# 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?

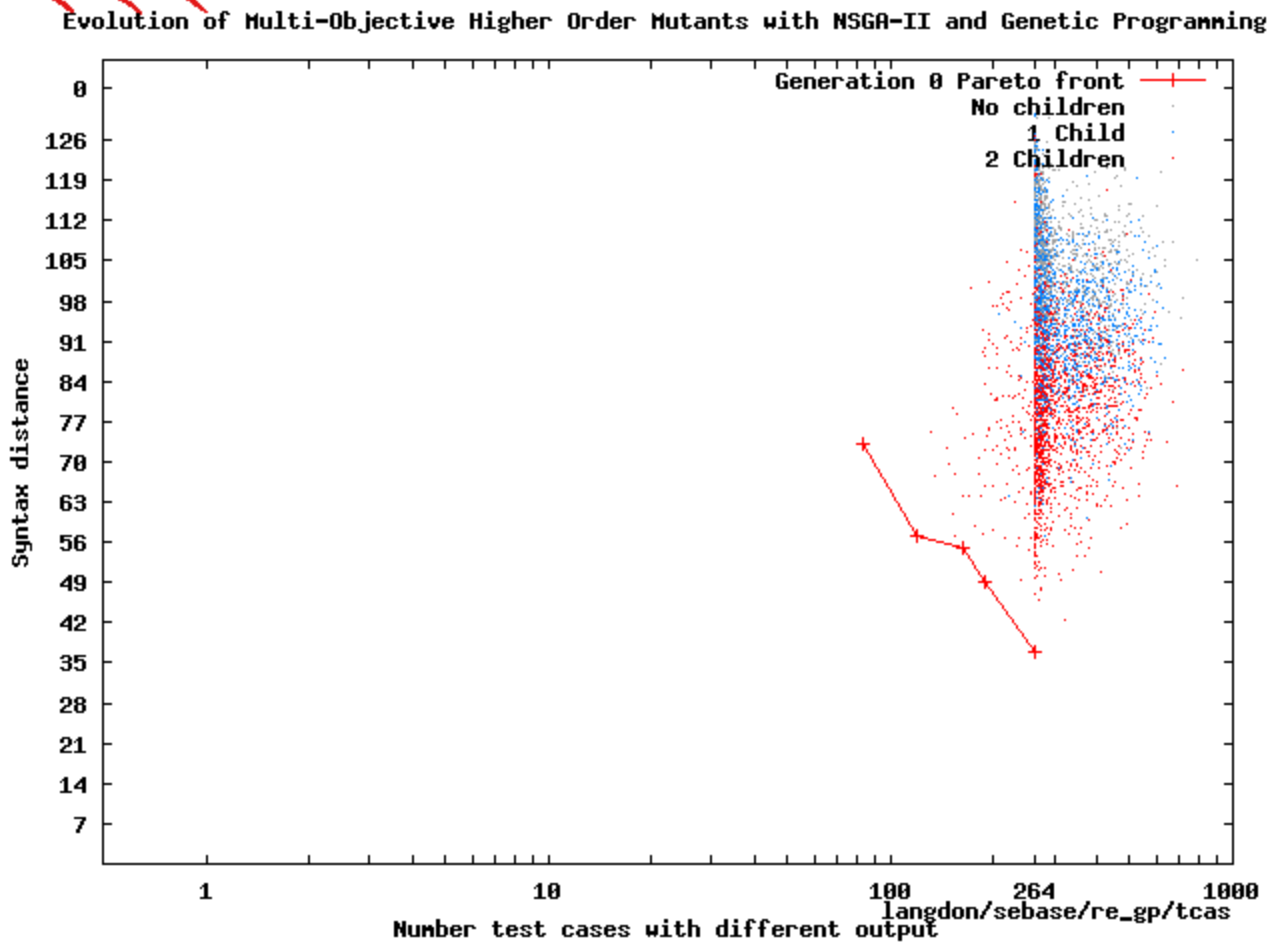
# Higher Order Mutation Testing

- 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

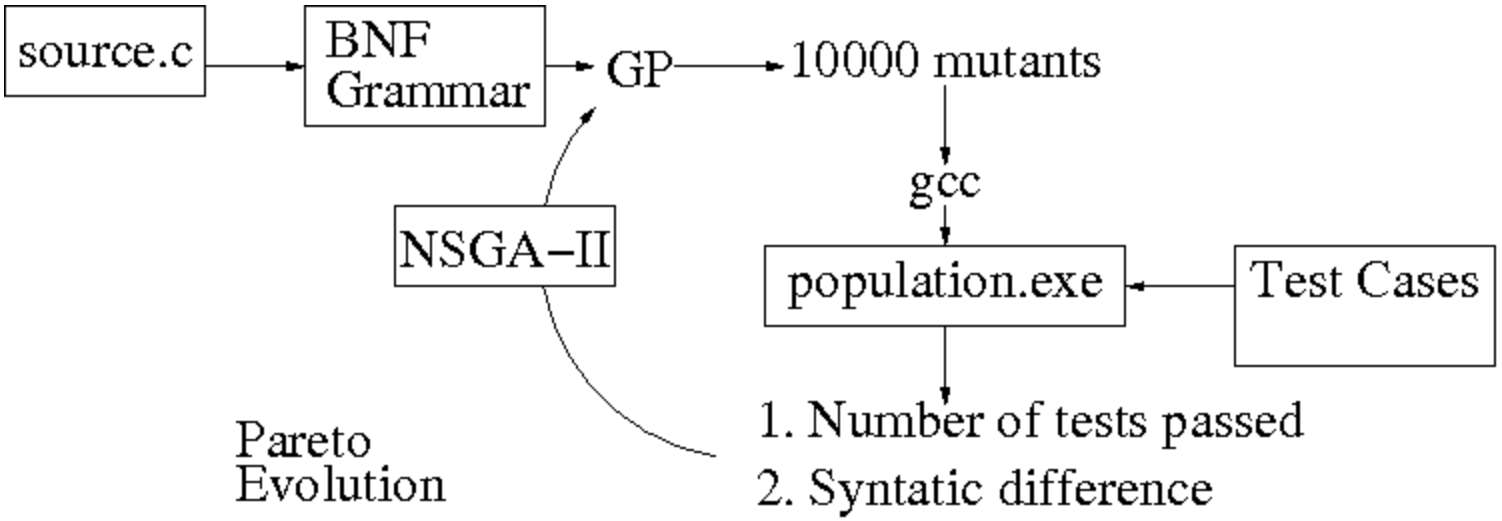
- By extending mutation testing to higher orders we allow mutants to be more complicated, emulating expensive post release bugs which require multiple changes to fix.
- To avoid trivial mutants which are detected by many tests we search for hard to kill mutants which pass almost all of the test suite.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

# Evolving High Order Mutants



# 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
gcc
NSGA–II
population.exe
Test Cases
Pareto Evolution
1. Number of tests passed
2. Syntatic difference

# 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 (I.e. branch coverage plus tests to cover all Boolean expressions.)

