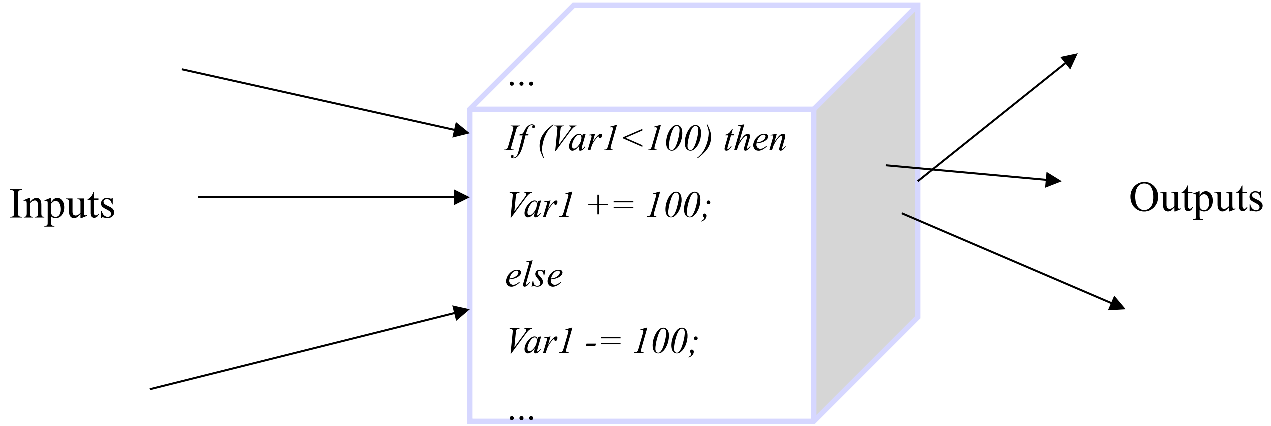
Software Verification and Validation

Structural Testing
Control Flow Graph

Structural Testing

- Structural testing or “White Box” testing or “Glass-Box” testing
- Use the logic and structure of the developed code for generating test cases
  - Source code must be available
- Key Concept: Code Coverage
  - For every element e in a specific class E, there should be at least one test case t that exercises that element e



Structural Testing for Code Coverage and Test Adequacy

- Measuring the adequacy of testing activities
  - When to stop?
  - Have we tested enough?
- Code coverage and test adequacy: How?
  - If one element in a specific class remains unexecuted in spite of execution of all test cases, we may generate additional test cases that exercise the remaining elements
- A complement to functional testing and not a substitution
  - Not for: How should I choose test cases
  - But for: What additional test cases I need that makes the entire test pool through

Structural Testing – Based on Control and Data Flow Criteria

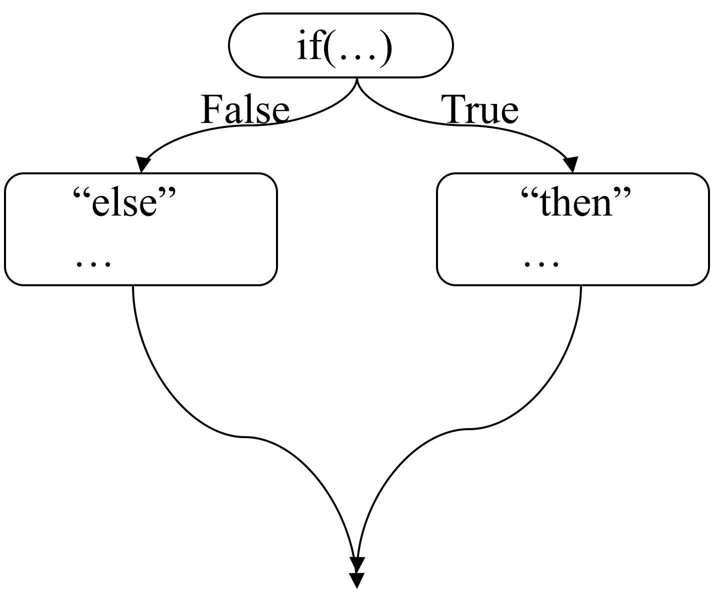
- Control flow testing criteria
  - Defined for particular classes of elements by requiring the execution of all such elements of the program
    - Example of elements: “if” statement
  - Criteria based on control flow testing
    - Function, statement, branch, path , etc. coverage criteria
- Data flow testing criteria
  - Defined for tracing the flow of data and data dependencies instead of control
  - Later in detail
  - Criteria based on data flow testing
    - Definition/use, Computation/use, etc. coverage criteria

