Multi-Objective
Higher Order Mutation Testing with
Genetic Programming

W. B. Langdon
King’s College, London
Introduction

• What is mutation testing
• 2 objectives: Hard to kill, little change to source
• Higher order mutation testing → mutant has more than one change
• How we search with genetic programming
• Results on 3 benchmarks (triangle, schedule, tcas)
• Future
• Conclusions
Mutation Testing

• Software testing is for detecting bugs.
• How good is a test suite?
  – How to improve it?
  – When to stop testing? (No bugs left to discover?)
• Mutation testing is the injection of changes similar to human programming bugs for testing.
• Does test suite detect change?
  Yes. Maybe test suite ok?
  No. Test suite needs improving? at the mutation?
Higher Order Mutation Testing

• The order of a mutant is the number of changes.
• 1st order means exactly one change is made to the code.
  – Most research is on first order mutants.
• Higher order means two or more changes.
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

Number test cases with different output
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 → BNF Grammar → GP → 10000 mutants

NSGA-II

Pareto Evolution

1. Number of tests passed
2. Syntactic difference

gcc

population.exe → Test Cases
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 {

Triangle BNF syntax

triangle ::= "int gettriXXX(int side1, int side2, int side3)\n"
triangle ::= "{\n
triangle ::= "  \n"
triangle ::= "int triang ;\n"
triangle ::= "  \n"
triangle ::= <line6A> <line6B> <line6C>
triangle ::= <line6A> ::= "if( side1" <compare> "0 || side2"
triangle ::= <line6B> ::= <compare> "0 || side3"
triangle ::= <line6C> ::= <compare> "0){\n
triangle ::= "return 4;\n"
triangle ::= "}\\n"
triangle ::= "  \n"
triangle ::= "triang = 0;\n"
triangle ::= "  \n"
triangle ::= "if(side1" <compare> "side2){\n
triangle ::= "triang = triang + 1;\n"
triangle ::= "}\\n"
triangle ::= "if(side1" <compare> "side3){\n
triangle ::= "triang = triang + 2;\n"
triangle ::= "}\\n"
triangle ::= "if(side2" <compare> "side3){\n"
Triangle BNF syntax 2

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

\(<\text{compare}>\) ::= <compare0> | <compare1>
\(<\text{compare0}>\) ::= <compare00> | <compare01>
\(<\text{compare00}>\) ::= \("<" | "\leq"\
\(<\text{compare01}>\) ::= \("\geq" | "\neq"\
\(<\text{compare1}>\) ::= <compare10>
\(<\text{compare10}>\) ::= \("=" | ")\)
Yue’s Triangle Test Cases

60 test cases chosen to test all branches in triangle.c (I.e. branch coverage plus tests to cover all Boolean expressions.)

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>equivalent</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, Crest
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
- 55 mutants killed by 1
- 8 first order mutants are killed by no test cases
- 7 first order mutant killed by 1 test case

Syntax distance vs. Number test cases with different output
High Order Triangle Mutants

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

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

<table>
<thead>
<tr>
<th>equivalent</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>
</tr>
<tr>
<td>second</td>
<td>0.0189</td>
<td>0.0044</td>
<td>2235</td>
<td>2649</td>
</tr>
<tr>
<td>third</td>
<td>0.0023</td>
<td>0.0009</td>
<td>2324</td>
<td>2649</td>
</tr>
<tr>
<td>fourth</td>
<td>0.0002</td>
<td>0.0002</td>
<td>2395</td>
<td>2650</td>
</tr>
<tr>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>2611</td>
<td>2650</td>
</tr>
</tbody>
</table>
High Order Schedule Mutants

schedule 1-4th Order and Random Comparison Mutants  WBL 5 Feb 2009

10000 High order
625625 4th order
45500 3rd order
2275 2nd order
70 1st order

Syntax distance

Number test cases with different output
tcas - aircraft collision avoidance

• 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

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

W. B. Langdon, Crest
Evolved tcas Mutants

- GP finds 7\textsuperscript{th} order mutant which is killed by only one test in generation 14.
- Fifth order mutant found in generation 44
- Second GP run found 4\textsuperscript{th} 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:

result = Own\_Below\_Threat() && (Cur\_Vertical\_Sep >= MINSEP) && (Down\_Separation <= ALIM());  
result = Own\_Below\_Threat() && (Cur\_Vertical\_Sep >= MINSEP) && (Down\_Separation >= ALIM());

return (Own\_Tracked\_Alt <= Other\_Tracked\_Alt);  
return (Own\_Tracked\_Alt < Other\_Tracked\_Alt);  
return (Other\_Tracked\_Alt <= Own\_Tracked\_Alt);  
return (Other\_Tracked\_Alt < Own\_Tracked\_Alt); (original in gray)

• 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).

W. B. Langdon, Crest
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
    \texttt{gcc} \texttt{–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
    \texttt{.h} by operating on source after include/macro
    expansion. \texttt{gcc} \texttt{–E}
gzip first order mutants

W. B. Langdon, Crest
gzip first order mutants

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

First order
Code where Mutant is equivalent or fails most

Count

cases when executed

test cases with different output

W. B. Langdon, Crest
gzip 2nd order sow’s ear mutants

W. B. Langdon, Crest
Future Work

• Coevolution: Mutants $\rightarrow$ better tests $\rightarrow$ tougher mutants
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.
• GP can find high order mutants which are both hard to find and do not make too many changes to the original source code.
The End !!!
More information on GP

- http://www.cs.ucl.ac.uk/staff/W.Langdon
  - GP and Data Structures, Kluwer, 1998