

Example on Mutation Testing

Dr. D. P. Mohapatra

Associate Professor

Department of CSE
National Institute of Technology, Rourkela-769008
Odisha



Example

Consider the following program. Generate five mutants and design test cases taking into account each mutant, using mutation testing. Calculate mutation score of your test suite.

```
main(){
float x,y,z;
printf("Enter values of three variables x,y,z");
scanf("%f%f%f",&x,&y,&z);
if(x>y){
if(x>z)
printf("%d is greatest",x);
else
printf("%d is greatest",z);
}
else{
if(y>z)
printf("%d is greatest",y);
else
printf("%d is greatest",z);
}
}
```



Solution

The instrumented program is given below.

```
1 main(){
2     float x,y,z;
3     printf("Enter values of three variables x,y,z");
4     scanf("%f%f%f",&x,&y,&z);
5     if(x>y){
6         if(x>z)
7             printf("%d is greatest",x);
8         else
9             printf("%d is greatest",z);
10    }
11 }
```



Test suite

Consider the test suite given in Table 1.

Table 1: Test suite for Q ??

Sl. No.	x	y	z	Expected Output
1	6	10	2	10
2	10	6	2	10
3	6	2	10	10
4	6	10	20	20



Mutated Statements

Table 2: Mutated statements

Mutant No.	Line No.	Original line	Modified line
M1	4	<code>if($x > y$)</code>	<code>if($x < y$)</code>
M2	4	<code>if($x > y$)</code>	<code>if($x > (y + z)$)</code>
M3	5	<code>if($x > z$)</code>	<code>if($x < z$)</code>
M4	8	<code>if($y > z$)</code>	<code>if($y = z$)</code>
M5	7	<code>printf("%d is greatest",z);</code>	<code>printf("%d is greatest",y);</code>



Actual Output of mutant M1

The mutated line numbers and changed lines are given in Table 2. The actual output obtained by executing the mutants M1-M5 is shown in Tables 3-7.

Table 3: Actual output of mutant M1

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	6
2	10	6	2	10	6
3	6	2	10	10	10
4	6	10	20	20	20



Actual output of mutant M2

Table 4: Actual output of mutant M2

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	10
2	10	6	2	10	10
3	6	2	10	10	10
4	6	10	20	20	20



Actual output of mutant M3

Table 5: Actual output of mutant M3

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	10
2	10	6	2	10	2
3	6	2	10	10	6
4	6	10	20	20	20



Actual output of mutant M4

Table 6: Actual output of mutant M4

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	10
2	10	6	2	10	10
3	6	10	2	10	10
4	6	10	20	20	10



Actual output of mutant M5

Table 7: Actual output of mutant M5

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	10
2	10	6	2	10	10
3	6	2	10	10	2
4	6	10	20	20	20



Additional test case

Table 8: Additional test case

Test case	x	y	z	Expected Output
5	10	5	6	10



Output of added test case

Table 9: Output of added test case

Test case	x	y	z	Expected output	Actual output
5	10	5	6	10	6



Revised Test suite

Table 10: Revised Test suite

Sl. No.	x	y	z	Expected Output
1	6	10	2	10
2	10	6	2	10
3	6	2	10	10
4	6	10	20	20
5	10	5	6	10



Mutation Score

Mutation score=Number of mutants killed/Total number of mutants=4/5=0.8

Higher the mutant score, better is the effectiveness of the test suite. The mutant M2 is live in the example. We may have to write a specific test case to kill this mutant. The additional test case is given in Table 8.

Now, when we execute test case 5, the actual output will be different from the expected output (Table 9), hence the mutant will be killed. This test case is very important and should be added to the given test suite. Therefore, the revised test suite is given in Table 10.



Software Testing

Dr. Durga Prasad Mohapatra
Professor
Department Of Computer Science &
Engineering
NIT Rourkela



Software Development Process

⌘ Software Development Life Cycle (or software development process):

↗ Series of identifiable stages that a software product undergoes during its life time:

☒ Feasibility study

☒ Requirements analysis and specification,

☒ Design,

☒ Coding,

☒ Testing

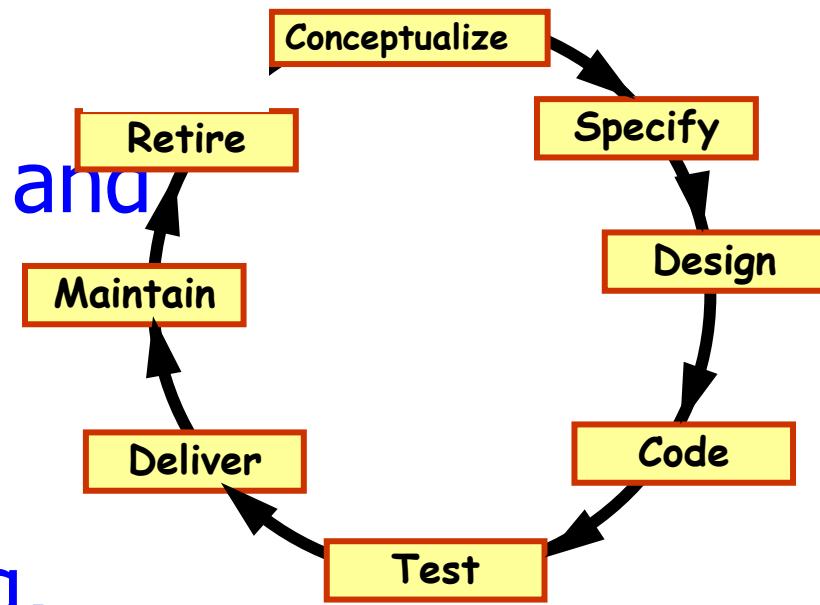
☒ Maintenance.

**Software Life
Cycle**

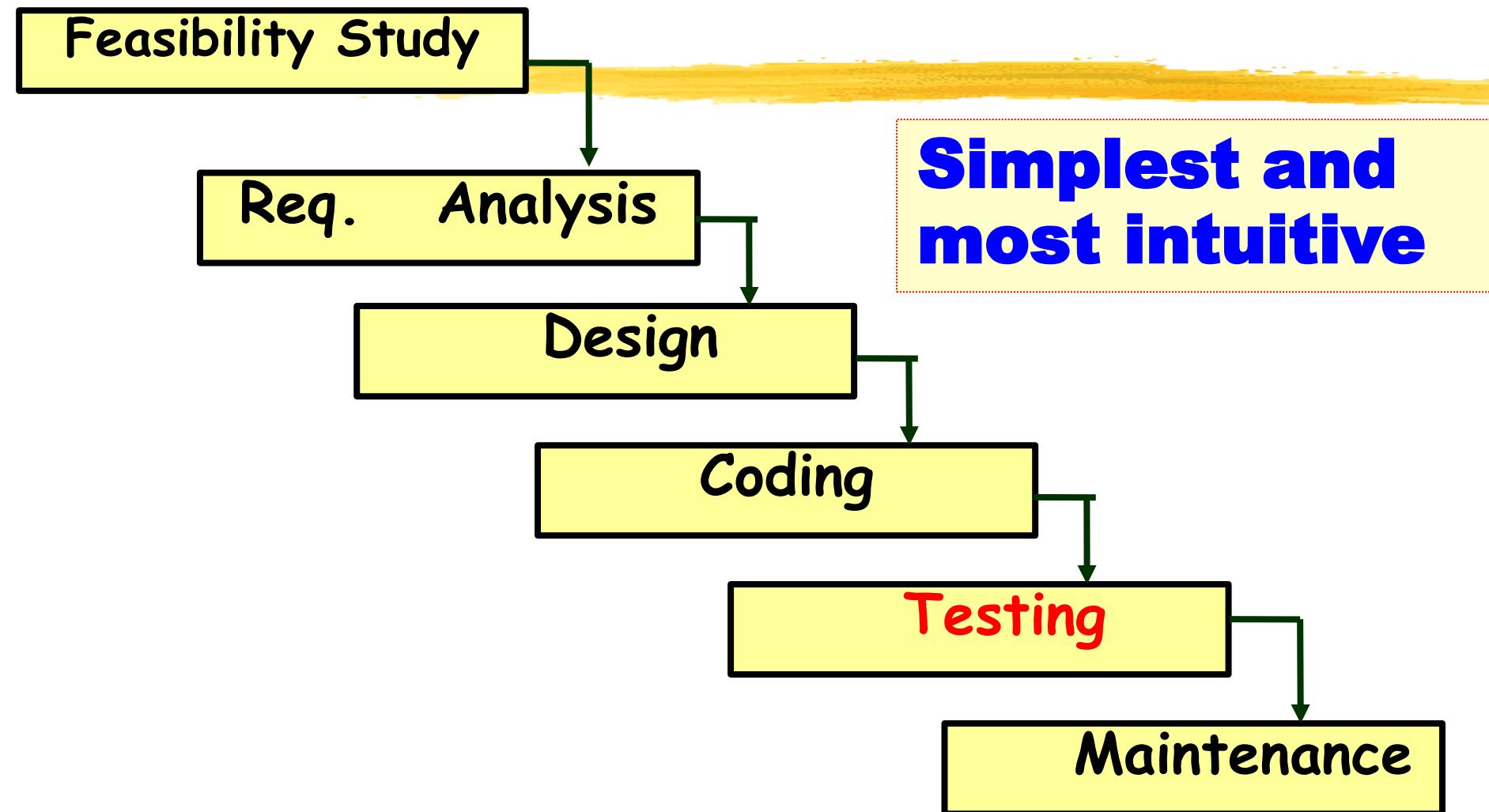
Classical Waterfall Model

⌘ Classical waterfall model divides life cycle into following phases:

- ▢ Feasibility study,
- ▢ Requirements analysis and specification,
- ▢ Design,
- ▢ Coding and unit testing,
- ▢ Integration and system testing,
- ▢ Maintenance.



Classical Waterfall Model



Defect Reduction Techniques



⌘ Review

⌘ Testing

⌘ Formal verification

⌘ Development process

⌘ Systematic methodologies

Why to 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

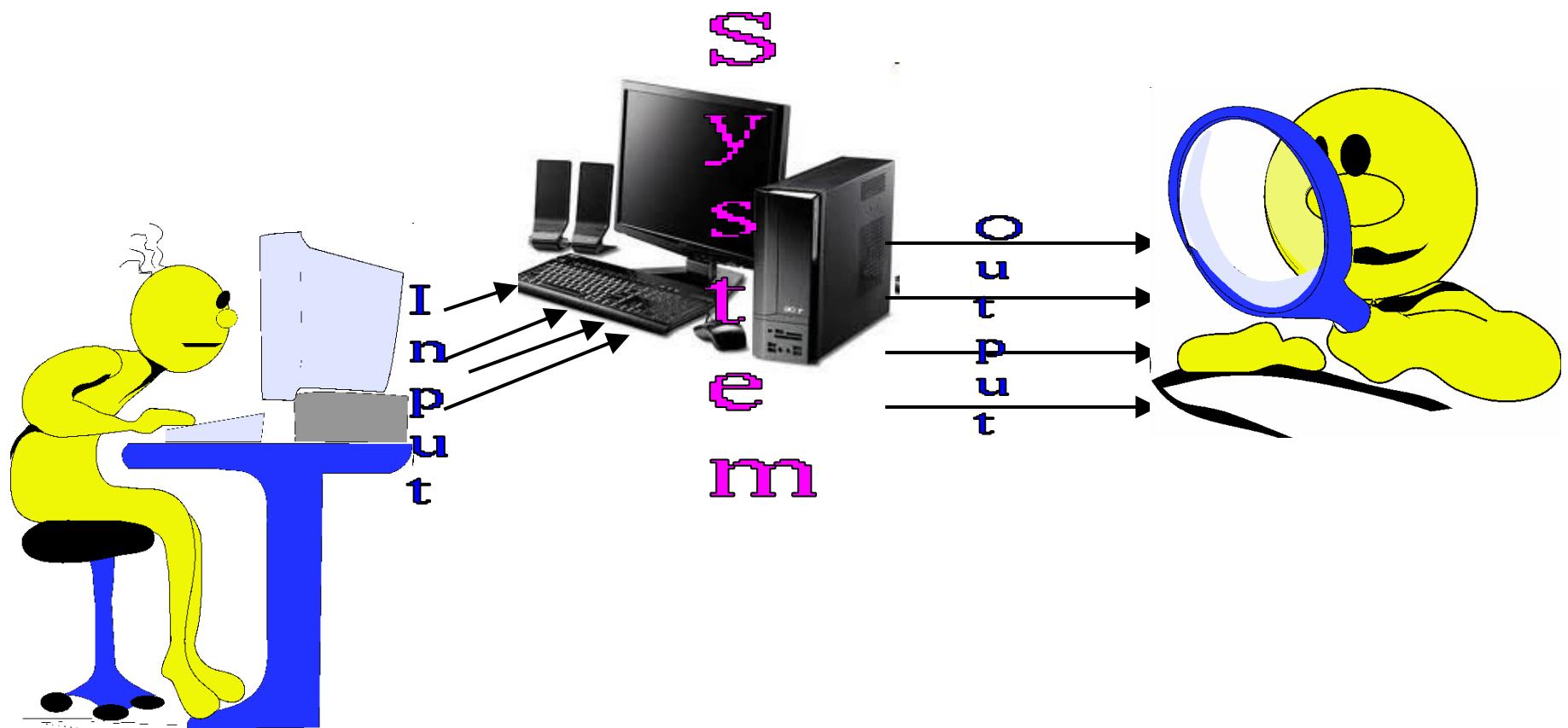
Organization of this lecture

- ⌘ Important concepts in program testing
- ⌘ Black-box testing:
 - ◻ equivalence partitioning
 - ◻ boundary value analysis
- ⌘ White-box testing
- ⌘ Debugging
- ⌘ Unit, Integration, and System testing
- ⌘ Summary

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.

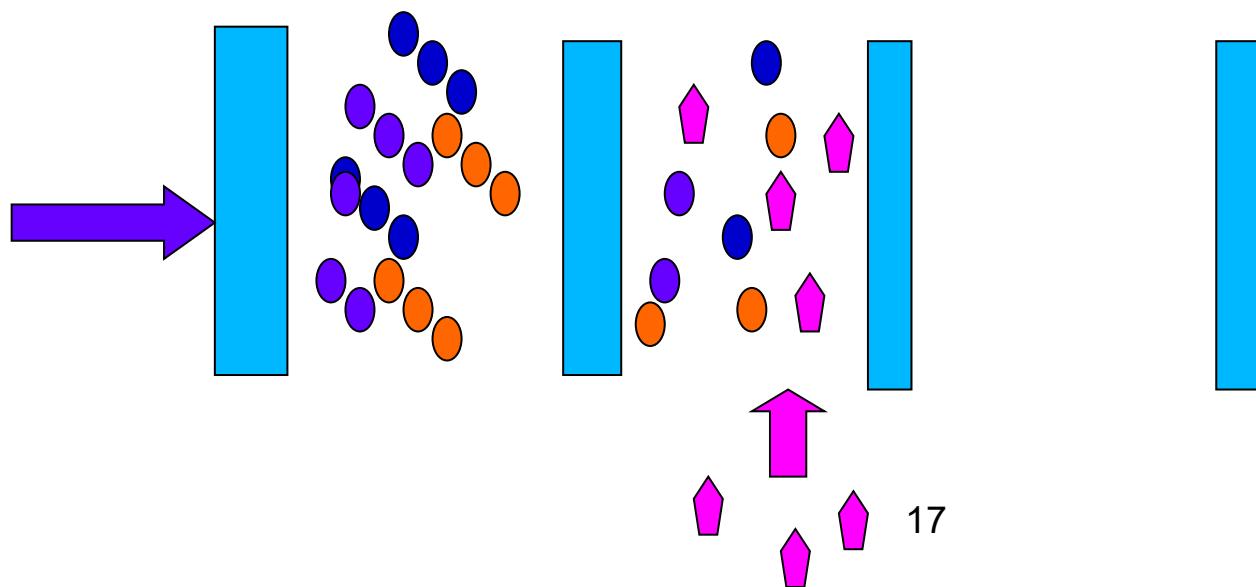
Error, Faults, and Failures

- ⌘ 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:

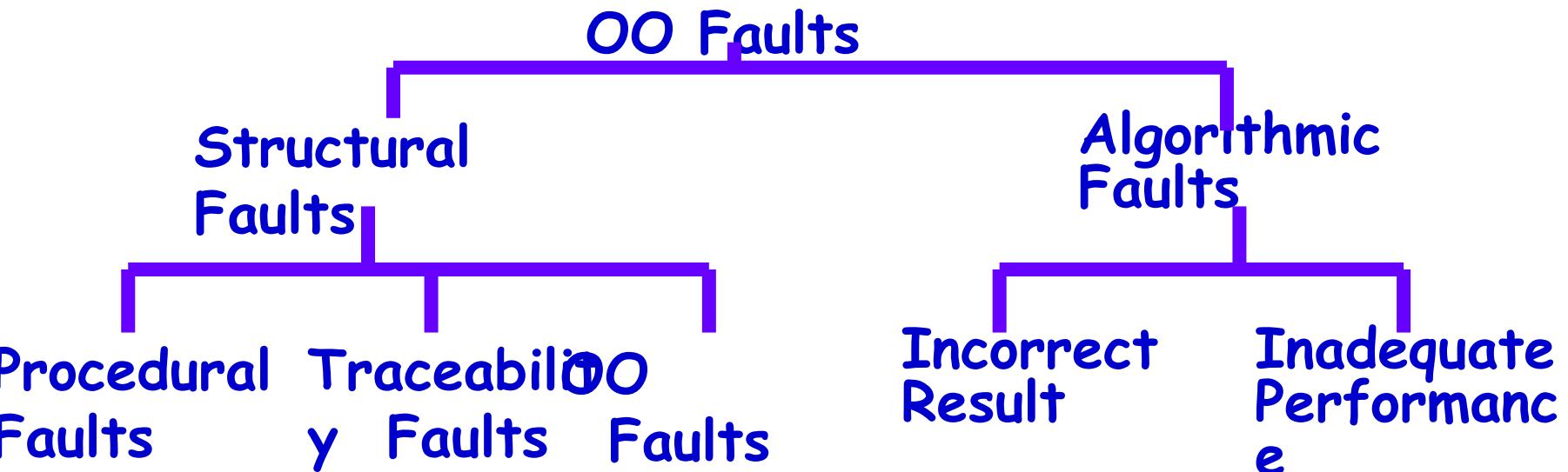
- └ Each detects only 70% bugs
- └ How many bugs would remain
- └ $1000 * (0.3)^4 = 81$ bugs

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

Black-box Testing

- ⌘ Test cases are designed using only **functional specification** of the software:
 - ↗ without any knowledge of the internal structure of the software / program.
- ⌘ For this reason, black-box testing is also known as [functional testing](#).

Black-box Testing



- ⌘ There are essentially two main approaches to design black box test cases:
 - ─ Equivalence class partitioning
 - ─ Boundary value analysis

Equivalence class partitioning

- ⌘ What does $a \equiv b \pmod{n}$ mean?
- ⌘ For a positive integer n , two integers a and b are said to be **congruent modulo n (or a is congruent to b modulo n)**, if a and b have the same remainder when divided by n (or equivalently if $a - b$ is divisible by n).
- ⌘ Congruence modulo n divides the set \mathbb{Z} of all integers into n subsets called **residue classes (equivalence classes)**.
- ⌘ It can be expressed as $a \equiv b \pmod{n}$, n is called the modulus.

Equivalence class partitioning

- # For example, if $n = 2$, then the two residue (equivalence) classes are the even integers and the odd integers.
- # So, $C1=\{0,2,4,6,8,10,\dots\}$
- # $C2=\{1,3,5,7,9,11,\dots\}$
- # If, $n = 3$, then the residue (equivalence) classes are as follows:
- # $C1=\{0,3,6,9, 12,\dots\}$
- # $C2=\{1,4,7,10,13,\dots\}$
- # $C3=\{2,5,8,11,14,\dots\}$

Equivalence Class Partitioning

- ⌘ Input values to a program are partitioned into equivalence classes.
- ⌘ Partitioning is done such that:
 - ↗ program behaves in similar ways to every input value belonging to an equivalence class.

Why define equivalence classes?

⌘ Test the code with just one representative value from each equivalence class:

◻ as good as testing using any other values from the equivalence classes.

Equivalence Class Partitioning

⌘ How do you determine the equivalence classes?

- ❑ examine the input data.
- ❑ few general guidelines for determining the equivalence classes can be given

Equivalence Class Partitioning

- ⌘ If the input data to the program is specified by a **range of values**:
 - ⏏ e.g. numbers between 1 to 5000.
 - ⏏ one valid and two invalid equivalence classes are defined.

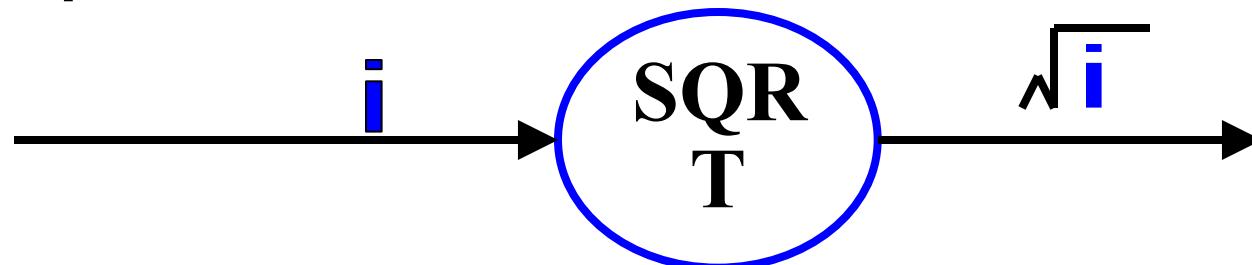


Equivalence Class Partitioning

- ⌘ If input is an enumerated set of values:
 - ─ e.g. {a,b,c}
 - ─ one equivalence class for valid input values
 - ─ another equivalence class for invalid input values should be defined.

Example

- ❖ A program reads an input value in the range of 1 and 5000:
 - ❖ computes the square root of the input number



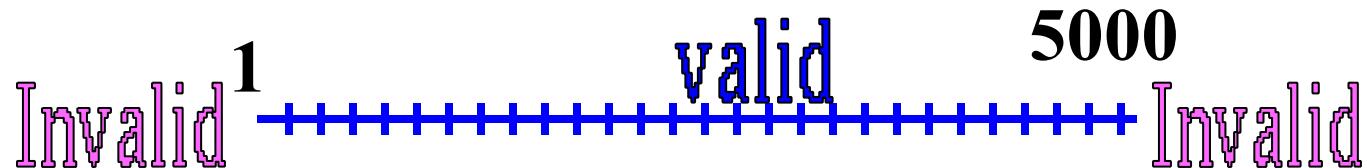
Example (cont.)

- ⌘ There are three equivalence classes:
 - ⌘ the set of negative integers,
 - ⌘ set of integers in the range of 1 and 5000,
 - ⌘ integers larger than 5000.



Example (cont.)

- ⌘ The test suite must include:
 - ◻ representatives from each of the three equivalence classes:
 - ◻ a possible test suite can be:
 $\{-5, 500, 6000\}$.



Example



⌘ A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Solution



- # 1. First we partition the domain of input as valid input values and invalid values, getting the following classes:
 - # $I_1 = \{<A, B, C> : 1 \leq A \leq 50\}$
 - # $I_2 = \{<A, B, C> : 1 \leq B \leq 50\}$
 - # $I_3 = \{<A, B, C> : 1 \leq C \leq 50\}$
 - # $I_4 = \{<A, B, C> : A < 1\}$

Solution



- # I5 = {<A, B, C> : A > 50}
- # I6 = {<A, B, C> : B < 1}
- # I7 = {<A, B, C> : B > 50}
- # I8 = {<A, B, C> : C < 1}
- # I9 = {<A, B, C> : C > 50}

Solution



⌘ Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class.

Solution

❖ The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

Solution



- # 2. We can derive another set of equivalence classes based on some possibilities for three integers, A, B, and C. These are given below:
 - # $I_1 = \{<A, B, C> : A > B, A > C\}$
 - # $I_2 = \{<A, B, C> : B > A, B > C\}$
 - # $I_3 = \{<A, B, C> : C > A, C > B\}$

Solution



- # I4 = { $\langle A, B, C \rangle : A = B, A \neq C$ }
- # I5 = { $\langle A, B, C \rangle : B = C, A \neq B$ }
- # I6 = { $\langle A, B, C \rangle : A = C, C \neq B$ }
- # I7 = { $\langle A, B, C \rangle : A = B = C$ }

Solution

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Example



- ⌘ A program determines the next date in the calendar. Its input is entered in the form of with the following range:
 - ⌘ $1 \leq mm \leq 12$
 - ⌘ $1 \leq dd \leq 31$
 - ⌘ $1900 \leq yyyy \leq 2025$

Example



⌘ Its output would be the next date or an error message 'invalid date.' Design test cases using equivalence class partitioning method.

Solution



- ⌘ First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:
- ⌘ $I_1 = \{ \langle m, d, y \rangle : 1 \leq m \leq 12 \}$
- ⌘ $I_2 = \{ \langle m, d, y \rangle : 1 \leq d \leq 31 \}$
- ⌘ $I_3 = \{ \langle m, d, y \rangle : 1900 \leq y \leq 2025 \}$
- ⌘ $I_4 = \{ \langle m, d, y \rangle : m < 1 \}$

Solution



⌘ I5 = { $\langle m, d, y \rangle : m > 12 \}$

⌘ I6 = { $\langle m, d, y \rangle : d < 1 \}$

⌘ I7 = { $\langle m, d, y \rangle : d > 31 \}$

⌘ I8 = { $\langle m, d, y \rangle : y < 1900 \}$

⌘ I9 = { $\langle m, d, y \rangle : y > 2025 \}$

Solution



⌘ The test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. The test cases are shown below:

Solution

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

Example



- ⌘ A program takes an angle as input within the range $[0, 360]$ and determines in which quadrant the angle lies. Design test cases using equivalence class partitioning method.

Solution



⌘ 1. First we partition the domain of input as valid and invalid values, getting the follow

⌘ $I_1 = \{ \langle \text{Angle} \rangle : 0 \leq \text{Angle} \leq 360 \}$

⌘ $I_2 = \{ \langle \text{Angle} \rangle : \text{Angle} < 0 \}$

⌘ $I_3 = \{ \langle \text{Angle} \rangle : \text{Angle} > 360 \}$

Solution



⌘ The test cases designed from these classes are shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	l_1
2	-1	Invalid input	l_2
3	361	Invalid input	l_3

Solution



- # 2. The classes can also be prepared based on the output criteria as shown below:
 - # O1 = {<Angle>: First Quadrant, if $0 \leq \text{Angle} \leq 90\}$
 - # O2 = {<Angle>: Second Quadrant, if $91 \leq \text{Angle} \leq 180\}$
 - # O3 = {<Angle>: Third Quadrant, if $181 \leq \text{Angle} \leq 270\}$

Solution



- # O4 = {<Angle>: Fourth Quadrant, if $271 \leq \text{Angle} \leq 360\}$ }
- # O5 = {<Angle>: Invalid Angle};
- # However, O5 is not sufficient to cover all invalid conditions this way. Therefore, it must be further divided into equivalence classes as shown in next slide:

Solution



- #O51 = {<Angle>: Invalid Angle, if Angle < 0}
- #O52 = {<Angle>: Invalid Angle, if Angle > 360}

Solution

Now the test cases can be designed from the above derived classes as shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	O_1
2	135	II Quadrant	O_2
3	250	III Quadrant	O_3
4	320	IV Quadrant	O_4
5	370	Invalid angle	O_{51}
6	-1	Invalid angle	O_{52}

Boundary Value Analysis

- ⌘ Some typical programming errors occur:
 - ⌘ at boundaries of equivalence classes
 - ⌘ might be purely due to psychological factors.
- ⌘ Programmers often fail to see:
 - ⌘ special processing required at the boundaries of equivalence classes.

Boundary Value Analysis



- ⌘ Programmers may improperly use < instead of <=
- ⌘ Boundary value analysis:
 - ↗ select test cases at the boundaries of different equivalence classes.

Example

- ⌘ For a function that computes the square root of an integer in the range of 1 and 5000:
 - ↗ test cases must include the values:
 $\{0, 1, 2, 4999, 5000, 5001\}$.



⌘ TS1 = {-5, 500, 6000} (EP)

⌘ TS2 = {0,1,2,4999,5000,5001} (BVA)

⌘ TS = TS1 U TS2

⌘ ={-5,0,1,2,500,4999,5000,5001, 6000}

More Examples on Testing



Ex:-1



❖ Q. Check if 2 Straight Lines Intersect and Print Their Point of Intersection

Ans: $y = mx + c$

Straight lines are given in the form of (m_1, c_1) and (m_2, c_2)

Step:1

Identify the Equivalent class

✓ Case 1: The lines are parallel i.e. $m_1 = m_2$

So points are $(1, 2)$ and $(1, 5)$

✓ Case 2: Coincident lines i.e. $m_1 = m_2$ and $c_1 = c_2$

And points are $(2, 3)$ and $(2, 3)$

Ex:-1 Contd.



$$m_1 \neq m_2$$

✓ Case 3: Lines intersecting at one point i.e.

The points may be (2, 5) and (3, 6).

So there are 3 equivalent classes.

There are no boundary values here.

So, Test Suite = $\{(1, 2), (1, 5), (2, 3), (2, 3), (2, 5), (3, 6)\}$

Ex:-2

- The Program Solves quadratic equations of the form

$$ax^2 + bx + c$$

It will accept 3 floating point values as input and it will give the roots e.g. The input may be (7.7, 3.3 and 4.5).

Ans: Equivalent Classes

I. $b^2 = 4ac$ inputs are: a= 2.0, b=4.0 and c=2.0

II. $b^2 > 4ac$ inputs are a=2.0, b= 5.0 and c=2.0

III. $b^2 < 4ac$ inputs are a=2.0, b= 3.0 and c=2.0

IV. Invalid Equation inputs are a=0, b= 0 and c=10.0

So, Test Suite = $\{(2.0,4.0,2.0),(2.0,5.0,2.0),(2.0,3.0,2.0),(0.0,0.0,10.0)\}$

Ex:-3



❖ Solves linear equations in upto 10 independent variables

$$\text{e.g. } 5x + 6y + z = 5$$

$$10x + 2y + 5z = 20$$

.....

.....

Example



❖ Equivalent Classes

I. Valid

- a. Many Solution ($\# \text{ var} < \# \text{eqns}$)
- b. No Solution ($\# \text{ var} > \# \text{eqns}$)
- c. Unique Solution ($\# \text{ var} = \# \text{eqns}$)

II. Invalid

- a. Too many Variables ($\# \text{ var} > 10$)
- b. Invalid Equation ($\# \text{ var} = 0$)

Ex:-3



⌘ Program Finds points of intersection of 2 circles

⌘ Equivalent Classes

- I. $r_1 + r_2 <$ distance between (x_1, y_1) and (x_2, y_2) i.e. not intersecting
- II. $r_1 + r_2 =$ distance i.e. touching at 1 point
- III. $r_1 + r_2 >$ distance i.e. intersecting at 2 points
- IV. Distance=0 and $r_1 = r_2$ i.e. overlapping
- V. Distance=0 and $r_1 \neq r_2$
- VI. Invalid circles

Example: Query Book Option in LIS

Example: Testing the option Query Book using a Keyword (e.g. Author name or title).

The equivalent classes are

- Not present in catalogue (SE, not present)
- Present in catalogue (SE, present, 15 issued, not available)
- Present in catalogue (SE, present, 10 issued, 5 available)

Black Box testing



- # Black-box testing attempts to find errors in the following categories:
- # 1. To test the modules independently .
- # 2. □ To test the functional validity of the software so that incorrect or missing functions can be recognized .
- # 3. □ To look for interface errors. □

Black Box testing



- ⌘ 4. To test the system behavior and check its performance ☐
- ⌘ 5. To test the maximum load or stress on the system.
- ⌘ 6. To test the software such that the user/customer accepts the system within defined acceptable limits.

BOUNDARY VALUE ANALYSIS (BVA)



- # BVA offers several methods to design test cases. Following are the few methods used:
 - # **1. BOUNDARY VALUE CHECKING (BVC)**
 - # **2. ROBUSTNESS TESTING METHOD**
 - # **3. WORST-CASE TESTING METHOD**
 - # **4. ROBUST WORST-CASE TESTING METHOD**

BOUNDARY VALUE CHECKING (BVC)



- ⌘ In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

- ⌘ The variable at its extreme value can be selected at:

BOUNDARY VALUE CHECKING (BVC)



- ⌘(a) Minimum value (Min)
- ⌘ (b) Value just above the minimum value (Min+)
- ⌘(c) Maximum value (Max)
- ⌘(d) Value just below the maximum value (Max-)

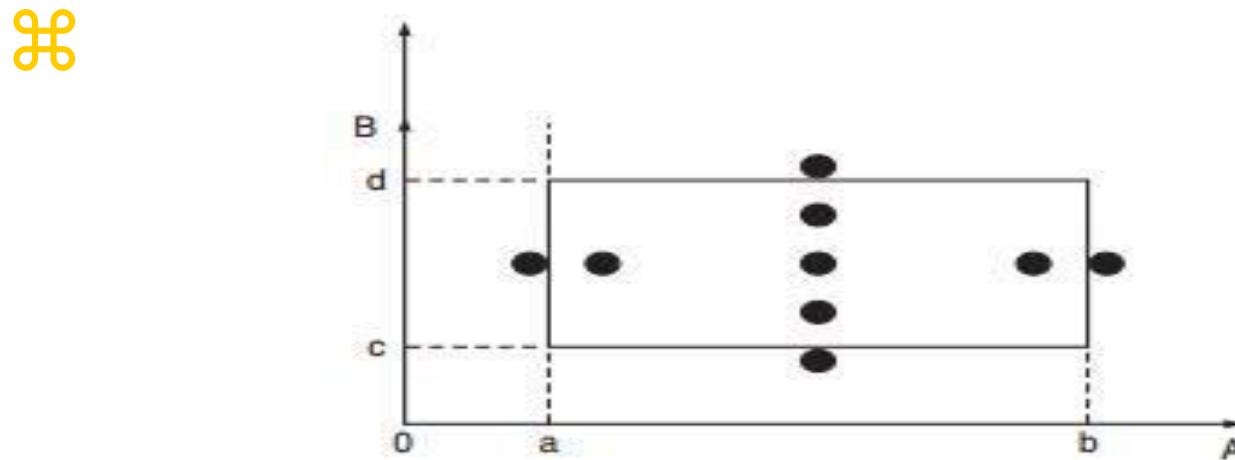
BOUNDARY VALUE CHECKING (BVC)

- ⌘ Let us take the example of two variables, A and B.
- ⌘ If we consider all the above combinations with nominal values, then following test cases (see Fig. 1) can be designed:
 - 1. Anom, Bmin
 - 2. Anom, Bmin+
 - 3. Anom, Bmax
 - 4. Anom, Bmax-
 - 5. Amin, Bnom
 - 6. Amin+, Bnom

BOUNDARY VALUE CHECKING (BVC)

- ⌘ 7. Amax, Bnom
- 9. Anom, Bnom

- 8. Amax-, Bnom



⌘ Fig: Boundary Value Checking

BOUNDARY VALUE CHECKING (BVC)



⌘ It can be generalized that for n variables in a module, $4n + 1$ test cases can be designed with boundary value checking method.

ROBUSTNESS TESTING METHOD



- ⌘ The idea of BVC can be extended such that boundary values are exceeded as: □
 - ⌘ 1. A value just greater than the Maximum value (Max+)
 - ⌘ 2. A value just less than Minimum value (Min-)

ROBUSTNESS TESTING METHOD



- ⌘ When test cases are designed considering the above points in addition to BVC, it is called robustness testing.
- ⌘ Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:
 - ⌘ 10. Amax+, Bnom 11. Amin-, Bnom

ROBUSTNESS TESTING METHOD



- ⌘ 12. Anom, Bmax+ 13. Anom, Bmin-
- ⌘ It can be generalized that for n input variables in a module, $6n + 1$ test cases can be designed with robustness testing.

WORST-CASE TESTING METHOD



- ⌘ We can again extend the concept of BVC by assuming more than one variable on the boundary.
- ⌘ It is called worst-case testing method.
- ⌘ Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

WORST-CASE TESTING METHOD



- ⌘ 10. Amin, Bmin
- 12. Amin, Bmin+
- 14. Amax, Bmin
- 16. Amax, Bmin+
- 18. Amin, Bmax
- 20. Amin, Bmax-
- 22. Amax, Bmax
- 24. Amax, Bmax-

- 11. Amin+, Bmin
- 13. Amin+, Bmin+
- 15. Amax-, Bmin
- 17. Amax-, Bmin+
- 19. Amin+, Bmax
- 21. Amin+, Bmax-
- 23. Amax-, Bmax
- 25. Amax-, Bmax-

WORST-CASE TESTING METHOD



- ⌘ It can be generalized that for n input variables in a module, $5n$ test cases can be designed with worst-case testing.

ROBUST WORST-CASE TESTING METHOD



- ⌘ In the previous method, the extreme values of a variable considered are of BVC only.
- ⌘ The worst case can be further extended if we consider robustness also, that is, in worst case testing if we consider the extreme values of the variables as in robustness testing method covered in Robustness Testing

ROBUST WORST-CASE TESTING METHOD



⌘ Again take the example of two variables, A and B. We can add the following test cases to the list of 25 test cases designed in previous section.

⌘ 26. Amin-, Bmin-

28. Amin, Bmin-

⌘ 27. Amin-, Bmin

29. Amin-, Bmin+

⌘ 30. Amin+, Bmin-

31. Amin-, Bmax



Solution



- ⌘ 32. Amax, Bmin-
- ⌘ 34. Amax-, Bmin-
- ⌘ 36. Amax+, Bmin
- ⌘ 38. Amax+, Bmin+
- ⌘ 40. Amax+, Bmax
- ⌘ 42. Amax+, Bmax-
- ⌘ 44. Amax+, Bnom

- 33. Amin-, Bmax-
- 35. Amax+, Bmax+
- 37. Amin, Bmin+
- 39. Amax+, Bmax+
- 41. Amax, Bmax+
- 43. Amax-, Bmax+
- 45. Anom, Bmax+

Solution



⌘ 46. Amin-, Bnom

⌘ 48. Amax+, Bmin-

47. Anom, Bmin-

49. Amin-, Bmax+

STATE TABLE – BASED TESTING



- ⌘ Tables are useful tools for representing and documenting many types of information relating to test case design.
- ⌘ Theses are beneficial for applications which can be described using state transition diagrams and state tables.

Basic terms related to State Table

1. Finite State Machine (FSM)

- ⌘ An FSM is a behavioral model whose outcome depends upon both the previous and current inputs.
- ⌘ This model can be prepared for software structure or software behavior.
- ⌘ It can be used as a tool for functional testing.

2. State Transition Diagrams or State Graph



- # A system or its components may have a number of states depending on its input and time.
- # States are represented by nodes.
- # *With the help of nodes and transition links between nodes, a STD or SG can be prepared.*
- # A state graph is the pictorial representation of an FSM.
- # Its purpose is to depict the states that a system or its components can assume.
- # It shows the events or circumstances that cause or result from a change from one state to another.

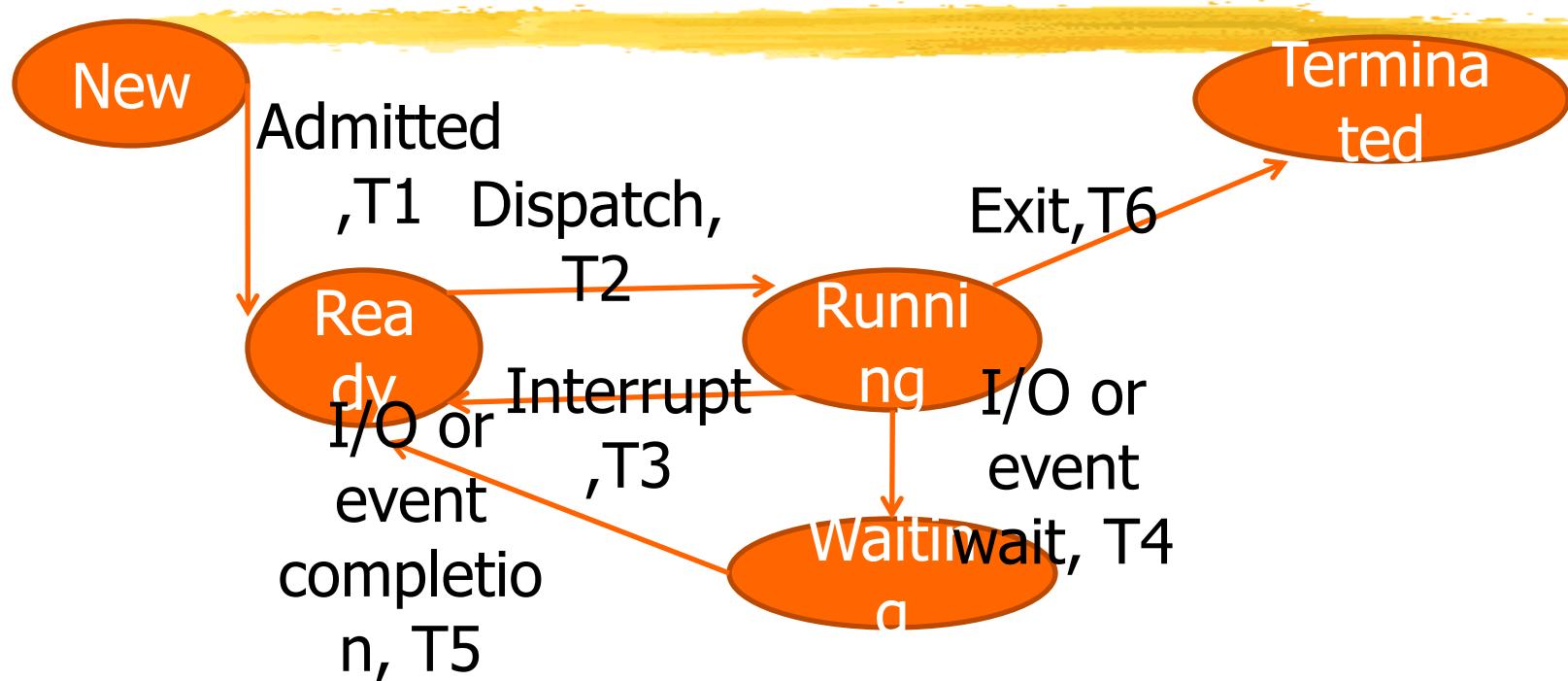
- 
- ⌘ Whatever is being modeled is subjected to inputs.
 - ⌘ As a result of these inputs, when one state is changed to another is called a *transition*.
 - ⌘ Transitions are represented by links that join the nodes.