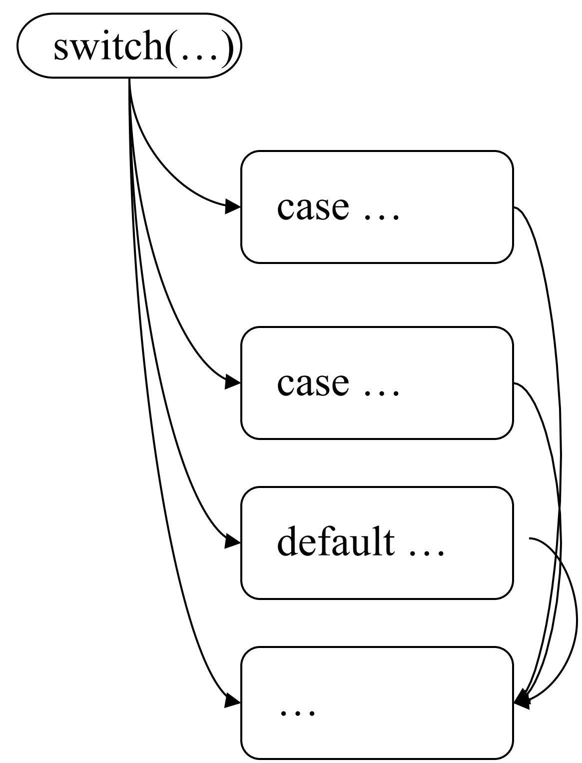
Structural Testing – Based on Control Flow Criteria

Control Flow Graphs

Control Flow Graph (CFG)

- Control flow of a module can be represented as control flow graph (CFG)
- A directed graph in which:
  - Nodes represent regions of the source code (e.g., a single statement, or a block of statements)
    - Basic block – A maximal program region with a single entry and single exit
    - Adjacent blocks can be collapsed into one block
  - Edges represent the program execution





