Mutation Testing

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong

Speaker Biographical Sketch

- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
- Guest Researcher, National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  – Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses
- Founder & Steering Committee co-Chair for the SERE conference
  (IEEE International Conference on Software Security and Reliability)
  (http://paris.utdallas.edu/sere13)
**Mutation Testing (1)**

- The process of program development is considered as *iterative* whereby an initial version of the program is *refined* by making simple, or a combination of simple changes, towards the final version.

- Mutation testing is a *code-based test assessment* and improvement technique.
  - Can be extended to architecture (e.g., Statecharts) and design (e.g., SDL)

**Mutation Testing (2)**

- It relies on the *competent programmer hypothesis* which is the following assumption:
  - Given a specification a programmer develops a program that is either correct or differs from the correct program by a combination of simple errors.

- It also relies on “*coupling effect*” which suggests that
  - Test cases that detect simple types of faults are sensitive enough to detect more complex types of faults.
**Mutant (1)**

- Given a program $P$, a mutant of $P$ is obtained by making a **simple change** in $P$.

**Example (1)**

- $a := b + c$
- $a := c + c$
- $a := b - c$
Example (2)

Program
1. int x, y;
2. if (x != 0)
3. y = 5;
4. else z = z - x;
5. if (z > 1)
6. z = z/x;
7. else
8. z = y;

Mutant
1. int x, y;
2. if (x != 0)
3. y = 5;
4. else z = z - x;
5. if (z > 1)
6. z = z/x;
7. else
8. z = y;

Example (3)

Program
1. int x, y;
2. if (x != 0)
3. y = 5;
4. else z = z - x;
5. if (z > 1)
6. z = z/x;
7. else
8. z = y;

Mutant
1. int x, y;
2. if (x != 0)
3. y = 5;
4. else z = z - x;
5. if (z < 1)
6. z = z/x;
7. else
8. z = y;
**Order of Mutants**

- First order mutants
  - One syntactic change
- Higher order mutants
  - Multiple syntactic changes
- Coupling effect

**Type of Mutants (1)**

- Distinguished mutants
- Live mutants
- Equivalent mutants
- Non-equivalent mutants
**Type of Mutants (2)**

- A mutant $m$ is considered *distinguished* (or *killed*) by a test case $t \in T$ if
  
  $$P(t) \neq m(t)$$

  where $P(t)$ and $m(t)$ denote, respectively, the observed behavior of $P$ and $m$ when executed on test input $t$.

- A mutant $m$ is considered *equivalent* to $P$ if
  
  $$P(t) = m(t)$$

  for any test case in the input domain.

---

**Distinguish a Mutant (1)**

- **Reachability**
  - Execute the mutated statement

- **Necessity**
  - Make a state change

- **Sufficiency**
  - Propagate the change to output
**Distinguish a Mutant (2)**

Program $P$
read $a$
if ($a > 3$)
then
 $x = 5$
else
 $x = 2$
endif
print $x$

Program $m$
read $a$
if ($a \geq 3$)
then
 $x = 5$
else
 $x = 2$
endif
print $x$

Mutant $m$ is distinguished by $a = 3$

**Equivalent Mutant (1)**

Program $P$
read $a$, $b$
$a = b$
$x = a + b$
print $x$

Program $m$
read $a$, $b$
$a = b$
$x = a + a$
print $x$

$P$ is equivalent to $m$
**Equivalent Mutant (2)**

- Consider the following program $P$

```c
int x, y, z;
scanf (&x, &y);
if (x>0)
    x = x + 1; z = x × (y - 1);
else
    x = x - 1; z = x × (y - 1);
```

- Here $z$ is considered the output of $P$

**Equivalent Mutant (3)**

- Now suppose that a mutant of $P$ is obtained by changing $x = x + 1$ to
  $x = abs(x) + 1$

- This mutant is equivalent to $P$ as no test case can distinguish it from $P$
**Mutation Score (1)**

- During testing a mutant is considered *live* if it has not been distinguished or proven equivalent.

- Suppose that a total of $M_t$ mutants are generated for program $P$.

- The *mutation score* of a test set $T$, designed to test $P$, is computed as:

$$MS(P, T) = \frac{M_k}{M_t - M_q}$$

  - $M_k$ – number of mutants killed
  - $M_q$ – number of equivalent mutants
  - $M_t$ – total number of mutants

---

**Mutation Score (2)**

- Mutation score:
  
  \[ \frac{\text{Number of mutants distinguished}}{\text{Total number of non-equivalent mutants}} \]

- Data flow score:
  
  \[ \frac{\text{Number of blocks (decisions, p-uses, c-uses, all-uses) covered}}{\text{Total number of feasible blocks (decisions, p-uses, c-uses, all-uses)}} \]
**Test Adequacy Criterion**

- A test \( T \) is considered *adequate* with respect to the mutation criterion if its mutation score is 1
  - Equivalent mutants?
  - Which mutant operators are used?

