Lecture 09: White-Box Testing (advanced) – Data-Flow Testing and Mutation Testing

Dietmar Pfahl

email: dietmar.pfahl@ut.ee
Lectures

• Lecture 1 (13.02) – Introduction to Software Testing
• Lecture 2 (20.02) – Basic Black-Box Testing Techniques: Boundary Value Analysis & Equivalence Class Partitioning
• Lecture 3 (27.02) – BBT advanced: C/E-Graphing & Combinatorial Testing
• Lecture 4 (05.03) – Basic White-Box Testing Techniques: Instruction & Control-Flow Coverage
• Lecture 5 (12.03) – BBT adv.: State-Transition, Random, Metamorphic Testing
• Lecture 6 (19.03) – Test Levels, Test Tools, Test Automation
• Lecture 7 (26.03) – BBT adv.: Exploratory Testing, Behavior Testing
• Lecture 8 (02.04) – BBT adv.: GUI / Visual Testing, Security, Usability, A/B Testing
• Lecture 9 (09.04) – WBT adv.: Data-Flow Testing, Mutation Testing
• Lecture 10 (16.04) – WBT adv.: Symbolic Execution, Static Code Analysis, Review
• Lecture 11 (23.04) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
• Lectures 12+13 (30.04 + 07.05) – Industry Guest Lectures (see course wiki)
• Lecture 14 (14.05) – Exam Preparation
Structure of Lecture 9

• Data Flow-Testing
• Mutation Testing
• Lab 9
White-Box Testing Techniques

- Control-Flow Testing
- Data-Flow Testing
- Mutation Testing
- Symbolic Execution
- Static Code Analysis
- Reviews

Lecture 10
White-Box Testing Techniques

- Control-Flow Testing
- Data-Flow Testing
- Mutation Testing
- Symbolic Execution
- Static Code Analysis
- Reviews

Lecture 10
Data Flow Testing – Motivation

• Middle ground in structural testing
  – Node (=statement) and edge (=branch) coverage don’t test interactions between statements
  – All-path testing is infeasible
  – Need a coverage criterion that is stronger than branch coverage but feasible

• Intuition: Statements interact through data flow
  – Value computed in one statement, used in another
  – Bad value computation is revealed only when used
Data Flow Testing

• Identifies paths in the program that go
  – from the **assignment** of a value to a variable to
  – the **use** of such variable,
  to make sure that the variable is properly used.

X←14; ..... Y← X-3;

**Goal:** Try to ensure that values are computed and used correctly
Data Flow Criteria

Source: Ammann & Offutt: Introduction to Software Testing, Edition 2 (Ch 06)

• Def (definition): A location where a value for a variable is stored into memory
• Use: A location where a variable’s value is accessed

X = 42

\[ Z = X \times 2 \]
\[ Z = X - 8 \]

\[ \text{Defs: def(1)} = \{X\} \]
\[ \text{def(5)} = \{Z\} \]
\[ \text{def(6)} = \{Z\} \]

\[ \text{Uses: use(5)} = \{X\} \]
\[ \text{use(6)} = \{X\} \]

The values given in def should reach at least one, some, or all possible uses
DU Pairs and DU Paths

Source: Ammann & Offutt: Introduction to Software Testing, Edition 2 (Ch 06)

• def(n): The set of variables that are defined in node n
• use(n): The set of variables that are used in node n

• DU pair: A pair of locations \((l_i, l_j)\) such that a variable \(v\) is defined at \(l_i\) and used at \(l_j\)

• Def-clear: A path from \(l_i\) to \(l_j\) is def-clear with respect to variable \(v\) if \(v\) is not given another value on any of the nodes in the path

• du-path: A simple sub-path that is def-clear with respect to \(v\) from a def of \(v\) to a use of \(v\)

• du\((n_i, n_j, v)\) – the set of du-paths from \(n_i\) to \(n_j\)
• du\((n_i, v)\) – the set of du-paths that start at \(n_i\)
Covering DU-Paths

Source: Ammann & Offutt: Introduction to Software Testing, Edition 2 (Ch 06)

• A test path $p$ du-covers sub-path $d$ with respect to $v$ if $p$ covers $d$ and the sub-path taken is def-clear with respect to $v$

• Three criteria:
  – Use every def (at least once)
  – Get to every use (of every def)
  – Cover all du-paths (from all defs to all uses)
Data Flow Testing – Criteria

