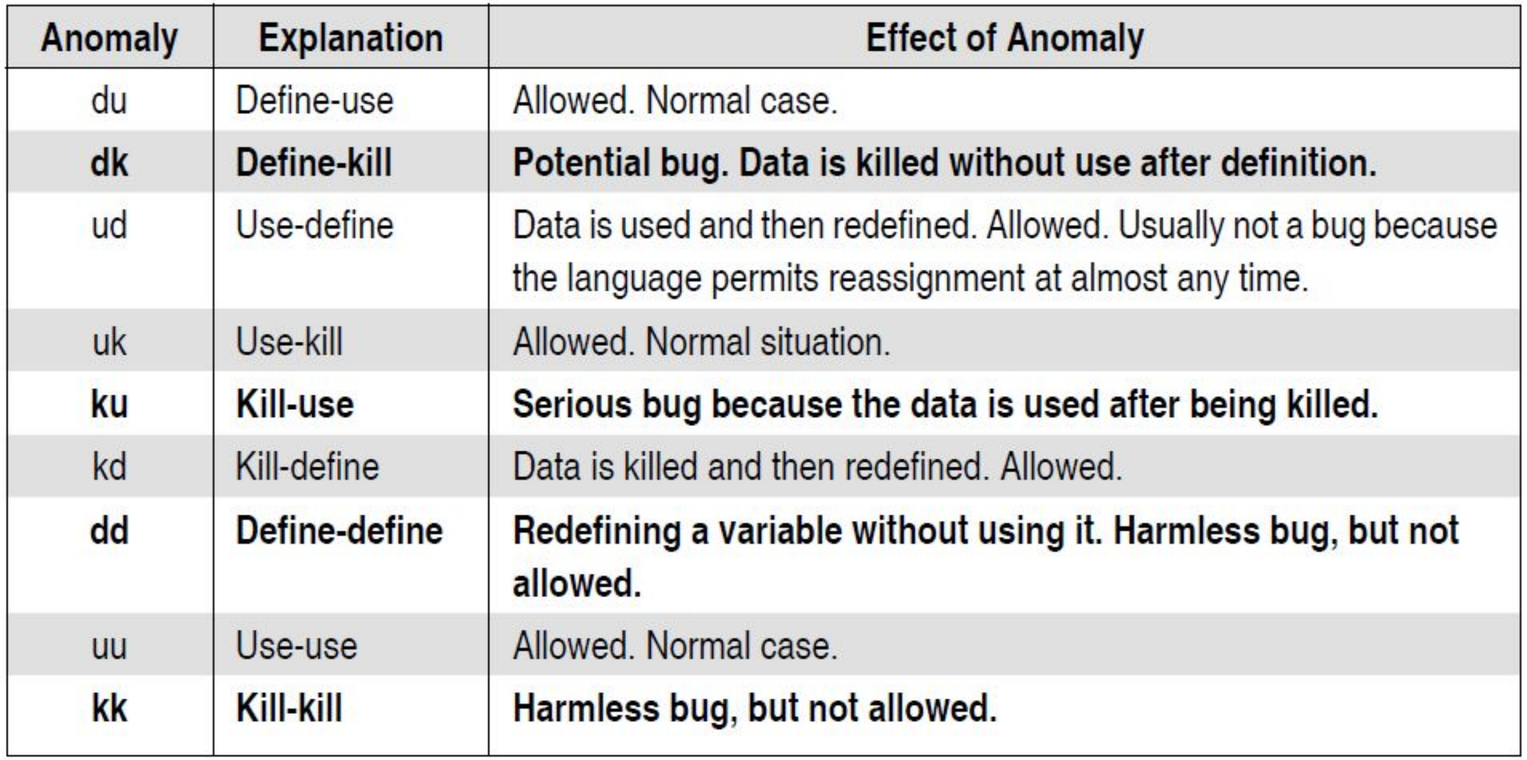


Def	C-use	P-use
x, y		
z	x	
		z, y
w	x	
y	y	
	x, y, w, z	

Data Flow Anomalies

- Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.
- An anomaly is denoted by a two-character sequence of actions.
- For example, ‘dk’ means a variable is defined and killed without any use, which is a potential bug.

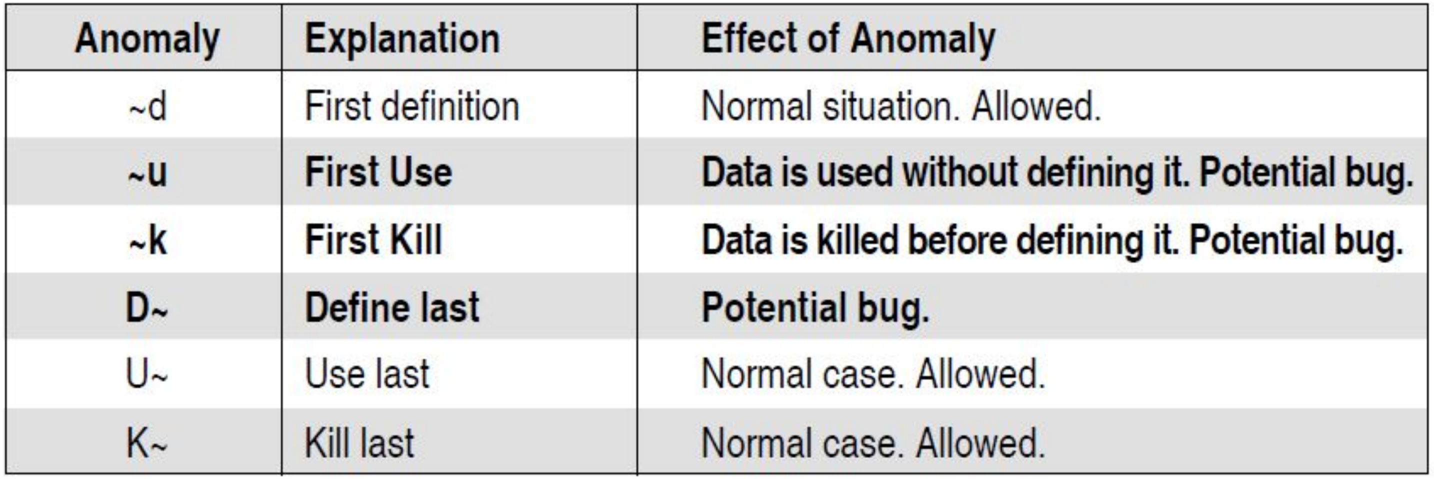
Data Flow Anomalies



Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	**Define-kill**	**Potential bug. Data is killed without use after definition.**
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	**Kill-use**	**Serious bug because the data is used after being killed.**
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	**Define-define**	**Redefining a variable without using it. Harmless bug, but not allowed.**
uu	Use-use	Allowed. Normal case.
kk	**Kill-kill**	**Harmless bug, but not allowed.**

Data Flow Anomalies

- ❖ In addition to the two-character data anomalies, there may be single-character data anomalies also.
- ❖ To represent these types of anomalies, we take the following conventions:
  - ■ ~x : indicates all prior actions are not of interest to x.
  - ■ x~ : indicates all post actions are not of interest to x.



Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	**First Use**	**Data is used without defining it. Potential bug.**
~k	**First Kill**	**Data is killed before defining it. Potential bug.**
D~	**Define last**	**Potential bug.**
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

Terminologies used in Data Flow Testing

Definition node: Defining a variable means assigning value to a variable for the very first time in a program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.

Usage node: Node n that belongs to G(P) is a usage node of variable v, if the value of variable v is used at the statement corresponding to node n. For example, output statements, assignment statements (right), conditional statements, loop control statements, etc.

A usage node can be of the following two types:

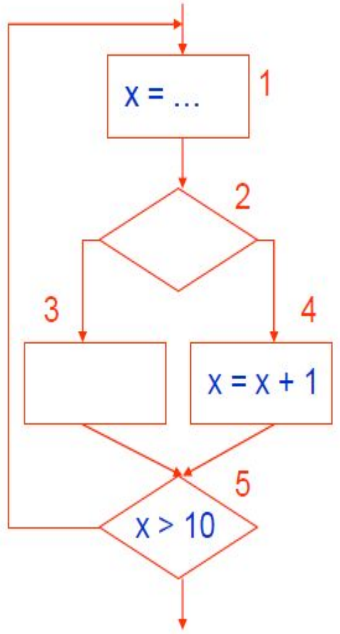
- **(i) Predicate Usage Node**: If usage node n is a predicate node, then n is a predicate usage node.
- **(ii) Computation Usage Node:** If usage node n corresponds to a computation statement in a program other than predicate, then it is called a computation usage node.

Terminologies used in Data Flow Testing

Definition-use path (du-path):
A du-path with respect to a variable v is a path between the definition node and the usage node of that variable. Usage node can either be a p-usage or a c-usage node.

Definition Clear Paths:

A path (i, n_1, n_2, ..., n_m, j) is a definition-clear path for a variable x from i to j if n_1 through n_m do not contain a definition of x.

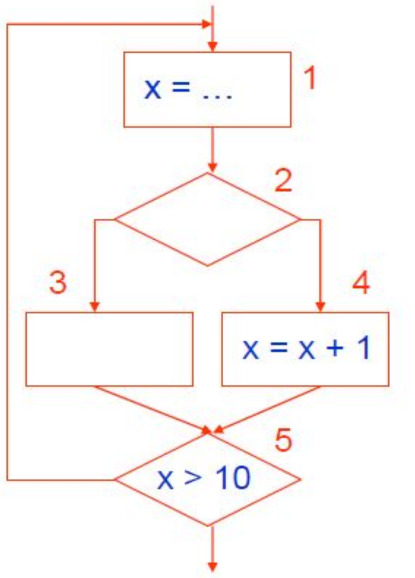


Terminologies used in Data Flow Testing

Definition-C-Use Associations:

Given a definition of x in node n_d and a c-use of x in node $n_{c\text{-use}}$, the presence of a definition-clear path for x from n_d to $n_{c\text{-use}}$ establishes the definition-c-use association (n_d, $n_{c\text{-use}}$, x).

(1,4, x)

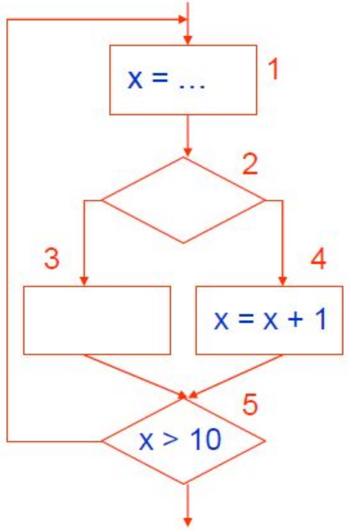


Terminologies used in Data Flow Testing

Definition-P-Use Associations:

Given a definition of x in node n_d and a p-use of x in node $n_{p\text{-}use}$, the presence of a definition-clear path for x from n_d to $n_{p\text{-}use}$ establishes the definition-p-use associations $(n_d, (n_{p\text{-}use}, t), x)$ and $(n_d, (n_{p\text{-}use}, f), x)$.

(1,(5, t), x)
(1,(5, f), x)



Dynamic Vs Static Anomaly Detection

Static Analysis is analysis done on source code without actually executing it.

- E.g., Syntax errors are caught by static analysis.

Dynamic Analysis is analysis done as a program is executing and is based on intermediate values that result from the program's execution.

- E.g., A division by 0 error is caught by dynamic analysis.

If a data-flow anomaly can be detected by static analysis then the anomaly does not concern testing. (Should be handled by the compiler.)

Anomaly Detection Using Compilers

Compilers are able to detect several data-flow anomalies using static analysis.

e.g., By forcing declaration before use, a compiler can detect anomalies such as:

-u

ku

Optimizing compilers are able to detect some dead variables.

Questions:

- Why isn’t static analysis enough?
- Why is testing required?
- Could a good compiler detect all data flow anomalies?

Answer: No. Detecting all data-flow anomalies is provably unsolvable.

Static Analysis Deficiencies

Current static analysis methods are inadequate for:

Dead Variables: Detecting unreachable variables is unsolvable in the general case.

Arrays: Dynamically allocated arrays contain garbage unless they are initialized explicitly. (-**u** anomalies are possible)

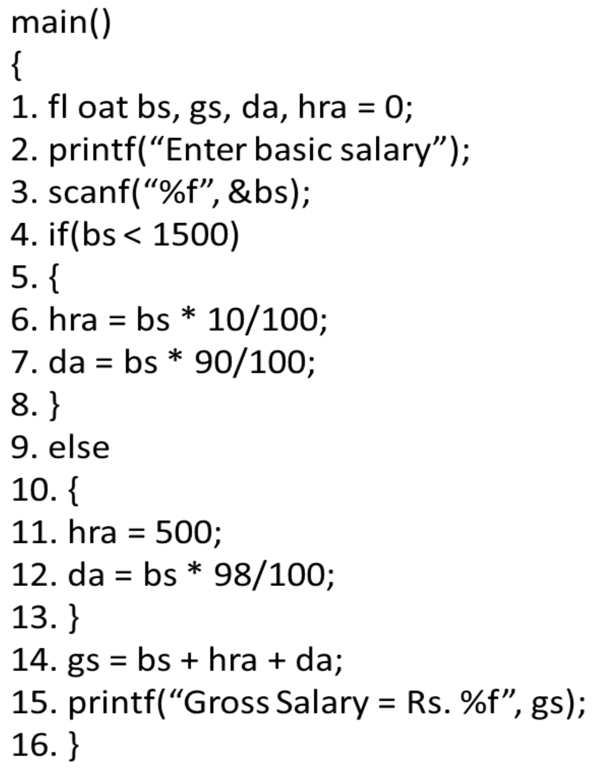
Pointers: Impossible to verify pointer values at compile time.

