Chapter 5
Dynamic Testing: White Box Testing Techniques

OXFORD
HIGHER EDUCATION

Objectives

- White box testing needs the full understanding of the logic/structure of the program.
- 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

Logic Coverage Criteria

- **Statement Coverage**

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.

```
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);
```

Test case 3: x > y
Test case 4: x < y

Logic Coverage Criteria

Decision or Branch Coverage

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

Condition Coverage

while ((I <= 5) && (J < COUNT))

- **Test case 1**: I <= 5, J < COUNT
- **Test case 2:** I > 5, J > COUNT

Logic Coverage Criteria

Decision / Condition Coverage

If (A && B)

- **Test Case 1:** A is True, B is False.
- **Test Case 2:** A is False, B is True.

Multiple Condition Coverage

- **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

Basis Path Testing

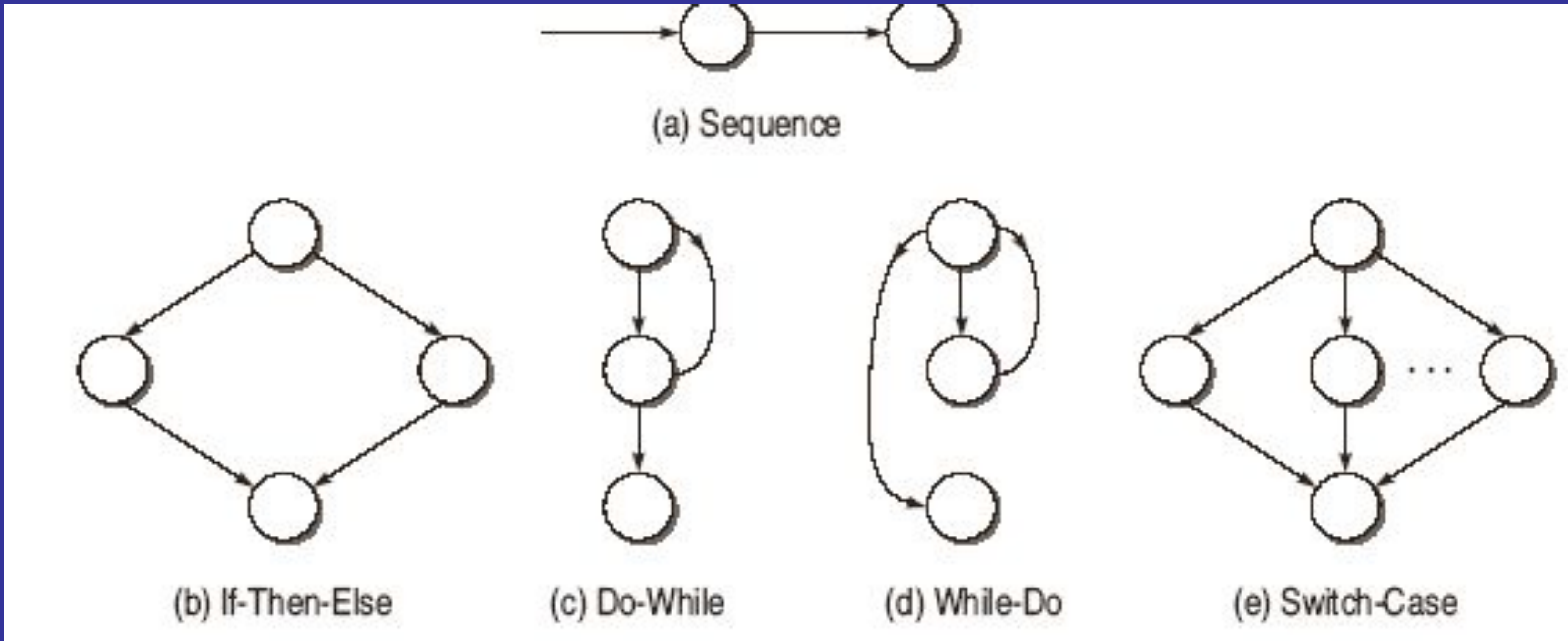
OXFORD
HIGHER EDUCATION

- Path Testing is based on control structure of the program for which flow graph is prepared.
- requires complete knowledge of the program’s structure.
- closer to the developer and used by him to test his module.
- The effectiveness of path testing is reduced with the increase in size of software under test.
- Choose enough paths in a program such that maximum logic coverage is achieved.

