Test Adequacy Assessment: Program Mutation

Chungnam Nat’l Univ.
Dept. of Computer Science & Engineering
Prof. Kim, Hyeon Soo
What is program mutation?

- Suppose that program $P$ has been tested against a test set $T$ and $P$ has not failed on any test case in $T$.
  - Now suppose we do the following:

  Changed to

  $$P \quad \rightarrow \quad P'$$

- What behavior do you expect from $P'$ against tests in $T$?
What is program mutation?

- $P'$ is known as a \textbf{mutant} of $P$.

- There might be a test $t$ in $T$ such that $P(t) \neq P'(t)$
  - In this case we say that $t$ \textit{distinguishes} $P'$ from $P$
  - Or, that $t$ has \textit{killed} $P'$

- There might be \textit{not} be any test $t$ in $T$ such that $P(t) \neq P'(t)$
  - In this case we say that $T$ is \textit{unable to distinguish} $P$ and $P'$
  - Hence $P'$ is considered \textit{live} in the test process
What is program mutation?

- If there does not exist any test case \( t \) in the **input domain** of \( P \) that distinguishes \( P \) from \( P' \)
  - then \( P' \) is said to be **equivalent** to \( P \).

- If \( P' \) is **not equivalent** to \( P \) but no test in \( T \) is able to distinguish it from \( P \)
  - then \( T \) is considered **inadequate**

- A non-equivalent and live mutant offers the tester an opportunity to generate a new test case and hence **enhance** \( T \).

- *We will refer to program mutation as mutation.*
Test adequacy using mutation

Given a test set T for program P that must meet requirements R, a test adequacy assessment procedure proceeds as follows.

- **Step 1:** Create a set M of mutants of P
  
  Let $M = \{ M_0, M_1, \ldots M_k \}$. Note that we have k mutants.

- **Step 2:** For each mutant $M_i$ find if there exists a $t$ in T such that $M_i(t) \neq P(t)$
  
  If such a $t$ exists then $M_i$ is considered killed and removed from further consideration.

- **Step 3:** At the end of Step 2 suppose that $k_1 \leq k$ mutants have been killed and $(k-k_1)$ mutants are live.
  
  - Case 1: $(k-k_1)=0$: T is adequate with respect to mutation.
  - Case 2: $(k-k_1)>0$ then we compute the mutation score (MS) as follows:
    
    $$MS = \frac{k_1}{k-e}$$
    
    Where $e$ is the number of equivalent mutants. Note: $e \leq (k-k_1)$.
Test enhancement using mutation

• One has the opportunity to enhance a test set $T$ after having assessed its adequacy.
  - **Step 1:** If the mutation score ($MS$) is 1, then some other technique, or a different set of mutants, needs to be used to help enhance $T$.
  - **Step 2:** If $MS$ is less than 1, then there exist live mutants that are not equivalent to $P$. Each live mutant needs to be distinguished from $P$.
  - **Step 3:** A new test $t$ is designed with the objective of distinguishing at least one of the live mutants; let us say this is mutant $m$.
  - **Step 4:** If $t$ does not distinguish $m$ then another test $t$ needs to be designed to distinguish $m$. Suppose that $t$ does distinguish $m$.
  - **Step 5:** It is also possible that $t$ also distinguishes other live mutants.
  - **Step 6:** One now adds $t$ to $T$ and re-computes $MS$.
  - Repeat the enhancement process from Step 1.
Error detection using mutation

- There is no guarantee that tests derived to distinguish live mutants will reveal a yet undiscovered error in P.

- Consider the following function `foo` that is required to return the sum of two integers x and y.
  - Clearly `foo` is incorrect

```c
int foo(int x, y){
    return (x-y); ← This should be return (x+y)
}
```
Error detection using mutation

Now suppose that foo has been tested using a test set T that contains two tests:

\[ T=\{ t1: <x=1, y=0>, t2: <x=-1, y=0>\} \]

First note that foo behaves perfectly fine on each test in
- i.e. foo returns the expected value for each test case in T
- Also, T is adequate with respect to all control and data flow based test adequacy criteria.
Error detection using mutation

Let us evaluate the adequacy of T using mutation

Suppose that the following three mutants are generated from foo.

M1: int foo(int x, y){
    return (x+y);
}

M2: int foo(int x, y){
    return (x-0);
}

M3: int foo(int x, y){
    return (0+y);
}

Note that

- M1 is obtained by replacing the - operator by a + operator
- M2 by replacing y by 0
- M3 by replacing x by 0.
Error detection using mutation

• Next we execute each mutant against tests in T
  - until the mutant is distinguished or we have exhausted all tests
• Here is what we get
  - $T = \{ t_1: <x=1, y=0>, t_2: <x=-1, y=0> \}$

<table>
<thead>
<tr>
<th>Test (t)</th>
<th>foo(t)</th>
<th>M1(t)</th>
<th>M2(t)</th>
<th>M3(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Live</th>
<th>Live</th>
<th>Killed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test (t)</td>
<td>foo(t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>
Error detection using mutation

- After executing all three mutants
  - we find that two are live and one is distinguished
  - Computation of mutation score requires us to determine if any of the live mutants is equivalent.