False Anomalies: Even an obvious bug (*e.g.*, **ku**) may not be a bug if the path along which the anomaly exists is unachievable. (Determining whether a path is or is not achievable is unsolvable.)

Example Static Example

Consider the program for calculating the gross salary of an employee in an organization. If his basic salary is less than Rs 1500, then HRA = 10% of basic salary and DA = 90% of the basic. If his salary is either equal to or above Rs 1500, then HRA = Rs 500 and DA = 98% of the basic salary.

Calculate his gross salary.



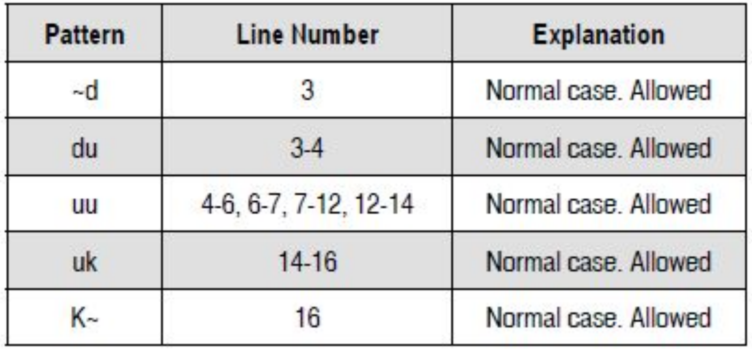
```
main()
{
1. fl oat bs, gs, da, hra = 0;
2. printf("Enter basic salary");
3. scanf("%f", &bs);
4. if(bs < 1500)
5. {
6. hra = bs * 10/100;
7. da = bs * 90/100;
8. }
9. else
10. {
11. hra = 500;
12. da = bs * 98/100;
13. }
14. gs = bs + hra + da;
15. printf("Gross Salary = Rs. %f", gs);
16. }
```

Find out the define-use-kill patterns for all the variables in the source code of this application.

Example(Continue)

```
main()
{
1. fl oat bs, gs, da, hra = 0;
2. printf("Enter basic salary");
3. scanf("%f", &bs);
4. if(bs < 1500)
5. {
6. hra = bs * 10/100;
7. da = bs * 90/100;
8. }
9. else
10. {
11. hra = 500;
12. da = bs * 98/100;
13. }
14. gs = bs + hra + da;
15. printf("Gross Salary = Rs. %f", gs);
16. }
```

For variable 'bs', the define-use-kill patterns are given below.

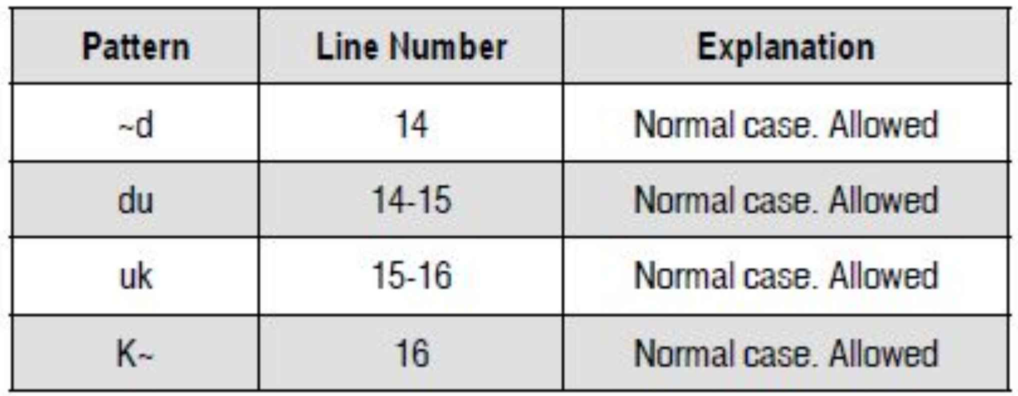


Pattern	Line Number	Explanation
~d	3	Normal case. Allowed
du	3-4	Normal case. Allowed
uu	4-6, 6-7, 7-12, 12-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

Example(Continue)

```
main()
{
1. fl oat bs, gs, da, hra = 0;
2. printf("Enter basic salary");
3. scanf("%f", &bs);
4. if(bs < 1500)
5. {
6. hra = bs * 10/100;
7. da = bs * 90/100;
8. }
9. else
10. {
11. hra = 500;
12. da = bs * 98/100;
13. }
14. gs = bs + hra + da;
15. printf("Gross Salary = Rs. %f", gs);
16. }
```

For variable 'gs', the define-use-kill patterns are given below.

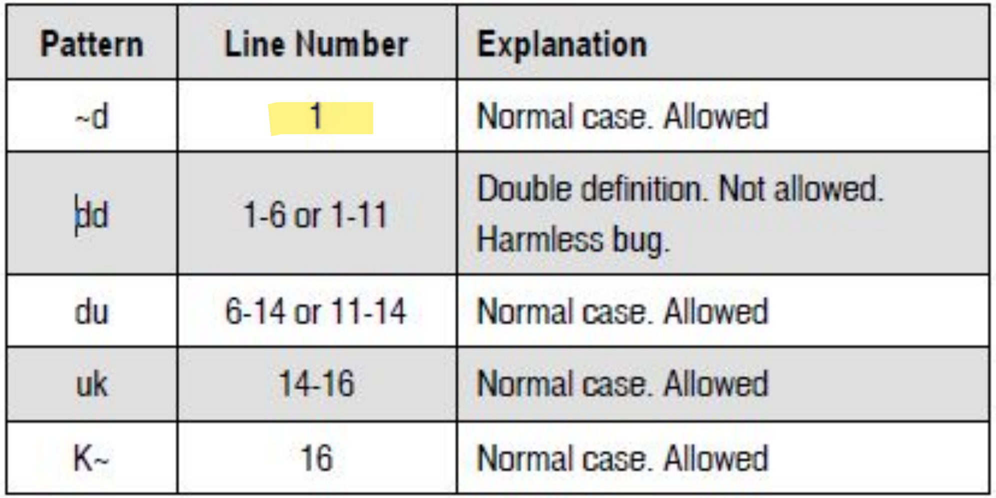


Pattern	Line Number	Explanation
~d	14	Normal case. Allowed
du	14-15	Normal case. Allowed
uk	15-16	Normal case. Allowed
K~	16	Normal case. Allowed

Example(Continue)

```
main()
{
1. fl oat bs, gs, da, hra = 0;
2. printf("Enter basic salary");
3. scanf("%f", &bs);
4. if(bs < 1500)
5. {
6. hra = bs * 10/100;
7. da = bs * 90/100;
8. }
9. else
10. {
11. hra = 500;
12. da = bs * 98/100;
13. }
14. gs = bs + hra + da;
15. printf("Gross Salary = Rs. %f", gs);
16. }
```

For variable 'hra', the define-use-kill patterns are given below.

