DATA FLOW TESTING

DATA FLOW TESTING

Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors.

Example:

```
int a;
if(a == 67) { }
```

Data flow testing closely examines the state of the data in the control flow graph.

STATE OF A DATA OBJECT

A data object can be in the following states:

(i) Defined (d):

A data object is called defined when it is initialized.

(ii) Killed/Undefined/Released (k):

When the data has been reinitialized? (i.e. exiting the loop or memory is released dynamically or a file has been closed).

(iii) Usage (u):

When the data object is used in somewhere in program?. {In general, we say that the usage is either computational use (c-use) or predicate use (p-use)}.

DATA-FLOW ANOMALIES

It represent the patterns of data usage which may lead to an incorrect execution of the code.

An anomaly is denoted by a Single (or) Two-character sequence of actions.

To represent Single-character data anomalies the following conventions are used:

~x : indicates first appearance of x.

x~ : indicates last appearance of 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.

There are nine possible two-character 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.**

DYNAMIC DATA FLOW TESTING

Used to uncover possible bugs in data usage during the execution of the code.

Various strategies employed for the creation of test cases are listed below:

(1) All-du Paths (ADUP):

Every du-path from every definition of every variable to every use of that definition should be exercised under some test.

(2) All-uses (AU):

Every use of the variable, there is a path from the definition of that variable to the use.

(3) All-p-uses/Some-c-uses (APU + C):

Every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use.

(4) All-c-uses/Some-p-uses (ACU + P):

Every variable and every definition of that variable, include at least one dc-path from the definition to every computational use.

(5) All-Predicate-Uses (APU):

Every variable there is a path from every definition to every p-use of that definition.

If there is a definition with no p-use following it, then it is dropped from contention.

(6) All-Computational-Uses (ACU):

Every variable, there is a path from every definition to every c-use of that definition.

If there is a definition with no c-use following it, then it is dropped from contention.

(7) All-Definition (AD):

Every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

DYNAMIC DATA FLOW TESTING - Example

```
1. read x;
2. if (x > 0)
3.   a = x + 1;
4. if (x <= 0)
5.   if (x < 1){
6.     x = x +1; goto (5)
       else
7.   a = x + 1; }
8. print a
```

Selection of Path for data flow:
(using statement coverage Technique):

Path 1:

