Graph Coverage for Source Code

CS 4501 / 6501
Software Testing

[Ammann and Offutt, “Introduction to Software Testing,” Ch. 7.3]
Structures for Criteria-Based Testing

Four structures for modeling software

Input space

Graph
- Source
- Design
- Specs
- Use cases

Logic
- Source
- Specs
- FSMs
- DNF

Syntax
- Source
- Models
- Integration
- Inputs

---R
R--R
RI-R
RIPR
Overview

- Graph coverage criteria are widely used on source code
- Define graph, then apply coverage criterion
- **Control flow graph (CFG):** the most common graph for source code
- Node coverage: execute every statement
- Edge coverage: execute every branch
- Data flow coverage: augment the CFG with
  - defs: statements that assign values to variables
  - uses: statements that use variables
Control Flow Graph (CFG)

- Represent the control flow of a piece of source code

  - **Nodes** represent basic blocks
    - **Basic blocks** represent sequences of instructions / statements that always execute together in sequence

  - **Edges** represent control flow (branch) between basic blocks
    - Transfer of control

  - **Initial nodes** correspond to a method’s entry points

  - **Final nodes** correspond to a method’s exit points
    - Return or throw in Java

  - **Decision nodes** represent choices in control flow
    - if or switch-case blocks or condition for loops in Java

- Can be annotated with extra information such as branch predicates, defs, and uses
Example: CFG for *if-else*

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

- **Basic blocks (nodes)**
  1: if (x < y)
  2: y=0; x = x+1;
  3: x = y;

- **Entry node**
  1

- **Decision nodes**
  1

- **Junction nodes**
  4

- **Exit nodes**
  4

- **Control flow (edges)**
  1 $\rightarrow$ 2
  1 $\rightarrow$ 3
  2 $\rightarrow$ 4
  3 $\rightarrow$ 4
Example: CFG for *If* without *else*

- Basic blocks (nodes)
  1: if \((x < y)\)
  2: \(y = 0; \ x = x + 1;\)

- Entry node
  1

- Decision nodes
  1

- Junction nodes
  3

- Exit nodes
  3

- Control flow (edges)
  1 \(\rightarrow\) 2
  1 \(\rightarrow\) 3
  2 \(\rightarrow\) 3

```c
if (x < y) {
    y = 0;
    x = x + 1;
}
```
Example: CFG for If with return

```c
if (x < y) {
    return;
}
print(x); return;
```

• Basic blocks (nodes)
  1: if (x < y)
  2: return;
  3: print(x); return;

• Entry node
  1

• Decision nodes
  1

• Junction nodes
  -

• Exit nodes
  2, 3

• Control flow (edges)
  1 → 2
  1 → 3
Loops

- Loops require extra nodes ("dummy" node)
  - Not directly derived from program statements

- Looping structures: while loop, for loop, do-while loop

- Common mistake
  - Try to have the edge go to the entry node
Example: CFG for a `while` loop

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

- **Basic blocks (nodes)**
  1: \(x = 0;\)
  2: \(\text{while}(x < y)\)
  3: \(y = f(x,y); x = x+1;\)

- **Control flow (edges)**
  1 \(\rightarrow\) 2
  2 \(\rightarrow\) 3
  2 \(\rightarrow\) 4
  3 \(\rightarrow\) 2

- **Entry node**
  1

- **Decision nodes**
  2

- **Junction nodes**
  -

- **Exit nodes**
  4
Example: CFG for a for loop

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

- Basic blocks (nodes)
  1: x = 0;
  2: x < y
  3: y = f(x,y);
  4: x++;

- Entry node
  1

- Decision nodes
  2

- Junction nodes
  -

- Exit nodes
  4

- Control flow (edges)
  1 → 2, 2 → 3, 2 → 5, 3 → 4, 4 → 2

Diagram:

- Entry node 1: x = 0
- Decision node 2: x < y
- Decision node 3: y = f(x,y)
- Decision node 4: x++
- Exit node 5: x >= y

- Node 1: implicitly initializes loop
- Node 2: implicitly increments loop
Example: CFG for a do-while loop

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

- **Basic blocks (nodes)**
  1: `x = 0; do`
  2: `y = f(x,y); x = x + 1;`
  3: `println(y);`

- **Control flow (edges)**
  1 → 2
  2 → 2
  2 → 3

- **Entry node**
  1

- **Decision nodes**
  2

- **Junction nodes**
  -

- **Exit nodes**
  3
Example: CFG for a loop with `break` and `continue`

```plaintext
x = 0;
while (x < y) {
    y = f(x, y);
    if (y==0)
        break;
    else if (y < 0)
    {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print(x);
```