- The number of mutants generated depends on \( P \) and the *mutant operators* applied on \( P \)

- A *mutant operator* is a rule that when applied to the program under test generates zero or more mutants

---

**Mutant Operator (1)**

- Consider the following program:
  ```c
  int abs (x);
  int x;
  {  
    if (x ≥ 0) x = 0 − x;
    return x;
  }
  ```
**Mutant Operator (2)**

- Consider the following rule:
  - Replace each relational operator in \( P \) by all possible relational operators excluding the one that is being replaced.

- Assuming the set of relational operators to be: \( \{<, >, \leq, \geq, ==, !=\} \), the above mutant operator will generate *a total of 5 mutants of \( P \)*.

**Mutant Operator (3)**

- Mutation operators are *language dependent*

- For Fortran a total of 22 operators were proposed

- For C a total of 77 operators were proposed
Mutant Operators for Fortran (1)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr</td>
<td>array reference for array reference replacement</td>
<td>cca</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value insertion</td>
<td>pds</td>
</tr>
<tr>
<td>acr</td>
<td>array reference for constant replacement</td>
<td>cca</td>
</tr>
<tr>
<td>aor</td>
<td>arithmetic operator replacement</td>
<td>cca</td>
</tr>
<tr>
<td>asr</td>
<td>array reference for scalar variable replacement</td>
<td>cca</td>
</tr>
<tr>
<td>cfr</td>
<td>constant for array reference replacement</td>
<td>cca</td>
</tr>
<tr>
<td>crn</td>
<td>comparable array name replacement</td>
<td>cca</td>
</tr>
<tr>
<td>cnp</td>
<td>constant replacement</td>
<td>pda</td>
</tr>
<tr>
<td>csr</td>
<td>constant for scalar variable replacement</td>
<td>cca</td>
</tr>
<tr>
<td>der</td>
<td>Do statement end replacement</td>
<td>sal</td>
</tr>
<tr>
<td>dsa</td>
<td>DATA statement alterations</td>
<td>pds</td>
</tr>
<tr>
<td>glt</td>
<td>GOTO label replacement</td>
<td>sal</td>
</tr>
<tr>
<td>lcr</td>
<td>logical connector replacement</td>
<td>pda</td>
</tr>
<tr>
<td>rot</td>
<td>relational operator replacement</td>
<td>pda</td>
</tr>
<tr>
<td>rls</td>
<td>RETURN statement replacement</td>
<td>sal</td>
</tr>
<tr>
<td>san</td>
<td>statement analysis (replacement by TRAP)</td>
<td>sal</td>
</tr>
<tr>
<td>srn</td>
<td>scalar variable for array reference replacement</td>
<td>cca</td>
</tr>
<tr>
<td>scc</td>
<td>scalar for constant replacement</td>
<td>cca</td>
</tr>
<tr>
<td>sde</td>
<td>statement deletion</td>
<td>sal</td>
</tr>
<tr>
<td>src</td>
<td>source constant replacement</td>
<td>cca</td>
</tr>
<tr>
<td>svr</td>
<td>scalar variable replacement</td>
<td>cca</td>
</tr>
<tr>
<td>soi</td>
<td>unary operator insertion</td>
<td>pda</td>
</tr>
</tbody>
</table>

Mutant Operators for Fortran (2)

- **san**: replace each statement by TRAP
  (an instruction that causes the program to halt, killing the mutant)
  - Which code coverage-based criterion will also be satisfied by killing all the san mutants?

- **rfr**: replace each statement in a subprogram by RETURN
### Mutant Operators for C (1)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Con</td>
<td>Constant for Constant Replacement</td>
</tr>
<tr>
<td>Csr</td>
<td>Constant for Scalar Replacement</td>
</tr>
<tr>
<td>CRCA</td>
<td>Required Constant Replacement</td>
</tr>
</tbody>
</table>

