L. Bottaci and G. Kapfhammer and N. Walkinshaw *eds.*, TAIC PART 2009, 4-6 September, p21–29, IEEE press.

Multi Objective Higher Order Mutation Testing with Genetic Programming

William B. Langdon, Mark Harman, Yue Jia
Department of Computer Science, CREST centre, King's College London, Strand, London, WC2R 2LS, UK

***Abstract*—In academic empirical studies, mutation testing has been demonstrated to be a powerful technique for fault finding. However, it remains very expensive and the few valuable traditional mutants that resemble real faults are mixed in with many others that denote unrealistic faults. These twin problems of expense and realism have been a significant barrier to industrial uptake of mutation testing. Genetic programming is used to search the space of complex faults (higher order mutants). The space is much larger than the traditional first order mutation space of simple faults. However, the use of a search based approach makes this scalable, seeking only those mutants that challenge the tester, while the consideration of complex faults addresses the problem of fault realism; it is known that 90% of real faults are complex (i.e. higher order). We show that we are able to find examples that pose challenges to testing in the higher order space that cannot be represented in the first order space.**

I. Introduction

Mutation testing is a fault-injection technique in which a set of mutant versions of a program is created. Usually each mutant is created by the insertion of a single simple fault. The faults are traditionally created by a small syntactic change, such as the replacement of one arithmetic or relational operator with another. If a test input can distinguish between a mutant and the original program, by causing each to produce a different output, then the test input is said to 'kill' the mutant. The effectiveness of a test suite can be assessed by measuring the percentage of mutants that are killed by members of the test suite [1]– [4].

Mutation testing has also been used to simulate other test coverage criteria, such as branch coverage and statement coverage. Indeed, any test adequacy criterion can be simulated by mutation testing. The mutation testing approach has also been used as a basis for test case generation.

Unfortunately, mutation testing is very costly. The number of simple syntactic changes (i.e. first order mutants) that can be performed on a program grows with the size of the program under test, making mutation testing an expensive, if highly effective approach to testing. Furthermore, many of the mutants generated by these simple syntactic fault insertions, are readily killed by the simplest of test cases, leading to much wasted effort killing rather trivial mutants. Though there are a large number of first order mutants, most are simply a waste of time from a testing point of view because they do not denote realistic faults, the vast majority of which are known to be complex (i.e. higher order mutants) [5].

At the heart of this problem lies the very nature of traditional mutation testing. That is, the approach starts with the assumption that a simple syntactic change is typical of a fault. This assumption is known as the competent programmer hypothesis. It states that most programmers are 'competent'; they will produce programs that are within a few keystrokes of being correct. Therefore, based on this hypothesis, faults can be legitimately simulated by a few simple syntactic changes.

However, though the competent programmer hypothesis has been stated as an underlying assumption of mutation testing in many papers, [6], it has not been demonstrated by empirical evidence. Indeed, recent work by Purushothaman and Perry [5] challenges the competent programmer hypothesis. This empirical study found that 90% of post release faults are, in fact, complex faults; faults that can only be fixed only by *several changes* to the syntax of the program at several different places. This observation and the consequent search for these 'subtle' or 'complex' faults was the motivation for recent work on Higher Order Mutation Testing (HOM Testing) [7]. [1]

We take a radical, perhaps even heretical stand point on mutation testing. We assert that mutation testing should be seeking *semantic* mutant programs rather than *syntactic* mutant programs. That is, rather than inserting faults that are syntactically close to the original program, we should be inserting faults that are semantically close to the original program. We explore the relationship between these two notions of similarity; syntactic and semantic in three programs, which are often used as testing benchmarks.

We use a multi objective Pareto optimal genetic programming approach [8] to explore the relationship between mutant syntax and mutant semantics with respect to given test sets. The industrial benchmarks include high quality test sets. A quality test set was created for the other benchmark by selecting test sets that achieve at least branch coverage. The GP algorithm evolves mutant programs according to two fitness functions: semantic difference and syntactic difference. Syntactic distance sums the number of changes weighted by the actual difference. (Details will be given at the end of Section III.) Semantic distance is measured as the number of test cases for which a mutant and original program behave differently. However, should they agree on all test cases, the mutant may be an equivalent mutant, which is undesirable.

[1] A higher order mutant is a program which has had multiple simple changes (first order mutations) made to it.

Therefore, a semantic distance of zero is treated as a special case. (By giving such mutants very poor scores they are normally immediately removed from the population).

A Pareto optimal approach means each objective is treated separately when comparing solutions. Thus a mutant which passes more of the test cases and is closer to the original program is naturally preferred. Similarly if a mutant beats another on one objective but has the same score on the other objective, it is again preferred. Naturally one that is worse on both objectives is not preferred. But when one mutant is better on one objective but worse on the other the two are both nondominant solutions. Figure 9 (towards the end of the paper) contains several "Pareto fronts" each of which contain several mutants. Mutants on the same Pareto front do not dominate each other, even though they pass different number of tests and lie at different distances from the original source. The whole Pareto front is kept and used to explore for further improvements.