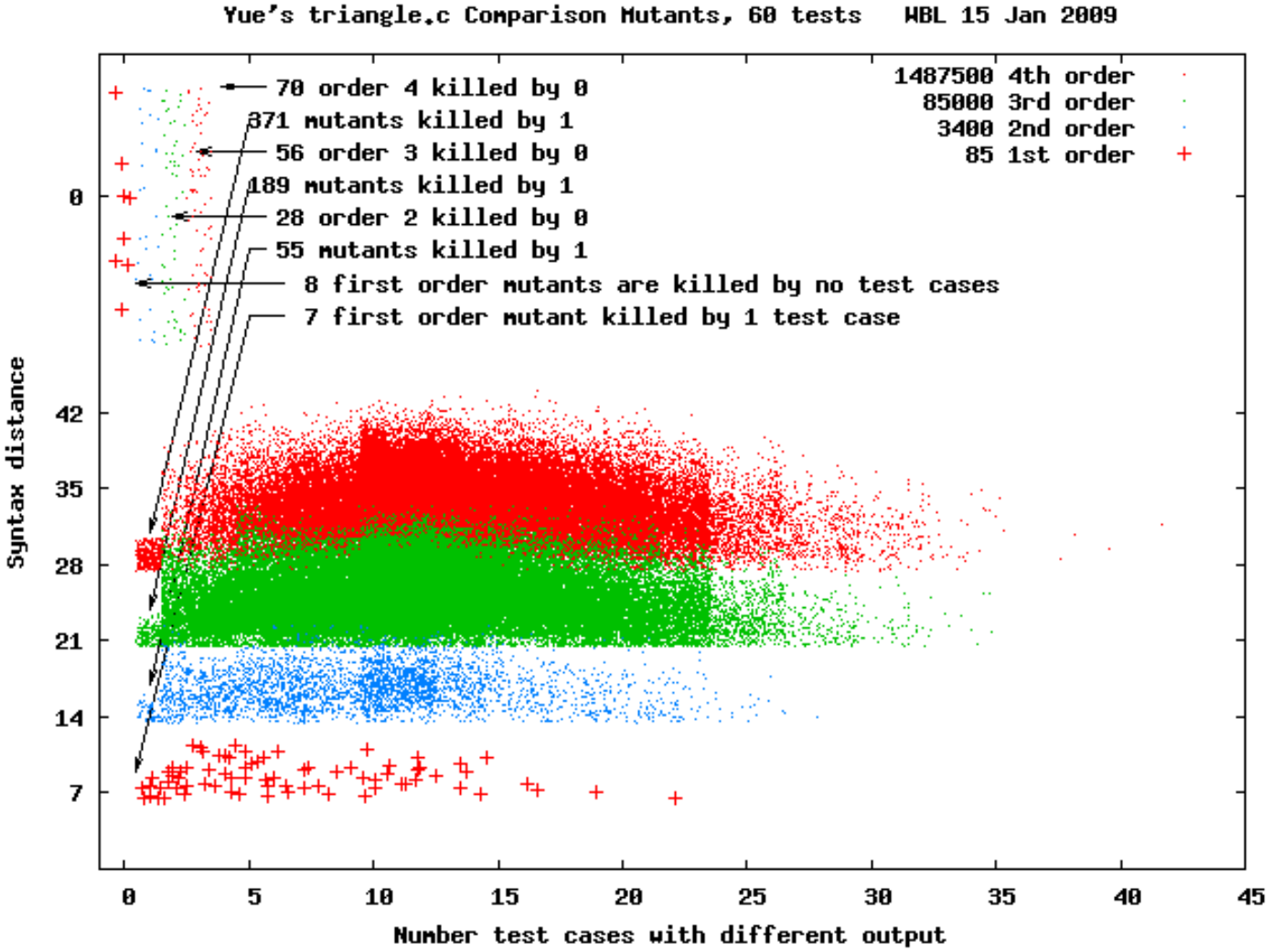
Three integers followed by expected result

# Triangle

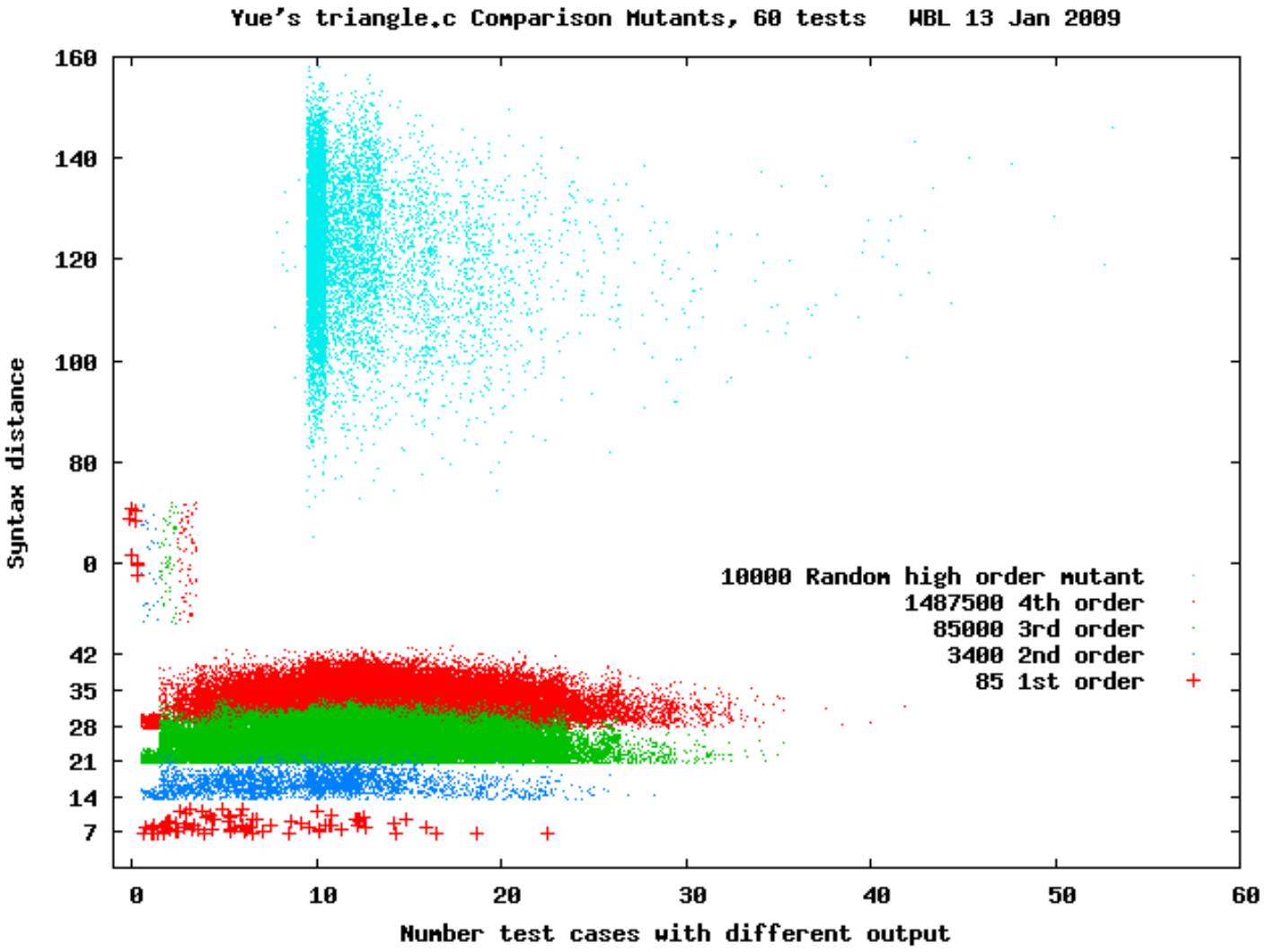
- 7 first order mutants are very hard to kill (fail only 1 test).
- 8 first order mutants are equivalent (pass all)

