## Graph-Based Testing

- **Introduction**
- **Basic Concepts**
- **Control Flow Testing**
- **Data Flow Testing**
- **Summary**

## Motivation

- **Graph-based testing** first builds a graph model for the program under test, and then tries to cover certain elements in the graph model.
- **Graph** is one of the most widely used structures for abstraction.
  - Transportation network, social network, molecular structure, geographic modeling, etc.
- **Graph** is a well-defined, well-studied structure
  - Many algorithms have been reported that allow for easy manipulation of graphs.
**Major Steps**

- **Step 1: Build a graph model**
  - What information to be captured, and how to represent those information?

- **Step 2: Identify test requirements**
  - A test requirement is a structural entity in the graph model that must be covered during testing

- **Step 3: Select test paths to cover those requirements**

- **Step 4: Derive test data so that those test paths can be executed**

---

**Graph Models**

- **Control flow graph**: Captures information about how the control is transferred in a program.

- **Data flow graph**: Augments a CFG with data flow information

- **Dependency graph**: Captures the data/control dependencies among program statements

- **Cause-effect graph**: Modeling relationships among program input conditions, known as *causes*, and output conditions, known as *effects*
Graph-Based Testing

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

Graph

- A graph consists of a set of nodes and edges that connect pairs of nodes.
- Formally, a graph $G = \langle N, N_0, N_f, E \rangle$:
  - $N$: a set of nodes
  - $N_0 \subseteq N$: a set of initial nodes
  - $N_f \subseteq N$: a set of final nodes
  - $E \subseteq N \times N$: a set of edges
- In our context, $N$, $N_0$, and $N_f$ contain at least one node.
Example

\[N = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}\]
\[N_0 = \{n_0, n_1, n_2\}\]
\[N_f = \{n_7, n_8, n_9\}\]
\[E = \{(n_0, n_3), (n_0, n_4), (n_1, n_4), (n_1, n_5), \ldots\}\]

Path, Subpath, Test Path

- A path is a sequence \([n_1, n_2, \ldots, n_M]\) of nodes, where each pair of adjacent nodes \((n_i, n_{i+1})\) is an edge.
  - The length of a path refers to the number of edges in the path

- A subpath of a path \(p\) is a subsequence of \(p\), possibly \(p\) itself.

- A test path is a path, possibly of length zero, that starts at an initial node, and ends at a final node.
  - Represents a path that is executed during a test run
Reachability

- A node $n$ is **syntactically reachable** from node $n'$ if there exists a path from $n'$ to $n$.
- A node $n$ is **semantically reachable** from node $n'$ if it is possible to execute a path from $n'$ to $n$ with some input.
- $reach(n)$: the set of nodes and edges that can be syntactically reached from node $n$.

**Example**

```
p1 = [n_0, n_3, n_7]
p2 = [n_1, n_4, n_8, n_5, n_1]
p3 = [n_4, n_8, n_5]
reach(n_0) = ?
reach(n_6) = ?
```
Software Testing and Maintenance

Visit & Tour

- A test path $p$ is said to **visit** a node $n$ (or an edge $e$) if node $n$ (or edge $e$) is in path $p$.
- A test path $p$ is said to **tour** a path $q$ if $q$ is a subpath of $p$. 
### Test Case vs Test Path

- **Test Case**: A test case is typically a specific input or combination of inputs designed to test a particular aspect of the software. It is usually a single, complete test scenario.
- **Test Path**: A test path is a sequence of nodes (or states) in a control flow graph where all nodes are connected by edges, representing the flow of control through the software.

**Diagram:**

- Node $n_1$: $a > b$
- Node $n_2$: $a = b$
- Node $n_3$: $a < b$

**Examples:**

- $t_1$: $(a = 0, b = 1) \Rightarrow p_1 = [n_0, n_3, n_3, n_2]$
- $t_2$: $(a = 1, b = 1) \Rightarrow p_2 = [n_0, n_3, n_2]$
- $t_3$: $(a = 2, b = 1) \Rightarrow p_3 = [n_0, n_2]$

### Graph-Based Testing

- **Introduction**
- **Basic Concepts**
- **Control Flow Testing**
- **Data Flow Testing**
- **Summary**
**Basic Block**

- A *basic block*, or simply a *block*, is a sequence of consecutive statements with a single *entry* and a single *exit* point.

- Control always enters a basic block at its *entry* point, and exits from its *exit* point.
  - No entry, halt, or exit inside a basic block

- If a *basic block* contains a single statement, then the *entry* and *exit* points coincide.

---

**Example**

```plaintext
begin
  int x, y, power;
  float z;
  input (x, y);
  if (y < 0)
    power = -y;
  else
    power = y;
  z = 1;
  while (power != 0) {
    z = z * x;
    power = power - 1;
  }
  if (y < 0)
    z = 1/z;
  output (z);
end
```

<table>
<thead>
<tr>
<th>Block</th>
<th>Lines</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4, 5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>11, 12</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
Function Calls

- Should a function call be treated like a regular statement or as a separate block of its own?

Control Flow Graph

- A control flow graph is a graph with two distinguished nodes, start and end.
  - Node start has no incoming edges, and node end has no outgoing edges.
  - Every node can be reached from start, and can reach end.