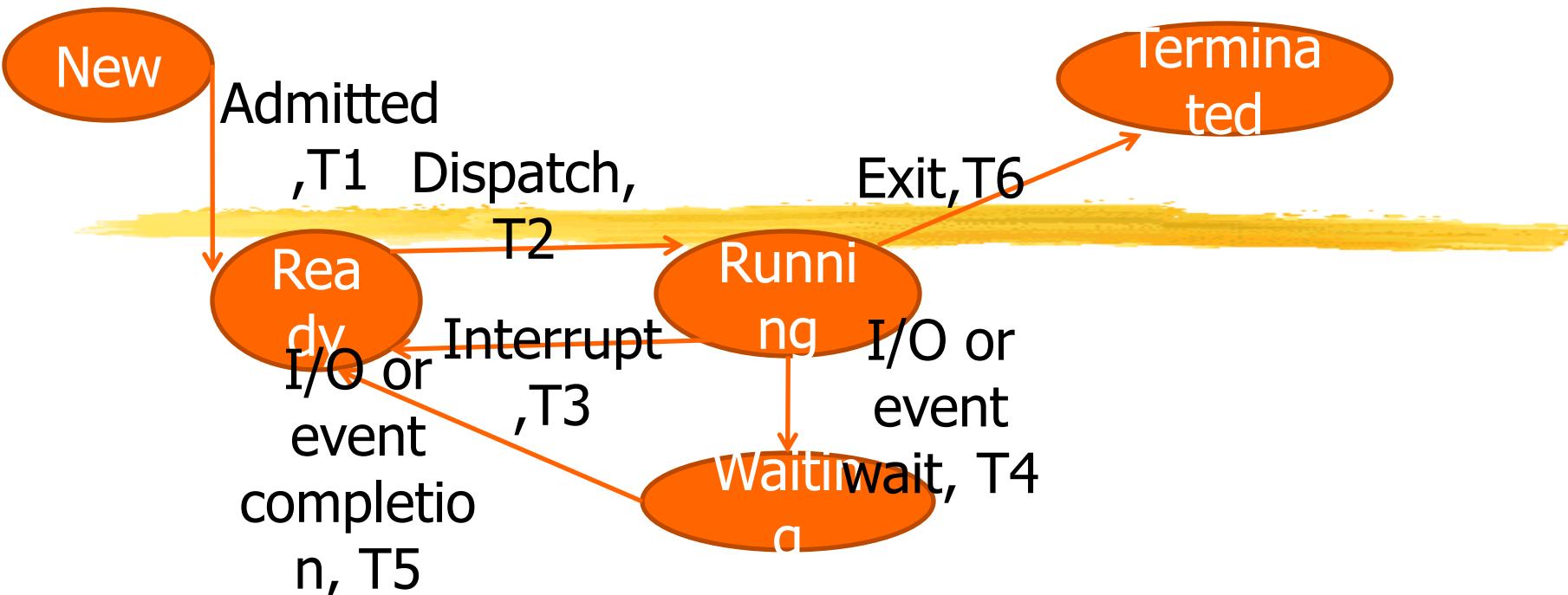
For example, a task in an O. S. can have its following states:

- i. **New State** : When a task is newly created
- ii. **Ready** : When the task is waiting in the ready queue for its turn.
- iii. **Running** : When Instructions of the task are being executed by CPU.
- iv. **Waiting** : When the task is waiting for an I/O event or reception of a signal
- v. **Terminated** : The task has finished execution.

State Graph



- i. New State : When a task is newly created
- ii. Ready : When the task is waiting in the ready queue for its turn.
- iii. Running : When Instructions of the task are being executed by CPU.
- iv. Waiting : When the task is waiting for an I/O event or reception of a signal
- v. Terminated : The task has finished execution.



Each arrow link provides two types of Information :

1. Transition events like admitted, dispatch, interrupt, etc.
2. The resulting output from a state like T1, T2, T3 etc.

T0=Task is in new state and waiting for admission to ready queue.

T1= A new task admitted to ready queue

T2= A ready task has started running

T3= Running task has been interrupted

T4= Running task is waiting for I/O or event

T5= Wait period of waiting task is over

T6= Task has completed execution

3. State Table

State/input Event	Admit	Dispatch	Interrupt	I/O or event Wait	I/O or event Wait Over	Exit
New	Ready/T1	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready / T1	Running /T2	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running /T2	Running /T2	Ready / T3	Waiting/T4	Running/T2	Terminated/T6
Waiting	Waiting/T4	Waiting/T4	Waiting/T4	Waiting/T4	Ready /T5	Waiting/T4

- Each row of the table corresponds to a state.
- Each column corresponds to an input condition
- The box at the intersection of a row and a column specifies the next state (transition) and the outputs, if any.

4. State Table-Based Testing



⌘ After reviewing the basics, we can start functional testing with state tables.

Steps:

1. Identify the states

The number of states in a state graph is the number of states we choose to recognize or model.



Find the number states as follows :

- ⌘ Identify all the component factors of the state.
- ⌘ Identify all the allowable values for each factor
- ⌘ The number of states is the product of the number of allowable values of all the factors.



2. Prepare state transition diagram after understanding transitions between states

- ⌘ After having all the states, identify the inputs on each state and transitions between states and prepare the state graph.
- ⌘ Every input state combination must have a specified transition.

- 
3. Convert the state graph into the state table as discussed earlier
 4. Analyze the state table for its completeness.

5. Create the corresponding test cases from the state table

Test case ID	Test Source	Input		Expected results	
		Current State	Event	Output	Next state
TC1	Cell 1	New	Admit	T1	Ready
TC2	Cell 2	New	Dispatch	T0	New
TC3	Cell 3	New	Interrupt	T0	New
TC4	Cell 4	New	I/O wait	T0	New
TC5	Cell 5	New	I/O wait over	T0	New
TC6	Cell 6	New	Exit	T0	New
TC7	Cell 7	Ready	Admit	T1	Ready
TC8	Cell 8	Ready	Dispatch	T2	Running
TC9	Cell 9	Ready	Interrupt	T1	Ready
TC10	Cell 10	Ready	I/O wait	T1	Ready
TC11	Cell 11	Ready	I/O wait over	T1	Ready
TC12	Cell 12	Ready	Exit	T1	Ready
TC13	Cell 13	Running	Admit	T2	Running
TC14	Cell 14	Running	Dispatch	T2	Running
TC15	Cell 15	Running	Interrupt	T3	Ready
TC16	Cell 16	Running	I/O wait	T4	Waiting
TC17	Cell 17	Running	I/O wait over	T2	Running
TC18	Cell 18	Running	Exit	T6	Terminated
TC19	Cell 19	Waiting	Admit	T4	Waiting
TC20	Cell 20	Waiting	Dispatch	T4	Waiting
TC21	Cell 21	Waiting	Interrupt	T4	Waiting
TC22	Cell 22	Waiting	I/O wait	T4	Waiting
TC23	Cell 23	Waiting	I/O wait over	T5	Ready
TC24	Cell 24	Waiting	Exit	T4	Waiting

- Test cases are produced in a tabular form known as the test case.
- **Test cases ID :** a unique identifier for each test case
- **Test Source :** a trace back to the corresponding cell in the state table.
- **Current state :** the initial condition to run the test
- **Event :** the input triggered by the user
- **Output :** the current value returned
- **Next State :** the new state achieved

Decision Table – Based Testing

- ⌘ Boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions.
- ⌘ There may be some critical behaviour to be tested when some combinations of input conditions are considered.
- ⌘ Decision table is another useful method to represent the information in a tabular method.
- ⌘ Decision table has the specialty to consider complex combinations of input conditions and resulting actions.
- ⌘ Decision tables obtain their power from logical expressions.
- ⌘ Each operand or variable in a logical expression takes on the value, TRUE or FALSE

Formation of Decision Table

Condition Stub		Rule 1	Rule 2	Rule 3	Rule 4	...
	C1 C2 C3	True False True	True True True	False False True	I True I	
Action Stub	A1 A2 A3	X	X	X	X	

Condition stub It is a list of input conditions for which the complex combination is made.

Action stub It is a list of resulting action which will be performed if a combination of input condition is satisfied.

Condition entry

- It is a specific entry in the table corresponding to input conditions mentioned in the condition stub.
- When the condition entry takes only two values – TRUE or FALSE then it is called **Limited Entry Decision Table**.
- When the condition entry takes several values , then it is called **Extended Entry Decision Table**.

Action entry It is the entry in the table for the resulting action to be performed

- List all actions that can be associated with a specific procedure.
- List all conditions during execution of the procedure.
- Associate specific sets of conditions with specific actions, eliminating impossible combinations and conditions; alternatively, develop every possible permutation of conditions.
- Define rules by indicating what action occurs for a set of conditions.

Test Case Design using Decision Table



- Interpret condition stubs as the inputs for the test case.
- Interpret action stubs as the expected output for the test case.
- Rule, which is the combination of input conditions, becomes the test case itself.
- If there are k rules over n binary conditions, there are at least k test cases and at the most 2^n test cases.

Example

A programme calculates the total salary of an employee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design test cases using decision table testing.

Solution

Entry

		Rule 1	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48 C2: Holidays or Sundays	I T	F F	T F
Action Stub	A1: Normal Salary A2: 1.25 of salary A3: 2.00 of salary	X	X	X

Decision Table

Test case ID	Working hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

Expanding the Immaterial Cases in Decision Table



- ⌘ These conditions means that the value of a particular condition in the specific rule does not make a difference whether it is TRUE or FALSE.
- ⌘ Sometimes expanding the decision table to spell out don't-care conditions can reveal hidden problems.

Example



Entry (Decision table)

		Rule 1	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48 C2: Holidays or Sundays	I T	F F	T F
Action Stub	A1: Normal Salary A2: 1.25 of salary A3: 2.00 of salary	X	X	X

The immaterial test case in rule 1 of the above table can be expanded by taking both T and F values of C1.

Entry (Expanded decision table)

		Rule 1-1	Rule 1-2	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48 C2: Holidays or Sundays	F T	T T	F F	T F
Action Stub	A1: Normal Salary A2: 1.25 of salary A3: 2.00 of salary	X	X	X	X

Entry (Expanded decision table)

Test case ID	Working hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary
4	30	Sunday	2.00 of salary

Orthogonal Array Testing



⌘ Orthogonal Arrays are two dimensional arrays of numbers that have the attractive feature that by selecting any two columns in the array, an even distribution of all pairwise combinations of values in the array can be achieved.

Example



- ⌘ In the following table , after selecting first two columns, we get four ordered pairs namely (0,0), (1,1), (0,1) and (1,0).
- ⌘ These pairs form all the possible ordered pairs of two-element set and each ordered pair appears exactly once.

Example

0	0	0
1	1	0
0	1	1
1	0	1

We obtain same values when selecting second and third or first and third column combination. An array exhibiting this feature is known as orthogonal array.

Orthogonal Array Testing



- ⌘ Orthogonal arrays can be used in software testing for pairwise interactions.
- ⌘ It provides uniformly distributed coverage for all variable pairwise combinations.
- ⌘ It is commonly used for integration testing like object oriented systems, where multiple subclasses can be substituted as the server for a client.

Orthogonal Array Testing



- # It is black-box testing technique.
- # OATS is used when the input to the system to be tested are low but if exhaustive testing is used then it is not possible to test completely every input of the system.
- # 100% OATS implies 100% pairwise testing.
- # OATS can be used for testing combinations of configurable options like a webpage that allows the other user to select:

Orthogonal Array Testing



- # Font style;
- # Font color;
- # Back ground Color;
- # Page layout;
- # Etc.

Steps to use OATS



- # Step 1: Identify the independent variables that are to be used for interaction. These will be mapped as “factor” (f) of array.
- # Step 2: Identify the maximum number of values, which each variable will take. There will be mapped as “levels” (p) of the array.
- # Search for an orthogonal array that has all factors from step 1 and all levels from step 2.

Steps to use OATS



- # Step 4: Map all the factors and levels with your requirements.
- # Translate them into suitable test cases.
- # Look out for any special test cases.
- # If we have 3 variables (parameters) that we have 3 value then the possible number of test cases using conventional technique is $3*3*3=27$ but if OATS is used, then number of test cases will be 9.

Example 1



- # Consider a scenario in which we need to derive test cases for a web page of a research paper that has four different sections:
 - #(a) Abstract
 - #(b) Related work
 - #(c) Proposed work
 - #(d) Conclusion

Example 1



⌘ The four section can be individually shown or hidden to the user or show message. Thus it is required to design the test condition to test interaction between different sections

Solution



- ⌘ Step 1: Number of independent variables or factors = 4.
- ⌘ Step 2: Value that each independent variable can take = 3 values (shown , hidden or error message).
- ⌘ Step 3: Orthogonal array would be 3^2 .
- ⌘ Step 4: The appropriate orthogonal array for 4 factors and 3 levels is shown in the table.

Solution



Experiment No.	Factor A	Factor B	Factor C	Factor D
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Solution



- ⌘ Step 5: Now, map the table with our requirements.
- ⌘ 1 will represent “Shown” value.
- ⌘ 2 will represent “Hidden” value.
- ⌘ 3 will represent “Error Message” value.
- ⌘ Factor A will represent “Abstract” section.
- ⌘ Factor B will represent “Related Work” section.
- ⌘ Factor C will represent “Proposed Work” section.

Solution



- # Factor A will represent “Conclusion” section.
- # Experiment will represent “Test Case #”.
- # Step 6: After mapping, the table will look like:

Solution



Test Case	Abstract	Related Work	Proposed work	Conclusion
Test Case 1	Shown	Shown	Shown	Shown
Test Case 2	Shown	Hidden	Hidden	Hidden
Test Case 3	Shown	Error Message	Error Message	Error Message
Test Case 4	Hidden	Shown	Hidden	Error Message
Test Case 5	Hidden	Hidden	Error Message	Shown
Test Case 6	Hidden	Error Message	Shown	Hidden
Test Case 7	Error Message	Shown	Error Message	Hidden
Test Case 8	Error Message	Hidden	Shown	Error Message
Test Case 9	Error Message	Error Message	Hidden	Shown

Debugging

- ⌘ Once errors are identified:
 - ↗ it is necessary identify the precise location of the errors and to fix them.
- ⌘ Each debugging approach has its own advantages and disadvantages:
 - ↗ each is useful in appropriate circumstances.

Brute-force method

- ⌘ This is the most common method of debugging:
 - ⌘ least efficient method.
 - ⌘ program is loaded with print statements
 - ⌘ print the intermediate values
 - ⌘ hope that some of printed values will help identify the error.

Symbolic Debugger

- ⌘ Brute force approach becomes more systematic:
 - ⌘ with the use of a symbolic debugger,
 - ⌘ symbolic debuggers get their name for historical reasons
 - ⌘ early debuggers let you only see values from a program dump:
 - ✗ determine which variable it corresponds to.

Symbolic Debugger

⌘ Using a symbolic debugger:

- ▢ values of different variables can be easily checked and modified
- ▢ single stepping to execute one instruction at a time
- ▢ **break points** and **watch points** can be set to test the values of variables.

Backtracking

- ⌘ This is a fairly common approach.
- ⌘ Beginning at the statement where an error symptom has been observed:
 - ↗ source code is traced backwards until the error is discovered.

Example

```
int main(){  
    int i,j,s;  
    i=1;  
    while(i<=10){  
        s=s+i;  
        i++; j=j++;}  
    printf("%d",s);  
}
```

Backtracking

- ⌘ Unfortunately, as the number of source lines to be traced back increases,
 - ⌘ the number of potential backward paths increases
 - ⌘ becomes unmanageably large for complex programs.

Cause-elimination method

- ⌘ Determine a list of causes:
 - ◻ which could possibly have contributed to the error symptom.
 - ◻ tests are conducted to eliminate each.
- ⌘ A related technique of identifying error by examining error symptoms:
 - ◻ software fault tree analysis.

Program Slicing

- ⌘ This technique is similar to back tracking.
- ⌘ However, the search space is reduced by defining slices.
- ⌘ A slice is defined for a particular variable at a particular statement:
 - ↗ set of source lines preceding this statement which can influence the value of the variable.

Example

```
int main(){
    int i,s;
    i=1; s=1;
    while(i<=10){
        s=s+i;
        i++;}
    printf("%d",s);
    printf("%d",i);
}
```

Debugging Guidelines

- ⌘ Debugging usually requires a thorough understanding of the program design.
- ⌘ Debugging may sometimes require full redesign of the system.
- ⌘ A common mistake novice programmers often make:
 - ☒ not fixing the error but the error symptoms.

Debugging Guidelines

- ⌘ Be aware of the possibility:
 - ↗ an error correction may introduce new errors.
- ⌘ After every round of error-fixing:
 - ↗ regression testing must be carried out.

Program Analysis Tools

⌘ An automated tool:

- ▢ takes program source code as input
- ▢ produces reports regarding several important characteristics of the program,
- ▢ such as size, complexity, adequacy of commenting, adherence to programming standards, etc.

Program Analysis Tools



- ⌘ Some program analysis tools:
 - ↗ produce reports regarding the adequacy of the test cases.
- ⌘ There are essentially two categories of program analysis tools:
 - ↗ Static analysis tools
 - ↗ Dynamic analysis tools

Static Analysis Tools

⌘ Static analysis tools:

- ▢ assess properties of a program without executing it.
- ▢ Analyze the source code
- ▢ provide analytical conclusions.

Static Analysis Tools

- ⌘ Whether coding standards have been adhered to?
 - ↗ Commenting is adequate?
- ⌘ Programming errors such as:
 - ↗ uninitialized variables
 - ↗ mismatch between actual and formal parameters.
 - ↗ Variables declared but never used, etc.

Static Analysis Tools

- ⌘ Code walk through and inspection can also be considered as static analysis methods:
- ⚠ however, the term static program analysis is generally used for automated analysis tools.

Dynamic Analysis Tools

- ⌘ Dynamic program analysis tools require the program to be executed:
 - ⌘ its behavior recorded.
 - ⌘ Produce reports such as adequacy of test cases.

Testing



- ⌘ The aim of testing is to identify all defects in a software product.
- ⌘ However, in practice even after thorough testing:
 - ↗ one cannot guarantee that the software is error-free.

Testing

⌘ The input data domain of most software products is very large:

⚠ it is not practical to test the software exhaustively with each input data value.

Testing

⌘ Testing does however expose many errors:

- ◻ testing provides a practical way of reducing defects in a system
- ◻ increases the users' confidence in a developed system.

Testing

- ⌘ Testing is an important development phase:
 - ⌈ requires the maximum effort among all development phases.
- ⌘ In a typical development organization:
 - ⌈ maximum number of software engineers can be found to be engaged in testing activities.

Testing

⌘ Many engineers have the wrong impression:

- ☐ testing is a secondary activity
- ☐ it is intellectually not as stimulating as the other development activities, etc.

Testing

- ⌘ Testing a software product is in fact:
 - (^)(as much challenging as initial development activities such as specification, design, and coding.)
- ⌘ Also, testing involves a lot of creative thinking.

Testing

❖ Software products are tested at three levels:

- ❖ Unit testing
- ❖ Integration testing
- ❖ System testing

Unit testing

❖ During unit testing, modules are tested in isolation:

- ◻ If all modules were to be tested together:
 - ☒ it may not be easy to determine which module has the error.

Unit testing

 Unit testing reduces debugging effort several folds.

 Programmers carry out unit testing immediately after they complete the coding of a module.

Integration testing

- ⌘ After different modules of a system have been coded and unit tested:
 - ─ modules are integrated in steps according to an integration plan
 - ─ partially integrated system is tested at each integration step.

System Testing



- ⌘ System testing involves:
 - ☐ validating a fully developed system against its requirements.

Integration Testing

- ❖ Develop the integration plan by examining the structure chart :
 - ❖ big bang approach
 - ❖ top-down approach
 - ❖ bottom-up approach
 - ❖ mixed approach

Example Structured Design

root

Valid-numbers

rms

Valid-numbers

Get-good-data

Compute-solution

rms

Display-solution

Get-data

Validate-data

Big bang Integration Testing

⌘ Big bang approach is the simplest integration testing approach:

- ─ all the modules are simply put together and tested.
- ─ this technique is used only for very small systems.

Big bang Integration Testing

⌘ Main problems with this approach:

◻ if an error is found:

- ☒ it is very difficult to localize the error
- ☒ the error may potentially belong to any of the modules being integrated.

◻ debugging errors found during big bang integration testing are very expensive to fix.

Bottom-up Integration Testing

⌘ Integrate and test the bottom level modules first.

⌘ A disadvantage of bottom-up testing:

- ◻ when the system is made up of a large number of small subsystems.
- ◻ This extreme case corresponds to the big bang approach.

Top-down integration testing



- ⌘ Top-down integration testing starts with the main routine:
 - ↗ and one or two subordinate routines in the system.
- ⌘ After the top-level 'skeleton' has been tested:
 - ↗ immediate subordinate modules of the 'skeleton' are combined with it and tested.

Mixed integration testing



❖ Mixed (or sandwiched) integration testing:

- ❑ uses both top-down and bottom-up testing approaches.
- ❑ Most common approach

Integration Testing

⌘ In top-down approach:

◻ testing waits till all top-level modules are coded and unit tested.

⌘ In bottom-up approach:

◻ testing can start only after bottom level modules are ready.

System Testing

⌘ There are three main kinds of system testing:

- ─ Alpha Testing
- ─ Beta Testing
- ─ Acceptance Testing

Alpha Testing



⌘ System testing is carried out by the test team within the developing organization.

Beta Testing



⌘ System testing performed by a select group of friendly customers.

Acceptance Testing



- ⌘ System testing performed by the customer himself:
 - ↗ to determine whether the system should be accepted or rejected.

Stress Testing



- ⌘ Stress testing (aka endurance testing):
 - ⇨ impose abnormal input to stress the capabilities of the software.
 - ⇨ Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

How many errors are still remaining?



- ⌘ Seed the code with some known errors:
 - ─ Artificial errors are introduced into the program.
 - ─ Check how many of the seeded errors are detected during testing.

Error Seeding

⌘ Let:

- ⌘ N be the total number of errors in the system
- ⌘ n of these errors be found by testing.
- ⌘ S be the total number of seeded errors,
- ⌘ s of the seeded errors be found during testing.

Error Seeding



⌘ $n/N = s/S$

⌘ $N = S n/s$

⌘ remaining defects:

$$N - n = n \left((S - s) / s \right)$$

Example



- ⌘ 100 errors were introduced.
- ⌘ 90 of these errors were found during testing
- ⌘ 50 other errors were also found.
- ⌘ Remaining errors =
$$50 \ (100-90)/90 = 6$$

Error Seeding



- ⌘ The kind of seeded errors should match closely with existing errors:
 - ↗ However, it is difficult to predict the types of errors that exist.
- ⌘ Categories of remaining errors:
 - ↗ can be estimated by analyzing historical data from similar projects.

Summary



⌘ Exhaustive testing of almost any non-trivial system is impractical.

↗ we need to design an optimal test suite that would expose as many errors as possible.

Summary



- ⌘ If we select test cases randomly:
 - ─ many of the test cases may not add to the significance of the test suite.
- ⌘ There are two approaches to testing:
 - ─ black-box testing
 - ─ white-box testing.

Summary



- # Black box testing is also known as **functional testing**.
- # Designing black box test cases:
 - ❑ requires understanding only SRS document
 - ❑ does not require any knowledge about design and code.
- # Designing white box testing requires knowledge about design and code.

Summary

⌘ We discussed black-box test case design strategies:

- ▢ equivalence partitioning
- ▢ boundary value analysis

⌘ We discussed some important issues in integration and system testing.

Cause-Effect Graphing

Dr. Durga Prasad Mohapatra
Professor
CSE Department
NIT Rourkela
durga@nitrkl.ac.in

Plan of the Talk

- ▶ Introduction to Testing
- ▶ Cause Effect Graphing
- ▶ Procedure used for the generation of tests
- ▶ Basic elements of a cause-effect graph
- ▶ Constraints amongst causes
- ▶ Constraint amongst effects
- ▶ Creating Cause-Effect Graph
- ▶ Test generation from a decision table
- ▶ Decision Table from cause-effect graph

Why Test?

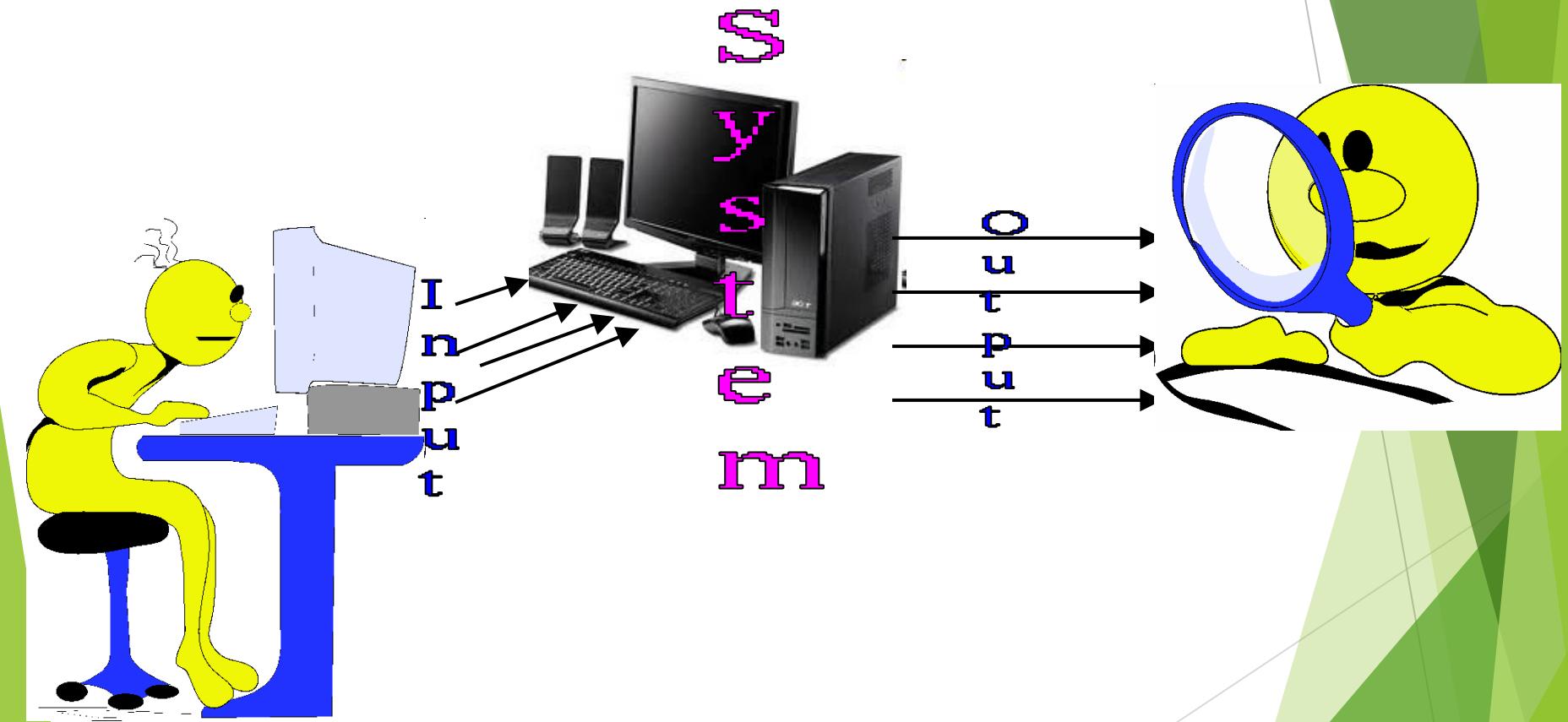


- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
 - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
 - The floating point number was larger than 32767
 - Efficiency considerations had led to the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

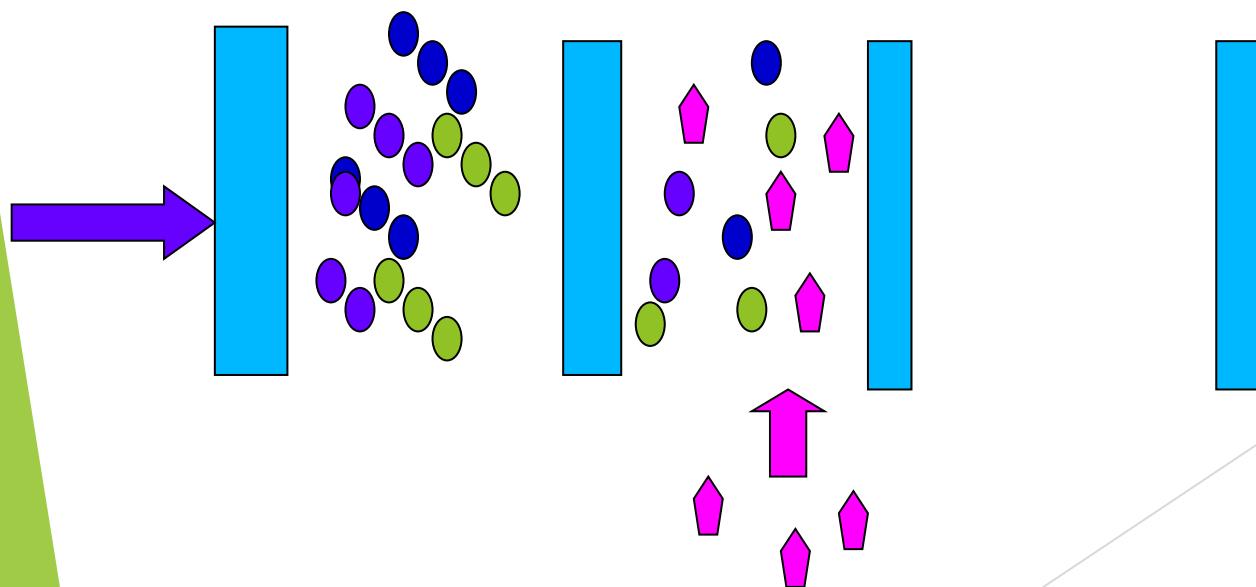
- ▶ Test Suite Design
- ▶ Run test cases and observe results to detect failures.
- ▶ Debug to locate errors
- ▶ Correct errors.

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.



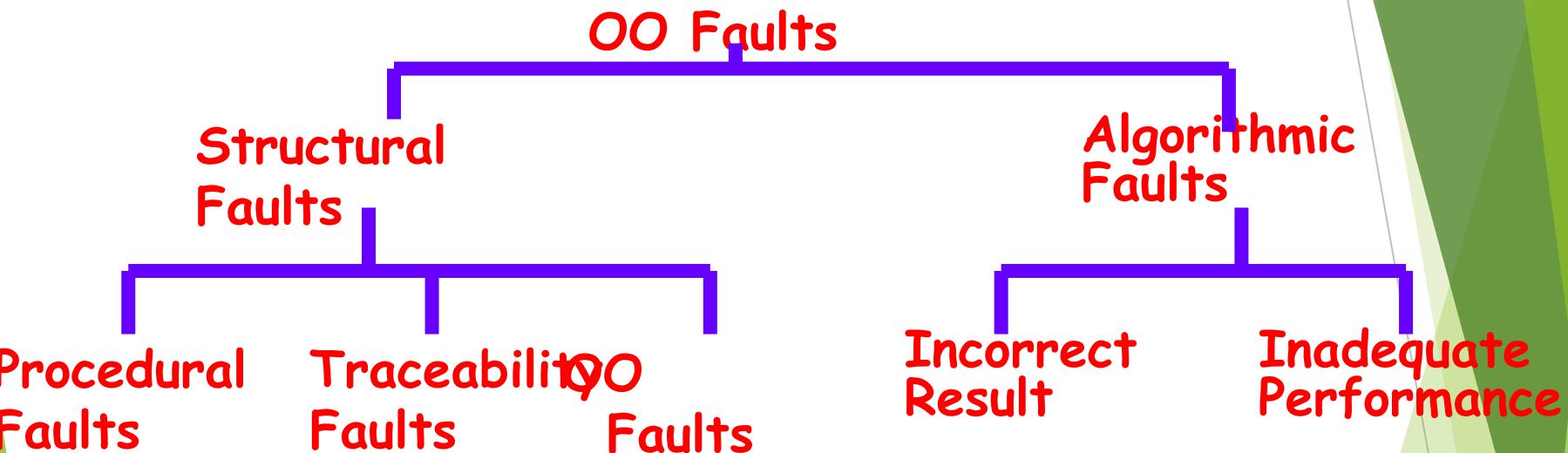
Pesticide Effect

- ▶ 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.

Black Box Testing

- ▶ Approaches to design black box test cases
 - ▶ Requirements Based Testing
 - ▶ Positive and Negative Testing
 - ▶ Boundary Value Analysis (BVA)
 - ▶ Boundary Value Checking (BVC)
 - ▶ Robustness Testing
 - ▶ Worst-Case Testing
 - ▶ Equivalence Partitioning
 - ▶ State Table Based Testing
 - ▶ Decision Table Based
 - ▶ Cause-Effect Graphing
 - ▶ Compatibility Testing
 - ▶ User Documentation Testing
 - ▶ Domain Testing

Cause-Effect Graphing

- ▶ Cause-effect graphing, also known as *dependency modeling*,
 - ▶ focuses on modelling dependency relationships amongst
 - ▶ program input conditions, known as *causes*, and
 - ▶ output conditions, known as *effects*.
- ▶ The relationship is expressed visually in terms of a cause-effect graph.
- ▶ The graph is a visual representation of a logical relationship amongst inputs and outputs that can be expressed as a Boolean expression.
- ▶ The graph allows selection of various combinations of input values as tests.
- ▶ The combinatorial explosion in the number of tests is avoided by using certain heuristics during test generation.

Cause-Effect Graphing (Contd..)

- ▶ A cause **is** any condition in the requirements that may effect the program output.
- ▶ An effect **is** the response of the program to some combination of input conditions.
 - ▶ For example, it may be
 - ▶ An error message displayed on the screen
 - ▶ A new window displayed
 - ▶ A database updated.
- ▶ An effect need not be an “output” visible to the user of the program.
- ▶ Instead, it could also be an internal *test point* in the program that can be probed during testing to check if some intermediate result is as expected.
 - ▶ For example, the intermediate test point could be ³¹ at the entrance into a method to indicate that indeed the method has been invoked.

Example

- ▶ Consider the requirement “Dispense food only when the DF switch is ON”
 - ▶ Cause is “DF switch is ON”.
 - ▶ Effect is “Dispense food”.
- ▶ This requirement implies a relationship between the “DF switch is ON” and the effect “Dispense food”.
- ▶ Other requirements might require additional causes for the occurrence of the “Dispense food” effect.

Cause and Effect Graphs

- ▶ Testing would be a lot easier:
 - ▶ if we could automatically generate test cases from requirements.
- ▶ Work done at IBM:
 - ▶ Can requirements specifications be systematically used to design functional test cases?

Cause and Effect Graphs

- ▶ Examine the requirements:
 - ▶ restate them as logical relation between inputs and outputs.
 - ▶ The result is a Boolean graph representing the relationships
 - ▶ called a **cause-effect graph**.

Cause and Effect Graphs

- ▶ Convert the graph to a decision table:
 - ▶ each column of the decision table corresponds to a test case for functional testing.

Steps to create cause-effect graph

- ▶ Study the functional requirements.
- ▶ Mark and number all causes and effects.
- ▶ Numbered causes and effects:
 - ▶ become nodes of the graph.

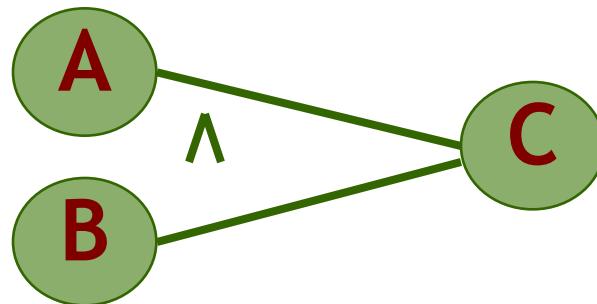
Steps to create cause-effect graph

- ▶ Draw causes on the LHS
- ▶ Draw effects on the RHS
- ▶ Draw logical relationship between causes and effects
 - ▶ as edges in the graph.
- ▶ Extra nodes can be added
 - ▶ to simplify the graph

Drawing Cause-Effect Graphs

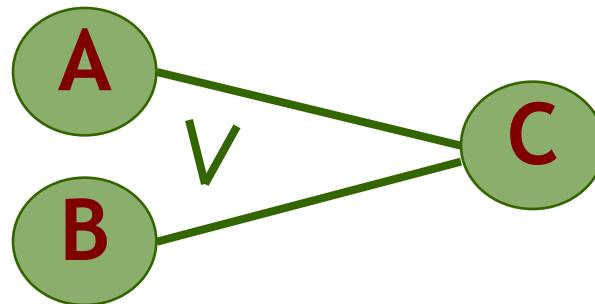


If A then B

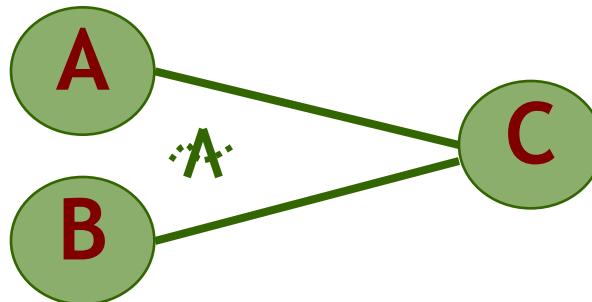


If (A and B)then C

Drawing Cause-Effect Graphs

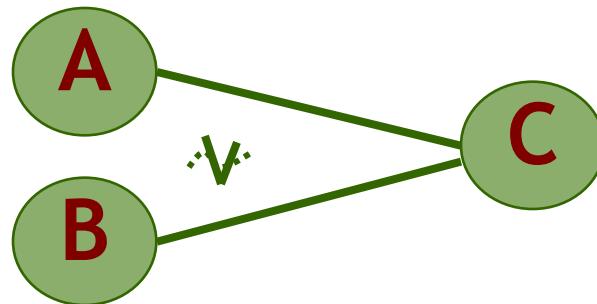


If (A or B)then C

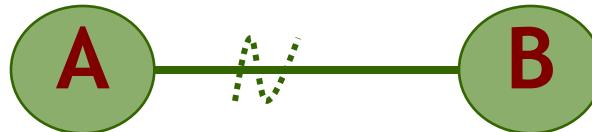


If (not(A and B))then C

Drawing Cause-Effect Graphs



If (not (A or B)) then C



If (not A) then B

Cause effect graph- Example

- ▶ A water level monitoring system
 - ▶ used by an agency involved in flood control.
 - ▶ **Input:** level(a,b)
 - ▶ a is the height of water in dam in meters
 - ▶ b is the rainfall in the last 24 hours in cms

Cause effect graph- Example

- ▶ Processing
 - ▶ The function calculates whether the level is safe, too high, or too low.
- ▶ Output
 - ▶ message on screen
 - ▶ level=safe
 - ▶ level=high
 - ▶ invalid syntax

Cause effect graph- Example

- ▶ We can separate the requirements into 5 clauses:
 - 1 ▶ first five letters of the command is “level”
 - 2 ▶ command contains exactly two parameters
 - ▶ separated by comma and enclosed in parentheses

Cause effect graph- Example

- ▶ Parameters A and B are real numbers:
 - 3 ▶ such that the water level is calculated to be low
 - 4 ▶ or safe.
- ▶ The parameters A and B are real numbers:
 - 5 ▶ such that the water level is calculated to be high.

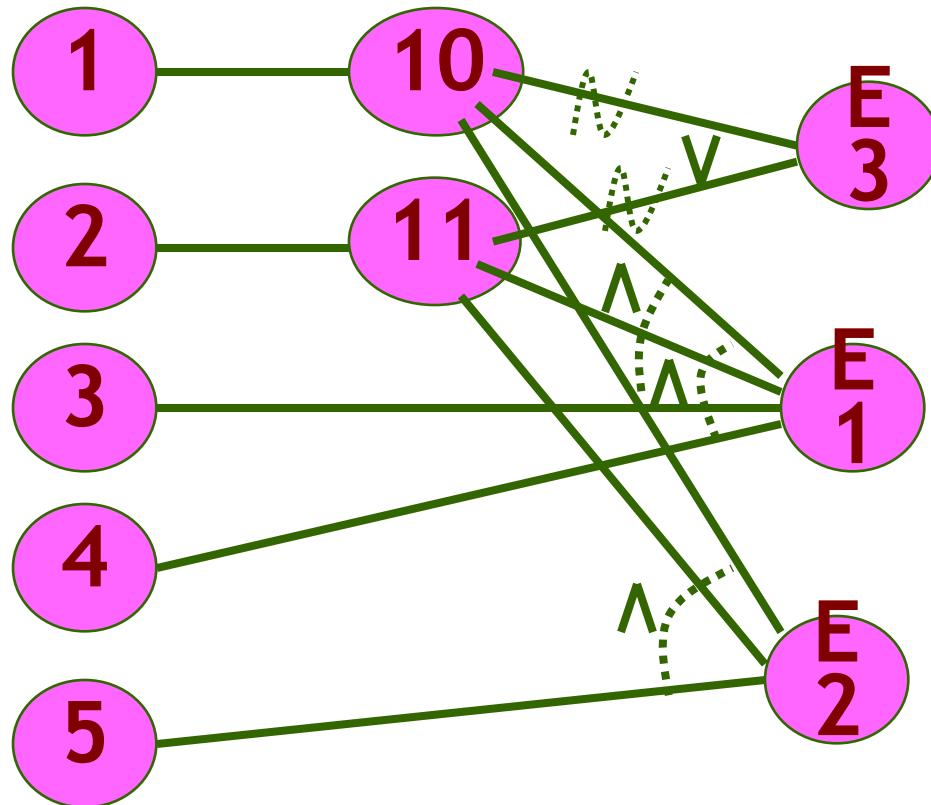
Cause effect graph- Example

- 10 ► Command is syntactically valid
- 11 ► Operands are syntactically valid.

Cause effect graph- Example

- ▶ Three effects
 - ▶ level = safe E1
 - ▶ level = high E2
 - ▶ invalid syntax E3

Cause effect graph- Example



Cause effect graph- Decision table

	Test 1	Test 2	Test 3	Test 4	Test 5	
Cause 1	I	I	I	S	I	
Cause 2	I	I	I	X	S	I = Invoked
Cause 3	I	S	S	X	X	x = don't care
Cause 4	S	I	S	X	X	s = suppressed
Cause 5	S	S	I	X	X	
Effect 1	P	P	A	A	A	
Effect 2	A	A	P	A	A	P = present
Effect 3	A	A	A	P	P	A = absent

Cause effect graph- Example

- ▶ Put a row in the decision table for each cause or effect:
 - ▶ in the example, there are five rows for causes and three for effects.

Cause effect graph- Example

- ▶ The columns of the decision table correspond to test cases.
- ▶ Define the columns by examining each effect:
 - ▶ list each combination of causes that can lead to that effect.

Cause effect graph- Example

- ▶ We can determine the number of columns of the decision table
 - ▶ by examining the lines flowing into the effect nodes of the graph.

Cause effect graph- Example

- ▶ Theoretically we could have generated $2^5=32$ test cases.
 - ▶ Using cause effect graphing technique reduces that number to 5.

Cause effect graph

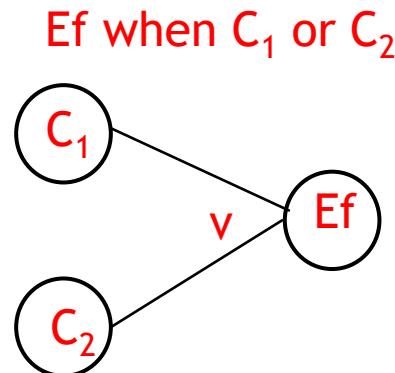
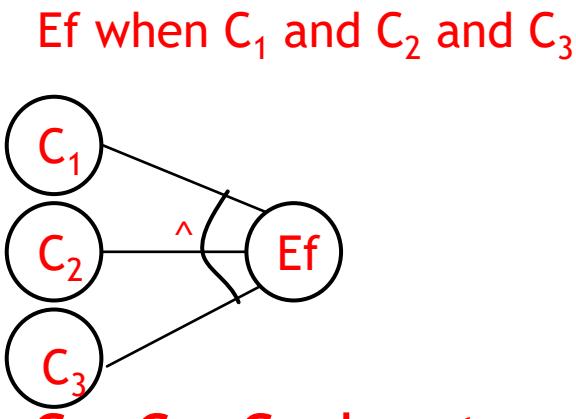
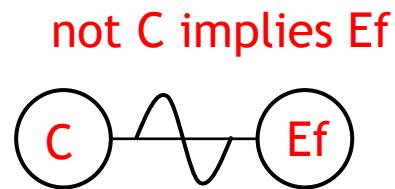
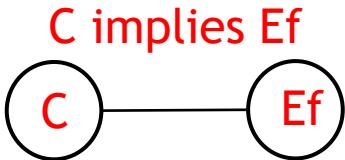
- ▶ Not practical for systems which:
 - ▶ include timing aspects
 - ▶ feedback from processes is used for some other processes.

Procedure used for the generation of tests

- ▶ Identify causes and effects by reading the requirements. Each cause and effect is assigned a unique identifier. Note that an effect can also be a cause for some other effect.
- ▶ Express the relationship between causes and effects using a cause-effect graph.
- ▶ Transform the cause-effect graph into a limited entry decision table, hereafter referred to as decision table.
- ▶ Generate tests from the decision table.

Basic elements of a cause-effect graph

- ▶ implication
- ▶ not (\sim)
- ▶ and (\wedge)
- ▶ or (\vee)



- ▶ C, C_1, C_2, C_3 denote causes.
- ▶ Ef denotes an effect.

