

# MUTATION TESTING

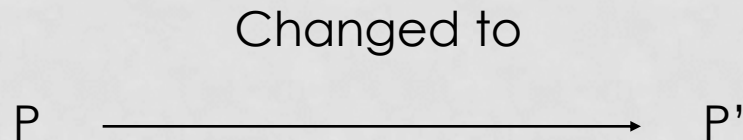


# WHAT IS ADEQUACY?

- Consider a program  $P$  written to meet a set  $R$  of functional requirements. We notate such a  $P$  and  $R$  as  $(P, R)$ . Let  $R$  contain  $n$  requirements labeled  $R_1, R_2, \dots, R_n$ .
- Suppose now that a set  $T$  containing  $k$  tests has been constructed to test  $P$  to determine whether or not it meets all the requirements in  $R$ . Also,  $P$  has been executed against each test in  $T$  and has produced correct behavior.
- We now ask: **Is  $T$  good enough?** This question can be stated differently as: **Has  $P$  been tested thoroughly?**, or as: **Is  $T$  adequate?**

# WHAT IS PROGRAM MUTATION?

- Suppose that program P has been tested against a test set T and P has not failed on any test case in T. Now suppose we do the following:



What behavior do you expect from P' against tests in T?

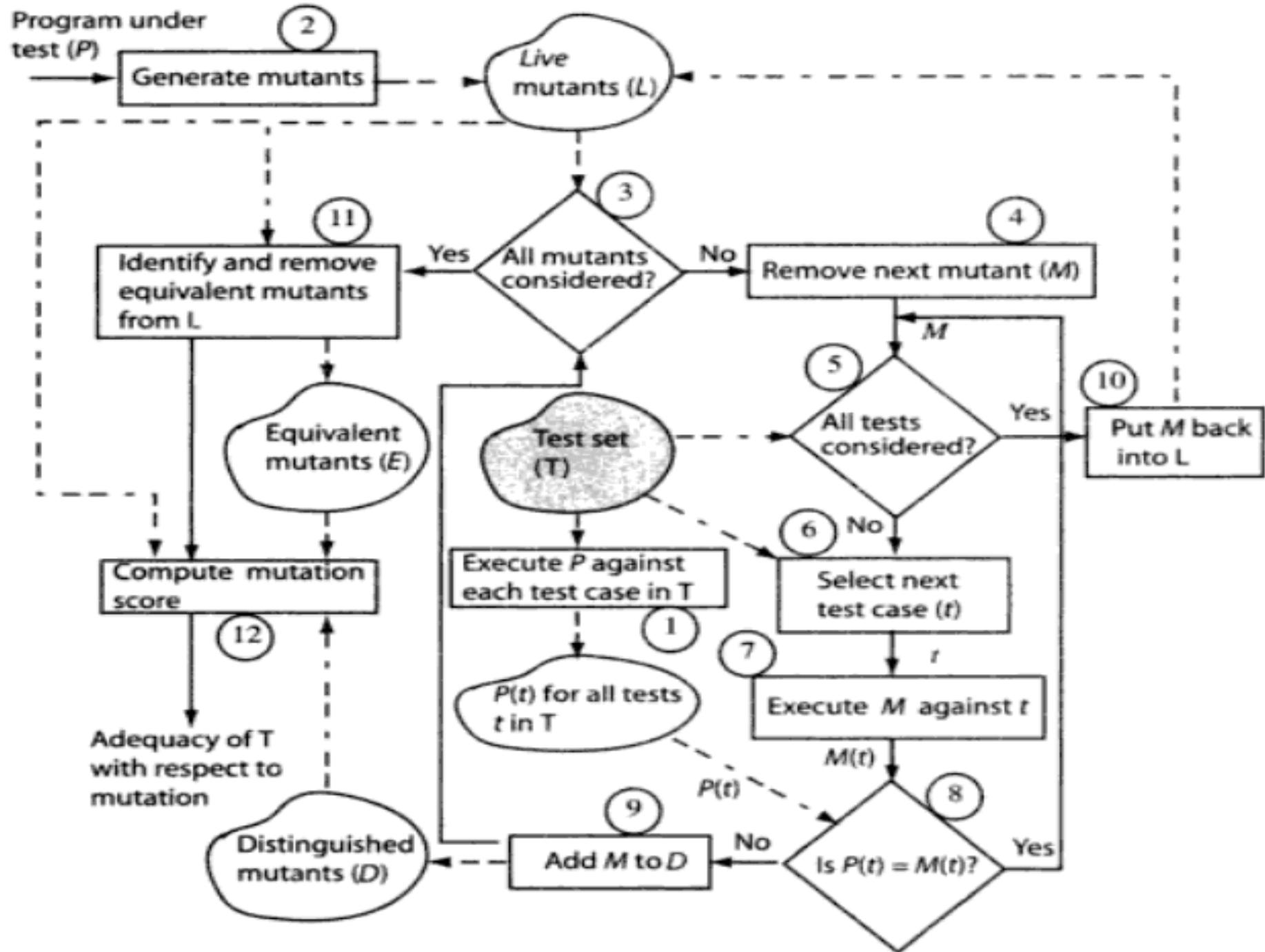
# WHAT IS PROGRAM MUTATION? [2]