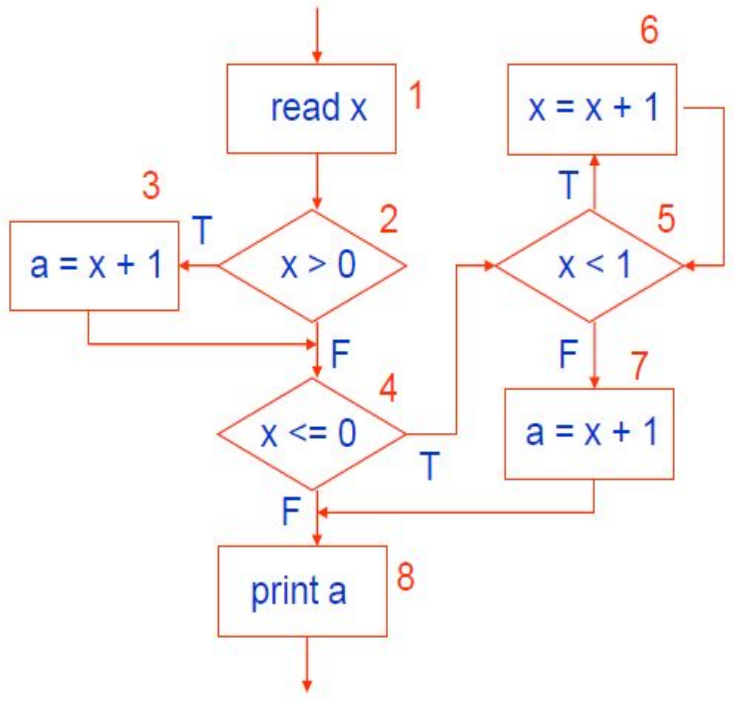


Pattern	Line Number	Explanation
~d	1	Normal case. Allowed
dd	1-6 or 1-11	Double definition. Not allowed. Harmless bug.
du	6-14 or 11-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

Dynamic Data Flow Testing

- ❖ Dynamic data flow testing is performed with the intention to uncover possible bugs in data usage during the execution of the code.
- ❖ Various strategies are employed for the creation of test cases.
  - ➢ All-defs coverage (AD)
  - ➢ All-c-uses coverage (ACU)
  - ➢ All-c-uses/some-p-uses coverage (ACU+ P)
  - ➢ All-p-uses coverage (APU)
  - ➢ All-p-uses/some-c-uses coverage (APU +C)
  - ➢ All-uses coverage (AU)
  - ➢ All-du-paths coverage (ADUP)

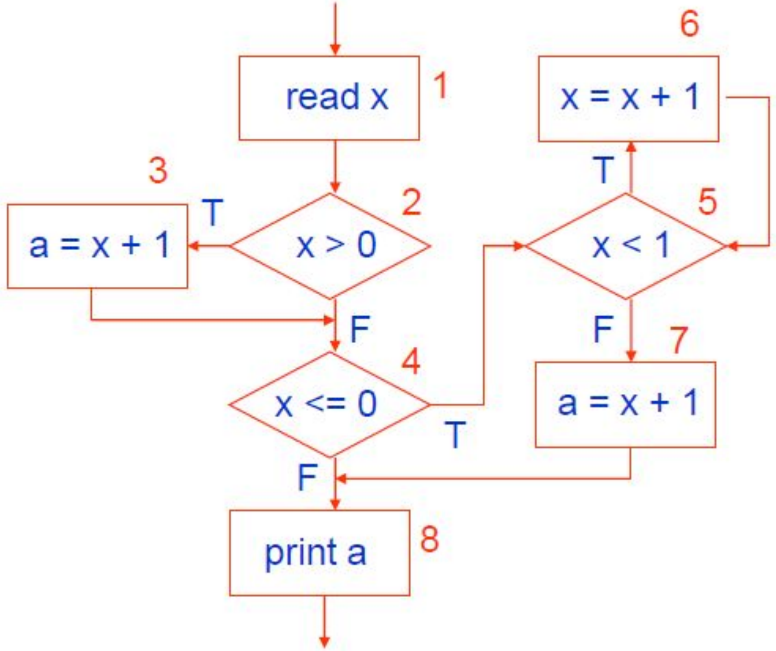
Example



x = 1
P1: (1, 2, 3, 4, 8)
a = 2

x = -1
P2: (1, 2, 4, 5, 6,5, 6, 5, 7, 8)
a = 2

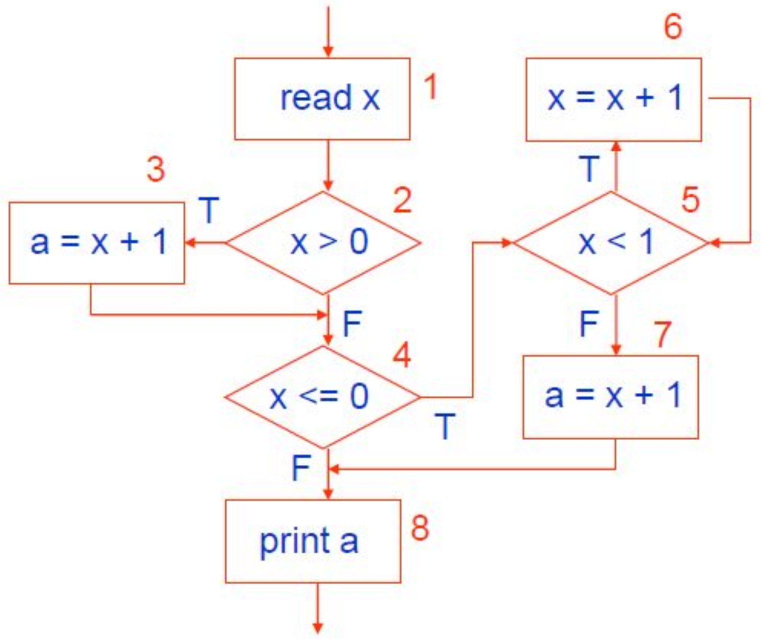
Example



Associations:
(1, (2, *t*), x)
(1, (2, *f*), x)
(1, 3, x)
(1, (4, *t*), x)
(1, (4, *f*), x)
(1, (5, *t*), x)
(1, (5, *f*), x)
(1, 6, x)
(1, 7, x)
(3, 8, a)
(6, 6, x)
(6, 7, x)
(6, (5, *t*), x)
(6, (5, *f*), x)
(7, 8, a)