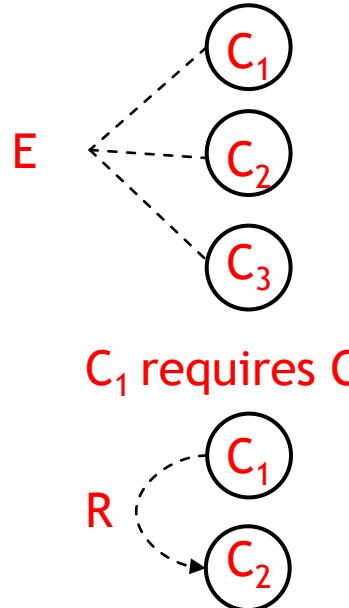
Semantics of basic elements

- ▶ C implies Ef : $\text{if}(C) \text{ then } Ef;$
- ▶ not C implies Ef : $\text{if}(\neg C) \text{ then } Ef;$
- ▶ Ef when C_1 and C_2 and C_3 : $\text{if}(C_1 \& \& C_2 \& \& C_3) \text{ then } Ef;$
- ▶ Ef when C_1 or C_2 : $\text{if}(C_1 \mid\mid C_2) \text{ then } Ef;$

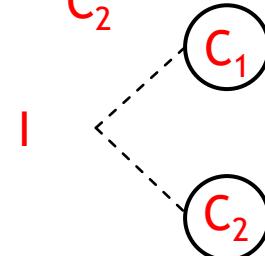
Constraints amongst causes (E,I,O,R)

- ▶ Constraints show the relationship between the causes.
- ▶ Exclusive (E)
- ▶ Inclusive (I)
- ▶ Requires (R)
- ▶ One and only one (O)

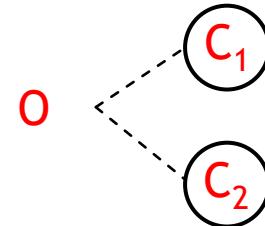
Exclusive: either C_1 or C_2 or C_3



Inclusive: at least C_1 or C_2



One and only one, of C_1 and C_2



Constraints amongst causes (E,I,O,R)

- ▶ Exclusive (E) constraint between three causes C_1 , C_2 and C_3 implies that exactly one of C_1 , C_2 , C_3 can be true.
- ▶ Inclusive (I) constraint between two causes C_1 and C_2 implies that at least one of the two must be present.
- ▶ Requires (R) constraint between C_1 and C_2 implies that C_1 requires C_2 .
- ▶ One and only one (O) constraint models the condition that one, and only one, of C_1 and C_2 must hold.

Possible values of causes constrained by E, I, R, O

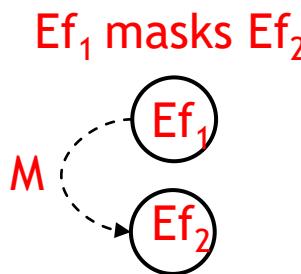
- ▶ A 0 or 1 under a cause implies that the corresponding condition is, respectively, false and true.
- ▶ The arity of all constraints, except R, is greater than 1, i.e., all except the R constraint can be applied to two or more causes; the R constraint is applied to two causes.
- ▶ A condition that is false (true) is said to be in the “0-state” (1 state).
- ▶ Similarly, an effect can be “present” (1 state) or “absent” (0 state).

Possible values of causes constrained by E, I, R, O

Constraint	Arity	Possible values		
		C1	C2	C3
$E(C_1, C_2, C_3)$	$n \geq 2$	0	0	0
		1	0	0
		0	1	0
		0	0	1
$I(C_1, C_2)$	$n \geq 2$	1	0	-
		0	1	-
		1	1	-
$R(C_1, C_2)$	$n=2$	1	1	-
		0	0	-
		0	1	-
$O(C_1, C_2, C_3)$	$n \geq 2$	1	0	0
		0	1	0
		0	0	1

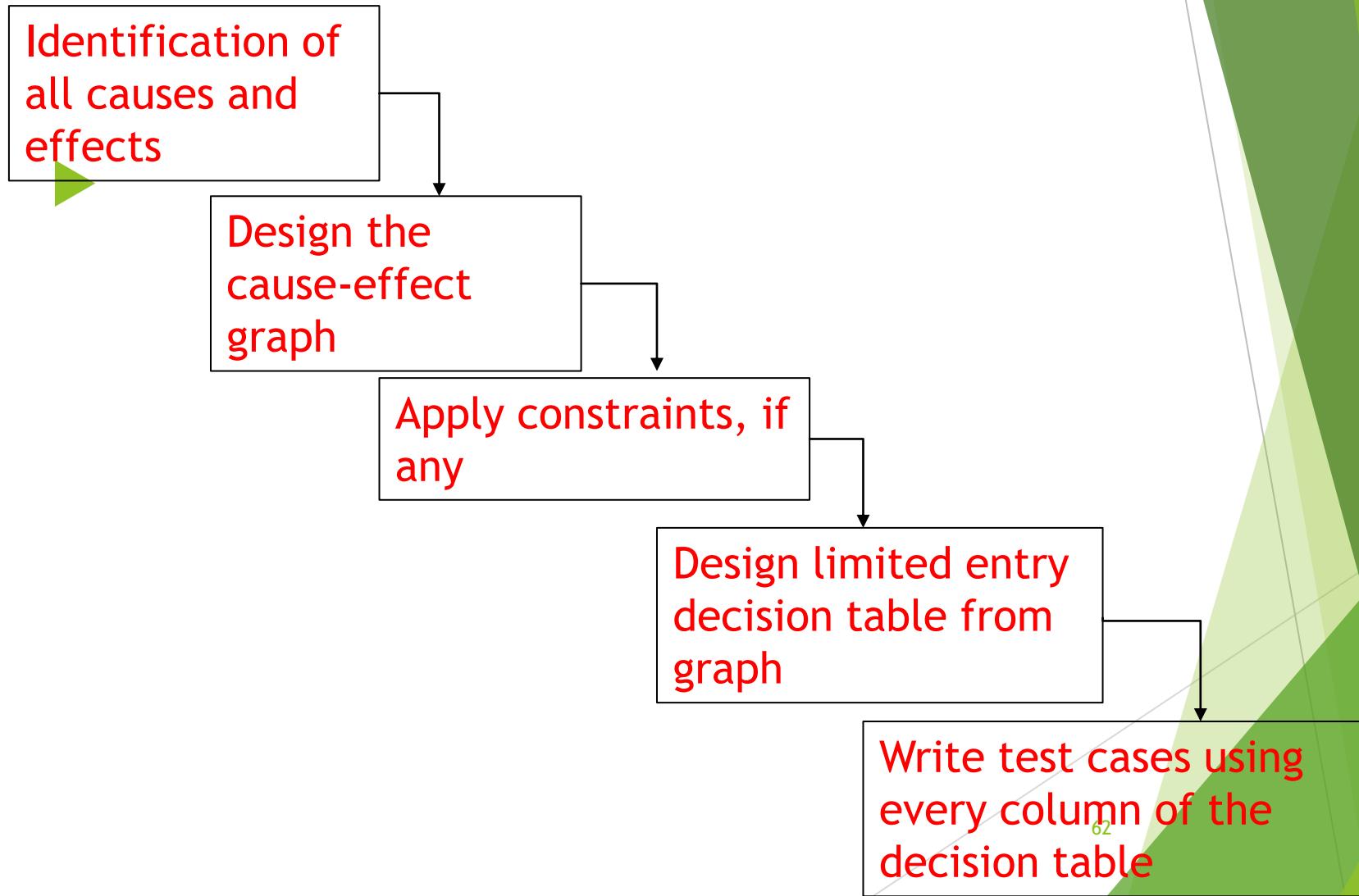
Constraint amongst effects

► Masking (M)



- Masking (M) constraint between two effects Ef_1 and Ef_2 implies that if Ef_1 is present, then Ef_2 is forced to be absent.

Steps for generating test cases using Cause-Effect Graph

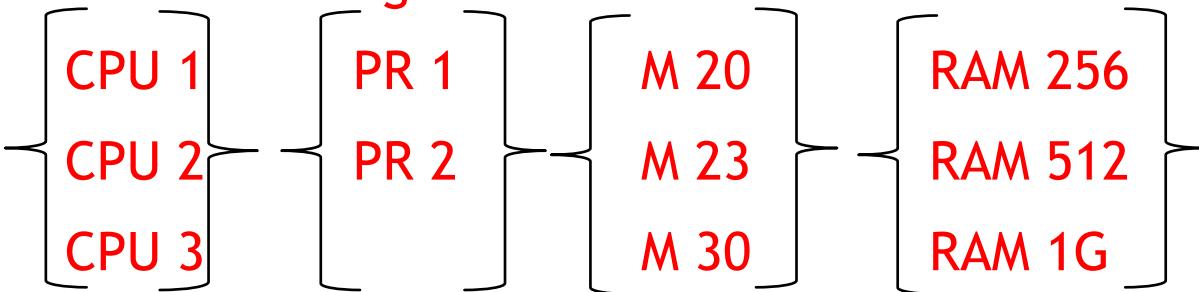


Creating Cause-Effect Graph

- ▶ The process of creating a cause-effect graph consists of two major steps.
- ▶ The causes and effects are identified by a careful examination of the requirements.
 - ▶ This process also exposes the relationships amongst various causes and effects as well as constraints amongst the causes and effects.
 - ▶ Each cause and effect is assigned a unique identifier for ease of reference in the cause-effect graph.
- ▶ The cause-effect graph is constructed to
 - ▶ express the relationships extracted from the requirements.
- ▶ When the number of causes and effects is large, say over 100 causes and 45 effects,
 - ▶ it is appropriate to use an incremental approach.

Example

- ▶ Consider the task of test generation for a GUI based computer purchase system.
- ▶ A web-based company is selling computers (CPU), printers (PR), monitors (M), and additional memory (RAM).
- ▶ An order configuration consists of one to four items as shown below.



- ▶ The GUI consists of four windows for displaying selections from CPU, Printer, Monitor and RAM and one window where any free giveaway items are displayed.
 - ▶ For each order, the buyer may select from three CPU models, two printer models, and three monitors.
 - ▶ There are separate windows one each for CPU, printer, and monitor that show the possible selections.
- For simplicity, assume that RAM is available only as an upgrade and that only one unit of each item can be purchased in one order.

Example

- ▶ Monitors M 20 and M 23 can be purchased with any CPU or as a stand-alone item.
- ▶ M 30 can only be purchased with CPU 3.
- ▶ PR 1 is available free with the purchase of CPU 2 or CPU 3.
- ▶ Monitors and printers, except for M 30, can also be purchased separately without purchasing any CPU.
- ▶ Purchase of CPU 1 gets RAM 256 upgrade.
- ▶ Purchase of CPU 2 or CPU 3 gets a RAM 512 upgrade.
- ▶ The RAM 1G upgrade and a free PR 2 is available when CPU 3 is purchased with monitor M 30.
- ▶ When a buyer selects a CPU, the contents of the printer and monitor windows are updated. Similarly, if a printer or a monitor is selected, contents of various windows are updated.

Example

- ▶ Any free printer and RAM available with the CPU selection is displayed in a different window marked “Free”.
- ▶ The total price, including taxes, for the items purchased is calculated and displayed in the “Price” window.
- ▶ Selection of a monitor could also change the items displayed in the “Free” window.
- ▶ Sample configurations and contents of the “Free” window are given below.

Items purchased	“Free” window	Price
CPU 1	RAM 256	\$499
CPU 1, PR 1	RAM 256	\$628
CPU 2, PR 2, M 23	PR 1, RAM 512	\$2257
CPU 3, M 30	PR 2, RAM 1G	\$3548

Example

- ▶ The first step in creating cause-effect graphing is to read the requirements carefully and make a list of causes and effects.
- ▶ A unique identifier, C_1 , through C_8 , has been assigned to each cause.
- ▶ Each cause listed below is a condition that can be true or false.
- ▶ C_1 : Purchase CPU 1.
- ▶ C_2 : Purchase CPU 2.
- ▶ C_3 : Purchase CPU 3.
- ▶ C_4 : Purchase PR 1.
- ▶ C_5 : Purchase PR 2.
- ▶ C_6 : Purchase M 20.
- ▶ C_7 : Purchase M 23.
- ▶ C_8 : Purchase M 30.
- ▶ For example, C_8 is true if monitor M 30 is purchased.

Example

- ▶ Note that while it is possible to order any of the items listed above, the GUI will update the selection available depending on which CPU, or any Other item, is selected.
- ▶ For example, if CPU 3 is selected for purchase then monitors M 20 and M 23 will not be available in the monitor selection window.
- ▶ Similarly, if monitor M 30 is selected for purchase, then CPU 1 and CPU 2 will not be available in the CPU window.
- ▶ Next, we identify the effects.
- ▶ In this example, the application software calculates and displays the list of items available free with the purchase and the total price.
- ▶ Hence, the effect is in terms of the contents of the “Free” and “Price” windows.

Example

- ▶ Calculation of the total purchase price depends on the items purchased and the unit price of each item.
- ▶ The unit price is obtained by the application from a price database.
- ▶ The price calculation and display is a cause that creates the effect of displaying the total price.
- ▶ For simplicity, we ignore the price related cause and effect.
- ▶ The set of effects in terms of the contents of the “Free” display window are listed below.

Ef_1 : RAM 256

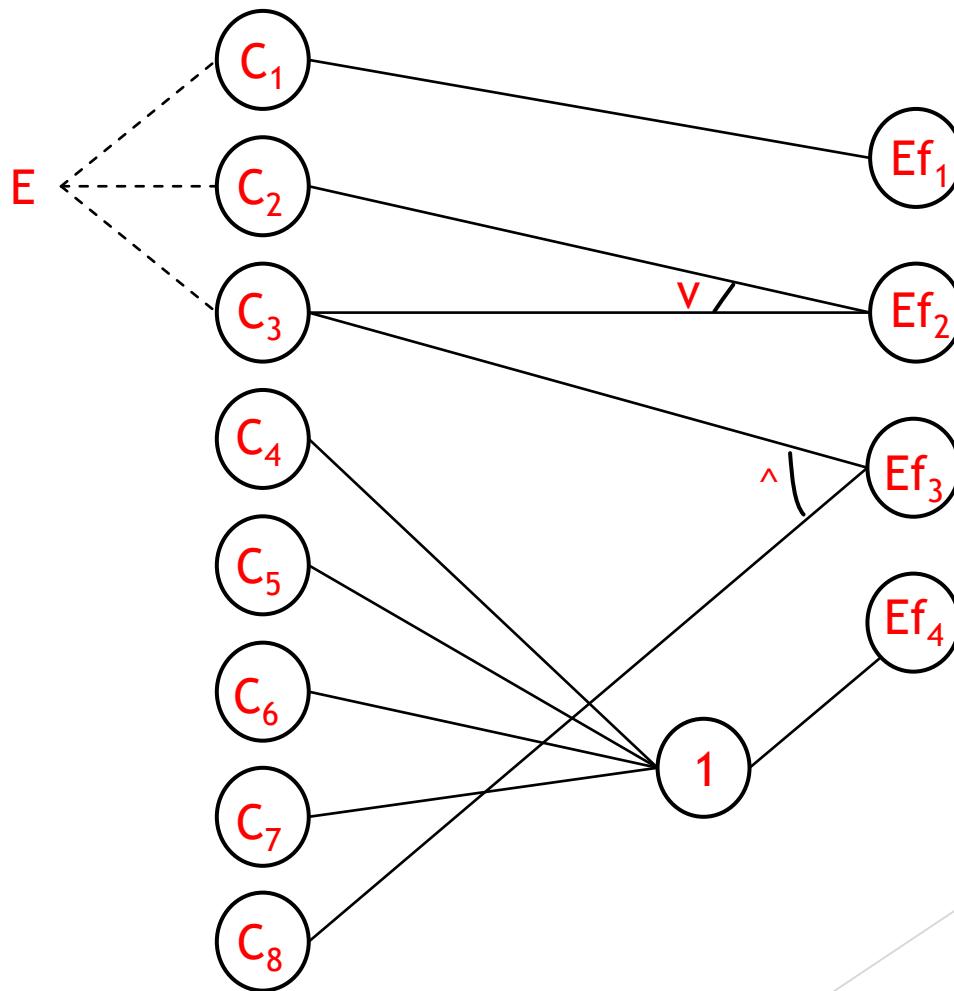
Ef_2 : RAM 512 and PR 1.

Ef_3 : RAM 1G and PR 2.

Ef_4 : No giveaway with this item.

Example

- Following shows the complete graph that expresses the relationships between C_1 through C_8 and effects Ef_1 , through Ef_4 .



Example

- ▶ From the above cause-effect graph, followings can be inferred.
 - ▶ C_1, C_2 and C_3 are constrained using the E (exclusive) relationship.
 - ▶ This expresses the requirement that only one CPU can be purchased in one order.
 - ▶ Similarly, C_3 and C_8 are related via the R (requires) constraint to express the requirement that monitor M 30 can only be purchased with CPU3.
 - ▶ Relationships amongst causes and effects are expressed using the basic elements.
- ▶ There is an intermediate node labelled 1 in the graph.
- ▶ Such intermediate nodes are often useful when an effect depends on conditions combined using more than one operator, for example, $(C_1 \wedge C_2) \vee C_3$.
- ▶ Also it can be noted that
 - ▶ Purchase of printers and monitors without any CPU leads to no free item (Ef_4).

Example

- The relationships between effects and causes shown in the graph can be expressed in terms of Boolean expressions as follows:

$$Ef_1 = C_1$$

$$Ef_2 = C_2 \vee C_3$$

$$Ef_3 = C_3 \wedge C_8$$

$$Ef_4 = C_4 \wedge C_5 \wedge C_6 \wedge C_7$$

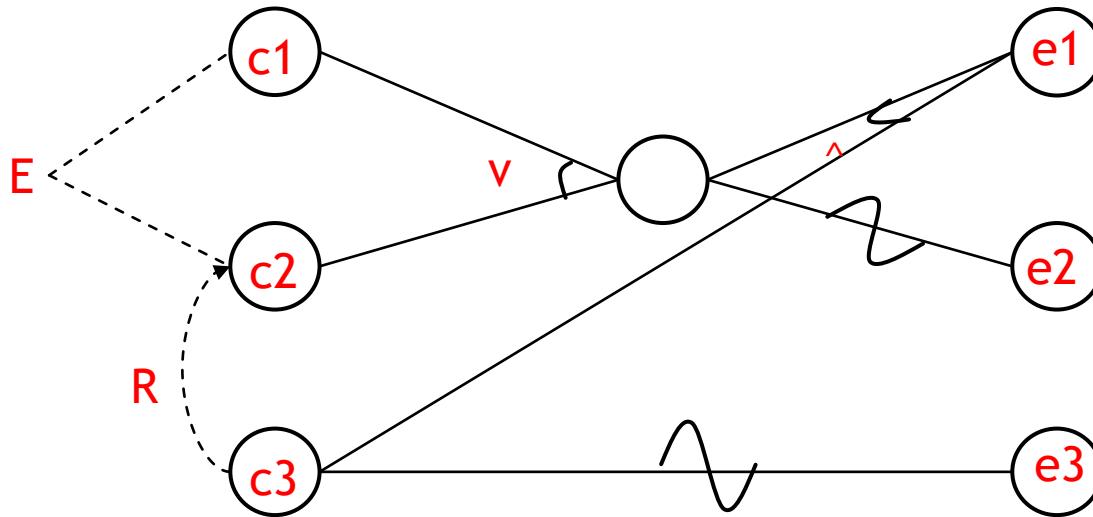
Another example

- ▶ Consider the example of keeping the record of marital status and number of children of a citizen.
- ▶ The value of marital status must be 'U' or 'M'.
- ▶ The value of the number of children must be digit or null in case a citizen is unmarried.
- ▶ If the information entered by the user is correct then an update is made.
- ▶ If the value of marital status of the citizen is incorrect, then the error message 1 is issued.
- ▶ Similarly, if the value of the number of children is incorrect, then the error message 2 is issued.

Answer

- ▶ Causes are
 - ▶ c1: marital status is U
 - ▶ c2: marital status is M
 - ▶ c3: number of children is a digit
- ▶ Effects are
 - ▶ e1: updation made
 - ▶ e2: error message 1 is issued
 - ▶ e3: error message 2 is issued

Answer



- ▶ There are two constraints
 - ▶ Exclusive (between c_1 and c_2) and
 - ▶ Requires (between c_3 and c_2)
- ▶ Causes c_1 and c_2 cannot occur simultaneously.
- ▶ For cause c_3 to be true, cause c_2 has to be true.

Heuristics to avoid combinatorial explosion

- ▶ While tracing back through a cause-effect graph,
 - ▶ we generate combinations of causes that set an intermediate node, or an effect, to a 0 or 1 state.
- ▶ Doing so in a brute force manner could lead to a large number of combinations.
- ▶ In the worst case, if n causes are related to an effect e , then the maximum number of combinations that bring e to a 1-state is 2^n .
- ▶ As tests are derived from the combinations of causes,
 - ▶ large values of n could lead to an exorbitantly large number of tests.
- ▶ We avoid such a combinatorial explosion by using simple heuristics related to the “AND” (^) and “OR” (v) nodes.
- ▶ Heuristics are based on the assumption that
 - ▶ certain types of errors are less likely to occur than others.

Heuristics to avoid combinatorial explosion

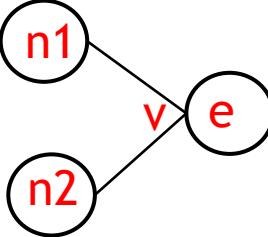
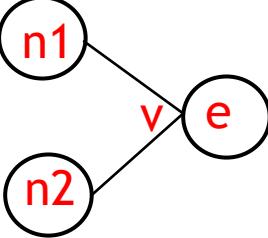
- ▶ Thus, while applying the heuristics to generate test
 - ▶ inputs will likely lead to a significant reduction in the number of tests generated,
 - ▶ it may also discard tests that would have revealed a program error.
- ▶ Hence, one must apply the heuristics with care and only when
 - ▶ the number of tests generated without their application is too large to be useful in practice.

Heuristics used during the generation of input combinations from a cause-effect graph

- ▶ The heuristics are labelled H_1 through H_4 .
- ▶ The leftmost column shows the node type in the cause-effect graph.
- ▶ The center column is the desired state of the dependent node.
- ▶ The rightmost column is the heuristics for generating combinations of inputs to the nodes that effect the dependent node e .
- ▶ For simplicity, only two nodes n_1 and n_2 and the corresponding effect e are shown.

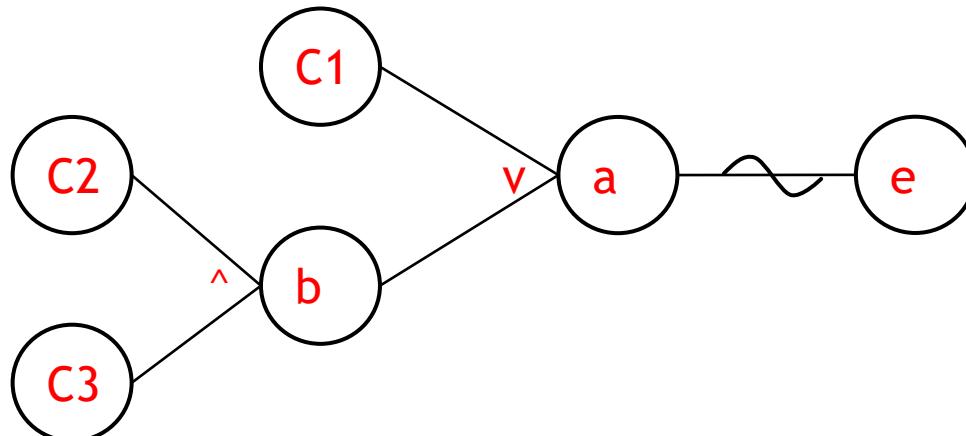
Heuristics used during the generation of input combinations from a cause-effect graph

Table 1

Node type	Desired state of e	Input combinations
	0	H1: Enumerate all combinations of inputs to n_1 and n_2 such that $n_1=n_2=0$.
	1	H2: Enumerate all combinations of inputs to n_1 and n_2 other than those for which $n_1=n_2=0$.
	0	H3: Enumerate all combinations of input to n_1 and n_2 such that each of the possible combinations of n_1 and n_2 appears exactly once and $n_1=n_2=1$ does not appear. Note that for two nodes, there are three such combinations: (0,0), (0,1) and (1,0). In general, for k nodes combined using the logical and operator to form e , there are 2^k-1 such combinations.
	1	H4: Enumerate all combinations of inputs to n_1 and n_2 such that $n_1=n_2=1$.

Example

- ▶ Consider the following cause-effect graph.
- ▶ We have at least two choices while tracing backwards to derive the necessary combinations.
- ▶ Derive all combinations first and then apply the heuristics.
- ▶ Derive combinations while applying the heuristics.



Example

- ▶ Suppose we require node e to be 1.
- ▶ Tracing backwards, this requirement implies that node a must be a 0 (zero).
- ▶ If we trace backwards further, without applying any heuristic, we obtain the following seven combinations of causes that bring e to 1 state.
- ▶ The last column lists the inputs to node a .
- ▶ The combinations that correspond to the inputs to node a listed in the rightmost column are separated by horizontal lines.

Example: Generating tests using heuristics

- ▶ First we note that node a matches the OR-node shown in the top half of Table 1.
- ▶ As we want the state of node a to be 0, heuristic H1 applies in this situation.
- ▶ H1 asks to enumerate all combinations of inputs to node a such that C1 and node b are 0.
- ▶ (0,0) is the only such combination and is listed in the last column of the following table.

	C1	C2	C3	Inputs to node a
1	0	0	0	C1=0, b=0
2	0	0	1	
3	0	1	0	
4	0	1	1	C1=0, b=1
5	1	0	0	C1=1, b=0
6	1	0	1	
7	1	1	0	

Example: Generating tests using heuristics

- ▶ Let us begin with (0,0)
- ▶ No heuristic applies to C1 as it has no preceding nodes.
- ▶ Node *b* is an AND-node as shown in the bottom half of Table.
- ▶ We want node *b* to be 0 and therefore H3 applies.
- ▶ In accordance with H3 we generate three combinations of inputs to node *b*: (0,0), (0,1) and (1,0).
- ▶ Notice that combination (1,1) is forbidden.
- ▶ Joining these combinations of C2 and C3 with C1=0,
 - ▶ we obtain the first three combinations listed in the preceding table.

Example: Generating tests using heuristics

- ▶ Though not required here, suppose that we were to consider the combination $C1=0, b=1$.
- ▶ Heuristic H4 applies in this situation.
- ▶ As both $C2$ and $C3$ are causes with no preceding nodes,
 - ▶ the only combination we obtain now is $(1,1)$.
- ▶ Combining this with $C1=0$ we obtain sequence 4 listed in the preceding table.

Example: Generating tests using heuristics

- ▶ We have completed derivation of combinations using the heuristics listed in Table 1.
- ▶ Note that the combinations listed above for $C1=1, b=0$ are not required.
- ▶ Thus, we have obtained only three combinations instead of the seven enumerated earlier.
- ▶ The reduced set of combinations is listed below.

	C1	C2	C3	Inputs to node <i>a</i>
1	0	0	0	$C1=0, b=0$
2	0	0	1	
3	0	1	0	

Example: Generating tests using heuristics

- ▶ Let us examine the rationale underlying the various heuristics for reducing the number of combinations.
- ▶ Heuristics H1 does not save us on any combinations.
- ▶ The only way an OR-node can cause its effect e to be 0 is for all its inputs to be 0.
- ▶ H1 suggests that we enumerate all such combinations.
- ▶ Heuristics H2 suggests we use all combinations that cause e to be 1 except those that cause $n_1=n_2=0$.
- ▶ To understand the rationale underlying H2 consider a program required to generate an error message when condition c_1 or c_2 is true.
- ▶ A correct implementation of this requirement is given below.

```
if(c1 v c2)printf("Error");
```

Example: Generating tests using heuristics

- ▶ Now consider the following erroneous implementation of the same requirement.

```
if(c1 v ¬c2)printf("Error");
```

- ▶ A test that sets both c1 and c2 true
 - ▶ will not be able to detect an error in the implementation above
 - ▶ if short circuit evaluation is used for Boolean expressions.
- ▶ However, a test that sets c1=0 and c2=1 will be able to detect this error.
- ▶ Hence H2 saves us from generating all input combinations that generate the pair (1,1) entering an effect in an OR-node.

Example: Generating tests using heuristics

- ▶ Heuristics H3 saves us from repeating the combinations of n_1 and n_2 .
- ▶ Once again this could save us a lot of tests.
- ▶ The assumption here is that
 - ▶ any error in the implementation of e will be detected by
 - ▶ tests that cover different combinations of n_1 and n_2 .
- ▶ Thus, there is no need to have two or more tests that
 - ▶ contain the same combination of n_1 and n_2 .

Example: Generating tests using heuristics

- ▶ Lastly, H4 for the AND-node is analogous to H1 for the OR-node.
- ▶ The only way an AND-node can cause its effect e to be 1
 - ▶ is for all its inputs to be 1.
- ▶ H4 suggests that we enumerate all such combinations.
- ▶ We stress, once again, that
 - ▶ while the heuristics will likely reduce the set of tests generated using cause-effect graphing,
 - ▶ they might also lead to useful tests being discarded.
- ▶ Of course, in general and prior to the start of testing,
 - ▶ it is almost impossible to know which of the test cases discarded will be useless and which ones useful.

Decision Table from cause-effect graph

- ▶ Each column of the decision table represents a combination of input values, and hence a test.
- ▶ There is one row for each condition and effect.
- ▶ Thus the table decision table can be viewed as an $N \times M$ matrix with
 - ▶ N being the sum of the number of conditions and effects and
 - ▶ M the number of tests.
- ▶ Each entry in the decision table is a 0 or 1
 - ▶ depending on whether or not the corresponding condition is false or true, respectively.
- ▶ For a row corresponding to an effect, an entry is 0 or 1
 - ▶ if the effect is not present or present, respectively.

Procedure for generating a decision table from a cause-effect graph

- ▶ **Input:** A cause-effect graph containing causes C_1, C_2, \dots, C_p and effects Ef_1, Ef_2, \dots, Ef_q .
- ▶ **Output:** A decision table DT containing $N=p+q$ rows and M columns, where M depends on the relationship between the causes and effects as captured in the cause-effect graph.
- ▶ **Procedure:** CEGDT
 - ▶ `/*`
 - ▶ i is the index of the next effect to be considered.
 - ▶ $next_dt_col$ is the next empty column in the decision table.
 - ▶ V_k : a vector of size $p+q$ containing 1's and 0's. V_j , $1 \leq j \leq p$, indicates the state of condition C_j and V_l , $p < l \leq p+q$, indicates the presence or absence of effect Ef_{l-p} .
 - ▶ `*/`

Procedure for generating a decision table from a cause-effect graph

- ▶ **Step 1** Initialize DT to an empty decision table.

next_dt_col=1.

- ▶ **Step 2** Execute the following steps for $i=1$ to q .

- 2.1 Select the next effect to be processed.

Let $e=Ef_i$.

- 2.2 Find combinations of conditions that cause e to be present.

Assume that e is present. Starting at e , trace the cause-effect graph backwards and determine the combinations of conditions C_1, C_2, \dots, C_p that lead to e being present.

Let V_1, V_2, \dots, V_{mi} be the combinations of causes that lead to e being present, i.e. in 1 state, and hence $m_i \geq 1$. Set $V_k(l), p < l \leq p+q$ to 0 or 1 depending on whether effect Ef_{l-p} is present or not for the combination of all conditions in V_k .

Procedure for generating a decision table from a cause-effect graph

2.3 *Update the decision table.*

Add V_1, V_2, \dots, V_{mi} to the decision table as successive columns starting at *next_dt_col*.

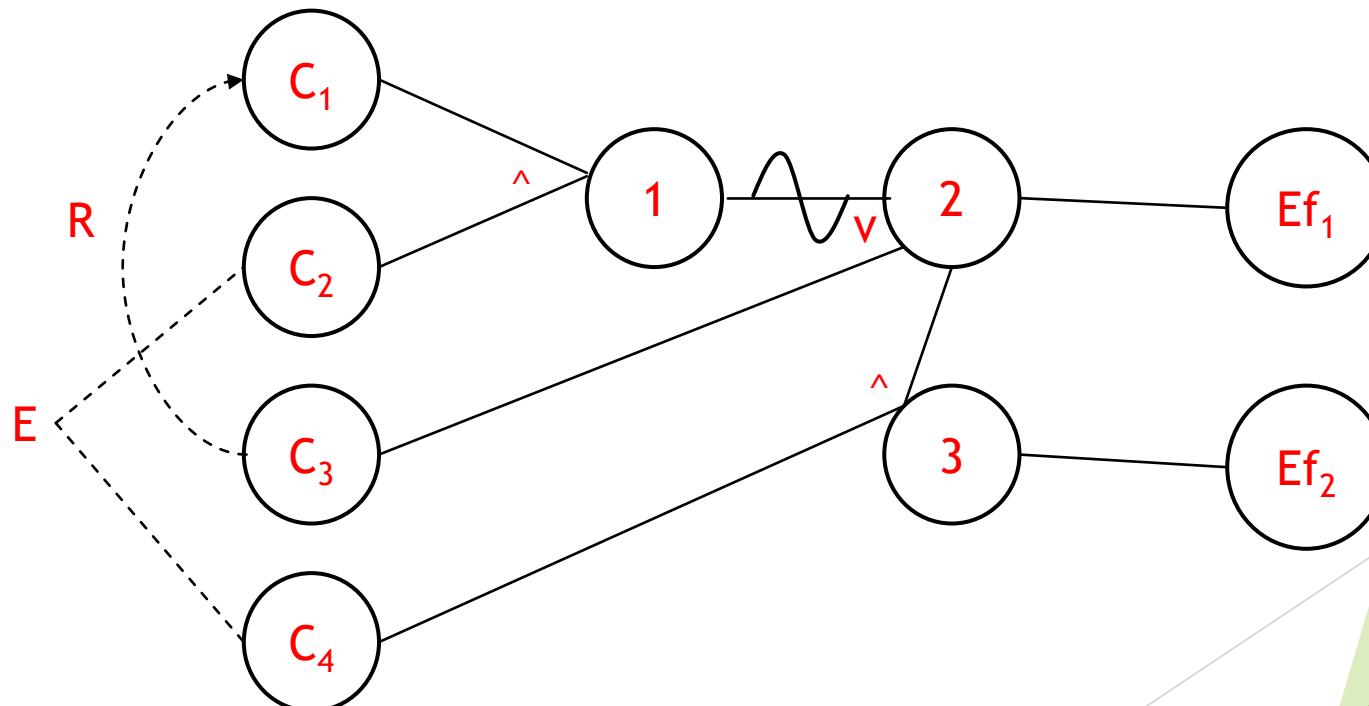
2.4 *Update the next available column in the decision table.*

next_dt_col=next_dt_col+m_i. At the end of this procedure, *next_dt_col-1* is the number of tests generated.

► End of Procedure CEGDT

Example

- ▶ Consider the cause effect graph shown below.
- ▶ It shows four causes labelled C_1, C_2, C_3 and C_4 and two effects labeled Ef_1 and Ef_2 .
- ▶ There are three intermediate nodes labeled 1, 2 and 3.



Example

- ▶ Step 1: Set $next_dt_col=1$ to initialize the decision table to empty.
- ▶ Next, $i=1$ and in accordance with Step 2.2, $e=Ef_1$.
- ▶ In accordance with Step 2.2, trace backwards from e to determine combinations that will cause e to be present.
- ▶ e must be present when node 2 is in 1-state.
- ▶ Moving backwards from node 2 in the cause-effect graph, any of the following three combinations of states of nodes 1 and C_3 will lead to e being present: $(0,1)$, $(1,1)$ and $(0,0)$.
- ▶ Node 1 is also an internal node and hence move further back to obtain the values of C_1 and C_2 that effect node 1.
- ▶ Combinations of C_1 and C_2 that brings node 1 to the 1-state is $(1,1)$ and combinations that bring it to 0-state are $(1,0)$, $(0,1)$ and $(0,0)$.

Example

- ▶ Combining this information with that derived earlier for nodes 1 and C_3 , we obtain the following seven combinations of C_1 , C_2 and C_3 that cause e to be present.

1 0 1

0 1 1

0 0 1

1 1 1

1 0 0

0 1 0

0 0 0

▶ Next, C_3 requires C_1 , which implies that C_1 must be in 1-state for C_3 to be in 1-state.

▶ This constraint makes infeasible the second and third combinations above.

Example

- In the end, we obtain the following five combinations of the four causes that lead to e being present.

1 0 1

1 1 1

1 0 0

0 1 0

0 0 0

- Setting C_4 to 0 and appending the values of Ef_1 and Ef_2 , we obtain the following five vectors.

V_1 1 0 1 0 1 0

V_2 1 1 1 0 1 0

V_3 1 0 0 0 1 0

V_4 0 1 0 0 1 0

V_5 0 0 0 0 1 0

Example

- ▶ Note that $m_1=5$ in Step 2.
- ▶ This completes the application of Step 2.2.
- ▶ The five vectors are transposed and added to the decision table starting at column *next_dt_col* which is 1.
- ▶ The decision table at Step 2.3 follows

	1	2	3	4	5
C_1	1	1	1	0	0
C_2	0	1	0	1	0
C_3	1	1	0	0	0
C_4	0	0	0	0	0
Ef_1	1	1	1	1	1
Ef_2	0	0	0	0	0

- ▶ We update *next_dt_col* to 6, increment *i* to 2 and get back to Step 2.1.

Example

- ▶ We now have $e=Ef_2$.
- ▶ Tracing backwards, we find that for e to be present, node 3 must be in the 1-state.
- ▶ This is possible with only one combination of node 2 and C_4 , which is (1,1).
- ▶ Earlier we derived the combinations of C_1 , C_2 and C_3 that lead node 2 into the 1-state.
- ▶ Combining these with the value of C_4 , we arrive at the following combination of causes that lead to the presence of Ef_2 .

1	0	1	1
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1

Example

- ▶ From the cause-effect graph, note that C_2 and C_4 cannot be present simultaneously.
- ▶ Hence, we discard the second and fourth combinations from the list above and obtain the following three feasible combinations.

1 0 1 1

1 0 0 1

0 0 0 1

- ▶ Appending the corresponding values of Ef_1 and Ef_2 to each of the above combinations, we obtain the following three vectors.

V_1 1 0 1 1 1 1

V_2 1 0 0 1 1 1

V_3 0 0 0 1 1 1

Example

- ▶ Transposing the vectors listed above and appending them as three columns to the existing decision table, we obtain the following

	1	2	3	4	5	6	7	8
C_1	1	1	1	0	0	1	1	0
C_2	0	1	0	1	0	0	0	0
C_3	1	1	0	0	0	1	0	0
C_4	0	0	0	0	0	1	1	1
Ef_1	1	1	1	1	1	1	1	1
Ef_2	0	0	0	0	0	1	1	1

- ▶ Next, we update *next_dt_col* to 9.
- ▶ Of course, doing so is useless as the loop set up in Step 2 is now terminated.
- ▶ The decision table listed above is the output.¹⁰¹

Test generation from a decision table

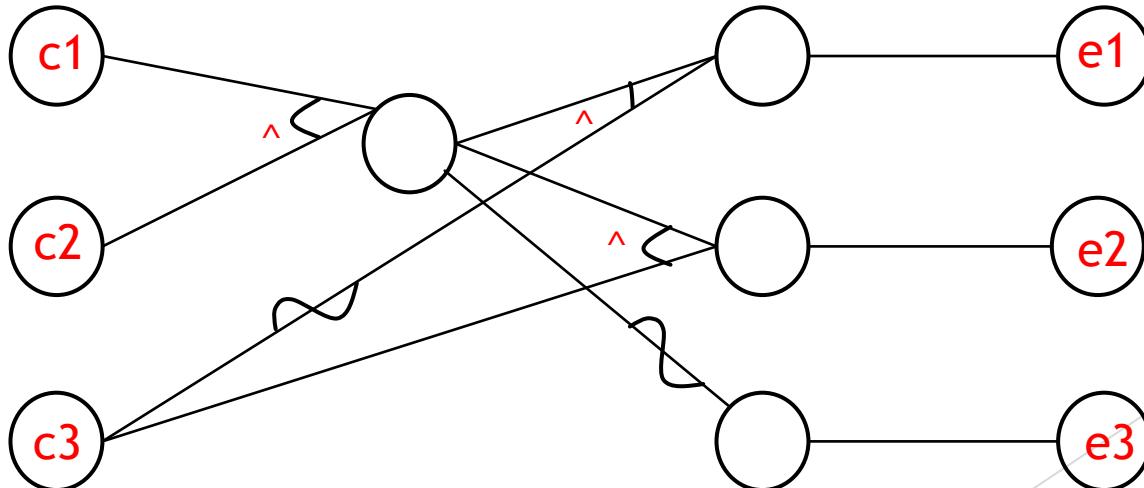
- ▶ Test generation from a decision table is relatively forward.
- ▶ Each column in the decision table generates at least one test input.
- ▶ Note that each combination might be able to generate more than one test when a condition in the cause-effect graph can be satisfied in more than one way.
- ▶ For example, consider the following cause:
- ▶ $C: x < 99$
- ▶ The condition above can be satisfied by many values such as $x=1$ and $x=49$.
- ▶ Also, C can be made false by many values of x such as $x=100$ and $x=999$.
- ▶ Thus, one might have a choice of values of input variables while generating tests using columns from a decision table

Example

- ▶ A tourist of age greater than 21 years and having a clean driving record is supplied a rental car.
- ▶ A premium amount is also charged if the tourist is on business,
- ▶ Otherwise, it is not charged.
- ▶ If the tourist is less than 21 year old, or does not have a clean driving record,
 - ▶ The system will display the following message: “Car cannot be supplied”.

