Lecture 10: White-Box Testing (advanced) – Data-Flow Testing and Mutation Testing

Spring 2022
Lectures

• Lecture 1 (10.02) – Introduction to Software Testing
• Lecture 2 (17.02) – Basic Black-Box Testing Techniques: Boundary Value Analysis & Equivalence Class Partitioning
• Lecture 3 (03.03) – BBT advanced: Combinatorial Testing
• Lecture 4 (10.03) – Basic White-Box Testing Techniques: Control-Flow Coverage
• Lecture 5 (17.03) – BBT adv.: State-Transition, Metamorphic, Random Testing
• Lecture 6 (24.03) – Test Levels, Test Tools, Test Automation
• Lecture 7 (31.03) – BBT adv.: Exploratory Testing, Behaviour Testing
• Lecture 8 (07.04) – BBT adv.: GUI / Visual Testing, Usability Testing, A/B Testing
• Lecture 9 (14.04) – Security Testing of Mobile Applications
• Lecture 10 (21.04) – WBT adv.: Data-Flow Testing / Mutation Testing
• Lecture 11 (28.04) – WBT adv.: Symbolic Execution, Static Code Analysis, Review
• Lecture 12 (05.05) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
• Lecture 13 (12.05) – Exam Preparation
• Lecture 14 (19.05) – Advanced Topics (optional)
White-Box Testing Techniques

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

Lecture 11
Structure of Lecture 10

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

• Node (=statement) and edge (=branch) coverage don’t test interactions between statements much
• 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

\[
\begin{align*}
X &= 42 \\
Z &= X \times 2 \\
Z &= X - 8
\end{align*}
\]

**Defs:**
- \(\text{def}(1) = \{X\}\)
- \(\text{def}(5) = \{Z\}\)
- \(\text{def}(6) = \{Z\}\)

**Uses:**
- \(\text{use}(5) = \{X\}\)
- \(\text{use}(6) = \{X\}\)

The values given in **defs** 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 - 8

Z = X*2

All-defs for X: [1, 2, 4, 5]

All-uses for X: [1, 2, 4, 5, 6]

All-du-paths for X: [1, 2, 4, 5], [1, 3, 4, 5], [1, 2, 4, 6], [1, 3, 4, 6]
Data Flow Testing Example

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

All-defs for X
[ 1, 2, 4, 5 ]

or

All-defs for X
[ 1, 2, 4, 6 ]

or ...

All-uses for X

All-du-paths for X

X = 42

Z = X - 8

Z = X*2

1
2
3
4
5
6
7
Data Flow Testing Example

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

All-defs for X

[ 1, 2, 4, 5 ]  
[ 1, 2, 4, 6 ]

All-uses for X

or

All-du-paths for X

or

...
Data Flow Testing Example

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

All-defs for $X$: 
$[1, 2, 4, 6]$

All-uses for $X$: 
$[1, 2, 4, 5]$
$[1, 3, 4, 5]$
$[1, 2, 4, 6]$
$[1, 3, 4, 6]$

All-du-paths for $X$: 
$[1, 2, 4, 5]$
$[1, 3, 4, 5]$
$[1, 2, 4, 6]$
$[1, 3, 4, 6]$
Data Flow Testing Example

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

All-defs for X

[ 1, 2, 4, 5 ]

or

[ 1, 2, 4, 6 ]

or ...
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.

```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
```

\[\text{Def}(0/1) = \{\text{age, gender, accept}\}\]

\[\text{P-use}(2) = \{\text{age, gender}\}\]

\[\text{Def}(3) = \{\text{accept}\}\]

\[\text{P-use}(4) = \{\text{age, gender}\}\]

\[\text{C-use}(6) = \{\text{accept}\}\]

\[\text{Def}(5) = \{\text{accept}\}\]
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(*, *) -> *
 degli: 0-1-2

Test cases needed to satisfy all-defs-paths criterion:
AccClient() is executed
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 \textit{age}, there are two DU pairs:

(a)\([0]-[2]\)

(b)\([0]-[4]\)

Test case(s) for ‘all-uses’:

AccClient\((*, *) \rightarrow *\)

\[\Rightarrow\] 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-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 \textit{age}, there are two DU pairs:

(a)[0]-[2]

(b)[0]-[4]

Test case(s) for ‘all-def-uses’:

\begin{itemize}
  \item AccClient(83,f) \rightarrow true
  \item AccClient(90,f) \rightarrow false
\end{itemize}

\Rightarrow 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 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 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(s) for ‘all-defs’:
AccClient(*) is executed
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; }
```
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(s) 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 case(s) 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 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(s) for ‘all-def-uses’:
AccClient(83,f) -> true
AccClient(90,f) -> false
covers {0-1-2, 0-1-2-3-4, 0-1-2-4}

[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`, what are the DU pairs?

