Coverage

- Literature of software testing is primarily concerned with various notions of coverage.

- Four basic kinds of coverage:
  - Graph coverage
  - Logic coverage
  - Input space partitioning
  - Syntax-based coverage

- Two purposes: to know what we have & haven’t tested, and to know when we can “safely” stop testing.
Need to Abstract Testing

- As we have seen, we can’t try all possible executions of a program

- How can we measure “how much testing” we have done and look for more things to test?
  - Could talk about modules we have and have not tested, or use cases explored
  - Could also talk structurally – what aspects of the source code have we tested?
Four Structures for Modeling Software

Graphs

Logic

Input Space

Syntax

Applied to

Source

Specs

Design

Use cases

Applied to

Source

FSMs

Specs

DNF

Applied to

Source

Models

Integ

Input
Graph Coverage

- **Graphs are the most commonly used structure for testing**
- **Graphs can come from many sources**
  - Control flow graphs
  - Design structure
  - FSMs and state charts
  - Use cases
- **Tests usually are intended to “cover” the graph in some way**
Definition of a Graph

- 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
  - $(n_i, n_j)$, $i$ is predecessor, $j$ is successor
Three Example Graphs

G_1: $N_0 = \{0\}$, $N_f = \{3\}$

G_2: $N_0 = \{0, 1, 2\}$, $N_f = \{7, 8, 9\}$

G_3: $N_0 = \{\}$, $N_f = \{3\}$

Not a valid graph
Paths in Graphs

- **Path**: A sequence of nodes – \([n_1, n_2, \ldots, n_M]\)
  - Each pair of nodes is an edge

- **Length**: The number of edges
  - A single node is a path of length 0

- **Subpath**: A subsequence of nodes in \(p\) is a subpath of \(p\)

- **Reach \((n)\)**: Subgraph that can be reached from \(n\)

![Diagram of paths in a graph]

**A Few Paths**
- \([0, 3, 7]\)
- \([1, 4, 8, 5, 1]\)
- \([2, 6, 9]\)

- \(\text{Reach } (0) = \{0, 3, 4, 7, 8, 5, 1, 9\}\)
- \(\text{Reach } (\{0, 2\}) = G\)
Test Paths and SESEs

- **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

- **SESE graphs**: All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N0 and Nf have exactly one node

---

Double-diamond graph
Four test paths
- \([0, 1, 3, 4, 6]\)
- \([0, 1, 3, 5, 6]\)
- \([0, 2, 3, 4, 6]\)
- \([0, 2, 3, 5, 6]\)
Visiting and Touring

- **Visit**: A test path $p$ visits node $n$ if $n$ is in $p$

  A test path $p$ visits edge $e$ if $e$ is in $p$

- **Tour**: A test path $p$ tours subpath $q$ if $q$ is a subpath of $p$

Path $[ 0, 1, 3, 4, 6 ]$

Visits nodes 0, 1, 3, 4, 6

Visits edges $(0, 1), (1, 3), (3, 4), (4, 6)$

Tours subpaths $[0, 1, 3], [1, 3, 4], [3, 4, 6], [0, 1, 3, 4], [1, 3, 4, 6]$
Tests and Test Paths

- **path** \( (t) \) : The test path executed by test \( t \)

- **path** \( (T) \) : The set of test paths executed by the set of tests \( T \)

- Each test executes one and only one test path

- A location in a graph (node or edge) can be reached from another location if there is a sequence of edges from the first location to the second
  - **Syntactic reach** : A subpath exists in the graph
  - **Semantic reach** : A test exists that can execute that subpath
Tests and Test Paths

Deterministic software – a test always executes the same test path

Non-deterministic software – a test can execute different test paths
Control Flow Graphs

- A **CFG** models all executions of a method by describing control structures
- **Nodes**: Statements or sequences of statements (basic blocks)
- **Edges**: Transfers of control
- **Basic Block**: A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses
- Rules for translating statements into graphs...
CFG : The if Statement

```c
if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}
```

Diagram:

1. `if (x < y) { y = 0; x = x + 1; } else { x = y; }`

   - Node 1: `if (x < y) { y = 0; x = x + 1; }`
   - Node 2: `y = 0` and `x = x + 1`
   - Node 3: `x = y`
   - Node 4: Transition between Node 1, Node 2, and Node 3
CFG : The if-Return Statement

