Efficient Multi-Objective Higher Order Mutation Testing with Genetic Programming

W. B. Langdon
University College, London
Introduction

• 2 objectives: Hard to kill, little change to source
• Higher order mutation testing $\rightarrow$ mutant has more than one change
• How we use genetic programming
• Results on 4 benchmarks (triangle, schedule, tcas, gzip)
• Coupling Hypothesis
• Non-Deterministic Mutants
• Conclusions
Multi-Objective Search

• By extending mutation testing to higher orders we allow mutants to be more complicated, emulating expensive post release bugs which require multiple changes to fix.
• To avoid trivial mutants which are detected by many tests we search for hard to kill mutants which pass almost all of the test suite.
• Two objectives → Pareto multi-objective search
Evolving High Order Mutants

Evolution of Multi-Objective Higher Order Mutants with NSGA-II and Genetic Programming
Evolving High Order Mutants

- C source converted to BNF grammar
- BNF describes original source plus mutations
- All comparisons can be mutated
- Strongly Typed GP crosses over BNF to give new high order mutants.
- Compile population of mutants to give one executable. Run it on test suite to give fitness.
- Select parents of next generation.
source.c $\rightarrow$ BNF Grammar $\rightarrow$ GP $\rightarrow$ 10000 mutants

NSGA-II

gcc

population.exe $\rightarrow$ Test Cases

Pareto Evolution

1. Number of tests passed
2. Syntactic difference

W. B. Langdon, UCL
Triangle.c

int gettri(int side1, int side2, int side3) {
    int triang;
    if (side1 <= 0 || side2 <= 0 || side3 <= 0) {
        return 4;
    }
    triang = 0;
    if (side1 == side2) {
        triang = triang + 1;
    }
    if (side1 == side3) {
        triang = triang + 2;
    }
    if (side2 == side3) {
        triang = triang + 3;
    }
    if (triang == 0) {
        if (side1 + side2 < side3 || side2 + side3 < side1 || side1 + side3 < side2) {
            return 4;
        }
        else {
            Potential mutation sites (comparisons) in red
        }
    }
}
Triangle BNF syntax

<line1> ::= "int gettriXXX(int side1, int side2, int side3)\n"
<line2> ::= "{\n"
<line3> ::= "    \n"
<line4> ::= "    int triang ;\n"
<line5> ::= "    \n"
<line6> ::= <line6A> <line6B> <line6C>
<line6A> ::= "if( side1 " <compare> "0 || side2"
<line6B> ::= <compare> "0 || side3"
<line6C> ::= <compare> "0)\n"
<line7> ::= "return 4;\n"
<line8> ::= "}\n"
<line9> ::= "    \n"
<line10> ::= "triang = 0;\n"
<line11> ::= "\n"
<line12> ::= "if(side1 " <compare> "side2)\n"
<line13> ::= "triang = triang + 1;\n"
<line14> ::= "}\n"
<line15> ::= "if(side1 " <compare> "side3)\n"
<line16> ::= "triang = triang + 2;\n"
<line17> ::= "}\n"
<line18> ::= "if(side2 " <compare> "side3)\n"
Triangle BNF syntax 2

<start> ::= <line1> <line2> <line3> <line4> <line5> <line6-23> <line24-41> <line42> <line43> <line44> <line45> <line46>
<line6-23> ::= <line6-14> <line15-23>
<line6-14> ::= <line6-9> <line10-12> <line13> <line14>
<line6-9> ::= <line6> <line7> <line8> <line9>
<line10-12> ::= <line10> <line11> <line12>
<line15-23> ::= <line15-19> <line20-23>
<line15-19> ::= <line15-16> <line17-18> <line19>
<line15-16> ::= <line15> <line16>

<compare> ::= <compare0> | <compare1>
<compare0> ::= <compare00> | <compare01>
<compare00> ::= "<" | "<="
<compare01> ::= "]=" | "]!="
<compare1> ::= <compare10>
<compare10> ::= "]=" | "]"
Yue’s Triangle Test Cases

-3 4 5 4
3 4 5 1
3 -4 5 4
3 4 -5 4
-3 -4 -5 4
3 -4 -5 4
-3 4 -5 4
-3 -4 5 4
-3 5 4 4
3 -5 4 4
5 3 -4 4
5 -3 4 4
3 3 5 2
5 3 5 2
3 4 4 2
3 4 8 4
3 9 5 4
12 4 5 4
4 5 12 4
-4 12 5 4

60 test cases chosen to test all branches in triangle.c

Three integers followed by expected result
Triangle

- 7 first order mutants are very hard to kill (fail only 1 test).
- 8 first order mutants are equivalent (pass all)

<table>
<thead>
<tr>
<th>Yue's triangle</th>
<th>silent</th>
<th>1</th>
<th>median</th>
<th>95%</th>
<th>all 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>first order</td>
<td>0.094118</td>
<td>0.082353</td>
<td>4</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>second</td>
<td>0.008235</td>
<td>0.016177</td>
<td>9</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>third</td>
<td>0.000659</td>
<td>0.002224</td>
<td>11</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>fourth</td>
<td>0.000047</td>
<td>0.000249</td>
<td>11</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>18</td>
<td>0</td>
</tr>
</tbody>
</table>