```c
bool AccClient(int age;  
gtype gender) {
    bool accept = false;
    if (gender==female & age<85)  
        accept = true;
    if (gender==male & age<80)
        accept = true;
    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

```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

```
[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]

DU paths:
(a) 1-2-4-6
(b) 3-4-6
(c) 5-6

Question:

Why is 1-2-3-4-6 not a du-path?
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

Question:
Why is 1-2-3-4-6 not a du-path?

```c
// Function to check if a client is accepted
bool AccClient(int age; gtype gender) {
    bool accept = false;
    if (gender==female & age<85)
        accept = true;
    if (gender==male & age<80)
        accept = true;
    return accept; }
```

Not definition-free!
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’:

```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 `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(83,f)` -> true
(c) `AccClient(79,m)` -> true

```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; }
```

Test cases needed to satisfy all-defs-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
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’:

```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 `accept`, there are three DU pairs:

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

Test cases for ‘all-uses’:

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’

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

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 `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’
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`:
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`:

<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)

```
[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`:

{2-3-4, 2-3-5-6, 4-3-4, 4-3-5-6}
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 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)

```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 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

Stronger

All p-uses, some c-uses

All uses

All def-use paths

Weak

All c-uses, some p-uses

All uses

All def-use paths

# tests
Data Flow Criteria

All c-uses  All defs  All p-uses

Weaker

Stronger

All uses

All def-use paths

All branches

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

Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria

Monica Hutchins, Herb Foster, Tunk Goradia, Thomas Ostrand
Siemens Corporate Research, Inc.
755 College Road East, Princeton, NJ 08540 U.S.A.
Email: mhutchins,bfoster,tgoradia,aostrand@siemens.com

Abstract
This paper reports an experimental study investigating the effectiveness of two code-based test adequacy criteria for identifying test cases that detect faults. The all-edges and all-defs (modified all-defs) coverage criteria were applied to 130 faulty program versions derived from seven moderate size kernel programs by sending realistic inputs. We measured several thousand test cases for each faulty program and examined the relationship between fault detection and coverage. Within the limited domain of our experiments, test sets achieving coverage levels over 80% usually showed significantly higher fault detection than randomly chosen test sets of the same size.

In addition, significant improvements in the effectiveness of coverage-based test sets actually occurred as coverage increased from 40% to 100%. However, the results also indicate that 100% code coverage alone is not a reliable indicator of the effectiveness of a test set. We also found that test sets based on control flow and data flow criteria are frequently complementary in their effectiveness.

1 Introduction
Control-flow-based code coverage criteria have been available to monitor the thoroughness of software tests at least since the 1980s [11, 12, 24]. More recently, data-flow-based methods have been defined and implemented in several tools (7, 19, 20). Various comparisons have been made of the theoretical relations between coverage methods [1]. However, the questions of real concern to researchers and potential users of these adequacy criteria deal with their actual effectiveness in detecting the presence of faults in programs. Test managers and developers would like to know whether the investment in systems to measure code coverage is worthwhile, and whether the effort to base for additional tests that increase coverage is worthwhile. They would also like to know the additional cost of achieving adequate coverage. Adequate coverage criteria are of practical interest because they are intended to improve the detection of faults. Our results showed that fault detection increased significantly if test sets were adequate or close to adequate according to the criteria.

In an effort to answer these questions, we have performed experiments comparing dataflow coverage and controlflow coverage using the dataflow coverage tool Tactic developed at Siemens Corporate Research [20]. To make our results as relevant as possible to professional software developers and testers, we searched available public archives for specifications and C programs that would be suitable for the study. We ended up with seven standard-size C programs, one of which we avoided 100 different faults.

Section 2 of the paper describes the test adequacy criteria that are measured by Tactic. Section 3 briefly describes some previous work relating to evaluation of adequacy criteria. Section 4 presents the goals of our study, chooses programs and the design of the experiments, and describes the programs used in the study. Section 5 summarizes some of the data analysis. Section 6 describes our observations. Section 7 contains conclusions.