Example

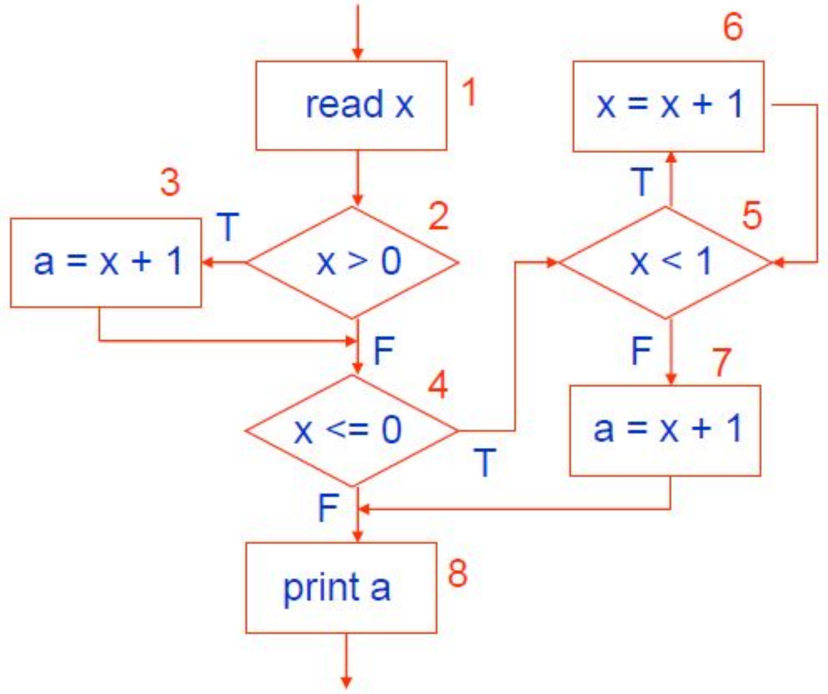
All-Def (AD): Test cases include a definition-clear path from every definition to some corresponding use (c-use or p-use).



Associations:	all-defs
(1, (2, *t*), x)	√
(1, (2, *f*), x)	
(1, 3, x)	
(1, (4, *t*), x)	
(1, (4, *f*), x)	
(1, (5, *t*), x)	
(1, (5, *f*), x)	
(1, 6, x)	
(1, 7, x)	
(3, 8, a)	√
(6, 6, x)	√
(6, 7, x)	
(6, (5, *t*), x)	
(6, (5, *f*), x)	
(7, 8, a)	√

Example

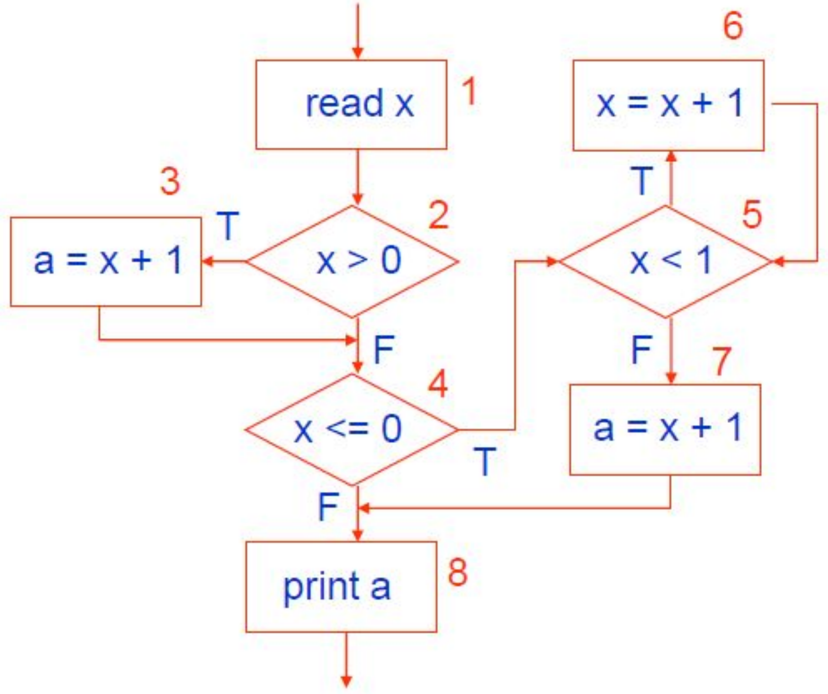
All-C-Uses(ACU): Test cases include a definition-clear path from every definition to all of its corresponding c-uses.



Associations:	all-c-uses
(1, (2, *t*), x)	
(1, (2, *f*), x)	
(1, 3, x)	√
(1, (4, *t*), x)	
(1, (4, *f*), x)	
(1, (5, *t*), x)	
(1, (5, *f*), x)	
(1, 6, x)	√
(1, 7, x)	√
(3, 8, a)	√
(6, 6, x)	√
(6, 7, x)	√
(6, (5, *t*), x)	
(6, (5, *f*), x)	
(7, 8, a)	√

Example

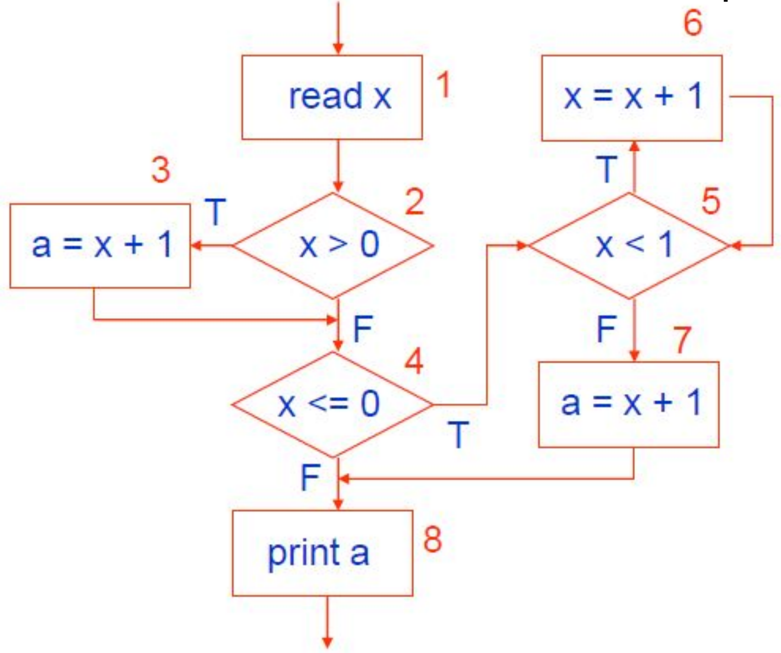
All-P-Uses(APU): Test cases include a definition-clear path from every definition to all of its corresponding p-uses.