- **All definitions paths**
  - requires that at least one path from each definition of a variable to one of its uses is executed

- **All uses paths**
  - requires that for each definition-use pair of a variable at least one simple definition-clear path is executed

- **All def-use paths**
  - requires that each simple (i.e., traversing a loop at most once) definition-clear path from a definition of a variable to its use is executed

- ...
Data Flow Testing Example

Source: Ammann & Offutt: Introduction to Software Testing, Edition 2 (Ch 06)

\[ X = 42 \]
\[ Z = X \times 2 \]
\[ Z = X - 8 \]

**All-defs for X**

[ 1, 2, 4, 5 ]

**All-uses for X**

[ 1, 2, 4, 5 ]

[ 1, 2, 4, 6 ]

**All-du-paths for X**

[ 1, 2, 4, 5 ]

[ 1, 3, 4, 5 ]

[ 1, 2, 4, 6 ]

or

[ 1, 3, 4, 6 ]
Data Flow Testing – Defs & Uses

- **Def** – assigned or changed
- **Uses** – utilized (not changed)
  - **C-use** (Computation) e.g. right-hand side of an assignment, an index of an array, parameter of a function.
  - **P-use** (Predicate) branching the execution flow, e.g. in an if statement, while statement, for statement.

```plaintext
[0] bool AccClient(int age; 
gtype gender)
[1] bool accept = false
[2] if (gender==female & age<85) 
[3]    accept = true;
[4] if (gender==male & age<80) 
[5]    accept = true;
[6] return accept
```

- Def(0/1) = \{age, gender, accept\}
- P-use(2) = \{age, gender\}
- P-use(4) = \{age, gender\}
- C-use(6) = \{accept\}
- Def(3) = \{accept\}
- Def(5) = \{accept\}
Data Flow Testing – Example

Considering age, what are the DU pairs?

```cpp
[0] bool AccClient(int age;
gtype gender) {
[1] bool accept = false;
[2] if (gender==female & age<85)
[3]   accept = true;
[4] if (gender==male & age<80)
[5]   accept = true;
[6] return accept; }
```
Data Flow Testing – Example

Considering **age**, there are two DU pairs:

(a) [0]-[2]
(b) [0]-[4]

Test case(s) for ‘all-defs’?

```c
[0] bool AccClient(int age;
           gtype gender) {
[1]   bool accept = false;
[2]   if (gender==female & age<85)
[3]       accept = true;
[4]   if (gender==male & age<80)
[5]       accept = true;
[6]   return accept; }
```
Data Flow Testing – Example

Considering age, there are two DU pairs:

(a)[0]-[2]
(b)[0]-[4]

Test case(s) for ‘all-defs’:
AccClient(*, *) -> *

→ covers: 0-1-2

(and: 0-1-2-x-4)
Data Flow Testing – Example

Considering age, there are two DU pairs:

(a)[0]-[2]
(b)[0]-[4]

Test case(s) for ‘all-uses’?

```c
[0] bool AccClient(int age;
    gtype gender) {
[1] bool accept = false;
[2] if (gender==female & age<85)
[3]    accept = true;
[4] if (gender==male & age<80)
[5]    accept = true;
[6] return accept; }
```
Data Flow Testing – Example

Considering age, there are two DU pairs:

(a)[0]-[2]
(b)[0]-[4]

Test case(s) for ‘all-uses’:

AccClient(\(*, *) -> *

covers: 0-1-2
and 0-1-2-(x)-4
Data Flow Testing – Example

Considering age, there are two DU pairs:

(a) [0]-[2]
(b) [0]-[4]

Test case for ‘all-def-uses’?

```c
[0] bool AccClient(int age;
            gtype gender) {
[1]   bool accept = false;
[2]   if (gender==female & age<85)
[3]       accept = true;
[4]   if (gender==male & age<80)
[5]       accept = true;
[6]   return accept }
```
Data Flow Testing – Example

Considering age, there are two DU pairs:
(a)[0]-[2]
(b)[0]-[4]

Test case for ‘all-def-uses’:
AccClient(f, 83) -> true
AccClient(f, 90) -> false

⇒ covers: 0-1-2, 0-1-2-3-4, and 0-1-2-4
Data Flow Testing – Example

Considering gender, what are the DU pairs and the associated def-use paths?

```cpp
[0] bool AccClient(int age;
               gtype gender) {
[1] bool accept = false;
[2] if (gender==female & age<85)
    [3]   accept = true;
[4] if (gender==male & age<80)
    [5]   accept = true;
[6] return accept; }
```
Data Flow Testing – Example