Set Input value for x = -1

Path Covered: 1,2,4,5,6,5,6,5,7,8

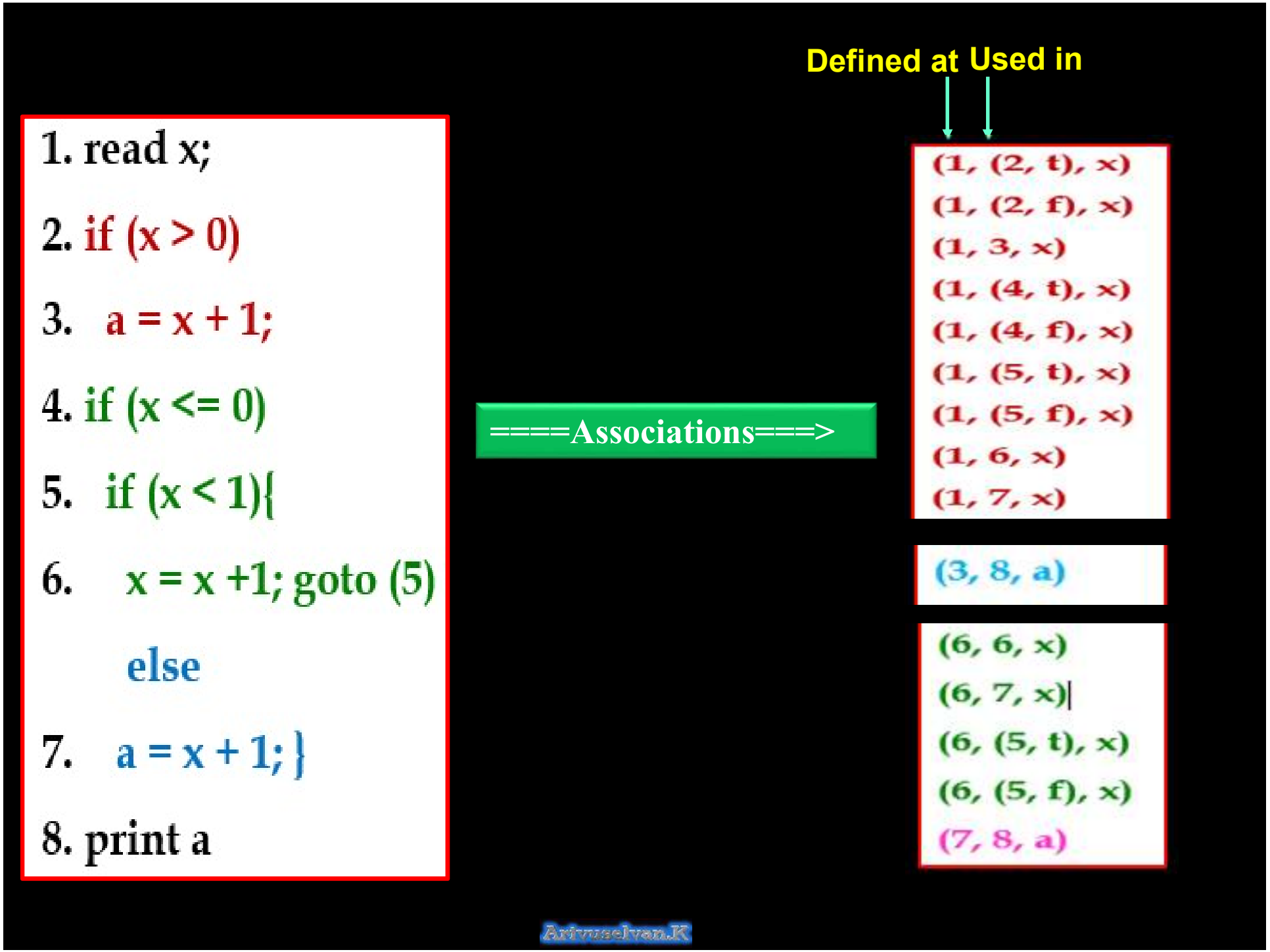
Output: 2

Path 2:

Set Input value for x = 1

Path Covered: 1,2,3,8

Output: 2


Defined at Used in
1. read x;
2. if (x > 0)
3. a = x + 1;
4. if (x <= 0)
5. if (x < 1){
6. x = x +1; goto (5)
else
7. a = x + 1; }
8. print a
====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)

STEP: 1

All Definitions path Coverage:

(1, (2, f), x)

(6, (5, f), x)

(3, 8, a)

(7, 8, a)

STEP: 2

All c-use path Coverage:

(1, 3, x)

(1, 6, x)

(1, 7, x)

(6, 6, x)

(6, 7, x)

(3, 8, a)

(7, 8, a)

STEP: 3

All p-use path Coverage:

(1, (2, t), x)

(1, (2, f), x)

(1, (4, t), x)

(1, (4, f), x)

(1, (5, t), x)

(1, (5, f), x)

(6, (5, t), x)

(6, (5, f), x)

STEP: 4

All c-use some p-use path coverage:

(i.e. List down all c-use for every definition, if for some definitions do not have c-use then include p-use)

(1, 3, x)

(1, 6, x)

(1, 7, x)

(6, 6, x)

(6, 7, x)

(3, 8, a)

(7, 8, a)

STEP: 5

All p-use some c-use path coverage:

(i.e. List down all p-use for every definition, if for some definitions do not have p-use then include c-use)

(1, (2, t), x)

(1, (2, f), x)

(1, (4, t), x)

(1, (4, f), x)

(1, (5, t), x)

(1, (5, f), x)

(6, (5, t), x)

(6, (5, f), x)

(3, 8, a)

(7, 8, a)

STEP: 6 All use path Coverage:

(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)

STEP: 7

All du path Coverage:

If some uses(c or p) have more than one path, then define all path.

(For example in above case to reach line number: 3, we have only one path, if we have more than one possible path then include all paths)

Example -2

```
1. read a, b, c;
2. if(a>b)
3.     x = a+1
4.     print x;
   else
5.     x = b-1
6.     print z;
```

Data	Define (d)	Use (u)
a	1	2, 3
b	1	2, 5
c	1	Not Applicable
x	3, 5	4
z	Not Applicable	6

STATIC DATA FLOW TESTING

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

Example:

Consider the program given below 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.       float 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.