---

Diagram of the control flow graph (CFG) corresponding to the above code snippet.
Example: CFG for \((\text{switch})\) case

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

Cases without break?
- Fall through to the next case
Example: CFG for Exceptions (try-catch)

```java
try
{
    s = br.readLine();
    if (s.length() > 96)
        throw new Exception("too long");
    if (s.length() == 0)
        throw new Exception("too short");
} catch (IOException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.getMessage();
}
return (s);
```
Exercise

```java
public static int numberOccurrences(char[] v, char c) {
    if (v == null)
        throw new NullPointerException();
    int n = 0;
    for (int i=0; i<v.length; i++)
    {
        if (v[i] == c)
            n++;
    }
    return n;
}
```

- **Basic blocks (nodes)**
  1: if (v == null)
  2: throw .. NPE;
  3: n=0; i=0;
  4: i < v.length;
  5: if (v[i] == c)
  6: n++;
  7: i++;
  8: return n;

- **Control flow (edges)**
  1→2, 1→3
  3→4
  4→5, 4→8
  5→6, 5→7
  6→7
  7→4

- **Entry node**
  1

- **Exit nodes**
  2, 8
CFG for `numberOccurrences()`

1. `v == null`
2. `v != null`
   - `throw ... NPE`
   - `n=0`, `i=0`
   - `i >= v.length` (branch to 8)
   - `i < v.length` (branch to 5)
3. `v !== null`
4. `return n`
5. `v[i] == c`
6. `v[i] !== c`
7. `i++`
8. `return n`

Diagram nodes:
- Node 1: Entry
- Node 2: Check for null
- Node 3: Check for non-null
- Node 4: Initialize `n=0`, `i=0`
- Node 5: Check if `i < v.length`
- Node 6: Increment `n` if `v[i] == c`
- Node 7: Increment `i` if `v[i] != c`
- Node 8: Return `n`
Applying Node Coverage

Test requirements
TR = \{1,2,3,4,5,6,7,8\}

Test paths
- t1 = [1,2]
- t2 = [1,3,4,5,6,7,4,8]

NC satisfied by \{t1, t2\}

Test case values (v,c)
- t1 = (null, ‘a’), expected NPE
- t2 = (\{‘a’\}, ‘a’), expected 1
Applying Edge Coverage

Test requirements

$$TR = \{(1,2), (1,3), (3,4), (4,5), (4,8), (5,6), (5,7), (6,7), (7,4)\}$$

Test paths

- $$t1 = [1,2]$$
- $$t2 = [1,3,4,5,6,7,4,8]$$
- $$t3 = [1,3,4,5,7,4,8]$$

EC satisfied by $$\{t1, t2, t3\}$$

Test case values $$(v,c)$$

- $$t1 = (null, 'a'),$$ expected NPE
- $$t2 = ({'a'}, 'a'),$$ expected 1
- $$t3 = ({'x'}, 'a'),$$ expected 0
Applying Edge-Pair Coverage

Test requirements
\[ TR = \{(1,2), (1,3,4), (3,4,8), (3,4,5), (4,5,6), (4,5,7), (5,6,7), (5,7,4), (6,7,4), (7,4,5), (7,4,8)\} \]

Test paths
\[ t1 = [1,2] \]
\[ t2 = [1,3,4,8] \]
\[ t3 = [1,3,4,5,7,4,5,6,7,4,8] \]

EPC satisfied by \{t1, t2, t3\}

Test case values \((v,c)\)
\[ t1 = (null, 'a'), \text{expected NPE} \]
\[ t2 = (\{} , 'a'), \text{expected 0} \]
\[ t3 = (\{'x','a'\} , 'a'), \text{expected 1} \]
Applying Prime Path Coverage

Deriving prime paths

- Enumerate all simple paths of length 0, 1, 2, 3, ... until no more simple paths are found