| Yue's triangle | | | | | |
|---|---|---|---|---|---|
| | equivalent | 1 | median | 95% | all 60 |
| first order | 0.094118 | 0.082353 | 4 | 15 | 0 |
| second | 0.008235 | 0.016177 | 9 | 18 | 0 |
| third | 0.000659 | 0.002224 | 11 | 20 | 0 |
| fourth | 0.000047 | 0.000249 | 11 | 21 | 0 |
| Random | 0 | 0 | 10 | 18 | 0 |

# High Order Triangle Mutants



# High Order Triangle Mutants



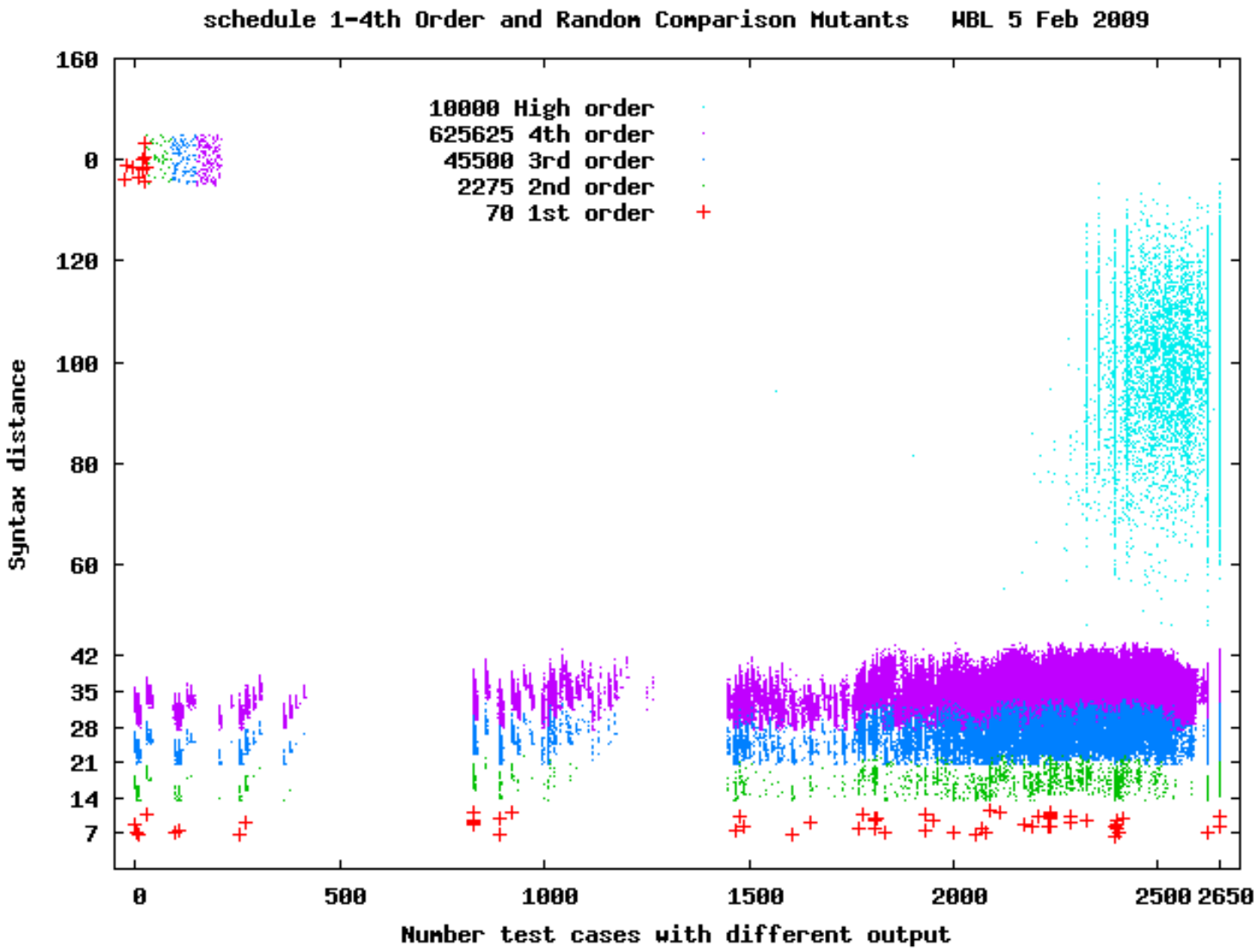

The 10 normal operation tests detect >99% of random mutants

# Schedule

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

| | equivalent | 1 | median | 0.95 | all 2650 |
|---|---|---|---|---|---|
| first order | 0.1429 | 0.0143 | 1806 | 2413 | 0.0143 |
| second | 0.0189 | 0.0044 | 2235 | 2649 | 0.0303 |
| third | 0.0023 | 0.0009 | 2324 | 2649 | 0.0480 |
| fourth | 0.0002 | 0.0002 | 2395 | 2650 | 0.0672 |
| Random | 0 | 0 | 2611 | 2650 | 0.2954 |