- In a CFG, a node is typically a basic block, and an edge indicates the flow of control from one block to another.
**Example**

![Diagram](attachment:image.png)

**Node Coverage**

- A test set $T$ satisfies **Node Coverage** on graph $G$ if and only if for every **syntactically reachable node $n$** in $N$, there is some path $p$ in $\text{path}(T)$ such that $p$ visits $n$.
  - $\text{path}(T)$: the set of paths that are exercised by the execution of $T$

- In other words, the set $\text{TR}$ of test requirements for **Node Coverage** contains each **reachable node in $G$**.
**Edge Coverage**

- The TR for **Edge Coverage** contains each reachable path of length up to 1, inclusive, in a graph $G$.
- Note that **Edge Coverage** subsumes **Node Coverage**.

**Node vs Edge Coverage**

![Diagram](image)
**Edge-Pair Coverage**

- The TR for *Edge-Pair Coverage* contains each reachable path of length up to 2, inclusive, in a graph $G$.
- This definition can be easily extended to paths of any length, although possibly with diminishing returns.

---

**Edge-Pair vs Edge Coverage**

[Diagram showing a graph with nodes labeled $n_1, n_2, n_3, n_4, n_5, n_6$ and edges labeled $a, b, c, d, e, f, g, h$.]
**Complete Path Coverage**

- The TR for Complete Path Coverage contain all paths in a graph.

How many paths do we need to cover in the above graph?

**Simple & Prime Path**

- A path is **simple** if no node appears more than once in the path, with the exception that the first and last nodes may be identical.

- A path is a **prime** path if it is a simple path, and it does not appear as a proper subpath of any other simple path.
**Prime Path Coverage**

- The TR for **Prime Path Coverage** contains every prime path in a graph.

**Example**

```
Prime paths = \{[n0, n1, n2], [n0, n1, n3, n4], [n1, n3, n4, n1],
[n3, n4, n1, n3], [n4, n1, n3, n4], [n3, n4, n1, n2]\}
Path (t1) = [n0, n1, n2]
Path (t2) = [n0, n1, n3, n4, n1, n3, n4, n1, n2]
T = \{t1, t2\}
```
**Infeasible Test Requirements**

- The notion of “tour” is rather strict.

Let \( q = [a, b, d] \), and \( p = [S_0, a, b, c, d, S_f] \).

Does path \( p \) tour path \( q \)?

**Sidetrips/Detours**

- **Tour**: Test path \( p \) is said to tour path \( q \) if and only if \( q \) is a subpath of \( p \).

- **Tour with sidetrips**: Test path \( p \) is said to tour path \( q \) with sidetrips if and only if every edge in \( q \) is also in \( p \) in the same order.

- **Tour with detours**: Test path \( p \) is said to tour path \( q \) with detours if and only if every node in \( q \) is also in \( p \) in the same order.
**Example**

```
S_0  1 → a  2 → b  5 → d  6 → S_f
     3  → c
```

```
S_0  1 → a  2 → b  5 → d  6 → S_f
     3  → c
```

**Best Effort Touring**

- If a test requirement can be met without a sidetrip (or detour), then it should be done so.
- In other words, sidetrips or detours should be allowed only if necessary.
Computing Prime Paths

- Step 1: Find all the simple paths
  - Find all simple paths of length 0, extend them to length 1, and then to length 2, and so on
- Step 2: Select those that are maximal

Example
### Example – Simple Paths (2)

<table>
<thead>
<tr>
<th>len = 0</th>
<th></th>
<th>len = 1</th>
<th></th>
<th>len = 2</th>
<th></th>
<th>len = 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. [0]</td>
<td>8. [0, 1]</td>
<td>17. [0, 1, 2]</td>
<td>25. [0, 1, 2, 3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. [1]</td>
<td>9. [0, 4]</td>
<td>18. [0, 1, 5]</td>
<td>26. [0, 1, 5, 6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16. [5, 6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**len = 4**

- 32. [2, 3, 1, 5, 6]!

### Example – Prime Paths

- 14. [4, 4]*
- 19. [0, 4, 6]!
- 25. [0, 1, 2, 3]!
- 26. [0, 1, 5, 6]!
- 27. [1, 2, 3, 1]*
- 28. [2, 3, 1, 2]*
- 30. [3, 1, 2, 3]*
- 32. [2, 3, 1, 5, 6]!
Example – Test Paths

- Start with the longest prime paths and extend them to the start and end nodes of the graph

1) [0, 1, 2, 3, 1, 5, 6]
2) [0, 1, 2, 3, 1, 2, 3, 1, 5, 6]
3) [0, 1, 5, 6]
4) [0, 4, 6]
5) [0, 4, 4, 6]

Graph-Based Testing

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary
**Definition/Use**

- **A definition** is a location where a value for a variable is stored into memory.
  - Assignment, input, parameter passing, etc.

- **A use** is a location where a variable’s value is accessed.
  - p-use: a use that occurs in a predicate expression, i.e., an expression used as a condition in a branch statement
  - c-use: a use that occurs in an expression that is used to perform certain computation

**Data Flow Graph**

- A **data flow graph (DFG)** captures the flow of data in a program.