Considering gender, there are two DU pairs with three def-use paths:

(a)[0]-[2]: 0-1-2
(b)[0]-[4]: 0-1-2-4, 0-1-2-3-4

Test cases for ‘all-def-uses’?

```c
[0] bool AccClient(int age;
gtype gender) {
    [1] bool accept = false;
    [2] if (gender==female & age<85)
        accept = true;
    [3] if (gender==male & age<80)
        accept = true;
    [4] return accept;
}
```
Data Flow Testing – Example

Considering gender, there are two DU pairs with three def-use paths:

(a) [0]-[2]: 0-1-2
(b) [0]-[4]: 0-1-2-4, 0-1-2-3-4

Test cases needed to satisfy all-defs-paths criterion:
AccClient() is executed

Test cases for ‘all-defs’:
AccClient(*, *)-> *
⇒ covers, e.g., 0-1-2

[0] bool AccClient(int age; gtype gender) {
[1]   bool accept = false;
[2]   if (gender==female & age<85)
[3]       accept = true;
[4]   if (gender==male & age<80)
[5]       accept = true;
[6]   return accept; }

ltat.05.006 / lecture 09 / © dietmar pfahl 2020
Data Flow Testing – Example

Considering gender, there are two DU pairs with three def-use paths:

(a) [0]-[2]: 0-1-2
(b) [0]-[4]: 0-1-2-4, 0-1-2-3-4

Test cases needed to satisfy all-uses-paths criterion:
AccClient() is executed

g-> covers, e.g., \{0-1-2, 0-1-2-4\}
Data Flow Testing – Example

Considering gender, there are two DU pairs with three def-use paths:

(a) [0]-[2]: 0-1-2
(b) [0]-[4]: 0-1-2-4, 0-1-2-3-4

Test cases for ‘all-def-uses’?

```java
[0] bool AccClient(int age;
    gtype gender) {
[1] bool accept = false;
[2] if (gender==female & age<85)
[3]     accept = true;
[4] if (gender==male & age<80)
[5]     accept = true;
[6] return accept; }
```
Data Flow Testing – Example

Considering gender, there are two DU pairs with three def-use paths:

(a) [0]-[2]: 0-1-2
(b) [0]-[4]: 0-1-2-4, 0-1-2-3-4

Test case for ‘all-def-uses’:
AccClient(f, 83) -> true
AccClient(f, 90) -> false
covers {0-1-2, 0-1-2-3-4, 0-1-2-4}

```cpp
[0] bool AccClient(int age;
  gtype gender) {
[1] bool accept = false;
[2] if (gender==female & age<85)
  accept = true;
[3] if (gender==male & age<80)
  accept = true;
[6] return accept; }
```
Data Flow Testing – Example

Considering `accept`, what are the DU pairs?

```c
[0] bool AccClient(int age;
      gtype gender) {
[1] bool accept = false;
[2] if (gender==female & age<85)
[3]    accept = true;
[4] if (gender==male & age<80)
[5]    accept = true;
[6] return accept; }
```
Data Flow Testing – Example

Considering `accept`, there are three DU pairs:
(a)[1]-[6] (b)[3]-[6] (c)[5]-[6]
What are the associated def-use-paths?

```c
[0] bool AccClient(int age;
gtype gender) { 
[1] bool accept = false;
[2] if (gender==female & age<85)
[3]    accept = true;
[4] if (gender==male & age<80)
[5]    accept = true;
[6] return accept; }
```
Data Flow Testing – Example

Considering `accept`, there are three DU pairs:
(a) [1]-[6]  (b) [3]-[6]  (c) [5]-[6]

DU paths:
(a) 1-2-4-6
(b) 3-4-6
(c) 5-6
Data Flow Testing – Example

Considering `accept`, there are three DU pairs:
(a)[1]-[6] (b)[3]-[6] (c)[5]-[6]