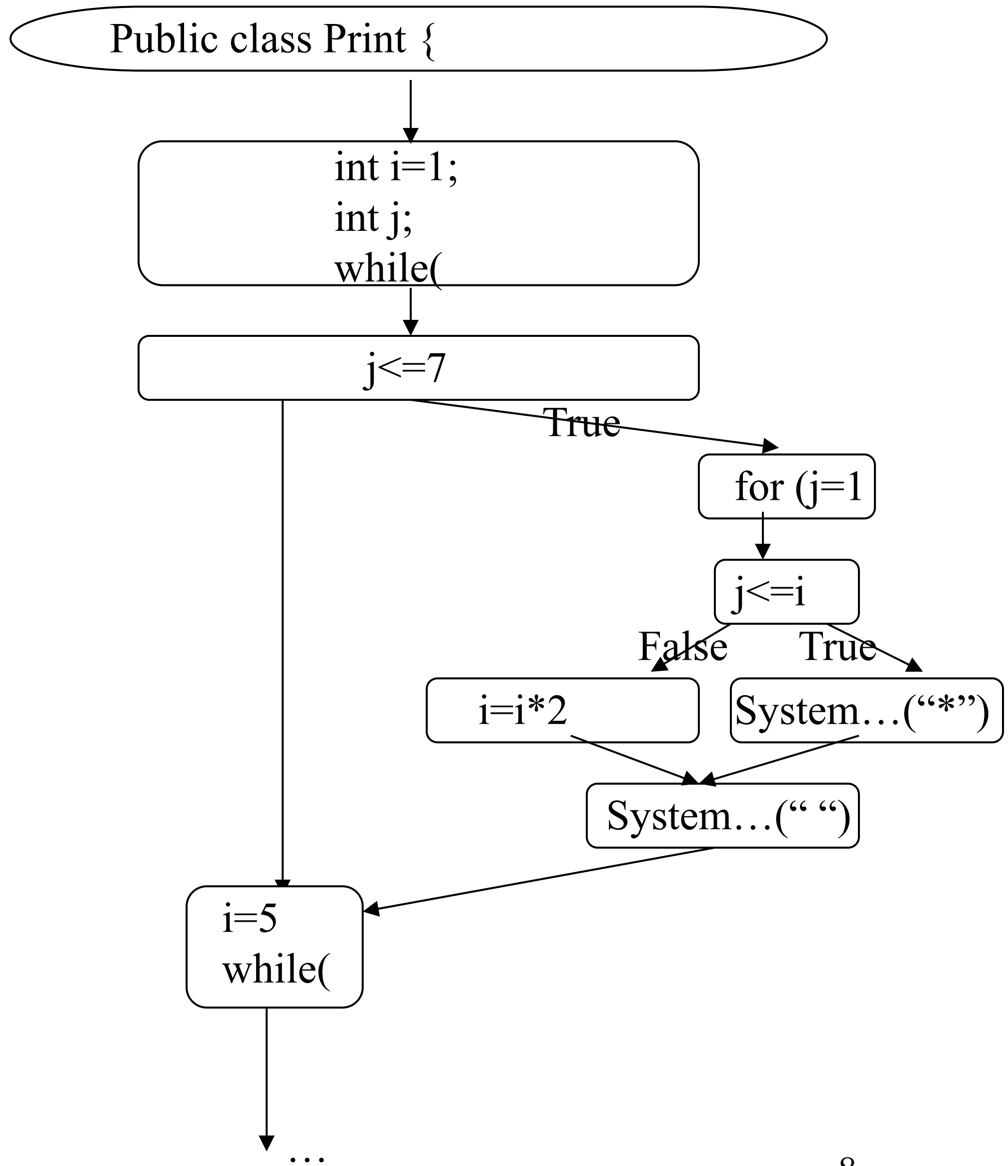
Control Flow Graph (CFG) – An Example

```
public class Print{
    public static void main(String[] args){
        int i=1;

        int j;
        while(i<=7){
            for(j=1;j<=i;j++)
                System.out.print("*");
            i=i*2;
            System.out.println();
        }
        i=5;
        while(i>=1){
            for(j=1;j<=i;j++)
                System.out.print("*");
            i=i-2;
            System.out.println();
        }
    }
}
```

```
C:\>javac Print.java

C:\>java Print

*

**

****

********

******

****

**
```

Control Flow Graph (CFG) – An Example

```
public class Print{
    public static void main(String[] args){
        int i=1;

        int j;
        while(i<=7){
            for(j=1;j<=i;j++)
              System.out.print("*");
            i=i*2;
            System.out.println();
            }
          i=5;
          while(i>=1){
              for(j=1;j<=i;j++)
                  System.out.print("*");
              i=i-2;
                   System.out.println();
              }
      }
}
```



Control Flow Graph (CFG)

- CFG retains some information about the program counter (PC)
  - PC : The address of the next instruction to be executed
- By CFG, we are able to determine
  1. Possible programs paths that can be executed
  2. Possible programs path that cannot be executed
- – CFG is used to define thoroughness criteria for testing

Control Flow Graph (CFG) – Important Issues

```
public class Test{
    public static void main(String[] args){
       int i=1; int L-value, H-value;

      if ( L-value == 1 || H-value == 1) {

          temp=1;

      } else {

          temp = 32 * L-value+H-value;
        }               3
     }
}
```

- Execution of a faulty statement may not always result in a failure
  - The internal state may not be corrupted with the given test case
  - The corrupt state may not propagate through execution to the exit point
- Example
  - If instead of 32, we had 3 erroneously. Test cases that execute the faulty statement with L-value=0 would not corrupt the state, leaving the fault unrevealed

Why Control Flow Testing Complements Functional Testing?