Associations:	APU
(1, (2, *t*), x)	√
(1, (2, *f*), x)	√
(1, 3, x)	
(1, (4, *t*), x)	√
(1, (4, *f*), x)	√
(1, (5, *t*), x)	√
(1, (5, *f*), x)	√
(1, 6, x)	
(1, 7, x)	
(3, 8, a)	
(6, 6, x)	
(6, 7, x)	
(6, (5, *t*), x)	√
(6, (5, *f*), x)	√
(7, 8, a)	

Example

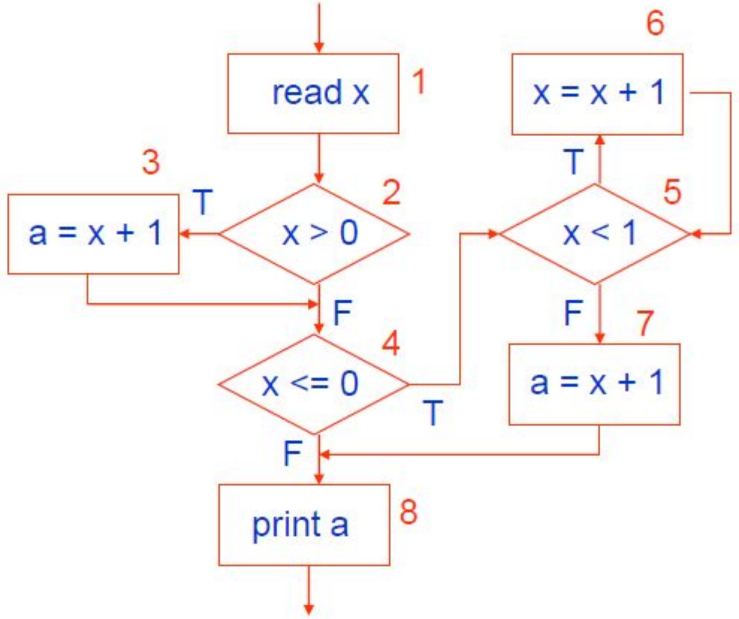
All-C-Uses/Some-P-Uses(ACU+ P): Test cases include a definition-clear path from every definition to all of its corresponding c-uses. In addition, if a definition has no c-use, then test cases include a definition-clear path to some p-use.



Associations:	ACU+ P
(1, (2, *t*), x)	
(1, (2, *f*), x)	
(1, 3, x)	√
(1, (4, *t*), x)	
(1, (4, *f*), x)	
(1, (5, *t*), x)	
(1, (5, *f*), x)	
(1, 6, x)	√
(1, 7, x)	√
(3, 8, a)	√
(6, 6, x)	√
(6, 7, x)	√
(6, (5, *t*), x)	
(6, (5, *f*), x)	
(7, 8, a)	√

Example

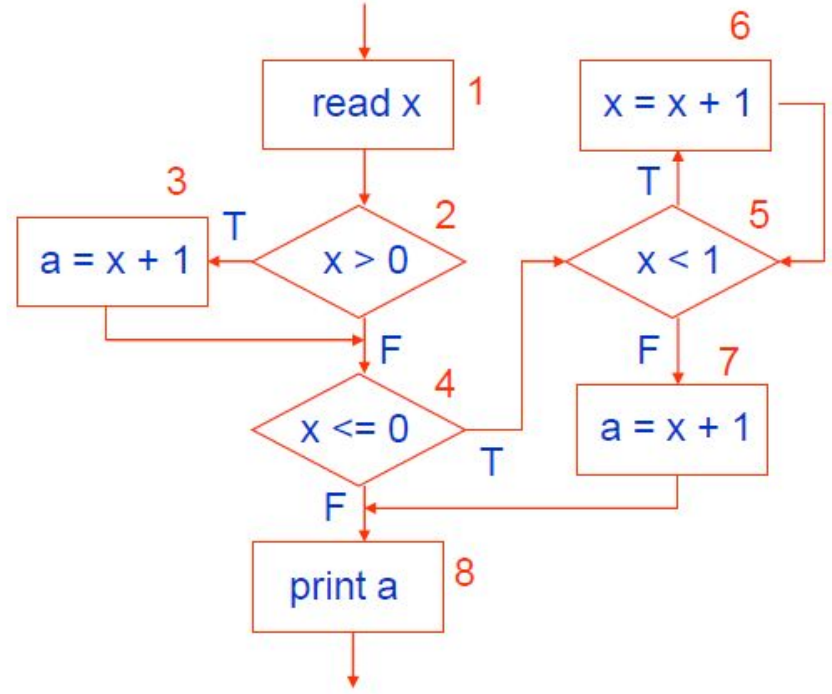
All-P-Uses/Some-C-Uses (APU+ C): Test cases include a definition-clear path from every definition to all of its corresponding p-uses. In addition, if a definition has no p-use, then test cases include a definition-clear path to some c-use.