- $P'$  is known as a **mutant** of  $P$ .
- There might be a test  $t$  in  $T$  such that  $P(t) \neq P'(t)$ . In this case we say that  $t$  **distinguishes**  $P'$  from  $P$ . Or, that  $t$  has **killed**  $P'$ .
- There might be not be any test  $t$  in  $T$  such that  $P(t) \neq P'(t)$ . In this case we say that  $T$  is unable to distinguish  $P$  and  $P'$ . Hence  $P'$  is considered **live** in the test process.

# WHAT IS PROGRAM MUTATION? [3]

- If there does not exist any test case  $t$  in the input domain of  $P$  that distinguishes  $P$  from  $P'$  then  $P'$  is said to be **equivalent** to  $P$ .
- If  $P'$  is not equivalent to  $P$  but no test in  $T$  is able to distinguish it from  $P$  then  $T$  is considered **inadequate**.
- A non-equivalent and live mutant offers the tester an opportunity to generate a new test case and hence **enhance**  $T$ .

*We will refer to program mutation as mutation.*



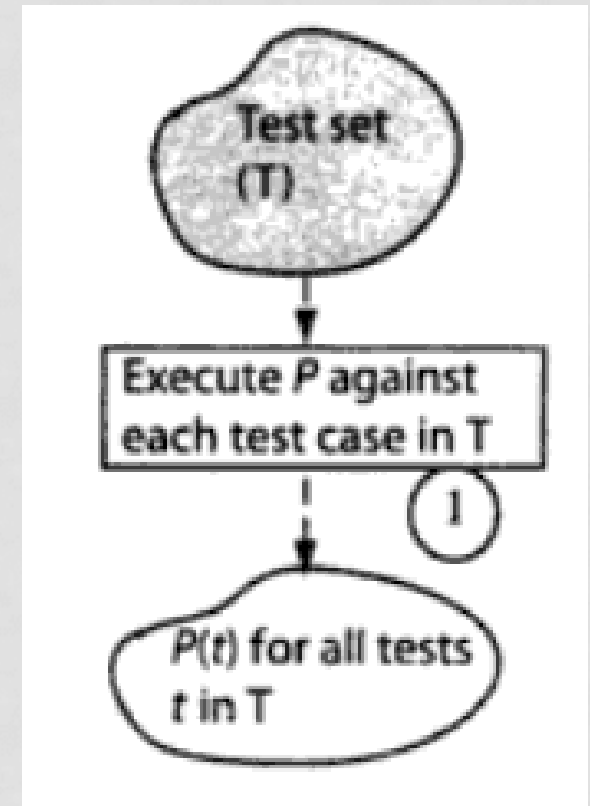
# 1 - PROGRAM EXECUTION

$P$  is supposed to compute the following function  $f(x, y)$ :

$$f(x, y) = \begin{cases} x + y & \text{if } x < y, \\ x * y & \text{otherwise.} \end{cases}$$

Suppose we have tested  $P$  against the following set of tests:

$$T_P = \left\{ \begin{array}{l} t_1 : \langle x = 0, y = 0 \rangle, \\ t_2 : \langle x = 0, y = 1 \rangle, \\ t_3 : \langle x = 1, y = 0 \rangle, \\ t_4 : \langle x = -1, y = -2 \rangle \end{array} \right\}$$