- In class exercise: Determine whether or not the two live mutants are equivalent to foo and compute the mutation score of T.

No MS = 1/(3-0) = 0.33
Error detection using mutation

• Let us examine the following two live mutants.

\[
\begin{align*}
\text{M1: } \quad & \text{int foo(int } x, y)\{ \\
& \text{return } (x+y); \\
& \}
\end{align*}
\]

\[
\begin{align*}
\text{M2: } \quad & \text{int foo(int } x, y)\{ \\
& \text{return } (x-0); \\
& \}
\end{align*}
\]

• Let us focus on M1
  - A test that distinguishes M1 from foo must satisfy the following condition:
    • \( x-y \neq x+y \) implies \( y \neq 0 \)
  - Hence we get t3: \( <x=1, y=1> \)
Error detection using mutation

- Executing \texttt{foo} on \( t_3 \) gives us \( \texttt{foo}(t_3) = 0 \)
  - However, according to the requirements we must get \( \texttt{foo}(t_3) = 2 \)
  - Thus \( t_3 \) distinguishes \( M_1 \) from \texttt{foo} and also reveals the error

\begin{verbatim}
M1: int foo(int x, y){
    return (x+y);
}
M2: int foo(int x, y){
    return (x-0);
}
\end{verbatim}

- \textit{In class exercise:} (a) Will any test that distinguishes also reveal the error? (b) Will any test that distinguishes \( M_2 \) reveal the error?

  (a) Yes (b) \( x-y \neq x-0 \) implies \( y \neq 0 \), yes
Guaranteed error detection

Sometimes there exists a mutant $P'$ of program $P$ such that any test $t$ that distinguishes $P'$ from $P$ also causes $P$ to fail.

More formally:
- Let $P'$ be a mutant of $P$ and $t$ be a test in the input domain of $P$.
- We say that $P'$ is an error revealing mutant if the following condition holds for any $t$.
  - $P'(t) \neq P(t)$
  - $P(t) \neq R(t)$, where $R(t)$ is the expected response of $P$ based on its requirements.

Is $M_1$ in the previous example an error revealing mutant?

What about $M_2$?
Distinguishing a mutant

A test case \( t \) that distinguishes a mutant \( m \) from its parent program \( P \) must satisfy the following three conditions:

- **Condition 1:** (Reachability) \( t \) must cause \( m \) to follow a path that arrives at the mutated statement in \( m \).

- **Condition 2:** (Infection) If \( S_{in} \) is the state of the mutant upon arrival at the mutant statement and \( S_{out} \) is the state soon after the execution of the mutated statement, then \( S_{in} \neq S_{out} \).

- **Condition 3:** (Propagation) If difference between \( S_{in} \) and \( S_{out} \) must propagate to the output of \( m \) such that the output of \( m \) is different from that of \( P \).

**Exercise:** Show that in the previous example both \( t_1 \) and \( t_2 \) satisfy the above three conditions for \( M3 \).
Equivalent mutants

• The problem of deciding whether or not a mutant is equivalent to its parent program is undecidable.
  - Hence there is no way to fully automate the detection of equivalent mutants.

• The number of equivalent mutants can vary from one program to another.
  - However, empirical studies have shown that one can expect about 5% of the generated mutants to be equivalent to the parent program.

• Identifying equivalent mutants is generally a manual and often time consuming process.
A misconception

- There is a widespread misconception that any “coverage” based technique, including mutation, will not be able to detect errors due to missing path.

- Consider the following programs.

**Program under test**
```c
int foo(int x, y){
  int p=0;
  if(x<y)
    p=p+1;
  return(x+p*y)
}
```

**Correct program**
```c
int foo(int x, y){
  int p=0;
  if(x<y)
    p=p+1;
  else
    p=p-1;
  return(x+p*y)
}
```
A misconception

• Suggest at least one mutant M of foo that is guaranteed to reveal the error (M is an error revealing mutant).

• Suppose T is decision adequate for foo
  - Is T guaranteed to reveal the error?

• Suppose T is def-use adequate for foo
  - Is T guaranteed to reveal the error?
Mutant operators

- A mutant operator $O$ – a function that maps the program under test to a set of $k$ (zero or more) mutants of $P$
Mutant operators

- A mutant operator creates mutants by making simple changes in the program under test.
  - e.g.
    - “variable replacement” mutant operator
      - replaces a variable name by another variable declared in the program
    - “relational operator replacement” mutant operator
      - replaces relational operator with another relational operator
## Mutant operators: Examples