### Mutant Operators for C (2)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLAN</td>
<td>Logical Operator by Arithmetic Operator</td>
</tr>
<tr>
<td>OLLN</td>
<td>Logical Operator by Logical Operator</td>
</tr>
<tr>
<td>OLN</td>
<td>Logical Negation</td>
</tr>
<tr>
<td>ORAN</td>
<td>Relational Operator by Arithmetic Operator</td>
</tr>
<tr>
<td>ORLN</td>
<td>Relational Operator by Logical Operator</td>
</tr>
<tr>
<td>OLRN</td>
<td>Relational Negation</td>
</tr>
<tr>
<td>OSA</td>
<td>Shift Assignment by Arithmetic Assignment</td>
</tr>
<tr>
<td>OSA</td>
<td>Shift Assignment by Logical Assignment</td>
</tr>
<tr>
<td>OSN</td>
<td>Shift Assignment by Negation</td>
</tr>
<tr>
<td>OSNR</td>
<td>Shift Assignment by Relational Assignment</td>
</tr>
<tr>
<td>OSNA</td>
<td>Shift Assignment by Shift Assignment</td>
</tr>
<tr>
<td>OSCA</td>
<td>Cast Assignment by Arithmetic Assignment</td>
</tr>
<tr>
<td>OSCA</td>
<td>Cast Assignment by Logical Assignment</td>
</tr>
<tr>
<td>OSNC</td>
<td>Cast Assignment by Negation</td>
</tr>
<tr>
<td>OSNR</td>
<td>Cast Assignment by Relational Assignment</td>
</tr>
<tr>
<td>OSNA</td>
<td>Cast Assignment by Shift Assignment</td>
</tr>
<tr>
<td>OSNA</td>
<td>Cast Assignment by Precedence Assignment</td>
</tr>
</tbody>
</table>

### Table 1: Constant Class Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Con</td>
<td>Constant for Constant Replacement</td>
</tr>
<tr>
<td>Csr</td>
<td>Constant for Scalar Replacement</td>
</tr>
<tr>
<td>CRCA</td>
<td>Required Constant Replacement</td>
</tr>
</tbody>
</table>

### Table 2: Statement Class Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRBC</td>
<td>Break Replacement by continue</td>
</tr>
<tr>
<td>SRBN</td>
<td>Break Out to Nth level</td>
</tr>
<tr>
<td>SDR</td>
<td>Continue Replacement by break</td>
</tr>
<tr>
<td>SDRN</td>
<td>Continue Out to Nth level</td>
</tr>
<tr>
<td>SDWR</td>
<td>do-while Replacement by while</td>
</tr>
<tr>
<td>SGLR</td>
<td>goto Label Replacement</td>
</tr>
<tr>
<td>SMTC</td>
<td>n-top continue</td>
</tr>
<tr>
<td>SMTC</td>
<td>n-top continue</td>
</tr>
<tr>
<td>SMTR</td>
<td>Move Brace Up and Down</td>
</tr>
<tr>
<td>SNRS</td>
<td>return Replacement</td>
</tr>
<tr>
<td>SDCL</td>
<td>Statement Deletion</td>
</tr>
<tr>
<td>SSWM</td>
<td>switch Statement Mutation</td>
</tr>
<tr>
<td>STRI</td>
<td>Trap on if Condition</td>
</tr>
<tr>
<td>SSTR</td>
<td>Trap on Statement Execution</td>
</tr>
<tr>
<td>SSWD</td>
<td>while Replacement by do-while</td>
</tr>
</tbody>
</table>
**Mutation Testing Procedure (1)**

- Given $P$ and a test set $T$
  - Generate mutants
  - Compile $P$ and the mutants
  - Execute $P$ and the mutants on each test case
  - Determine equivalent mutants
  - Determine mutation score
  - If mutation score is not 1 then improve the test case and repeat from Step 3

**Mutation Testing Procedure (2)**

- In practice the above procedure is implemented incrementally

- One applies a few selected mutant operators to $P$ and computes the mutation score with respect to the mutants generated

- Once these mutants have been distinguished or proven equivalent, another set of mutant operators is applied
Mutation Testing Procedure (3)

- This procedure is repeated until either all the mutants have been exhausted or some external condition forces testing to stop.

Tools for Mutation Testing

- Mothra: for Fortran, developed at Purdue, 1990
- Proteum: for C, developed at the University of São Paulo at São Paulo in Brazil.
Comparison Criterion

Mutation Testing (© 2013 Professor W. Eric Wong, The University of Texas at Dallas)

Mutation Hypothesis

- More difficulty to satisfy
- More expensive
- More effective in fault detection
**Subsumption**

![Subsumption Diagram]

**Program Classification**

- SDSU (single definition, single use)
- SDMU (single definition, multiple uses)
- MDSU (multiple definitions, single use)
- MDMU (multiple definitions, multiple uses)
**Program Classification**