- To build a DFG, we first build a **CFG** and then annotate each node \( n \) in the CFG with the following two sets:
  - \( \text{def}(n) \): the set of variables defined in node \( n \)
  - \( \text{use}(n) \): the set of variables used in node \( n \)
**Example (1)**

```plaintext
1. begin
2. float x, y, z = 0.0;
3. int count;
4. input (x, y, count);
5. do {
6.   if (x <= 0) {
7.     if (y >= 0) {
8.       z = y * z + 1;
9.     }
10.   } else {
11.     z = 1/x;
12.   }
13.   y = x * y + z;
14.   count = count – 1;
15. } while (count > 0)
16. output (z);
17. end
```

**Example (2)**

```
def={x, y, z, count}
def={} use = {x}
def={} use = {y}
def={} use = {y, z}
def={} use = {count, y, z}
def={} use = {z}
```

<table>
<thead>
<tr>
<th>Node</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>2</td>
<td>5, 6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8, 9, 10</td>
</tr>
<tr>
<td>5</td>
<td>11, 12, 13</td>
</tr>
<tr>
<td>6</td>
<td>14, 15, 16</td>
</tr>
<tr>
<td>7</td>
<td>17, 18</td>
</tr>
</tbody>
</table>
**DU-pair & DU-path**

- A du-pair is a pair of locations \((i, j)\) such that a variable \(v\) is defined in \(i\) and used in \(j\).

- Suppose that variable \(v\) is defined at node \(i\), and there is a use of \(v\) at node \(j\). A path \(p = (i, n_1, n_2, ..., n_k, j)\) is def-clear w.r.t. \(v\) if \(v\) is not defined along the subpath \(n_1, n_2, ..., n_k\).

- A definition of a variable \(v\) reaches a use of \(v\) if there is a def-clear path from the definition to the use w.r.t. \(v\).

- A du-path for a variable \(v\) is a simple path from a definition of \(v\) to a use of \(v\) that is def-clear w.r.t. \(v\).

**Example**

- Consider the previous example:
  - Path \(p = (1, 2, 5, 6)\) is def-clear w.r.t variables \(x, y\) and \(\text{count}\), but is not def-clear w.r.t. variable \(z\).
  - Path \(q = (6, 2, 5, 6)\) is def-clear w.r.t variables \(\text{count}\) and \(y\).
  - Path \(r = (1, 2, 3, 4)\) is def-clear w.r.t variables \(y\) and \(z\).
Notations

- **Def-path set** $\text{du}(n, v)$: the set of *du-paths* w.r.t variable $v$ that start at node $n$.
- **Def-pair set** $\text{du}(n, n', v)$: the set of *du-paths* w.r.t variable $v$ that start at node $n$ and end at node $n'$.
- Note that $\text{du}(n, v) = \bigcup_{n'} \text{du}(n, n', v)$.

All-Defs Coverage

- For each def-path set $S = \text{du}(n, v)$, the TR for *All-Defs Coverage* contains at least one path in $S$.
- Informally, for each def, we need to tour at least one path to at least one use.
**All-Uses Coverage**

- For each def-pair set $S = du(n, n', v)$, the TR for All-Uses Coverage contains at least one path in $S$.
- Informally, it requires us to tour at least one path for every def-use pair.

**All-DU-Paths Coverage**

- For each def-pair set $S = du(n, n', v)$, the TR for All-DU-Paths Coverage contains every path in $S$.
- Informally, this requires to tour every du-path.
**Example**

```
   n_0
  /   |
 n_1   n_2
   \
    |
 n_3
```

- $\text{def}(0) = \{x\}$
- $\text{use}(4) = \{x\}$
- $\text{use}(5) = \{x\}$

<table>
<thead>
<tr>
<th>all-defs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1-3-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>all-uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1-3-4</td>
</tr>
<tr>
<td>0-1-3-5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>all-du-paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1-3-4</td>
</tr>
<tr>
<td>0-1-3-5</td>
</tr>
<tr>
<td>0-2-3-4</td>
</tr>
<tr>
<td>0-2-3-5</td>
</tr>
</tbody>
</table>

**Why data flow?**

- Consider the previous example. Assume that there is a fault in line 14, which is supposed to be $y = x + y + z$.

- Does the following test set satisfy edge coverage? Can the test set detect the above fault?

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>-2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>t2</td>
<td>-2</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>t3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>t4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Graph-Based Testing

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

Subsumption Hierarchy

- Complete Path Coverage
  - Prime Path Coverage
    - All-du-paths Coverage
    - All-Uses Coverage
    - All-Defs Coverage
  - Edge-pair Coverage
    - Edge Coverage
    - Node Coverage
**Recap**

- **Graph** provides a good basis for systematic test selection.

- **Control flow testing** focuses on the transfer of control, while **data flow testing** focuses on the definitions of data and their subsequent use.

- **Control flow coverage** is defined in terms of **nodes**, **edges**, and **paths**; **data flow coverage** is defined in terms of **def**, **use**, and **du-path**.