Control Flow Graph

- Node
- Edges or Links
- Decision Node
- Junction Node
- Regions

Flow Graph Notations for Different Programming Constructs



Path Testing Terminology

- Path
- Segment
- Path Segment
- Length of a Path
- Independent Path

Path Testing Terminology

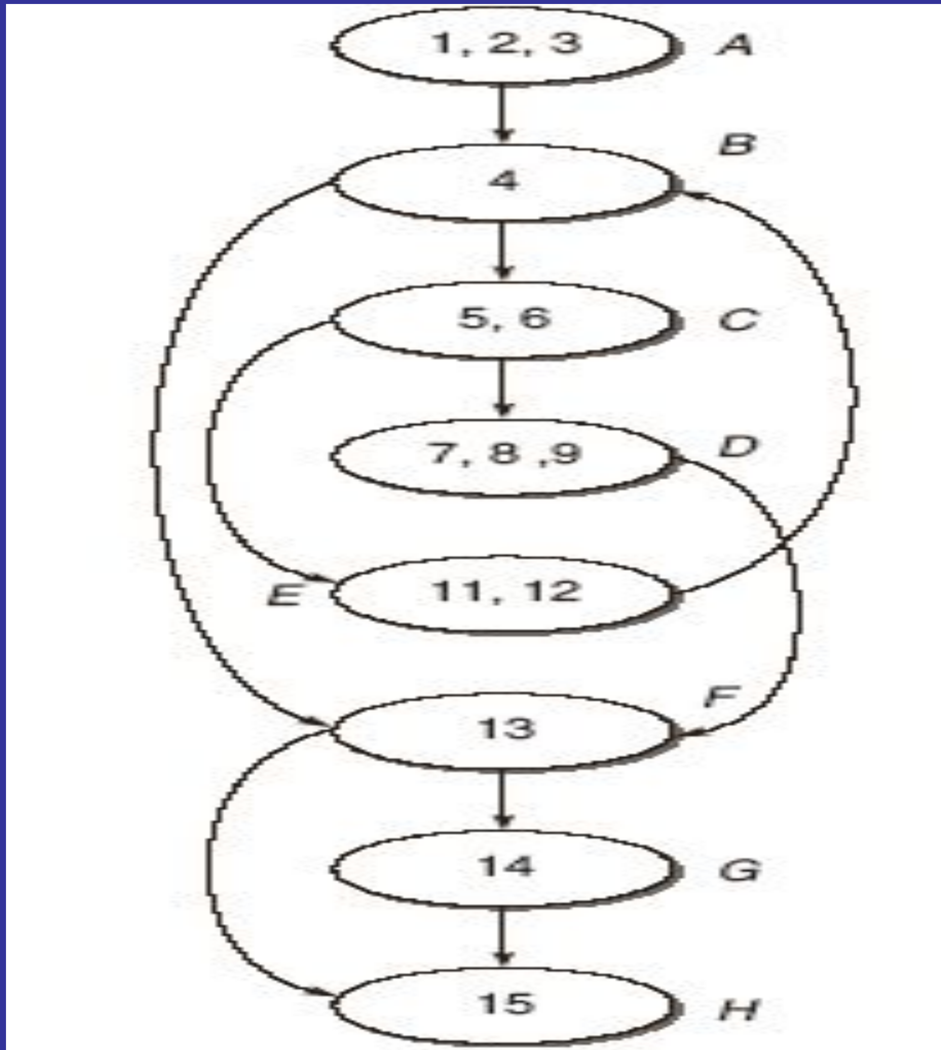
Cyclomatic Complexity

- $V(G) = e-n+2P$
- $V(G) = d + P$
- $V(G)$ = number of Regions in the graph

Example

```
main()
{
    int number, index;
    printf(“Enter a number”);
    scanf(“%d, &number);
    index = 2;
    while(index <= number - 1)
    {
        if (number % index == 0)
        {
            printf(“Not a prime number”);
            break;
        }
        index++;
    }
    if(index == number)
        printf(“Prime number”);
} //end main
```