Answer

- ▶ Causes are
 - ▶ c1: Age is over 21
 - ▶ c2: Driving record is clean
 - ▶ c3: Tourist is on business
- ▶ Effects are
 - ▶ e1: Supply a rental car without premium charge
 - ▶ e2: Supply a rental car with premium charge
 - ▶ e3: Car cannot be supplied



Decision Table and Test Cases

	1	2	3	4
c1: Over 21?	F	T	T	T
c2: Driving record clean?	-	F	T	T
c3: On business?	-	-	F	T
e1: Supply a rental car without premium charge			X	
e2: Supply a rental car with premium charge				X
e3: Car cannot be supplied	X	X		

Test Case	Age	Driving_record_clean	On_business	Expected Output
1	20	Yes	Yes	Car cannot be supplied
2	26	No	Yes	Car cannot be supplied
3	62	Yes	No	Supply a rental car without premium charge
4	62	Yes	Yes	Supply a rental car with ₁₀₅ premium charge

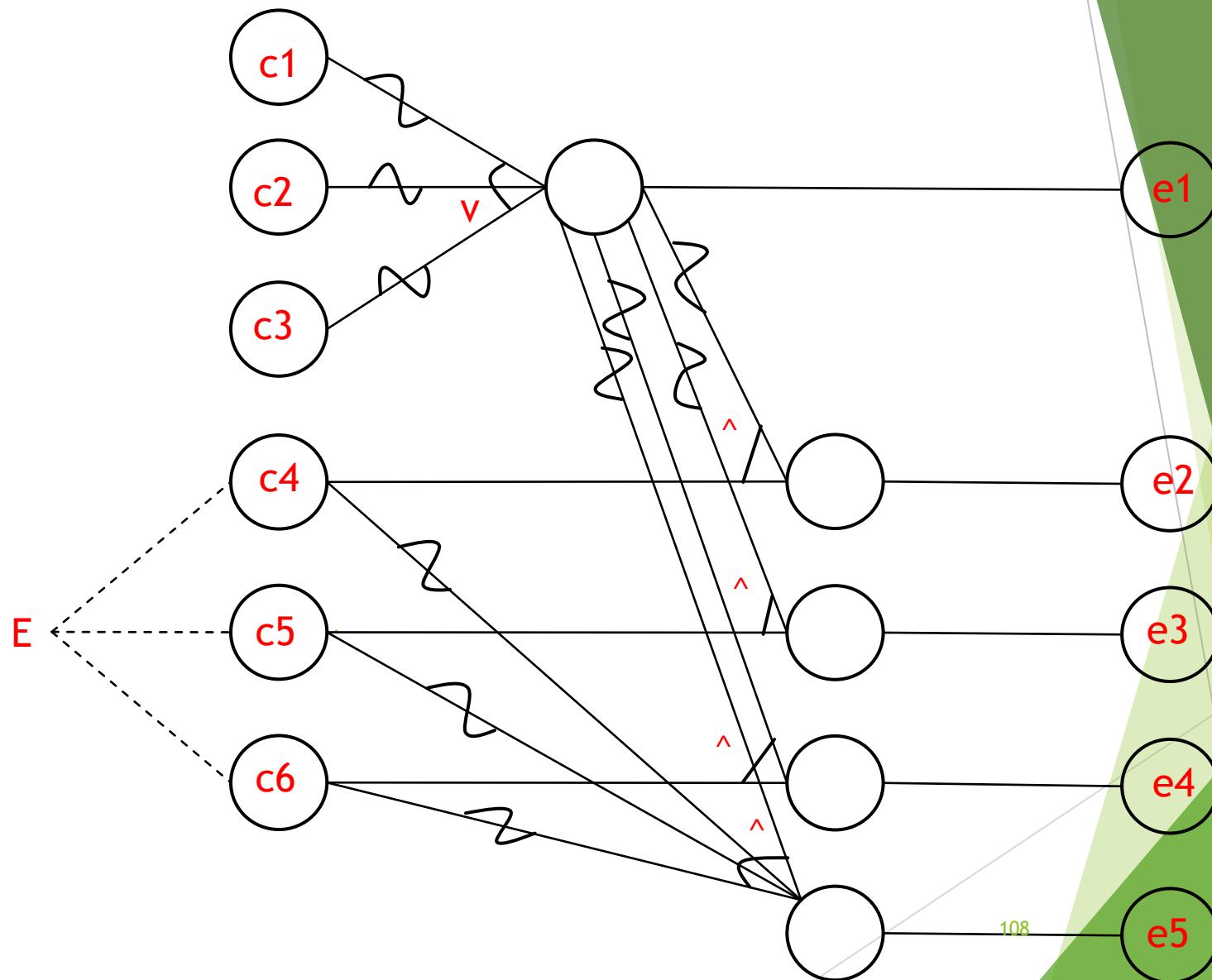
Example 2: Triangle Classification Problem

- ▶ Consider a program for classification of a triangle.
- ▶ Its input is a triple of positive integers (say a, b and c) and the input values are greater than zero and less than or equal to 100.
- ▶ The triangle is classified according to the following rules:
 - ▶ Right angled triangle: $c^2=a^2+b^2$ or $a^2=b^2+c^2$ or $b^2=c^2+a^2$
 - ▶ Obtuse angled triangle: $c^2>a^2+b^2$ or $a^2>b^2+c^2$ or $b^2>c^2+a^2$
 - ▶ Acute angled triangle: $c^2< a^2+b^2$ or $a^2< b^2+c^2$ or $b^2< c^2+a^2$
 - ▶ The program output may have one of the following words: [Acute angled triangle, Obtuse angled triangle, Right angled triangle, Invalid triangle, Input values are out of range]

Answer

- ▶ Causes are:
 - ▶ c1: side “a” is less than the sum of sides “b” and “c”.
 - ▶ c2: side “b” is less than the sum of sides “a” and “c”.
 - ▶ c3: side “c” is less than the sum of sides “a” and “b”.
 - ▶ c4: square of side “a” is equal to the sum of squares of sides “b” and “c”.
 - ▶ c5: square of side “a” is greater than the sum of squares of sides “b” and “c”.
 - ▶ c6: square of side “a” is less than the the sum of squares of sides “b” and “c”.
- ▶ Effects are:
 - ▶ e1: Invalid triangle
 - ▶ e2: Right angle triangle
 - ▶ e3: Obtuse angled triangle
 - ▶ e4: Acute angled triangle
 - ▶ e5: Impossible stage

Cause-Effect Graph



Decision Table

	1	2	3	4	5	6	7	8	9	10	11
c1: $a < b+c$	F	T	T	T	T	T	T	T	T	T	T
c2: $b < a+c$	-	F	T	T	T	T	T	T	T	T	T
c3: $c < a+b$	-	-	F	T	T	T	T	T	T	T	T
c4: $a^2 = b^2 + c^2$	-	-	-	T	T	T	T	F	F	F	F
c5: $a^2 > b^2 + c^2$	-	-	-	T	T	F	F	T	T	F	F
c6: $a^2 < b^2 + c^2$	-	-	-	T	F	T	F	T	F	T	F
e1: Invalid triangle	X	X	X								
e2: Right angle triangle							X				
e3: Obtuse angled triangle									X		
e4: Acute angled triangle										X	
e5: Impossible					X	X	X	X			X

Thank You

Cause-Effect Graphing

Dr. Durga Prasad Mohapatra
Professor
CSE Department
NIT Rourkela
durga@nitrkl.ac.in

Plan of the Talk

- ▶ Introduction to Testing
- ▶ Cause Effect Graphing
- ▶ Procedure used for the generation of tests
- ▶ Basic elements of a cause-effect graph
- ▶ Constraints amongst causes
- ▶ Constraint amongst effects
- ▶ Creating Cause-Effect Graph
- ▶ Test generation from a decision table
- ▶ Decision Table from cause-effect graph

Why Test?

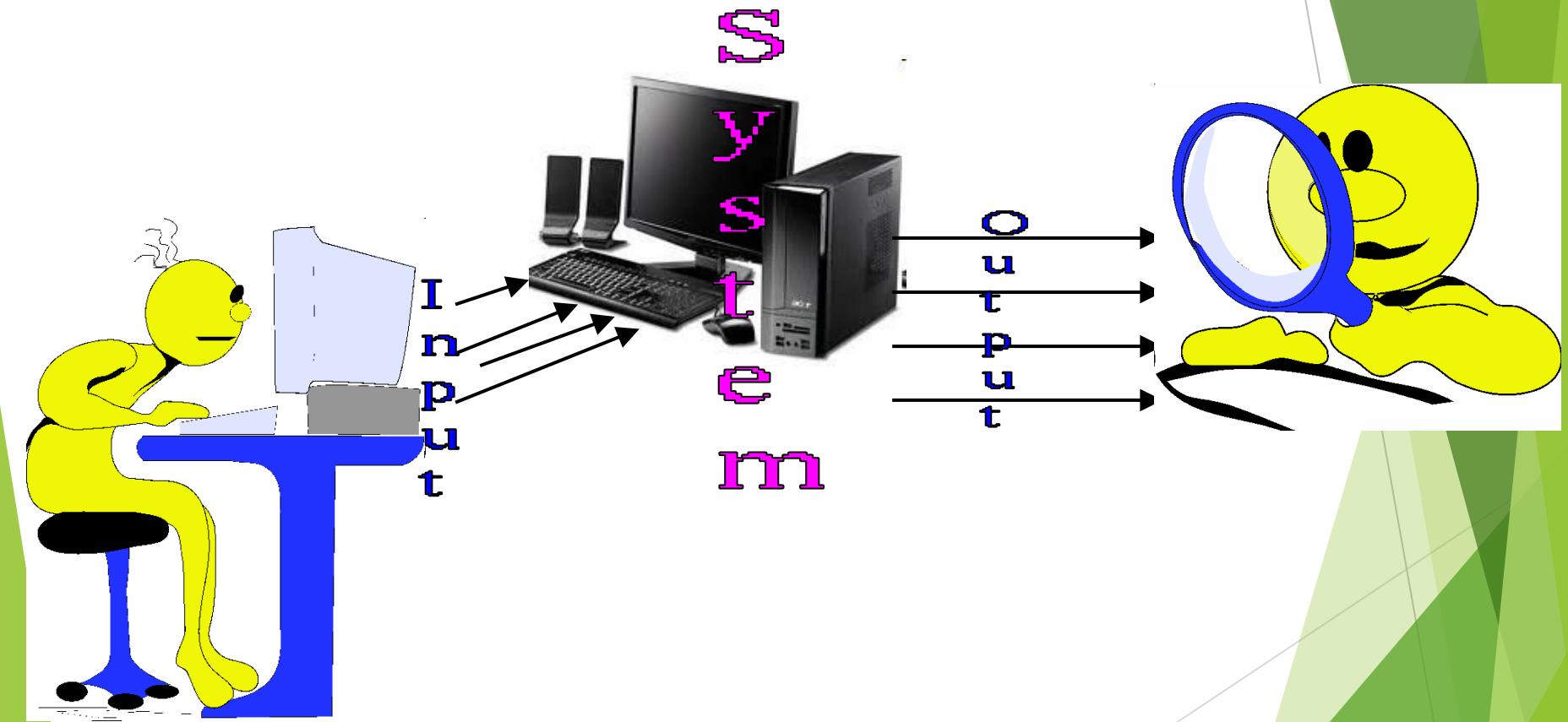


- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
 - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
 - The floating point number was larger than 32767
 - Efficiency considerations had led to the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

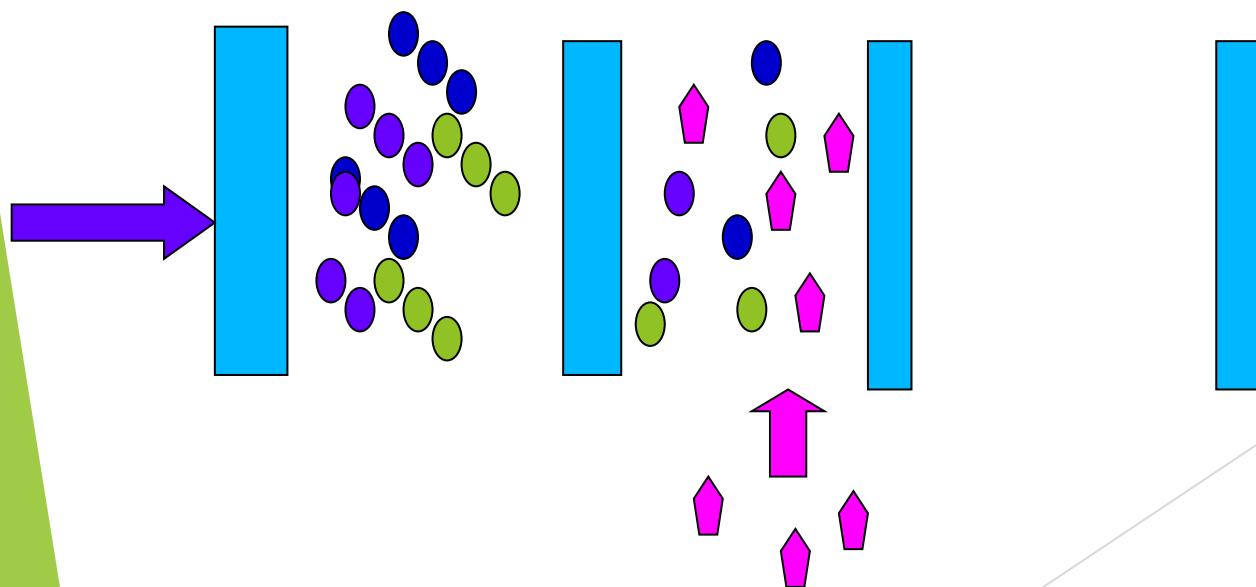
- ▶ Test Suite Design
- ▶ Run test cases and observe results to detect failures.
- ▶ Debug to locate errors
- ▶ Correct errors.

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.



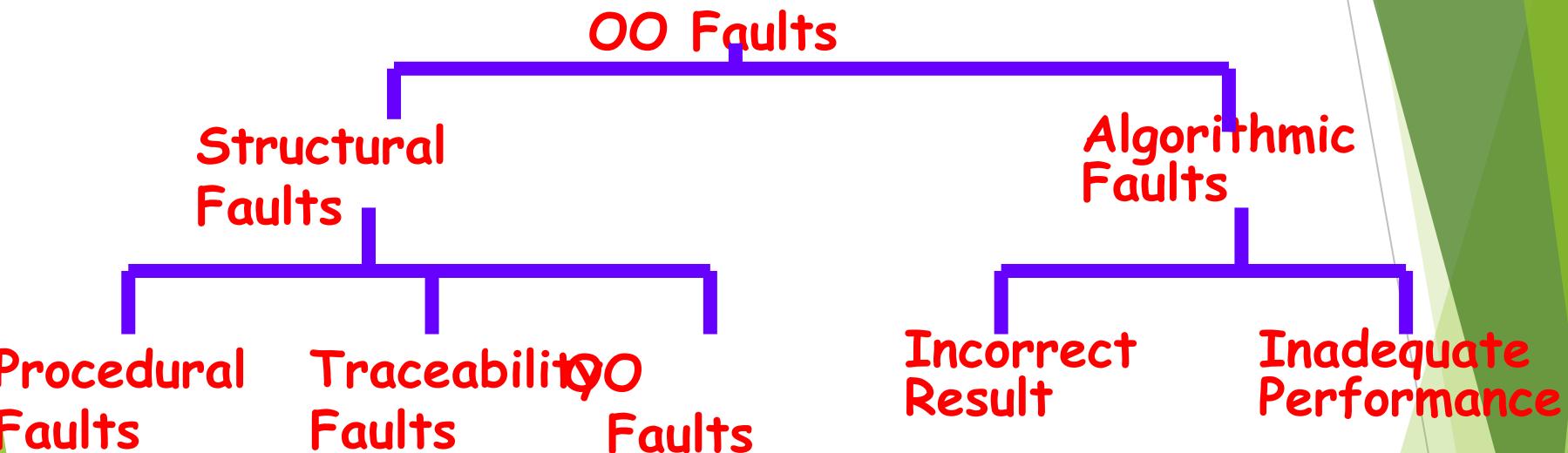
Pesticide Effect

- ▶ 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.

Black Box Testing

- ▶ Approaches to design black box test cases
 - ▶ Requirements Based Testing
 - ▶ Positive and Negative Testing
 - ▶ Boundary Value Analysis (BVA)
 - ▶ Boundary Value Checking (BVC)
 - ▶ Robustness Testing
 - ▶ Worst-Case Testing
 - ▶ Equivalence Partitioning
 - ▶ State Table Based Testing
 - ▶ Decision Table Based
 - ▶ Cause-Effect Graphing
 - ▶ Compatibility Testing
 - ▶ User Documentation Testing
 - ▶ Domain Testing

Cause-Effect Graphing

- ▶ Cause-effect graphing, also known as *dependency modeling*,
 - ▶ focuses on modelling dependency relationships amongst
 - ▶ program input conditions, known as *causes*, and
 - ▶ output conditions, known as *effects*.
- ▶ The relationship is expressed visually in terms of a cause-effect graph.
- ▶ The graph is a visual representation of a logical relationship amongst inputs and outputs that can be expressed as a Boolean expression.
- ▶ The graph allows selection of various combinations of input values as tests.
- ▶ The combinatorial explosion in the number of tests is avoided by using certain heuristics during test generation.

Cause-Effect Graphing (Contd..)

- ▶ A cause **is** any condition in the requirements that may effect the program output.
- ▶ An effect **is** the response of the program to some combination of input conditions.
 - ▶ For example, it may be
 - ▶ An error message displayed on the screen
 - ▶ A new window displayed
 - ▶ A database updated.
- ▶ An effect need not be an “output” visible to the user of the program.
- ▶ Instead, it could also be an internal *test point* in the program that can be probed during testing to check if some intermediate result is as expected.
 - ▶ For example, the intermediate test point could be ³¹ at the entrance into a method to indicate that indeed the method has been invoked.

Example

- ▶ Consider the requirement “Dispense food only when the DF switch is ON”
 - ▶ Cause is “DF switch is ON”.
 - ▶ Effect is “Dispense food”.
- ▶ This requirement implies a relationship between the “DF switch is ON” and the effect “Dispense food”.
- ▶ Other requirements might require additional causes for the occurrence of the “Dispense food” effect.

Cause and Effect Graphs

- ▶ Testing would be a lot easier:
 - ▶ if we could automatically generate test cases from requirements.
- ▶ Work done at IBM:
 - ▶ Can requirements specifications be systematically used to design functional test cases?

Cause and Effect Graphs

- ▶ Examine the requirements:
 - ▶ restate them as logical relation between inputs and outputs.
 - ▶ The result is a Boolean graph representing the relationships
 - ▶ called a **cause-effect graph**.

Cause and Effect Graphs

- ▶ Convert the graph to a decision table:
 - ▶ each column of the decision table corresponds to a test case for functional testing.

Steps to create cause-effect graph

- ▶ Study the functional requirements.
- ▶ Mark and number all causes and effects.
- ▶ Numbered causes and effects:
 - ▶ become nodes of the graph.

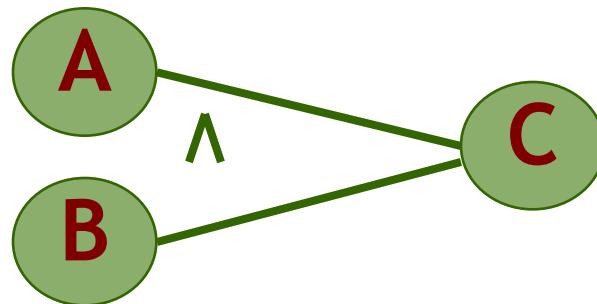
Steps to create cause-effect graph

- ▶ Draw causes on the LHS
- ▶ Draw effects on the RHS
- ▶ Draw logical relationship between causes and effects
 - ▶ as edges in the graph.
- ▶ Extra nodes can be added
 - ▶ to simplify the graph

Drawing Cause-Effect Graphs

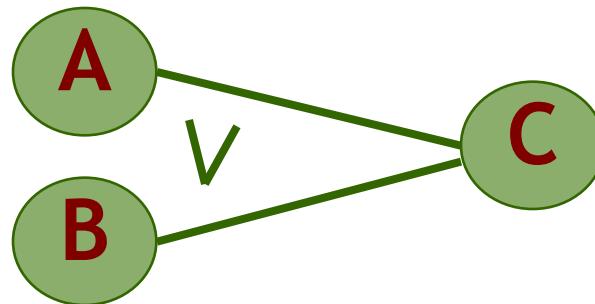


If A then B

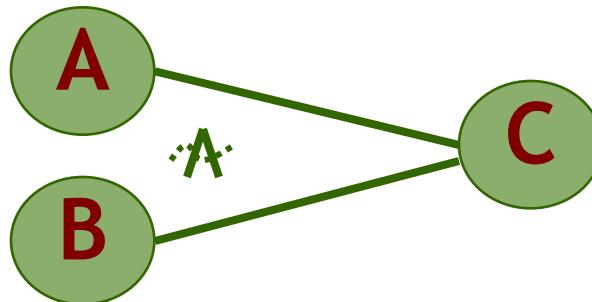


If (A and B)then C

Drawing Cause-Effect Graphs

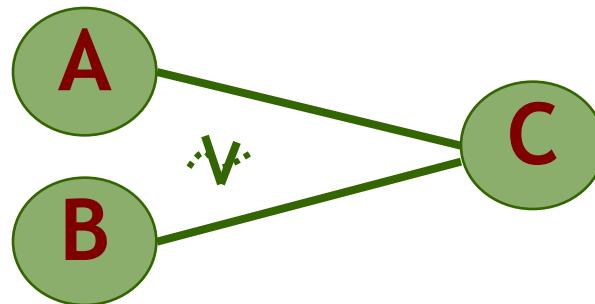


If (A or B)then C

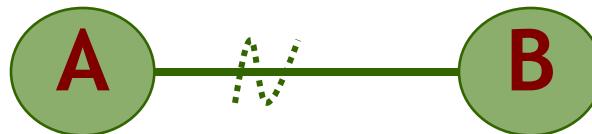


If (not(A and B))then C

Drawing Cause-Effect Graphs



If (not (A or B)) then C



If (not A) then B

Cause effect graph- Example

- ▶ A water level monitoring system
 - ▶ used by an agency involved in flood control.
 - ▶ **Input:** level(a,b)
 - ▶ a is the height of water in dam in meters
 - ▶ b is the rainfall in the last 24 hours in cms

Cause effect graph- Example

- ▶ Processing
 - ▶ The function calculates whether the level is safe, too high, or too low.
- ▶ Output
 - ▶ message on screen
 - ▶ level=safe
 - ▶ level=high
 - ▶ invalid syntax

Cause effect graph- Example

- ▶ We can separate the requirements into 5 clauses:
 - 1 ▶ first five letters of the command is “level”
 - 2 ▶ command contains exactly two parameters
 - ▶ separated by comma and enclosed in parentheses

Cause effect graph- Example

- ▶ Parameters A and B are real numbers:
 - 3 ▶ such that the water level is calculated to be low
 - 4 ▶ or safe.
- ▶ The parameters A and B are real numbers:
 - 5 ▶ such that the water level is calculated to be high.

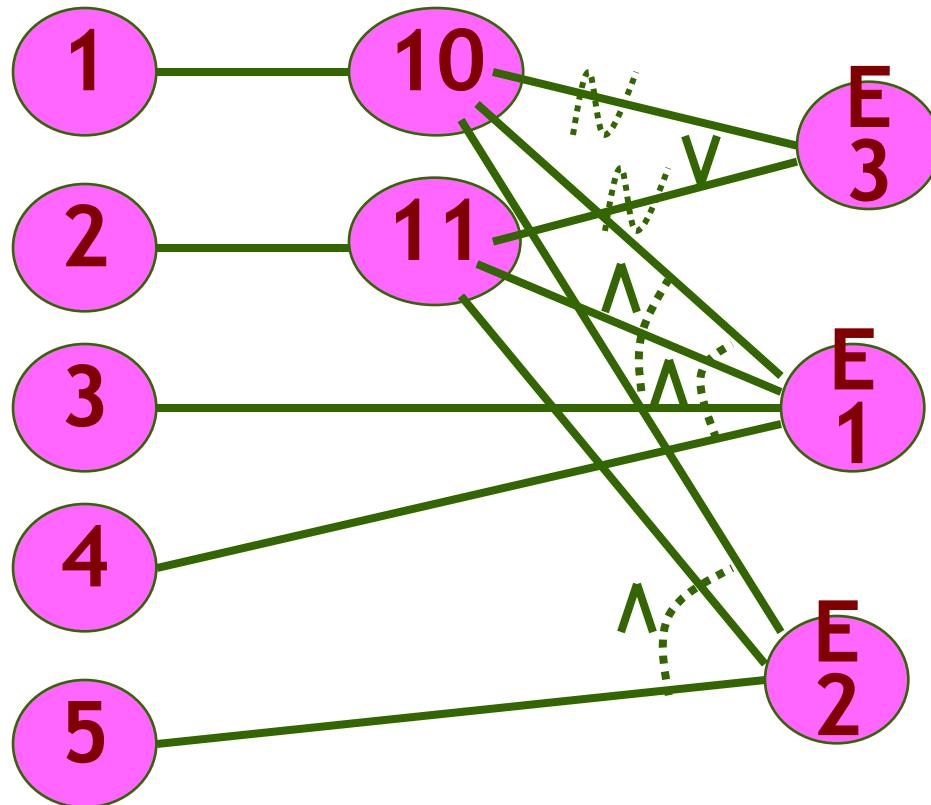
Cause effect graph- Example

- 10 ► Command is syntactically valid
- 11 ► Operands are syntactically valid.

Cause effect graph- Example

- ▶ Three effects
 - ▶ level = safe E1
 - ▶ level = high E2
 - ▶ invalid syntax E3

Cause effect graph- Example



Cause effect graph- Decision table

	Test 1	Test 2	Test 3	Test 4	Test 5	
Cause 1	I	I	I	S	I	
Cause 2	I	I	I	X	S	I = Invoked
Cause 3	I	S	S	X	X	x = don't care
Cause 4	S	I	S	X	X	s = suppressed
Cause 5	S	S	I	X	X	
Effect 1	P	P	A	A	A	
Effect 2	A	A	P	A	A	P = present
Effect 3	A	A	A	P	P	A = absent

Cause effect graph- Example

- ▶ Put a row in the decision table for each cause or effect:
 - ▶ in the example, there are five rows for causes and three for effects.

Cause effect graph- Example

- ▶ The columns of the decision table correspond to test cases.
- ▶ Define the columns by examining each effect:
 - ▶ list each combination of causes that can lead to that effect.

Cause effect graph- Example

- ▶ We can determine the number of columns of the decision table
 - ▶ by examining the lines flowing into the effect nodes of the graph.

Cause effect graph- Example

- ▶ Theoretically we could have generated $2^5=32$ test cases.
 - ▶ Using cause effect graphing technique reduces that number to 5.

Cause effect graph

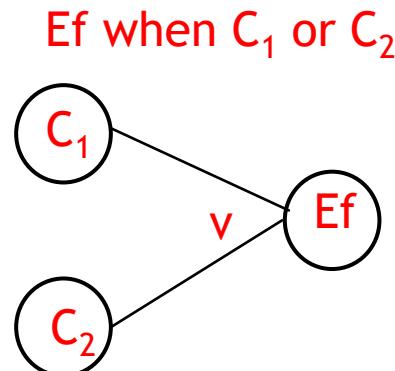
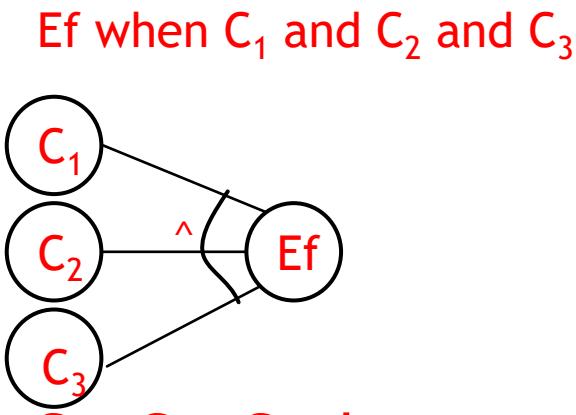
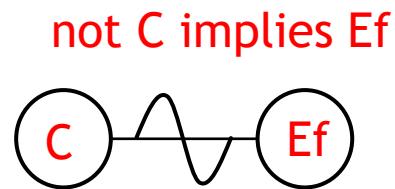
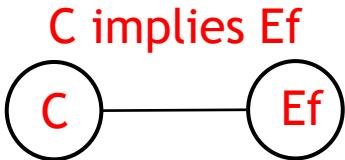
- ▶ Not practical for systems which:
 - ▶ include timing aspects
 - ▶ feedback from processes is used for some other processes.

Procedure used for the generation of tests

- ▶ Identify causes and effects by reading the requirements. Each cause and effect is assigned a unique identifier. Note that an effect can also be a cause for some other effect.
- ▶ Express the relationship between causes and effects using a cause-effect graph.
- ▶ Transform the cause-effect graph into a limited entry decision table, hereafter referred to as decision table.
- ▶ Generate tests from the decision table.

Basic elements of a cause-effect graph

- ▶ implication
- ▶ not (\sim)
- ▶ and (\wedge)
- ▶ or (\vee)



- ▶ C, C_1, C_2, C_3 denote causes.
- ▶ Ef denotes an effect.

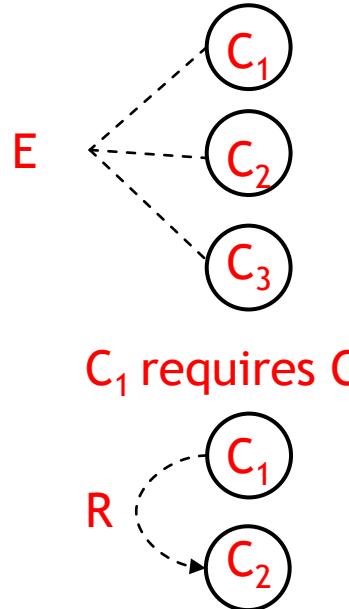
Semantics of basic elements

- ▶ C implies Ef : $\text{if}(C) \text{ then } Ef;$
- ▶ not C implies Ef : $\text{if}(\neg C) \text{ then } Ef;$
- ▶ Ef when C_1 and C_2 and C_3 : $\text{if}(C_1 \& \& C_2 \& \& C_3) \text{ then } Ef;$
- ▶ Ef when C_1 or C_2 : $\text{if}(C_1 \mid\mid C_2) \text{ then } Ef;$

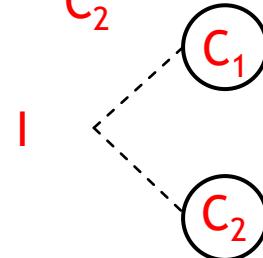
Constraints amongst causes (E,I,O,R)

- ▶ Constraints show the relationship between the causes.
- ▶ Exclusive (E)
- ▶ Inclusive (I)
- ▶ Requires (R)
- ▶ One and only one (O)

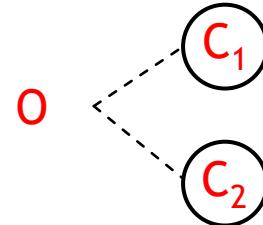
Exclusive: either C_1 or C_2 or C_3



Inclusive: at least C_1 or C_2



One and only one, of C_1 and C_2



Constraints amongst causes (E,I,O,R)

- ▶ Exclusive (E) constraint between three causes C_1 , C_2 and C_3 implies that exactly one of C_1 , C_2 , C_3 can be true.
- ▶ Inclusive (I) constraint between two causes C_1 and C_2 implies that at least one of the two must be present.
- ▶ Requires (R) constraint between C_1 and C_2 implies that C_1 requires C_2 .
- ▶ One and only one (O) constraint models the condition that one, and only one, of C_1 and C_2 must hold.

Possible values of causes constrained by E, I, R, O

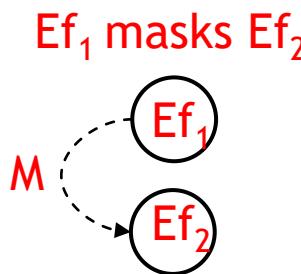
- ▶ A 0 or 1 under a cause implies that the corresponding condition is, respectively, false and true.
- ▶ The arity of all constraints, except R, is greater than 1, i.e., all except the R constraint can be applied to two or more causes; the R constraint is applied to two causes.
- ▶ A condition that is false (true) is said to be in the “0-state” (1 state).
- ▶ Similarly, an effect can be “present” (1 state) or “absent” (0 state).

Possible values of causes constrained by E, I, R, O

Constraint	Arity	Possible values		
		C1	C2	C3
$E(C_1, C_2, C_3)$	$n \geq 2$	0	0	0
		1	0	0
		0	1	0
		0	0	1
$I(C_1, C_2)$	$n \geq 2$	1	0	-
		0	1	-
		1	1	-
$R(C_1, C_2)$	$n=2$	1	1	-
		0	0	-
		0	1	-
$O(C_1, C_2, C_3)$	$n \geq 2$	1	0	0
		0	1	0
		0	0	1

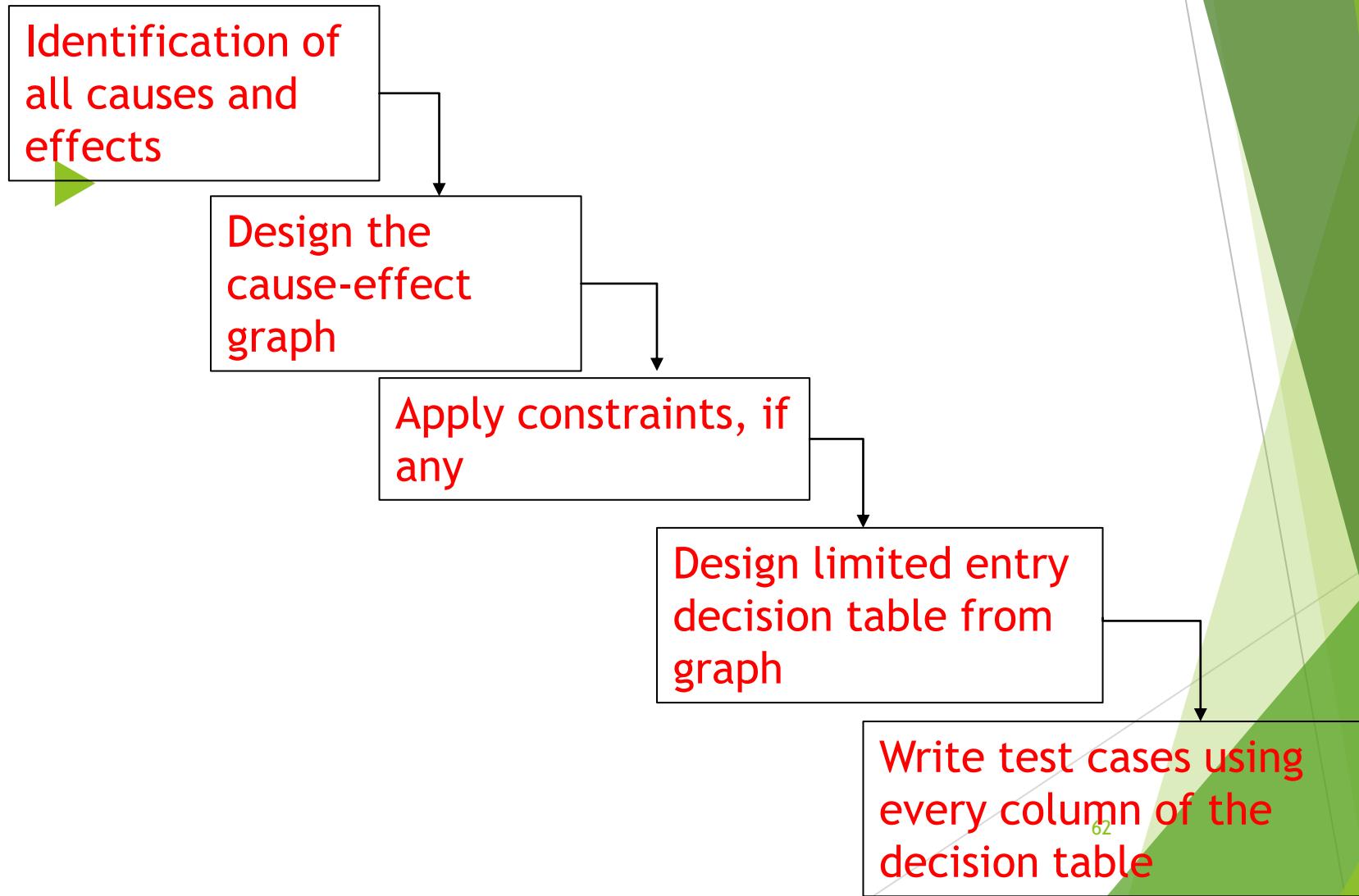
Constraint amongst effects

► Masking (M)



- Masking (M) constraint between two effects Ef_1 and Ef_2 implies that if Ef_1 is present, then Ef_2 is forced to be absent.

Steps for generating test cases using Cause-Effect Graph

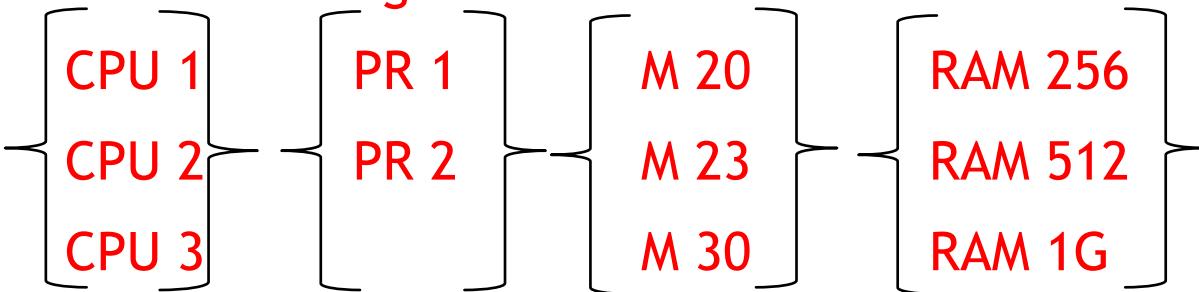


Creating Cause-Effect Graph

- ▶ The process of creating a cause-effect graph consists of two major steps.
- ▶ The causes and effects are identified by a careful examination of the requirements.
 - ▶ This process also exposes the relationships amongst various causes and effects as well as constraints amongst the causes and effects.
 - ▶ Each cause and effect is assigned a unique identifier for ease of reference in the cause-effect graph.
- ▶ The cause-effect graph is constructed to
 - ▶ express the relationships extracted from the requirements.
- ▶ When the number of causes and effects is large, say over 100 causes and 45 effects,
 - ▶ it is appropriate to use an incremental approach.

Example

- ▶ Consider the task of test generation for a GUI based computer purchase system.
- ▶ A web-based company is selling computers (CPU), printers (PR), monitors (M), and additional memory (RAM).
- ▶ An order configuration consists of one to four items as shown below.



- ▶ The GUI consists of four windows for displaying selections from CPU, Printer, Monitor and RAM and one window where any free giveaway items are displayed.
 - ▶ For each order, the buyer may select from three CPU models, two printer models, and three monitors.
 - ▶ There are separate windows one each for CPU, printer, and monitor that show the possible selections.
- For simplicity, assume that RAM is available only as an upgrade and that only one unit of each item can be purchased in one order.

Example

- ▶ Monitors M 20 and M 23 can be purchased with any CPU or as a stand-alone item.
- ▶ M 30 can only be purchased with CPU 3.
- ▶ PR 1 is available free with the purchase of CPU 2 or CPU 3.
- ▶ Monitors and printers, except for M 30, can also be purchased separately without purchasing any CPU.
- ▶ Purchase of CPU 1 gets RAM 256 upgrade.
- ▶ Purchase of CPU 2 or CPU 3 gets a RAM 512 upgrade.
- ▶ The RAM 1G upgrade and a free PR 2 is available when CPU 3 is purchased with monitor M 30.
- ▶ When a buyer selects a CPU, the contents of the printer and monitor windows are updated. Similarly, if a printer or a monitor is selected, contents of various windows are updated.

Example

- ▶ Any free printer and RAM available with the CPU selection is displayed in a different window marked “Free”.
- ▶ The total price, including taxes, for the items purchased is calculated and displayed in the “Price” window.
- ▶ Selection of a monitor could also change the items displayed in the “Free” window.
- ▶ Sample configurations and contents of the “Free” window are given below.

Items purchased	“Free” window	Price
CPU 1	RAM 256	\$499
CPU 1, PR 1	RAM 256	\$628
CPU 2, PR 2, M 23	PR 1, RAM 512	\$2257
CPU 3, M 30	PR 2, RAM 1G	\$3548

Example

- ▶ The first step in creating cause-effect graphing is to read the requirements carefully and make a list of causes and effects.
- ▶ A unique identifier, C_1 , through C_8 , has been assigned to each cause.
- ▶ Each cause listed below is a condition that can be true or false.
- ▶ C_1 : Purchase CPU 1.
- ▶ C_2 : Purchase CPU 2.
- ▶ C_3 : Purchase CPU 3.
- ▶ C_4 : Purchase PR 1.
- ▶ C_5 : Purchase PR 2.
- ▶ C_6 : Purchase M 20.
- ▶ C_7 : Purchase M 23.
- ▶ C_8 : Purchase M 30.
- ▶ For example, C_8 is true if monitor M 30 is purchased.

Example

- ▶ Note that while it is possible to order any of the items listed above, the GUI will update the selection available depending on which CPU, or any Other item, is selected.
- ▶ For example, if CPU 3 is selected for purchase then monitors M 20 and M 23 will not be available in the monitor selection window.
- ▶ Similarly, if monitor M 30 is selected for purchase, then CPU 1 and CPU 2 will not be available in the CPU window.
- ▶ Next, we identify the effects.
- ▶ In this example, the application software calculates and displays the list of items available free with the purchase and the total price.
- ▶ Hence, the effect is in terms of the contents of the “Free” and “Price” windows.

Example

- ▶ Calculation of the total purchase price depends on the items purchased and the unit price of each item.
- ▶ The unit price is obtained by the application from a price database.
- ▶ The price calculation and display is a cause that creates the effect of displaying the total price.
- ▶ For simplicity, we ignore the price related cause and effect.
- ▶ The set of effects in terms of the contents of the “Free” display window are listed below.

Ef_1 : RAM 256

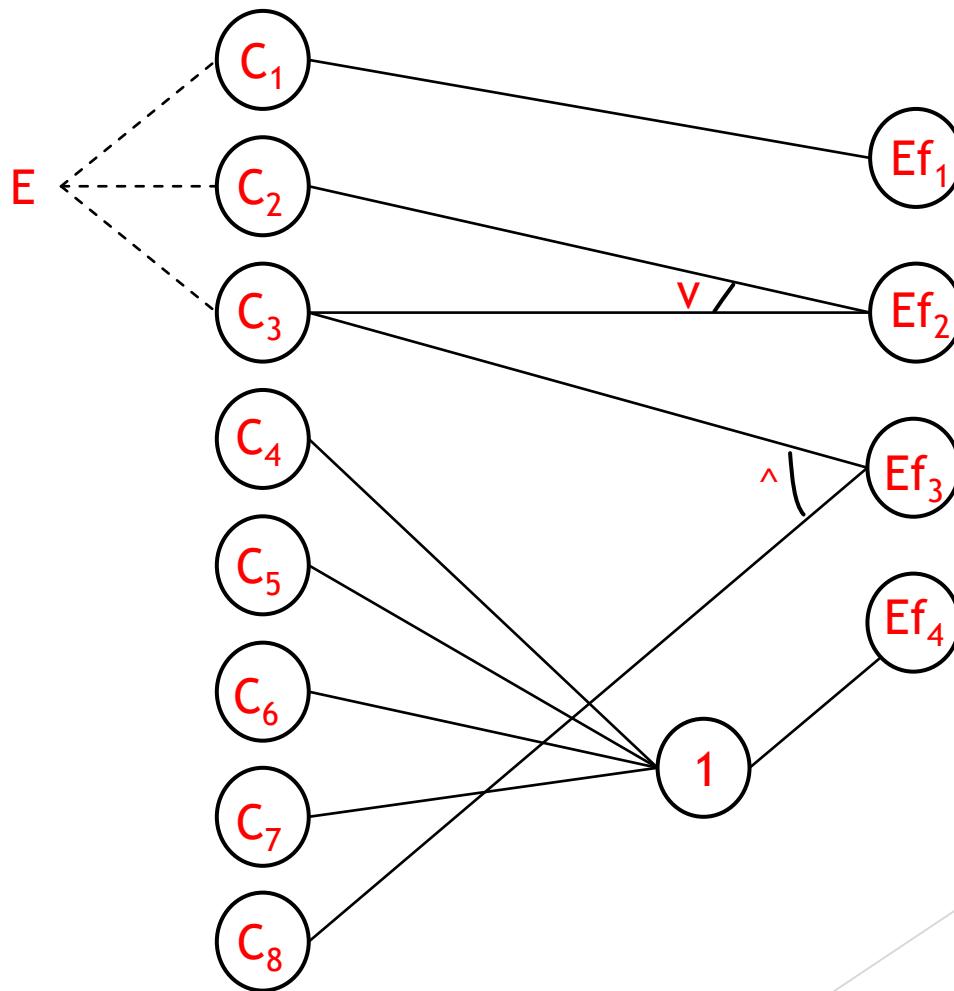
Ef_2 : RAM 512 and PR 1.

Ef_3 : RAM 1G and PR 2.

Ef_4 : No giveaway with this item.

Example

- Following shows the complete graph that expresses the relationships between C_1 through C_8 and effects Ef_1 , through Ef_4 .



Example

- ▶ From the above cause-effect graph, followings can be inferred.
 - ▶ C_1, C_2 and C_3 are constrained using the E (exclusive) relationship.
 - ▶ This expresses the requirement that only one CPU can be purchased in one order.
 - ▶ Similarly, C_3 and C_8 are related via the R (requires) constraint to express the requirement that monitor M 30 can only be purchased with CPU3.
 - ▶ Relationships amongst causes and effects are expressed using the basic elements.
- ▶ There is an intermediate node labelled 1 in the graph.
- ▶ Such intermediate nodes are often useful when an effect depends on conditions combined using more than one operator, for example, $(C_1 \wedge C_2) \vee C_3$.
- ▶ Also it can be noted that
 - ▶ Purchase of printers and monitors without any CPU leads to no free item (Ef_4).

