Mutation testing

I am grateful to Kostas Adamopoulos for his permission to use slides he prepared about mutation testing as part of this week’s slides.

Faults and Failures

- **Fault**: is the group of incorrect statements in the program that causes a failure.

- **Failure**: is an external, incorrect behaviour of a program (i.e. an incorrect output or a runtime failure).
MT is based on two basic Assumptions

• The Competent Programmer Hypothesis:
  In general programmers are competent. That is, the programs they write are “nearly” correct. The program differs from a correct version in only a few small ways.

• The Coupling Effect Hypothesis:
  Large program faults, particularly those of a semantic nature are “coupled” with smaller syntactic faults that can be detected with mutation testing

  (Hypothesised 1978, supported empirically 1992, demonstrated theoretically 1995, but is it true?).

What is Mutation Testing

• White-box, error-based testing technique
• Build-in adequacy criteria
  – The quality of a test set is measured according to its effectiveness or ability to detect faults
• Tool support
• Goal is generating good sets of tests rather than finding faults
The Idea Underpinning Mutation Testing

• Seeding the implementation with a fault (mutating the original program) by applying a mutation operator
• Then determine whether testing identifies this fault
• Different result:
  – the fault introduced has been identified
  – If the test case distinguishes between the mutant and the original program
  – it is said to kill the mutant
• Same result:
  – the mutant and the original program produce identical results
  – the mutant is still alive

How it works (again)

If P and P’ give different results, then mutant P’ is killed by test case T, so T can detect the difference between the correct and buggy program...

If P and P’ give the same results then either

• Test case T not good enough to detect the fault... so we need to come with a better test

• Or P and P’ are equivalent programs... P’ is an equivalent mutant, no tests can be devised that can distinguish them

so... this is mainly a way to judge the effectiveness of our test data, to improve them, and by doing that to detect faults (syntactic & semantic ones)
Test Data Effectiveness

• Mutation testing provides a way to judge the effectiveness of the test data
  – The test set should kill all the mutants
  – If not, then we can improve the test set

• Test generation may be based on mutation testing
  – Tests are generated to kill the mutants

• In the same way mutants should be of high performance
  – Difficult to be eliminated

Mutation Score (MS)

• Mutation Score (Adequacy Score) of Program P and test set T is:
  \[ MS(P,T) = \frac{\text{# of Killed Mutants}}{\text{# of Non-Equivalent Mutants}} \]

Where, Non-Equivalent Mutants = Total Mutants – Equivalent Mutants

\[ 0 <= MS <= 1 \quad \text{or} \quad 0 <= MS\% <= 100 \]

Test data is mutation-adequate if its mutation score is 100%
(in this case they kill all non-equivalent mutants)
Mutants

• Original Program P
• Mutant P’ of P
  – A program similar to P
  – P’ differs from P by a single mutation
  – Each kind of mutation corresponds to a typical error programmers usually make
    • “Off-by-one”, spelling, typos, etc.
    • i.e. imagine that P’ was identical to P except that exactly one + was changed to a *

Mutation Operators
(Also called: mutant operator, mutagenic operator, mutagen, mutation transformation, mutation rule)

• It is a rule that is applied to a program to create mutants
  – Replace each operand by every other syntactically legal operand
  – Modify expressions by replacing operators and inserting new operators
  – Delete entire statements, etc.
• Categorised into mutation classes
  – Statement, Operator, Variable, Constant, etc.
Examples of Mutation Operators

Some Mothra Mutation Operators for Fortran (from a total of 22 operators)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aar</td>
<td>array for array replacement</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value insertion</td>
</tr>
<tr>
<td>car</td>
<td>constant for array replacement</td>
</tr>
<tr>
<td>crp</td>
<td>constant replacement</td>
</tr>
<tr>
<td>sdl</td>
<td>statement deletion</td>
</tr>
<tr>
<td>uoi</td>
<td>unary operation insertion</td>
</tr>
</tbody>
</table>

Proteum has 71 Mutation Operators for C categorised as follows: Statement 15, Operator 46, Variable 7, Constant 3

Creating a Mutant

Apply a Mutation Operator: Change + to *

Original Program $P$

$\cdots \cdots$
$\cdots \cdots$
$c := a + b$ ;
$\cdots \cdots$

Mutant $P'$ of Original Program $P$

$\cdots \cdots$
$\cdots \cdots$
$c := a \ast b$ ;
$\cdots \cdots$
Testing a Mutant