Test cases for ‘all-defs’:
(a) `AccClient(*, 90)` -> false
(b) `AccClient(f, 83)` -> true
(c) `AccClient(m, 79)` -> true

```
[0] bool AccClient(int age;
    gtype gender) {
[1]    bool accept = false;
[2]    if (gender==female & age<85)
[3]        accept = true;
[4]    if (gender==male & age<80)
[5]        accept = true;
[6]    return accept; }
```
Data Flow Testing – Example

```c
[0] bool AccClient(int age;
    gtype gender) {
[1]   bool accept = false;
[2]   if (gender==female & age<85)
[3]       accept = true;
[4]   if (gender==male & age<80)
[5]       accept = true;
[6]   return accept; }
```

Considering `accept`, there are three DU pairs:
(a)[1]-[6]  (b)[3]-[6]  (c)[5]-[6]

Test cases for ‘all-uses’:

Same as for ‘all-defs’
Data Flow Testing – Example

Considering `accept`, there are three DU pairs:

(a)[1]-[6]  (b)[3]-[6]  (c)[5]-[6]

Test cases for ‘all-def-uses’:

Test cases needed to satisfy all-def-uses-paths criterion:
(a) `AccClient()` is executed and
   if[2] and if[4] are false
(b) `AccClient()` is executed and
   if[2] is true and if[4] is false
(c) `AccClient()` is executed and
   if[4] is true

Same as for ‘all-defs’
Data Flow Testing – Loops

Factorial (C program)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition line</th>
<th>Use line</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>result</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>result</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

DU-paths for variable result:
Data Flow Testing – Loops

Factorial (C program)

```
[1] public int factorial(int n){
[2]     int i, result = 1;
[3]     for (i=2; i<=n; i++) {
[4]         result = result * i;
[5]     }
[6]     return result;
[7] }
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition line</th>
<th>Use line</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>result</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>result</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

DU-paths for variable result:
Data Flow Testing – Loops

Factorial (C program)

```
[1] public int factorial(int n) {
[2]   int i, result = 1;
[3]   for (i=2; i<=n; i++) {
[4]     result = result * i;
[5]   }
[6]   return result;
[7] }
```

DU-paths for variable result:
{2-3-4, 2-3-5-6, 4-3-4, 4-3-5-6}
Data Flow Testing – Loops

Factorial (C program)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition line</th>
<th>Use line</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>result</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>result</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Why are not DU-paths

- 4-4
- 4-6
- 4-3-4-3-4
- 2-3-4-3-5-6

for variable result?

DU-paths for variable result:

{2-3-4, 2-3-5-6, 4-3-4, 4-3-5-6}
Data Flow Testing – Loops

Factorial (C program)