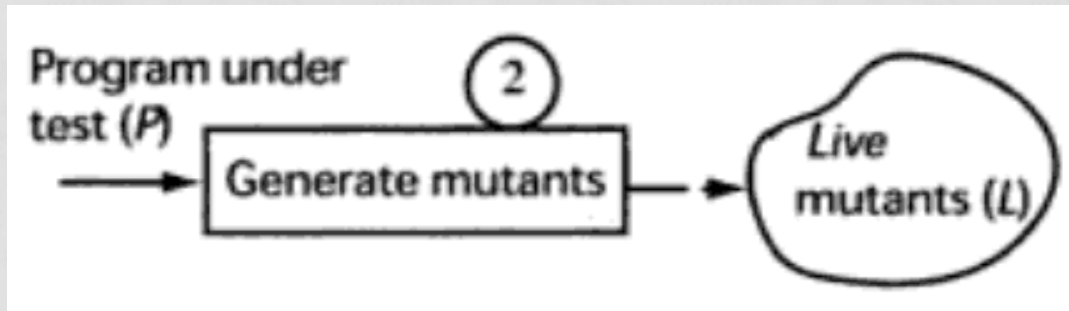
# 1 - PROGRAM EXECUTION

```
begin
    int x, y;
    input(x,y);
    if(x<y)
        output(x+y);
    else
        output(x*y);
end
```

Test case ( $t$ )	Expected output $f(x, y)$	Observed output $P(t)$
$t_1$	0	0
$t_2$	1	1
$t_3$	0	0
$t_4$	2	2



## 2 - MUTANT GENERATION



- Replace '+' with '-'
- Replace '\*' with '/'
- Replace variable 'v' with 'v+1'

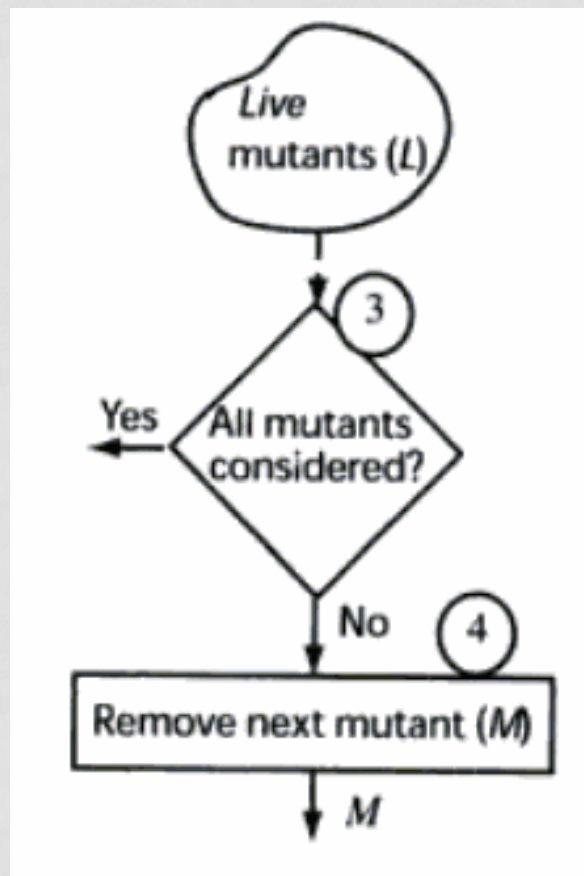
## 2 - MUTANT GENERATION

Line	Original	Mutant ID	Mutant(s)
1	begin		None
2	int x, y		None
3	input (x, y)		None
4	if (x<y)	$M_1$ $M_2$	if (x+1<y) if (x<y+1)
5	then		None
6	output(x+y)	$M_3$ $M_4$ $M_5$	output(x+1+y) output(x+y+1) output(x-y)
7	else		None
8	output(x*y)	$M_6$ $M_7$ $M_8$	output((x+1)*y) output(x*(y+1)) output(x/y)
9	end		None