- SDSU
  - M (Mothra) subsumes AU, CU, and PU
- SDMU
  - M (Mothra) subsumes AU, CU, and PU
- MDSU (multiple definitions, single use)
  - M (Mothra) subsumes CU, but not PU, and AU
- MDMU
  - M (Mothra) does not subsume CU, PU, and AU

**SDSU (1)**

\[
\begin{align*}
  s_i: & \quad x = \text{expr} \\
  s_j: & \quad y = f(x, \cdots) \\
  & \quad \text{a c-use of } x \text{ at } s_j
\end{align*}
\]

Case 1: Suppose that \((d_i(x), u_j(x))\) is a c-use pair. For a mutant obtained by mutating the \text{expr} at \(s_i\) to be distinguished, the following three conditions must be satisfied: (1) \(s_i\) is reached (reachability condition), (2) a state change occurs immediately after some execution of \(s_i\) (necessity condition), and (3) this state change propagates to the output via the only use of \(x\) at \(s_j\) (sufficiency condition). These conditions imply that the c-use \((d_i(x), u_j(x))\) pair is also covered as a consequence of distinguishing a mutant at \(s_i\). Hence, the c-use must be covered by a test case that distinguishes such a mutant at \(s_i\).
**SDSU (2)**

Case II: Next, suppose that $(s_i, s_j(x))$ is a p-use pair. Let $s_k$ and $x, 1 \leq k, l \leq u$, denote the successors of $s_j$. The reachability condition of distinguishing the true mutant at $s_j$ requires $s_i$ to be executed. If $s_k$ is not executed without having $s_i$ executed first, then $x$ is undefined at $s_j$.

A reference to such an $x$ at $s_j$ makes $P$ behave incorrectly which is contrary to our assumption that $P$ behaves correctly on a mutation adequate test set. Hence, $s_i$ must have been executed before $s_j$.

Immediately after execution of $s_i$, either $s_k$ or $s_l$ must have been executed. Suppose, without loss of generality, that $s_k$ was executed. Thus, distinguishing the true mutant at $s_j$ causes the path containing $s_i, s_j$, and $s_k$ to be executed in that order. Similarly, distinguishing the false mutant at $s_j$ causes the execution of a path containing $s_i, s_j$, and $s_l$ in that order. Hence, the p-use must have been covered by the test cases that distinguished the true and false mutants at $s_j$.

**SDMU (1)**

Figure 2: Structure of programs with one definition and multiple uses of $x$. Uses of $x$ in $f$ and $pred$ are, respectively, computational and predicate uses.

Figure 2, let $x \in V$ be defined at statement $s_i$ and used at $u, u > 1$, statements $s_j, s_{j_2}, \ldots, s_{j_u}$. This structure leads to $u$ def-use pairs any of which could be a c-use or a p-use pair. As before, below we distinguish between these two types of pairs.
**SDMU (2)**

Case I: Consider the c-use pair \((d_i(z), u_{ik}(x))\), for some \(k, 1 \leq k \leq u\). Given that a mutant at \(s_i\) has been distinguished by some test case \(t\) in the mutation adequate test set for this program implies that control must have reached \(s_i\). However, when control reaches \(s_i\), \(x\) is used and hence must have been defined prior to control reaching \(s_i\). In case it was not then \(P(t)\) would be incorrect which is contrary to our assumption. As there is only one definition of \(x\), \(s_i\) must have been executed prior to the execution of \(s_i\) thereby covering the c-use pair. As this argument applies to any program variable having a c-use, we have shown that all c-uses in \(P\) are covered by a mutation adequate test set.

**SDMU (3)**

Case II: Let \((d_i(x), u_{ik}(x))\), for some \(k, 1 \leq k \leq u\), be a p-use pair. The arguments used in Case II in the proof of Theorem 1 are applicable to this case also. Thus, distinguishing the true and false mutants at \(s_i\) guarantees the coverage of this p-use. This argument is valid for all variables in \(P\) and hence we have shown that all p-uses in \(P\) are covered by a mutation adequate test set.
MDSU (1)

- Lemma 1 *For Category III, M_R subsumes CU*

- Proof: The proof follows from the arguments used in Case I of the proof of Theorem 1 applied to all c-use pairs

MDSU (2)

```plaintext
program P_1
begin
integer a, b, x, y, z;
read a, b, y, z;
if (a > 1) then
    x := 2;
else
    x := 5;
endif
if (x = b) then
    print y;
else
    print z;
endif
end
```

Figure 3: A program in category MDSU that has a mutation adequate test set but not p-use adequate.
MDSU (3)

Table 1: A mutation but not p-use adequate test set of \( P_1 \)