Example

- The relationships between effects and causes shown in the graph can be expressed in terms of Boolean expressions as follows:

$$Ef_1 = C_1$$

$$Ef_2 = C_2 \vee C_3$$

$$Ef_3 = C_3 \wedge C_8$$

$$Ef_4 = C_4 \wedge C_5 \wedge C_6 \wedge C_7$$

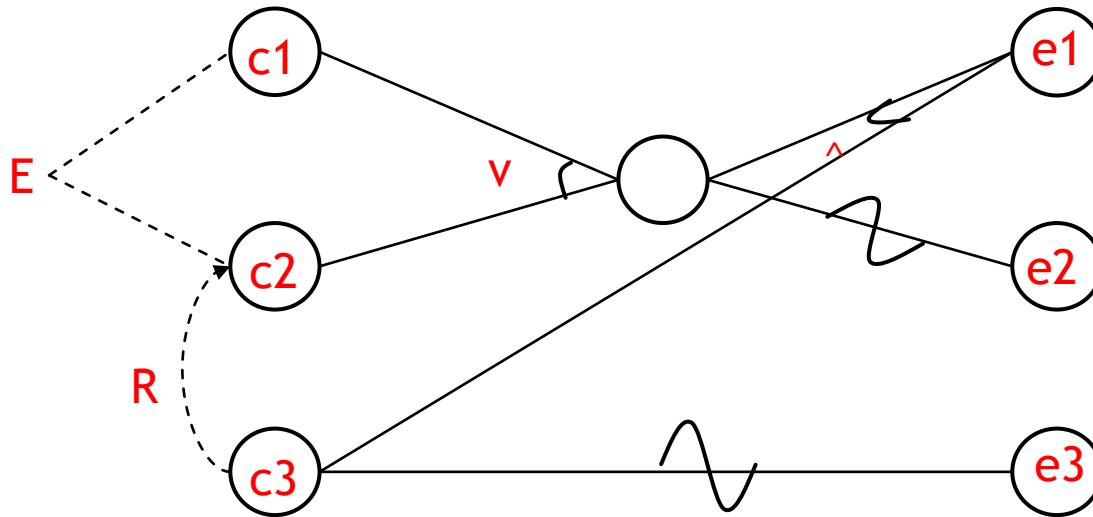
Another example

- ▶ Consider the example of keeping the record of marital status and number of children of a citizen.
- ▶ The value of marital status must be 'U' or 'M'.
- ▶ The value of the number of children must be digit or null in case a citizen is unmarried.
- ▶ If the information entered by the user is correct then an update is made.
- ▶ If the value of marital status of the citizen is incorrect, then the error message 1 is issued.
- ▶ Similarly, if the value of the number of children is incorrect, then the error message 2 is issued.

Answer

- ▶ Causes are
 - ▶ c1: marital status is U
 - ▶ c2: marital status is M
 - ▶ c3: number of children is a digit
- ▶ Effects are
 - ▶ e1: updation made
 - ▶ e2: error message 1 is issued
 - ▶ e3: error message 2 is issued

Answer



- ▶ There are two constraints
 - ▶ Exclusive (between c1 and c2) and
 - ▶ Requires (between c3 and c2)
- ▶ Causes c1 and c2 cannot occur simultaneously.
- ▶ For cause c3 to be true, cause c2 has to be true.

Heuristics to avoid combinatorial explosion

- ▶ While tracing back through a cause-effect graph,
 - ▶ we generate combinations of causes that set an intermediate node, or an effect, to a 0 or 1 state.
- ▶ Doing so in a brute force manner could lead to a large number of combinations.
- ▶ In the worst case, if n causes are related to an effect e , then the maximum number of combinations that bring e to a 1-state is 2^n .
- ▶ As tests are derived from the combinations of causes,
 - ▶ large values of n could lead to an exorbitantly large number of tests.
- ▶ We avoid such a combinatorial explosion by using simple heuristics related to the “AND” (^) and “OR” (v) nodes.
- ▶ Heuristics are based on the assumption that
 - ▶ certain types of errors are less likely to occur than others.

Heuristics to avoid combinatorial explosion

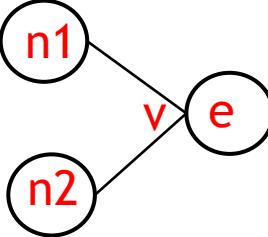
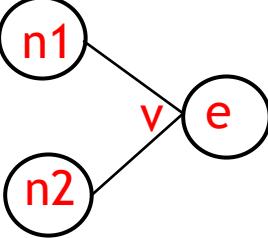
- ▶ Thus, while applying the heuristics to generate test
 - ▶ inputs will likely lead to a significant reduction in the number of tests generated,
 - ▶ it may also discard tests that would have revealed a program error.
- ▶ Hence, one must apply the heuristics with care and only when
 - ▶ the number of tests generated without their application is too large to be useful in practice.

Heuristics used during the generation of input combinations from a cause-effect graph

- ▶ The heuristics are labelled H_1 through H_4 .
- ▶ The leftmost column shows the node type in the cause-effect graph.
- ▶ The center column is the desired state of the dependent node.
- ▶ The rightmost column is the heuristics for generating combinations of inputs to the nodes that effect the dependent node e .
- ▶ For simplicity, only two nodes n_1 and n_2 and the corresponding effect e are shown.

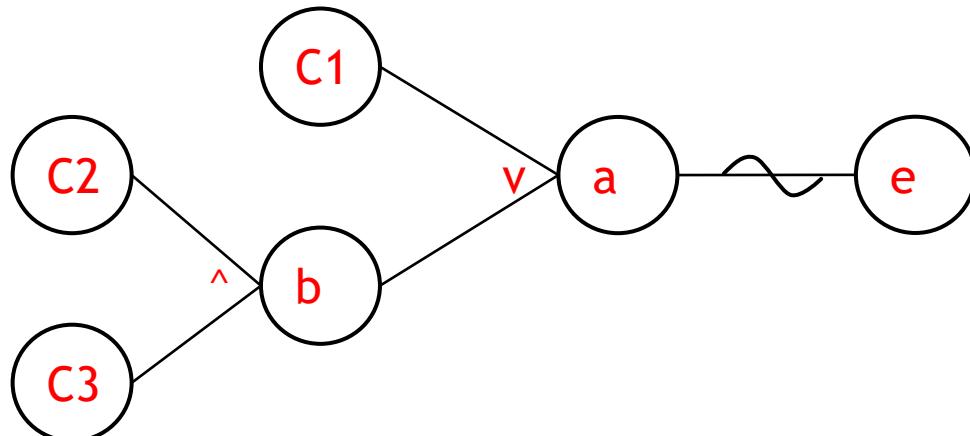
Heuristics used during the generation of input combinations from a cause-effect graph

Table 1

Node type	Desired state of e	Input combinations
	0	H1: Enumerate all combinations of inputs to n_1 and n_2 such that $n_1=n_2=0$.
	1	H2: Enumerate all combinations of inputs to n_1 and n_2 other than those for which $n_1=n_2=0$.
	0	H3: Enumerate all combinations of input to n_1 and n_2 such that each of the possible combinations of n_1 and n_2 appears exactly once and $n_1=n_2=1$ does not appear. Note that for two nodes, there are three such combinations: (0,0), (0,1) and (1,0). In general, for k nodes combined using the logical and operator to form e , there are 2^k-1 such combinations.
	1	H4: Enumerate all combinations of inputs to n_1 and n_2 such that $n_1=n_2=1$.

Example

- ▶ Consider the following cause-effect graph.
- ▶ We have at least two choices while tracing backwards to derive the necessary combinations.
- ▶ Derive all combinations first and then apply the heuristics.
- ▶ Derive combinations while applying the heuristics.



Example

- ▶ Suppose we require node e to be 1.
- ▶ Tracing backwards, this requirement implies that node a must be a 0 (zero).
- ▶ If we trace backwards further, without applying any heuristic, we obtain the following seven combinations of causes that bring e to 1 state.
- ▶ The last column lists the inputs to node a .
- ▶ The combinations that correspond to the inputs to node a listed in the rightmost column are separated by horizontal lines.

Example: Generating tests using heuristics

- ▶ First we note that node a matches the OR-node shown in the top half of Table 1.
- ▶ As we want the state of node a to be 0, heuristic H1 applies in this situation.
- ▶ H1 asks to enumerate all combinations of inputs to node a such that C1 and node b are 0.
- ▶ (0,0) is the only such combination and is listed in the last column of the following table.

	C1	C2	C3	Inputs to node a
1	0	0	0	C1=0, b=0
2	0	0	1	
3	0	1	0	
4	0	1	1	C1=0, b=1
5	1	0	0	C1=1, b=0
6	1	0	1	
7	1	1	0	

Example: Generating tests using heuristics

- ▶ Let us begin with (0,0)
- ▶ No heuristic applies to C1 as it has no preceding nodes.
- ▶ Node *b* is an AND-node as shown in the bottom half of Table.
- ▶ We want node *b* to be 0 and therefore H3 applies.
- ▶ In accordance with H3 we generate three combinations of inputs to node *b*: (0,0), (0,1) and (1,0).
- ▶ Notice that combination (1,1) is forbidden.
- ▶ Joining these combinations of C2 and C3 with C1=0,
 - ▶ we obtain the first three combinations listed in the preceding table.

Example: Generating tests using heuristics

- ▶ Though not required here, suppose that we were to consider the combination $C1=0, b=1$.
- ▶ Heuristic H4 applies in this situation.
- ▶ As both $C2$ and $C3$ are causes with no preceding nodes,
 - ▶ the only combination we obtain now is $(1,1)$.
- ▶ Combining this with $C1=0$ we obtain sequence 4 listed in the preceding table.

Example: Generating tests using heuristics

- ▶ We have completed derivation of combinations using the heuristics listed in Table 1.
- ▶ Note that the combinations listed above for $C1=1, b=0$ are not required.
- ▶ Thus, we have obtained only three combinations instead of the seven enumerated earlier.
- ▶ The reduced set of combinations is listed below.

	C1	C2	C3	Inputs to node <i>a</i>
1	0	0	0	$C1=0, b=0$
2	0	0	1	
3	0	1	0	

Example: Generating tests using heuristics

- ▶ Let us examine the rationale underlying the various heuristics for reducing the number of combinations.
- ▶ Heuristics H1 does not save us on any combinations.
- ▶ The only way an OR-node can cause its effect e to be 0 is for all its inputs to be 0.
- ▶ H1 suggests that we enumerate all such combinations.
- ▶ Heuristics H2 suggests we use all combinations that cause e to be 1 except those that cause $n_1=n_2=0$.
- ▶ To understand the rationale underlying H2 consider a program required to generate an error message when condition c_1 or c_2 is true.
- ▶ A correct implementation of this requirement is given below.

```
if(c1 v c2)printf("Error");
```

Example: Generating tests using heuristics

- ▶ Now consider the following erroneous implementation of the same requirement.

```
if(c1 v ¬c2)printf("Error");
```

- ▶ A test that sets both c1 and c2 true
 - ▶ will not be able to detect an error in the implementation above
 - ▶ if short circuit evaluation is used for Boolean expressions.
- ▶ However, a test that sets c1=0 and c2=1 will be able to detect this error.
- ▶ Hence H2 saves us from generating all input combinations that generate the pair (1,1) entering an effect in an OR-node.

Example: Generating tests using heuristics

- ▶ Heuristics H3 saves us from repeating the combinations of n_1 and n_2 .
- ▶ Once again this could save us a lot of tests.
- ▶ The assumption here is that
 - ▶ any error in the implementation of e will be detected by
 - ▶ tests that cover different combinations of n_1 and n_2 .
- ▶ Thus, there is no need to have two or more tests that
 - ▶ contain the same combination of n_1 and n_2 .

Example: Generating tests using heuristics

- ▶ Lastly, H4 for the AND-node is analogous to H1 for the OR-node.
- ▶ The only way an AND-node can cause its effect e to be 1
 - ▶ is for all its inputs to be 1.
- ▶ H4 suggests that we enumerate all such combinations.
- ▶ We stress, once again, that
 - ▶ while the heuristics will likely reduce the set of tests generated using cause-effect graphing,
 - ▶ they might also lead to useful tests being discarded.
- ▶ Of course, in general and prior to the start of testing,
 - ▶ it is almost impossible to know which of the test cases discarded will be useless and which ones useful.

Decision Table from cause-effect graph

- ▶ Each column of the decision table represents a combination of input values, and hence a test.
- ▶ There is one row for each condition and effect.
- ▶ Thus the table decision table can be viewed as an $N \times M$ matrix with
 - ▶ N being the sum of the number of conditions and effects and
 - ▶ M the number of tests.
- ▶ Each entry in the decision table is a 0 or 1
 - ▶ depending on whether or not the corresponding condition is false or true, respectively.
- ▶ For a row corresponding to an effect, an entry is 0 or 1
 - ▶ if the effect is not present or present, respectively.

Procedure for generating a decision table from a cause-effect graph

- ▶ **Input:** A cause-effect graph containing causes C_1, C_2, \dots, C_p and effects Ef_1, Ef_2, \dots, Ef_q .
- ▶ **Output:** A decision table DT containing $N=p+q$ rows and M columns, where M depends on the relationship between the causes and effects as captured in the cause-effect graph.
- ▶ **Procedure:** CEGDT
 - ▶ `/*`
 - ▶ i is the index of the next effect to be considered.
 - ▶ $next_dt_col$ is the next empty column in the decision table.
 - ▶ V_k : a vector of size $p+q$ containing 1's and 0's. V_j , $1 \leq j \leq p$, indicates the state of condition C_j and V_l , $p < l \leq p+q$, indicates the presence or absence of effect Ef_{l-p} .
 - ▶ `*/`

Procedure for generating a decision table from a cause-effect graph

- ▶ **Step 1** Initialize DT to an empty decision table.

next_dt_col=1.

- ▶ **Step 2** Execute the following steps for $i=1$ to q .

- 2.1 Select the next effect to be processed.

Let $e=Ef_i$.

- 2.2 Find combinations of conditions that cause e to be present.

Assume that e is present. Starting at e , trace the cause-effect graph backwards and determine the combinations of conditions C_1, C_2, \dots, C_p that lead to e being present.

Let V_1, V_2, \dots, V_{mi} be the combinations of causes that lead to e being present, i.e. in 1 state, and hence $m_i \geq 1$. Set $V_k(l), p < l \leq p+q$ to 0 or 1 depending on whether effect Ef_{l-p} is present or not for the combination of all conditions in V_k .

Procedure for generating a decision table from a cause-effect graph

2.3 *Update the decision table.*

Add V_1, V_2, \dots, V_{mi} to the decision table as successive columns starting at *next_dt_col*.

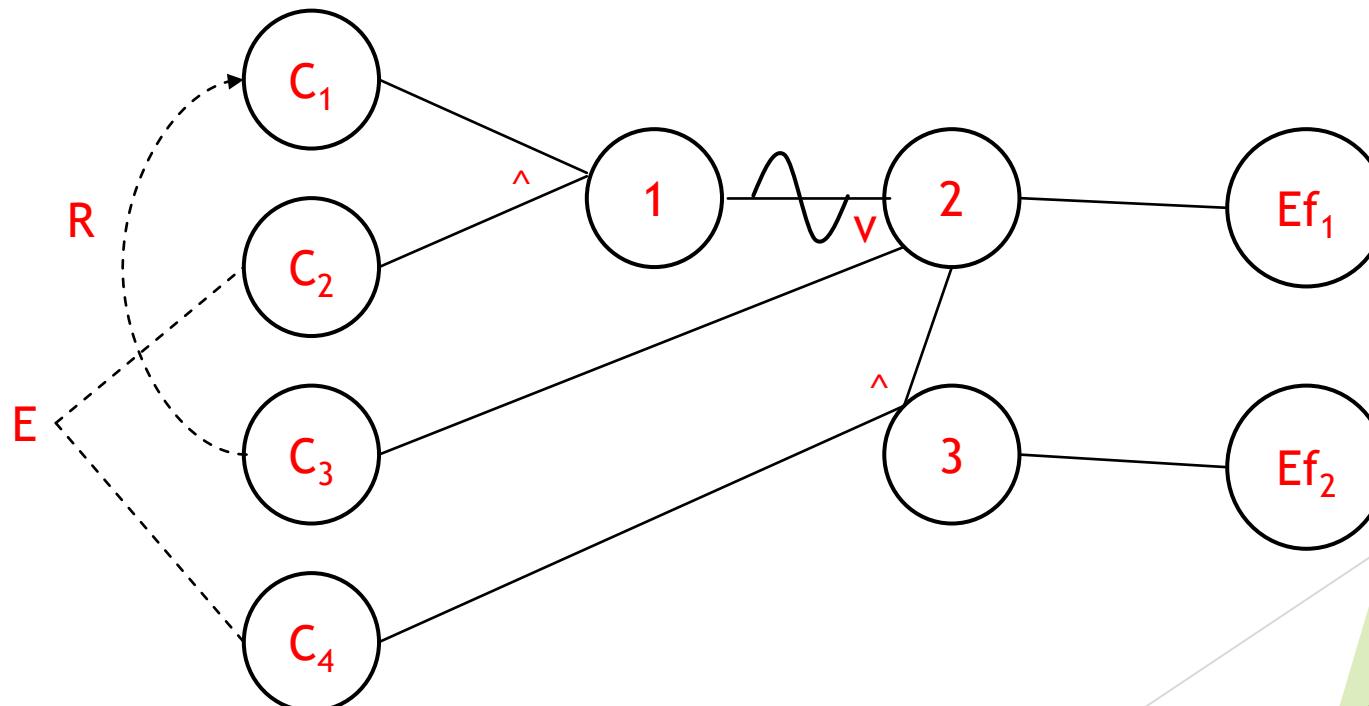
2.4 *Update the next available column in the decision table.*

next_dt_col=next_dt_col+m_i. At the end of this procedure, *next_dt_col-1* is the number of tests generated.

► End of Procedure CEGDT

Example

- ▶ Consider the cause effect graph shown below.
- ▶ It shows four causes labelled C_1, C_2, C_3 and C_4 and two effects labeled Ef_1 and Ef_2 .
- ▶ There are three intermediate nodes labeled 1, 2 and 3.



Example

- ▶ Step 1: Set $next_dt_col=1$ to initialize the decision table to empty.
- ▶ Next, $i=1$ and in accordance with Step 2.2, $e=Ef_1$.
- ▶ In accordance with Step 2.2, trace backwards from e to determine combinations that will cause e to be present.
- ▶ e must be present when node 2 is in 1-state.
- ▶ Moving backwards from node 2 in the cause-effect graph, any of the following three combinations of states of nodes 1 and C_3 will lead to e being present: $(0,1)$, $(1,1)$ and $(0,0)$.
- ▶ Node 1 is also an internal node and hence move further back to obtain the values of C_1 and C_2 that effect node 1.
- ▶ Combinations of C_1 and C_2 that brings node 1 to the 1-state is $(1,1)$ and combinations that bring it to 0-state are $(1,0)$, $(0,1)$ and $(0,0)$.

Example

- ▶ Combining this information with that derived earlier for nodes 1 and C_3 , we obtain the following seven combinations of C_1 , C_2 and C_3 that cause e to be present.

1 0 1

0 1 1

0 0 1

1 1 1

1 0 0

0 1 0

0 0 0

▶ Next, C_3 requires C_1 , which implies that C_1 must be in 1-state for C_3 to be in 1-state.

▶ This constraint makes infeasible the second and third combinations above.

Example

- In the end, we obtain the following five combinations of the four causes that lead to e being present.

1 0 1

1 1 1

1 0 0

0 1 0

0 0 0

- Setting C_4 to 0 and appending the values of Ef_1 and Ef_2 , we obtain the following five vectors.

V_1 1 0 1 0 1 0

V_2 1 1 1 0 1 0

V_3 1 0 0 0 1 0

V_4 0 1 0 0 1 0

V_5 0 0 0 0 1 0

Example

- ▶ Note that $m_1=5$ in Step 2.
- ▶ This completes the application of Step 2.2.
- ▶ The five vectors are transposed and added to the decision table starting at column *next_dt_col* which is 1.
- ▶ The decision table at Step 2.3 follows

	1	2	3	4	5
C_1	1	1	1	0	0
C_2	0	1	0	1	0
C_3	1	1	0	0	0
C_4	0	0	0	0	0
Ef_1	1	1	1	1	1
Ef_2	0	0	0	0	0

- ▶ We update *next_dt_col* to 6, increment *i* to 2 and get back to Step 2.1.

Example

- ▶ We now have $e=Ef_2$.
- ▶ Tracing backwards, we find that for e to be present, node 3 must be in the 1-state.
- ▶ This is possible with only one combination of node 2 and C_4 , which is (1,1).
- ▶ Earlier we derived the combinations of C_1 , C_2 and C_3 that lead node 2 into the 1-state.
- ▶ Combining these with the value of C_4 , we arrive at the following combination of causes that lead to the presence of Ef_2 .

1	0	1	1
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1

Example

- ▶ From the cause-effect graph, note that C_2 and C_4 cannot be present simultaneously.
- ▶ Hence, we discard the second and fourth combinations from the list above and obtain the following three feasible combinations.

1 0 1 1

1 0 0 1

0 0 0 1

- ▶ Appending the corresponding values of Ef_1 and Ef_2 to each of the above combinations, we obtain the following three vectors.

V_1 1 0 1 1 1 1

V_2 1 0 0 1 1 1

V_3 0 0 0 1 1 1

Example

- ▶ Transposing the vectors listed above and appending them as three columns to the existing decision table, we obtain the following

	1	2	3	4	5	6	7	8
C_1	1	1	1	0	0	1	1	0
C_2	0	1	0	1	0	0	0	0
C_3	1	1	0	0	0	1	0	0
C_4	0	0	0	0	0	1	1	1
Ef_1	1	1	1	1	1	1	1	1
Ef_2	0	0	0	0	0	1	1	1

- ▶ Next, we update *next_dt_col* to 9.
- ▶ Of course, doing so is useless as the loop set up in Step 2 is now terminated.
- ▶ The decision table listed above is the output.¹⁰¹

Test generation from a decision table

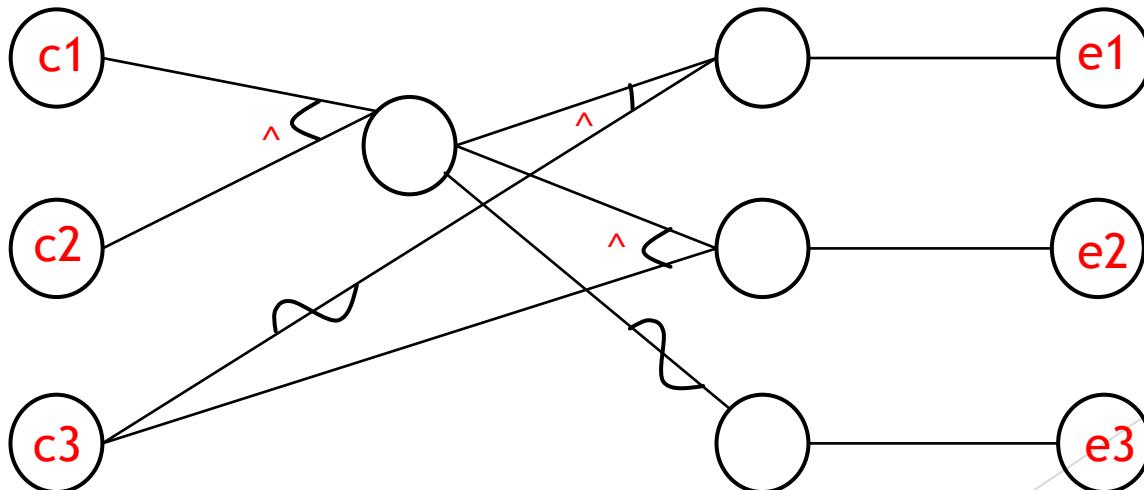
- ▶ Test generation from a decision table is relatively forward.
- ▶ Each column in the decision table generates at least one test input.
- ▶ Note that each combination might be able to generate more than one test when a condition in the cause-effect graph can be satisfied in more than one way.
- ▶ For example, consider the following cause:
- ▶ $C: x < 99$
- ▶ The condition above can be satisfied by many values such as $x=1$ and $x=49$.
- ▶ Also, C can be made false by many values of x such as $x=100$ and $x=999$.
- ▶ Thus, one might have a choice of values of input variables while generating tests using columns from a decision table

Example

- ▶ A tourist of age greater than 21 years and having a clean driving record is supplied a rental car.
- ▶ A premium amount is also charged if the tourist is on business,
- ▶ Otherwise, it is not charged.
- ▶ If the tourist is less than 21 year old, or does not have a clean driving record,
 - ▶ The system will display the following message: “Car cannot be supplied”.

Answer

- ▶ Causes are
 - ▶ c1: Age is over 21
 - ▶ c2: Driving record is clean
 - ▶ c3: Tourist is on business
- ▶ Effects are
 - ▶ e1: Supply a rental car without premium charge
 - ▶ e2: Supply a rental car with premium charge
 - ▶ e3: Car cannot be supplied



Decision Table and Test Cases

	1	2	3	4
c1: Over 21?	F	T	T	T
c2: Driving record clean?	-	F	T	T
c3: On business?	-	-	F	T
e1: Supply a rental car without premium charge			X	
e2: Supply a rental car with premium charge				X
e3: Car cannot be supplied	X	X		

Test Case	Age	Driving_record_clean	On_business	Expected Output
1	20	Yes	Yes	Car cannot be supplied
2	26	No	Yes	Car cannot be supplied
3	62	Yes	No	Supply a rental car without premium charge
4	62	Yes	Yes	Supply a rental car with ₁₀₅ premium charge

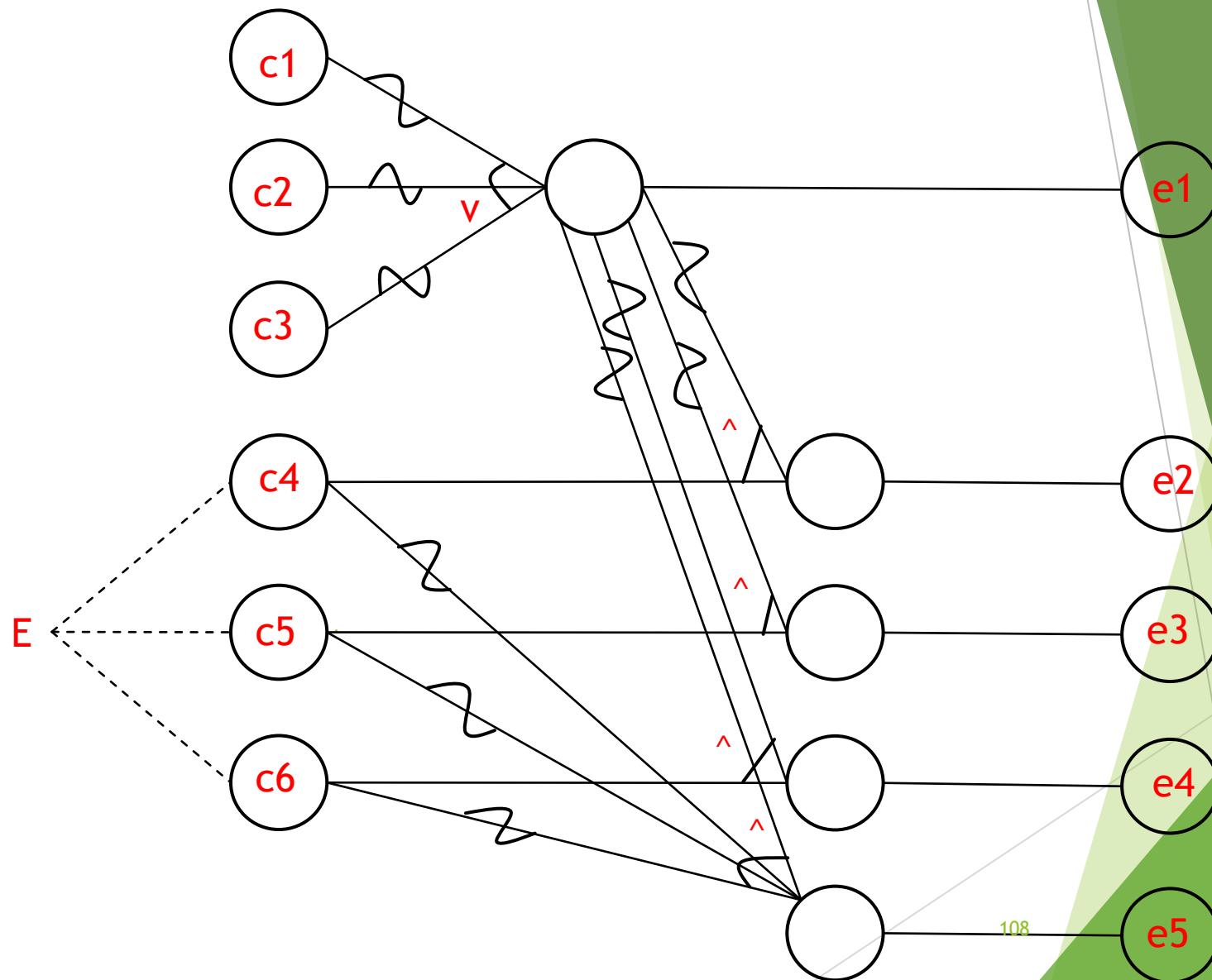
Example 2: Triangle Classification Problem

- ▶ Consider a program for classification of a triangle.
- ▶ Its input is a triple of positive integers (say a, b and c) and the input values are greater than zero and less than or equal to 100.
- ▶ The triangle is classified according to the following rules:
 - ▶ Right angled triangle: $c^2=a^2+b^2$ or $a^2=b^2+c^2$ or $b^2=c^2+a^2$
 - ▶ Obtuse angled triangle: $c^2>a^2+b^2$ or $a^2>b^2+c^2$ or $b^2>c^2+a^2$
 - ▶ Acute angled triangle: $c^2< a^2+b^2$ or $a^2< b^2+c^2$ or $b^2< c^2+a^2$
 - ▶ The program output may have one of the following words: [Acute angled triangle, Obtuse angled triangle, Right angled triangle, Invalid triangle, Input values are out of range]

Answer

- ▶ Causes are:
 - ▶ c1: side “a” is less than the sum of sides “b” and “c”.
 - ▶ c2: side “b” is less than the sum of sides “a” and “c”.
 - ▶ c3: side “c” is less than the sum of sides “a” and “b”.
 - ▶ c4: square of side “a” is equal to the sum of squares of sides “b” and “c”.
 - ▶ c5: square of side “a” is greater than the sum of squares of sides “b” and “c”.
 - ▶ c6: square of side “a” is less than the the sum of squares of sides “b” and “c”.
- ▶ Effects are:
 - ▶ e1: Invalid triangle
 - ▶ e2: Right angle triangle
 - ▶ e3: Obtuse angled triangle
 - ▶ e4: Acute angled triangle
 - ▶ e5: Impossible stage

Cause-Effect Graph



Decision Table

	1	2	3	4	5	6	7	8	9	10	11
c1: $a < b+c$	F	T	T	T	T	T	T	T	T	T	T
c2: $b < a+c$	-	F	T	T	T	T	T	T	T	T	T
c3: $c < a+b$	-	-	F	T	T	T	T	T	T	T	T
c4: $a^2 = b^2 + c^2$	-	-	-	T	T	T	T	F	F	F	F
c5: $a^2 > b^2 + c^2$	-	-	-	T	T	F	F	T	T	F	F
c6: $a^2 < b^2 + c^2$	-	-	-	T	F	T	F	T	F	T	F
e1: Invalid triangle	X	X	X								
e2: Right angle triangle							X				
e3: Obtuse angled triangle									X		
e4: Acute angled triangle										X	
e5: Impossible					X	X	X	X			X

Thank You

Concolic Testing: A modern software testing technique

**Dr. Durga Prasad Mohapatra
Professor**

Department of CSE, NIT Rourkela



Seminar Outline

1 [Introduction](#)

2 [Fundamental Ideas](#)

3 [Survey of Related works](#)



Introduction

Software Testing is an important phase in SDLC.

- Helps to achieve software dependability and improve quality.
 - Time consuming: Approximately 40% of software development time is devoted to testing.
-
- Testing can be done in two ways – Manual vs. Automated.
 - **An automation tool** for test case generation can effectively reduce the time required in testing.
 - **Automation tool** can be designed to generate test cases for different test coverage criteria.



Reliability and Coverage Model -By Prof. Aditya Mathur

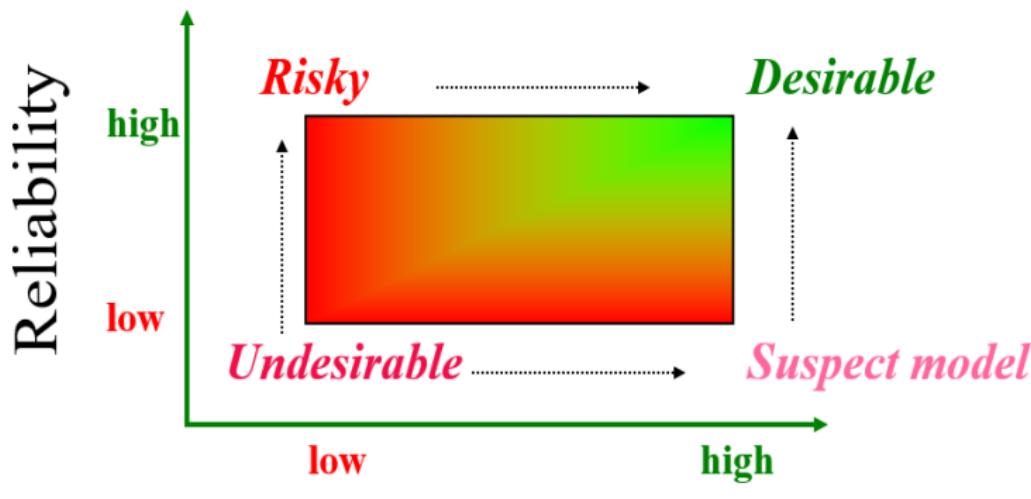


Figure 1: Reliability and Coverage



Software Testing

“No product of human intellect comes out right the first time. We rewrite sentences, rip out knitting stitches, replant gardens, remodel houses, and repair bridges. Why should software be any different?”

- By Wiener, Ruth: Digital Woes, Why We Should Not Depend on Software.



Software Testing

- The purpose of the verification process is to detect and report errors that have been introduced in the development process.
- The verification process must ensure that the produced software implements intended function completely and correctly, while avoiding unintended function.
- Verification is an integral process, which is coupled with every development step. Testing quality at the end of the life cycle is impractical.



Software Testing: Coverage

- Coverage refers to the extent to which a given verification activity has satisfied its objectives: in essence, providing an exit criteria for when to stop. That is, what is “enough” is defined in terms of coverage.
- Coverage is a measure, not a method or a test. As a measure, coverage is usually expressed as the percentage of an activity that is accomplished.
- **Our goal, then, should be to provide enough testing to ensure that the probability of failure due to hibernating bugs is low enough to accept. “Enough” implies judgment.**



RTCA/DO178-B/C standards

- RTCA stands for Radio Technical Commission for Aeronautics.
- DO-178B (and DO-278) are used to assure safety of avionics software.
- DO-178C includes the coverage analysis of Object-Oriented Programs used in safety of avionics software.
- These documents provide guidance in the areas of SW development, configuration management, verification and the interface to approval authorities (e.g., FAA, EASA).



Levels of Software

- Different failure conditions require different software conditions → 5 levels

Failure Condition	Software Level
Catastrophic	Level A
Hazardous/Severe - Major	Level B
Major	Level C
Minor	Level D
No Effect	Level E

Figure 2: Levels of Software



Relation of Coverages

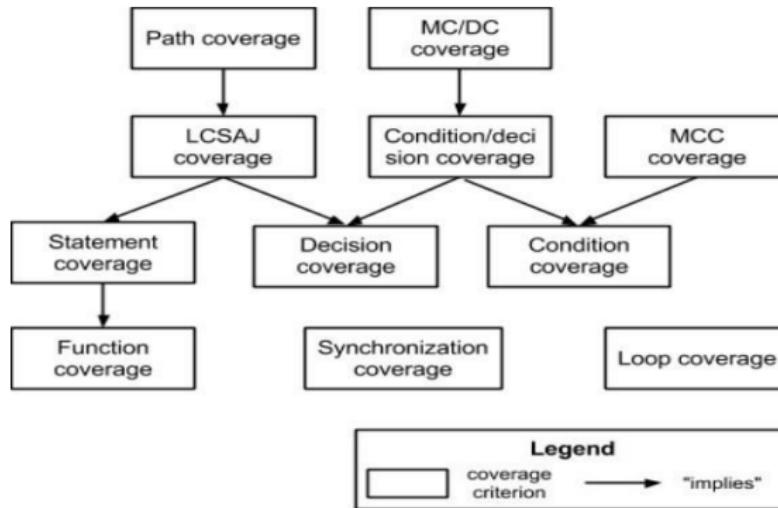


Figure 3: Relation of Coverages



LCSAJ Coverage

- **LCSAJ** stands for Linear Code Sequence and Jump.
- It is a white box **testing** technique to identify the **code coverage**.
- It begins at the start of the program or branch and ends at the end of the program or the branch.
- **LCSAJ** is ordinarily equivalent to **statement coverage**.



Types of Structural Coverage

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once		•	•	•	•	•
Every statement in the program has been invoked at least once	•					
Every decision in the program has taken all possible outcomes at least once		•		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			•	•	•	•
Every condition in a decision has been shown to independently affect that decision's outcome					•	• ^b
Every combination of condition outcomes within a decision has been invoked at least once						•

weakest → strongest



Figure 4: Types

Definitions

Predicate / Boolean Expression

A predicate is an expression that evaluates to a boolean value, and which is required for our approach.

A simple example is: $((a > b) \vee C) \wedge p(x)$. Predicates may contain boolean variables, non-boolean variables that are compared with relational operators, and calls to function that return a boolean value, all three of which may be joined with logical operators.



Definitions

Predicate Coverage

For each $p \in P$, Test Requirement (TR) for predicate coverage contains two requirements: p evaluates to true, and p evaluates to false.



Definitions

Clause / Atomic Condition

A clause is a predicate that does not contain any of the logical operators.

The predicate $((a > b) \vee C) \wedge p(x)$ contains three clauses; a relational expression $(a > b)$, a boolean variable C and a boolean function call $p(x)$.



Definitions

Clause Coverage

For each $c \in C$, TR for clause coverage contains two requirements: c evaluates to true, and c evaluates to false.



Definitions

Statement Coverage

- The statement coverage based strategy aims to design the test cases so as to execute every statement in a program at least once.
- The principle idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.



Definitions

Branch Coverage

Each decision should take all possible outcomes at least once either true or false.

For example if($m > n$), the test cases are (1) $m \leq n$, (2) $m > n$



Example

```
1 int computeGCD(x,y)
2     int x,y;
3     {
4         while (x!=y){
5             if(x>y) then
6                 x=x-y;
7             else
8                 y=y-x;
9         }
10    return x;
11 }
```

Figure 5: Euclid's GCD computation program



Test cases for Statement Coverage

- To design the test cases for the statement coverage, the conditional expression of the while statements needs to be made true and the conditional expression of the if statement needs to be made both true and false.
- By choosing the test set $\{(x=3,y=3), (x=4,y=3), (x=3,y=4)\}$, all the statements of the program would be executed at-least once.



Test cases for Statement Coverage

- For the GCD program, the test cases for branch coverage can be $\{(x=3,y=3), (x=3,y=2), (x=4,y=3),(x=3,y=4)\}$.



Observation

- It is easy to show that branch coverage based testing is a stronger testing than statement coverage-based testing.
- We can prove this by showing that branch coverage ensures statement coverage, but not vice-versa.



Definitions

Concolic Testing

- The concept of CONCOLIC testing combines the CONConcrete constraints execution and symbOLIC constraints execution to automatically generate test cases for full path coverage.
- This testing generates test suites by executing the program with random values.
- At execution time both concrete and symbolic values are saved for execution path.
- During execution, the variables are stored in some symbolic values such as x_0 , and y_0 , instead of x and y.
- The next iteration of the process forces the selection of different paths.



Definitions

Concolic Testing

- The tester selects a value from the path constraints and negates the values to create a new path value. Then the tester finds concrete constraints to satisfy the new path values.
- The selection of values is responsible by the Constraint Solver which is a part of Concolic tester.
- These constraints are inputs for all next executions. This concolic testing is performed iteratively until exceeds the threshold value or sufficient code coverage is obtained.



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Reads input values

We want to violate this



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();      ← 5
2: y = input();      ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Assign Random Values



Concolic Testing Example

Concolic Testing

Example

```
1: x = input(); ← 5
2: y = input(); ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x		
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();      ← 5
2: y = input();      ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	5	i_0
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();      ← 5
2: y = input();      ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	5	i_0
y	10	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

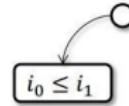
```
1: x = input();      ← 5
2: y = input();      ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	5	i_0
y	10	i_1

Execution Tree



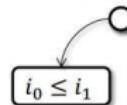
Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Execution Tree



Concolic Testing Example

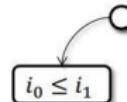
Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```



Execution Tree



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$

Execution Tree

$$i_0 \leq i_1$$



Concolic Testing Example

Concolic Testing

Example

```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$

Execution Tree

$i_0 \leq i_1$



Concolic Testing Example

Concolic Testing

Example

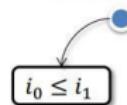
```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x		
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

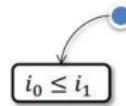
```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

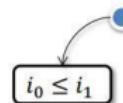
```
1: x = input();      ← 20
2: y = input();      ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

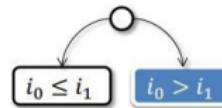
```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

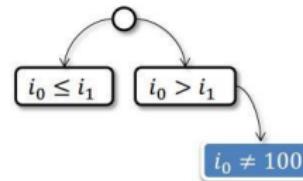
```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree



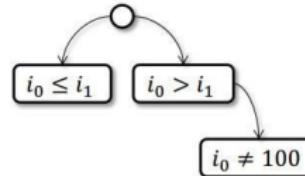
Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Execution Tree



Concolic Testing Example

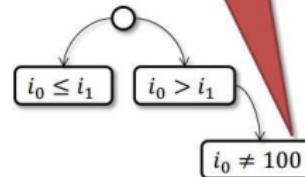
Concolic Testing

Example

```
1: x = input();  
2: y = input();  
3: if (x > y) {  
4:     assert(x != 100);  
5: }
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$
and $i_0 = 100$

Execution Tree



Concolic Testing Example

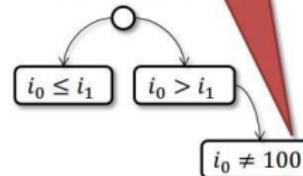
Concolic Testing

Example

```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$
and $i_0 = 100$

Execution Tree



Concolic Testing Example

Concolic Testing

Example

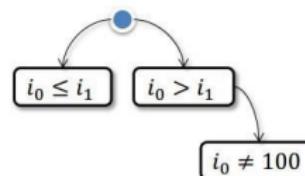
```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x		
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

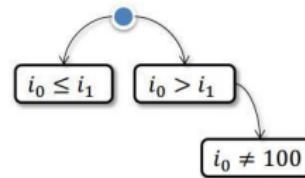
```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

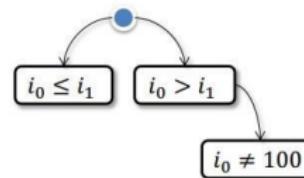
```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

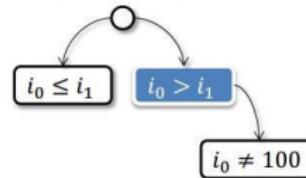
```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

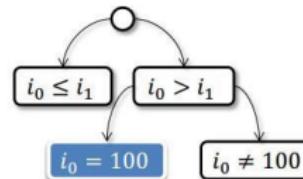


Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree



Definitions

Concolic Testing: Final report

- Concolic testing technique explored total 3 paths for the example program.
- It has generated total three test cases for the variables x and y. These are shown in Table 1.

Table 1: Test cases

Test cases	TC1	TC2	TC3
x	5	20	100
y	10	3	4



Definitions

Distributed Concolic Testing

- *Distributed Concolic Testing (DCT)* is an extension of the original concolic testing approach that uses several computing nodes in a distributed manner.
- It significantly reduces the time to generate test cases with better efficiency.
- Concolic testing is the combination of concrete and symbolic testing. In addition, distributed concolic testing is scalable and achieves a linear speedup by using a large number of computing nodes for test case generation.



Concolic Testers

In Table 1, we have compared concolic testing tools with respect to the following parameters/features: 1) variables types; 2) pointers; 3) native calls; 4) non-linear arithmetic operations; 5) bitwise operations; 6) array offsets and 7) function pointers.

The abbreviation used in Table 1 are the following: “Y

- “means the tool supports the feature. “N”means
- the tool does not support the feature.
- “P”means the tool can partially support the feature.
- “NA”means unknown.



Concolic Testers

Table 2: Summary of concolic tester with their properties.