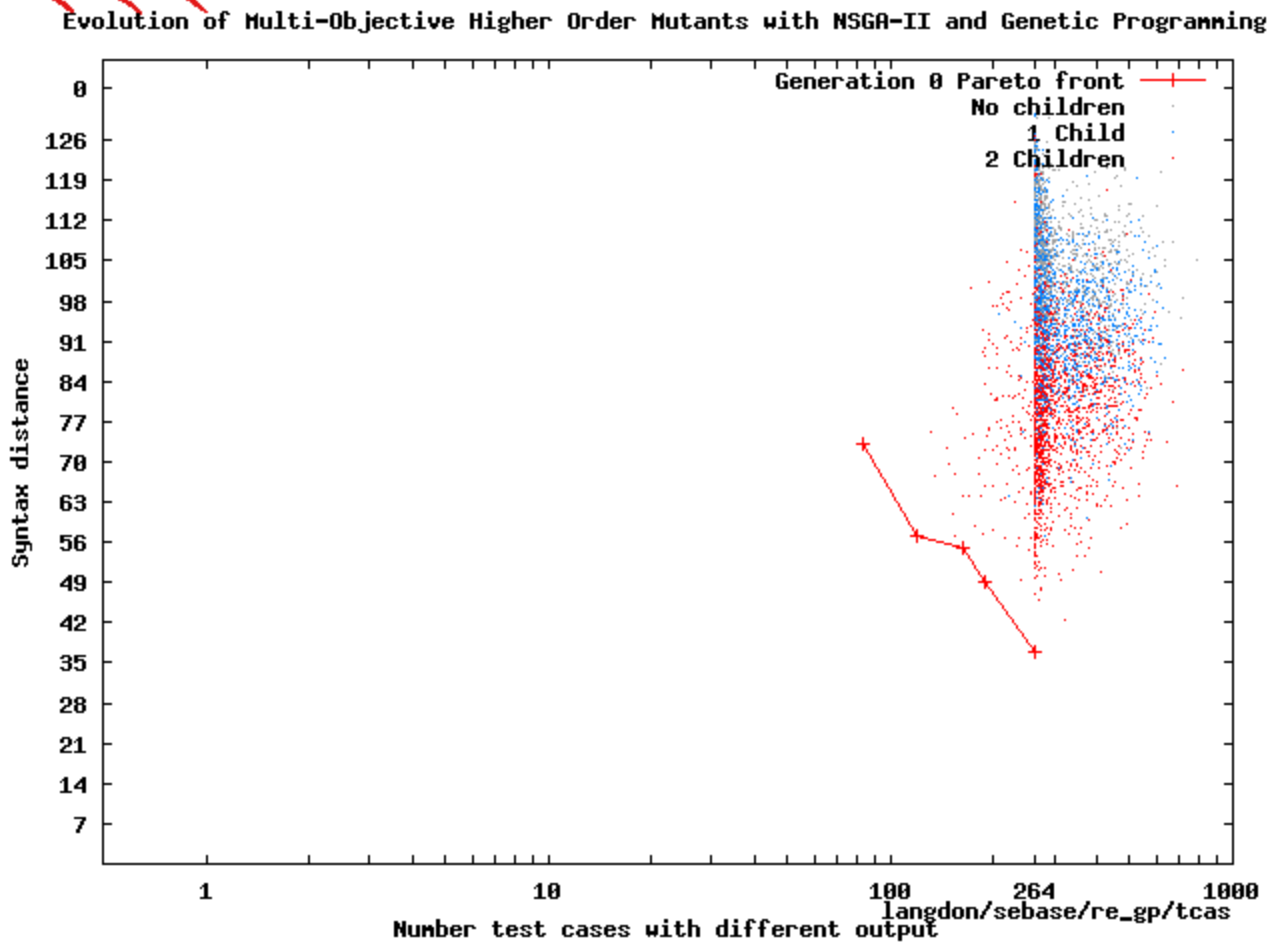
# High Order Schedule Mutants



# 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



# 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 3rd 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);
```

Line 112 Own_Below_Threat()

Line 117 Own_Above_Threat()