Solution

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

```
    main()
    {
1.      float 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. }
```

Solution

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

```
    main()
    {
1.      float 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. }
```

Solution

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

Pattern	Line Number	Explanation
~d	7	Normal case. Allowed
du	7-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

```
    main()
    {
1.      float 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. }
```

Solution

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

```
    main()
    {
1.      float 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. }
```

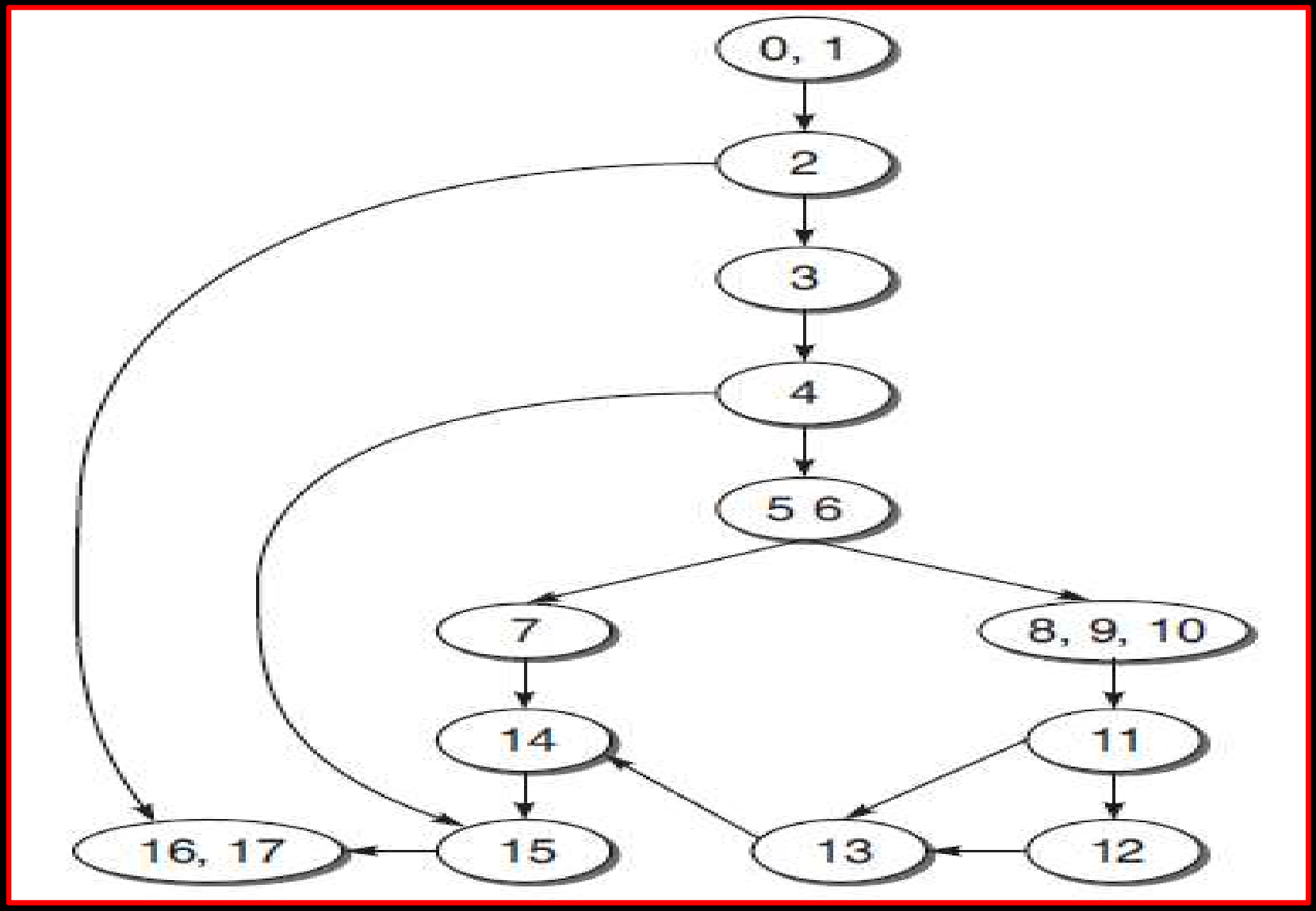
From the above static analysis, it was observed that static data flow testing for the variable 'hra' discovered one bug of double definition in line number 1.