The primary novelty and contributions are:

1) This is the first paper to explore the relationship between mutant syntax and semantics and the first to use Pareto optimality in mutation testing (though this has been used in other forms of testing [9], [10]). Pareto optimality, more normally associated with GAs [11], has only been used a little in GP [12, Sec. 3.9]. This is also one of the few papers (other than [13]) to tackle mutation testing using a GP–based approach.
2) We confirm the intuition underlying the well–known Mutation Testing Coupling Hypothesis. I.e. Monte Carlo sampling of higher order mutants confirms the widely held belief that adding test cases to a faulty program tends to make it more error–prone. However, as the result reveal, there remain a non–trivial set of higher order mutants that are hard to kill.
3) The Pareto optimal search is able to find higher order mutants of the TCAS aircraft Traffic alert and Collision Avoidance System program that are harder to kill than any of the first order mutants. This is an example where adding more faults to a program makes it less error–prone; it is harder (though not impossible) to detect the faultiness of the resulting higher order mutant. Such very–hard–to–kill higher order mutants denote highly subtle interactions of faulty behaviour and so may be useful in revealing insight into problem cases and in driving test data generators to generate better quality test data.
4) We show how the exploration of the space of higher order mutants may reveal insights into the structure of the test suite. For example, we found that the differential behaviour of test cases in the presence of higher order mutants creates a clear distinction (in two of the three programs studied) between those test cases that target wrong functionality compared to those that target miss–handled exceptions.

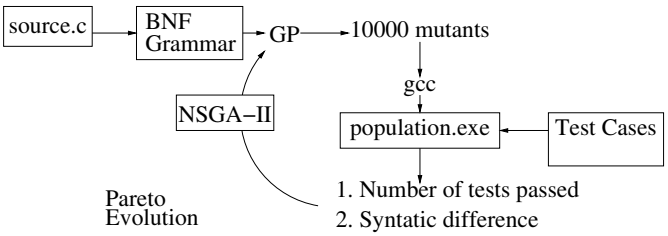

Fig. 1. High order Multi-objective mutation testing. The BNF grammar tells GP where it can insert mutations into the original program `source.c`. Initially GP creates a population of random mutations, which are compiled and run against the test suite providing two objectives to NSGA-II. NSGA-II selects the mutants to retain using a non-dominated Pareto approach and instructs the GP which mutants to recombine or further change. The evolutionary cycle continues for 50 or 500 generations.

The next section outlines how our system works. Section III describes the mutants, whilst Section IV describes how multiple changes are made to C source code. Sections V–VII describe the three benchmarks (Triangle, schedule and tcas) and results obtained. We conclude in Section VIII.

II. How it Works. Optimising two Objectives: Difficulty to Kill and Small Source Changes

Deb's well known Non-dominated Sorting Genetic Algorithm - II (NSGA-II v1.1, [14]) was down loaded. It is a multi-objective evolutionary algorithm, which every generation, uses crossover and mutation to create a new population. The next generation is given by a Pareto optimal selection from both the new offspring and their parents. Thus it is naturally elitist. Fitness sharing is built in to ensure the population does not bunch together on the Pareto front. To adapt it to our GP, `nsga2r.c` was split in two. The first decides which individuals are to be parents and the second combines the offspring and parent populations. Crossover, mutation and fitness calculation are done externally by the strongly typed GP and the multi-objective fitness passed to NSGA-II. Figure 1 shows the relationships between GP, NSGA-II, the grammar and the existing testing regime. (Details will be given in Table II.)

NSGA-II was given two objectives: to minimise the number of tests failed and to minimise the syntactic distance between the mutant and the original program. (Since the goal is not to re-engineer the original program, but to find interesting high order mutants, programs created by GP which behave identically to the original program are penalised by giving them infinitely poor fitness.)

III. Mutants

The Higher Order Mutation Testing Paradigm raises a natural question: what is a higher order mutant? If one were to form a higher order mutant from any possible combination of an arbitrary number of arbitrary first order mutations, then one could use higher order mutation to transform *any* program into *any other* program. Therefore, we think of a set of higher order mutants that is generated by a *chosen set* of first order mutants, rather than allowing arbitrary or unspecified first order mutations. That is, a higher order mutant, is a program

that can be obtained by applying several operations drawn from a set of first order mutations operations, F, to the original program.

This allows us to explore the relationship between the simple faults and the so–called complex faults. Simple faults are the first order mutants. Whilst complex faults are higher order mutants. The higher order mutants are defined by a chosen fault model F or with respect to a certain class of interesting programming language constructs for which we admit first order mutation.