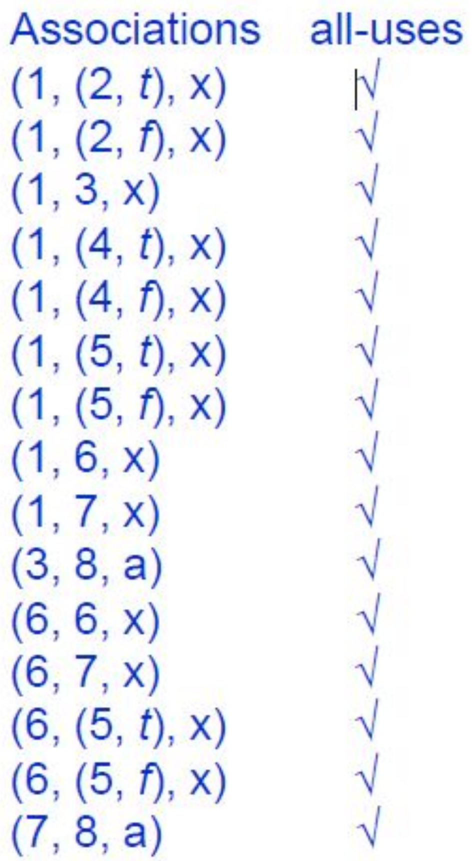


Associations:	APU+ C
(1, (2, *t*), x)	√
(1, (2, *f*), x)	√
(1, 3, x)	
(1, (4, *t*), x)	√
(1, (4, *f*), x)	√
(1, (5, *t*), x)	√
(1, (5, *f*), x)	√
(1, 6, x)	
(1, 7, x)	
(3, 8, a)	√
(6, 6, x)	
(6, 7, x)	
(6, (5, *t*), x)	√
(6, (5, *f*), x)	√
(7, 8, a)	√

Example

All-Uses (AU): Test cases include a definition-clear path from every definition to each of its uses including both c-uses and p-uses.

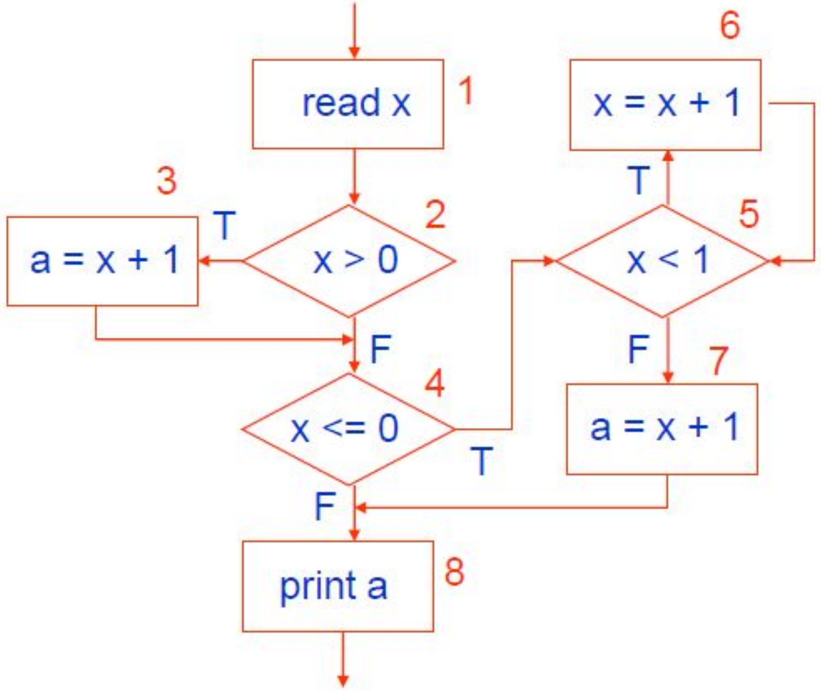




Associations	all-uses
(1, (2, *t*), x)	√
(1, (2, *f*), x)	√
(1, 3, x)	√
(1, (4, *t*), x)	√
(1, (4, *f*), x)	√
(1, (5, *t*), x)	√
(1, (5, *f*), x)	√
(1, 6, x)	√
(1, 7, x)	√
(3, 8, a)	√
(6, 6, x)	√
(6, 7, x)	√
(6, (5, *t*), x)	√
(6, (5, *f*), x)	√
(7, 8, a)	√

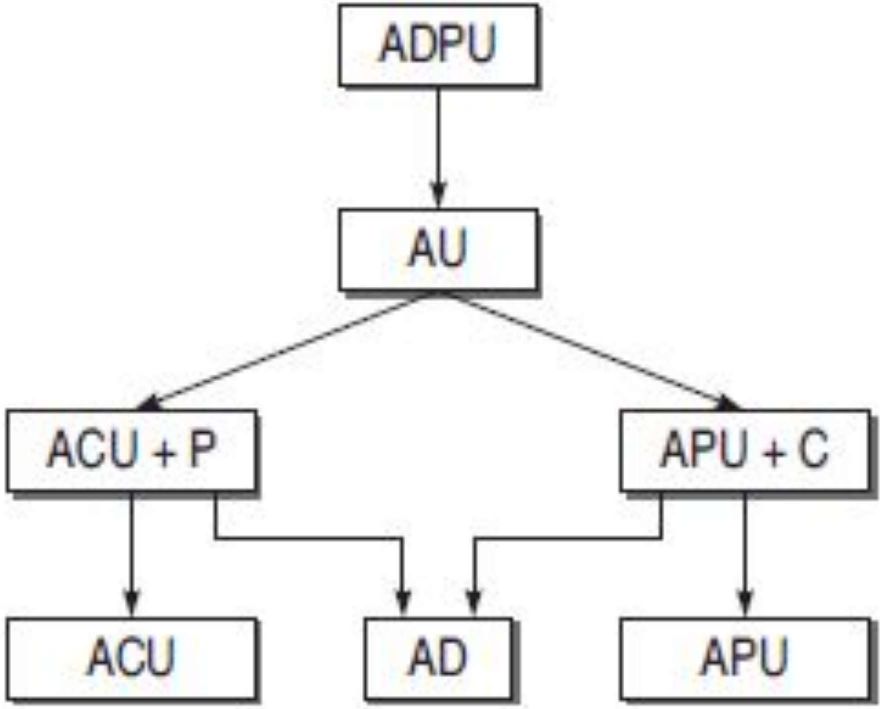
Example

All-DU-Paths(ADUP): Test cases include all du-paths for each definition. Therefore, if there are multiple paths between a given definition and a use, they must all be included.