- Implementation of a single item of the specification by multiple parts of the program
  - Selection of test cases from the specification would not ensure the execution of all parts of the implementation
    - Eg. Implementation of hash tables (collision and no collision)
- Test cases satisfying control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria
  - E.g., *missing path* faults

Control Flow Testing and Unexecuted Elements

- Unexecuted elements can be because of:
  - Natural differences between specification and implementation
  - Flaws of the development process
  - Inadequacy of the specification (not to include those cases)
  - Inadequate functional testing

The Concept of Coverage Criteria

- Set of coverage elements covered by a coverage criterion
- Coverage element
  - A property that must be true about a set of test cases and their ability to exercise some elements of a given code
- More than one test case can exercise the same element
- One test case can exercise several elements
- Measure of coverage
  - The percentage of the elements exercised by the set of test cases

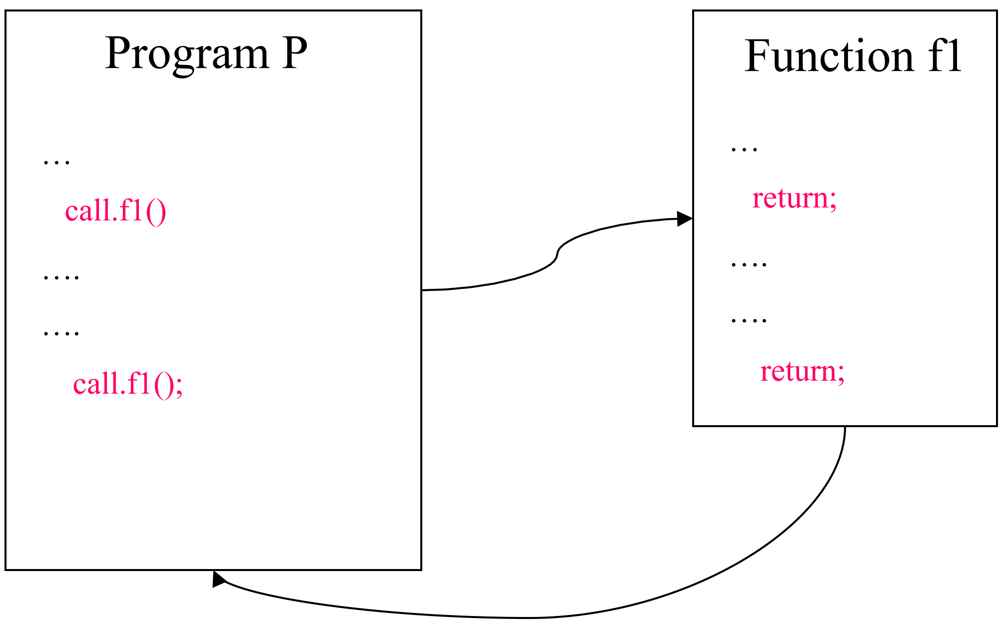
Function Coverage

- Also known as: routine, method, or procedure coverage
- If the set of test cases causes the program to call every function in the program, then we have 100% function coverage
- Example: A program P with five functions: f1, f2, f3, f4, and f5
  - Calling f1 causes calling f2 and f5
  - Calling f2 causes calling f3
  - f4 called directly

  - Ideal test case: A test case that causes calling all functions!!! Not possible
  - Ideal test suite: Minimum number of test cases causing
    - TC1: Calling f1
    - TC2: Calling f4
  - 100% function coverage

Procedure Call Coverage Testing

- Different than function coverage!!
- Known as: "Procedure entry and exit testing"
- Testing "all" calls to a procedure (entry points)
  - Calling from different modules
- Testing "all" exits from a procedure (exit points)
  - Existing from different points (testing "*return*" statements)



Statement Coverage

- Nodes of the control flow graph
- Idea – To reveal any possible fault, we must exercise every statement
- If for every statement in a given code, there is at least one test case in the test suite that executes that statement, the test suites is 100% statement coverage adequate
  - = Every node in the control flow graph model is visited by at least one test case
- For every test suite T
  - T_StatementCoverage = [#Statement Executed] / [Total # of Statements]
  - If T_StatementCoverage = 1, T satisfies statement adequacy criterion

Statement Coverage

- If for every statement in a given code, there is at least one test case in the test suite that executes that statement, the test suites is 100% statement coverage adequate.