We study the set of first order mutation operators that replace one relational operator with another. This is an interesting set because its corresponding higher order mutant set denotes the set of ways in which one might alter the flow of control within the program. However, the higher order mutants can only influence data flow indirectly, by altering control flow and cannot alter computation by mutating rvalues. In this way, a higher order mutant denotes a 'partially jumbled' computation composed of the same basic computations (arithmetic expressions) as the original.

This approach is based the suggestion that programmers are likely to commit subtle complex faults that might resemble such slightly anomalous control flow. Though this remains a conjecture, we shall see that from this first order set, and for high quality test data and real world programs, it is possible to construct higher order mutants that are harder to kill than *any* of the first order mutants. This lends some evidence to support the belief that the higher order mutants are both interesting and potentially complex in the sense of Purushothaman and Perry [5].

To give a syntactic distance measure, we placed the six C comparison operations < ,<=, ==, !=, >=, > in order. The distance of one comparison from another is their distance in this order plus six if they differ at all. The total distance of a mutant is the sum of the individual distances for each comparison it contains. The constant factor (6) ensures a second order mutant will always have a larger distance than a first order mutant.

Our distance measure tries to capture the idea that some changes are bigger than others and generally more changes make the program more different than fewer. Thus a changing < to <= implies a distance of 7, whilst changing it to a == has a distance of 8. But changing a < to a <= and another a < to a <= (two changes) has a distance of 14.

Notice that the distance is only based on comparing the original program and the final mutant. This distance does not depend upon how many intermediate changes (which may have undone or redone) there have been.

IV. Strongly Typed Grammar Based GP for Mutation Testing

The target source code is automatically analysed to create a BNF grammar tree which describes all its possible mutants. Unlike most BNF's, the grammar consists mostly of terminals which regenerate the fixed portions of the source code. However all comparisons are replaced by the rule <compare>, which is defined thus:

```
<compare>::=  <compare0> | <compare1>
<compare0>::=  <compare00> | <compare01>
<compare00>::=  "<" | "<="
<compare01>::=  "==" | "!="
<compare1>::=  <compare10>
<compare10>::=  ">=" | ">"
```

Notice how three levels of binary choices are needed to cover all six possible comparisons.

Excluding <compare>, each line of the source is converted into a unique rule in the grammar. For efficiency as much of the source code is converted into as few rules as possible. Indeed a line of C code which does not contain any comparison operations is converted into a single rule which has only one production which is itself a terminal containing the whole line. Lines are grouped into a hierarchy by a binary chop process.

For example, in tcas lines 7 to 10 are described by rule <line7-10> (cf. Figure 6) which has two productions: <line7-8> and <line9-10> which each cover two rules.

```
<line7-10>   ::=   <line7-8> <line9-10>
<line7-8>    ::=   <line7> <line8>
<line9-10>   ::=   <line9> <line10>
```

This ensures lines of code that are close together in the source code are close together in the grammar parse tree.

The strongly typed GP uses the name of the grammar rules (i.e. their left hand side) as the rule's type. This means we have more types than is common in (non-grammar based) strongly typed GP. Crossover chooses one crossover point uniformly at random from the first parent's grammar. This gives the type of the crossover point. The crossover point in the second parent must be of the same type. For example, if <line7-10> is chosen in the first parent, then it must also be chosen in the second. Thus, in this example, the child will inherit lines 1–6 and 11 to the program's end from the first parent and lines 7–10 from the second parent. Notice crossover automatically takes advantage of any modularity the programmer explicitly coded in her choice of how to layout the source code. Mutation similarly chooses a rule from the BNF. Say it also chose rule <line7-10>. All the grammar below the chosen point is re-created at random. (Mutation ensures at least one change is made.) Thus a mutation at <line7-10> will randomly replace the comparison operation in line 7 and in line 10. (Lines 8 and 9 do not contain any comparisons.) Implementation details can be found in [15], [16].

Given the rigidity of the grammars we are using to construct mutants, it might be argued that we do not need the expressive power of GP and a simpler evolutionary algorithm could be used. However we automatically get genetic operations which are tailored to the source code we are investigating.