Original Program P

\[ \text{Result } R \]

\[ c := a + b ; \]

Apply Test Case T to both P and P'

Mutant P'

\[ \text{Result } R' \]

\[ c := a * b ; \]

If \( R' \neq R \) then mutant P' is killed by Test Case T

If \( R' = R \) then Improve Test Case T and test again

Loop: If we continue improving test case T and still getting \( R' = R \) then P' is possibly an Equivalent Mutant

An Example of a Mutant

• Program P

\[ \ldots \]

\[ x = y + z \]

\[ \ldots \]

• A mutant P' of P

\[ \ldots \]

\[ x = y * z \]

\[ \ldots \]

• Test case1 \((y = 3, z = 1)\) kills the mutant

\[ X = 4 \quad X = 3 \]

• Test case2 \((y = 2, z = 2)\), the mutant is still live

\[ X = 4 \quad X = 4 \]
Equivalent Mutant Problem

• An equivalent mutant is syntactically different from the original program, but has the same behavior.
• The general problem of deciding whether a mutant is equivalent to the original program is theoretically undecidable.
• This is a hugely important obstacle which will need to be overcome to facilitate mutation testing in practice.

Example of an Equivalent Mutant

• Original Program P
  
  ....
  If (y==2 && z==2)
  x = y + z
  ....

    X = 4

• Mutant P’ of P
  
  ....
  If (y==2 && z==2)
  x = y * z
  ....

    X = 4

P’ is an equivalent mutant of P because no possible test can ever kill this mutant. If the condition is true the mutated statement returns x=4 for any possible test case, same as the original statement.
Equivalent Mutant Problem

• Determining whether a mutant is equivalent is not decidable
• How do we know whether a mutant which remains un killed is simply hard to kill (stubborn) or equivalent?
• Could we avoid generating equivalent mutants?

Large Number of Mutants
Large Number of Mutants

• Even for simple programs n can be a very large number
• N depends on the size of P and on how many mutation operators we apply on P
• Reducing the number of mutants is the second problem that needs to be addressed

Existing Methodologies to Reduce Large Number of Mutants

• Selective Mutation
  • Reduces the number of mutation operators applied
• Mutant Sampling
  • Randomly selects a subset of mutants
  
  The number of mutants under test is reduced
Advances in Mutation Testing

- Reduction technique: Selective Mutation
- Approximation technique: Weak Mutation
- Algorithmic execution technique: Schema-based Mutation
- Heuristics for detecting equivalent mutants
- Algorithms for automatic test data generation
- Interface Mutation, Class Mutation
- Distribution of computational expense
- Avoiding human intensiveness

Selective Mutation
(a “do fewer” approach)

- Applying mutation with only the most critical mutation operators being used – “key” operators
  - provide almost the same coverage as non-selective mutation
  - Select only mutants that are truly distinct from other mutants
  - decreases the number of mutants produced
  - reduces computational cost significantly
- Getting as much testing strength as possible with as few mutants as possible
Mutation Sampling
(a “do fewer” approach)

- Sampling only a randomly selected subset of the mutants to run
- Using samples of some a priori fixed size
- Using samples without a priori fixed size
  - select mutants until sufficient evidence has been collected to determine that a statistically appropriate sample size has been reached

Weak Mutation
(a “do smarter” approach)

- An approximation technique that compares the internal states of the mutant and the original program immediately after execution of the mutated portion of the program
- Reduces the computational cost, but do we really get what we want?
Weak Mutation Example

\[
x = x + y ; \\
\ldots \\
x = z ; \\
\text{Print} \ (x) ;
\]

\[
x = x * y ; \\
\ldots /\text{.inspect} \ x */ \\
x = z ; \\
\text{Print} \ (x) ;
\]

Using Distributed Computational Resources (a “do smarter” approach)

- Using novel computer architectures to distribute the computational expense over several machines
Using Intelligent Algorithms
(a “do smarter” approach)

- Intelligently storing state information, this technique factors the expense of running a mutant over several related mutant executions and thereby lowers the total computational cost

Schema-based Mutation
(a “do faster” approach)

- Not mutating an intermediate form
- The Mutant Schema Generation (MSG) method encodes all mutations into one source-level program, a “metamutant”
- This program is compiled (once), with the same compiler used during development and is executed in the same operational environment at compiled-program speeds
Example

\[ x = x + y \]

Switch (N)

Case 1: \( x = x \times y; \)
Case 2: \( x = x \div y; \)
Case 3: \( x = x \mod y; \)
Case 4: \( x = z \times y; \)
Case 5: ... 