- One test case can cover
  - TC1 = {“a5#”}

```
int let, dig, other, c;
    let = dig = other = 0;
    while( (c=getchar( )) != '\0' )
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
              ++let;
        else if( '0'<=c && c<='9' )
             ++dig;
        else
             ++other;
     printf("%d letters, %d digits, %d others\n", let, dig, other);
```

Statement Coverage

- If for every statement in a given code, there is at least one test case in the test suite that executes that statement, the test suites is 100% statement coverage adequate.

- One test case can cover
  - TC1={“a5#”}

```
int let, dig, other, c;
    let = dig = other = 0;
    while( (c=getchar( )) != '\0' )
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
              ++let;
        else if( '0'<=c && c<='9' )
             ++dig;
        else
             ++other;
     printf("%d letters, %d digits, %d others\n", let, dig, other);
```

Block (Node) Coverage

- Block – Sequence of statements such that if one of the statement is executed, the other statements in the block are executed as well.
- A block = A node in control flow graph
- For every test suite T
  - T_BlockCoverage = [#Node Executed] / [Total # of Nodes]
  - If T_BlockCoverage = 1, T satisfies block adequacy criterion
- The relation between statement and block coverage criteria is monotonic
  - For two given test suites T1 and T2:
    - If T1_StatementCoverage > T2_StatementCoverage, then T1_BlockCoverage > T2_BlockCoverage

Block (Node) Coverage

- Block – Sequence of statements such that if one of the statement is executed, the other statements in the block are executed as well.

```
int let, dig, other, c;
    let = dig = other = 0;
    while( (c=getchar( )) != '\0' )
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
              ++let;
        else if( '0'<=c && c<='9' )
             ++dig;
        else
             ++other;
     printf("%d letters, %d digits, %d others\n", let, dig, other);
```

- One test case can cover
    - TC1={“a5#”}

Block (Node) Coverage

- Block – Sequence of statements such that if one of the statement is executed, the other statements in the block are executed as well.
- 100% statement coverage = 100% Block coverage

- One test case can cover
  - TC1={“a5#”}

```
int let, dig, other, c;
    let = dig = other = 0;
    while( (c=getchar( )) != '\0' )
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
              ++let;
        else if( '0'<=c && c<='9' )
             ++dig;
        else
             ++other;
     printf("%d letters, %d digits, %d others\n", let, dig, other);
```

Branch (Decision) Coverage

- Exercising all possible decisions about the conditional expression (e.g., if)
  - True and False
- Branch coverage
  - Two coverage elements for each decision in a code
    - TRUE element – There is at least one test case that evaluates the decision to “true”
    - FALSE element - There is at least one test case that evaluates the decision to “false”
- Every edge in the control flow graph is exercised by a test case
- A test suite might achieve 100% statement and block coverage but “not” achieve 100% branch coverage
  - How?

Branch (Decision) Coverage

- A new program with two lines commented
  - There is no test case that covers the “false” branch of the latest if statement
- Branch coverage is usually used to test whether there is a missing code or not

```
int let, dig, other, c;
   let = dig = other = 0;
   while( (c=getchar( )) != '\0' )
      if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
          ++let;
      else if( '0'<=c && c<='9' )
          ++dig;
      //else
      //    ++other;
   printf("%d letters, %d digits, %d others\n", let, dig, other);
```