Example



Example

- **Cyclomatic Complexity**

  V(G) = e – n + 2 * P

  = 10 – 8 +2

  = 4

  V(G) = Number of predicate nodes + 1

  = 3 (Nodes B,C and F) + 1

  = 4

  V(G) = No. of Regions

- = 4 (R1, R2, R3, R4)

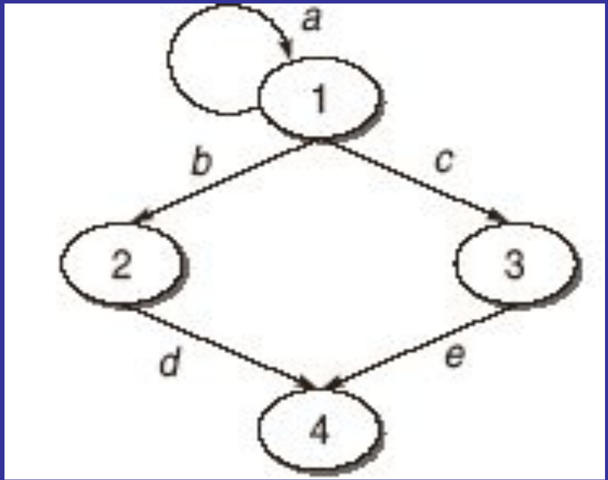
Example

Independent Paths

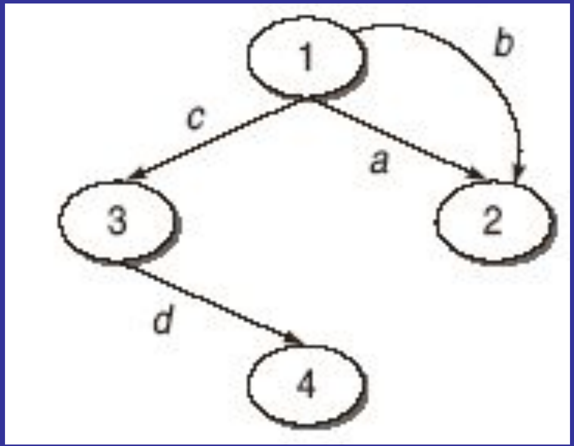
- A-B-F-H
- A-B-F-G-H
- A-B-C-E-B-F-G-H
- A-B-C-D-F-H