<table>
<thead>
<tr>
<th>Test case number</th>
<th>((a, b, y, z)) values</th>
<th>Test case number</th>
<th>((a, b, y, z)) values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((2, 3, 5, 6))</td>
<td>7</td>
<td>((1, 5, 7, 6))</td>
</tr>
<tr>
<td>2</td>
<td>((2, 1, 5, 6))</td>
<td>8</td>
<td>((3, 4, 5, 3))</td>
</tr>
<tr>
<td>3</td>
<td>((2, 5, 5, 6))</td>
<td>9</td>
<td>((2, 2, 5, 6))</td>
</tr>
<tr>
<td>4</td>
<td>((1, 6, 5, 6))</td>
<td>10</td>
<td>((1, 1, 5, 6))</td>
</tr>
<tr>
<td>5</td>
<td>((1, 4, 5, 6))</td>
<td>11</td>
<td>((3, 2, 7, 6))</td>
</tr>
<tr>
<td>6</td>
<td>((1, 2, 2, 6))</td>
<td>12</td>
<td>((1, 3, 7, 6))</td>
</tr>
</tbody>
</table>

Proof: Figure 3 shows a program that has two p-use pairs for variable \( x \). Table 1 lists a mutation adequate test set which does not cover the p-use pair consisting of the definition \( x := 2 \) and its use in the predicate \( x = b \) because the successor print \( y \) is not executed.

Empirical Study: Subsumption

• All-uses scores using mutation adequate test sets are, in general, higher than the mutation scores using all-uses adequate test sets.
Conclusion on Subsumption

Cost Metrics

- Number of executions
- Number of test cases
- Test case generation
- Learning testing tools
- Identifying equivalent mutants & infeasible all-uses
Reducing Mutation Cost

- The cost of mutation testing can be reduced if the number of mutants to be examined is reduced

Selection Mutation

- Select proper mutant operators
  - ror: relational operator replacement
  - lcr: logical connector replacement
  - abs: absolute value insertion
  - sdl: statement deletion
**abs Mutant Operator**

\[ a := b + c \]

- \[ a := |b| + c \]
- \[ a := -|b| + c \]
- \[ a := 0 + c \]

**ror Mutant Operator**

*relational operator replacement*

\[ a < b + c \]

- \[ > \]
- \[ >= \]
- \[ <= \]
- \[ != \]
- \[ true \]
- \[ false \]
**Random x% Mutation**

- Randomly select a small percentage of mutants from each mutant type

---

**Weak Mutation (1)**

- **Reachability**
  - Execute the mutated statement

- **Necessity**
  - Make a state change

- **Sufficiency**
  - Propagate the change to output
**Weak Mutation - Advantage**

- Weak mutation *reduces the amount of execution* for distinguishing each mutant

**Weak Mutation - Disadvantage**

- The disadvantage of weak mutation testing is that there is *no guarantee* that the different immediate effect will cause a different final result.
Weak Mutation (2)

- Weak mutation is as effective as strong mutation if the weak mutation hypothesis is true:
  - (reachability and necessity) $\rightarrow$ sufficiency
- Experiments have shown that this is true for 61% of all the cases studied

Weak Mutation (3)

- Suppose that the weak mutation hypothesis does not hold for a particular fault (say $F$). That is, there exists a non-empty input set that satisfies the reachability and necessity conditions while not producing a detectable failure.

- But in the code under test, there will be many locations with potential faults, each with its own reachability and necessity conditions. It may be that satisfying those other conditions will force the execution of this fault (i.e., $F$) in a way that must produce a detectable failure (namely to satisfy the sufficiency condition).

- Thus, the weak mutation hypothesis may not hold when a single fault is considered alone, but may hold when the fault is considered as part of a larger program (which has many faults).
Reduction Measurement (1)

- Size reduction:
  \[
  1 - \frac{\text{Average size of test sets adequate with respect to alternate mutation}}{\text{Average size of mutation adequate test sets}}
  \]

- Expense reduction:
  \[
  1 - \frac{\text{Total number of mutants examined when using alternate mutation}}{\text{Total number of mutants examined in mutation}}
  \]

Reduction Measurement (2)

- Mutation score reduction
- All-uses scores reduction
**Observation**

- Compared to mutation, randomly selected $x\%$ mutation and $abs/rror$ mutation provide:
  - Significant size reduction
  - Significant expense reduction
  - Small reduction on mutation scores
  - Small reduction on all-uses scores
Incremental Testing Strategy

- Fewer mutant operators
- All-uses
- Mutation
- More mutant operators

Cost Differential
Error Detection Effectiveness

Mutation Testing (© 2013 Professor W. Eric Wong, The University of Texas at Dallas)