SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR

RUBAB.JAFFAR@NU.EDU.PK

Part II-Software Testing

Structural Testing

Lecture # 22, 23, 24

1,2,4 Nov

TODAY'S OUTLINE

- Structural Testing
- Control Flow Testing
  - Branch testing
  - Statement testing
  - Condition testing
  - Path testing
- Guest session

STRUCTURAL TESTING

- More technical than functional testing.
- It attempts to design test cases from the source code and not from the specifications.
- The source code becomes the base document which is examined thoroughly in order to understand the internal structure and other implementation details.
- Structural testing techniques are also known as white box testing techniques
- Many structural testing techniques are available
  - control flow testing,
  - data flow testing,
  - slice based testing and
  - mutation testing.

CONTROL FLOW TESTING

- Identify paths of the program and write test cases to execute those paths. PATHS?
- There may be too many paths in a program and it may not be feasible to execute all of them. As the number of decisions increase in the program, the number of paths also increase accordingly.
- Every path covers a portion of the program. We define ‘coverage’ as a ‘percentage of source code that has been tested with respect to the total source code available for testing’.
- Write test cases to achieve a reasonable level of coverage using control flow testing.
- The most reasonable level may be to test every statement of a program at least once before the completion of testing.
- Testing techniques based on program coverage criterion may provide an insight about the effectiveness of test cases.

CONTROL FLOW TESTING

- Some of such techniques are discussed which are part of control flow testing.
- Statement Coverage
- Branch Coverage
- Condition Coverage

STATEMENT COVERAGE

- We want to execute every statement of the program in order to achieve 100% statement coverage.
- Consider the following portion of a source code along with its program graph.

```
      #include<stdio.h>
      #include<conio.h>

1.    void main()
2.    {
3.    int a,b,c,x=0,y=0;
4.    clrscr();
5.    printf("Enter three numbers:");
6.    scanf("%d %d %d",&a,&b,&c);
7.    if((a>b)&&(a>c)){
8.              x=a*a+b*b;
9.    }
10.   if(b>c){
11.             y=a*a-b*b;
12.   }
13.   printf("x= %d y= %d",x,y);
14.   getch();
15.   }
```

$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & 6 \arrow[dlll] \\
 & & 7 \arrow[dl] \arrow[dddr] & & & \\
 & 8 \arrow[d] & & & & \\
 & 9 \arrow[drr] & & & & \\
 & & & 10 \arrow[dl] \arrow[dddr] & & \\
 & & 11 \arrow[d] & & & \\
 & & 12 \arrow[drr] & & & \\
 & & & & 13 \arrow[d] & \\
 & & & & 14 \arrow[r] & 15
\end{tikzcd}$$

TEST CASE

- a=9, b=8, c=7, all statements are executed and we have achieved 100% statement coverage by only one test case. The total paths of this program graph are given as:
  - 1–7, 10–15
  - 1–7, 10, 13–15
  - 1–10, 13–15
  - 1–15
- The cyclomatic complexity of this graph is:
- V(G) = e – n + 2P = 16 – 15 + 2 = 3
- Hence, independent paths are three and are given as:
  - 1–7, 10, 13–15
  - 1–10, 13–15
  - 1–7, 10–15
- Only one test case may cover all statements but will not execute all possible four paths and not even cover all independent paths.

BRANCH COVERAGE

- We want to test every branch of the program. Hence, we wish to test every ‘True’ and ‘False’ condition of the program.
- If we select a = 9, b = 8, c = 7, we achieve 100% statement coverage and the path followed is given as (all true conditions): Path = 1–15
- We also want to select all false conditions with the following inputs:
- a = 7, b = 8, c = 9, the path followed is Path = 1–7, 10, 13–15
- These two test cases out of four are sufficient to guarantee 100% branch coverage. The branch coverage does not guarantee 100% path coverage but it does guarantee 100% statement coverage.