V. The triangle Benchmark

A. Triangle Code and Test Suite

The triangle program is often used as an example in software engineering studies. We used a simplified version of DeMillo *et al.*'s [6] translated from Fortran into C. See Figure 2. It takes the lengths of three sides of a triangle and classifies it as either scalene, isosceles or equilateral or it is not a triangle. (Since the layout chosen by the programmer influences the strength of the crossover linkage between potential mutation sites, the layout in Figure 2. follows that in `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 {
            return 1;
        }

    }

    if(triang > 3){
        return 3;
    }
    else if ( triang == 1 && side1 + side2 > side3) {
        return 2;
    }
    else if (triang == 2 && side1 + side3 > side2){
        return 2;
    }
    else if (triang == 3 && side2 + side3 > side1){
        return 2;
    }

    return 4;
}
```

Fig. 2. `triangle.c`. It contains 17 mutable comparisons: 7 ==, 4 >, 3 < and 3 <=. However there are, initially, no >= or != comparisons.

The test set used by DeMillo *et al.*'s [6] achieved only statement coverage. Therefore we generated test cases covering all the possible branches. (Since we are going to modify the program it is important to cover all branches, including branches containing statements that are unreachable in the original program). However, using this branch coverage test set, in earlier experiments, we found many mutants related to conditional statements were not detected. This was because the test set did not cover some of the Boolean sub-expressions of the conditional statements. Therefore, we extended our test set with the test cases covering all of those Boolean sub-expressions. The final test set contains 60 tests.

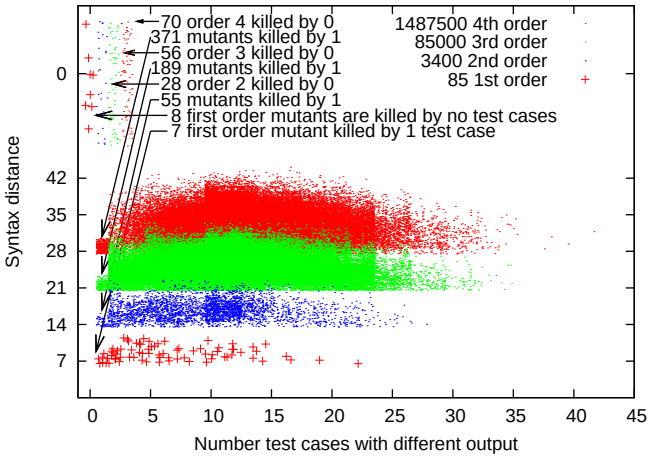

Fig. 3. Fitness of all low order `triangle.c` mutants. (Noise added to spread data.) Remember even though equivalent mutants pass all the test case they are given an infinitely bad score and hence are plotted at the top left.

TABLE I
NUMBER OF LOW ORDER `triangle.c` MUTANTS. THIRD ROW (EQUIV) IS THOSE THAT PASS ALL TEST CASES. LAST ROW (1 TEST) GIVES THOSE THAT FAIL JUST ONE TEST. (FRACTION 10^{-6} OF MUTANTS OF THE SAME ORDER IS GIVEN IN BRACKETS.)

Order	1^{st} (10^{-6})	2^{nd} (10^{-6})	3^{rd} (10^{-6})	4^{th} (10^{-6})
Number	85	3400	85 000	1 487 500
Equiv	8 (94 118)	28 (8 235)	56 (659)	70 (47)
1 test	7 (82 353)	55 (16 177)	189 (2 223)	371 (249)

B. Triangle Mutants

`triangle.c` contains 17 comparison operators. (All of them comparing `int` with `int`.) Therefore there are: $17 \times 5 = 85$ programs with one change, $\frac{17\ 16}{2} \times 5\ 5 = 3400$ with two changes, $\frac{17\ 16\ 15}{2\ 3} \times 5\ 5\ 5 = 85\,000$ with three changes and so on. The total search space is $6^{17} = 16.9267\,10^{12}$.

In Figure 3 we plot the fitness of all the 1.5 million mutants up to order 4. Notice the number of mutants grows exponentially with order and that the fraction of both equivalent mutants[2] and the hardest to detect[3] fall rapidly with number of changes made, cf. Table I. The GP was also able to find these low order and hard to detect mutants but since there are only 85 first order mutants, it is easier to enumerate them.

Ten of our 60 triangle test cases are extremely effective against random mutants. They each individually detect more than 99% of the high order mutants. This causes the vertical concentration of points in the fitness scatter plot in Figure 4.

[2] Equivalent mutants are those that make no detectable difference and so pass all the test cases.

[3] I.e. fail just one test.

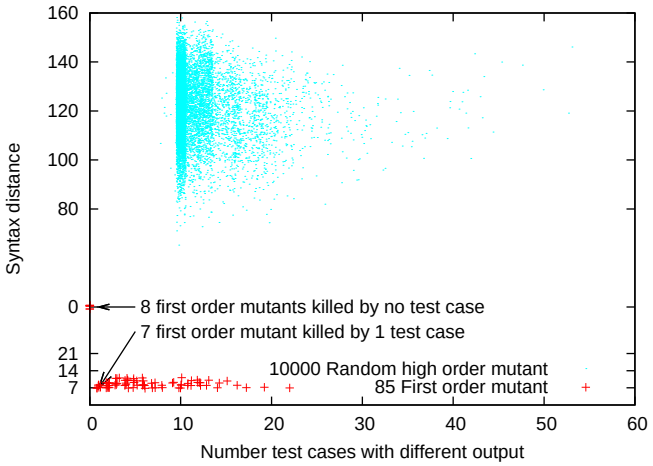

Fig. 4. 10 000 random `triangle.c` mutants (from GP's initial population). Noise added to spread data. 61% fail the same ten tests.

These ten tests are all those that check for normal operation. Whereas the other 50 tests check that `triangle.c` detects "not a triangle". It looks like high order mutants are easily detected by tests for correct operation because correct operation requires more of the code to be executed and so there is more chance of striking the mutated code before one of the `return` statements in Figure 2.

VI. The schedule Benchmark

A. Code and Test Suite, Robustness Improvements

`schedule.c` (SIR version 2.0) was down loaded from the Software-artifact Infrastructure Repository [17]. It consists of 412 lines of code, split into 18 procedures and 2650 test cases. Each test case provides up to five inputs from the command line and reads up to 289 integers from one of the 2650 input files. It produces an ordered list of the scheduled processes. The output consists of up to 45 integers and possibly an error message.