- Pick the prime paths among all derived simple paths
Applying Prime Path Coverage

v == null
v != null
return n
i++

! – cannot be extended
* – is a cycle
Applying Prime Path Coverage

List test paths (including test case values and expected value) that satisfy PPC

<table>
<thead>
<tr>
<th>Prime path</th>
<th>Covered by</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2]</td>
<td>t1</td>
</tr>
<tr>
<td>[1,3,4,8]</td>
<td>t2, t4, t5</td>
</tr>
<tr>
<td>[1,3,4,5,6,7]</td>
<td>t3</td>
</tr>
<tr>
<td>[1,3,4,5,7]</td>
<td>t3, t4</td>
</tr>
<tr>
<td>[4,5,7,4]</td>
<td>t3, t4, t5</td>
</tr>
<tr>
<td>[4,5,6,7,4]</td>
<td>t3</td>
</tr>
<tr>
<td>[5,7,4,5]</td>
<td>t3</td>
</tr>
<tr>
<td>[5,7,4,8]</td>
<td>t4</td>
</tr>
<tr>
<td>[5,6,7,4,5]</td>
<td>t4, t5</td>
</tr>
<tr>
<td>[5,6,7,4,8]</td>
<td>t3, t5</td>
</tr>
<tr>
<td>[6,7,4,5,6]</td>
<td>t5</td>
</tr>
<tr>
<td>[7,4,5,7]</td>
<td>t4</td>
</tr>
<tr>
<td>[7,4,5,6,7]</td>
<td>t3, t5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test paths</th>
<th>Test case values (v,c)</th>
<th>Expected values</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>[1,2]</td>
<td>(null, ‘a’)</td>
</tr>
<tr>
<td>t2</td>
<td>[1,3,4,8]</td>
<td>({} , ‘a’)</td>
</tr>
<tr>
<td>t3</td>
<td>[1,3,4,5,7,5,6,7,4,8]</td>
<td>({‘x’, ‘a’}, ‘a’)</td>
</tr>
<tr>
<td>t4</td>
<td>[1,3,4,5,6,7,4,5,7,4,8]</td>
<td>(‘a’, ‘x’), ‘a’)</td>
</tr>
<tr>
<td>t5</td>
<td>[1,3,4,5,6,7,4,5,6,7,4,8]</td>
<td>(‘a’, ‘a’), ‘a’)</td>
</tr>
</tbody>
</table>

PPC satisfied by \{t1, t2, t3, t4, t5\}
Applying Data Flow Coverage

Deriving test requirements
- List all du-pairs
- Based on du-pairs, derive du-paths
- Must be def-clear paths
- All Defs Coverage (ADC)
  - For each def, at least one use must be reached
- All Uses Coverage (AUC)
  - For each def, all uses must be reached
- All DU-Paths Coverage (ADUPC)
  - For each def-use pair, all paths between defs and uses must be covered

Diagram:

1. Def(1) = \{v, c\}
   - Use(1, 2) = \{v\}
   - Use(1, 3) = \{v\}
2. Def(3) = \{n, i\}
3. Use(4, 8) = \{i, v\}
4. Use(4, 5) = \{i, v\}
5. Use(6) = \{n\}
   - Def(6) = \{n\}
6. Use(5, 6) = \{v, i, c\}
7. Use(5, 7) = \{v, i, c\}
   - Def(7) = \{i\}
8. Use(7) = \{i\}
   - Def(7) = \{i\}

v and c are forwarded parameters.
**DU-Pairs → DU-Paths**

<table>
<thead>
<tr>
<th>DU Pairs</th>
<th>DU Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, (1, 2)]</td>
<td>[1,3]</td>
</tr>
<tr>
<td>[1, (1, 3)]</td>
<td>[1,2]</td>
</tr>
<tr>
<td>[1, (4, 8)]</td>
<td>[1,3,4,8]</td>
</tr>
<tr>
<td>[1, (4, 5)]</td>
<td>[1,3,4,5]</td>
</tr>
<tr>
<td>[1, (5, 6)]</td>
<td>[1,3,4,5,7]</td>
</tr>
<tr>
<td>[1, (5, 7)]</td>
<td>[1,3,4,5,7]</td>
</tr>
<tr>
<td>[1, (5, 7)]</td>
<td>[1,3,4,5,6]</td>
</tr>
<tr>
<td>[1, (5, 6)]</td>
<td>[1,3,4,5,6]</td>
</tr>
<tr>
<td>[3,8]</td>
<td>[3,4,8]</td>
</tr>
<tr>
<td>[3,6]</td>
<td>[3,4,5,6]</td>
</tr>
<tr>
<td>[6,8]</td>
<td>[6,7,4,8]</td>
</tr>
<tr>
<td>[6,6]</td>
<td>[6,7,4,5,6]</td>
</tr>
<tr>
<td>[3,7]</td>
<td>[3,4,5]</td>
</tr>
<tr>
<td>[7,7]</td>
<td>[3,4,8]</td>
</tr>
<tr>
<td>[3, (4, 8)]</td>
<td>[3,4,5,6]</td>
</tr>
<tr>
<td>[3, (4, 5)]</td>
<td>[3,4,5,7]</td>
</tr>
<tr>
<td>[3, (5, 6)]</td>
<td>[3,4,5,6,7]</td>
</tr>
<tr>
<td>[3, (5, 7)]</td>
<td>[7,4,5]</td>
</tr>
<tr>
<td>[7, (4, 8)]</td>
<td>[7,4,8]</td>
</tr>
<tr>
<td>[7, (4, 5)]</td>
<td>[7,4,5,6]</td>
</tr>
<tr>
<td>[7, (5, 6)]</td>
<td>[7,4,5,7]</td>
</tr>
<tr>
<td>[7, (5, 7)]</td>
<td>[7,4,5,6,7]</td>
</tr>
<tr>
<td>[7, (5, 7)]</td>
<td>[7,4,5,6,7]</td>
</tr>
</tbody>
</table>

