Efficient Multi-Objective Higher Order Mutation Testing with Genetic Programming

W. B. Langdon

University College, London

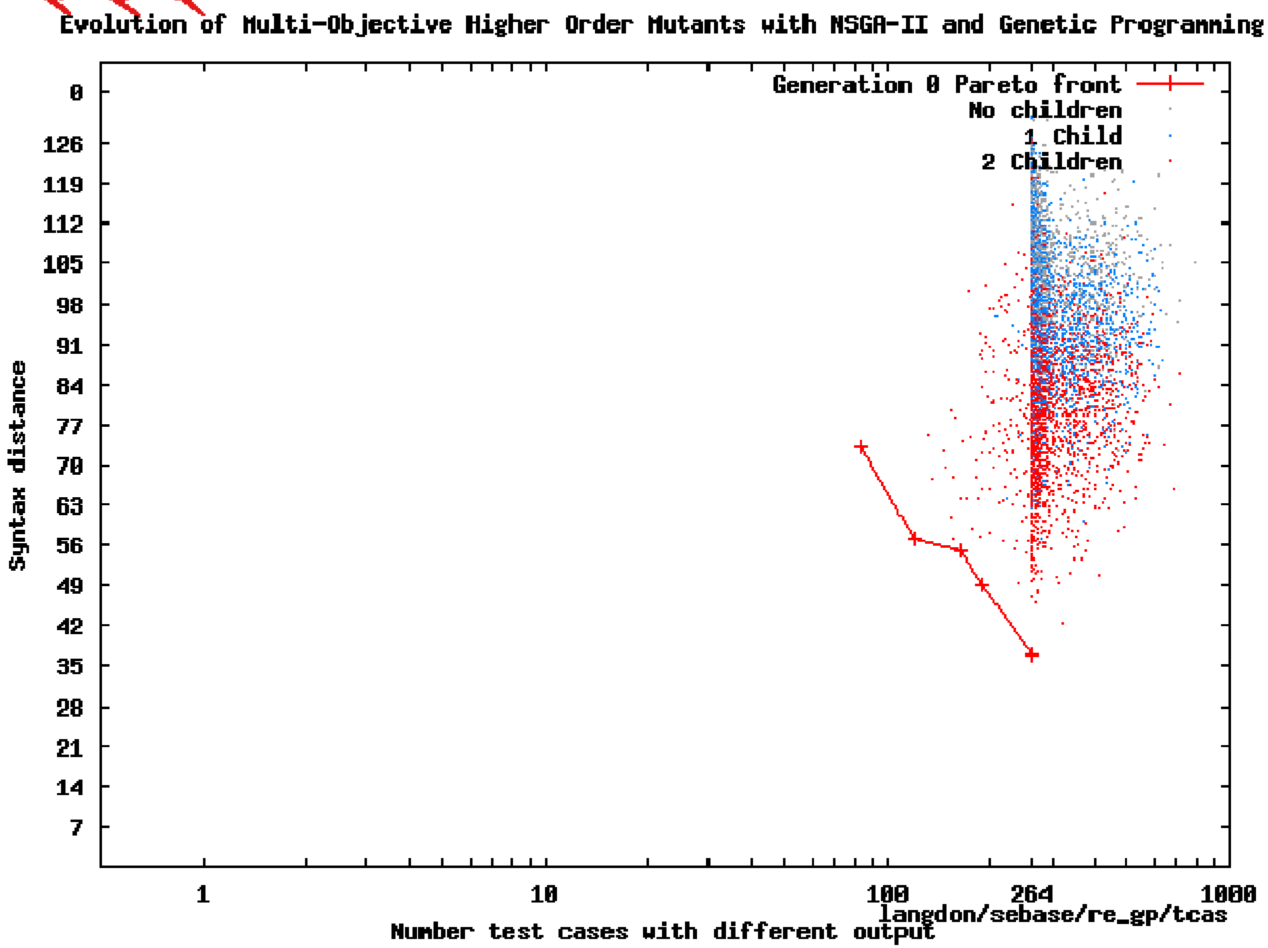
Introduction

- 2 objectives: Hard to kill, little change to source
- Higher order mutation testing→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

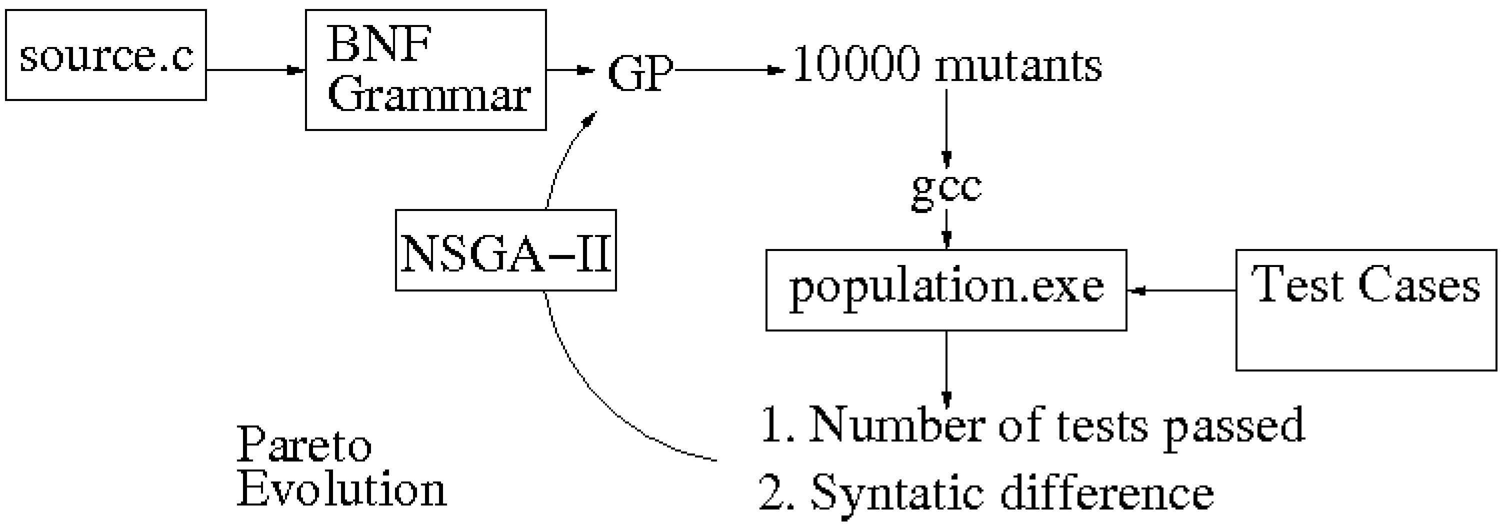
- By extending mutation testing to higher orders we allow mutants to be more complicated, emulating expensive post release bugs which require multiple changes to fix.
- To avoid trivial mutants which are detected by many tests we search for hard to kill mutants which pass almost all of the test suite.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