DATA FLOW GRAPH

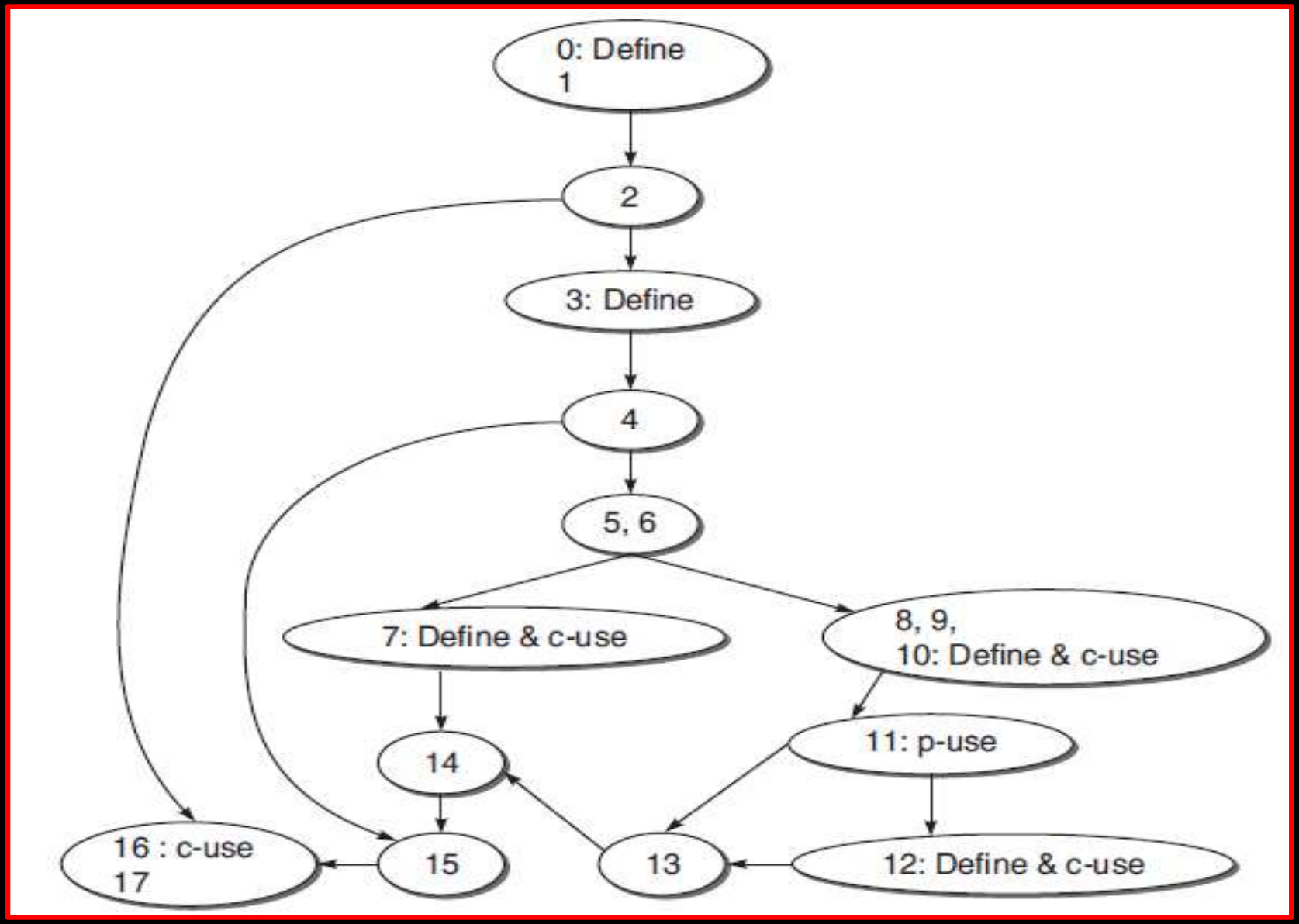
Consider the program given below. Draw its control flow graph and data flow graph for each variable used in the program, and derive data flow testing paths with all the strategies discussed above using dynamic testing strategies .

```
     main()
     {
     int work;
 0.  double payment =0;
 1.  scanf("%d", work);
 2.  if (work > 0) {
 3.       payment = 40;
 4.  if (work > 20)
 5.  {
 6.       if(work <= 30)
 7.            payment = payment + (work - 25) * 0.5;
 8.       else
 9.       {
10.            payment = payment + 50 + (work -30) * 0.1;
11.                 if (payment >= 3000)
12.                      payment = payment * 0.9;
13.       }
14.  }
15.  }
16.  printf("Final payment", payment);
```

