Test Adequacy Assessment using Program Mutation

Dr. Iichul Yoon (icyoon@sunny.korea.ac.kr)

Sources: Foundations of Software Testing (textbook)

Mutation Operators

- A mutation operator $O$ is a function that maps the program under test to a set of $k$ (zero or more) mutants of $P$.

\[ O(P) \rightarrow M1 \rightarrow M2 \rightarrow \ldots \rightarrow Mk \]

- A mutation operator creates mutants by making simple changes in the program under test.
  - e.g., variable replacement, relational operator replacement
  - In practice, first-order mutants are used. Why?

Mutation Operators: Examples

<table>
<thead>
<tr>
<th>Mutation operator</th>
<th>In P</th>
<th>In mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable replacement</td>
<td>$z = x \times y + 1$</td>
<td>$x = x \times y + 1$; $z = x \times x + 1$</td>
</tr>
<tr>
<td>Relational operator replacement</td>
<td>$\text{if } (x&lt;y)$</td>
<td>$\text{if } (x&gt;y)$; $\text{if } (x \leq y)$</td>
</tr>
<tr>
<td>Off-by-1</td>
<td>$z = x \times y + 1$</td>
<td>$z = x \times (y + 1) + 1$; $z = (x + 1) \times y + 1$</td>
</tr>
<tr>
<td>Replacement by 0</td>
<td>$z = x \times y + 1$</td>
<td>$z = 0 \times y + 1$; $z = 0$</td>
</tr>
<tr>
<td>Arithmetic operator replacement</td>
<td>$z = x \times y + 1$</td>
<td>$z = x \times y - 1$; $z = x \times y - 1$</td>
</tr>
</tbody>
</table>

Mutation Operators: Basis

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

- Error studies have revealed that programmers make simple mistakes.
  - e.g., instead of using $x < y + 1$, one might use $x < y$.

- Rely on coupling effect for “complex mistakes”
  - Test data that distinguishes all (faulty) programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors
  - During an analysis of the mutant behavior in relation to that of its parent, tester discovers complex faults.
Mutation Operators: Goodness

- Guidelines and experience are important in the design of mutation operators
  - How should we judge whether or not that a set of mutation operators is "good enough?" or "better"?

- 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.

Mutation 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.

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.
  - i.e., the program written to satisfy a set of requirements will be a few mutants away from a correct program, and
  - Can be corrected by a series of simple mutations
Fault Detection using Mutation

Let’s take another example.

Example 7.8.1: Consider a function named misCond. It takes zero or more sequence of integers in array data as input. It is required to sum all integers in the sequence starting from the first integer and terminating at the first 0. Thus, for example, if the input sequence is (6 5 0), then the program must output 11. For the sequence (6 5 0 4), misCond must also output 11.

Here is the code and an example

```c
1 int misCond(data, link, P);
2 int data[10], link[10], F;
3 int i;
4 int L;
5 int T;
6 int -1;
7 int 0;
8 int 1;
9 int L;
10 return(0);
```

Line 7 has to be:

```c
1 int misCond(data, link, P);
2 int data[10], link[10], F;
3 int i;
4 int L;
5 int T;
6 int -1;
7 int 0;
8 int 1;
9 int L;
10 return(0);
```

A mutant (line 9 is changed)

```c
1 if (data[F] ≠ 0) sum = sum + data[L];
```

Type of Mutants

Let P’ be a mutant of P and t a test in the input domain D of P. We say that P’ is an error revealing mutant if the following condition holds.

Error-revealing mutants

For any t ∈ D such that P’(t) ≠ P(t), P(t) ≠ R(t), where R(t) is the expected response of P based on its requirements.

Error-hinting mutants

P’ is equivalent to P and P’(t) ≠ R(t)

Reliability indicating mutants

P’(t) ≠ P(t) for some t ∈ D and P(t) = R(t)

Mutation Operators for C

Examples are implemented in Proteum

What are not mutated?

- Declarations
- The address operator (&)
- Format strings in I/O functions
- Function declaration headers
- Control line
- Function name indicating a function call
- Parameters are mutated but not the function name (e.g., scanf)
- Preprocessor conditionals
Mutation Operators for C: Examples

- Required constant replacement
  - Use the set \( \{0, 1, -1, u_i\} \) where \( u_i \) denotes user-specified integer
  - Reference to an entity via a pointer is replaced by \textit{null}

- Example
  - Consider a statement \( k = j + \ast p \), where \( k \) and \( j \) are integers and \( p \) is a pointer to an integer.
  - Mutants
    - \( k = 0 + \ast p \)
    - \( k = 1 + \ast p \)
    - \( k = -1 + \ast p \)
    - \( k = u_i + \ast p \)
    - \( K = j + \text{null} \)

Tools for Mutation Testing

- Examples
  - \textit{Proteum for C}
  - \textit{muJava for Java}

- Features
  - Mutation operators, mutant generation and execution
  - Test set management
  - Execution of the SUT against \( T \) and saving the output for comparison against that of mutants.
  - Mutation score computation 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 Operators for C: Examples

- Trap on statement execution
  - Intended to reveal unreachable code
  - Replace each statement by \texttt{trap_on_statement()}
    - Mutant execution terminates when trap statement is executed.

- Example

```
while (i <= y)
    { if (x < y)
        y = x;
    else
        x = y;
    }
```

Mutation and System Testing

- Mutation can be used to assess system test adequacy

  - Recommended procedure
    - \textbf{Step 1}: Identify a set \( U \) of application units critical to the safe and secure functioning of the application. Repeat \textbf{Step 2-4} for each in \( U \).
    - \textbf{Step 2}: Select a small set of (effective) mutation operators.
    - \textbf{Step 3}: Apply the operators to the selected unit.
    - \textbf{Step 4}: Assess the adequacy of \( T \) using the mutants generated. If necessary, enhance \( T \).

- Mutation testing is needed for applications with stringent availability, security, safety requirements.
Summary

- Mutation testing is the most powerful technique for the assessment and enhancement of tests.
  - Mutation must be applied incrementally and with assistance from good tools.
  - Often recommended for unit testing, but can be used for assessing and enhancing system tests.
  - Highly recommended, if the top QA goals of the system are high availability, security, and safety.

- Identification of equivalent mutants is an undecidable problem
  - Similar to the identification of infeasible paths in control or data flow based test assessment.