To ease automatic testing, `printf`s were replaced by code to direct the scheduled process identifiers (`int`s) to a buffer and to replace the two textual error messages by two status codes. The outputs generated by the unmodified version of the schedule program (as created by SIR's `runall.sh` script) were converted to the new format. During mutation testing a mutant is said to have failed a test if any of its outputs do not match that of the original program or if its status does not match the original error message (if any).

Unlike `triangle.c`, `schedule.c` accesses arrays, dynamically creates and deletes data structures, uses pointers to them, and runs `for` loops. Thus we are faced with the likelihood that mutants will cause: array indexing errors, run out of memory, corrupt the heap, read or write illegal memory (with unknown consequences) and loops will not terminate. If a mutant does one of these then we say it has failed that test. However we must ensure that a single mutant does not affect the testing of other mutants or itself when running a different test.

Initial experiments showed it would not be feasible to use the normal isolation and protection provided by the operating system. This is because the overhead of creating and starting a separate process per mutant and per test case is too large. Instead all the mutants for a generation are compiled together and run on each test case. In this way it is feasible to test hundreds of thousands of mutants on all 2650 test cases. To allow us to do this additional checks were added to the source code:

- Heap memory large enough for all of the test cases is allocated before testing is started. The original calls to allocate and free memory are replaced by using this area. It is cleared between each test and checks added that it is not exceeded.
- Before any pointer is used, it is checked. I.e. has a legal value consistent with its type.
- Index checks are made before all array accesses.
- Code is added to terminate each `for` loop if the total number of loop iterations for an individual test exceeds ten times the maximum required by the unmodified program.

If any of these checks fail the test is safely aborted and the mutant is said to have failed that test. The next is started knowing it is safe to do so and it will not be affected by the previous failure.

B. Schedule Mutants

`schedule.c` contains 14 comparison operators. (All of them comparing `int` with `int`.) Therefore there are 70 first order mutants. The number of tests which detect them is plotted in Figure 5. Ten of the 70 make no visible difference but one first order mutant fails a single test.

Figure 5 shows random high order `schedule.c` mutants are easily detected. Most of the tests are good at finding them. Even the worst test detects most high order mutants. `schedule.c` is much more complicated than the triangle program. Perhaps this is why multiple mutations scattered at random are more easily detected than in `triangle.c`. The GP was also able to solve the problem, but since there are only 70 first order mutants, it is easy to enumerate them.

VII. The tcas (aircraft Traffic alert and Collision Avoidance System) Benchmark

A. tcas Benchmark, Robustness Improvements, Creating and Testing Mutants

tcas (SIR version 2.0) was down loaded from the Software-artifact Infrastructure Repository [17]. It consists of 135 lines of C code (excluding comments and blank lines) with 40 branches [18, Table 2] and 1608 test cases with up to 12 input parameters and one output. The supplied `main()` was recoded to return tcas' answer to the GP rather than printing it. Similarly when tcas detects an error, instead of it printing an error message and `exit`ing, it returns a unique error code to the GP.

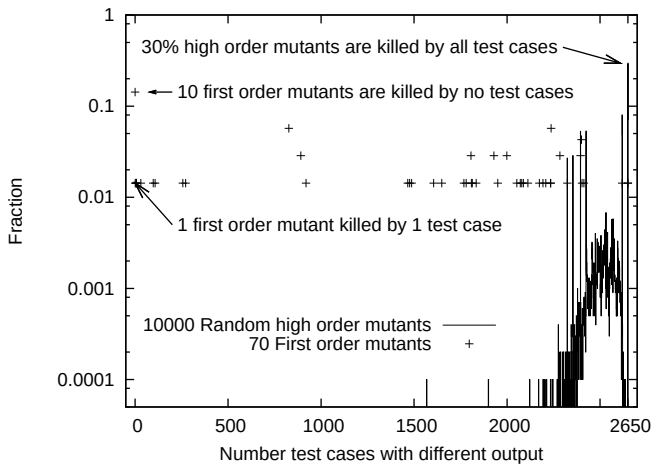

Fig. 5. Fitness of all first order (+) and 10 000 random `schedule.c` mutants. Note log scale. More than 90% of high order mutants fail more than 90% of the test cases.

`tcas.c` contains an array but no array index checking. This leads to array bound errors, which we have reported. A check was added. This ensures mutants cannot crash the GP and they cannot affect the execution of each other. This check cannot be mutated by the GP. Similarly the existing array index check (on `argv` in tcas' `main`) cannot by mutated.