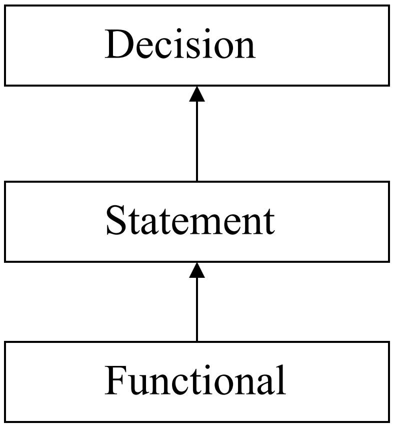
Branch (Decision) Coverage

- Needed test cases
  - if(('A'<=c && c<='Z') || ('a'<=c && c<='z')) : true
  - if(('A'<=c && c<='Z') || ('a'<=c && c<='z')) : false
  - if('0'<=c && c<='9') : true
  - if('0'<=c && c<='9') : false

```
int let, dig, other, c;
    let = dig = other = 0;
    while( (c=getchar( )) != '\0' )
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
             ++let;
        else if( '0'<=c && c<='9' )
             ++dig;
        else
             ++other;
     printf("%d letters, %d digits, %d others\n", let, dig, other);
```

The Relations So Far

- Statement coverage implies functional coverage
  - Statement coverage is stronger than functional coverage
- Decision coverage implies statement coverage
  - Decision coverage stronger than statement coverage

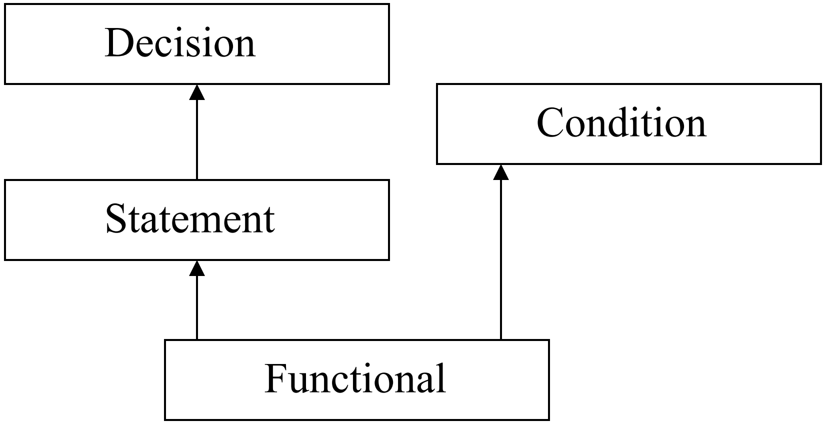


Condition Coverage

- Testing Boolean expressions (predicates) controlling a branch
- Condition – Atomic expression with no logical operators within decisions
  - Logical operators: &&, ||, etc.
  - ('A'<=c && c<='Z') || ('a'<=c && c<='z') = Four conditions
  - ('A'<=c && c<='Z') || ('a'<=c && c<='z') = Four conditions
  - ('0'<=c && c<='9') = Two conditions
  - ('0'<=c && c<='9') = Two conditions
- Condition Adequacy Criterion
  - Each evaluation (true and false) of condition must be covered
  - There is at least one test case that evaluates the condition’s decision to be TRUE
  - There is at least one test case that evaluates the condition’s decision to be FALSE

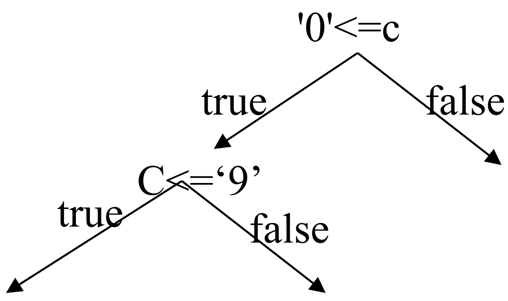
The Relations So Far

- Condition coverage is not stronger than decision coverage
  - We can have 100% condition coverage without having 100% decision coverage
  - We can have 100% decision coverage without having 100% condition coverage
- Condition coverage and decision coverage are incomparable