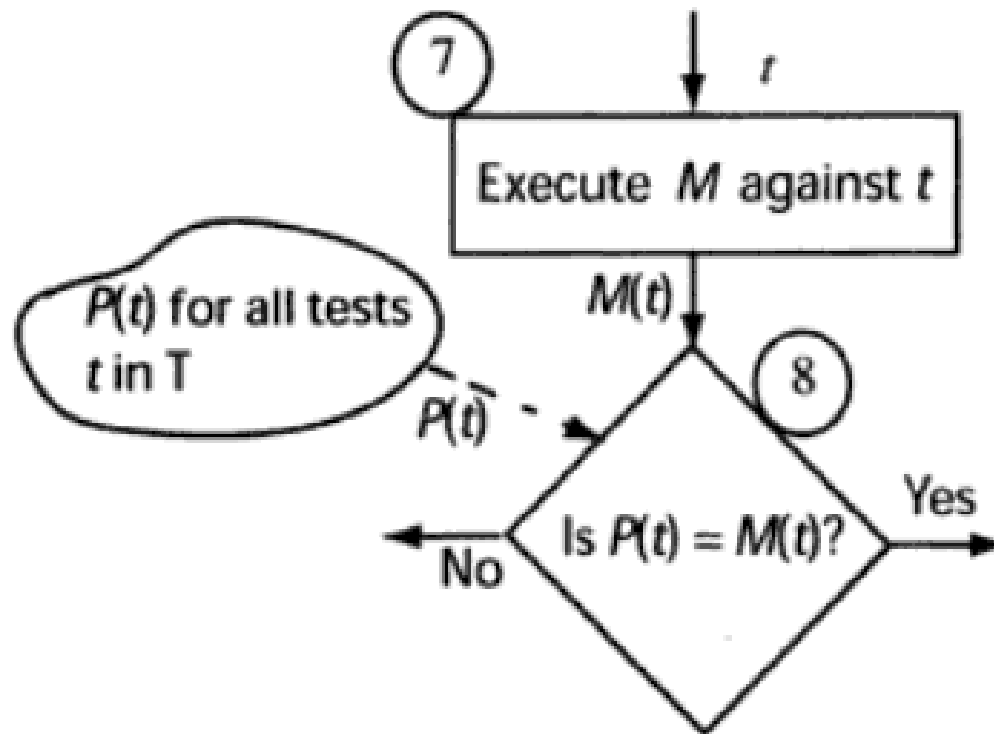
$L = \{M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8\}$

## 3 & 4 - SELECT NEXT MUTANT

$L = \{M1, M2, M3, M4, M5, M6, M7, M8\}$



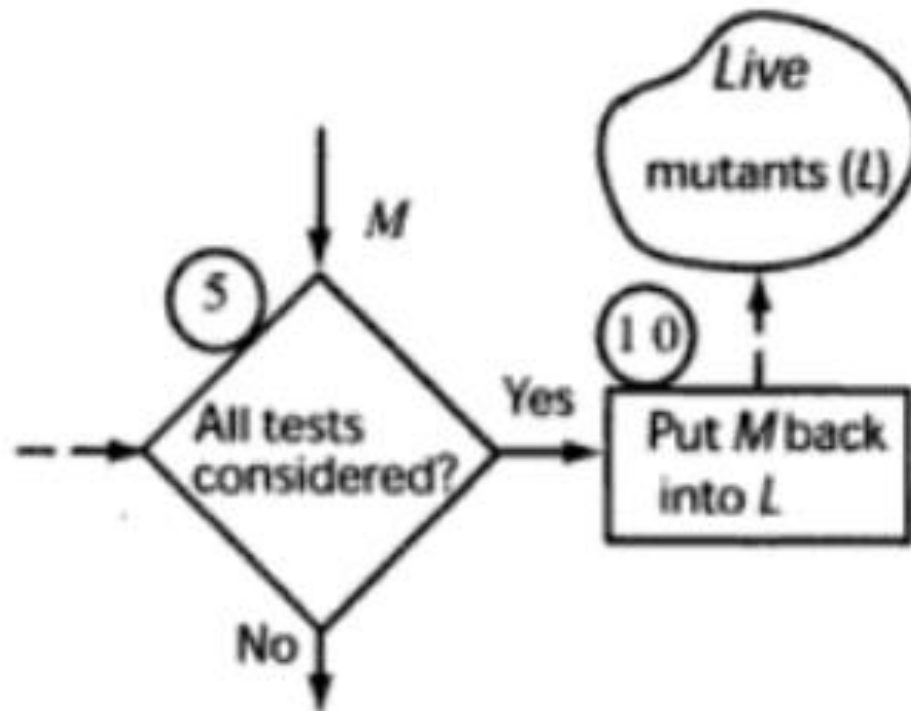
# 7, 8 & 9 – MUTANT EXECUTION AND CLASSIFICATION