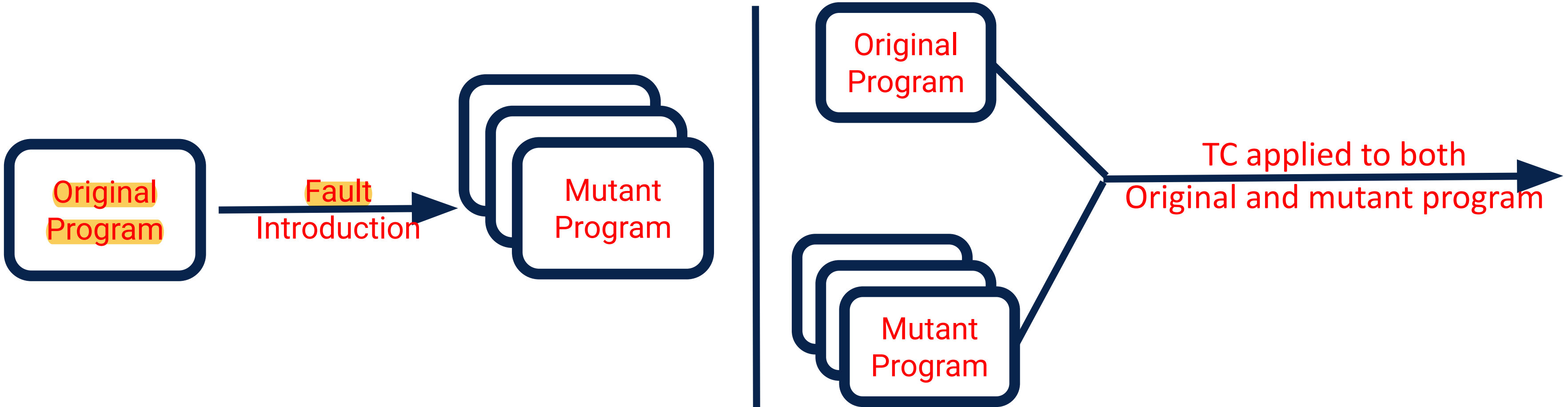
Associations	all-du-paths
(1, (2, *t*), x)	√
(1, (2, *f*), x)	√
(1, 3, x)	√
(1, (4, *t*), x)	√
(1, (4, *f*), x)	√
(1, (5, *t*), x)	√
(1, (5, *f*), x)	√
(1, 6, x)	√
(1, 7, x)	√
(3, 8, a)	√
(6, 6, x)	√
(6, 7, x)	√
(6, (5, *t*), x)	√
(6, (5, *f*), x)	√
(7, 8, a)	√

Ordering Of Data Flow Strategies



Mutation Testing

- It is a type of software testing in which certain statements of the source code are changed/mutated to check if the test cases are able to find errors in source code,
- The goal of Mutation Testing is to ensure the quality of test cases in terms of robustness that it should fail the mutated source code.
- The changes made in the mutant program should be kept extremely small that it does not affect the overall objective of the program.



Mutation Testing

Following are the steps to execute mutation testing(mutation analysis):

- ❖ **Step 1**: Faults are introduced into the source code of the program by creating many versions called mutants. Each mutant should contain a single fault, and the goal is to cause the mutant version to fail which demonstrates the effectiveness of the test cases.
- ❖ **Step 2**: Test cases are applied to the original program and also to the mutant program.
- ❖ **Step 3**: Compare the results of an original and mutant program.
- ❖ **Step 4**: If the original program and mutant programs generate the different output, then that the mutant is killed by the test case. Hence the test case is good enough to detect the change between the original and the mutant program.
- ❖ **Step 5**: If the original program and mutant program generate the same output, Mutant is kept alive. In such cases, more effective test cases need to be created that kill all mutants.

Mutation Testing

Types of Mutation Testing

1. **Statement Mutation** - developer cut and pastes a part of a code of which the outcome may be a removal of some lines
2. **Value Mutation**- values of primary parameters are modified
3. **Decision Mutation-** control statements are to be changed

Mutation Testing

How to Create Mutant Programs?

A mutation is nothing but a single syntactic change that is made to the program statement. Each mutant program should differ from the original program by one mutation.

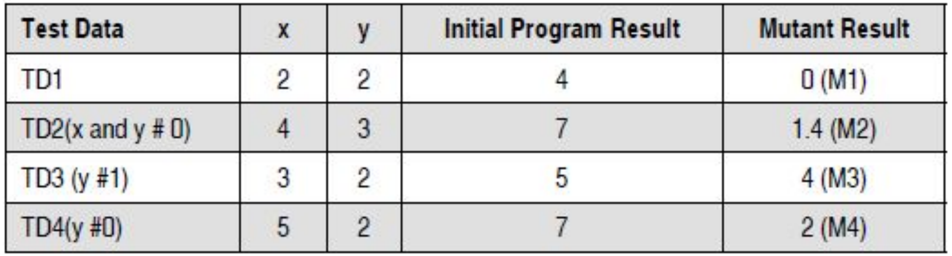
Example

```
if (a > b)
x = x + y;
else
x = y;
printf(“%d”, x);
```

We can consider the following mutants for the above example:

```
M1: x = x – y;
M2: x = x / y;
M3: x = x + 1;
M4: printf(“%d”, y);
```

The results of the initial program and its mutants are shown below.



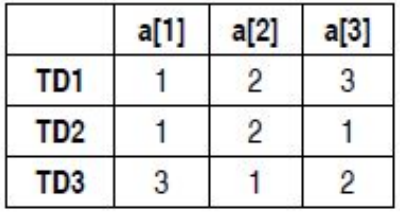
Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y # 0)	4	3	7	1.4 (M2)
TD3 (y #1)	3	2	5	4 (M3)
TD4(y #0)	5	2	7	2 (M4)

Mutation Testing: Example

Consider the program P shown below.

```
r = 1;
for (i = 2; i<=3; ++i) {
if (a[i] > a[r])
r = i; }
```

Let us consider the following test data selection:



	a[1]	a[2]	a[3]
TD1	1	2	3
TD2	1	2	1
TD3	3	1	2

The mutants for P are:

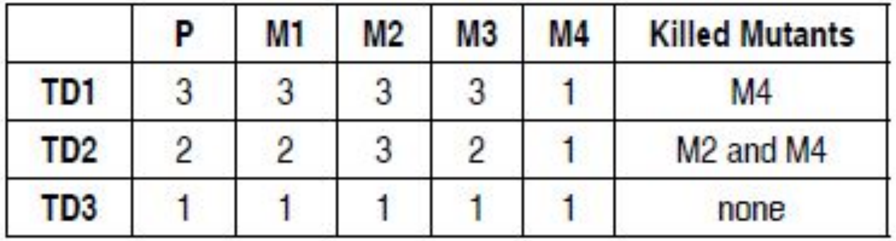
```
M1:
r = 1;
for (i = 1; i<=3; ++i) {
if (a[i] > a[r])
r = i;
}
```

```
M2:
r = 1;
for (i = 2; i<=3; ++i) {
if (i > a[r])
r = i;
}
```

```
M3:
r = 1;
for (i = 2; i<=3; ++i) {
if (a[i] >= a[r])
r = i;
}
```

```
M4:
r = 1;
for (i = 1; i<=3; ++i) {
if (a[r] > a[r])
r = i;
}
```

We apply these test data to mutants, M1, M2, M3, M4.



	P	M1	M2	M3	M4	Killed Mutants
TD1	3	3	3	3	1	M4
TD2	2	2	3	2	1	M2 and M4
TD3	1	1	1	1	1	none

END