Variable v

Variable c

Variable n

Variable i

**defs after uses, these are valid DU pairs**
ADC: DU-Paths → Test Paths

<table>
<thead>
<tr>
<th>DU Paths</th>
<th>Test paths that satisfy AllDefs Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 3]</td>
<td></td>
</tr>
<tr>
<td>[1, 2]</td>
<td></td>
</tr>
<tr>
<td>[1, 3, 4, 8]</td>
<td>v: [1, 3, 4, 8]</td>
</tr>
<tr>
<td>[1, 3, 4, 5]</td>
<td>c: [1, 3, 4, 5, 6, 7, 4, 8]</td>
</tr>
<tr>
<td>[1, 3, 4, 5, 7]</td>
<td></td>
</tr>
<tr>
<td>[1, 3, 4, 5, 6]</td>
<td></td>
</tr>
<tr>
<td>[3, 4, 8]</td>
<td>n: [1, 3, 4, 5, 6, 7, 4, 8]</td>
</tr>
<tr>
<td>[3, 4, 5, 6]</td>
<td></td>
</tr>
<tr>
<td>[6, 7, 4, 8]</td>
<td>i: [1, 3, 4, 5, 7, 4, 8]</td>
</tr>
<tr>
<td>[6, 7, 4, 5, 6]</td>
<td></td>
</tr>
<tr>
<td>[3, 4, 5]</td>
<td></td>
</tr>
<tr>
<td>[3, 4, 8]</td>
<td></td>
</tr>
<tr>
<td>[3, 4, 5, 6]</td>
<td></td>
</tr>
<tr>
<td>[3, 4, 5, 7]</td>
<td></td>
</tr>
<tr>
<td>[3, 4, 5, 6, 7]</td>
<td></td>
</tr>
<tr>
<td>[7, 4, 5]</td>
<td></td>
</tr>
<tr>
<td>[7, 4, 5, 6]</td>
<td></td>
</tr>
<tr>
<td>[7, 4, 5, 7]</td>
<td></td>
</tr>
<tr>
<td>[7, 4, 5, 6, 7]</td>
<td></td>
</tr>
</tbody>
</table>
AUC: DU-Paths → Test Paths

Test paths that satisfy All Uses Coverage

<table>
<thead>
<tr>
<th>Variable</th>
<th>All Use Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>[1,2]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,8]</td>
</tr>
<tr>
<td>c</td>
<td>[1,3,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,8]</td>
</tr>
<tr>
<td>n</td>
<td>[1,3,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,5,6,7,4,8]</td>
</tr>
<tr>
<td>i</td>
<td>[1,3,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,5,6,7,4,8]</td>
</tr>
</tbody>
</table>
# ADUPC: DU-Paths → Test Paths

## Test paths that satisfy All DU-Paths Coverage

<table>
<thead>
<tr>
<th>Variable</th>
<th>All DU Path Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>[1,3,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,2]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,8]</td>
</tr>
<tr>
<td>c</td>
<td>[1,3,4,5,6,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,8]</td>
</tr>
<tr>
<td>n</td>
<td>[1,3,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,5,6,7,4,8]</td>
</tr>
<tr>
<td>i</td>
<td>[1,3,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,6,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,8]</td>
</tr>
<tr>
<td></td>
<td>[1,3,4,5,7,4,5,6,7,4,8]</td>
</tr>
</tbody>
</table>
Summary

- A common application of graph coverage criteria is to program source – control flow graph (CFG)

- Applying graph coverage criteria to control flow graphs is relatively straightforward

- A few decisions must be made to translate control structures into the graph

- We use basic blocks when assigning program statements to nodes while some tools assign each statement to a unique node.
  - Coverage is the same, although the bookkeeping will differ