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Graph Matrices



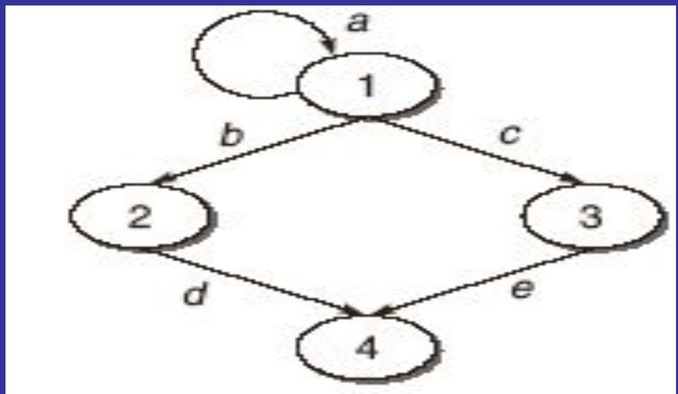
	1	2	3	4
1	a	b	c	
2				d
3				e
4				



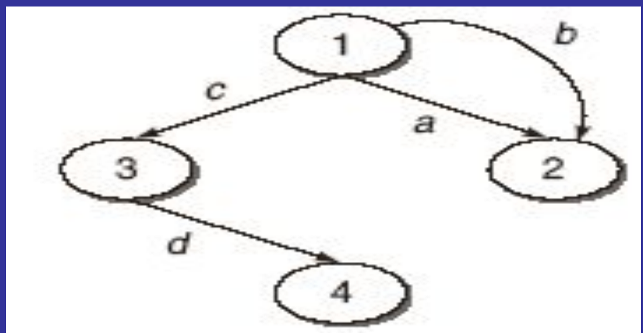
	1	2	3	4
1		a+b	c	
2				
3				d
4				

Graph Matrices

Connection Matrix



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



	1	2	3	4
1		1	1	
2				
3				1
4				

Graph Matrices

- **Use of Connection Matrix in finding Cyclomatic Complexity Number**

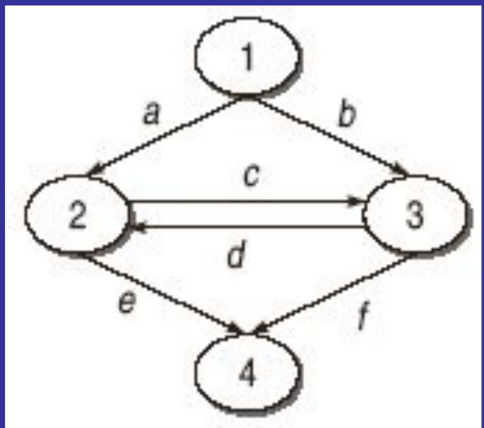
	1	2	3	4	
1	1	1	1		3 − 1 = 2
2				1	1 − 1 = 0
3				1	1 − 1 = 0
4					
	Cyclomatic number = 2+1 = 3				

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

Graph Matrices

- **Use of Graph Matrix for Finding the Set of all Paths**

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

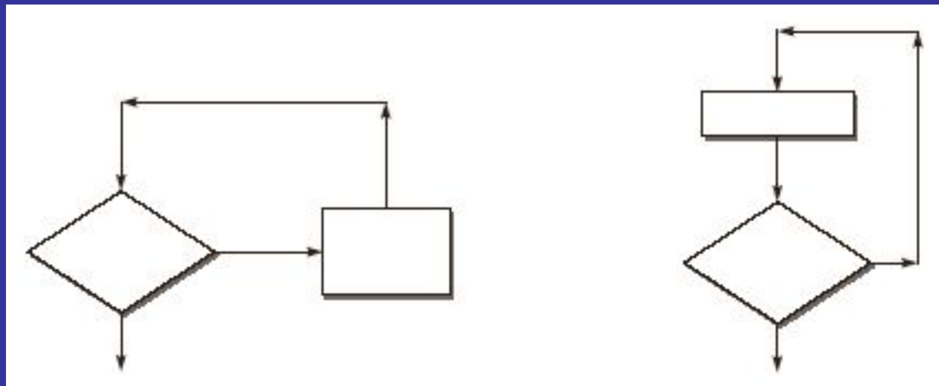


$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

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

Loop Testing

Simple Loops

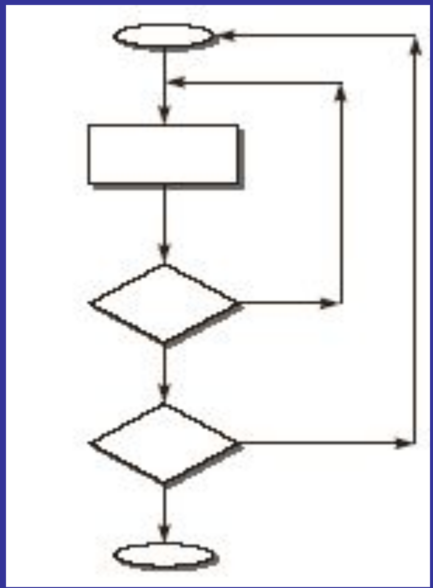


- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for the min, min+1, min-1, max-1, max and max+1 number of iterations through the loop.

Loop Testing

Nested Loops

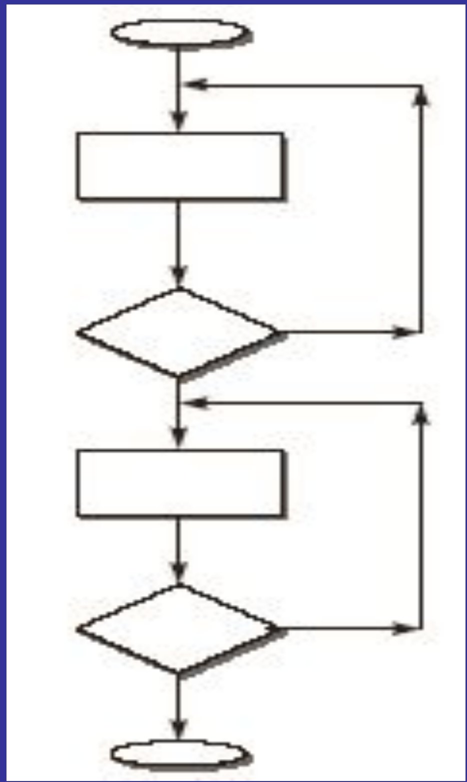
Number of possible test cases grow geometrically. Thus the strategy is to start with the innermost loops while holding outer loops to their minimum values. Continue this outward in this manner until all loops have been covered



Loop Testing

Concatenated Loops

loops are concatenated if it is possible to reach one after exiting the other while still on a path from entry to exit.