CONDITION COVERAGE

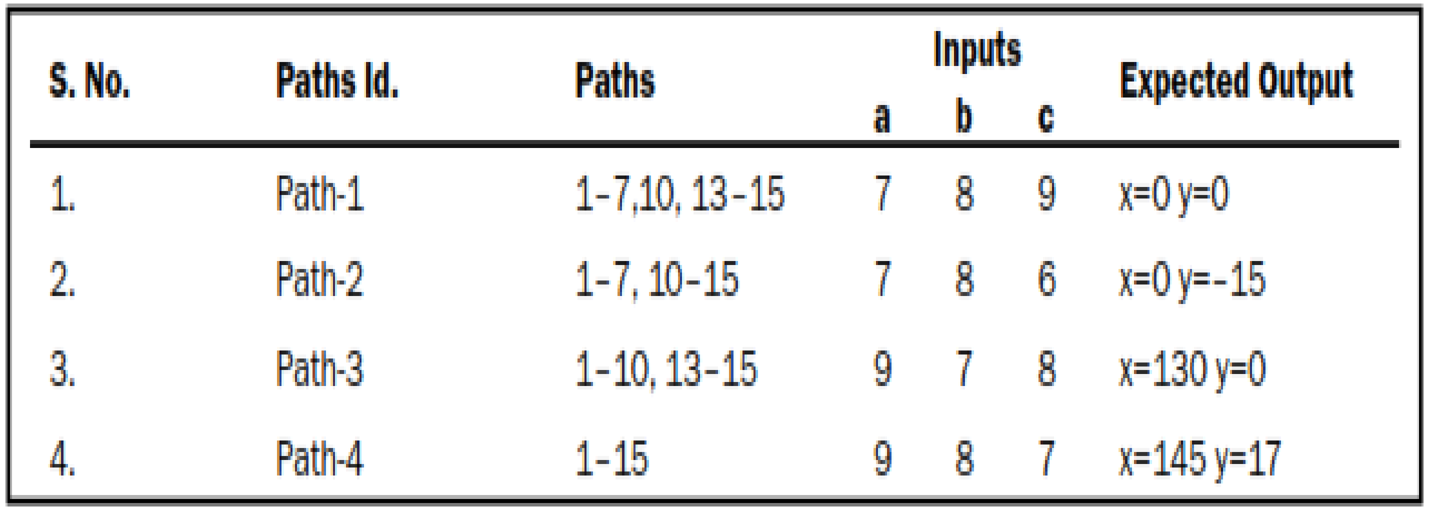
- Condition coverage is better than branch coverage because we want to test every condition at least once. However, branch coverage can be achieved without testing every condition.
- Considering the example on slide 6, statement number 7 has two conditions (a>b) and (a>c). There are four possibilities namely:
  - First is true, second is false
  - Both are true
  - First is false, second is true
  - Both are false
- If a > b and a > c, then the statement number 7 will be true (first possibility). However, if a < b, then second condition (a > c) would not be tested and statement number 7 will be false (third and fourth possibilities). If a > b and a < c, statement number 7 will be false (second possibility). Hence, we should write test cases for every true and false condition. Selected inputs may be given as:
- a = 9, b = 8, c = 10 (second possibility – first is true, second is false)
- a = 9, b = 8, c = 7 (first possibility when both are true)
- a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is false)
- Hence, these three test cases out of four are sufficient to ensure the execution of every condition of the program.

PATH COVERAGE

- In this coverage criteria, we want to test every path of the program. There are too many paths in any program due to loops and feedback connections. It may not be possible to achieve this goal of executing all paths in many programs. If we do so, we may be confident about the correctness of the program. If it is unachievable, at least all independent paths should be executed.
- 1–7, 10–15
- 1–7, 10, 13–15
- 1–10, 13–15
- 1–15

TEST CASES FOR ALL PATHS

- Execution of all these paths increases confidence about the correctness of the program. Inputs for test cases are given as:



S. No.	Paths Id.	Paths	Inputs a	Inputs b	Inputs c	Expected Output
1.	Path-1	1-7,10, 13-15	7	8	9	x=0 y=0
2.	Path-2	1-7, 10-15	7	8	6	x=0 y=-15
3.	Path-3	1-10, 13-15	9	7	8	x=130 y=0
4.	Path-4	1-15	9	8	7	x=145 y=17

PATH TESTING

- Path testing guarantee statement coverage, branch coverage and condition coverage.

EXAMPLE: PERFORM PATH TESTING

```
public static void search ( int key, int 0elemArray, Result r ) {
1. int bottom =0 ;
2. int top =elemArray.length - 1 ; int mid;
3. r.found =false ;
4. r.index =-1 ;
5. while ( bottom <= top ) {
6 mid =(top + bottom) / 2 ;
7 if (elemArray [mid] = key) {
8 rindex = mid;
9 r.found =true ;
10 return ; } // if part
else {
11 if (elemArray [mid] < key)
12 bottom = mid + 1 ;
else
13 top = mid - 1 ; }
} //while loop
14. } //search
```

That is all