Evolving High Order Mutants



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

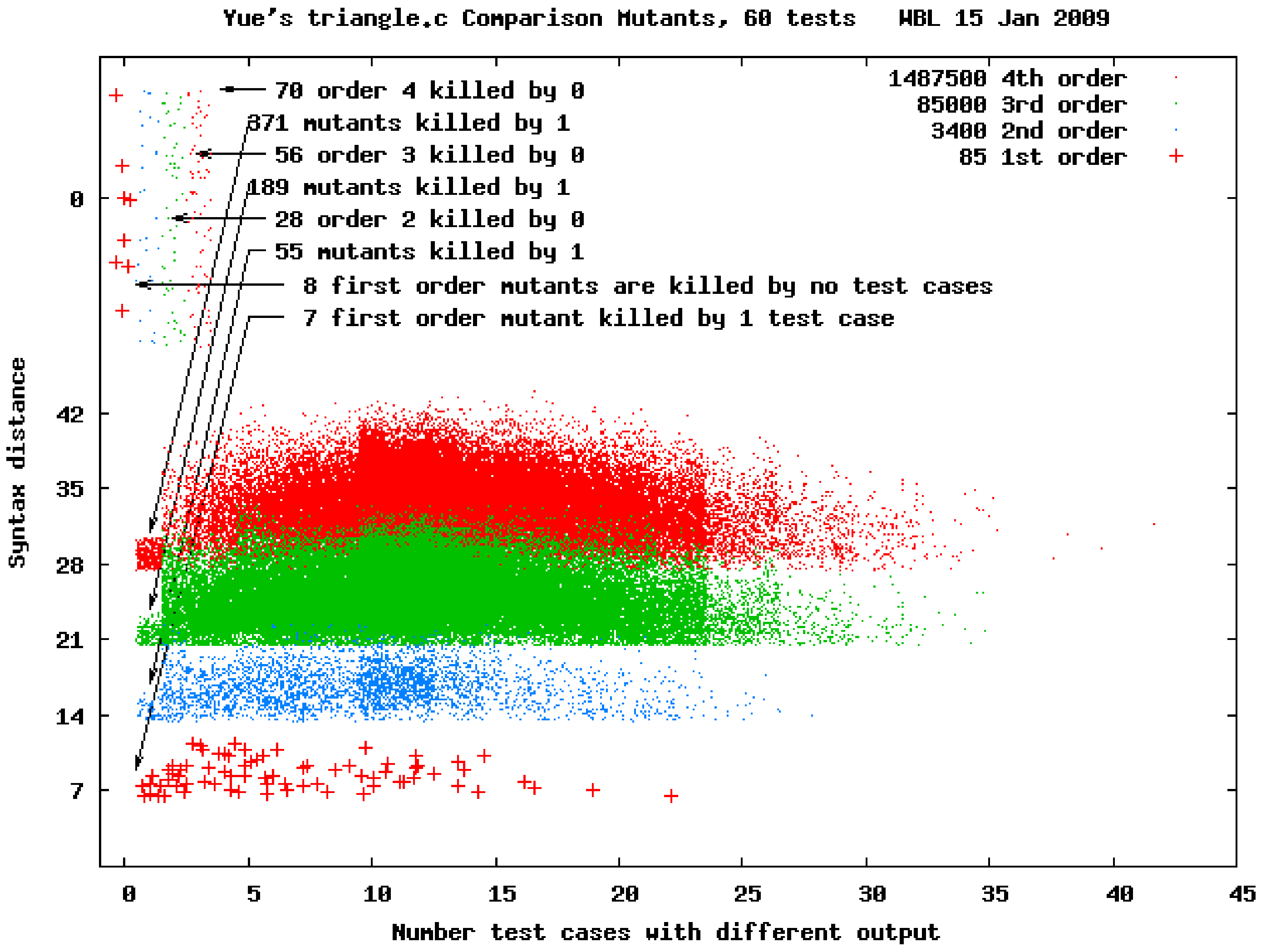
Three integers followed by expected result

