LTAT.05.006: Software Testing

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

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Spring 2019
Lectures (J. Liivi 2-111)

- Lecture 1 (14.02) – Introduction to Software Testing
- Lecture 2 (21.02) – Basic Black-Box Testing Techniques
- Lecture 3 (28.02) – BBT advanced: Combinatorial Testing
- Lecture 4 (07.03) – Basic White-Box Testing Techniques
- Lecture 5 (14.03) – Test Lifecycle, Test Tools, Test Automation
- Lecture 7 (28.03) – BBT advanced: State-Transition Testing & Exploratory Testing
- Lecture 8 (04.04) – BBT advanced: Security, Usability and A/B Testing
- Lecture 9 (11.04) – WBT advanced: Data-Flow Testing / Mutation Testing
- Lecture 11 (25.04) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
- 02.05 - no lecture
- Lecture 12 (09.05) – Industry Guest Lecture (to be announced)
- Lecture 13 (16.05) – 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.

\[ X \leftarrow 14; \quad \ldots \quad Y \leftarrow X-3; \]

**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

The values given in **defs** should **reach** at least one, some, or all possible **uses**

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

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

- First, we make sure every def reaches a use

**All-defs coverage (ADC)**: For each set of du-paths \( S = du(n, v) \), Test-Set covers at least one path \( p \) in \( S \).

- Then we make sure that every def reaches all possible uses

**All-uses coverage (AUC)**: For each set of du-paths to uses \( S = du(n_i, n_j, v) \), Test-Set covers at least one path \( p \) in \( S \).

- Finally, we cover all the paths between defs and uses

**All-du-paths coverage (ADUPC)**: For each set \( S = du(n_i, n_j, v) \), Test-Set covers every path \( p \) in \( S \).
Data Flow Testing Example

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

\[ X = 42 \]

\[ Z = X - 8 \]

\[ Z = X \times 2 \]

All-defs for \( X \)

\[
\begin{align*}
[ &1, 2, 4, 5] \\
\text{or} \\
[ &1, 2, 4, 6]
\end{align*}
\]

All-uses for \( X \)

\[
\begin{align*}
[ &1, 2, 4, 5] \\
\text{or} \\
[ &1, 2, 4, 6]
\end{align*}
\]

All-du-paths for \( X \)

\[
\begin{align*}
[ &1, 2, 4, 5] \\
\text{or} \\
[ &1, 3, 4, 5] \\
[ &1, 2, 4, 6] \\
[ &1, 3, 4, 6]
\end{align*}
\]
Data Flow Testing – Definitions

- **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.

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

Considering age, 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 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’?

[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?

```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 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’?

```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
```
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
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

Test cases for ‘all-uses’: AccClient(*, *) -> *
→ 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’?

```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 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\}
Data Flow Testing – Example

[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, what are the DU pairs?
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

```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]

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

Considering `accept`, there are three DU pairs:

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

Test cases for ‘all-uses’:

```
[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
```

Test cases needed to satisfy all-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 – Example

Considering `accept`, there are three DU pairs:

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

Test cases for ‘all-def-uses’:

Same as for ‘all-defs’

```
[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 – Loops

Factorial (C program)

```c
public int factorial(int n){
    int i, result = 1;
    for (i=2; i<=n; i++) {
        result = result * i;
    }
    return result;
}
```

DU-paths for variable `result`:

<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>
Data Flow Testing – Loops

