White-Box Testing

White-Box Testing

- White-box testing (testing) is based on knowledge of the internal logic of an application's code.
- It determines whether the program-code structure and logic is faulty.
- White-box test cases are accurate only if the tester knows what the program is supposed to do.

Most Significant White-Box Techniques

- ❑Code coverage
- ❑Fault injection
- ❑Mutation testing

Code Coverage

- Code coverage defines the degree of source code which has been tested
- There are several criteria for the code coverage:

Statement coverage: The line of code coverage granularity.

Decision (branch) coverage: Control structure (for example, if-else) coverage granularity.

Condition coverage: Boolean expression (true-false) coverage granularity.

Paths coverage: Every possible route coverage granularity.

Function coverage: Program functions coverage granularity.

Entry/exit coverage: Call and return of the coverage granularity.

Statement Coverage

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$$

Decision Coverage

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

Fault Injection

- Fault injection is the process of injecting faults into software to determine how well (or badly) some SUT (System Under Test) behaves.
- Defects can be said to propagate AND
- The effects of defects are visible in program states
  - an error exists
  - a fault becomes a failure.

  Briefly;
- In software testing, fault injection is a technique for improving the coverage of a test by introducing faults to test code paths

System Under Test

- SUT is executed with specific input values to find failures in its behavior.
- The actual SUT should ensure that the design and code, environment (libraries, operating system and network support, and so on) are correct

Fault Injection Methods

Compile-Time Injections

- It is a fault injection technique where source code is modified to inject simulated faults into a system.

Run-Time Injections

- It makes use of software trigger to inject a fault into a software system during run time.
- The Trigger can be of two types
  - Time Based triggers
  - Interrupt Based Trigger

Mutation Testing

- Mutation testing validates tests and their data by running them against many copies of the SUT containing different, single, and deliberately inserted changes.
- Mutation testing helps to identify omissions in the code.

Value Mutation

```
public int Segment(int t[], int l, int u){
        // Assumes t is in ascending order, and l<u,
        // counts the length of the segment
        // of t with each element l<t[i]<u
        int k = 0;

        for(int i=0; i<t.length && t[i]<u; i++){
                if(t[i]>l){
                        k++;
                }
        }
        return(k);
}
```

Mutating to k=1 causes miscounting

Here we might mutate the code to read i=1, a test that would kill this would have t length 1 and have l < t[0] < u, then the program would fail to count t[0] and return 0 rather than 1 as a result

Statement Mutation

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Here we might consider deleting this statement (then count would be zero for all inputs, we might also duplicate this line in which case all counts would be doubled.

Decision Mutation

```
public int Segment(int t[], int l, int u){
        // Assumes t is in ascending order, and l<u,
        // counts the length of the segment
        // of t with each element l<t[i]<u
        int k = 0;

        for(int i=0; i<t.length && t[i]<u; i++){
                if(t[i]>l){
                        k++;
                }
        }
        return(k);
}
```

Mutating to t[i]>u will cause miscounting

We can model "one-off" errors in the loop bound by changing this condition to i<=t.length - provided array bounds are checked exactly this will provoke an error on every execution.

Coverage Criteria

Graph Coverage

Logic Coverage

Coverage Criteria

- A coverage criterion is simply a recipe for generating **test requirements (TR)** in a systematic way.

In other words:

- A coverage criterion is a rule or collection of rules that impose test requirements (TR) on a test set.

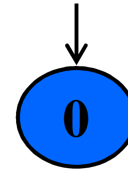
Graph Coverage Criteria

- Structural Coverage Criteria : Defined on a graph just in terms of nodes and edges
- Data Flow Coverage Criteria : Requires a graph to be annotated with references to variables

Graph and Test Paths

- A set N of nodes, N is **not empty**
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set E of edges, each edge from one node to another
- Test Path : A path that starts at an initial node and ends at a final node
  - Test paths represent execution of test cases
    - ✓ Some test paths can be executed by many tests
    - ✓ Some test paths cannot be executed by any tests

Node Coverage

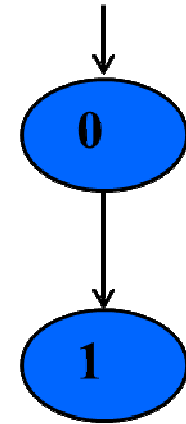


- A graph with only one node will not have any edges
- Node Coverage : TR contains each reachable node in G
- The length of Node Coverage is 0

In other words:

- Node Coverage: Test set satisfies node coverage on graph iff for every syntactically reachable node n in set N

Edge Coverage

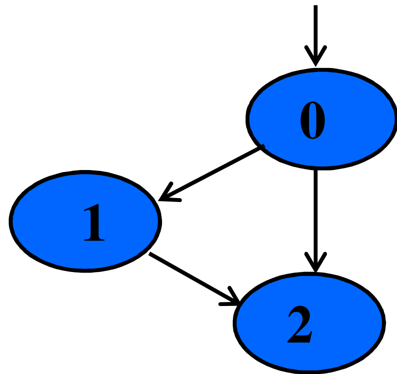


- Edge coverage is slightly stronger than node coverage
- Edge coverage needs to require Node Coverage on the graph

  BUT

- Edge Coverage will not subsume Node Coverage
- So we define "length up to 1" instead of simply "length 1"
  - The "length up to 1" allows for graphs with one node and no edges

Structural Coverage Example -1

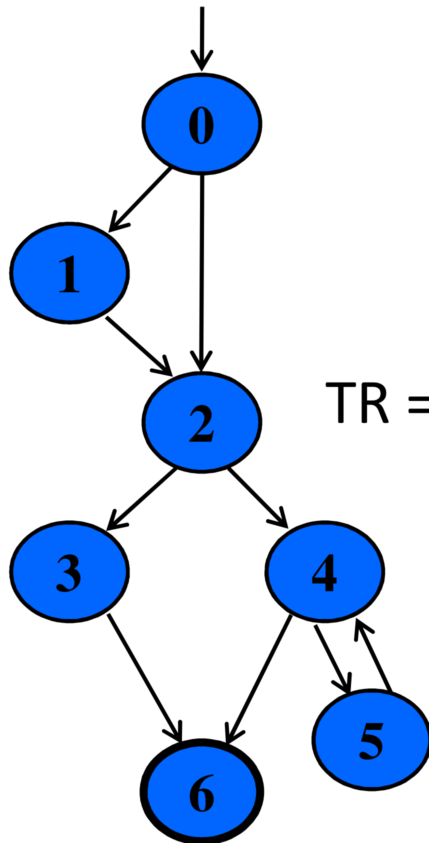


<u>Node Coverage (NC)</u> : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

<u>Edge Coverage</u>(EC) : TR = { (0,1), (0, 2), (1, 2) }
Test Paths = [0, 1, 2] [0, 2]

- NC and EC are only different when there is an edge and another subpath between a pair of nodes (as in an "if-else" statement)

Structural Coverage Example -2



Node Coverage

TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]

Edge Coverage

TR = { (0,1), (0,2), (1,2), (2,3), (2,4), (3,6), (4,5), (4,6), (5,4) }

Test Paths: [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]

path (t_1)= [0, 1, 2, 3, 6]

path (t_2)=[0, 2, 4, 5, 4, 6]

T={t_1 , t_2}