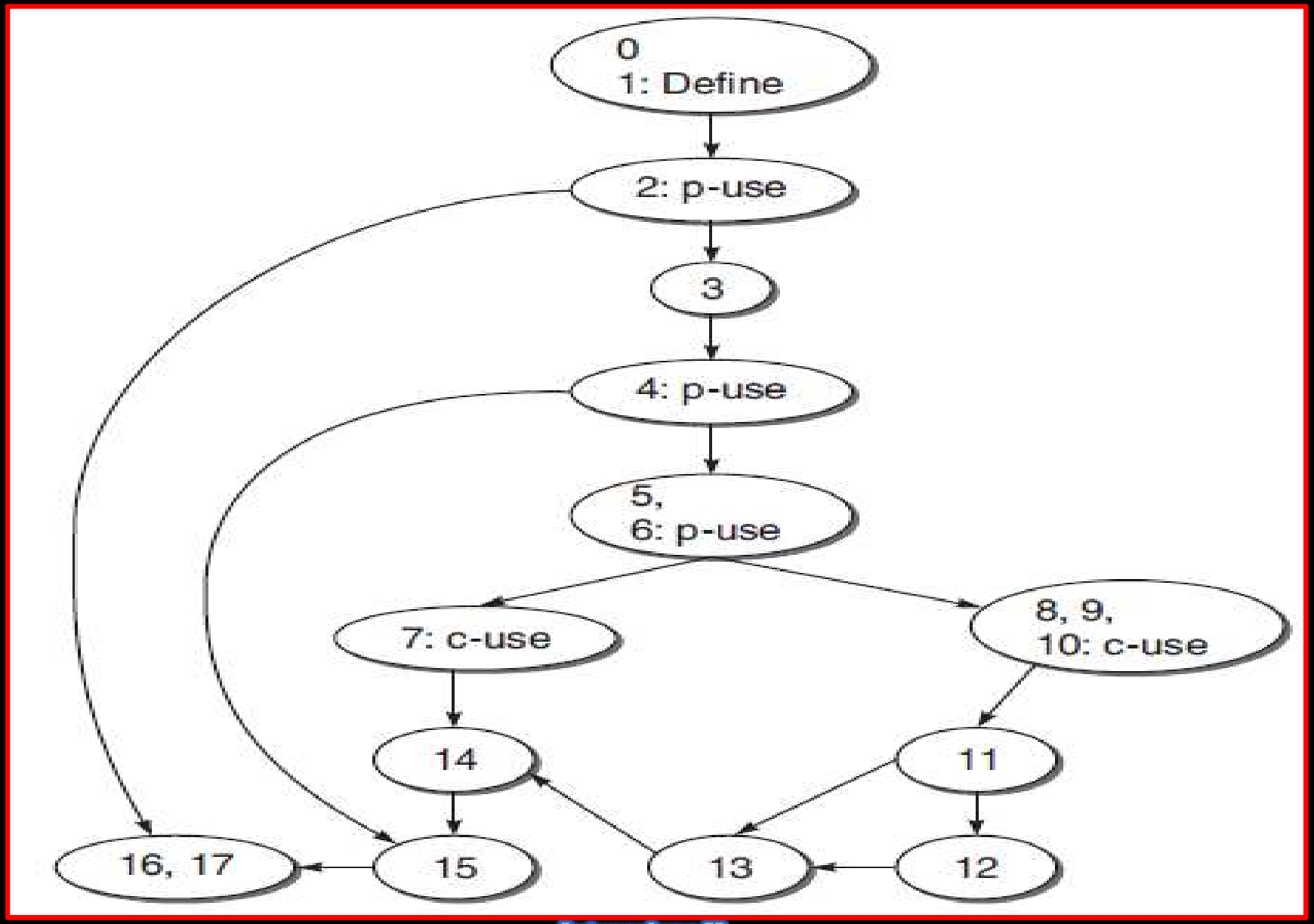
Control flow graph for the given program.



Data flow graph for the variable 'payment'.