Triangle

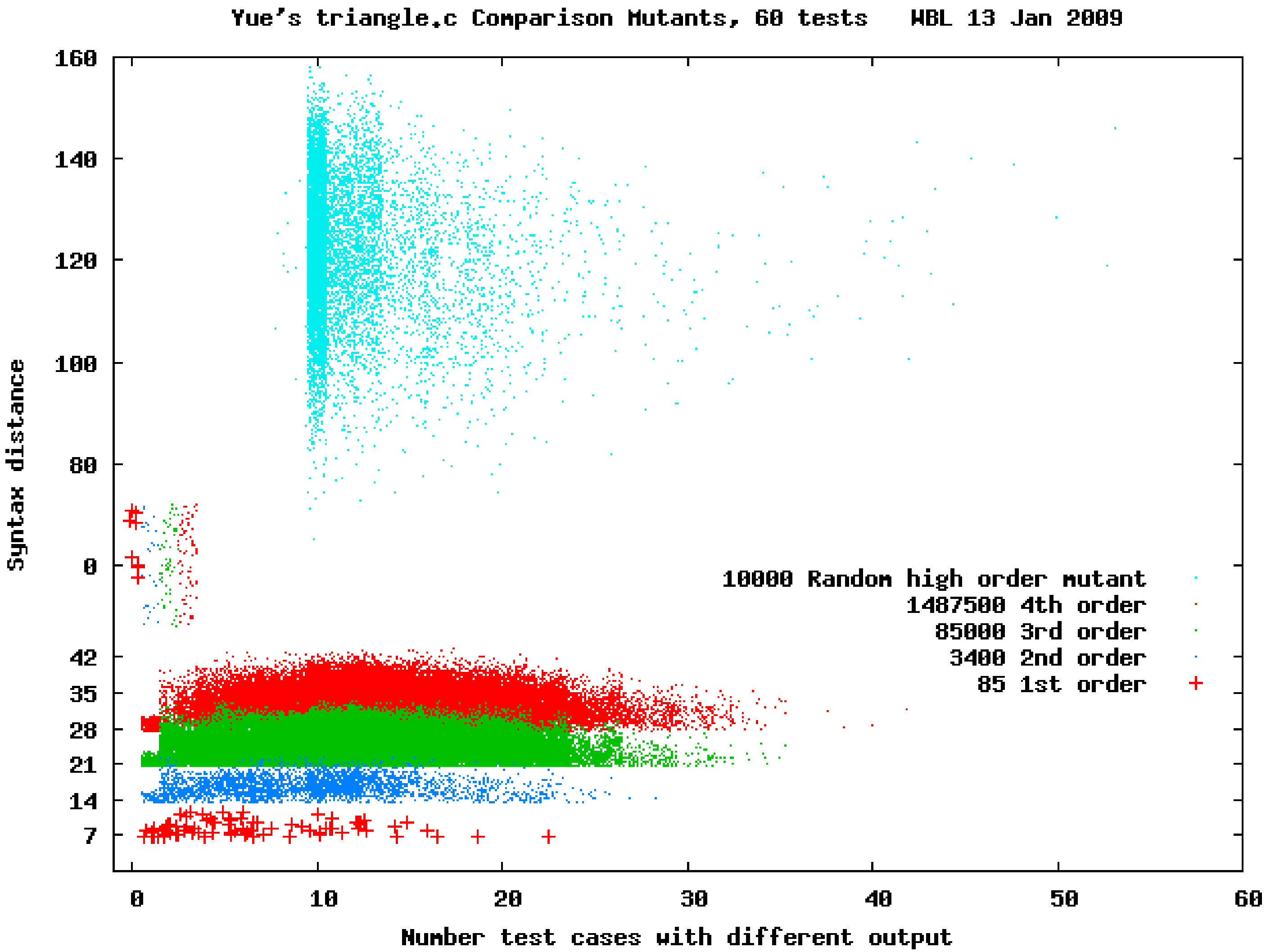
- 7 first order mutants are very hard to kill (fail only 1 test).
- 8 first order mutants are equivalent (pass all)

Yue's triangle					
	silent	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

Coupling Hypothesis

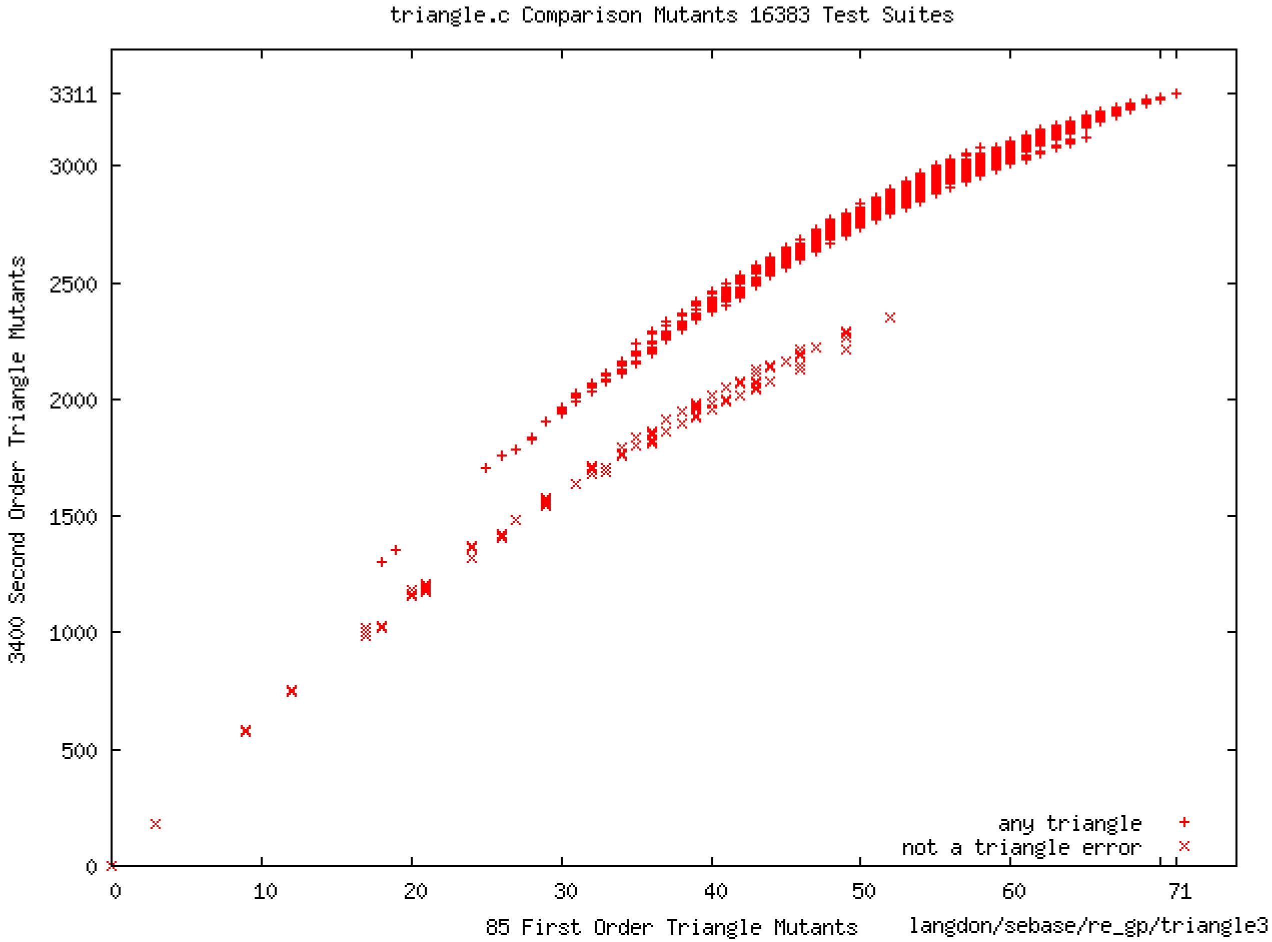
"Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants."

– Richard DeMillo

Plots strengthen this.