W. B. Langdon, UCL
High Order Triangle Mutants

Yue’s triangle.c Comparison Mutants, 60 tests  WBL 15 Jan 2009

- 70 order 4 killed by 0
- 871 mutants killed by 1
- 56 order 3 killed by 0
- 189 mutants killed by 1
- 28 order 2 killed by 0
- 95 mutants killed by 1
- 8 first order mutants are killed by no test cases
- 7 first order mutant killed by 1 test case

Number test cases with different output

Syntax distance
High Order Triangle Mutants

The 10 normal operation tests detect >99% of random mutants
Coupling Hypothesis

“Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants.”

— Richard DeMillo

Plots strengthen this.
Created 16,383 test suites
Coupling First and Second Order Mutants

Fraction of 2\textsuperscript{nd} order mutants killed by a triangle test suite strongly correlated to fraction of 1\textsuperscript{st} order mutants it kills.
Coupling First and Third Order Mutants

Fraction of third order mutants killed by a triangle test suite is strongly correlated to the fraction of first order mutants it kills.
Competent Programmer Hypothesis

“errors should be detectable as small deviations from the intended program.”

— Richard DeMillo

Purushothaman and Perry suggest faults that remain after testing are complex.
Schedule

• 1 first order very hard to kill (only 1 test).
• 10 first order mutants are equivalent (pass all)

<table>
<thead>
<tr>
<th></th>
<th>silent</th>
<th>1</th>
<th>median</th>
<th>0.95</th>
<th>all 2650</th>
</tr>
</thead>
<tbody>
<tr>
<td>first order</td>
<td>0.1429</td>
<td>0.0143</td>
<td>1806</td>
<td>2413</td>
<td>0.0143</td>
</tr>
<tr>
<td>second</td>
<td>0.0189</td>
<td>0.0044</td>
<td>2235</td>
<td>2649</td>
<td>0.0303</td>
</tr>
<tr>
<td>third</td>
<td>0.0023</td>
<td>0.0009</td>
<td>2324</td>
<td>2649</td>
<td>0.0480</td>
</tr>
<tr>
<td>fourth</td>
<td>0.0002</td>
<td>0.0002</td>
<td>2395</td>
<td>2650</td>
<td>0.0672</td>
</tr>
<tr>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>2611</td>
<td>2650</td>
<td>0.2954</td>
</tr>
</tbody>
</table>
High Order Schedule Mutants
tcas

• 1 first order hard to kill (only passes 3 tests).
• No first order passes only 1 or 2 tests.
• 24 first order mutants are equivalent (pass all).
• As with triangle and schedule, high order tcas mutants (HOM) are easy to kill but show some interesting structure:
  – 428 tests are ineffective against HOM
  – 936 tests are almost ineffective against HOM
  – 264 tests kill almost all HOM. These tests check for aircraft threats.
Evolution of tcas Mutants
Evolved tcas Mutants

• GP finds 7th order mutant which is killed by only one test in generation 14.
• Fifth order mutant found in generation 44
• Second GP run found 4th order (generation 90) and third order mutant (generation 105).
• All of these are harder to kill than any first order mutant. They affect similar parts of the code but are not all semantically identical.
Evolved 3\textsuperscript{rd} order tcas Mutant

• Changes lines 101, 112, 117:

\[
\text{result} = \text{Own\_Below\_Threat()} \&\& \ (\text{Cur\_Vertical\_Sep} >= \text{MINSEP}) \&\& \ (\text{Down\_Separation} <= \text{ALIM}());
\]

\[
\text{result} = \text{Own\_Below\_Threat()} \&\& \ (\text{Cur\_Vertical\_Sep} >= \text{MINSEP}) \&\& \ (\text{Down\_Separation} >= \text{ALIM}());
\]

\[
\text{return} \ (\text{Own\_Tracked\_Alt} \leq \text{Other\_Tracked\_Alt}) ;
\]

\[
\text{return} \ (\text{Own\_Tracked\_Alt} < \text{Other\_Tracked\_Alt}) ;
\]

\[
\text{return} \ (\text{Other\_Tracked\_Alt} \leq \text{Own\_Tracked\_Alt}) ;
\]

\[
\text{return} \ (\text{Other\_Tracked\_Alt} < \text{Own\_Tracked\_Alt}) ;
\]

• 101 and 117 are silent but 112 fails 12 tests.

• Passes all tests except test 1400. Should return 0 but mutant returns DOWNWARD\_RA.