Factorial (C program)

```c
[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 not DU-paths
- 4-4
- 4-6
- 2-3-4-3-5-6
for variable return?

DU-paths for variable result:
{2-3-4, 2-3-5-6, 4-3-4, 4-3-5-6}
Intra- and Inter-class Data-flow Testing

• Exercise sequences of methods
  • From setting or modifying a field value
  • To using that field value

• Mainly interested in caller/callee relation:
  • Consider coupling-based criteria:
    • Only consider ‘last def’ (before calling) and ‘first use’ (when called) / also consider def-use of return value

• We need a control flow graph that encompasses more than a single method ...
Class diagram of a more complex example program ...
The intra-class control flow graph

Control flow for each method
+ node for class
+ edges
  from node class to the start nodes of the methods
  from the end nodes of the methods to node class

=> control flow through sequences of method calls
Definition-Use (DU) pairs

instance variable `legalConfig`

<model (1.2), isLegalConfiguration (7.2)>
<addComponent (4.6), isLegalConfiguration (7.2)>
<removeComponent (5.4), isLegalConfiguration (7.2)>
<checkConfiguration (6.2), isLegalConfiguration (7.2)>
<checkConfiguration (6.3), isLegalConfiguration (7.2)>
<addComponent (4.9), isLegalConfiguration (7.2)>

Each pair corresponds to a test case
note that

- some pairs may be infeasible
- to cover pairs we may need to find complex sequences
Definition-Use (DU) pairs

instance variable \texttt{legalConfig}

\begin{itemize}
  \item \texttt{model (1.2), isLegalConfiguration (7.2)}
  \item \texttt{addComponent (4.6), isLegalConfiguration (7.2)}
  \item \texttt{removeComponent (5.4), isLegalConfiguration (7.2)}
  \item \texttt{checkConfiguration (6.2), isLegalConfiguration (7.2)}
  \item \texttt{checkConfiguration (6.3), isLegalConfiguration (7.2)}
  \item \texttt{addComponent (4.9), isLegalConfiguration (7.2)}
\end{itemize}

\textbf{If} (!\texttt{isLegalConfig})

\begin{itemize}
  \item \texttt{checkConfiguration (7.2)}
  \item \texttt{return legalConfig (7.4)}
\end{itemize}
Class diagram of a more complex example program ...

(c) 2008 Mauro Pezzè & Michal Young
Inspectors and modifiers

• Classify methods (on execution paths) as
  • *inspectors*: use, but do not modify, instance variables
  • *modifiers*: modify, but not use instance variables
  • *inspector/modifiers*: use and modify instance variables

• Example – class *Slot*:
  • Slot()  *modifier*
  • bind()  *modifier*
  • unBind()  *modifier*
  • isBound()  *inspector*

Note:
This is different to the previous slide where only methods of the same class were involved
Definitions from modifiers

Definitions of instance variable *slot* in class *Model*

addComponent (4.5)
addComponent (4.7)
addComponent (4.8)
selectModel (2.3) – not shown
removeComponent (5.3) – not sh.

Slot() modifier
bind() modifier
unBind() modifier
isBound() inspector
Uses from inspectors

Uses of instance variable `slot` in class `Model`
removeComponent (5.2) – not shown
checkConfiguration (6.4)
checkConfiguration (6.5)
checkConfiguration (6.7)

Slot() modifier
bind() modifier
unBind() modifier
isBound() inspector

void checkConfiguration() 6.1
legalConfig = true 6.2
int i = 0 6.3

i < slot.length 6.4
Slot slot = slots[i] 6.5
++i 6.6
if (slot.required && !slot.isBound()) 6.7
legalConfig = false 6.8
exit checkConfiguration 6.9
DF-Testing for Web Applications

- **Definition-use testing**: all navigation paths from every definition of a variable to every use of it is exercised.

- **All-uses testing**: at least one navigation path from every definition of a variable to every use of it is exercised.
Data Flow Criteria

Weaker

All c-uses
All defs
All p-uses

All c-uses, some p-uses
All p-uses, some c-uses

All uses

Stronger

All def-use paths

# tests
Data Flow Criteria

- **All c-uses**
- **All defs**
- **All p-uses**

**Weaker**
- All uses
- All def-use paths
- All paths

**Stronger**
- All branches
- # tests

- All uses, some p-uses
- All p-uses, some c-uses
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 degree of coverage and size were (manually) produced

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

• Result for one program:
### Comparing Effectiveness of Control-Flow & Data-Flow Test Criteria

#### TABLE I. SYSTEMS UNDER TEST

<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></th>
<th>% of detected faults</th>
<th></th>
<th>% of detected faults</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement</td>
<td>32%</td>
<td>5%</td>
<td>10%</td>
<td>0%</td>
</tr>
<tr>
<td>Branch</td>
<td>32%</td>
<td>18%</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>MC/DC</td>
<td>24%</td>
<td>18%</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>Loop</td>
<td>12%</td>
<td>5%</td>
<td>18%</td>
<td>0%</td>
</tr>
<tr>
<td>All Control-flow</td>
<td>44%</td>
<td>24%</td>
<td>33%</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>84</td>
<td>41</td>
<td>23</td>
</tr>
</tbody>
</table>

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

Lecture 10
Mutation Testing (Fault-Based Testing)

Original Program → Fault Introduction → Mutant Program

Test Cases Applied to Both Original & Mutant Program

output is compared. If results for original and mutant program are same, mutant 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 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
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 (II, &&, ..., ) 
• Change arithmetic operator (*, +, -, ..., ) 
• Change constant name / value 
• Change variable name / initialisation 
• Change (or even delete) statement 
• ... 

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

if (a || b)
   c = a + b;
else
   c = 0;

if (a || b)
   c = a + b;
else
   c = 0;

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

if (a || b)
   c = a * b;
else
   c = 0;
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

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

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

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

```c
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 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 Example

nbrs = new int[range]
public int max(int[] a) {
    intimax := 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.
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>2</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Relational Operator Mutant

```
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;
}
```

Need a test case with two identical max entries in a[.], e.g., (1, 3, 3)
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>
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., <code>&lt;</code> becomes <code>&lt;=</code>, <code>&gt;=</code> becomes <code>&gt;</code>, etc.).</td>
</tr>
<tr>
<td>Negate Conditionals</td>
<td>Replaces all conditionals with their negated counterpart (e.g., <code>==</code> becomes <code>!=</code>, <code>&lt;</code> becomes <code>&gt;=</code>, etc.).</td>
</tr>
<tr>
<td>Math</td>
<td>Replaces binary arithmetic operations from either integer or floating-point arithmetic with another operation (e.g., <code>+</code> becomes <code>−</code>, <code>*</code> becomes <code>/</code>, 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 <code>null</code>, integer return values are replaced with <code>0</code>, 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 16 & 17) – Mutation Testing (10 points)

Lab 9 Instructions & Tools

Submission Deadlines:
• Tuesday Labs: Monday, 22 Apr, 23:59
• Wednesday Labs: Tuesday, 23 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
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)

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

Instructions

Mutation Testing Tool: PIT

SUT: Minimum Binary Heap (incl. Test code)

Report: Detected faults, Mutation coverage, Code coverage

Improved Test Suite

Mutants

Improved Test Suite
Next Week

• Quiz 9:
  • Data-flow Testing & Mutation Testing

• Lab 9:
  – Mutation Testing

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