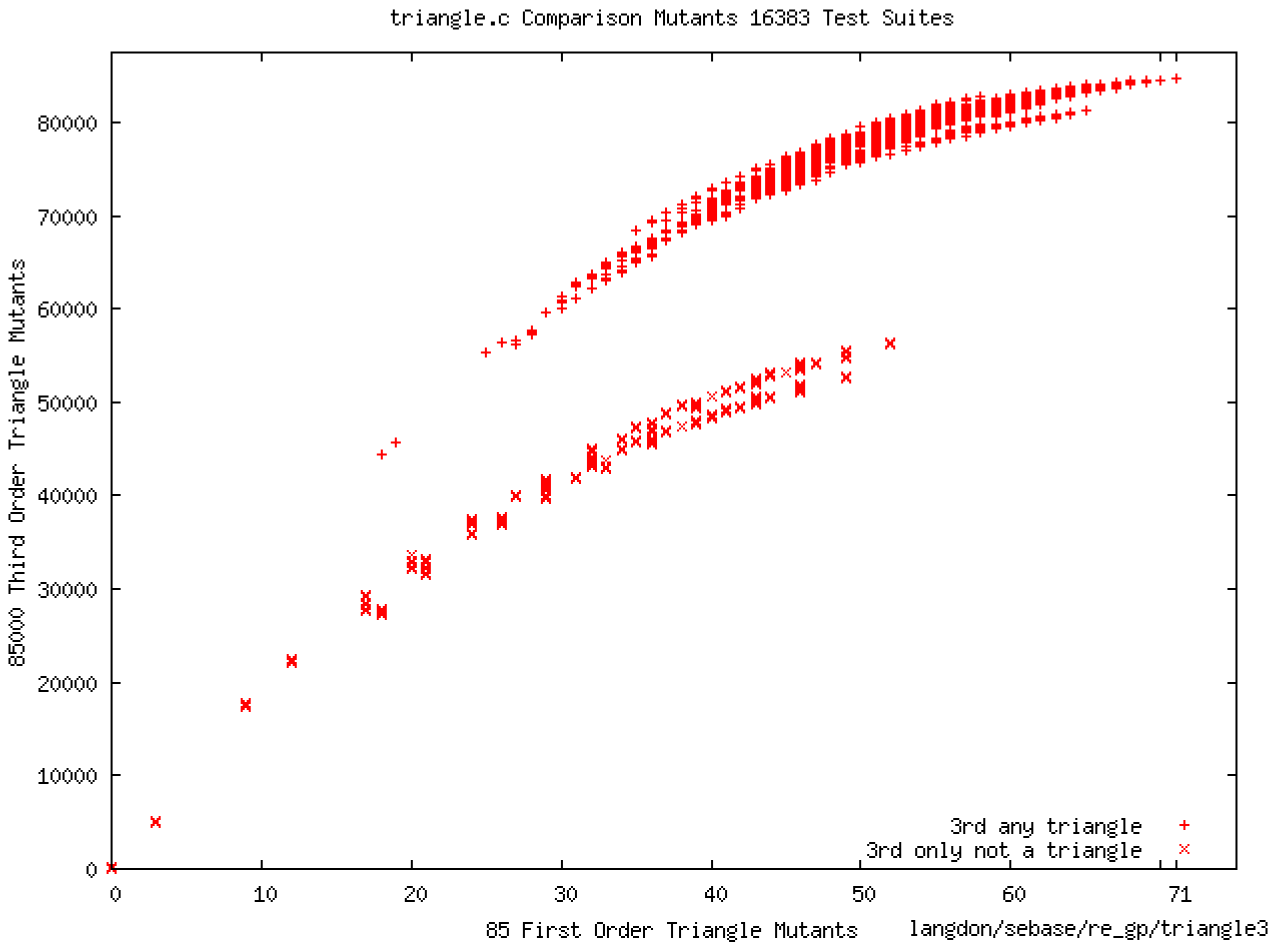
Created 16,383 test suites

Coupling First and Second Order Mutants



Fraction of 2nd order mutants killed by a triangle test suite strongly correlated to fraction of 1st 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

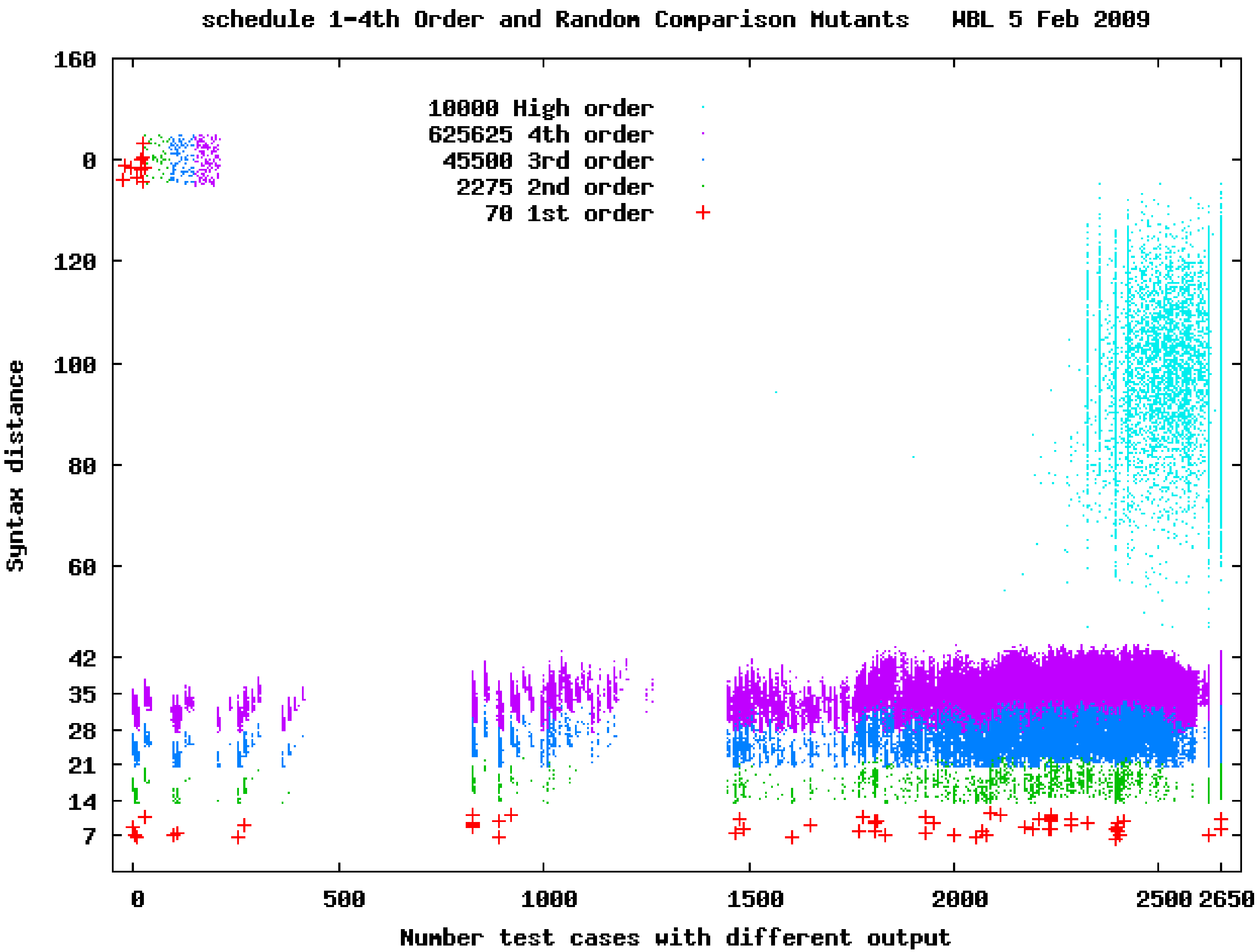
Purushothaman and Perry suggest faults that remain after testing are complex.