Mutation Testing Tools

- **Mothra**: (for Fortran 77)
  - Downloadable: [http://www.isse.gmu.edu/~ofut/rsrch/mut.html](http://www.isse.gmu.edu/~ofut/rsrch/mut.html)
  - For UNIX systems
  - 22 mutation operators
  - Interpretive approach

- **Proteum**: PROgram TEsting Using Mutants (for C)
  - Proteum/IM, Proteum/IM 2.0, Proteum/FSM, Proteum/ST, Proteum/PN
  - Downloadable?
  - For UNIX systems
  - 71 operators
  - Separate compilation approach

- **Jester**: JUnit test tester (for Java)

- **Insure++**: (for C++)
  - Commercial product
Mothra

- Mothra is a suite of tools for performing mutation testing for Fortran 77
  - Interpretive execution
  - Mutgen: for generating mutants
  - A testing harness for running a test on a set of mutant programs and recording the results
  - Godzilla: for automatically generating test cases

Using Mothra

- Select and generate a set of mutants
- Generate an initial set of test cases and the corresponding outputs that they generate
- Confirm the outputs are correct
- Repeat until all mutants are killed:
  - Run the mutants on the test sets
  - Equivalent mutants
  - Generate and confirm new tests
- When you are done, you have an adequate suite of tests
Proteum Family Tools

- Proteum is a suite of tools for performing mutation testing for C programs
  - Unit testing (Proteum), Integration testing (Proteum/IM), both (Proteum/IM 2.0), Finite State Machines (Proteum/FSM), Statecharts Specifications (Proteum/ST), Petri Nets specification (Proteum/PN)
  - Test case handling (execution, inclusion/exclusion, etc), Mutant handling (creation, selection, execution, analysis), Adequate Analysis (mutation score and reports)
  - Allows separate compilation; each mutant is individually created, compiled, linked, and run
  - This approach can be significantly faster (15-20 times) than an interpretive system, if mutant run times greatly exceed individual compilation/link times… else compilation bottleneck may result

Fundamental premise of Mutation Testing

- In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects the fault.
Future Testing Systems

• Programmer submits a program unit
• System replies with a set of input/output pairs that are guaranteed to form an effective test of the unit by being close to mutation adequate

Current Research at King’s

• Kostas Adamopoulos is a PhD student working on Mutation Testing

• We are looking at search as a way of attacking the twin problems of number of mutants and equivalent mutants.

• This part of the lecture is not examinable. Feel free to leave now (quietly) if you are not interested.

• Make sure you come back for the tutorial though …

• Ok, for the two of you left, come to the front…
Co-evolution?

- GA for Mutants
  - Fitness is measured according to the ability to avoid being killed
  - If this ability is too high then penalize the fitness of this mutant because it probably an equivalent one

- GA for Test Cases
  - Fitness is measured according to the ability of killing mutants

Two competitive populations. Can this lead to Co-evolution?

Co-evolution

- Fitness of each individual of one population is re-evaluated with respect to the other population
- Achieves selective mutation
  - Mutation operators not selected a priori
  - Individual mutants selected
- Tailored to the specific program under test, based upon their fitness
- Guarantees non equivalent mutants
- Stubborn mutants might also be eliminated (?)
- The robustness of the algorithm probably will rediscover eliminated stubborn mutants (?)
Work in Progress and Future Work

- Mutation tool + GAs for Co-evolution
- Comparison of real results and simulation
- Comparative analysis with other methodologies (selective mutation, mutant sampling)

High-order Mutants

- This methodology could be used to check the validity of the Coupling Effect Hypothesis
- Large program faults, particularly those of a semantic nature are “coupled” with smaller syntactic faults that can be detected with mutation testing
- Will an effective test set for simple, first-order mutants be in the same level of effectiveness for more complex, high-order mutants?