2 The Test Adequacy Criteria

2.1 Dataflow Coverage

Dataflow-based adequacy criteria stipulate that a test set must include certain data-use associations that exist in the code. A def of a memory location is an operation that writes a value to the location. A use of a location is an operation that reads the location’s current value. A def-use association (DU) for a given location is a pair consisting of a def and a use of the location, such that there is a controlflow path in the code from the def to the use on which there is no intermediate redefinition or modification of the location. A test case exercises a particular def-use association if the test case causes execution to arrive at the site of the def operation and execute the use, without having executed any other def or modification of the memory location. Test cases are irrelevant if in the set exercises the DU. Reuse of a DU is not allowed.

Note that a DU is defined in terms of static properties of the code, i.e., its test cases must contain a path in the code’s controlflow graph, while satisfying a DU is defined in terms of dynamic execution. To satisfy the all-

• 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

• Result for one program:
  - Up to 25 tests
  - Up to 50 tests
  - 2% steps

• Close to 100% DU-path coverage requires larger test suites than close to 100% branch coverage
• Test suites were plotted with the specific coverage criterion in mind: a 2%-step coverage improvement is shown in the graph (for each, DU and edge coverage)
• Low coverage suites are not shown

Source:
Structure of Lecture 10

• Data Flow-Testing
• Mutation Testing
• Lab 9
Mutation Testing (Fault-Based Testing)

Assumption: tests pass in original => Mutant killed

Output is compared. If a test fails, different behavior has been detected => Mutant 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 injected (seeded) faults

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

1. Take a program and a test set 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

Note: Inspecting the buggy line in the mutant and adding tests may disclose bugs in original program
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  
- ...
Example Mutants

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

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

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

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

\[
\begin{align*}
\text{if} \ (x \ || \ y) \\
\quad &c = a \times b; \\
\text{else} \\
\quad &c = 0;
\end{align*}
\]

Mutant 1

Mutant 2
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

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

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

- Equivalent Mutant 1

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

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

- Equivalent Mutant 2
Program Example

```java
class Main {
    public int mxi(int[] a) {
        int imax = 0;
        for (int i = 1; i <= a.length; i++) {
            if (a[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>
Program Example

```java
nbrs = new int[range]
public int mxi(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 (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 mxi(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

```java
nbrs = new int[range]
public int mxi(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 (2)</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2 (1)</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0 (0)</td>
</tr>
</tbody>
</table>
Variable Name Mutant

nbrs = new int[range]

public int mxi(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 (2)</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2 (1)</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0 (0)</td>
</tr>
</tbody>
</table>

Killed!
Relational Operator Mutant

nhrs = new int[range]

public int mxi(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)

<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!
Variable Operator Mutant

nbrs = new int[range]

public int mxi(int[] a) {
    int imax := 0;
    for (int i = 0; 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 detecting wrong loop counting

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
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>
</table>
| Conditionals Boundary        | Replaces relational operators with their boundary counterpart (e.g.,  
                            |   `<` becomes `<=`, `>=` becomes `>`, etc.).                                   |
| Negate Conditionals          | Replaces all conditionals with their negated counterpart (e.g.,       |
                            |   `==` becomes `!=`, `<` becomes `>=`, etc.).                                    |
| Math                         | Replaces binary arithmetic operations from either integer or      |
                            |   floating-point arithmetic with another operation (e.g., `+`         |
                            |   becomes `-`, `*` becomes `/`, etc.).                                         |
| Increments                   | Replaces increments of local variables with decrements and vice versa.   |
| Invert Negatives             | Inverts the negation of integer and floating point numbers.              |
| Return Values                | 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.).                    |
| Void Method Call             | Removes method calls to void methods.                                    |
Structure of Lecture 10

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

Lab 9 (week 35: Apr 26 & 27) – Mutation Testing (9 points)

Lab 9 Instructions & Tools

Submission Deadlines:
- Tuesday Labs: Monday, 02 May, 23:59
- Wednesday Labs: Tuesday, 03 May, 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)

Instructions

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

Mutation Testing Tool: PIT

SUT: Minimum Binary Heap (incl. Test code)

Report:
- Detected faults
- Mutation coverage
- Code coverage

Improved Test Suite

Mutants
Next Week

• Quiz 9 → Moodle (opens after Lecture 10)

• Lab 9:
  – Mutation Testing

• Lecture 11:
  White-Box Testing (advanced):
  – Symbolic Execution
  – Static Code Analysis
  – Document Inspection / Code Review