• Fitness 1,23 (1 tests failed, syntax distance=23).
## Efficiency Techniques

<table>
<thead>
<tr>
<th>Problems</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison of text (including error messages)</td>
<td>Replace <code>printf</code> results and error messages by saving output inside test harness and using status codes</td>
</tr>
<tr>
<td>Array indexing errors</td>
<td>Automatic array index checking before all array accesses</td>
</tr>
<tr>
<td>Run out of memory or corrupt the heap</td>
<td>Allocate heap memory large enough for all of the test cases</td>
</tr>
<tr>
<td>Read or write illegal memory</td>
<td>Automatic pointer checking before it is used.</td>
</tr>
<tr>
<td>Compiler overhead</td>
<td>Compile once with ability to select mutants</td>
</tr>
<tr>
<td>Non-terminated loops</td>
<td>Loop counter technique to kill mutants</td>
</tr>
<tr>
<td>Harmful system calls and IO operations</td>
<td>Record original program's use of system calls and IO by instrumenting the code.</td>
</tr>
<tr>
<td></td>
<td>Intercept and check system &amp; IO during mutation testing</td>
</tr>
<tr>
<td>Heavy disk usage</td>
<td>Combine tests into a single file.</td>
</tr>
<tr>
<td></td>
<td>A potential alternative might be to use RAM disk</td>
</tr>
<tr>
<td>Non-deterministic mutants</td>
<td>Force the initialisation of all variables</td>
</tr>
</tbody>
</table>
gzip

• Time to compile. Time to test
• Frame work needs to be robust to mutant code:
  – Time out looping mutants (For and goto)
  – Protect against invalid array indexes and pointers
    bgcc -fbounds_checking
  – Protect against trashing files. Intercept IO and system
  – Trap exceptions
• heavy use of macros and conditional compilation
  – Avoid mutations changing configuration but allow in
    .h by operating on source after include/macro
    expansion. gcc -E
gzip

• 5680 lines of C. SIR test suite (only 211 tests).
• Highly non-uniform testing.
• Concentrate on well tested code.
gzip first order mutants

SIR gzip 211 of 214 tests 1st Order Comparison Mutants WBL 6 Nov 2009

W. B. Langdon, UCL
gzip first order mutants

SIR gzip 211 of 214 tests 1st Order Comparison Mutants WBL 13 Sep 2009

Code where Mutant is equivalent or fails most

First order

W. B. Langdon, UCL
gzip well tested code

- 5680 lines of C. SIR test suite (only 211 tests).
- Highly non-uniform testing.
- Concentrate on well tested code.
- 496 1st order mutation sites
- 84 well tested
  - 78 potentially equivalent
  - 342 fail more than half SIR test suite
gzip 2nd order sow’s ear mutants

Of 87,150 2\textsuperscript{nd} mutants all except 4 are similar to 1st order.
2nd order interactions of “easy” 1st order

- Two easy to kill 1st partially conceal each other
- Both in for loop of `scan_tree()`
- if (count < min_count) mutated to <= Fails 89% of tests. Other 1st order is 13 lines later
- if (curlen == nextlen) ..else.. mutated to != Causes min_count=3 to be replaced by min_count=4 and vice-versa. Fails more than 99% of tests.
- In many tests first line is if(count<4) sometimes with both changes this becomes if(count<=3) and the test is past.
gzip 2nd order sow’s ear mutants

Of 87,150 2nd mutants all except 4 are similar to 1st order.
Two equivalent 1\textsuperscript{st} make each other visible

Both mutation sites are in longest\_match()

A \texttt{!=} at the start of the outer loop is replaced by \texttt{< !} = is partially for efficiency, when 1\textsuperscript{st} order makes a difference it is trapped by \texttt{if} 8 lines later

\texttt{if(len>best\_len)} is replaced by \texttt{(len>= best\_len)} so the last occurrence of the longest match rather than first is used.

When both changes are made the \texttt{if} no longer traps the effect of \texttt{<} and most tests are failed.
gzip Monte Carlo sample of high order

All but 7 tests highly effective against very high order mutants (as with other benchmarks). 7 in main() or close to it.
Non-Deterministic Mutants

• Mutating, even correct code, may cause it to produced different output when run again with the same input. Thus a non-deterministic mutant may or may not be killed by a test suite.

• Catching non-repeatable faults by repeated runs is not practical.

• Mutation test harness attempts to avoid non-determinism by:
  – Ensuring all variable are initialised (even in mutated code)
  – Controlling I/O and system calls.
  – Avoiding timing problems. Using loop counting, rather than time out, to detect and report indefinite loops.
  – Avoiding tests which rely on variable details of user input (eg. timing).
Conclusions

• Random high order mutants are easy to kill but may provide insight into code and test suite.
• Mutation testing can be viewed as multi-objective search.
• Genetic programming can find high order mutants which are both hard to kill and do not make too many changes to the original source code.
• Evidence for the Coupling Hypothesis
• Importance of non-deterministic mutants

Journal of Systems and Software  
doi:10.1016/j.jss.2010.07.027
The End !!!
More information on GP

- [http://www.cs.ucl.ac.uk/staff/W.Langdon](http://www.cs.ucl.ac.uk/staff/W.Langdon)