<table>
<thead>
<tr>
<th>Mutant operator</th>
<th>In P</th>
<th>In mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable replacement</td>
<td>z=x*y+1;</td>
<td>x=x<em>y+1; z=x</em>x+1;</td>
</tr>
<tr>
<td>Relational operator replacement</td>
<td>if (x&lt;y)</td>
<td>if(x&gt;y) if(x&lt;=y)</td>
</tr>
<tr>
<td>Off-by-1</td>
<td>z=x*y+1;</td>
<td>z=x*(y+1)+1; z=(x+1)*y+1;</td>
</tr>
<tr>
<td>Replacement by 0</td>
<td>z=x*y+1;</td>
<td>z=0*y+1; z=0;</td>
</tr>
<tr>
<td>Arithmetic operator replacement</td>
<td>z=x*y+1;</td>
<td>z=x*y-1; z=x+y-1;</td>
</tr>
</tbody>
</table>
Mutants: First order and higher order

• A mutant obtained by making exactly “one change”
  – considered first order

• A mutant obtained by making two changes
  – a second order mutant

• Similarly higher order mutants can be defined
  – For example, a second order mutant of \( z = x + y \); is \( x = z + y \);
  – where the variable replacement operator has been applied twice

• In practice only first order mutants are generated for two reasons:
  – (a) to lower the cost of testing
  – (b) most higher order mutants are killed by tests adequate with respect to first order mutants [See coupling effect later]
Mutant operators: basis

- A mutant operator models a simple mistake that could be made by a programmer

- Several error studies have revealed that programmers--novice and experts--make simple mistakes
  - For example, instead of using $x<y+1$ one might use $x<y$

- While programmers make “complex mistakes” too, mutant operators model simple mistakes.
  - because of “coupling effect”
Mutant operators: Goodness

• The design of mutation operators is based on guidelines and experience
  - It is thus evident that two groups might arrive at a different set of mutation operators for the same programming language

• How should we judge whether or not that a set of mutation operators is “good enough?”

• Informal definition:
  - Let S1 and S2 denote two sets of mutation operators for language L.
  - Based on the effectiveness criteria, we say that S1 is superior to S2 if mutants generated using S1 guarantee a larger number of errors detected over a set of erroneous programs.
Mutant operators: Goodness

• Generally one uses a small set of highly effective mutation operators rather than the complete set of operators.

• Experiments have revealed relatively small sets of mutation operators for C and Fortran.
  – We say that one is using “constrained” or “selective” mutation when one uses this small set of mutation operators.
Mutant operators: Language dependence

- For each programming language one develops a set of mutant operators.

- Languages differ in their syntax thereby offering opportunities for making mistakes that differ between two languages.
  - This leads to differences in the set of mutant operators for two languages.

- Mutant operators have been developed for languages such as Fortran, C, Ada, Lisp, and Java.
  - [See the textbook for a comparison of mutant operators across several languages.]
Competent programmer hypothesis (CPH)

- CPH states that given a problem statement, a programmer writes a program P that is in the general neighborhood of the set of correct programs.
  - An extreme interpretation of CPH is that when asked to write a program to find the account balance, given an account number, a programmer is unlikely to write a program that deposits money into an account.
  - Of course, while such a situation is unlikely to arise, a devious programmer might certainly write such a program.
Competent programmer hypothesis (CPH)

- A more reasonable interpretation of the CPH is that
  - the program written to satisfy a set of requirements will be a few mutants away from a correct program.

- The CPH assumes that
  - the programmer knows of an algorithm to solve the problem at hand
  - if not, he will find one prior to writing the program

- Mistakes will lead to a program that can be corrected by applying one or more first order mutations.
Coupling effect

• Coupling effect:
  – “Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors”

• It is during an analysis of the behavior of the mutant in relation to that of its parent that one discovers complex faults.
Tools for mutation testing

• There are few mutation testing tools available freely
  – Proteum for C from Professor Maldonado
  – muJava for Java from Professor Jeff Offutt
Tools for mutation testing: Features

- A typical tool for mutation testing offers the following features:
  - A selectable palette of mutation operators.
  - Management of test set T.
  - Execution of the program under test against T and saving the output for comparison against that of mutants.
  - Generation of mutants.
  - Mutant execution and computation of mutation score using user identified equivalent mutants.
  - Incremental mutation testing
    - i.e. allows the application of a subset of mutation operators to a portion of the program under test.
Mutation and system testing

- Adequacy assessment using mutation is often recommended only for relatively small units
  - e.g. a class in Java or a small collection of functions in C
- However, given a good tool, one can use mutation to assess adequacy of system tests.
Mutation and system testing

- The procedure is recommended to assess the adequacy of system tests:
  - **Step 1**: Identify a set $U$ of application units that are critical to the safe and secure functioning of the application.
    - Repeat the following steps for each unit in $U$.
  - **Step 2**: Select a small set of mutation operators.
    - This selection is best guided by the operators defined by Eric Wong or Jeff Offutt. [See the textbook for details.]
  - **Step 3**: Apply the operators to the selected unit.
  - **Step 4**: Assess the adequacy of $T$ using the mutants so generated.
    - If necessary, enhance $T$.
  - **Step 5**: Repeat Steps 3 and 4 for the next unit until all units have been considered.