```
[1] public int factorial(int n){
[2]     int i, result = 1;
[3]     for (i=2; i<=n; i++) {
[4]         result = result * i;
[5]     }
[6]     return result;
[7] }
```

Why are not DU-paths for variable result:
• 4-4
• 4-6
• 4-3-4-3-4
• 2-3-4-3-5-6

for variable result?

Answer:
4-4 is use-def (and not def-use)
4-6 is not feasible (must always go through line 3)
4-3-4-3-4 and 2-3-4-5-6 are not def-clear
Data Flow Criteria

- All c-uses
- All uses
- All def-use paths

Weaker:
- All c-uses
- All p-uses
- All uses
- All def-use paths

Stronger:
- All c-uses, some p-uses
- All p-uses, some c-uses

# tests
Data Flow Criteria

- All c-uses
- All def-uses
- All uses
- All def-use paths
- All paths

Weaker

- All uses
- All def-use paths
- All paths

Stronger

- All branches

# tests
Comparing Effectiveness of Control-Flow & Data-Flow Test Criteria

- Compared branch (edge) coverage with def-use path coverage
- 10 people independently planted 130 bugs in different versions of seven C programs
- Test generation procedure was done such that many different tests suites with different degrees of coverage and size were (manually) produced

Source:
Comparing Effectiveness of Control-Flow & Data-Flow Test Criteria

• Result for one program:

  • Compared branch (edge) coverage with def-use path coverage
  • 10 people independently planted 130 bugs in different versions of seven C programs
  • Test generation procedure was done such that many different tests suites with different degrees of coverage and size were (manually) produced

Source:
Comparing Effectiveness of Control-Flow & Data-Flow Test Criteria

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>Test KLOC</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>96</td>
<td>50</td>
<td>2,205</td>
</tr>
<tr>
<td>Closure</td>
<td>90</td>
<td>83</td>
<td>7,927</td>
</tr>
<tr>
<td>Math</td>
<td>85</td>
<td>19</td>
<td>3,602</td>
</tr>
<tr>
<td>Time</td>
<td>28</td>
<td>53</td>
<td>4,130</td>
</tr>
<tr>
<td>Lang</td>
<td>22</td>
<td>6</td>
<td>2,245</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>321</strong></td>
<td><strong>211</strong></td>
<td><strong>20,109</strong></td>
</tr>
</tbody>
</table>

Source:
Comparing Effectiveness of Control-Flow & Data-Flow Test Criteria

<table>
<thead>
<tr>
<th>Test Criteria</th>
<th>Statement</th>
<th>Branch</th>
<th>MC/DC</th>
<th>Loop</th>
<th>All Control-flow</th>
<th># of undetected faults by control flow criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32%</td>
<td>32%</td>
<td>24%</td>
<td>12%</td>
<td>44%</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>5%</td>
<td>18%</td>
<td>18%</td>
<td>5%</td>
<td>24%</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>10%</td>
<td>18%</td>
<td>18%</td>
<td>18%</td>
<td>33%</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>0%</td>
<td>11%</td>
<td>11%</td>
<td>0%</td>
<td>15%</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>14%</td>
<td>18%</td>
<td>25%</td>
<td>8%</td>
<td>29%</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>10%</td>
<td>19%</td>
<td>19%</td>
<td>8%</td>
<td>28%</td>
<td>198</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Criteria</th>
<th>% of detected faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>def-use (DUA)</td>
<td>86% 87% 80% 91% 50%</td>
</tr>
<tr>
<td>Data &amp; control-flow</td>
<td>92% 90% 87% 92% 65%</td>
</tr>
</tbody>
</table>

Source:  
Structure of Lecture 9

- Data Flow-Testing
- Mutation Testing
- Lab 9
White-Box Testing Techniques

- Control-Flow Testing
- Data-Flow Testing
- Mutation Testing
- Symbolic Execution
- Static Code Analysis
- Reviews
Mutation Testing (Fault-Based Testing)

Assumption: tests pass in original

Test Cases Applied to Both Original & Mutant Program

Output is compared. If a test fails, different behavior has been detected => Mutant killed
Mutation Testing Procedure

1. Take a program and test data generated for that program
2. Create a number of similar programs (mutants), each differing from the original in a small way
3. The original test data are then run through the mutants
4. If tests detect all changes in mutants, then the mutants are dead and the test suite adequate
   Otherwise: Create more test cases and iterate 2-4 until a sufficiently high number of mutants is killed
Assessing Test Suite Quality

• Idea
  – I make $n$ copies of my program, each copy with a known number $m_n$ of (unique) faults
  – Assume introduced faults are exactly like real faults in every way
  – I run my test suite on the programs with seeded faults ...
    • ... and the tests reveal 20% of the introduced faults

• What can I infer about my test suite?
Mutation Testing – Terminology

• **Mutant** – new version of the program with a small deviation (=fault) from the original version

• **Killed** mutant – new version detected by the test suite

• **Live** mutant – new version *not* detected by the test suite
Examples of Mutation Operations

• Change relational operator (<, >, …)
• Change logical operator (||, &,
  …)
• Change arithmetic operator (*, +,
  -, …)
• Change constant name / value
• Change variable name / initialisation
• Change (or even delete) statement
• …

more examples
http://pitest.org/quickstart/mutators/
Example Mutants

\[
\begin{align*}
&\text{if } (a \lor b) \\
&\quad \text{c = a + b;} \\
&\text{else} \\
&\quad \text{c = 0;}
\end{align*}
\]

\[
\begin{align*}
&\text{if } (a \land b) \\
&\quad \text{c = a + b;} \\
&\text{else} \\
&\quad \text{c = 0;}
\end{align*}
\]

\[
\begin{align*}
&\text{if } (a \lor b) \\
&\quad \text{c = a \ast b;} \\
&\text{else} \\
&\quad \text{c = 0;}
\end{align*}
\]
Types of Mutants

Not interesting:

- **Stillborn mutants**: Syntactically incorrect – killed by compiler, e.g., \( x = a ++ b \)
- **Trivial mutants**: Killed by almost any test case
- **Equivalent mutant**: Always acts in the same behaviour as the original program, e.g., \( x = a + b \) and \( x = a - (-b) \)

Those mutants are interesting which behave differently than the original program, and we do not (yet) have test cases to identify them.
Equivalent Mutants