Data Flow Testing

OXFORD
HIGHER EDUCATION

Detect improper use of data values due to coding errors.

Closely examines the state of the data in the control flow graph resulting in a richer test suite than the one obtained from control flow graph based path testing strategies like branch coverage, all statement coverage, etc

Data Flow Testing

- **Defined (d):**
- **Killed / Undefined / Released (k):**
- **Usage (u):**

  computational use (c-use) or predicate use (p-use).

Data Flow Testing

- **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

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.

Data Flow Testing

Terminology used in Data Flow Testing

- **Definition Node**

  Input statements, Assignment statements, Loop control statements, Procedure calls, etc.
- **Usage Node**

  Output statements, Assignment statements (Right), Conditional statements, Loop control statements, etc.
- **Loop Free Path Segment**
- **Simple Path Segment**
- **Definition-Use Path (du-path)**

  A du-path with respect to a variable v is a path between definition node and usage node of that variable. Usgae node can be p-usage or c-usgae node.
- **Definition-Clear path(dc-path)**

  A dc-path with respect to a variable v is a path between definition node and usage node such that no other node in the path is a defining node of variable v.

Data Flow Testing

Static Data Flow Testing

With static analysis, the source code is analyzed without executing it.

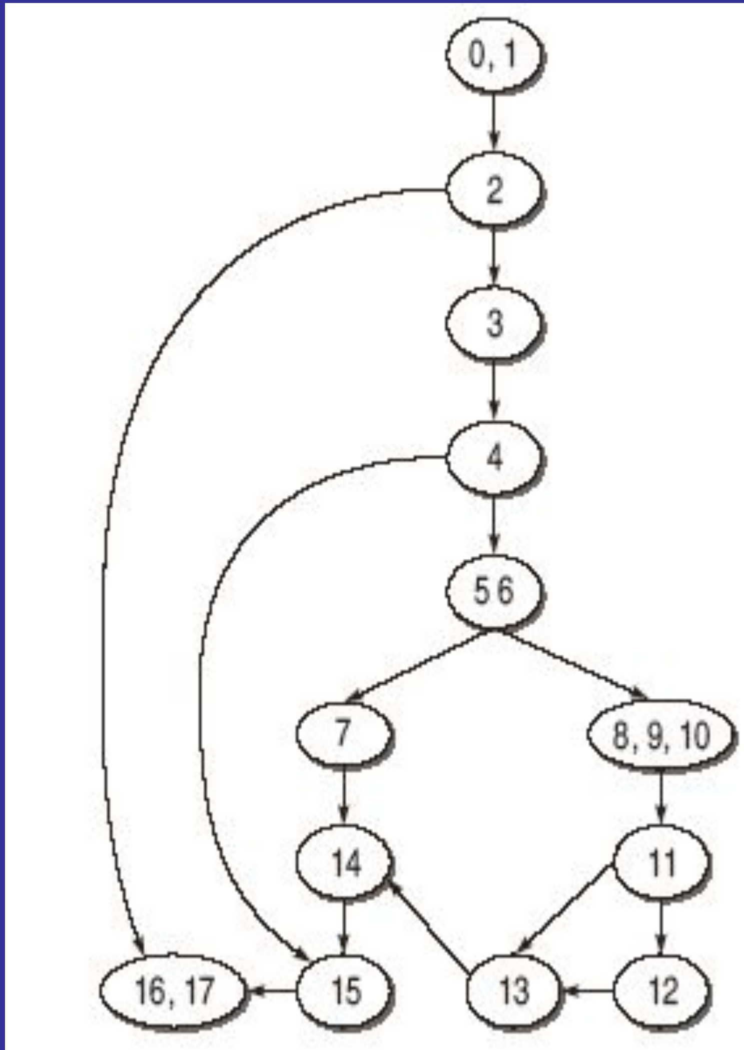
Dynamic Data-Flow Testing

- **All-du Paths (ADUP)**
- **All-uses (AU)**
- **All-p-uses / Some-c-uses (APU + C)**
- **All-c-uses / Some-p-uses (ACU + P)**
- **All-Predicate-Uses(APU)**
- **All-Computational-Uses(ACU)**
- **All-Definition (AD)**