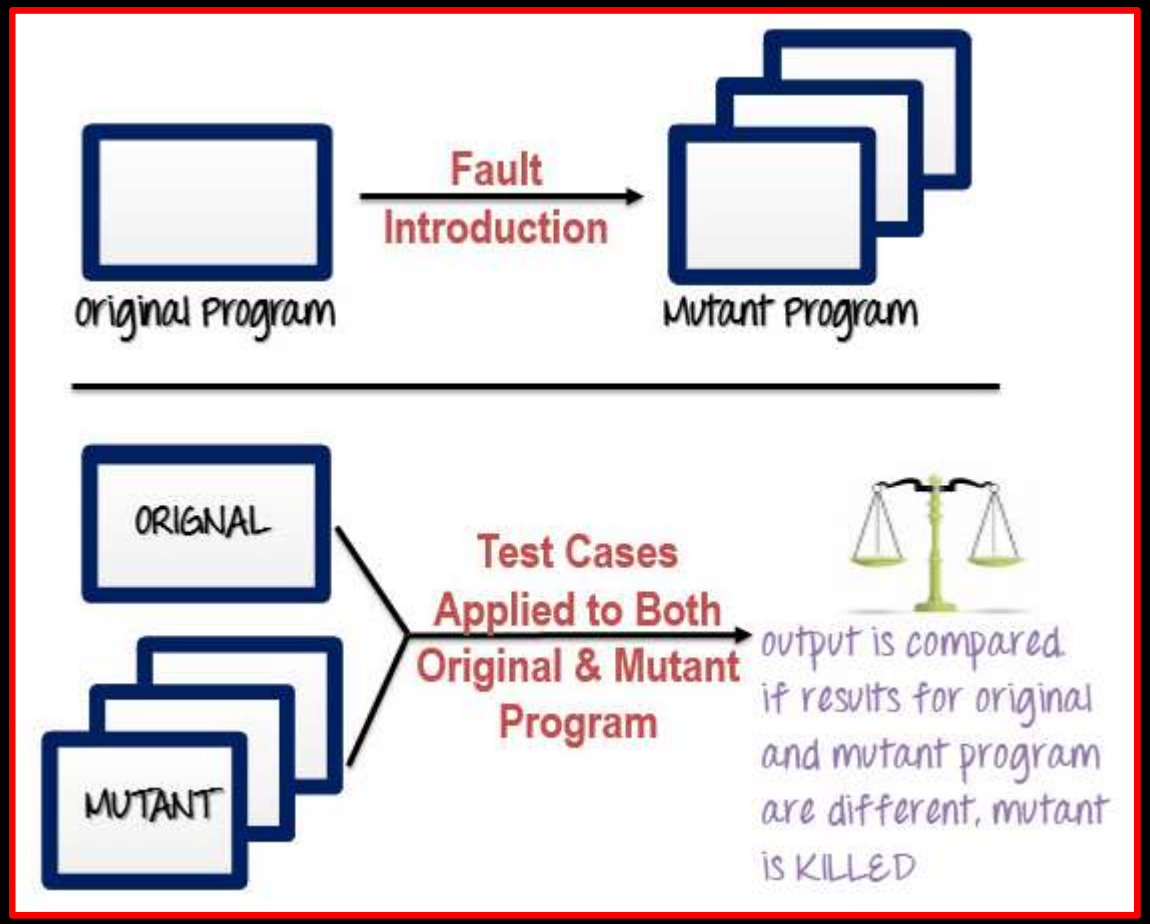


Data flow graph for the variable 'work'.