There are 14 comparison operations non-uniformly scattered through the 8 mutable functions. A BNF grammar (cf. Figure 6) describing the mutable functions was automatically created. Each generation our strongly typed GP system used the grammar to generate up to 10 000 mutations of tcas. These were compiled together by `gcc` into a single executable. Each mutant was run on all 1608 tests and the number of times it produced a different answer was recorded. (A small efficiency improvement could have been made by not running those tests which are trapped by immutable tests. Since these are the array index changes, which GP cannot change, the mutant must pass these tests.)

As is usual in GP, the initial population was constructed from random individuals. (I.e. very high order mutants, in which almost all comparisons are changed.) We anticipate finding interesting mutants which are good at passing tcas's test cases and not too dissimilar from it. As we have already seen in the triangle and schedule programs (Figures 4 and 5) and confirmed by Figure 7, random high order mutants are naturally some distance away from the goal. An alternative would be to start the evolving population nearer the anticipated solutions by seeding it with low order mutants. This would undoubtably introduce a bias, which might be beneficial. However, if successful, seeding would tend to confirm our initial assumptions, rather than challenge them. Therefore we chose to avoid this particular bias and allow evolution to move the population. We tried two strategies to allow the population to move some distance: a large population (10 000) for 50 generations and a small population (100) for 500 generations, cf. Table II. Both worked and came to different solutions.

TABLE II
STRONGLY TYPED GRAMMAR GP TO FIND HARD TO KILL TCAS MUTANTS

Primitives:	The function and terminal sets are defined by the BNF grammar (cf. Figure 6). BNF rules with two options correspond to binary GP functions. The rest of the BNF grammar correspond to GP terminals.
Fitness:	Two objectives. 1) minimise the number of tcas test cases failed 2) minimise the syntactic difference (Section III) from `tcas.c`. However programs which pass all test cases are treated as if they failed `INT_MAX` tests.
Selection:	NSGA-II. I.e. Pareto multi-objective rank based binary tournament selection on combined current and offspring populations.
Population	size = 100 or 10000
Initial pop:	Ramped half-and-half 3:7
Parameters:	90% subtree crossover. 10% subtree mutation. Max tree depth 17 (no tree size limit)
Termination:	500 or 50 generations

B. tcas Mutants

Since for each comparison there are five possible mutations, there are 70 (5×14) first order mutations. The fitness (i.e. semantic and syntactic differences from tcas itself) are plotted in Figure 7. About a third (24) of the 1^{st} order mutants are not discovered by any of the 1608 test cases. Many of the rest are fairly easy to find and fail many tests. However there is one first order mutant which fails only three tests.

Monte Carlo sampling (cf. dots in Figure 7) shows there are 264 tcas tests which defeat 98.33% of random programs but 428 (Figure 8) which are passed by all 10 000 high order mutants.

Figure 8 plots the expected output from tcas for each test case. The three groups identified in the previous paragraph (cf. also Figure 7) are high lighted by sorting the tests by their effectiveness against random high order mutants. As with triangle and schedule, the most effective tests are those that check for more than default operation. For tcas this means all the tests which expect either UPWARD_RA or DOWNWARD_RA (i.e. aircraft collision threat identified) are highly effective. A few of the 428 tests which are always ineffective are due to non-mutable error detection (lower two set of points in Figure 8). Most of the tcas test suite checks for UNRESOLVED. Some of these are totally ineffective against random mutations but most are fairly poor and find only about 1.67% of them.

C. Running the Pareto GP on tcas

Figure 9 shows NSGA-II progressively improved the initial random high order mutants. It both reduced the number of test cases able to find them and their syntactic distance from tcas. In generation 13 a 7^{th} order mutant was found which is killed by only one test. In generation 44 a 5^{th} order mutant was found which is defeated by only one test. Although the 5^{th} order mutant is a subset of the 7^{th}, i.e. the 7^{th} has two additional changes, they behave differently and fail on different tests. No mutant which failed on two tests was found in this run but the second run (Section VII-D3) found several.