Data Flow Testing

```
main()
{
int work;
double payment =0;
scanf("%d", work);
if (work > 0) {
     payment = 40;
if (work > 20)
{
     if(work <= 30)
          payment = payment + (work - 25) * 0.5;
     else
     {
          payment = payment + 50 + (work -30) * 0.1;
               if (payment >= 3000)
                    payment = payment * 0.9;
     }
}
}
printf("Final payment", payment);
```

Data Flow Testing



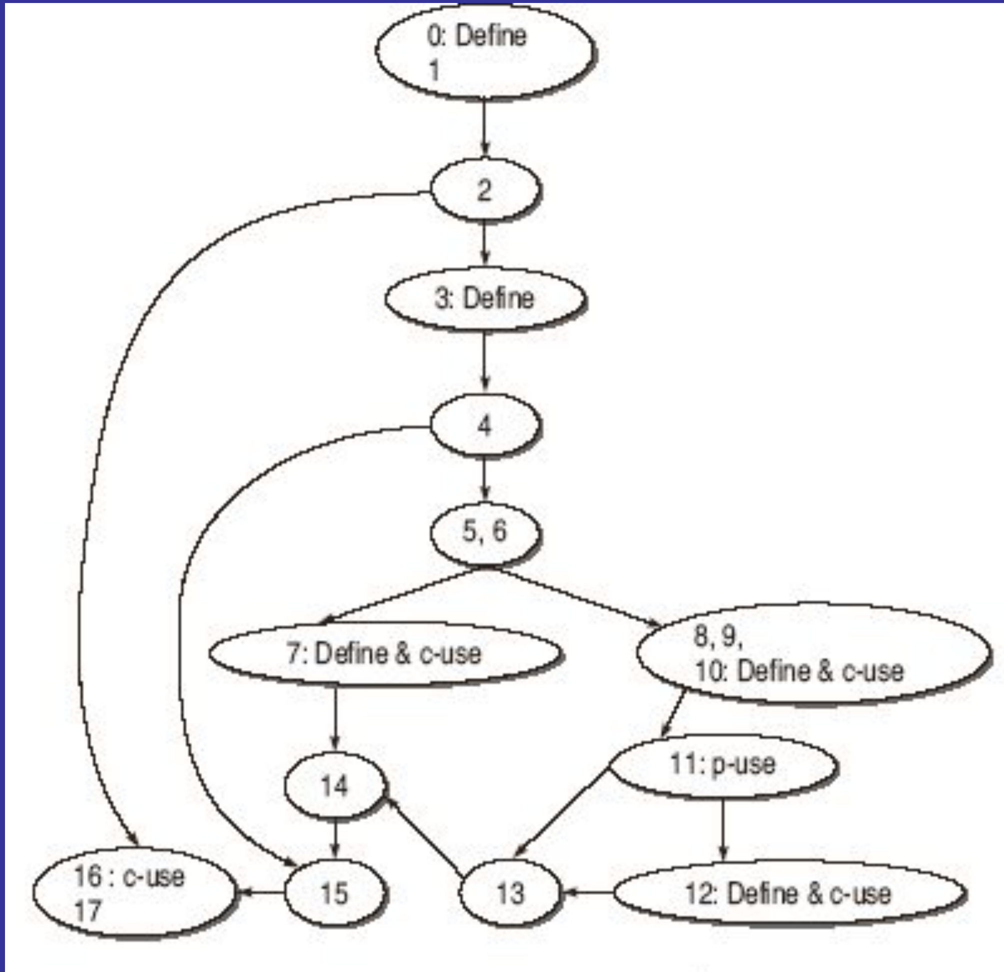
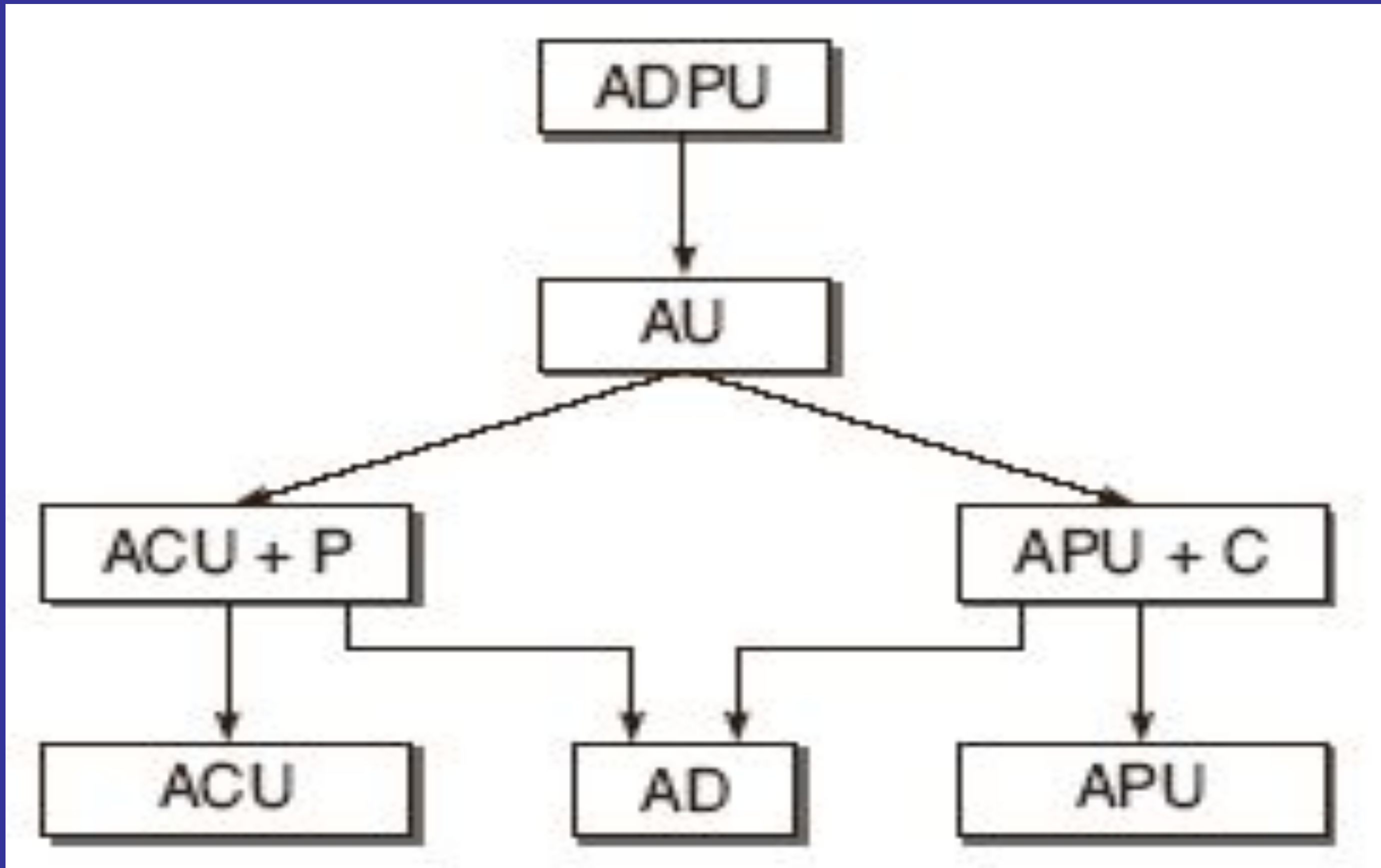


Figure 5.13 Data flow graph for 'payment'

Data Flow Testing



Mutation Testing

- Mutation testing is the process of mutating some segment of code(putting some error in the code) and then testing this mutated code with some test data. If the test data is able to detect the mutations in the code,

- Mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data. During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail.

- Faulty programs are called *mutants* of the original program and a mutant is said to be *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead*

Mutation Testing

- Let us take one example of C program shown below

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

We can consider the following mutants for above example:

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

Mutation Testing

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)