Schedule

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

	silent	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

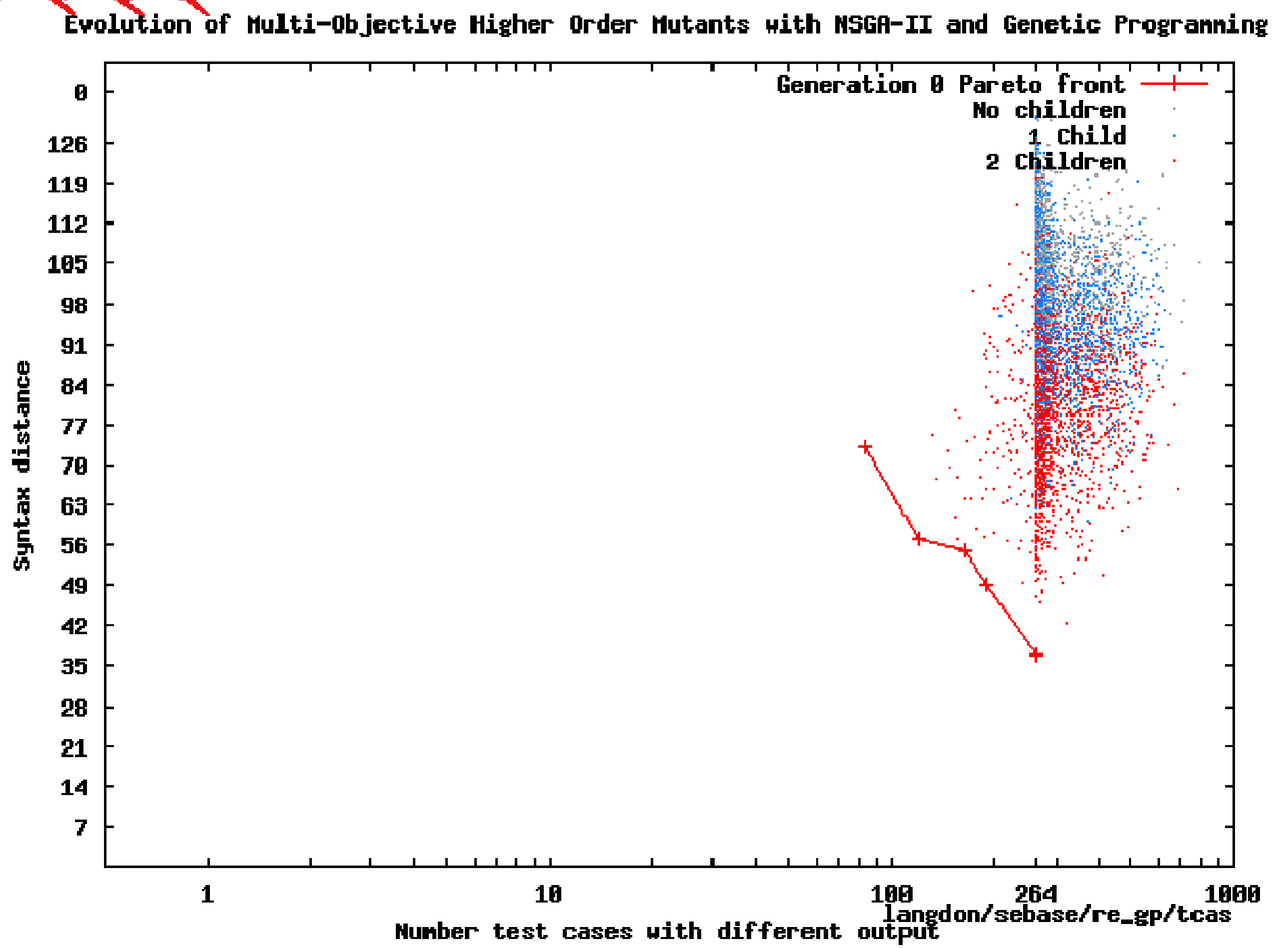
High Order Schedule Mutants



tcas

- 1 first order hard to kill (only passes 3 tests).
- No first order passes only 1 or 2 tests.
- 24 first order mutants are equivelent (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);
```

Line 112 Own_Below_Threat()