```
<line1>::= "bool Non_Crossing_Biased_ClimbXXX()\n"
<line2>::= "{\n"
<line3>::= "int upward_preferred;\n"
<line4>::= "int upward_crossing_situation;\n"
<line5>::= "bool result;\n"
<line6>::= "\n"
<line7>::= "upward_preferred = Inhibit_Biased_Climb()"
    <compare> "Down_Separation;\n"
<line8>::= "if (upward_preferred)\n"
<line9>::= "{\n"
<line10>::= "result = !(Own_Below_ThreatXXX()) ||
    ((Own_Below_ThreatXXX()) && (!(Down_Separation"
    <compare> "ALIM())));\n"
<line11>::= "}\n"
<line12>::= "else\n"
<line13>::= "{\n"
<line14>::= <line14A> <line14B>
<line14A>::= "result = Own_Above_ThreatXXX() &&
    (Cur_Vertical_Sep" <compare> "MINSEP) &&
    (Up_Separation"
<line14B>::= <compare> "ALIM());\n"
<line15>::= "}\n"
<line16>::= "return result;\n"

                          ⋮

<start>::= <line1> <line2> <line3> <line4> <line5>
    <line6> <line7-30> <line31-54> <line55>
    <line56> <line57> <line58> <line59> <line60>
    <line61> <line62> <line63> <line64> <line65>
    <line66> <line67> <line68> <line69> <line70>
    <line71> <line72> <line73>
<line7-30>::= <line7-17> <line18-28> <line29> <line30>
<line7-17>::= <line7-10> <line11-14> <line15>
    <line16> <line17>
<line7-10>::= <line7-8> <line9-10>
<line7-8>::= <line7> <line8>
<line9-10>::= <line9> <line10>
<line11-14>::= <line11> <line12> <line13> <line14>
<line18-28>::= <line18> <line19> <line20> <line21>
     <line22> <line23> <line24> <line25-26> <line27-28>
<line25-26>::= <line25> <line26>
<line27-28>::= <line27> <line28>
<line31-54>::= <line31> <line32-43> <line44-54>
<line32-43>::= <line32-35> <line36-39> <line40>
    <line41> <line42> <line43>
<line32-35>::= <line32> <line33> <line34> <line35>
<line36-39>::= <line36> <line37> <line38> <line39>
<line44-54>::= <line44-49> <line50-54>
<line44-49>::= <line44> <line45> <line46> <line47>
    <line48> <line49>
<line50-54>::= <line50> <line51> <line52-53> <line54>
<line52-53>::= <line52> <line53>
```

Fig. 6. Fragments of Backus-Naur form grammar used to specify tcas mutants. In total it contains 104 rules. (Common code which is not mutable is excluded. This gives a minor efficiency gain.) `<compare>` is defined in Section IV. `XXX` is replaced by the individual mutant's identification before it is compiled.

D. Hard to Detect tcas Mutants

At the left hand side of Figure 9 there are two points ("Gen 14-44" and "Gen 45-50") above $x = 1$. They represent two high order tcas mutants which pass all but one test.

1) Seventh Order tough tcas Mutant:
The mutant changes lines 87 in function `Non_Crossing_Biased_Climb()` (twice), 101 in `Non_Crossing_Biased_Descend()` (twice), 112 in `Own_Below_Threat()` 117 in `Own_Above_Threat()` and line 127 in `alt_sep_test()`.

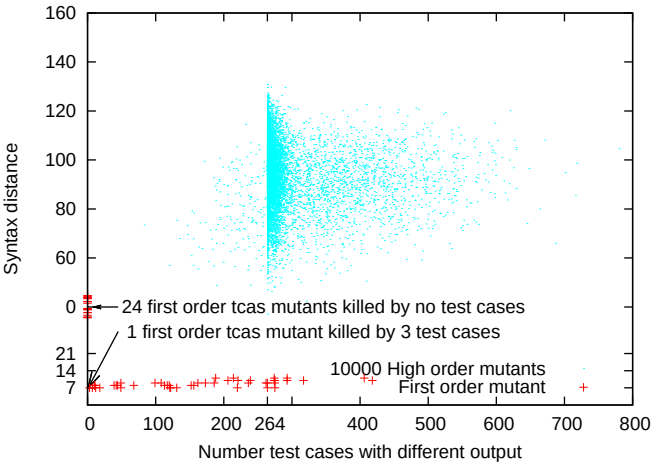

Fig. 7. Fitness 70 first order tcas mutants (+) and 10 000 random high order tcas mutants (dots).

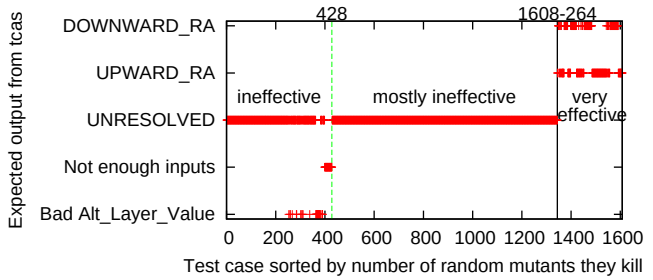

Fig. 8. Expected value returned by tcas v. effectiveness of test against random mutants. The effectiveness of tcas tests falls into three main groups (highlighted by vertical lines) with little difference between members of the same group. On the right are 264 tests which check for threat of collision. They are failed by almost all high order mutants.