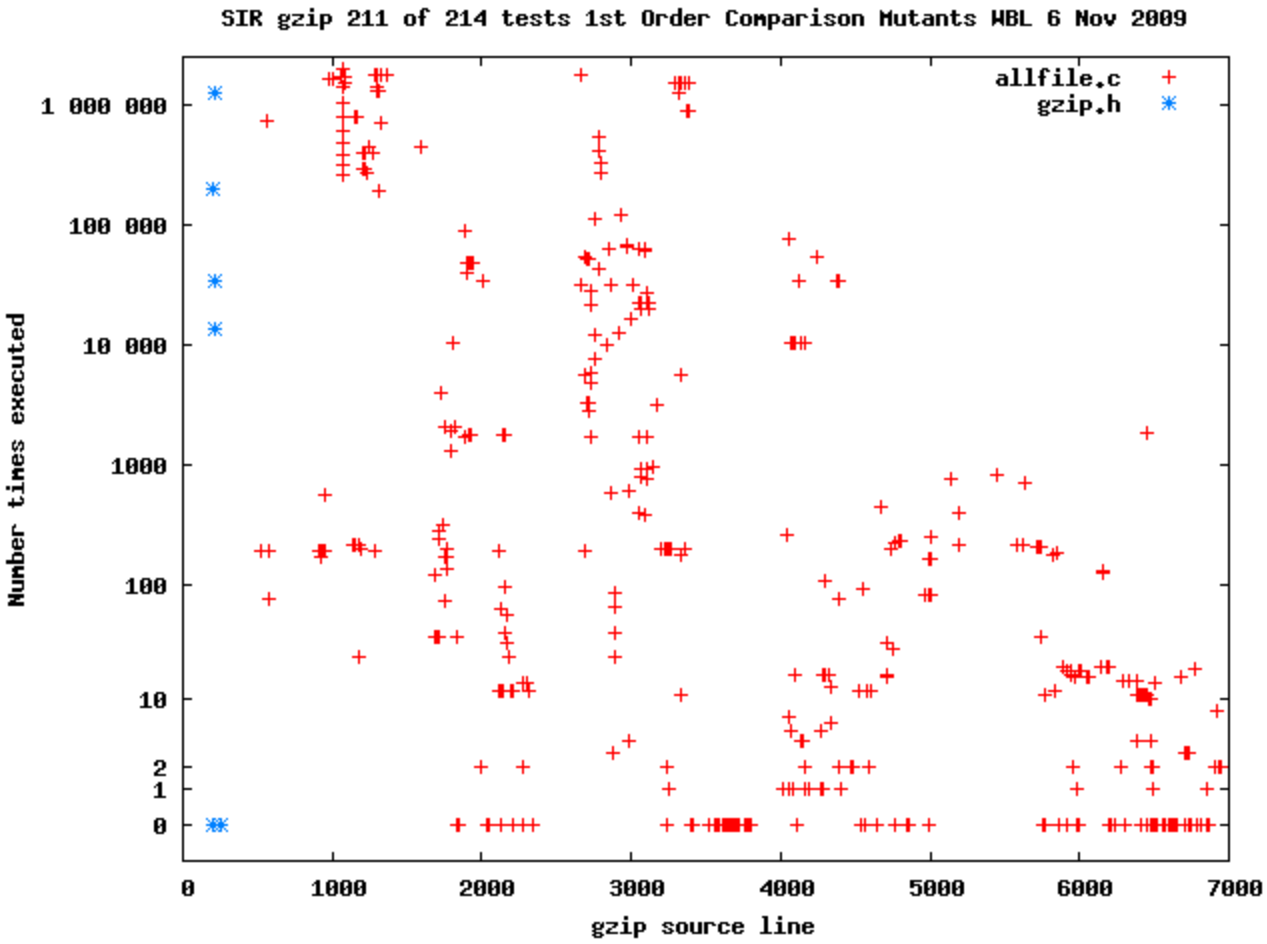
(original in gray)

- 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).

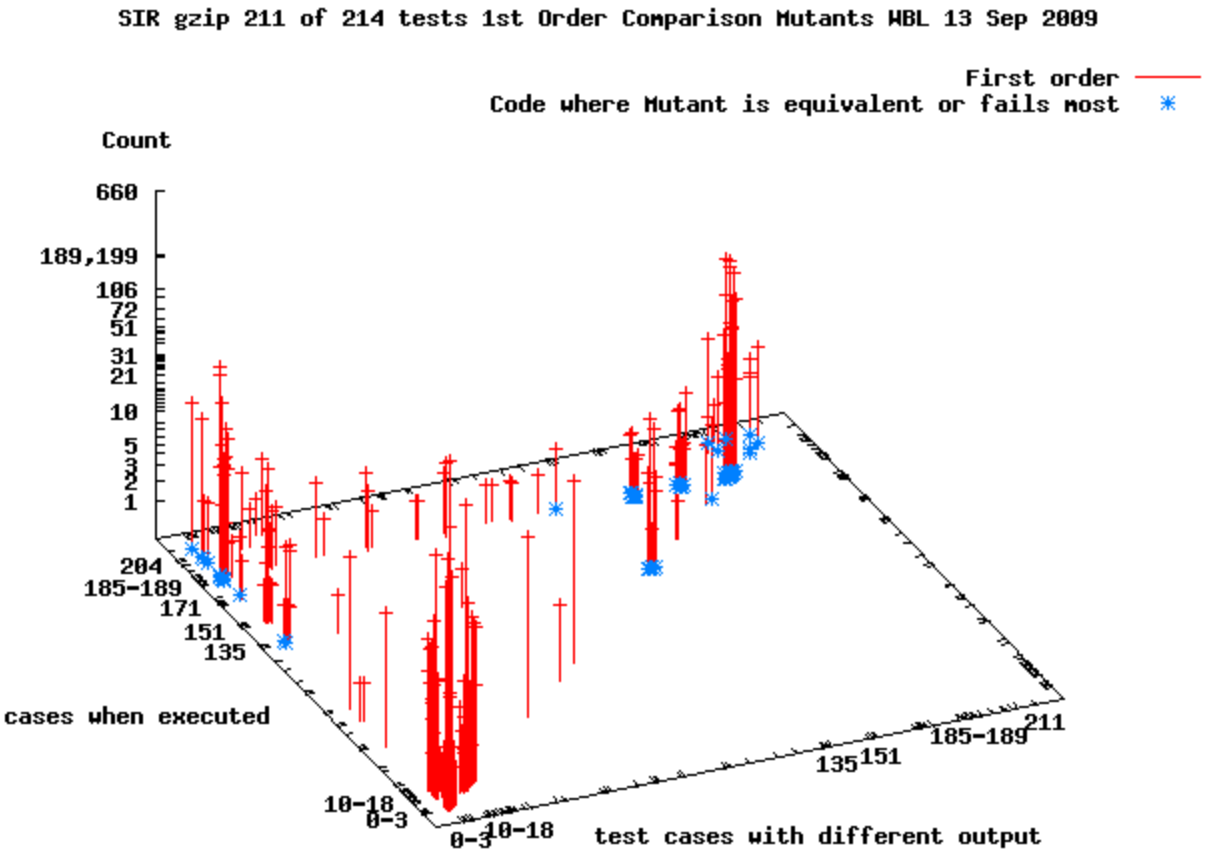
# gzip

- 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

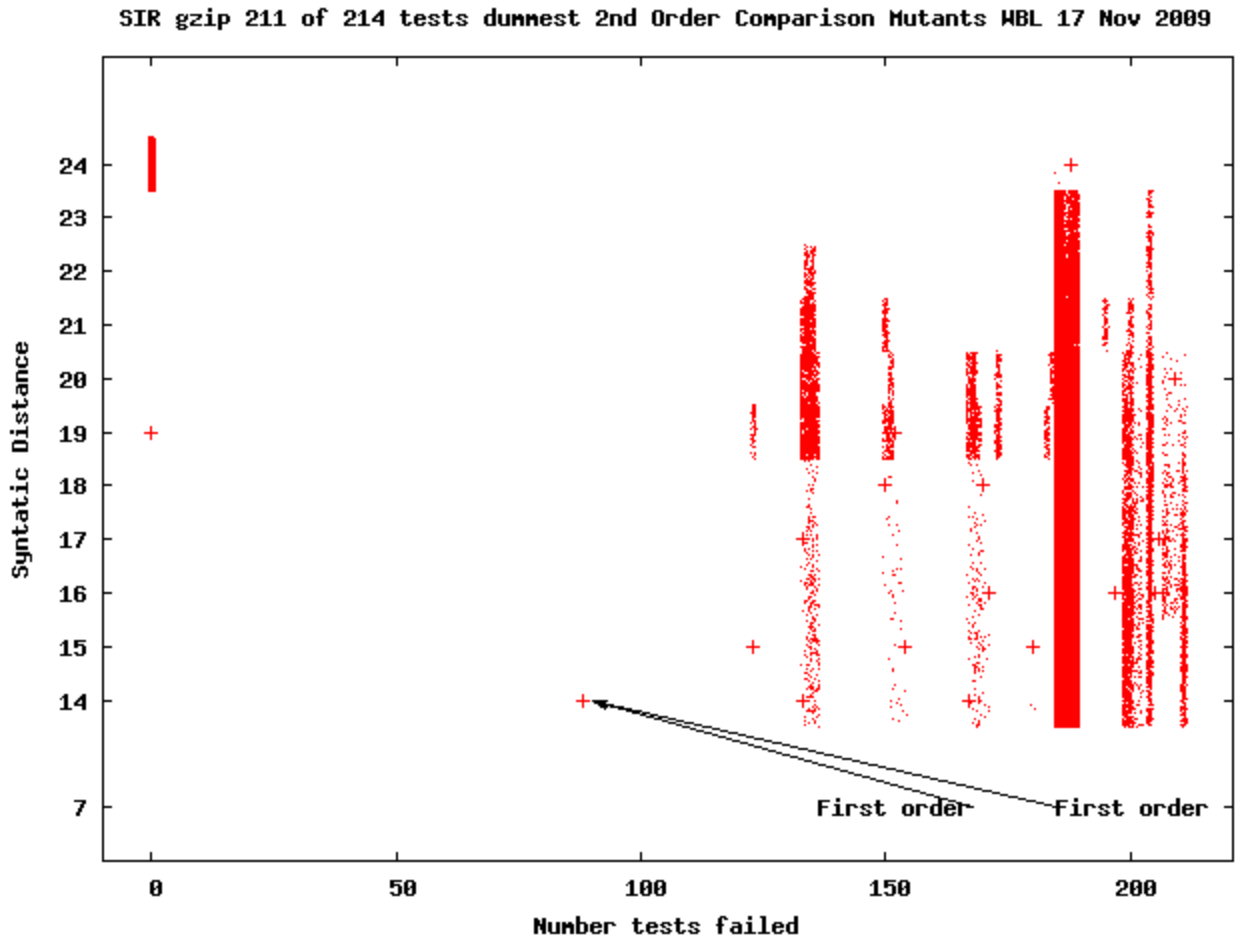
# gzip first order mutants



# gzip first order mutants



# gzip 2nd order sow's ear mutants



# Future Work

- Coevolution: Mutants→better tests→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
  - *A Field Guide to Genetic Programming*, **Free**, 2008
  - *Foundations of GP*, Springer, 2002
  - *GP and Data Structures*, Kluwer, 1998