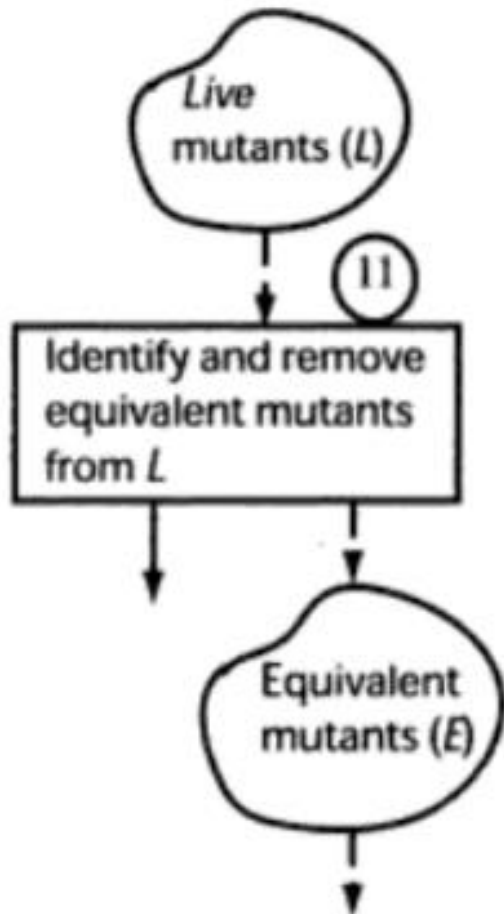
# 7, 8 & 9 – MUTANT EXECUTION AND CLASSIFICATION

Program	t1	t2	t3	t4	D
P(t)	0	1	0	2	{}
<b>Mutant</b>					
M1(t)	0	0*	NE	NE	{M1}
M2(t)	0	1	0	2	{}
M3(t)	0	2*	NE	NE	{M1, M3, M4}
M4(t)	0	2*	NE	NE	{M1, M3, M4, M5}
M5(t)	0	-1*	NE	NE	{M1, M3}
M6(t)	0	1	0	0*	{M1, M3, M4, M5, M6, M7}
M7(t)	0	1	1*	NE	{M1, M3, M4, M5, M6, M7, M8}
M8(t)	U*	NE	NE	NE	{M1, M3, M4, M5, M6}

# 10 - LIVE MUTANTS



# 11 - CHECK IF EQUIVALENT MUTANT



$$f_P(x, y) = \begin{cases} x + y & \text{if } x < y, \\ x * y & \text{otherwise.} \end{cases}$$

$$g_{M_2} P(x, y) = \begin{cases} x + y & \text{if } x < y + 1, \\ x * y & \text{otherwise.} \end{cases}$$

Not a equivalent mutant because we can add an extra test case such that Mutant can be distinguished.

$t : \langle x = 1, y = 1 \rangle$

# EQUIVALENT MUTANT EXAMPLE

## Sample Program P

.....

.....

```
if(x == 2 && y == 2)
```

```
  z = x + y;
```

.....

## Mutant P'

.....