```
return (Other_Tracked_Alt <= Own_Tracked_Alt);
return (Other_Tracked_Alt <   Own_Tracked_Alt);
```

Line 117 Own_Above_Threat()

- 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

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

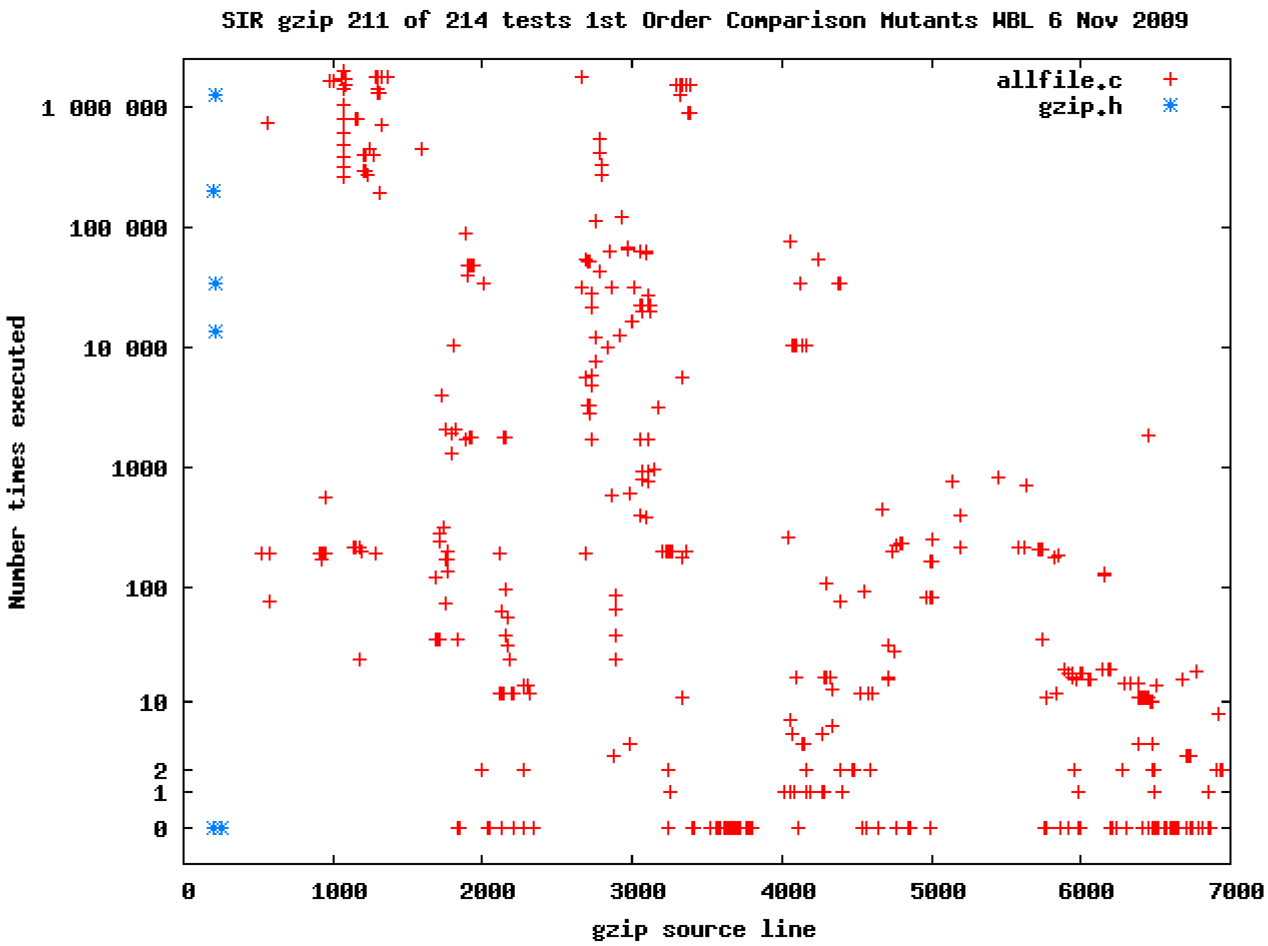
gzip

- 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

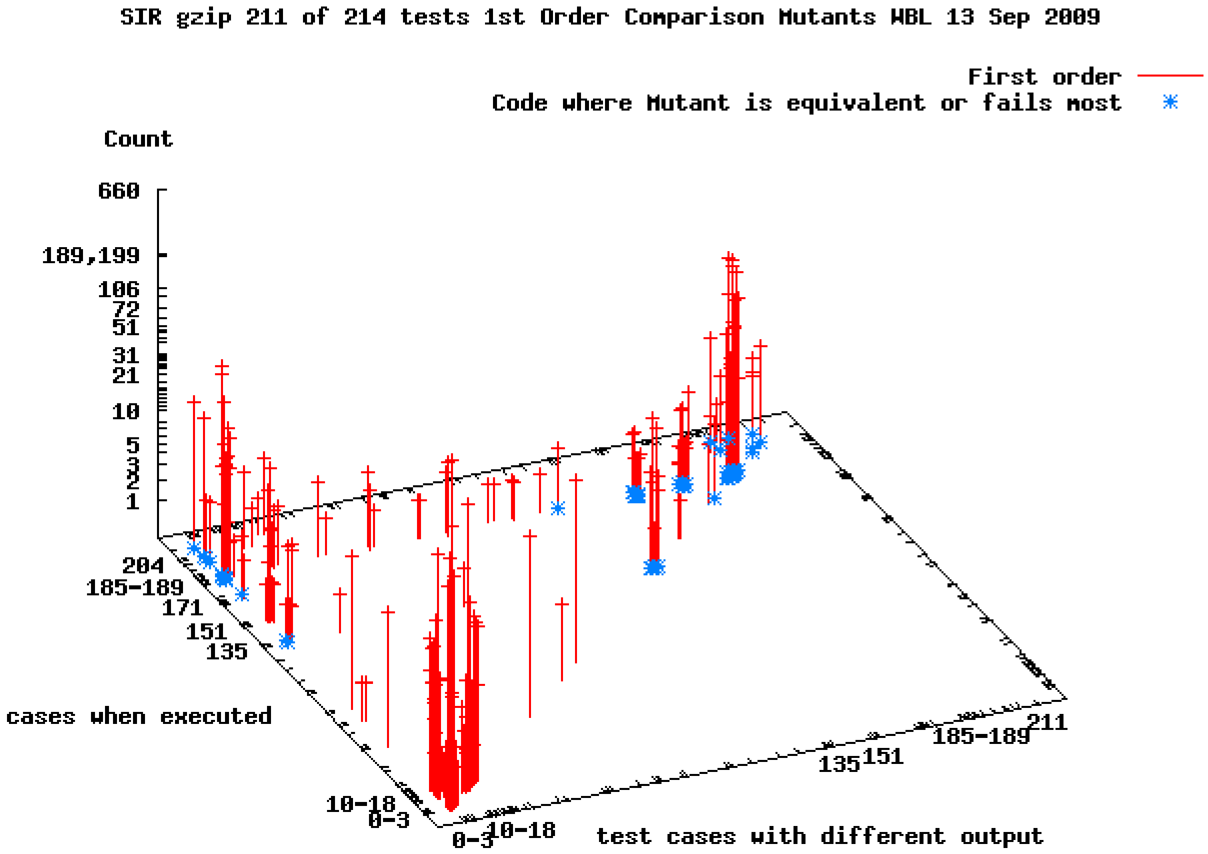
gzip

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

gzip first order mutants



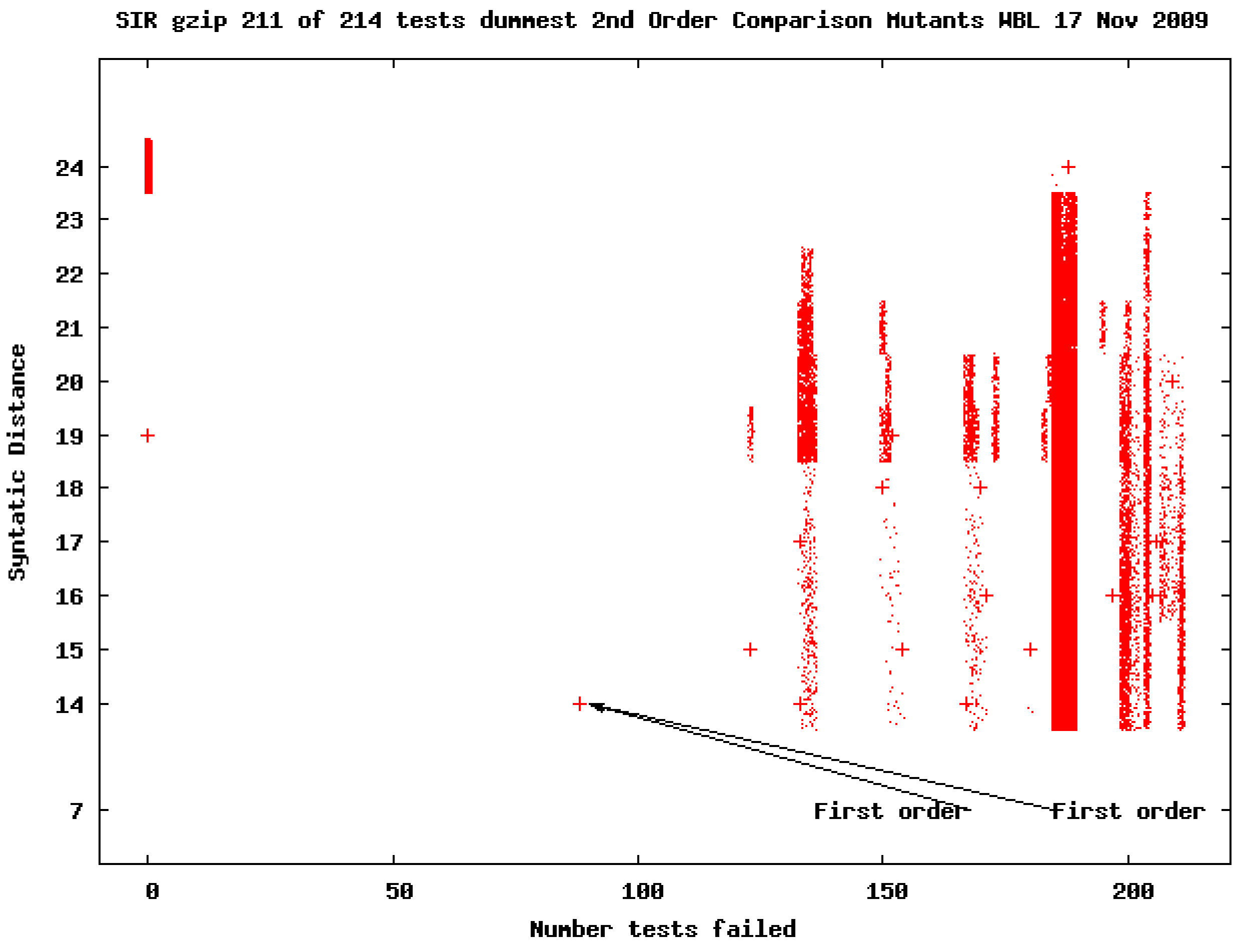
gzip first order mutants



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