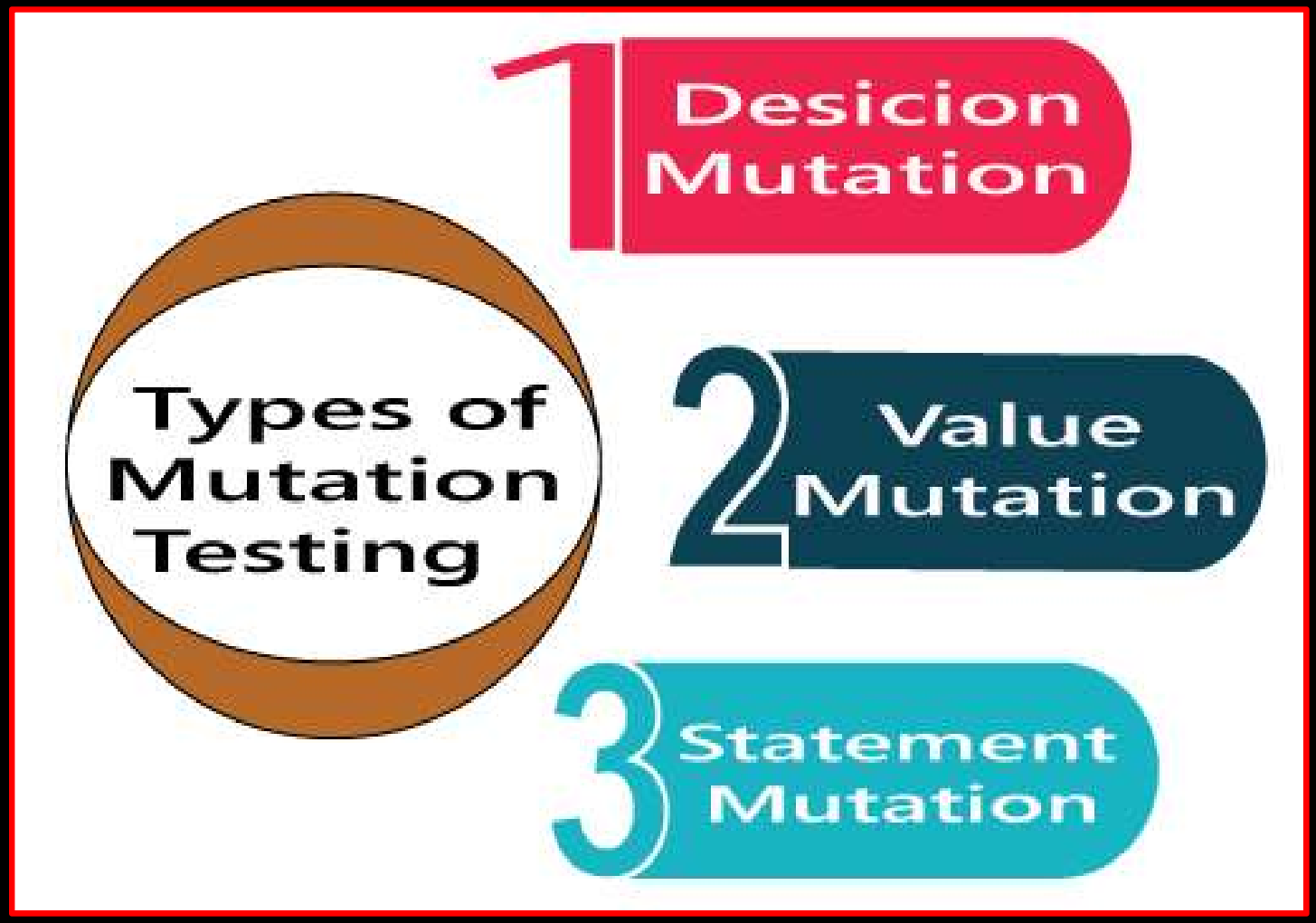


MUTATION TESTING

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, then the test data is quite good, otherwise we must focus on the quality of test data.



(1) Decision mutations:

We will do the modification in arithmetic and logical operator to detect the errors in the program.

Original Code	Modified Code

if(p >q)
r= 5;
else
r= 15;
	if(p < q)
	r = 5;
	else
	r = 15;

(2) Value mutations:

The values will modify to identify the errors in the program, and generally, we will change the following: Small value to higher value / Higher value to Small value.

Original Code	Modified Code

int add =9000008;
int p = 65432;
int q =12345;
int r = (p+ q);
	int mod = 9008;
	int p = 65432;
	int q =12345;
	int r= (p + q);

MUTATION TESTING

(3) Statement Mutations:

we can do the modifications into the statements by removing or replacing the line.

Original Code	Modified Code

if(p * q)
r = 15;
else
r = 25;
	if(p* q)
	s = 15;
	else
	s = 25;

PRIMARY MUTANTS

When the mutants are single modifications of the initial program using some operators they are called primary mutants.

```
...
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);`

SECONDARY MUTANTS

when multiple levels of mutation are applied on the initial program they are called secondary mutants

```
if (a < b)
        c = a;
```

Now, mutants for this code may be as follows:

M1 :
```
if (a <= b-1)
        c = a;
```
M2:
```
if (a+1 <= b)
        c = a;
```
M3:
```
if (a == b)
        c = a+1;
```