Tool Name	Supporting Language	Supporting Platform	Support ConstraintsSolver	Support for float/double	Support for pointer	Support for nativecall	Support for nonlinear arithmetic op.	Support for binseep.	Support for offset	Support for functionpointer
DART	C	NA	LPSOLVER	N	N	N	NA	NA	N	N
SMART	C	LINUX	LPSOLVER	N	N	N	NA	NA	N	N
CUTE	C	LINUX	LPSOLVER	N	Y	N	NA	NA	N	N
jCUTE	JAVA	LINUXWINDOWS	NA	N	-	N	NA	NA	N	N
CREST	C	LINUX	YICES	N	N	N	P	P	N	N
EIE	C	LINUX	SIP	N	Y	N	Y	Y	Y	N
KLEE	C	LINUX	SIP	N	Y	P	Y	Y	Y	NA
RASET	C	LINUX	SIP	N	Y	N	Y	Y	Y	NA
FUZZ	JAVA	LINUX	BULTONFF	N	NA	N	N	N	NA	NA
PATHCRAWLER	C	NA	NA	NA	NA	N	NA	NA	NA	NA
PEX	.NET	WINDOWS	Z3	N	NA	N	NA	NA	NA	NA
SAGE	MACHINECODE	WINDOWS	DECOLVER	NA	N	Y	NA	NA	NA	NA
APOLLO	PHP	WINDOWS	CHOCO	NA	NA	N	NA	N	NA	NA
SCORE	C	LINUX	Z3SMT Solver	Y	N	N	Y	N	NA	NA



Comparison of related works

Table 3: Summary of different work on concolic testing.

S.No	Authors	Testing Type	FrameWork Type	Input Type	Output Type
1	Das et al. [16]	Concolic Testing, MC/DC	BCT, CREST, CA	C-Program	MC/DC %
2	Bokil et al. [24]	SC, DC, BC, MC/DC	AutoGen	C-Program	Test data, Time
3	Kim et al. [20]	HCT	SMT Solver, CREST	Flash storage Platform Software	Reduction Ratio
4	Majumdar et al. [17]	HCT, BC	CUTE	Editor in C-Language	Test Cases
5	Burnim et al. [21]	Heuristics Concolic Testing, BC	CREST	Software Application in C	Branch Covered
6	Kim et al. [23]	Concolic Testing	CREST	Embedded C Application	Branch Covered
7	Kim et al. [22]	Concolic Testing	CONBOL	Embedded Software	BC%, Time
8	Kim et al. [19, 25]	Distributed Concolic Testing	SCORE	Embedded C Program	BC%, Effectiveness
9	Sen et al. [26]	Concolic Testing, BC	CUTE, JCUTE	C and Java Programs	Test Cases, BC%, Time



Characteristics of different approaches

Table 4: Characteristics of different approaches on concolic testing.

SNo	Authors	Generated Test Cases	Measuring Coverage%	Determined Time Constraints	Computed Speed
1	Dasetal. [16]	C	C	X	X
2	Bokletal. [24]	C	X	C	X
3	Kmetetal. [20]	C	X	X	X
4	Majumdar etal. [17]	C	X	X	X
5	Burimetal. [21]	C	X	X	X
6	Kmetetal. [23]	C	X	X	X
7	Kmetetal. [22]	C	C	C	X
8	Kmetetal. [19,25]	C	C	X	C
9	Sasetal. [26]	C	C	C	X



References >> I

- 1 Palacios, M., Garc'ia-Fanjul, J., Tuya, J. and Spanoudakis, G., 2015. Coverage-based testing for service level agreements. *IEEE Transactions on Services Computing*, 8(2), pages 299-313.
- 2 Fraser, G. and Arcuri, A., 2014. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2), page 8.
- 3 Jones, JA. and Harrold, MJ. 2013. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*. 29(3), pages 195-209.
- 4 Baluda, M., Braione, P., Denaro, G. and Pezz'e, M., 2011. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Journal*, 19(4), pages 725-751.



References >> II

- 5 Jiang B, Tse TH, Grieskamp W, Kicillof N, Cao Y, Li X, Chan WK. 2011. *Assuring the model evolution of protocol software specifications by regression testing process improvement*. Software: Practice and Experience. 41(10). pages 1073-1103.
- 6 McMinn, P. 2004. Search-based software test data generation: a survey: Research articles. *Software Testing, Verification and Reliability*, 14(2), pages 105–156.
- 7 Harman M., Hu L., Hierons R., Wegener J., Sthamer H., Baresel A., and Roper M., 2004. Testability Transformation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30(1), pages 1-14.
- 8 Wegener, J., Baresel, A. and Sthamer, H., 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14), pages 841-854.
- 9 Duran, J.W. and Ntafos, S.C., 1984. An evaluation of random testing. *IEEE transactions on software engineering*, (4), pages 438-444.



References >> III

- 10 Miller, W., and Spooner, DL. 1976. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223-226,
- 11 Kuhn, DR. 1999. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), pages 411-424.
- 12 Ferguson, R. and Korel, B., 1996. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1), pages. 63-86.
- 13 DeMillo, R., and Offutt, J., 1993. Experimental results from an Automatic Test Case Generation. *ACM Transaction on Software Engineering Methodology*, 2(2), pages 109-175.
- 14 Ntafos, S.C., 1988. A comparison of some structural testing strategies. *IEEE Transactions on software engineering*, 14(6), pages 868-874.
- 15 Chilenski, J., and Miller. S. 1994 Application of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, pages 193-200.



References >> IV

- [16] Das A., and Mall R., 2013. Automatic Generation of MC/DC Test Data. *International Journal of Software Engineering, Acta Press* 2(1).
- [17] Majumder R., and Sen K., 2007. Hybrid Concolic Testing. *Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA* pages 416-426.
- [18] Hayhurst, KJ., Veerhusen DS., Chilenski, JJ., Rierson, LK. 2001. A Practical Tutorial on Modified Condition/Decision Coverage, *NASA/TM-2001-210876*.
- [19] Kim Y., and Kim M., 2011. SCORE: a scalable concolic testing tool for reliable embedded software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* pages 420-423.
- [20] Kim, M., Kim, Y. and Choi, Y., 2012. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing*, 24(3), pages 355-374.



References >> V

- [21] Burnim J., and Sen K., 2008. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443-446, Washington, D.C., USA.
- [22] Kim Y., Kim Y., Kim T., Lee G., Jang Y., and Kim M., 2013. Automated unit testing of large industrial embedded software using concolic testing. *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)* pages 519-528.
- [23] Kim M., and Kim Y.,and Jang Y., 2012. Industrial application of concolic testing on embedded software: Case studies. *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 390-399.
- [24] Bokil, P., Darke, P., Shrotri, U., and Venkatesh, R., 2009. Automatic Test Data Generation for C Programs. In *proceedings 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement*.



References >> VI

- 25 Kim M., Kim Y., and Rothermel G., 2012. A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation. *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 340-349.
- 26 Sen K., and Agha G., 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools (Tools Paper). *DTIC Document*.



Thank You!





Data Flow Testing

Dr. Durga Prasad Mohapatra
Professor
CSE Department
NIT Rourkela

Introduction

- In path coverage, the emphasis was to cover a path using statement or branch coverage.
- However, data and data integrity are as important as code and code integrity of a module.
- We have checked every possibility of the control flow of a module. But what about the data flow in the module?
- These questions can be answered,, if we consider data objects in the control flow of a module.

Introduction

cont...

- Data flow testing is a white-box technique that can be used to detect improper use of data values due to coding errors.
- Errors may be unintentionally introduced in a program by programmers,
 - e.g. a programmer might use a variable without defining it.
- Data flow testing gives a chance to look out for
 - inappropriate data definition, their use in predicate, computations, and termination.

Introduction

cont...

- It identifies the potential bugs
 - by examining the patterns in which that piece of data is used.
- Example: If an out-of-scope data is being used in a computation, then it is a bug. There may be several patterns like this which indicate data anomalies.

Introduction

cont...

- To examine the patterns, the control flow graph of a program is used.
- This test strategy selects the paths in the module's control flow such that various sequences of data objects can be chosen.
- The major focus is on the points at which the data receives values and the places at which the data initialized has been referenced.
- Thus, we have to choose enough paths in the control flow to ensure that every data is initialized before use and all the defined data have been used somewhere.

Data Flow-Based Testing

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

Example

```
1 X(){  
2 int a=5; /* Defines variable a */  
....  
3 While(c>5) {  
4 if (d<50)  
5     b=a*a; /*Uses variable a */  
6     a=a-1; /* Defines variable a */  
....  
7 }  
8 print(a); } /*Uses variable a */
```

Data Flow-Based Testing cont ...

- For a statement numbered S,
 - $\text{DEF}(S) = \{X / \text{statement } S \text{ contains a definition of } X\}$
 - $\text{USES}(S) = \{X / \text{statement } S \text{ contains a use of } X\}$
 - Example: 1: **a=b;** $\text{DEF}(1)=\{a\}$, $\text{USES}(1)=\{b\}$.
 - Example: 2: **a=a+b;** $\text{DEF}(1)=\{a\}$, $\text{USES}(1)=\{a,b\}$.

Data Flow-Based Testing cont ...

- A variable X is said to be **live** at statement S1, if
 - X is defined at a statement S, and
 - there exists a path from S to S1 not containing any definition of X.

DU Chain Example

```
1 X(){  
2     int a=5; /* Defines variable a */  
3     While(c>5) {  
4         if (d<50)  
5             b=a*a; /*Uses variable a */  
6             a=a-1; /* Defines variable a */  
7     }  
8     print(a); } /*Uses variable a */
```

Definition-use chain (DU chain)

- $[X, S, S_1]$,
 - S and S_1 are statement numbers,
 - $X \in \text{DEF}(S)$,
 - $X \in \text{USES}(S_1)$, and
 - the definition of X in the statement S is **live** at statement S_1 .

Data Flow-Based Testing Strategy

- 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.

Example

```
1 X(){  
2   B1; /* Defines variable a */  
3   While(C1) {  
4     if (C2)  
5       if(C4) B4; /*Uses variable a */  
6     else B5;  
7     else if (C3) B2;  
8     else B3;    }  
9   B6 }
```

Example cont ...

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

Data Flow Testing

cont...

- It also closely examines the state of the data in the CFG resulting in a richer test suite
 - than the one obtained from CFG based path testing strategies like statement coverage, branch coverage, etc.

States of a Data Object

- Defined (d):
- Killed / Undefined / Released (k):
- Usage (u):
- Computational use (c-use) or
- Predicate use (p-use).

State of a Data Object cont ...

A data object can be in the following states:

- **Defined (d)** A data object is called defined when it is initialized, i.e., when it is on the left side of an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on.

State of a Data Object

cont...

- ***Kill/Undefined/Released (k)***
- 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.

State of a Data Object cont...

- **Usage (*u*)** When the data object is on the right side of assignment or the control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc.
- In general, we say that the usage is either computational use (c-use) or predicate use (p-use).

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.

Data-Flow Anomalies

cont...

- Example: ‘dk’ means a variable is defined and killed without any use, which is a **potential bug**.
- There are nine possible two-character combinations out of which only four are data anomalies, as shown in next Table.

Table I: Two-character 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 redefine. 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.

- Not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur.
- There may be single-character data anomalies also.
- To represent these types of anomalies, we take the following conventions:
 - $\sim x$: indicates all prior actions are not of interest to x .
 - $x\sim$: indicates all post actions are not listed of interest to x .

Table 2: Single-character data-flow anomalies

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

Some Terminologies

Suppose P is a program that has a graph G (P) and a set of variables V. The graph has a single entry and exit node.

- **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.

Some Terminologies

contd...

- **Usage node** It means the variable has been used in some statement of the program. Node n that belongs to $G(p)$ is usage node of variable v , if the value of variable v is used at the statements corresponding to node n.

Some Terminologies

contd...

- A usage node can be of the following two types:
 - 1) Predicate usage Node: If usage node n is a predicate node, then n is a predicate usage node.
 - 2) 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.

Some Terminologies

contd...

- **Loop-free path segment** It is a path segment for which every node is visited once at most.
- **Simple path segment** It is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.
- **Definition-use path (du-path)** A du-path with respect to a variable v is a path between the definition node and usage node of that variable, Usage node can either be a p-usage or a c-usage node.

Some Terminologies

contd...

- **Definition-clear path (dc-path)** A dc-path with respect to a variable v is a path between the definition node and the usage node such that no other node in the path is a defining node of variable v .
- The du paths which are not dc paths are important, as these are potential spots for testing persons.
- Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths.
- The du-paths which are not dc-paths need more attention.

Static Data Flow Testing

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

EXAMPLE:

Consider a program for calculating the gross salary of an employee in an organization. If his basic salary < 1500, then HRA=10% of the Basic and DA=90% of basic. If his salary >= 1500, then HRA=500 and DA=98% of basic. Calculate the 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 program

Solution

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

Define-use-kill patterns for variable 'bs'

Solution

cont...

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

Define-use-kill patterns for variable ‘gs’

Solution

cont...

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

Define-use-kill patterns for variable '**da**'

Solution

cont...

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

Define-use-kill patterns for variable 'hra'

From the above static data flow testing, only one bug is found, i.e in variable HRA of double definition.

Summary

- Discussed the basic concepts of data flow testing.
- Explained DU Chain.
- Presented the different states of a data object.
- Explained the different data-flow anomalies.
- Explained static data flow testing with an example.

References

1. Rajib Mall, Fundamentals of Software Engineering, (Chapter – 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 5), Second Edition, Oxford University Press, 2016.



Thank you



Data Flow Testing

cont...

Dr. Durga Prasad Mohapatra
Professor
CSE Department
NIT Rourkela

Static Analysis is not Enough

- All anomalies using static analysis cannot be determined and this is an unsolvable problem.
- For example, if the variable is used in an array as the index, we cannot determine its state by static analysis.
- Or it may be the case that the index is generated dynamically during execution, therefore we cannot guarantee what the state of the array element is referenced by that index.

Dynamic Data Flow Testing

- The test cases are designed in such a way that every definition of data variable to each of its use is traced to each of its definition.
- Various strategies are employed for the creation of test cases.
- All these strategies are defined below.

Dynamic Data Flow Testing

cont...

- **All-du Paths (ADUP)** It states that every definition of every variable to every use of that definition should be exercised under some test. It is the strongest data flow testing strategies.

Dynamic Data Flow Testing

cont...

- **All-uses (AU)** This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.

Dynamic Data Flow Testing

cont...

- **All-p-uses/Some-c-uses (APU + C)** This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use. If there are definitions of the variable with no p-use following it, then add computational use (c-use) test cases as required to cover every definition.
- Note: A dc-path (definition-clear path) with respect to a variable v is a path between the definition node & the usage node s.t. that no other node in the path is a defining node of variable v .

Dynamic Data Flow Testing

cont...

- **All-c-uses/Some-p-uses (ACU + P)** This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every computational use. If there are definitions of the variable with no c-use following it, then add predicate use (p-use) test cases as required to cover every definition.

Dynamic Data Flow Testing

cont...

- **All-Predicate-uses (APU)** It is derived from the APU + C strategy and states that for 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.

Dynamic Data Flow Testing

cont...

- **All-Computational-Uses (ACU)** It is derived from the strategy ACU + P strategy and states that for 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.

Dynamic Data Flow Testing

cont...

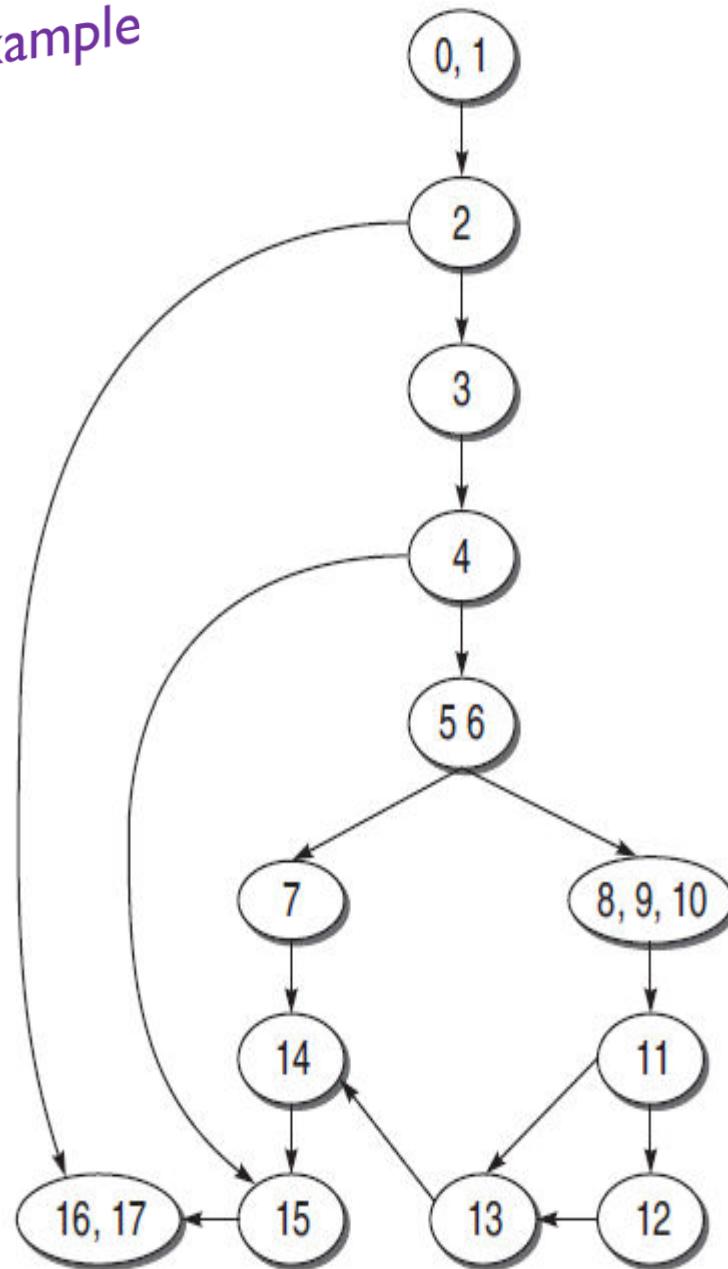
- **All-Definition (AD)** It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

Example

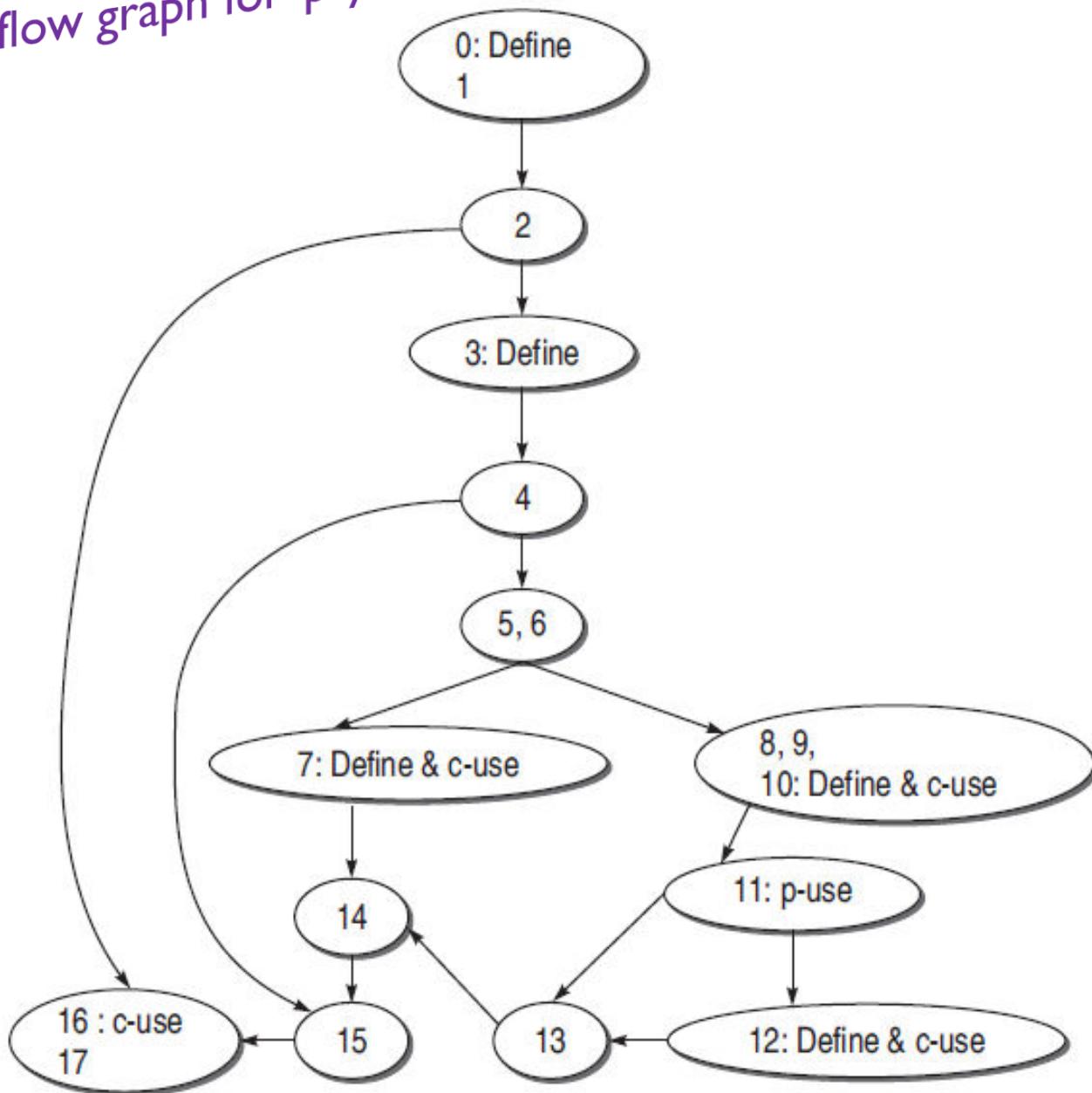
- Consider a program given below. Draw its control flow graph and data flow graph for each variable used in the program, and derive data flow testing paths with all the strategies discussed above.

```
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);
```

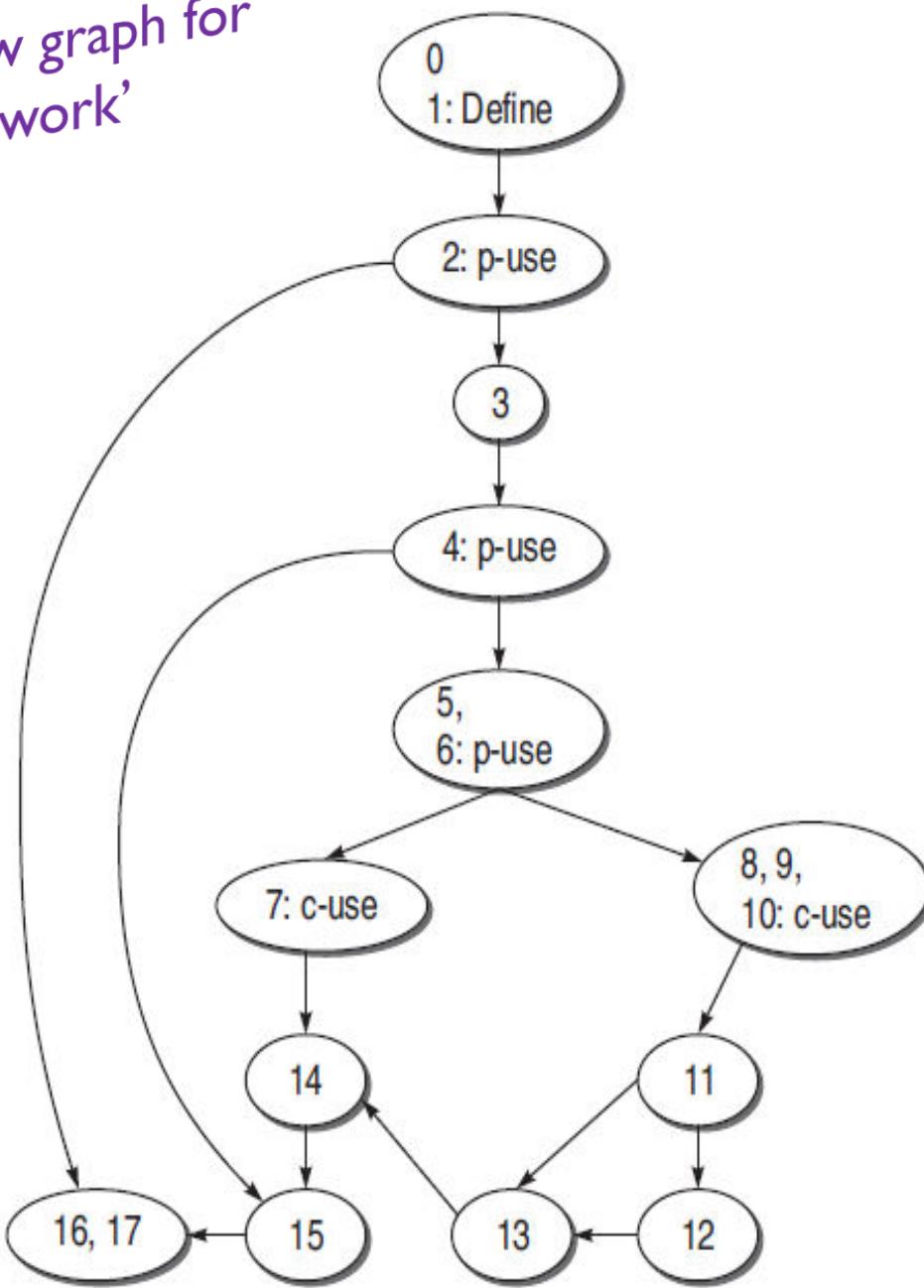
CFG for the Example



Data flow graph for 'payment'



Data flow graph for variable 'work'



- Prepare a list of all the definition nodes and usage nodes for all the variables in the program.

Variable	Defined At	Used At
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

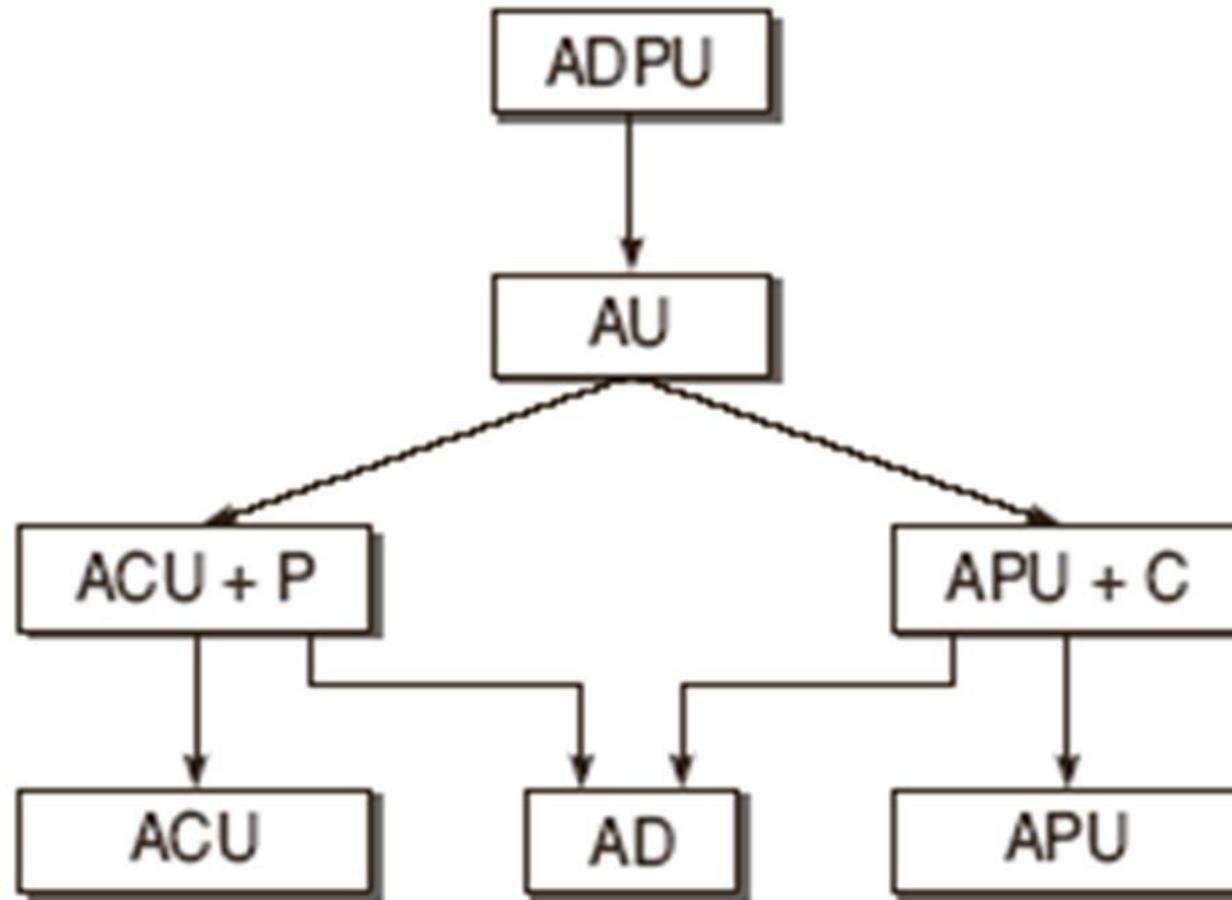
Strategy	Payment	Work
All Uses(AU)	3-4-5-6-7 10-11 10-11-12 12-13-14-15-16 3-4-5-6-8-9-10	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All p-uses(APU)	0-1-2-3-4-5-6-8-9-10-11	1-2 1-2-3-4 1-2-3-4-5-6
All c-uses(ACU)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10

Strategy	Payment	Work
All-p-uses / Some-c-uses (APU + C)	0-1-2-3-4-5-6-8-9-10-11	1-2
	10-11-12	1-2-3-4
	12-13-14-15-16	1-2-3-4-5-6
		1-2-3-4-5-6-8-9-10
All-c-uses / Some-p-uses (ACU + P)	0-1-2-16	1-2-3-4-5-6-7
	3-4-5-6-7	1-2-3-4-5-6-8-9-10
	3-4-5-6-8-9-10	1-2-3-4-5-6
	3-4-15-16	
	7-14-15-16	
	10-11-12	
	10-11-13-14-15-16	
	12-13-14-15-16	
	0-1-2-3-4-5-6-8-9-10-11	

Strategy	Payment	Work
All-du-paths (ADUP)	0-1-2-3-4-5-6-8-9-10-11	1-2
	0-1-2-16	1-2-3-4
	3-4-5-6-7	1-2-3-4-5-6
	3-4-5-6-8-9-10	1-2-3-4-5-6-7
	3-4-15-16	1-2-3-4-5-6-8-9-10
	7-14-15-16	
	10-11-12	
	10-11-13-14-15-16	
	12-13-14-15-16	
All Definitions (AD)	0-1-2-16	1-2
	3-4-5-6-7	
	7-14-15-16	
	10-11	
	12-13-14-15-16	

Ordering of data flow testing strategies

- While selecting a test case, we need to analyse the relative strengths of various data flow testing strategies.



Ordering of data flow testing strategies

cont...

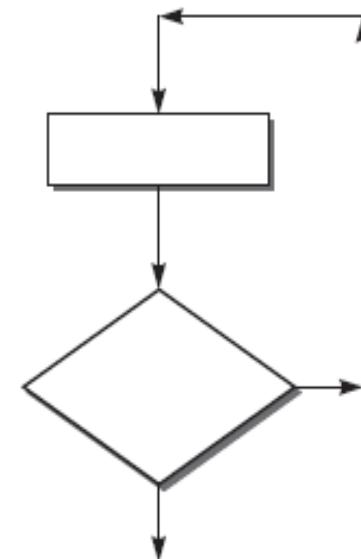
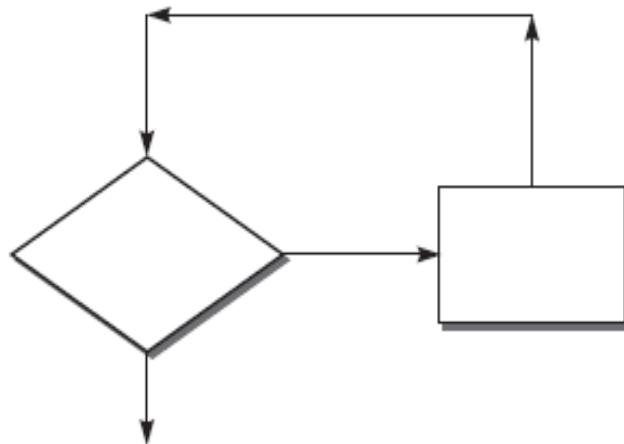
- The previous Figure depicts the relative strength of the data flow strategies.
- In this figure, the relative strength of testing strategies reduces along the direction of the arrow.
- It means that all-du-paths (ADUP) is the strongest criterion for selecting the test cases.

Loop testing

- Loop testing can be viewed as an extension to branch coverage.
- Loops are important in the software from the testing view point. If loops are not tested properly, bugs can go undetected.
- Loop testing can be done effectively while performing development testing (unit testing by the developer) on a module.
- Sufficient test cases should be designed to test every loop thoroughly.
- There are four different kinds of loops. How each kind of loop is tested, is discussed below.

Simple loops

- Simple loops mean, we have a single loop in the flow, as shown below.



Testing Simple Loops

- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.

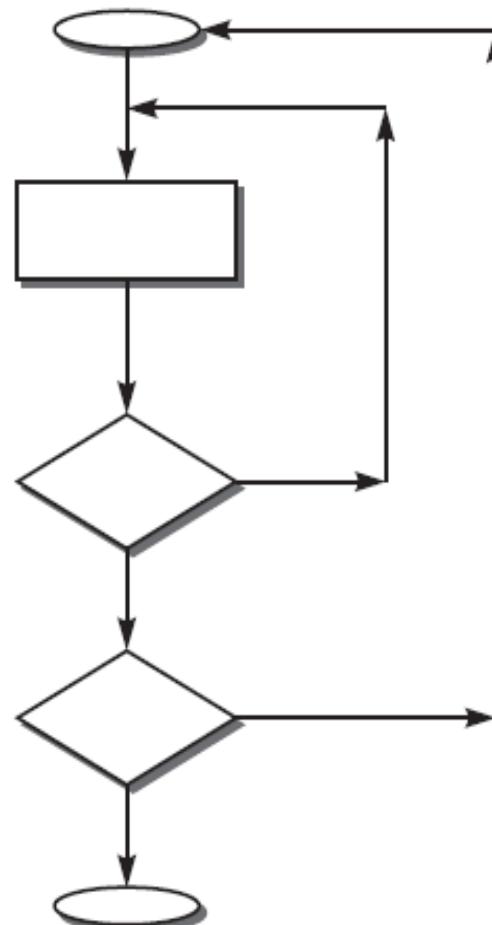
Testing Simple Loops cont ...

- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say max and min) in the loop. It means we should test for min, min+1, min-1, max-1, max, and max+1 number of iterations through the loop.

Nested loops

- When two or more loops are embedded, it is called a nested loop, as shown in next slide. If we have nested loops in the program, it becomes difficult to test.

Nested loops



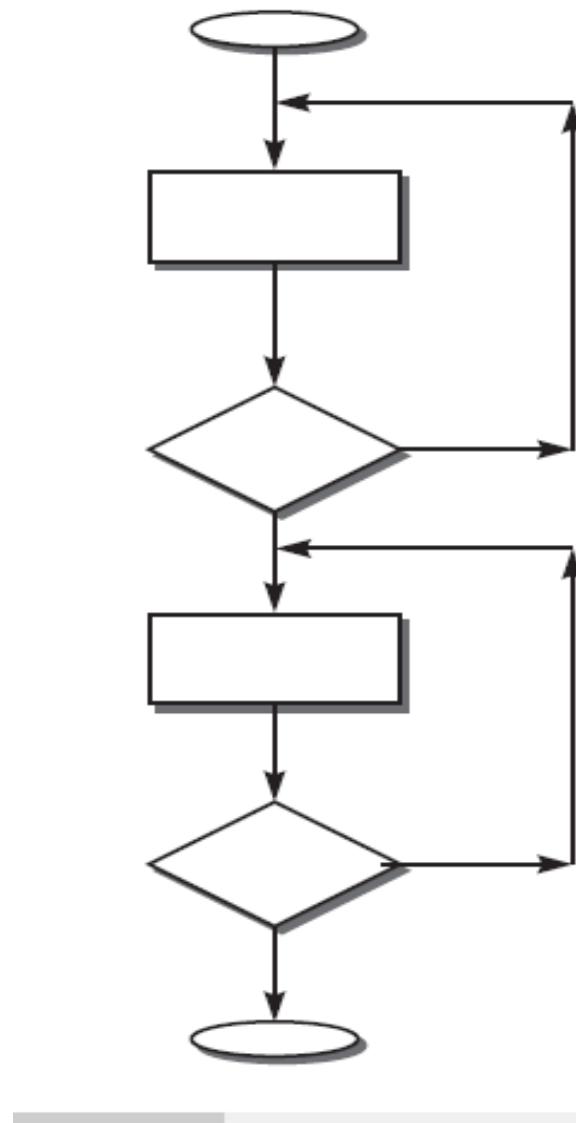
Testing Nested Loops

- If we adopt the approach of simple tests to test the nested loops, then the number of possible test cases grows 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

Concatenated loops

- The loops in a program may be concatenated (shown in next slide).
- Two loops are concatenated if it is possible to reach one, after exiting the other, while still on a path from entry to exit.
- If the two loops are not on the same path, then they are not concatenated.
- The two loops on the same path may or may not be independent.
- If the loop control variable for one loop is used for another loop, then they are concatenated, but nested loops should be treated like nested only.

Concatenated loops



Testing Concatenated Loops

- The concatenated loop may be treated as a sequence of two or more numbers of simple loops.
- So, the strategy for testing of simple loops may be extended to testing of concatenated loops.

Unstructured loops

- This type of loops is really impractical to test.
- They must be redesigned or at least converted into simple or concatenated loops.

Summary

- Explained dynamic data flow testing strategies with an example.
- Presented the ordering of different data flow testing strategies.
- Discussed different loop testing strategies.

References

- I. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 5), Second Edition, Oxford University Press, 2016.



Thank you

ERROR GUESSING

- Error guessing is the preferred method used when all other methods fail.
- Sometimes error guessing is used to test some special cases.
- Error guessing is a very practical case wherein the tester uses his intuition, experience, knowledge of project, bug history and makes a guess about where the bug can be.

Basic idea is to make a list of possible errors in error prone situations and then develop the test cases.

No general procedure for this tech exists. It is an adhoc process.

Example - Special cases in the

- roots of a quadratic equation
 - Though, we do consider this case, there are chances that we overlook it while testing, as it has two cases:
 - i) If $a=0$ then the equation is no longer quadratic.
 - ii) For calculation of roots, the division is by zero. So, we may overlook.

- What will happen when all the inputs are negative?
- What will happen when the inputs list is empty?

Testing and Debugging

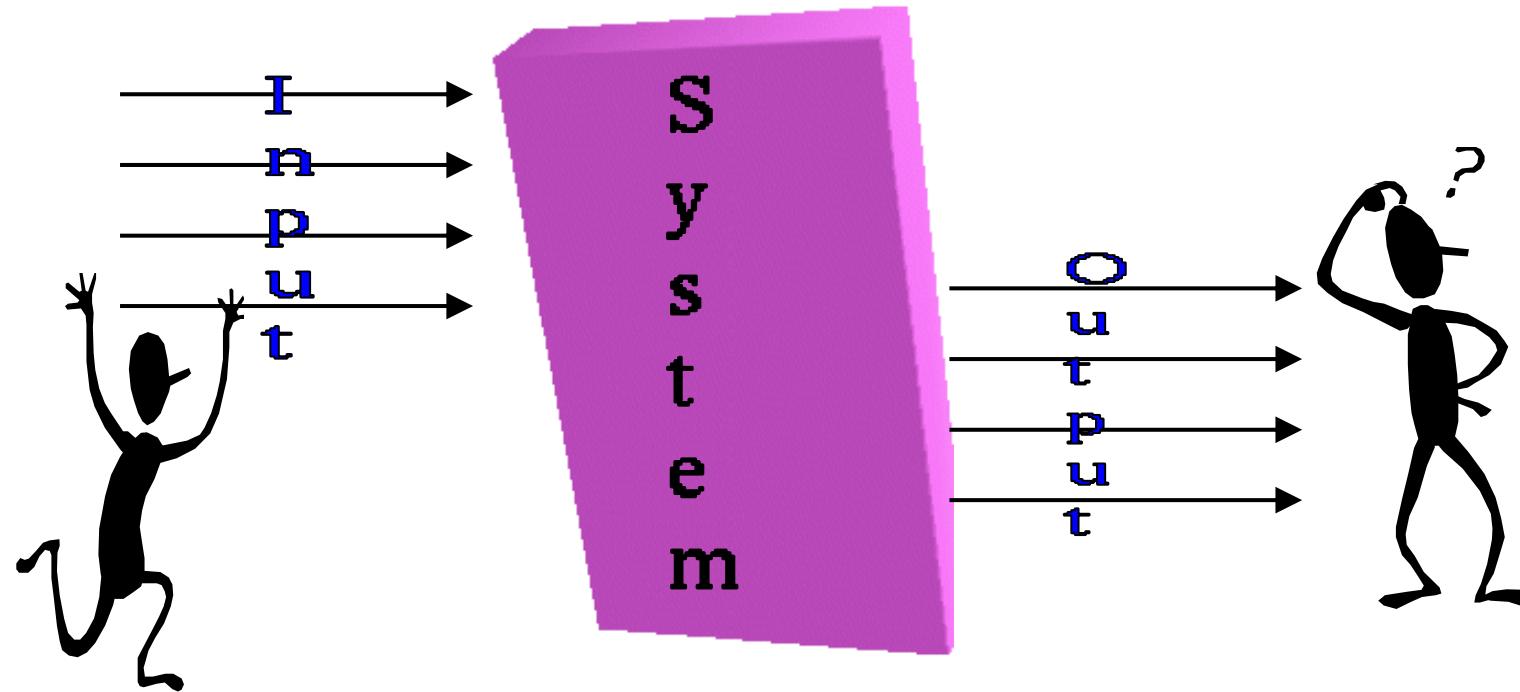
(Lecture 11)

Prof. R. Mall
Dept. of CSE, IIT, Kharagpur

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.

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 (aka defect or bug).
 - Mere presence of an error may not lead to a failure.

Error, Faults, and Failures

- . A fault is an incorrect state entered during program execution:
 - A variable value is different from what it should be.
 - A fault may or may not lead to a failure.

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 two main approaches to design test cases:
 - Black-box approach
 - White-box (or glass-box) approach

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.

Black-Box Testing

- . There are essentially two main approaches to design black box test cases:
 - Equivalence class partitioning
 - Boundary value analysis

Equivalence Class Partitioning

- Input values to a program are partitioned into equivalence classes.
- Partitioning is done such that:
 - Program behaves in similar ways to every input value belonging to an equivalence class.

Why Define Equivalence Classes?

- Test the code with just one representative value from each equivalence class:
 - As good as testing using any other values from the equivalence classes.

Equivalence Class Partitioning

- How do you determine the equivalence classes?
 - Examine the input data.
 - Few general guidelines for determining the equivalence classes can be given

Equivalence Class Partitioning

- If the input data to the program is specified by a **range of values**:
 - e.g. numbers between 1 to 5000.
 - One valid and two invalid equivalence classes are defined.

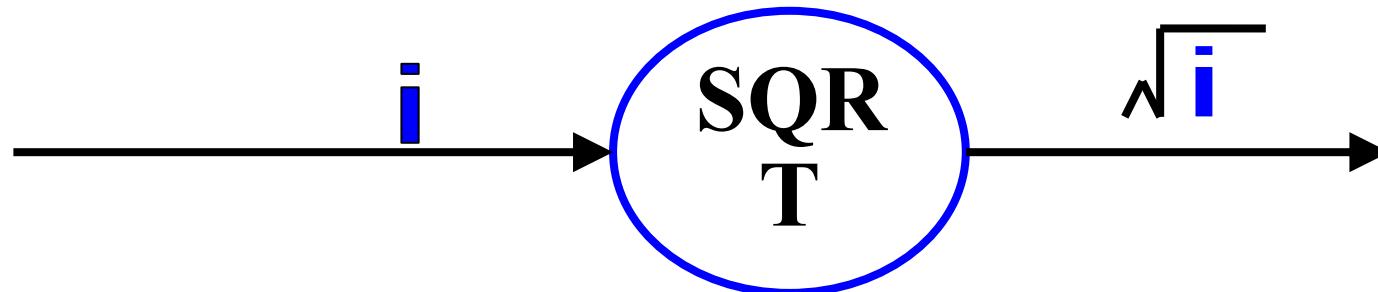


Equivalence Class Partitioning

- . If input is an enumerated set of values:
 - e.g. {a,b,c}
 - One equivalence class for valid input values.
 - Another equivalence class for invalid input values should be defined.