If \( x < y \)
{
  return;
}
print (x);
return;

No edge from node 2 to 3. The return nodes must be distinct.
Loops

- Loops require “extra” nodes to be added
- Nodes that do not represent statements or basic blocks
CFG : while and for Loops

```c
x = 0;
while (x < y)
{
    y = f(x, y);
    x = x + 1;
}
```

```c
x = 0  
1  
dummy node
2
x < y  
x >= y
3
y = f(x, y)
4 
x = x + 1
```

```c
for (x = 0; x < y; x++)
{
    y = f(x, y);
}
```

```c
x = 0  
1  
2
x < y  
x >= y
3
y = f(x, y)
4
x = x + 1
5
```

- `x = 0;` implicitly initializes loop
- `x < y` implicitly increments loop
CFG : The case (switch) Structure

```c
read ( c );
switch ( c ) {
    case 'N':
        y = 25;
        break;
    case 'Y':
        y = 50;
        break;
    default:
        y = 0;
        break;
}
print (y);
```

Diagram:
1. read (c);
2. c = 'N'
   - y = 25; break;
3. c = 'Y'
   - default
   - y = 0; break;
4. y = 50; break;
5. print (y);
Graph Coverage

- Statement coverage
- Branch coverage
- Path coverage
- Data flow (def-use) coverage
Statement/Basic Block Coverage

if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}

Treat as one node because if one statement executes the other must also execute (code is a basic block)

Statement coverage: Cover every node of these graphs
Branch Coverage

But consider this if-then structure. For branch coverage can’t just cover all nodes, but must cover all edges – get to node 3 both after 2 and without executing 2!
Path Coverage

if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}

if (x < y)
{
    y = 0;
    x = x + 1;
}

How many paths through this code are there? Need one test case for each to get path coverage

To get statement and branch coverage, we only need two test cases:
1 2 4 5 6 and 1 3 4 6

Path coverage needs two more:
1 2 4 6
1 3 4 5 6
1 3 4 5 6

In general: exponential in the number of conditional branches!
Data Flow (Def-Use) Coverage

Annotate program with locations where variables are defined and used (very basic static analysis)

Def-use pair coverage requires executing all possible pairs of nodes where a variable is first defined and then used, without any intervening re-definitions

E.g., this path covers the pair where x is defined at 1 and used at 7: 1 2 3 5 6 7

But this path does NOT: 1 2 3 4 5 6 7

May be many pairs, some not actually executable

x = 3;
y = 3;
if (w) {
    x = y + 2;
}
if (z) {
    y = x – 2;
}
n = x + y

Def(x)
Use(y)

x = 3
Def(x)
y = 3
Def(y)
x = y + 2
Def(x)
Use(y)
y = x – 2
Def(y)
Use(x)

!w
w
!w
z
!z

Use(x) Use(y)
Logic Coverage

What if, instead of:

```c
if (x < y)
{
    y = 0;
    x = x + 1;
}
```

we have:

```c
if (((a>b) || G)) && (x < y))
{
    y = 0;
    x = x + 1;
}
```

Now, branch coverage will guarantee that we cover all the edges, but does not guarantee we will do so for all the different logical reasons.

We want to test the logic of the guard of the if statement.
# Active Clause Coverage

<table>
<thead>
<tr>
<th></th>
<th>(a &gt; b) or ( G )</th>
<th>(x &lt; y)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

With these values for \( G \) and \( (x < y) \), \( (a > b) \) determines the value of the predicate.

With these values for \( (a > b) \) and \( (x < y) \), \( G \) determines the value of the predicate.

With these values for \( (a > b) \) and \( G \), \( (x < y) \) determines the value of the predicate.
Input Domain Partitioning

- **Partition scheme** $q$ of domain $D$
- The partition $q$ defines a set of blocks, $B_q = b_1, b_2, \ldots, b_Q$
- The partition must satisfy two properties:
  1. blocks must be pairwise disjoint (no overlap)
  2. together the blocks cover the domain $D$ (complete)

Coverage then means using at least one input from each of $b_1, b_2, b_3, \ldots$