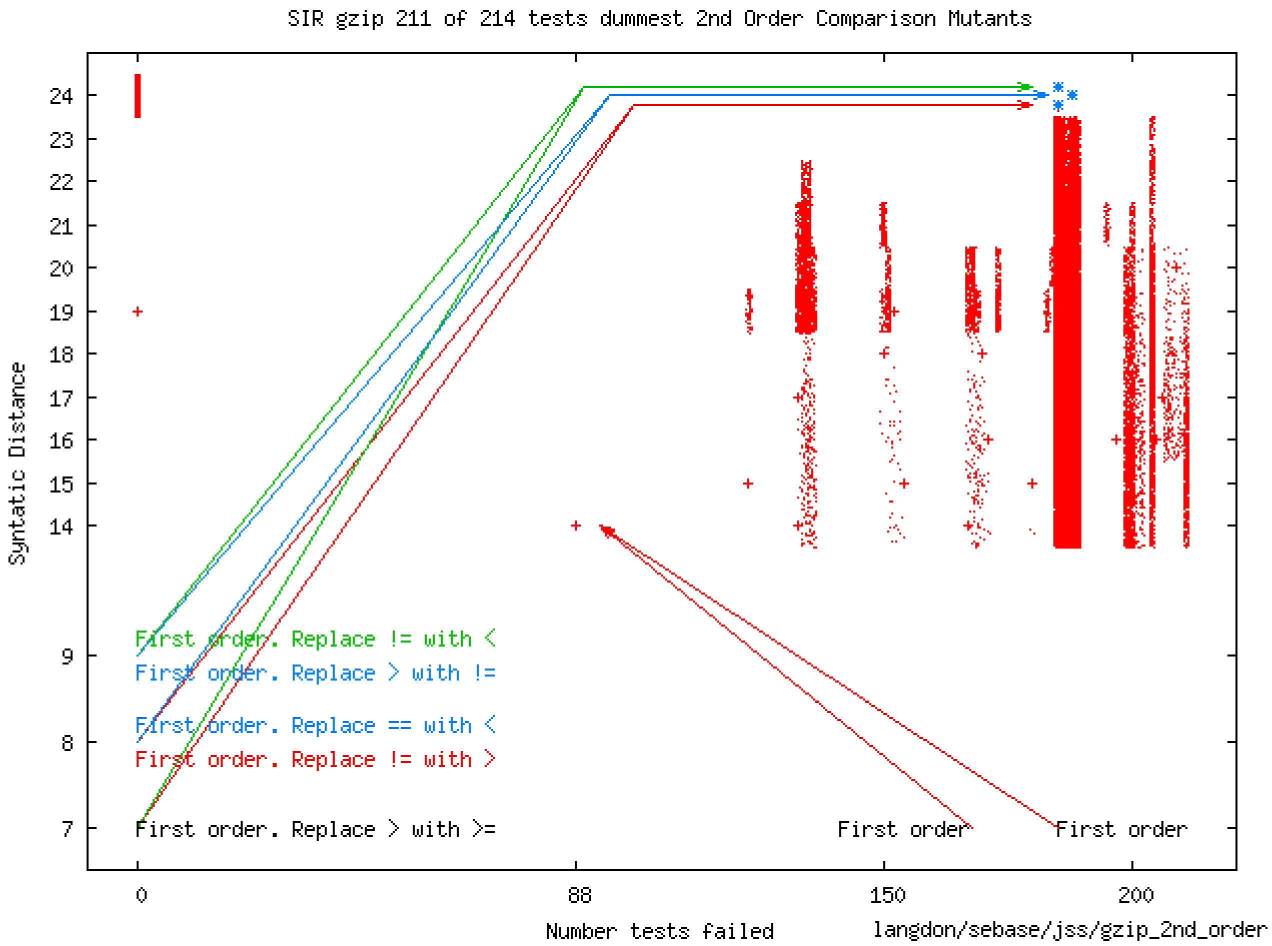


Of 87,150 2nd mutants all except 4 are similar to 1st order.

2^{nd} order interactions of "easy" 1^{st} order

- Two easy to kill 1^{st} partially conceal each other
- Both in for loop of scan_tree()
- if (count < min_count) mutated to <= Fails 89% of tests. Other 1^{st} 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

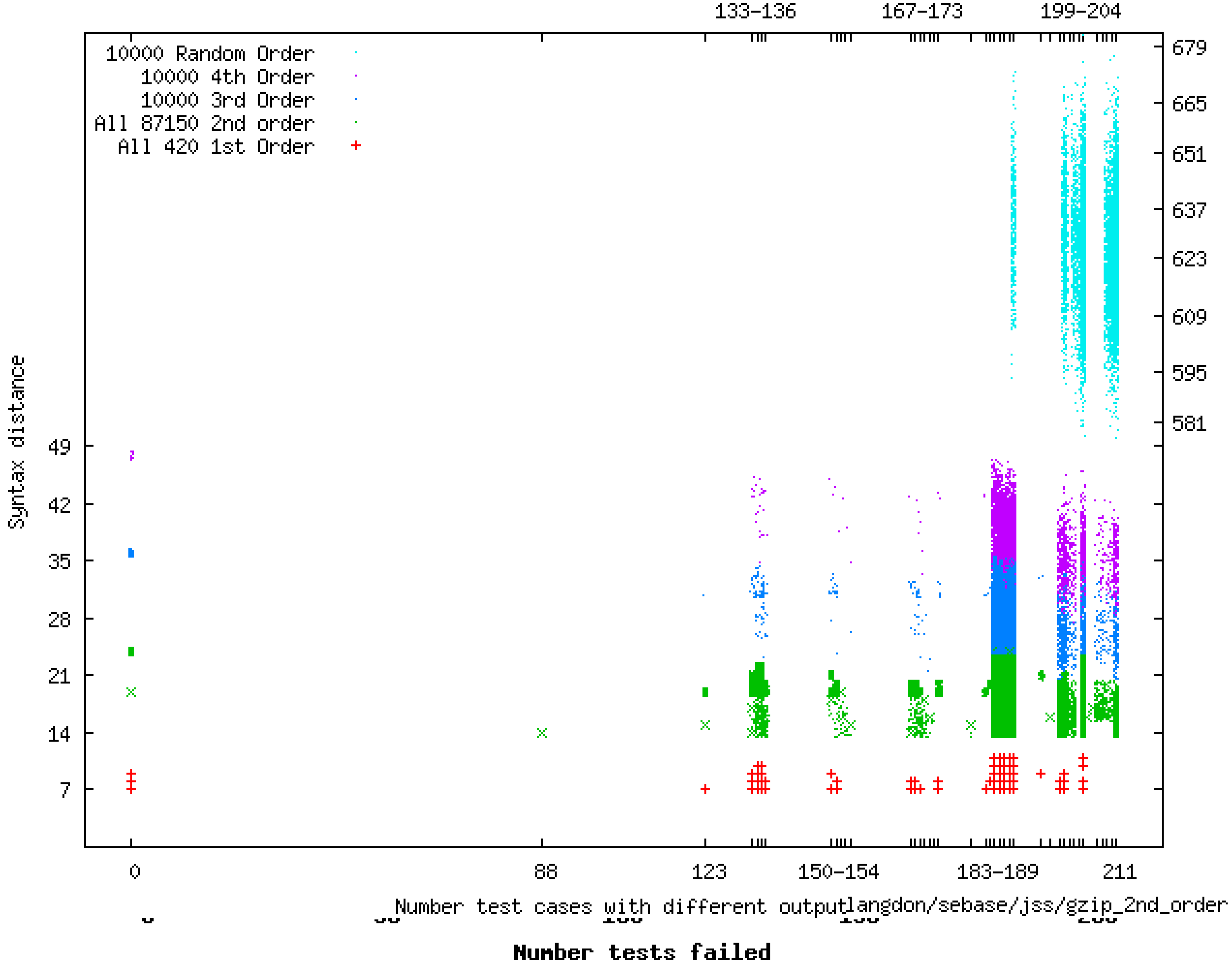


Of 87,150 2^{nd} mutants all except 4 are similar to 1st order.

2^{nd} order interactions of equivelent 1^{st} order

- Two equivelent 1^{st} make each other visible
- Both mutation sites are in longest_match()
- A!= at the start of the outer loop is replaced by <
  != is partially for efficiency, when 1^{st} order makes a difference it is trapped by if 8 lines later
- if(len>best_len) is replaced by (len>= best_len) so the last occurrence of the longest match rather than first is used.
- When both changes are made the if no longer traps the effect of < 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
  - *A Field Guide to Genetic Programming*, **Free**, 2008
  - *Foundations of GP*, Springer, 2002
  - *GP and Data Structures*, Kluwer, 1998