Example

- . A program reads an input value in the range of 1 and 5000:
 - Computes the square root of the input number



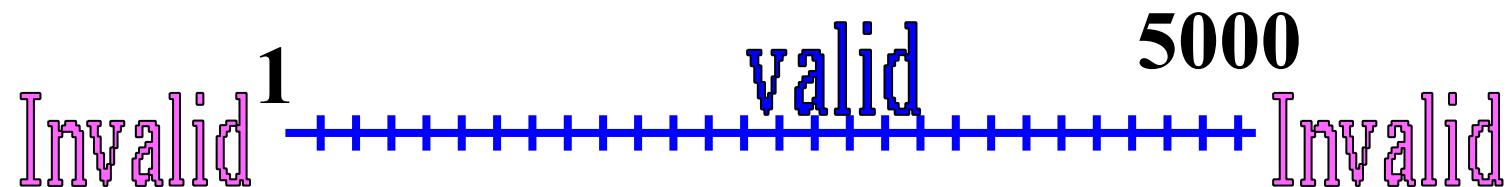
Example (cont.)

- There are three equivalence classes:
 - The set of negative integers,
 - Set of integers in the range of 1 and 5000,
 - Integers larger than 5000.



Example (cont.)

- . The test suite must include:
 - Representatives from each of the three equivalence classes:
 - A possible test suite can be:
 $\{-5, 500, 6000\}$.



Boundary Value Analysis

- Some typical programming errors occur:
 - At boundaries of equivalence classes
 - Might be purely due to psychological factors.
- Programmers often fail to see:
 - Special processing required at the boundaries of equivalence classes.

Boundary Value Analysis

- Programmers may improperly use < instead of <=
- Boundary value analysis:
 - Select test cases at the boundaries of different equivalence classes.

Example

- . For a function that computes the square root of an integer in the range of 1 and 5000:
 - Test cases must include the values: {0,1,5000,5001}.



Debugging

- Once errors are identified:
 - It is necessary identify the precise location of the errors and to fix them.
- Each debugging approach has its own advantages and disadvantages:
 - Each is useful in appropriate circumstances.

Brute-Force method

- . This is the most common method of debugging:
 - Least efficient method.
 - Program is loaded with print statements
 - Print the intermediate values
 - Hope that some of printed values will help identify the error.

Symbolic Debugger

- . Brute force approach becomes more systematic:
 - With the use of a symbolic debugger,
 - Symbolic debuggers get their name for historical reasons
 - Early debuggers let you only see values from a program dump:
 - . Determine which variable it corresponds to.

Symbolic Debugger

- Using a symbolic debugger:
 - Values of different variables can be easily checked and modified
 - Single stepping to execute one instruction at a time
 - Break points and watch points can be set to test the values of variables.

Backtracking

- This is a fairly common approach.
- Beginning at the statement where an error symptom has been observed:
 - Source code is traced backwards until the error is discovered.

Example

```
int main(){  
    int i,j,s;  
    i=1;  
    while(i<=10){  
        s=s+i;  
        i++; j=j++;}  
    printf("%d",s);  
}
```

Backtracking

- Unfortunately, as the number of source lines to be traced back increases,
 - the number of potential backward paths increases
 - becomes unmanageably large for complex programs.

Cause-elimination method

- Determine a list of causes:
 - which could possibly have contributed to the error symptom.
 - tests are conducted to eliminate each.
- A related technique of identifying error by examining error symptoms:
 - software fault tree analysis.

Program Slicing

- This technique is similar to back tracking.
- However, **the search space is reduced by defining slices.**
- A slice is defined for a particular variable at a particular statement:
 - set of source lines preceding this statement which can influence the value of the variable.

Example

```
int main(){  
    int i,s;  
    i=1; s=1;  
    while(i<=10){  
        s=s+i;  
        i++;}  
    printf("%d",s);  
    printf("%d",i);  
}
```

Debugging Guidelines

- Debugging usually requires a thorough understanding of the program design.
- Debugging may sometimes require full redesign of the system.
- A common mistake novice programmers often make:
 - not fixing the error but the error symptoms.

Debugging Guidelines

- . Be aware of the possibility:
 - an error correction may introduce new errors.
- . After every round of error-fixing:
 - regression testing must be carried out.

Program Analysis Tools

- An automated tool:
 - takes program source code as input
 - produces reports regarding several important characteristics of the program,
 - such as size, complexity, adequacy of commenting, adherence to programming standards, etc.

Program Analysis Tools

- Some program analysis tools:
 - Produce reports regarding the adequacy of the test cases.
- There are essentially two categories of program analysis tools:
 - **Static analysis tools**
 - **Dynamic analysis tools**

Static Analysis Tools

- . Static analysis tools:
 - Assess properties of a program without executing it.
 - Analyze the source code
 - . Provide analytical conclusions.

Static Analysis Tools

- . Whether coding standards have been adhered to?
 - Commenting is adequate?
- . Programming errors such as:
 - uninitialized variables
 - mismatch between actual and formal parameters.
 - Variables declared but never used, etc.

Static Analysis Tools

- . Code walk through and inspection can also be considered as static analysis methods:
 - However, the term static program analysis is generally used for automated analysis tools.

Dynamic Analysis Tools

- Dynamic program analysis tools require the program to be executed:
 - its behavior recorded.
 - Produce reports such as adequacy of test cases.

Testing

- The aim of testing is to identify all defects in a software product.
- However, in practice even after thorough testing:
 - one cannot guarantee that the software is error-free.

Testing

- . The input data domain of most software products is very large:
 - It is not practical to test the software exhaustively with each input data value.

Testing

- . Testing does however expose many errors:
 - Testing provides a practical way of reducing defects in a system
 - Increases the users' confidence in a developed system.

Testing

- Testing is an important development phase:
 - requires the maximum effort among all development phases.
- In a typical development organization:
 - maximum number of software engineers can be found to be engaged in testing activities.

Testing

- Many engineers have the wrong impression:
 - testing is a secondary activity
 - it is intellectually not as stimulating as the other development activities, etc.

Testing

- . Testing a software product is in fact:
 - as much challenging as initial development activities such as specification, design, and coding.
- . Also, testing involves a lot of creative thinking.

Testing

- Software products are tested at three levels:
 - Unit testing
 - Integration testing
 - System testing

Unit testing

- During unit testing, modules are tested in isolation:
 - If all modules were to be tested together:
 - . it may not be easy to determine which module has the error.

Unit testing

- Unit testing reduces debugging effort several folds.
 - Programmers carry out unit testing immediately after they complete the coding of a module.

Integration testing

- . After different modules of a system have been coded and unit tested:
 - modules are integrated in steps according to an integration plan
 - partially integrated system is tested at each integration step.

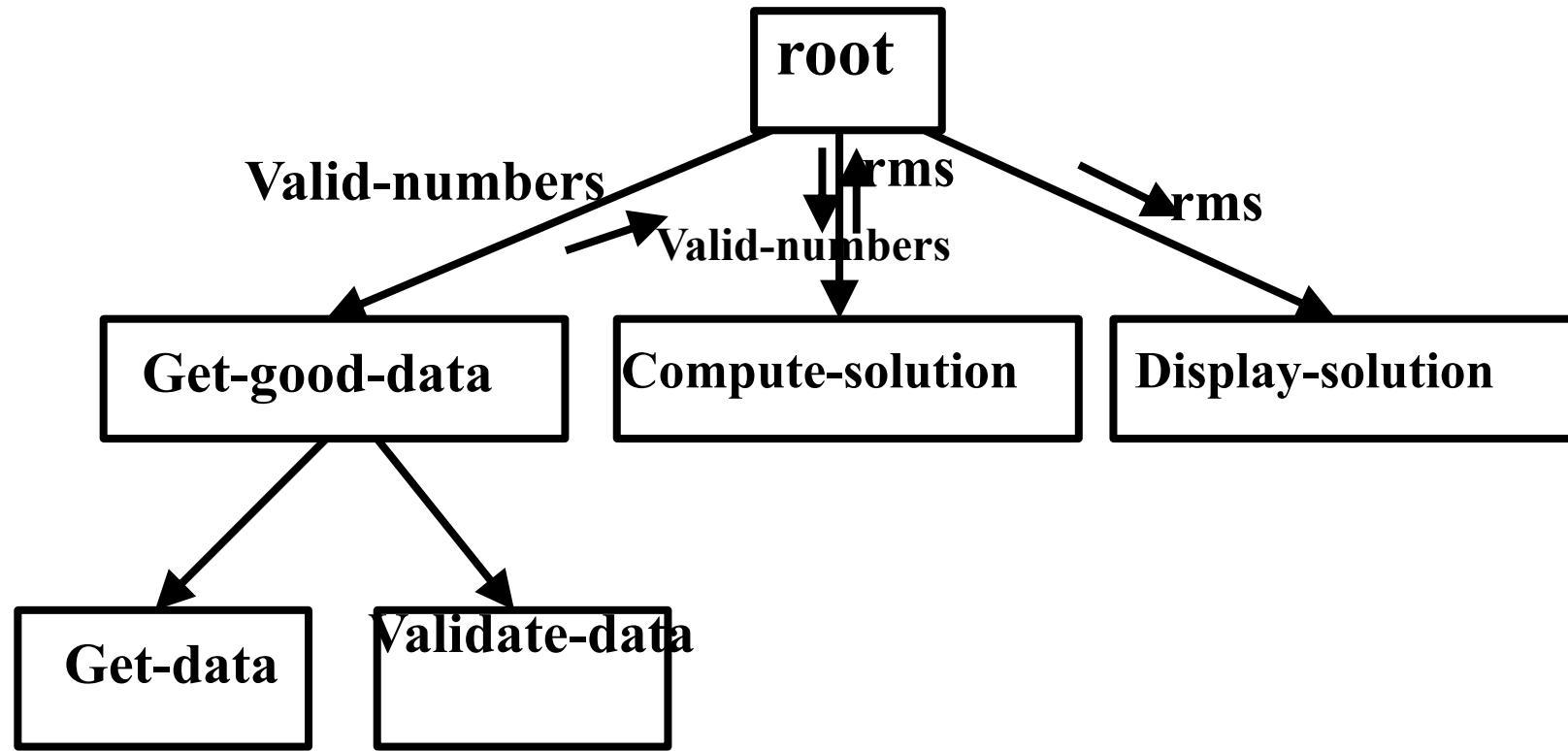
System Testing

- . System testing involves:
 - validating a fully developed system against its requirements.

Integration Testing

- . Develop the integration plan by examining the structure chart :
 - big bang approach
 - top-down approach
 - bottom-up approach
 - mixed approach

Example Structured Design



Big Bang Integration Testing

- . Big bang approach is the simplest integration testing approach:
 - all the modules are simply put together and tested.
 - this technique is used only for very small systems.

Big Bang Integration Testing

- Main problems with this approach:
 - If an error is found:
 - It is very difficult to localize the error
 - The error may potentially belong to any of the modules being integrated.
 - Debugging errors found during big bang integration testing are very expensive to fix.

Bottom-up Integration Testing

- Integrate and test the bottom level modules first.
- A disadvantage of bottom-up testing:
 - when the system is made up of a large number of small subsystems.
 - This extreme case corresponds to the big bang approach.

Top-down integration testing

- Top-down integration testing starts with the main routine:
 - and one or two subordinate routines in the system.
- After the top-level 'skeleton' has been tested:
 - immediate subordinate modules of the 'skeleton' are combined with it and tested.

Mixed Integration Testing

- Mixed (or sandwiched) integration testing:
 - uses both top-down and bottom-up testing approaches.
 - Most common approach

Integration Testing

- . In top-down approach:
 - testing waits till all top-level modules are coded and unit tested.
- . In bottom-up approach:
 - testing can start only after bottom level modules are ready.

System Testing

- . There are three main kinds of system testing:
 - Alpha Testing
 - Beta Testing
 - Acceptance Testing

Alpha Testing

- . System testing is carried out by the test team within the developing organization.

Beta Testing

- System testing performed by a select group of friendly customers.

Acceptance Testing

- . System testing performed by the customer himself:
 - to determine whether the system should be accepted or rejected.

Stress Testing

- . Stress testing (aka endurance testing):
 - impose abnormal input to stress the capabilities of the software.
 - Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

How Many Errors are Still Remaining?

- . Seed the code with some known errors:
 - artificial errors are introduced into the program.
 - Check how many of the seeded errors are detected during testing.

Error Seeding

- Let:
 - N be the total number of errors in the system
 - n of these errors be found by testing.
 - S be the total number of seeded errors,
 - s of the seeded errors be found during testing.

Error Seeding

- $n/N = s/S$
- $N = S n/s$
- remaining defects:
$$N - n = n ((S - s)/ s)$$

Example

- . 100 errors were introduced.
- . 90 of these errors were found during testing
- . 50 other errors were also found.
- . Remaining errors= $50 \ (100-90)/90 = 6$

Error Seeding

- The kind of seeded errors should match closely with existing errors:
 - However, it is difficult to predict the types of errors that exist.
- Categories of remaining errors:
 - can be estimated by analyzing historical data from similar projects.

Summary

- Exhaustive testing of almost any non-trivial system is impractical.
 - we need to design an optimal test suite that would expose as many errors as possible.

Summary

- . If we select test cases randomly:
 - many of the test cases may not add to the significance of the test suite.
- . There are two approaches to testing:
 - black-box testing
 - white-box testing.

Summary

- Black box testing is also known as **functional testing**.
- Designing black box test cases:
 - Requires understanding only SRS document
 - Does not require any knowledge about design and code.
- Designing white box testing requires knowledge about design and code.

Summary

- . We discussed black-box test case design strategies:
 - Equivalence partitioning
 - Boundary value analysis
- . We discussed some important issues in integration and system testing.

Mutation Testing

Dr. Durga Prasad Mohapatra
Dept. of Computer Science & Engineering
NIT Rourkela

Introduction

- 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 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 the test data.

Introduction cont...

- Mutation testing helps a user create test data by interacting with user to iteratively strengthen quality of test data.
- During mutation testing, faults are introduced into a program.
- 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.

Introduction cont...

- A mutant is said to be killed when a test case causes it to fail.
- When this happens, the mutant is considered dead and no longer needs to remain in the testing process, since the faults represented by that mutant have been detected.

Introduction cont...

- The software is first tested:
 - using an initial testing method based on white-box strategies.
- After the initial testing is complete,
 - mutation testing is taken up.

Main Idea

- The idea behind mutation testing:
 - make a few arbitrary small changes to a program at a time.

Main Idea cont ...

- Insert faults into a program:
 - Check whether the test suite is able to detect these.
 - This either validates or invalidates the test suite.

Main Idea cont ...

- Each time the program is changed,
 - it is called a mutated program
 - the change is called a mutant.

Main Idea cont ...

- 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.

Main Idea cont ...

- 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.

Primitive changes

- The primitive changes can be:
 - Deleting a statement
 - Altering an arithmetic operator,
 - Changing the value of a constant,
 - Changing a data type, etc.

Traditional Mutation Operators

- Deletion of a statement
- Boolean:
 - Replacement of a statement with another
 - eg. == and >=, < and <=
 - Replacement of boolean expressions with true or false
 - eg. **a || b** with **true**
 - Replacement of arithmetic
 - eg. * and +, / and -
 - Replacement of a variable (ensuring same scope/type)

Underlying Hypotheses

- Mutation testing is based on the following two hypotheses:
 - **The Competent Programmer Hypothesis**
 - **The Coupling Effect**

Both of these were proposed by DeMillo et al., 1978

The Competent Programmer Hypothesis

- Programmers create programs that are close to being correct:
 - Differ from the correct program by some simple errors.

The Coupling Effect

- Complex errors are caused due to several simple errors.
- It therefore suffices to check for the presence of the simple errors

Types of mutants

- Primary Mutant
- Secondary Mutant

Primary Mutant

When the mutants are single modification of the initial program using some operators, they are called **primary mutants**.

Example

```
if(a>b)
    x=x + y;
else
    x = y;
Printf("%d", x);
.....
```

We can consider following mutant for this example;

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

Example cont ...

The results of the initial program and its mutant

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)

Secondary Mutants

- When multiple levels of mutation are applied on the initial program, then, this class of mutant is called secondary Mutant.
- In this case, it is very difficult to identify the initial program form its mutants.

Example

If($a < b$)

$c = a;$

Mutants for this code may be as follows:

M1 : if($a \leq b - 1$)

$c = a;$

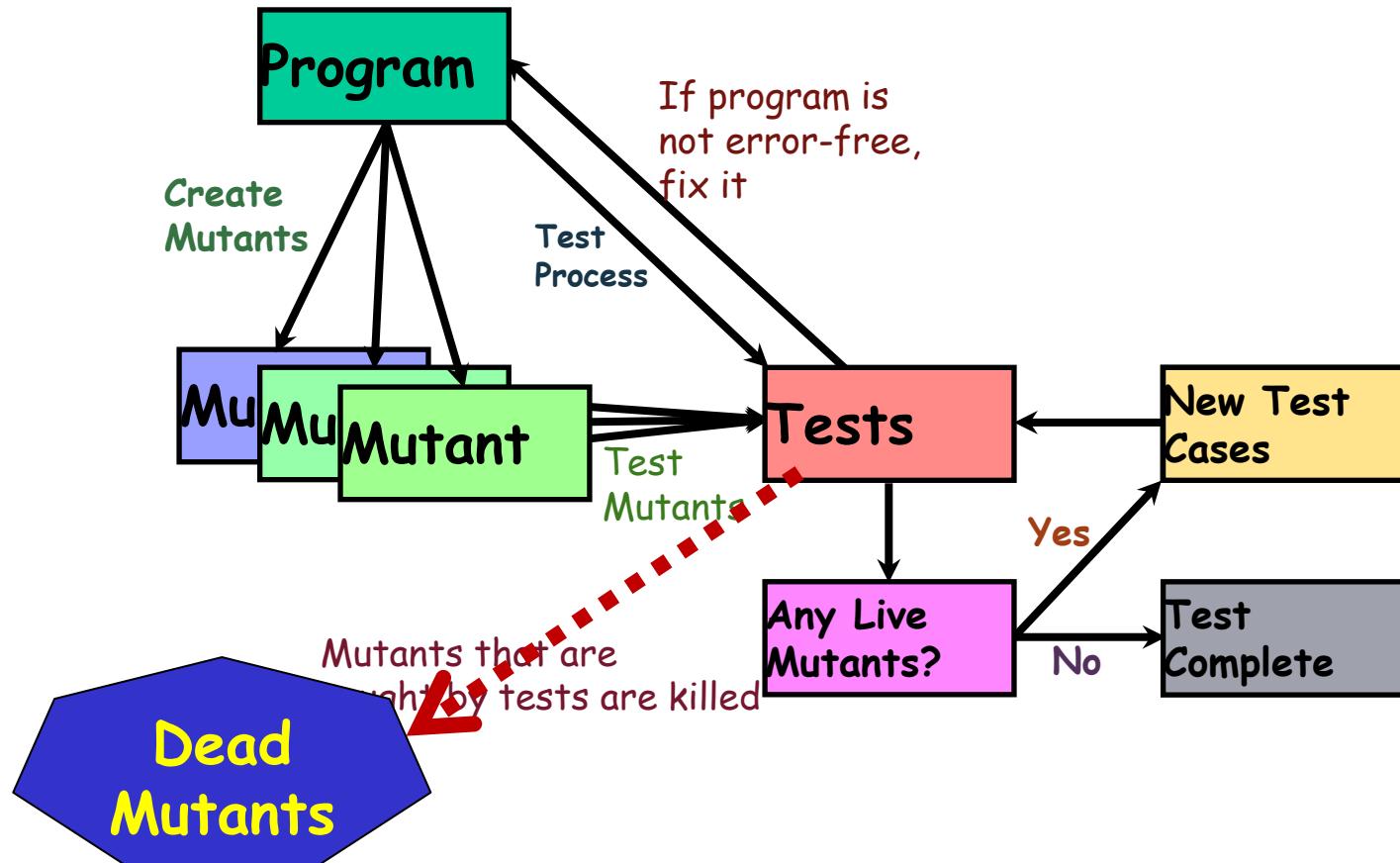
M2 : if($a + 1 \leq b$)

$c = a;$

M3 : if($a == b$)

$c = a + 1;$

Mutation Testing Process



Mutation Testing Process cont...

- Construct the mutants of a test program.
- Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
- If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
- If the output is correct, that test case is executed against each live mutant.
- If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.

Mutation Testing Process cont...

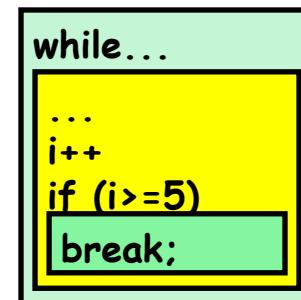
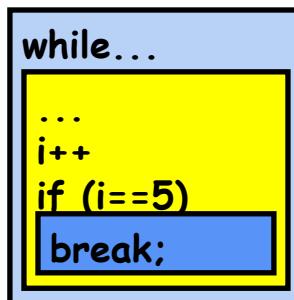
- After each test case has been executed against each live mutant, each remaining mutant falls into one of the following two categories:
 - ✓ One, the mutant is functionally equivalent to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it.
 - ✓ Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.

Mutation Testing Process cont...

- The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data.
- If the mutation score is 100%, then the test data is called mutation adequate.

Equivalent Mutants

- There may be surviving mutants that **cannot be killed**,
 - These are called **Equivalent Mutants**
- Although syntactically different:
 - These mutants are **indistinguishable** through testing.
- Therefore have to be checked 'by hand'



Disadvantages of Mutation Testing

- Equivalent mutants
- Computationally very expensive.
 - A large number of possible mutants can be generated.
- Certain types of faults are very difficult to inject.
 - Only simple syntactic faults introduced

Quiz 1

- Identify one advantage and one disadvantage of the mutation test technique.

Quiz 1: Solution

- Identify two advantages and two disadvantages of the mutation test technique.
- **Adv:**
 - Can be automated
 - Helps effectively strengthen black box and coverage-based test suite
- **Disadv:**
 - Equivalent mutants

Thank You

Positive and Negative Testing

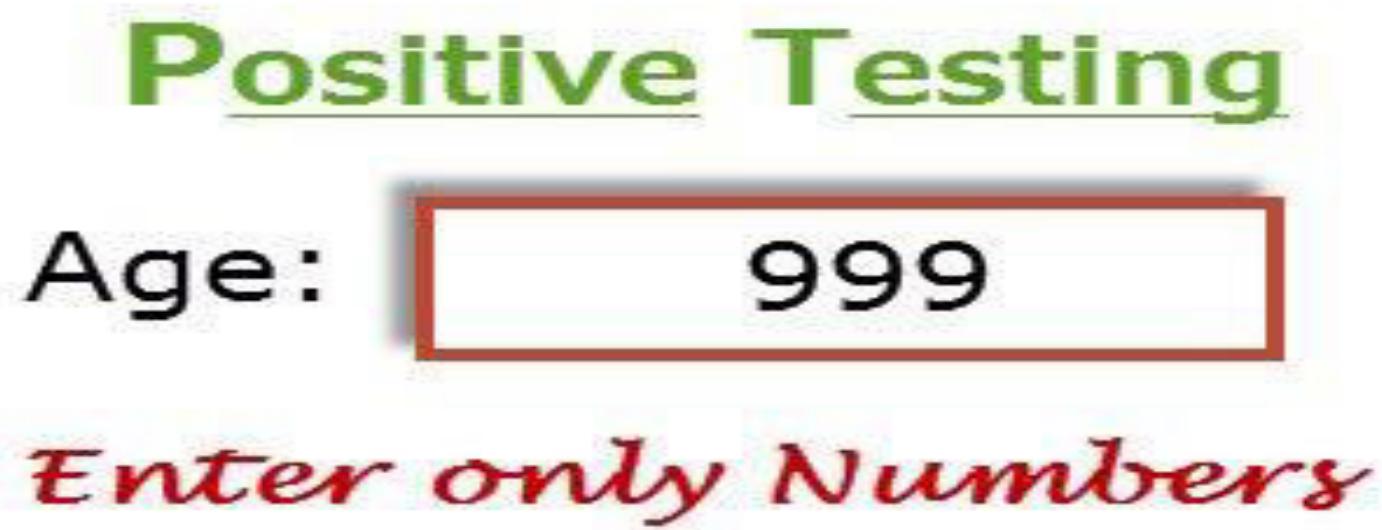
DR. D. P. MOHAPATRA
ASSOCIATE PROFESSOR
DEPARTMENT OF CSE
NIT, ROURKELA

Positive Testing

- Positive testing tries to prove that a given product does what it is supposed to do.
- Positive Testing is testing process where the system is validated against the valid input data.
- In this testing tester always check for only valid set of values and check if an application behaves as expected with its expected inputs.
- The main intention of this testing is to check whether software application not showing error when not supposed to and showing error when supposed to.
- Such testing is to be carried out keeping positive point of view and only execute the positive scenario.
- When a test case verifies the requirements of the product with a set of expected output, it is called *positive test case*.
- The purpose of positive testing is to prove that the product works as per specification and expectations.
- A product delivering an error when it is expected to give an error, is also a part of positive testing.
- Positive testing can thus be said to check the product's behaviour for positive and negative conditions as stated in the requirement.

Example of Positive testing

- Consider a scenario where you want to test an application which contains a simple textbox to enter age and requirements say that it should take only integer values.
- So here provide only positive integer values to check whether it is working as expected or not is the Positive Testing.
- Most of the applications developers implement Positive scenarios where testers get less defects count around positive testing.



An example

- Let us take a lock and key.
- We do not know how the levers in the lock work, but we only know the set of inputs (the number of keys, specific sequence of using the keys and the direction of turn of each key) and the expected outcome (locking and unlocking).
- For example, if a key is turned clockwise it should unlock and if turned anticlockwise it should lock.
- To use the lock one need not understand how they work.
- However, it is essential to know the external functionality of the lock and key system.
- Some of the functionality that you need to know to use the lock are given as follows.

Some functionalities to use the lock

Functionality	What you need to know to use
Features of a lock	It is made of metal, has a hole provision to lock, has a facility to insert the key and the keyhole ability to turn clockwise or anticlockwise.
Features of key	It is made of metal and created to fit into a particular lock's keyhole.
Actions performed	Key inserted and turned clockwise to unlock. Key inserted and turned anticlockwise to lock.
States	Locked. Unlocked.
Inputs	Key turned clockwise or anticlockwise.
Expected outcome	Locking. Unlocking.

Sample requirements specification for lock and key system

Sl. No.	Requirements identifier	Description	Priority (High, med, low)
1	BR-01	Inserting the key numbered 123-456 and turning it clockwise should facilitate locking	H
2	BR-02	Inserting the key numbered 123-456 and turning it anticlockwise should facilitate unlocking	H
3	BR-03	Only key number 123-456 can be used to lock and unlock	H
4	BR-04	No other object can be used to lock	M
5	BR-05	No other object can be used to unlock	M
6	BR-06	The lock must not open even when it is hit with a heavy object	M
7	BR-07	The lock and key must be made of metal and must weigh approximately 150 grams	L
8	BR-08	Lock and unlock directions should be changeable for usability of left-handers	L

Example of positive test cases

Req. No.	Input 1	Input 2	Current state	Expected output
BR-01	Key 123-456	Turn clockwise	Unlocked	Locked
BR-01	Key 123-456	Turn clockwise	Locked	No change
BR-02	Key 123-456	Turn anticlockwise	Unlocked	No change
BR-02	Key 123-456	Turn anticlockwise	Locked	Unlock
BR-04	Hairpin	Turn clockwise	Locked	No change

Example of positive test condition for positive testing

- Take the first row in the table.
- When the lock is in an unlocked state and you use key 123-456 and turn it clockwise, the expected outcome is to get it locked.
- During test execution, if the test results in locking, then the test is passed.

Example of negative test condition for positive testing

- In the fifth row of the table, the lock is in locked state.
- Using a hairpin and turning it clockwise should not cause a change in state or cause any damage to the lock.
- On test execution, if there are no changes, then this positive test case is passed.

Negative Testing

- Negative testing is done to show that the product does not fail when an unexpected input is given.
- The purpose of negative testing is to try and break the system.
- Negative testing covers scenarios for which the product is not designed and coded.
- In other words, the input values may not have been represented in the specification of the product.
- These test conditions can be termed as unknown conditions for the product as far as the specifications are concerned.
- But, at the end-user level, there are multiple scenarios that are encountered and that need to be taken care of by the product.
- It becomes even more important for the tester to know the negative situations that may occur at the end-user level so that the application can be tested and made foolproof.
- A negative test would be a product *not delivering an error when it should or delivering an error when it should not*.

Negative Testing

- Negative Testing is testing process where the system is validated against the invalid input data.
- A negative test checks if an application behaves as expected with its negative inputs.
- The main intention of this testing is to check whether software application not showing error when supposed to and showing error when not supposed to.
- Such testing is to be carried out keeping negative point of view and only execute the test cases for only invalid set of input data.
- Negative testing is a testing process to identify the inputs where system is not designed or un-handled inputs by providing different invalid inputs.
- The main reason behind Negative testing is to check the stability of the software application against the influences of different variety of incorrect validation data set.
- The Negative testing helps to improve the testing coverage of your software application under test.
- Both positive and negative testing approaches are equally important for making your application more reliable and stable.

Example of Negative Testing

- Consider a textbox example which should accept only integer values.
- So here provide the characters like “abcd” in the age textbox and check the behavior of application, either it should show a validation error message for all invalid inputs (for all other than integer values) or system should not allow to enter a non integer values.

Negative Testing

Age:

abcd

Enter only Numbers

Example of negative test cases

S. No.	Input 1	Input 2	Current state	Expected output
1	Some other lock's key	Turn clockwise	Lock	Lock
2	Some other lock's key	Turn anticlockwise	Unlock	Unlock
3	Thin piece of wire	Turn anticlockwise	Unlock	Unlock
4	Hit with a stone		Lock	Lock

- In the above table, there are no requirement numbers.
- This is because negative testing focuses on test conditions that lie outside the specification.
- Since all the test conditions are outside the specification, they cannot be categorized as positive and negative test conditions.
- Some people consider all of them as negative test conditions, which is technically correct.

Positive and negative testing scenarios

- If the requirement is saying that password text field should accept 6 – 20 characters and only alphanumeric characters.
- **Positive Test Scenarios**
 - Password textbox should accept 6 characters
 - Password textbox should up to 20 characters
 - Password textbox should accept any value in between 6-20 chars length.
 - Password textbox should accept all numeric and alphabets values.
- **Negative Test scenarios**
 - Password textbox should not accept less than 6 characters
 - Password textbox should not exceed more than 20 characters
 - Password textbox should not accept special characters

Positive and Negative Test Scenarios with Example

- You are doing the testing on login form which have following fields like Username field, Password field, and Sign In, Sign Up, Cancel, Login Button etc.
- Now positive scenario of login form is that you enter the valid username and password in the username and password field, and then click on Login Button to check whether the user is able to login or not.
- Negative scenario of login form is that you leave the password field blank and fill the username field, and then click on Login Button to check whether the user is able to login or not.

Consideration for both the testing

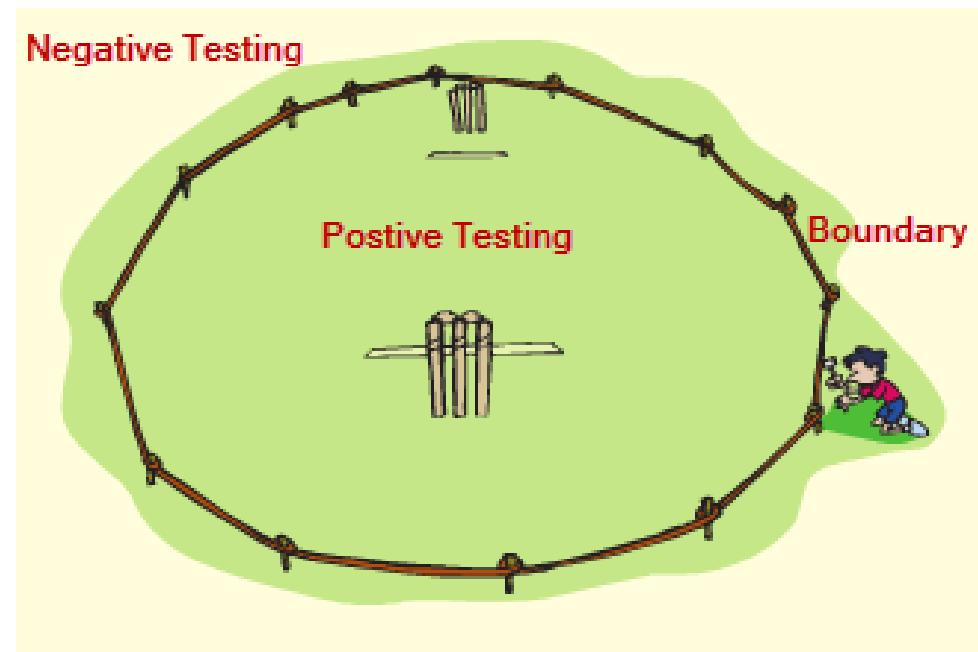
- In both the testing, following needs to be considered:
 - Input data
 - Action which needs to be performed
 - Output Result

Testing Technique used for Positive and Negative Testing

- Following techniques are used for Positive and negative validation of testing is:
 - Boundary Value Analysis
 - Equivalence Partitioning

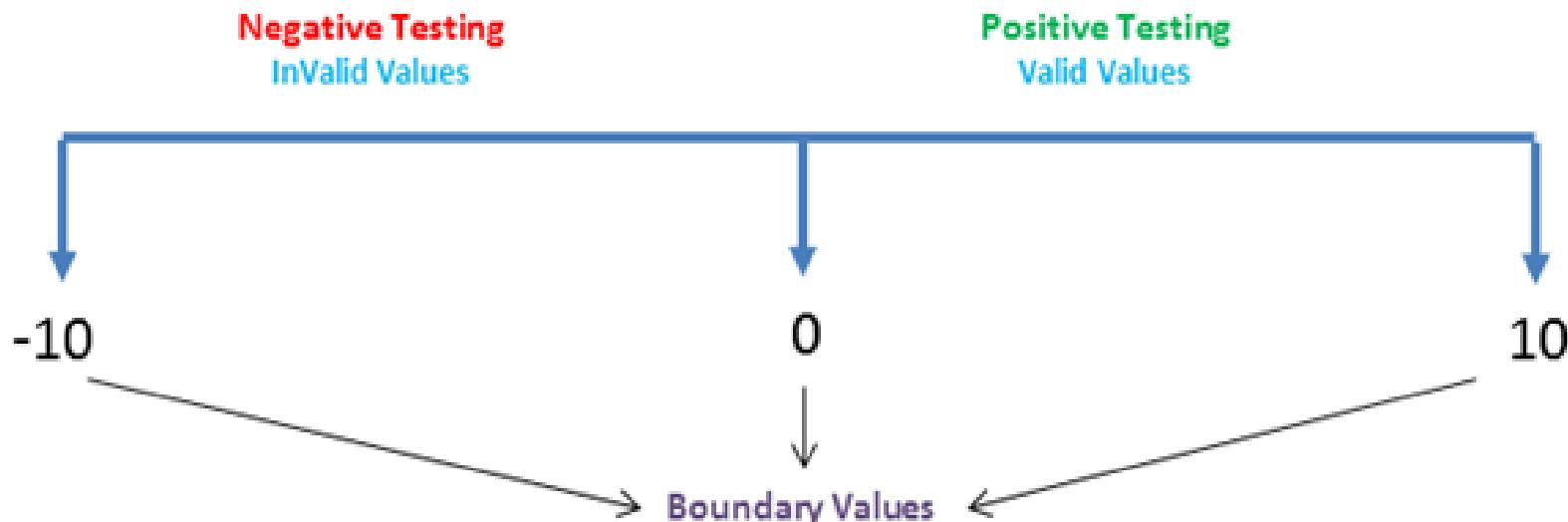
Boundary Value Analysis

- This is one of the software testing technique in which the test cases are designed to include values at the boundary.
- If the input data is used within the boundary value limits, then it is said to be Positive Testing.
- If the input data is picked outside the boundary value limits, then it is said to be Negative Testing.



Example

- A system can accept the numbers from 0 to 10 numeric values.
- All other numbers are invalid values.
- Under this technique , boundary values 0 , 10 and -10 will be tested.

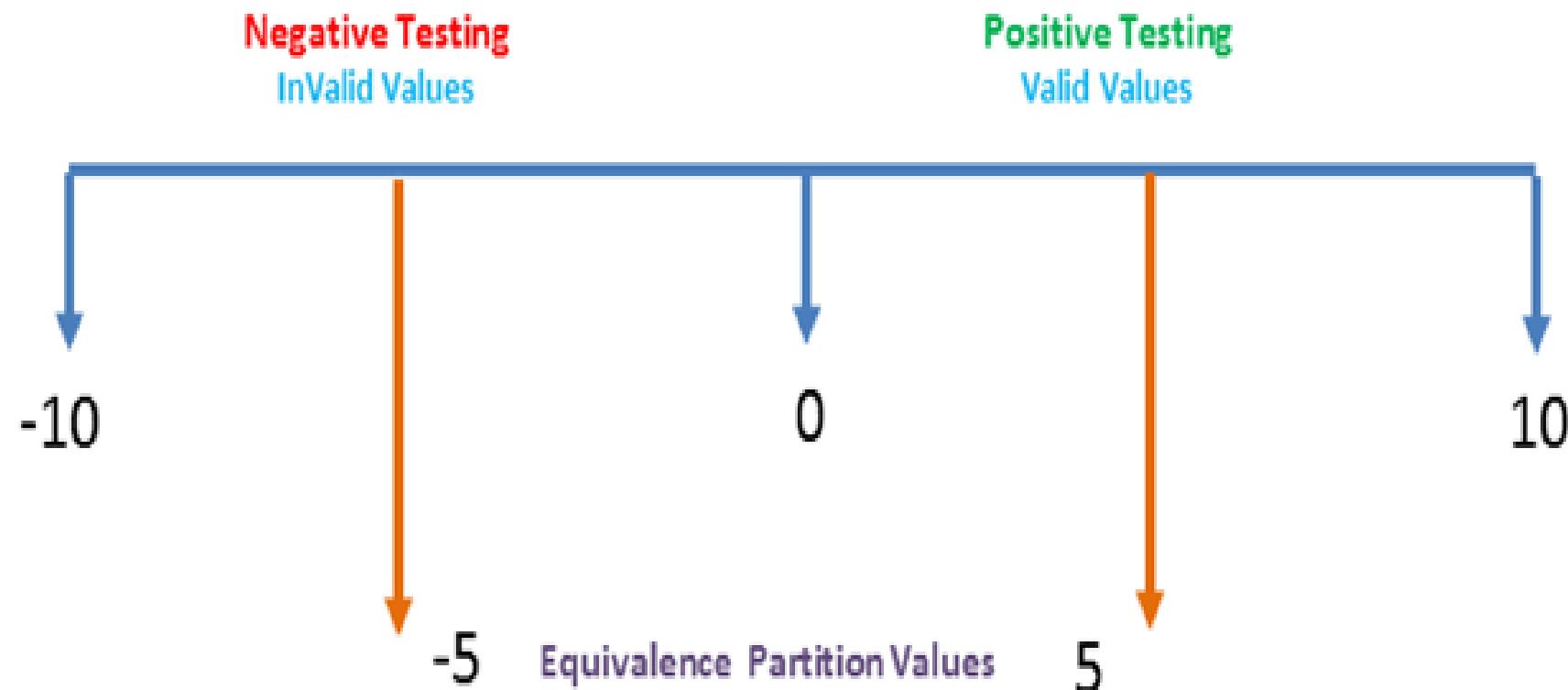


Equivalence Partitioning

- This is a software testing technique which divides the input data into many partitions.
- Values from each partition must be tested at least once.
- Partitions with valid values are used for Positive Testing.
- While partitions with invalid values are used for negative testing.

Example

- Numeric values Zero to ten can be divided to two(or three)partition.
- In our case we have two partitions -10 to -1 and 0 to 10.
- Sample values (5 and -5) can be taken from each part to test the scenarios.



Difference between positive and negative testing

- For positive testing if all documented requirements and test conditions are covered, then coverage can be considered to be 100 percent.
- If the specifications are very clear, then coverage can be achieved.
- In contrast, there is no end to negative testing and 100 percent coverage in negative testing is impractical.
- Negative testing requires a high degree of creativity among the testers to cover as many “unknowns” as possible to avoid failure at a customer site.



Top Distinction between Positive and Negative Testing

Positive Testing (Valid)	Negative Testing (Invalid)
1. Positive Testing means testing the application or system by giving valid data.	1. Negative Testing means testing the application or system by giving invalid data.
2. In this testing tester always check for only valid set of values.	2. In this testing tester always check for only invalid set of values.
3. Positive Testing is done by keeping positive point of view for example checking the mobile number field by giving numbers only like 9999999999.	3. Negative Testing is done by keeping negative point of view for example checking the mobile number field by giving numbers and alphabets like 99999abcde.
4. It is always done to verify the known set of Test Conditions .	4. It is always done to break the project and product with unknown set of Test Conditions.
5. This Testing checks how the product and project behave by providing valid set of data.	5. This Testing covers those scenarios for which the product is not designed and coded by providing invalid set of data.
6. Main aim means purpose of this Testing is to prove that the project and product works as per the requirements and specifications.	6. Main aim means purpose of this Testing is try to break the application or system by providing invalid set of data .
7. This type of Testing always tries to prove that a given product and project always meets the requirements and specifications of a client and customer.	7. Negative Testing is that in which tester attempts to prove that the given product and project does, which is not said in the client and customer requirements.

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?