Multi-Condition (Compound Condition) Coverage

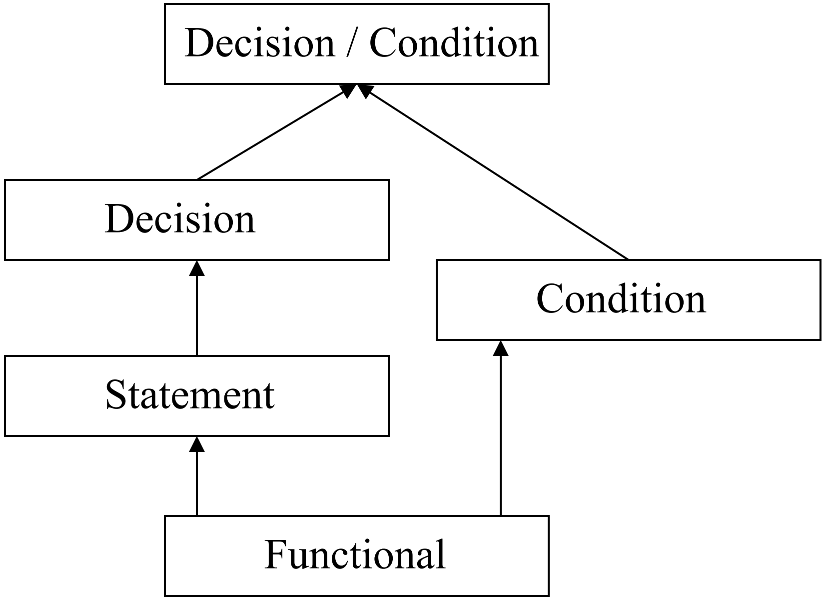
- k coverage elements for each decision
  - k = # feasible combination of true/false values for conditions in decisions
- For compound Boolean expression, different combinations of condition's decision must be considered
- Similar as covering paths to leaves of the evaluation tree for the expression
- ('0'<=c && c<='9')
  - '0"<=c :T c<='9':T
  - '0"<=c :T c<='9':F
  - '0"<=c :F c<='9':T
  - '0"<=c :F c<='9':F



Covering three paths

Decision / Condition (Condition / Decision (CD)) Coverage (DC)

- Covering all decision coverage elements as well as covering all condition coverage elements
  - 100% Condition coverage + 100% decision coverage = 100% CD



Modified Condition / Decision Coverage (MC/DC)

1. Every entry and exit point of the program has been invoked, at least once
2. Every condition in a decision has been evaluated all possible truth values, at least once
3. Every decision has been taken all possible branches, at least once

Short Circuit (SC) Evaluation

- Short circuit evaluation
  - When evaluating X && Y, if X is false, do not evaluate Y
  - When evaluating X || Y, if X is true, do not evaluate Y
- Condition coverage does not consider it
- Example
  - if('0'<=c && c<='9')

Into

```
if('0'<=c) {
        if(c <='9') {
```

Short Circuit (SC) Evaluation

- SC condition coverage:
  - Two coverage elements for each condition
    1. There is at least one test case that evaluate the condition to true assuming SC evaluation
    2. There is at least one test case that evaluate the condition to false assuming SC evaluation
  - By SC all conditions and decisions are executed both ways
    - SC is stronger than DC coverage

Path Coverage

- Exercising some sequence of decisions
- Path – Sequence of statements executed during one entire run of the program
- Element of path coverage
  - There is at least one test case that executes the path
- Path coverage – One element coverage for each path
- Path adequacy criterion
  - A test suite T satisfies the path coverage criterion for a program P if and only if for each path p of P, there is at least one test case that exercises that path
- Each condition in the code is part of a path
  - Path coverage is stronger than condition coverage

Path Coverage and Cyclomatic Complexity

- Path_coverage = #Executed_paths / #Total_paths
- In most cases, programs contain loops
  – The denominator approaches to infinity
- From graph theory
  – Basis set – A subset of sub-paths forming all other paths
  – Connected graph -
  – Every connected graph with n nodes, e edges, and c connected components has a basis set of e - n + c
  – By adding an edge from exit to the entry, we are able to produce a single connected component. The formula then is: e - n + 2

Loop Coverage

- Covering loops with respect to six elements for each loop with condition C
  - At least one test case causing C to be evaluated false
    - Executes the loop 0 times
  - At least one test case causing C to be evaluated true for the first time, false for the second time
    - Executes the loop only 1 times
  - At least one test case causing C to be evaluated true for two times
  - If there exists a boundary value n for the loop
    - At least one test case causing C to be evaluated true n time
    - At least one test case causing C to be evaluated true n-1 time
    - At least one test case causing C to be evaluated true n+1 time