```
if (a == 2 && b == 2)
    c = a + b;
else
    c = 0;

int index=0;
while (...)
{
    . . .;
    index++;
    if (index==10)
        break;
}
```

```
if (a == 2 && b == 2)
    c = a * b;
else
    c = 0;

int index=0;
while (...)
{
    . . .;
    index++;
    if (index>=10)
        break;
}
```
Program Example

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] > a[imax]
            imax := i;
    return imax;
}
Program Example

nobjs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] > a[imax]
            imax := i;
    return imax;
}

Program returns the index of the array element with the maximum value.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Program returns the index of the array element with the maximum value.
Variable Name Mutant

nbrs = new int[range]
public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if i > a[imax]
            imax:= i;
    return imax;
}

Program returns the index of the (first) array element with the maximum value.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Variable Name Mutant

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if i > a[imax]
            imax:= i;
    return imax;
}

Program returns the index of the (first) array element with the maximum value.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1&gt;2</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Killed!
Relational Operator Mutant

```java
nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] >= a[imax]
            imax := i;
    return imax;
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Need a test case with two identical max entries in a[.], e.g., (1, 3, 3)

Not killed!
Variable Operator Mutant

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 0; i < range; i++)
        if a[i] > a[imax]
            imax := i;
    return imax;
}

Need a test case detecting wrong loop counting

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Not killed!
Mutation Testing Assumptions

• Competent programmer hypothesis:
  – Programs are nearly correct
    • Real faults are small variations from the correct program
    • => Mutants are reasonable models of real faulty programs

• Coupling effect hypothesis:
  – Tests that find simple faults also find more complex faults
    • Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too
Mutation Testing Tool: PIT

Real world mutation testing

PIT is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

Get Started
## Default Mutation Operators in PIT

<table>
<thead>
<tr>
<th>Mutation operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditionals Boundary</td>
<td>Replaces relational operators with their boundary counterpart (e.g., $&lt;$ becomes $\leq$, $\geq$ becomes $&gt;$, etc.).</td>
</tr>
<tr>
<td>Negate Conditionals</td>
<td>Replaces all conditionals with their negated counterpart (e.g., $==$ becomes $!=$, $&lt;$ becomes $\geq$, etc.).</td>
</tr>
<tr>
<td>Math</td>
<td>Replaces binary arithmetic operations from either integer or floating-point arithmetic with another operation (e.g., $+$ becomes $-$, $*$ becomes $/$, etc.).</td>
</tr>
<tr>
<td>Increments</td>
<td>Replaces increments of local variables with decrements and vice versa.</td>
</tr>
<tr>
<td>Invert Negatives</td>
<td>Inverts the negation of integer and floating point numbers.</td>
</tr>
<tr>
<td>Return Values</td>
<td>Changes the return value of a method depending on the return type (e.g., non-null return values are replaced with null, integer return values are replaced with 0, etc.).</td>
</tr>
<tr>
<td>Void Method Call</td>
<td>Removes method calls to void methods.</td>
</tr>
</tbody>
</table>
Structure of Lecture 9

• Data Flow-Testing
• Mutation Testing
• Lab 9
Lab 9 – Mutation Testing

Lab 9 (week 33: Apr 14 & 15) – Mutation Testing (10 points)

Lab 9 Instructions & Tools

Submission Deadlines:
  • Tuesday Labs: Monday, 20 Apr, 23:59
  • Wednesday Labs: Tuesday, 21 Apr, 23:59
  • Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late

SUT: Minimum Binary Heap (incl. Test code)
Mutation Testing Tool: PIT
Lab 9 – Mutation Testing (cont’d)

- Part 1 – Code Defenders Game (during lab)

- Part 2 – Lab 9 Assignment (started in lab and completed at home)

http://code-defenders.org
Lab 9 – Mutation Testing (cont’d)

**Instructions**

**Mutation Testing:**
Run tests, kill mutants
Add tests, kill more mutants, detect faults

**SUT: Minimum Binary Heap (incl. Test code)**

**Mutation Testing Tool:**
PIT

**Report:**
Detected faults
Mutation coverage
Code coverage

**Improved Test Suite**
Next Week

• Quiz 9 → Moodle

• Lab 9:
  – Mutation Testing

• Lecture 10:
  White-Box Testing (advanced):
  – Symbolic Execution
  – Static Code Analysis
  – Document Inspection / Code Review