.....

```
if(x == 2 && y == 2)
```

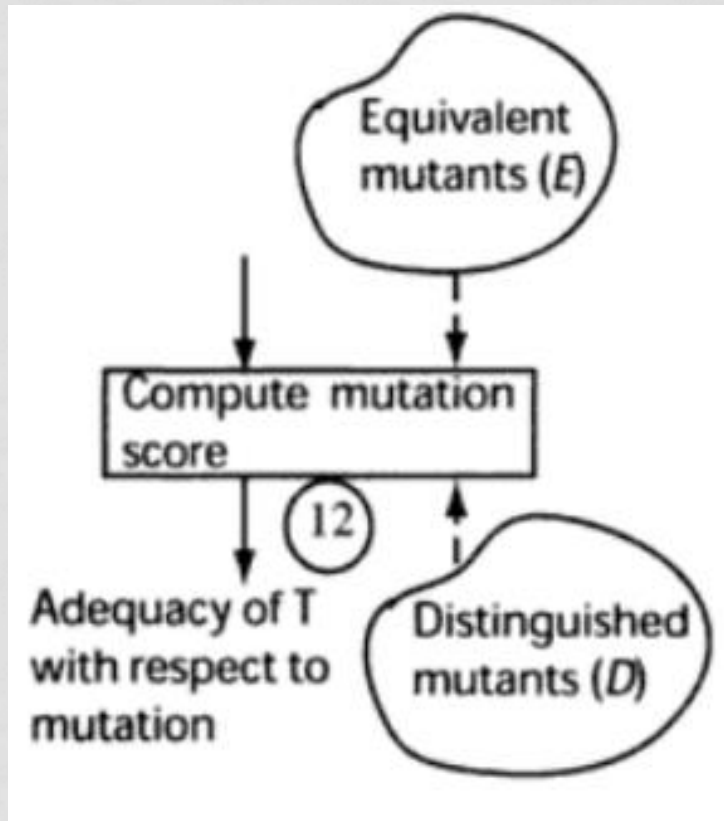
```
  z = x * y;
```

.....

The value of Z will be equal to 4; any test set will be unable to determine any faults with this program because the value will always be equal to 4.



# 12 – COMPUTATION OF MUTATION SCORE



**If no equivalent mutants**

$$MS = \frac{|D|}{|D| + |L|}$$

**With equivalent mutants**

$$MS = \frac{|D|}{|M| - |E|}$$

# HIGH COST MUTATION TESTING

- The main drawback of mutation testing is the high cost of running vast number of mutants against the set of test cases.
- The number of mutants generated for a program is proportional to the product of the number of data references and the number of data objects.
- This number is large for even small programs.

# ERROR DETECTION USING MUTATION

- Consider the following function **foo** that is required to return the sum of two integers  $x$  and  $y$ . Clearly **foo** is incorrect.

```
int foo(int x, y){  
  return (x-y);  
}
```

← This should be return (x+y)

# ERROR DETECTION USING MUTATION

- Now suppose that **foo** has been tested using a test set T that contains two tests:

$$T = \{ t1: \langle x=1, y=0 \rangle, t2: \langle x=-1, y=0 \rangle \}$$

- First note that **foo** behaves perfectly fine on each test in, i.e. **foo** returns the expected value for each test case in T. Also, T is adequate with respect to all control and data flow based test adequacy criteria.

# ERROR DETECTION USING MUTATION

Let us evaluate the adequacy of T using mutation. Suppose that the following three mutants are generated from **foo**.

M1: 

```
int foo(int x, y){  
  return (x+y);  
}
```

M2: 

```
int foo(int x, y){  
  return (x-0);  
}
```

M3: 

```
int foo(int x, y){  
  return (0+y);  
}
```

- Note that M1 is obtained by replacing the - operator by a + operator, M2 by replacing y by 0, and M3 by replacing x by 0.

# ERROR DETECTION USING MUTATION

Next we execute each mutant against tests in  $T$  until the mutant is distinguished or we have exhausted all tests. Here is what we get.

$$T = \{ t1: \langle x=1, y=0 \rangle, t2: \langle x=-1, y=0 \rangle \}$$

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1	1	1	1	0
t2	-1	-1	-1	0
		Live	Live	Killed

# ERROR DETECTION USING MUTATION

After executing all three mutants we find that two are live and one is distinguished. Computation of mutation score requires us to determine if any of the live mutants is equivalent.

*Determine whether or not the two live mutants are equivalent to foo and compute the mutation score of T.*



# ERROR DETECTION USING MUTATION

Let us examine the following two live mutants.

M1: 

```
int foo(int x, y){  
    return (x+y);  
}
```

M2: 

```
int foo(int x, y){  
    return (x-0);  
}
```

Let us focus on M1. A test that distinguishes M1 from **foo** must satisfy the following condition:

$x-y \neq x+y$  implies  $y \neq 0$ .

Hence we get t3:  $\langle x=1, y=1 \rangle$



# ERROR DETECTION USING MUTATION [7]

Executing **foo** on t3 gives us **foo**(t3)=0. However, according to the requirements we must get **foo**(t3)=2. Thus t3 distinguishes M1 from **foo** and also reveals the error. Correct the error and restart mutation testing from step 1

M1: 

```
int foo(int x, y){  
    return (x+y);  
}
```

M2: 

```
int foo(int x, y){  
    return (x-0);  
}
```