Loop Coverage

- TC1 = “\0”
- TC2 = “a\0”
- TC3 = “ab\0”
- No boundary n

```
while( (c=getchar( )) != '\0' )
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
             ++let;
        else if( '0'<=c && c<='9' )
             ++dig;
        else
             ++other;
```

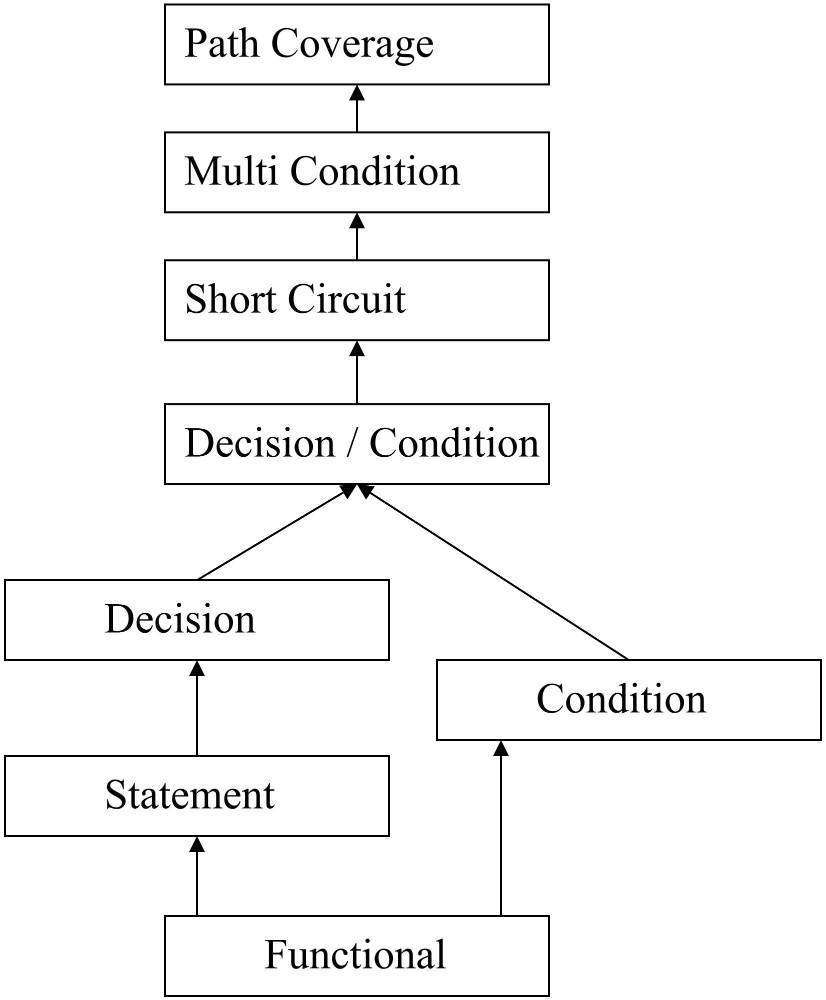
Loop Coverage

- TC1 = “\0”
- TC2 = “a\0”
- TC3 = “ab\0”
- TC4 = “abcdefghi\0”
- TC5 = “abcdefgh\0”
- TC6 = “abcdefghij\0”

```
while( (i < 9)
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
             ++let;
        else if( '0'<=c && c<='9' )
             ++dig;
        else
             ++other;
```

Relationships of Coverage

- Loop coverage not comparable



Coverage and Software Safety Standards

- According to US Radio Technical Commission for Aeronautics (RTCA) and DO-178B, five levels of criticalness
  - Level E – Failure of software has no effect on system
    - No code coverage required
  - Level D – Failure of software has minor effect on system
    - E.g., Inconvenience
    - No code coverage required
  - Level C – Failure of software has major effect on system
    - E.g., Discomfort
    - Statement coverage
  - Level B – Failure of software has sever major effect on system
    - E.g., Injury
    - Decision
  - Level A - Failure of software has disaster effect on system
    - E.g., Crash or explosion
    - Modified Condition / Decision coverage