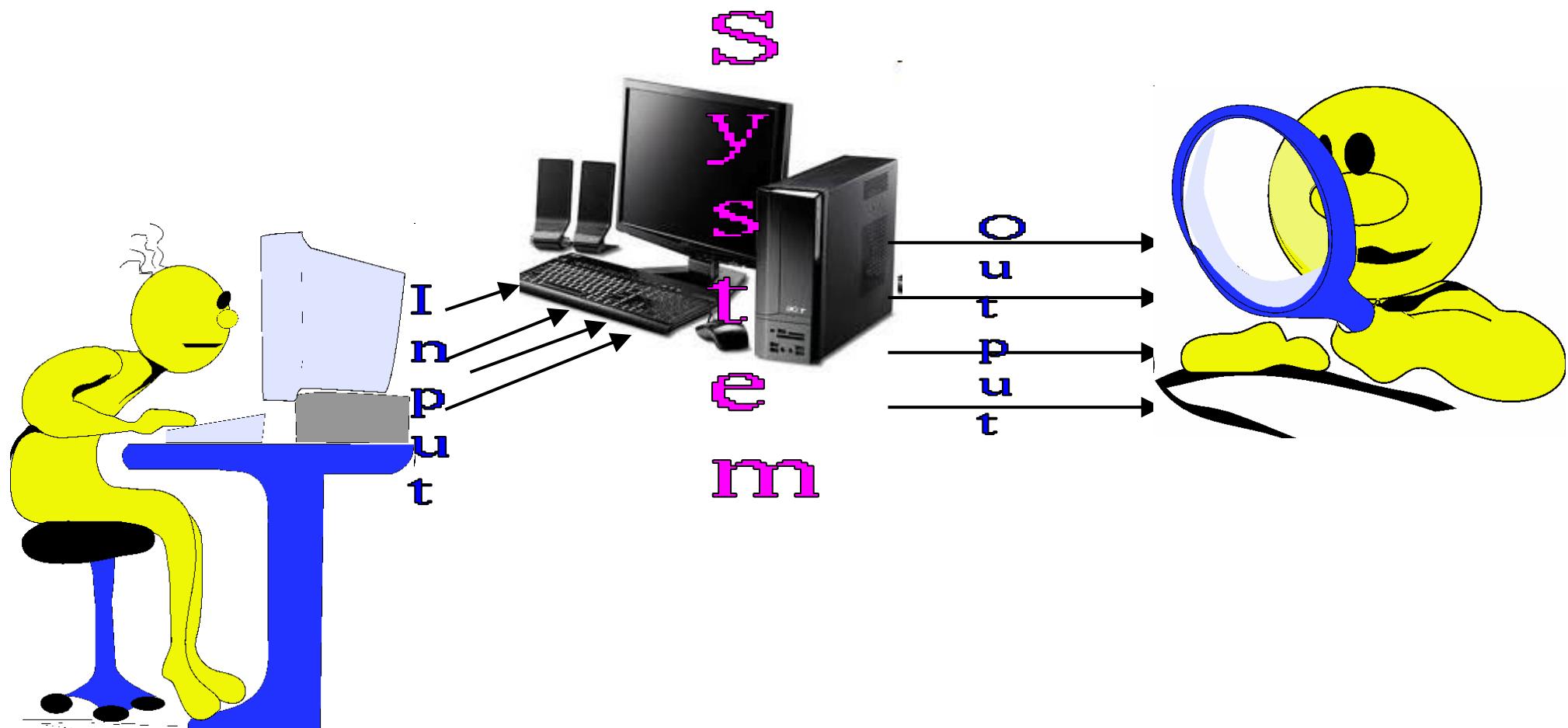


- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
 - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
 - The floating point number was larger than 32767
 - Efficiency considerations had led to the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

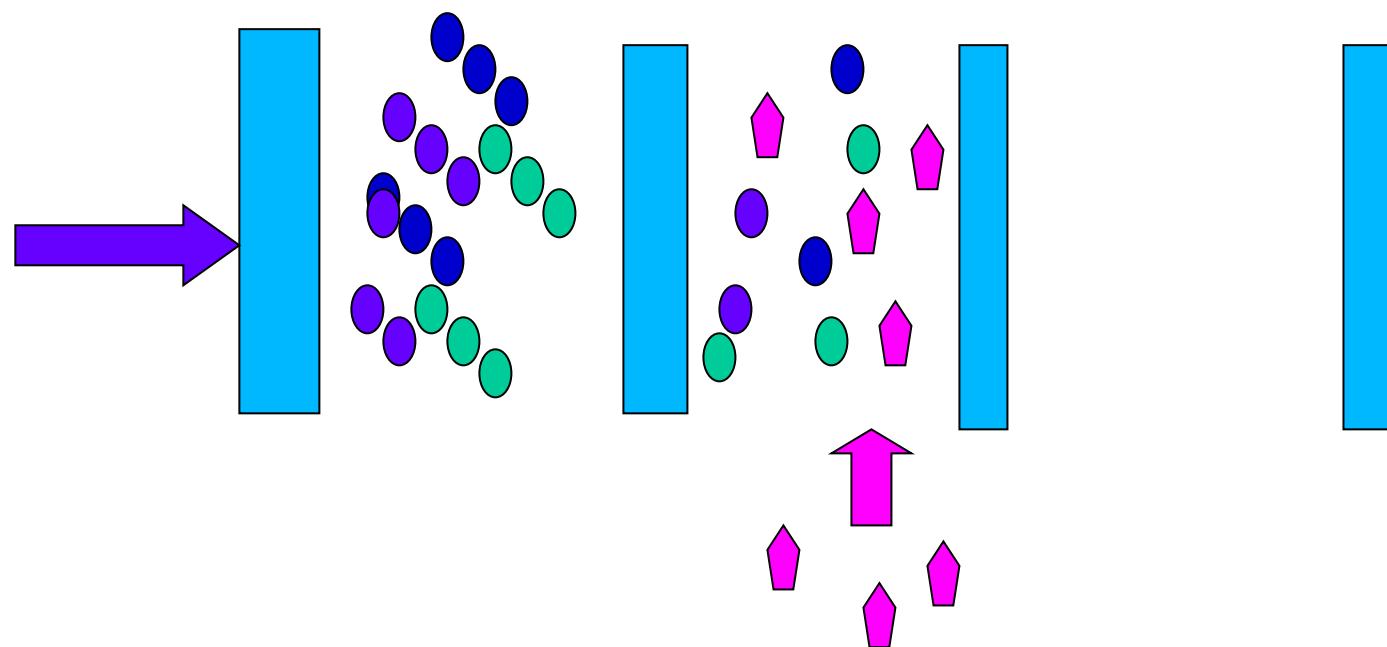
- Test Suite Design
- Run test cases and observe results to detect failures.
- Debug to locate errors
- Correct errors.

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.



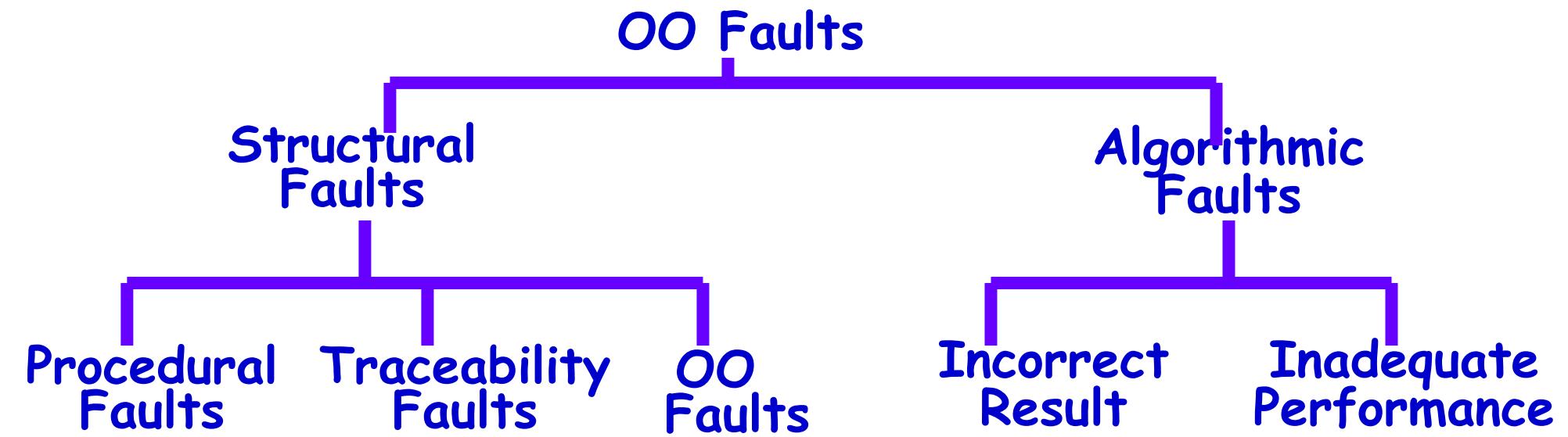
Pesticide Effect

- 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?

## Black-box

- 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.

## White-box

- 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){
- 1 while (x != y){
- 2   if (x>y) then                      **Euclid's GCD Algorithm**
- 3       x=x-y;
- 4   else y=y-x;
- 5 }
- 6 return x;                      }

# 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.

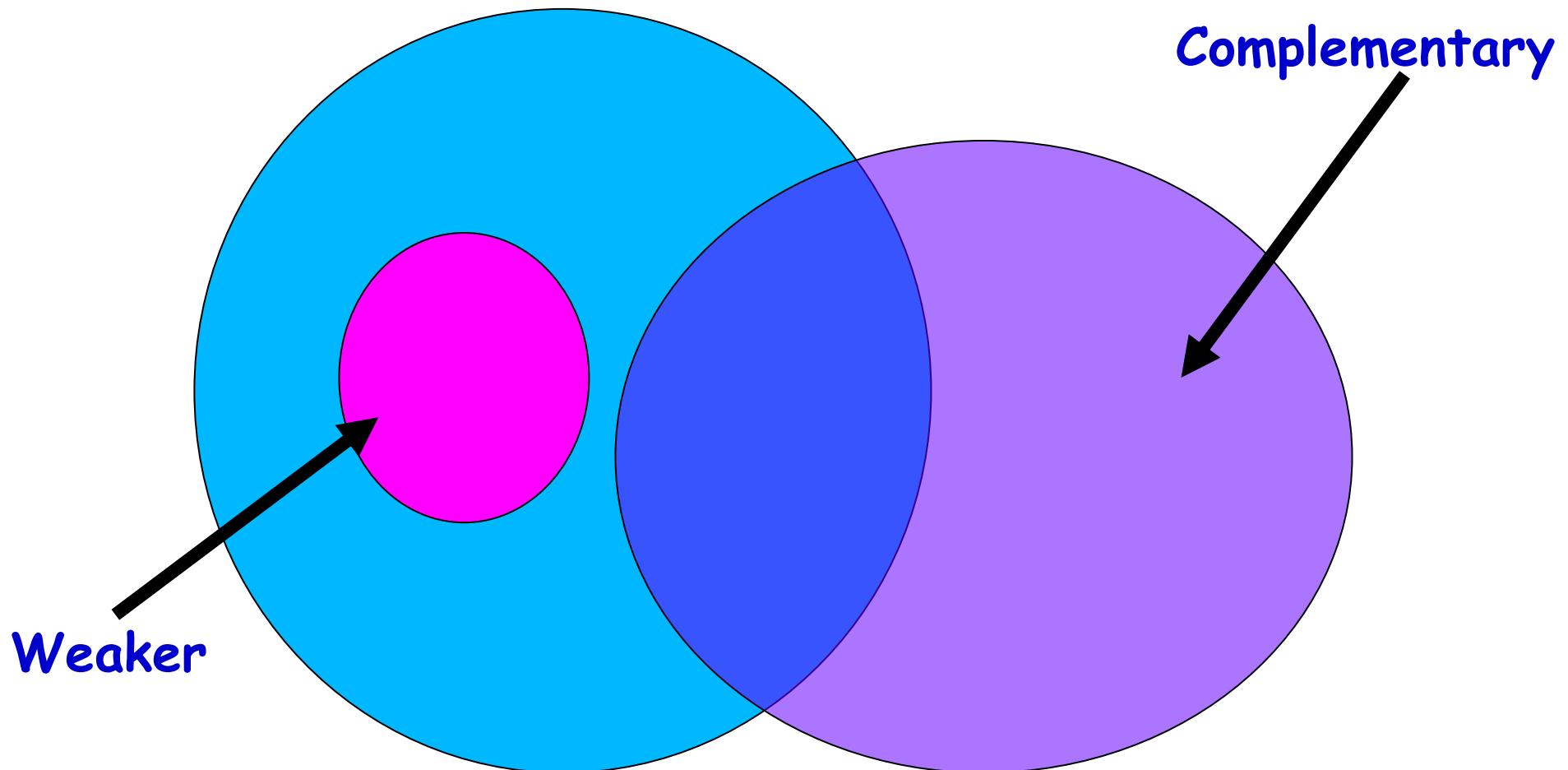
# Branch Coverage

- 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:

# executed branches

# 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
  - $((c1.\text{and}.c2).\text{or}.c3)$ :
- 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)

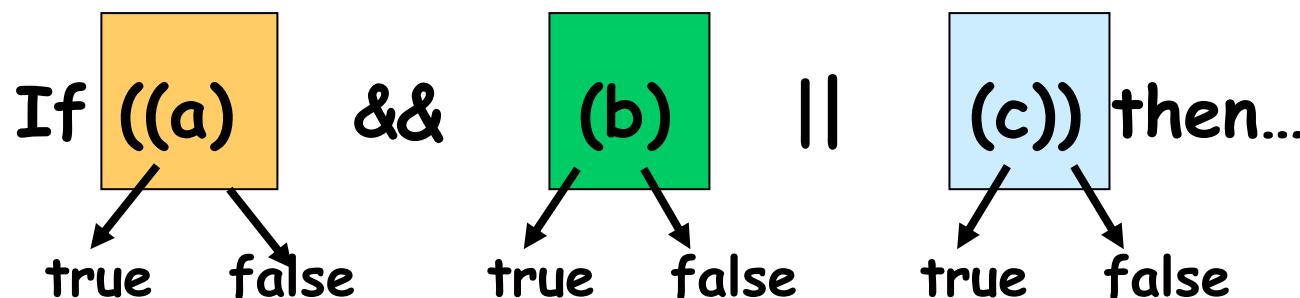
- Motivation: Effectively test **important combinations** of conditions, without exponential blowup in test suite size
  - “Important” combinations means: Each basic condition shown to independently affect the outcome of each decision
- Requires:
  - For each basic condition  $C$ , two test cases,
  - 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.

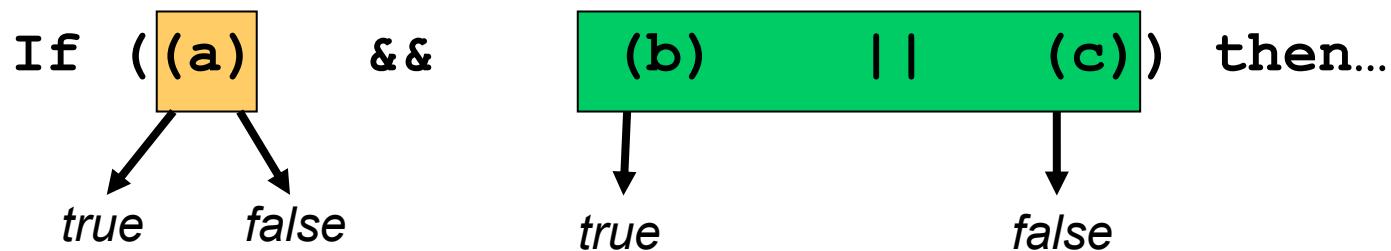
# Condition Coverage

- Every condition in the decision has taken all possible outcomes at least once.



# MC/DC Coverage

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



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

$(( (a \parallel b) \&& c) \parallel d) \&& e$

| 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

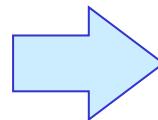
- MC/DC is
  - basic condition coverage (C)
  - branch coverage (DC)
  - plus one additional condition (M):  
every condition must *independently affect* the decision's output
- It is subsumed by compound conditions and subsumes all other criteria discussed so far
  - stronger than statement and branch 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.

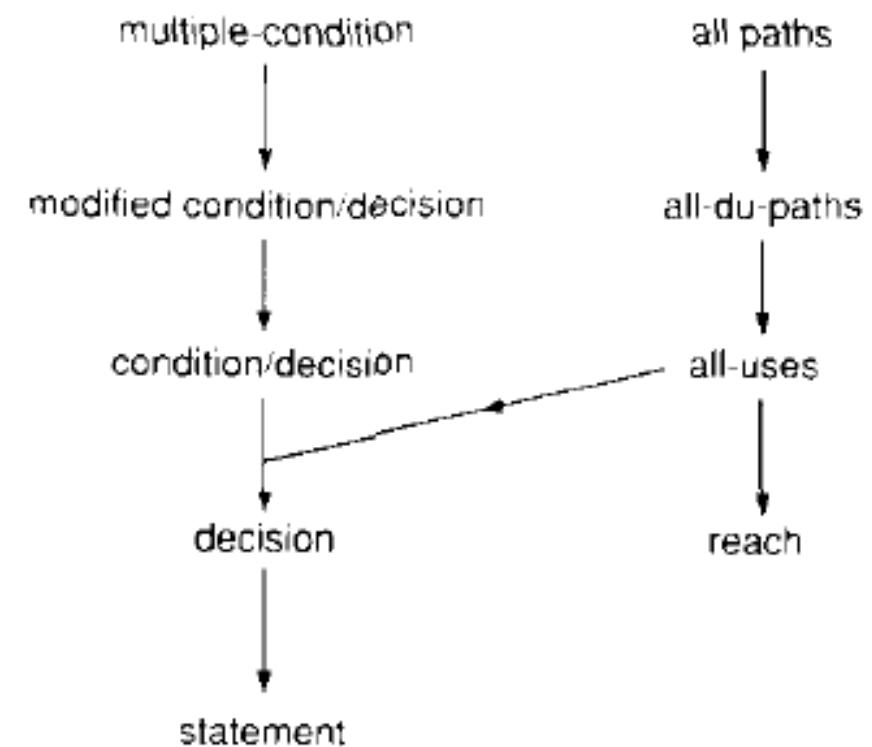


Fig. 2 Subsumption hierarchy

# 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.

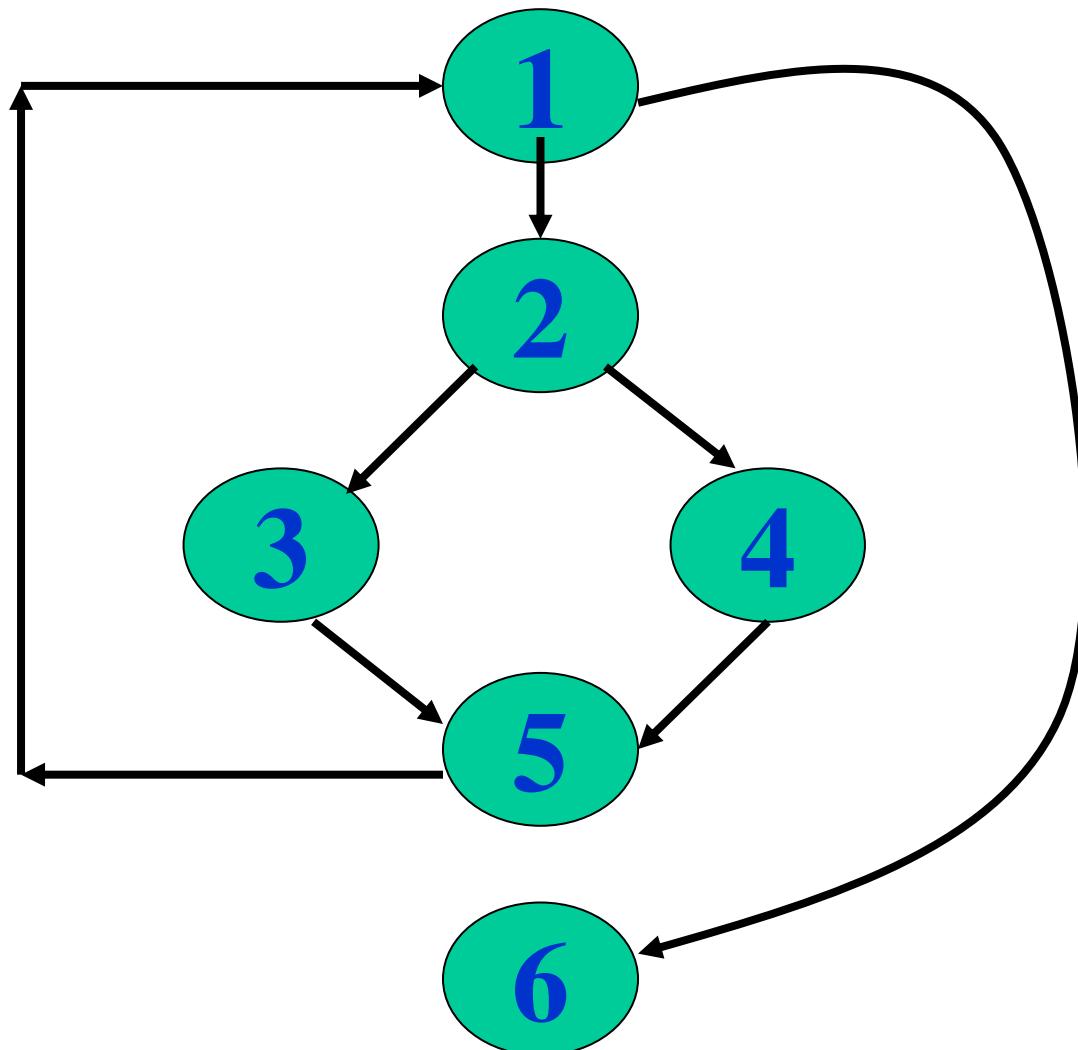
# How to Draw Control Flow Graph?

- 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;              }

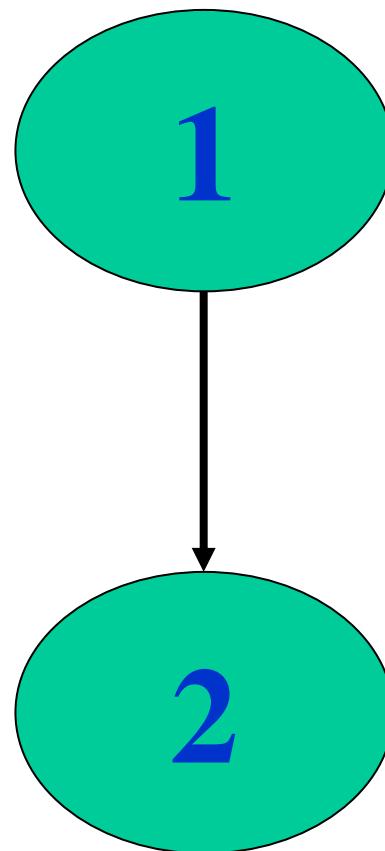
# Example Control Flow Graph



# How to Draw Control flow Graph?

- Sequence:

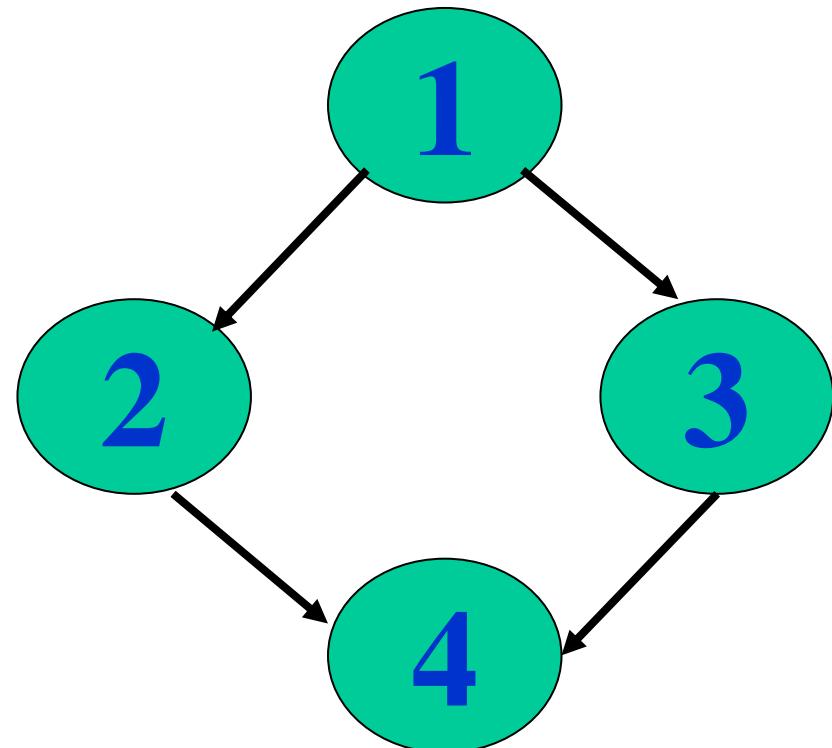
- 1  $a=5;$
- 2  $b=a*b-1;$



# How to Draw Control Flow Graph?

- Selection:

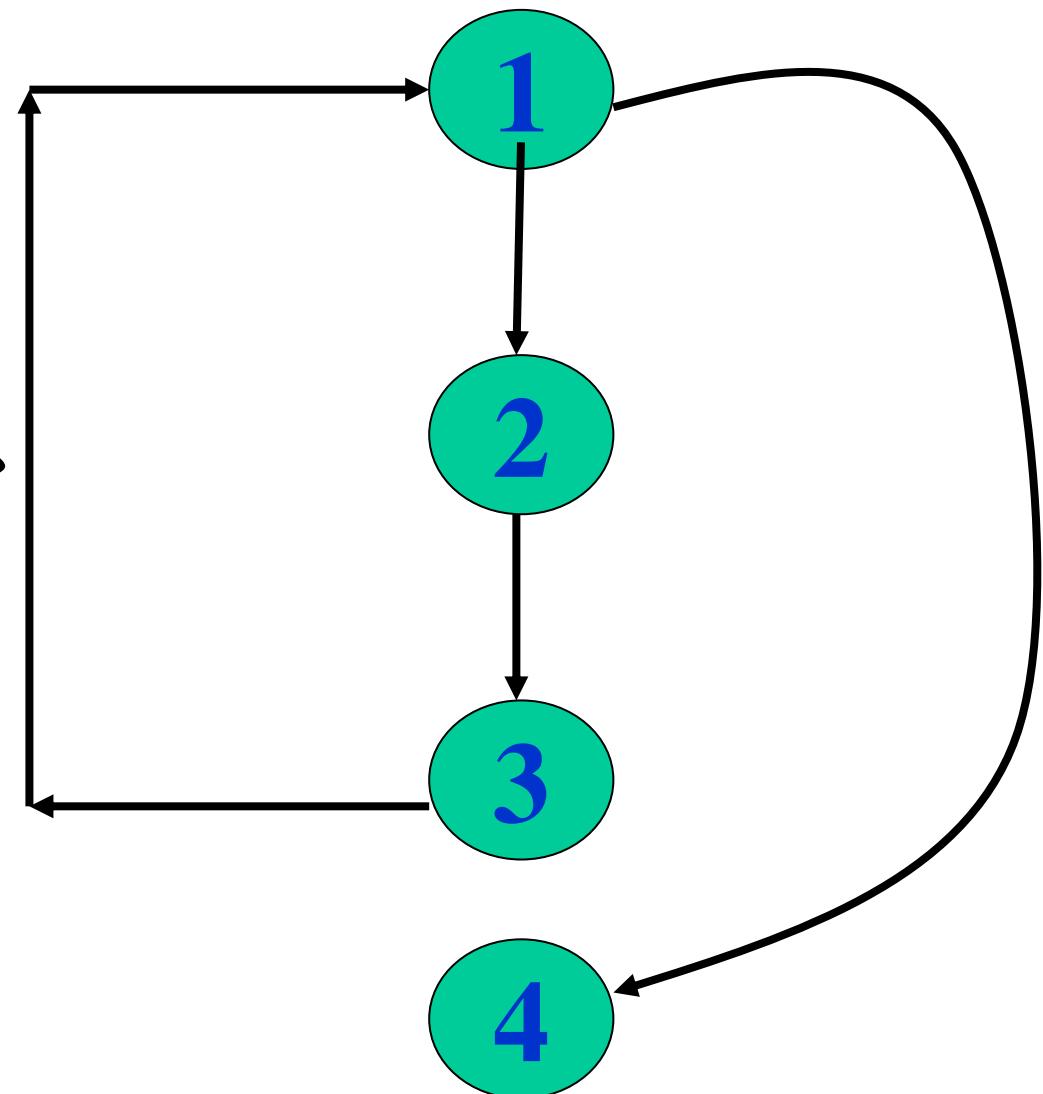
- 1 if( $a > b$ ) then
- 2                     $c = 3;$
- 3 else     $c = 5;$
- 4  $c = c * c;$



# 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 { . . . } ;

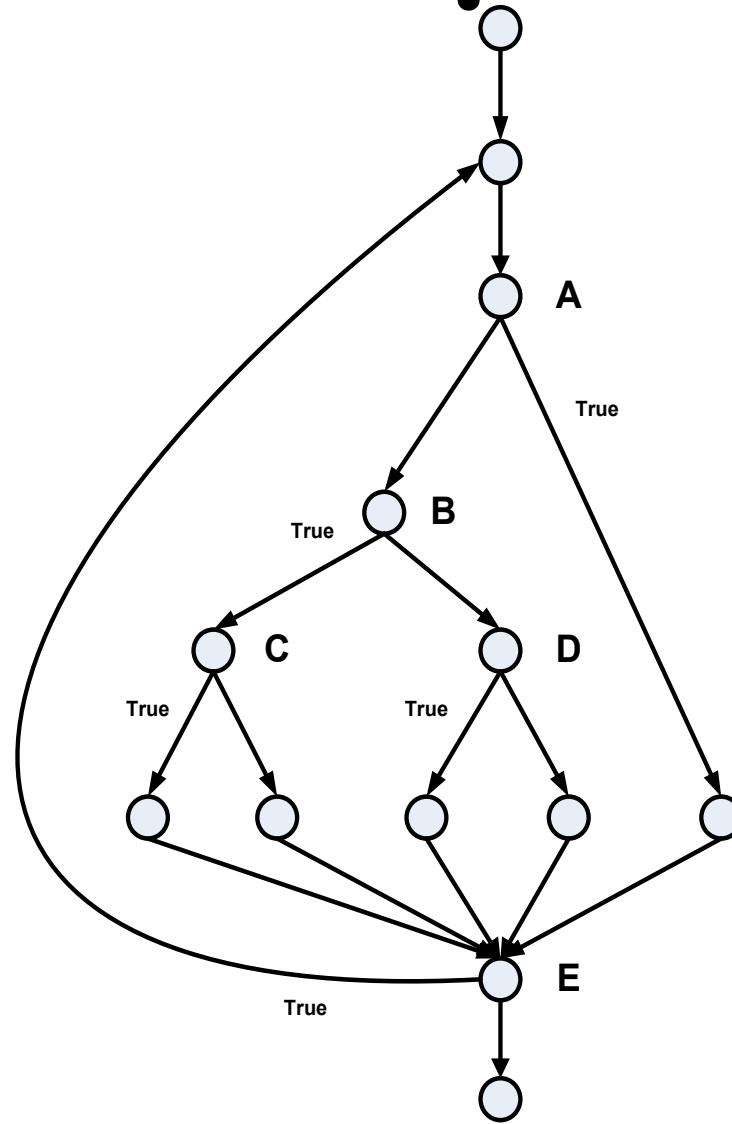
        else { . . . } ;

    }

}

While (E) ;

# Example Control Flow Graph



# 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.

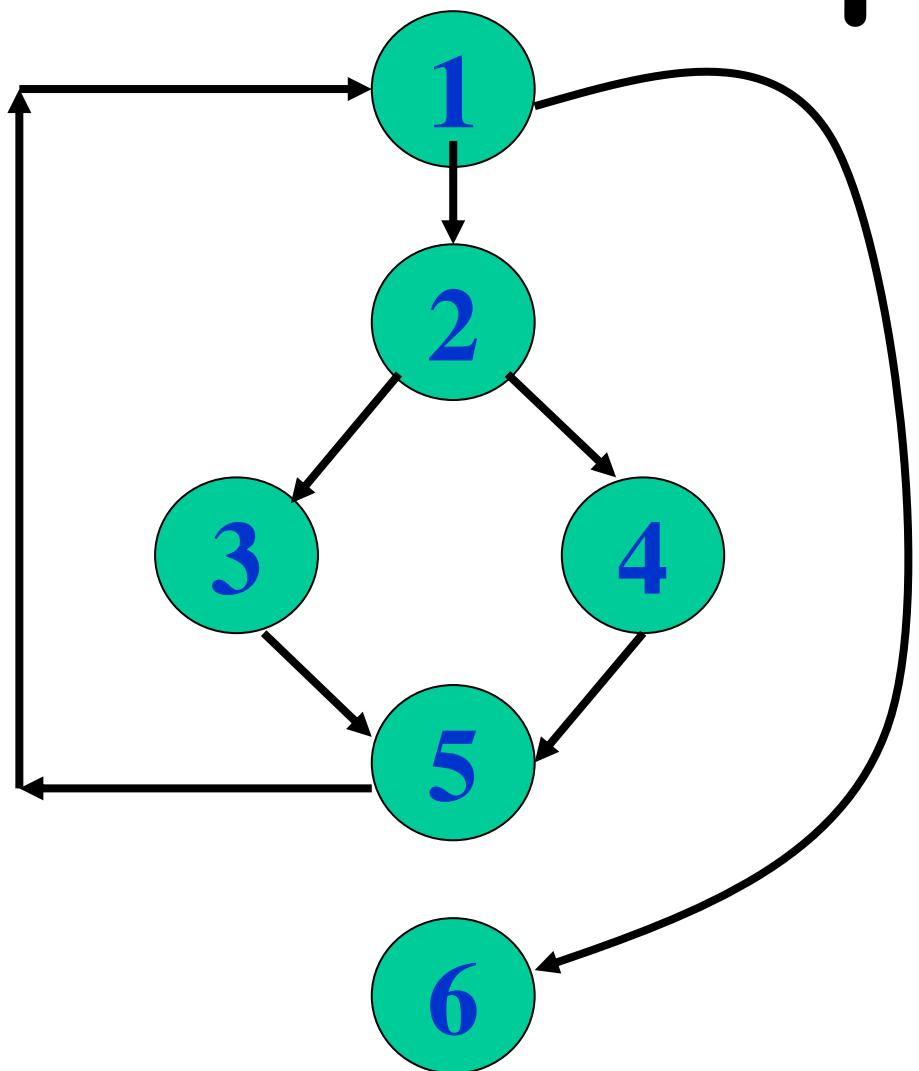
# McCabe's Cyclomatic Metric

- 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

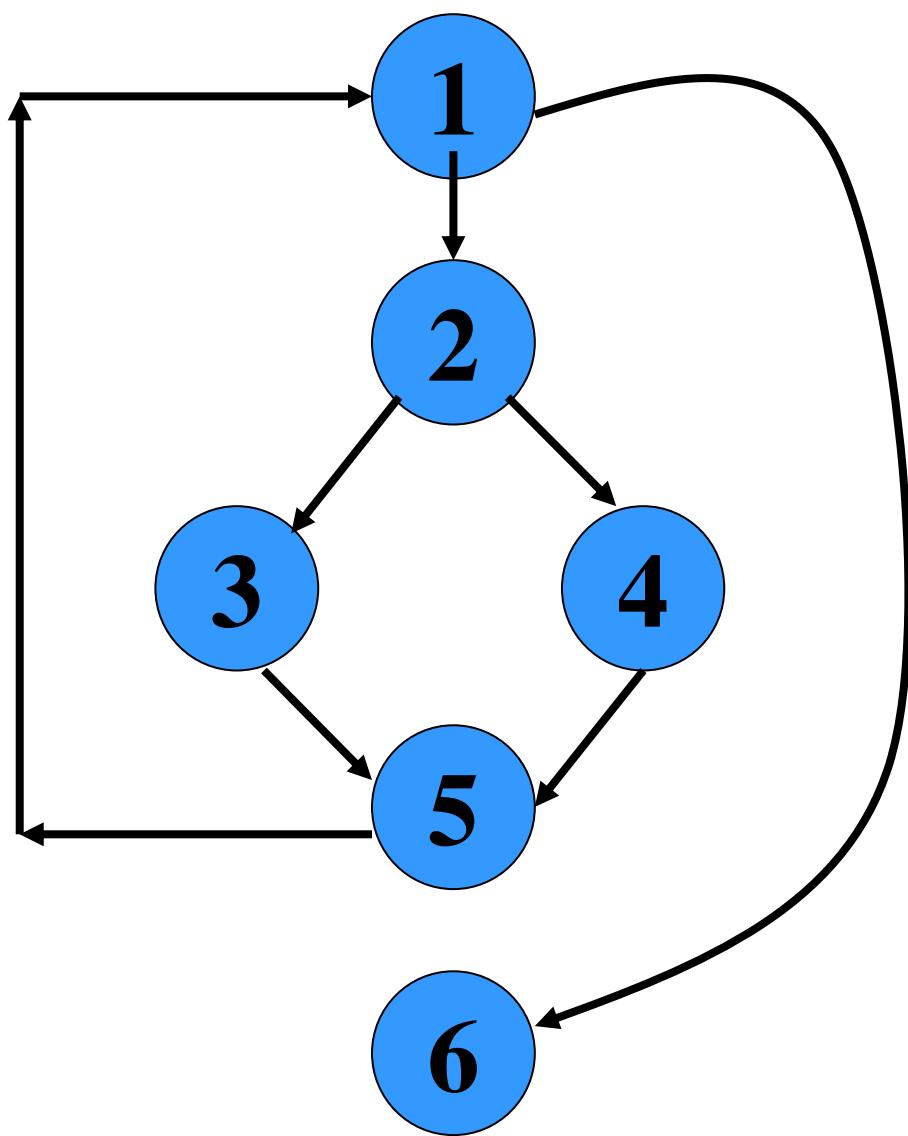


Cyclomatic complexity =  
 $7 - 6 + 2 = 3.$

# 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.

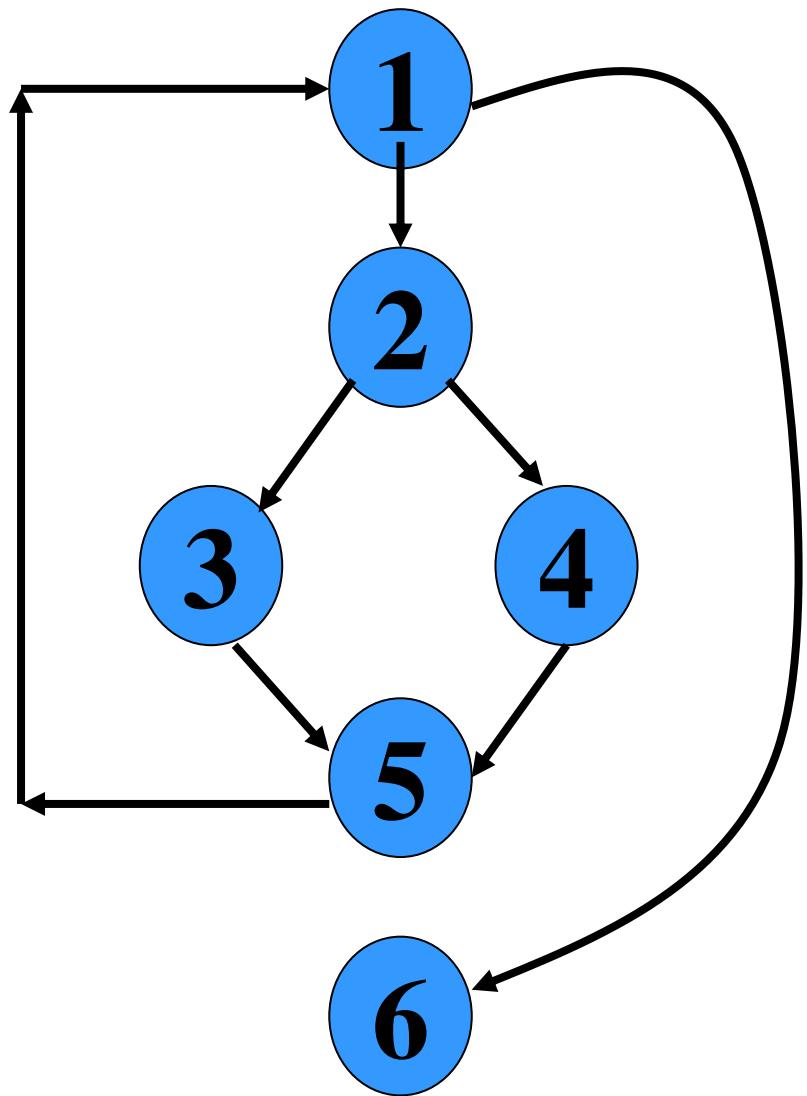
# Derivation of Test Cases

- 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){
- 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 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

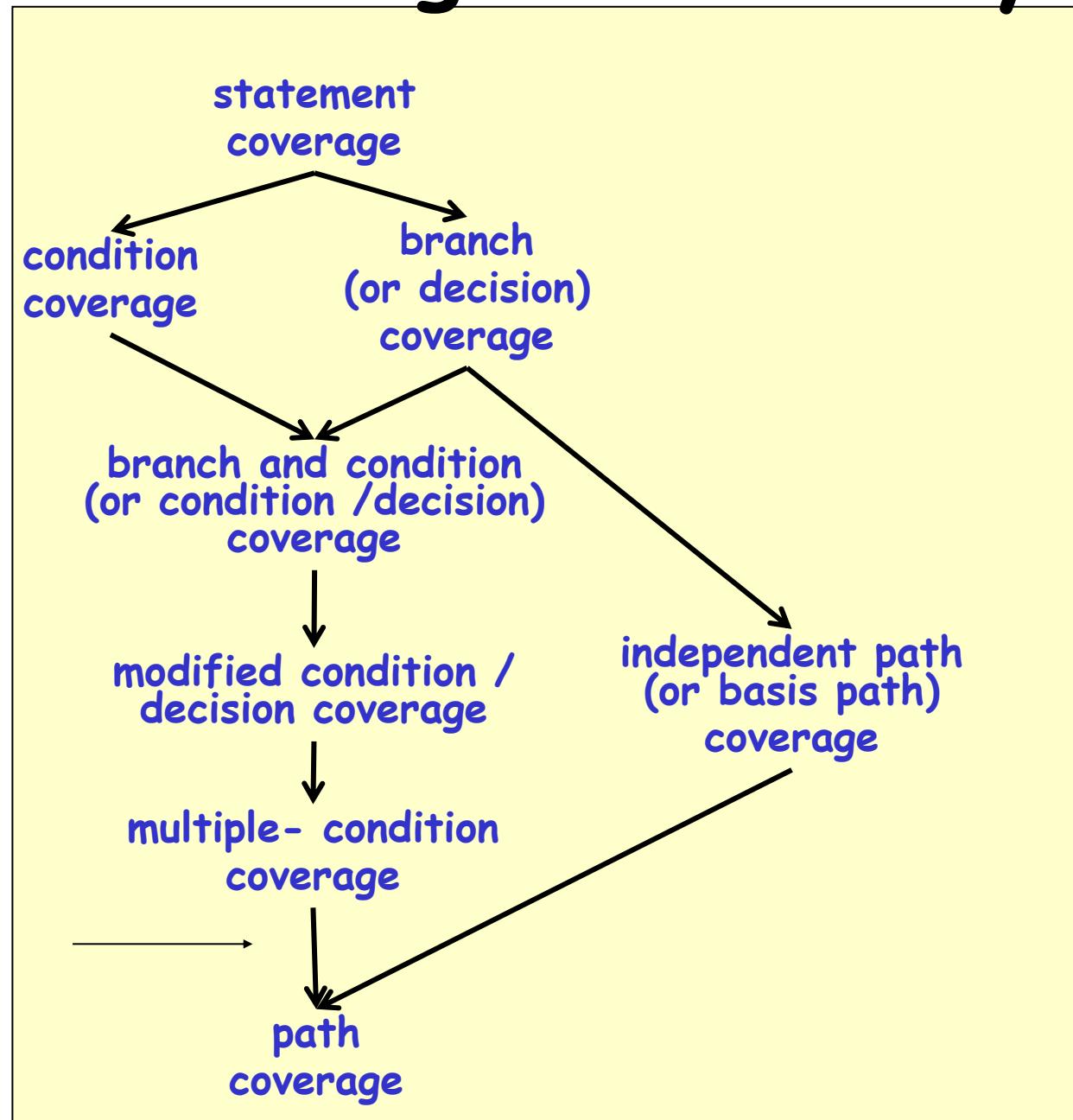
- From maintenance perspective,
  - Limit cyclomatic complexity of modules
    - To some reasonable value.
  - Good software development organizations:
    - Restrict cyclomatic complexity of functions to a maximum of ten or so. <sup>95</sup>

# 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  $S$ ,
  - $\text{DEF}(S) = \{X \mid \text{statement } S \text{ contains a definition of } X\}$
  - $\text{USES}(S) = \{X \mid \text{statement } S \text{ contains a use of } X\}$
  - Example: 1:  $a=b$ ;  $\text{DEF}(1)=\{a\}$ ,  $\text{USES}(1)=\{b\}$ .
  - Example: 2:  $a=a+b$ ;  $\text{DEF}(1)=\{a\}$ ,  $\text{USES}(1)=\{a,b\}$ .

# Data Flow-Based Testing

- A variable  $X$  is said to be **live** at statement  $S_1$ , if
  - $X$  is defined at a statement  $S$ :
  - There exists a path from  $S$  to  $S_1$  not containing any definition of  $X$ .

# DU Chain Example

```
1 X0{
2 a=5; /* Defines variable a */
3 While(C1) {
4 if (C2)
5 b=a*a; /*Uses variable a */
6 a=a-1; /* Defines variable a */
7 }
8 print(a); } /*Uses variable a */
```

# Definition-use chain (DU chain)

- $[X, S, S_1]$ ,
  - $S$  and  $S_1$  are statement numbers,
  - $X$  in  $\text{DEF}(S)$
  - $X$  in  $\text{USES}(S_1)$ , and
  - the definition of  $X$  in the statement  $S$  is live at statement  $S_1$ .

# 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

```
• 1 X(){
• 2 B1; /* Defines variable a */
• 3 While(C1) {
• 4 if (C2)
• 5 if(C4) B4; /*Uses variable a */
• 6 else B5;
• 7 else if (C3) B2;
• 8 else B3; }
• 9 B6 }
```

# Data Flow-Based Testing

- $[a, 1, 5]$ : a DU chain.
- Assume:
  - $\text{DEF}(X) = \{B1, B2, B3, B4, B5\}$
  - $\text{USED}(X) = \{B2, B3, B4, B5, B6\}$
  - 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.