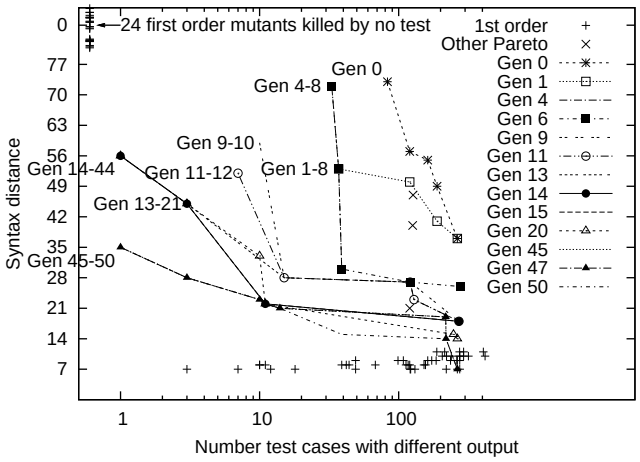

Fig. 9. Evolution of tcas NSGA-II Pareto front with 10 000 mutants per gen

a) line 127 input 10 v. `NO_INTENT` $= \rightsquigarrow \leq$*:*
In normal operation this change would have no impact since it tests one of the twelve inputs directly, and `NO_INTENT` is the smallest of its legal values. (Input 10 is not used elsewhere by tcas.) The tcas test cases include 18 illegal values for input 10, seven of which are less than `NO_INTENT` (0). This would suggest that as a first order mutant, it would be easier to detect

than average. However its effect is totally masked by the rest of tcas. I.e. as an isolated first order mutant this change to line 127 is not detected by the test suite.

b) lines 112 & 117, Comparing inputs 4 & 6, $< \rightsquigarrow \leq$: These two mutants can be thought of as a pair. They occur in paired routines `Own_Below_Threat()` and `Own_Above_Threat()` and both compare inputs 4 and 6. Further the two routines are used together.

Replacing $<$ by $\leq$ clearly can only change behaviour when inputs 4 and 6 are equal. (Again neither input 4 nor 6 are modified by tcas.) There are 23 such test cases. Therefore either mutation by itself could in principle fail up to 23 tests. However 22 of these are masked by the combined effects of tcas itself and the other six changes.

Oddly, the first order mutation on line 112 (input 4 $<$ input 6 $\rightsquigarrow$ input 4 $\leq$ input 6) is masked by the rest of tcas in 11 of the 23 test cases. However the very similar mutation on line 117 (input 6 $<$ input 4 $\rightsquigarrow$ input 6 $\leq$ input 4) is always masked and so is equivalent.

c) lines 101 and 87, checking inputs 1 and 8: There are two mutations on each line. We group these four mutations together since they appear in the same location in two complementary routines, `Non_Crossing_Biased_Climb()` and `Non_Crossing_Biased_Descend()`. Again the pair of routines are used together. In line 87 input 1 $\geq$ 300 && input 8 $\geq$ `ALIM()` is mutated to input 1 $<$ 300 && input 8 $\leq$ `ALIM()` (Again neither input is modified by tcas.) Line 101 has been mutated in somewhat similar way input 1 $\geq$ 300 && input 8 $\geq$ `ALIM()` becomes input 1 $\neq$ 300 && input 8 $\leq$ `ALIM()`.

As isolated first order mutants all four pass all the tests. This may be because they are nested within routines which themselves are nested in the logic of tcas so that the comparisons are seldom made when tcas is run. However they appear to interact with the three other mutations to make the combination very tough to test against.

2) Fifth Order tough tcas Mutant: Referring back to the left hand side of Figure 9 we see evolution continues after generation 14 so that in generation 45 a fifth order mutant is discovered which also passes a single test. Since it syntactically closer to tcas, it replaces the seventh order mutant on the Pareto front.

While not an immediate descendent of the 7^{th} order mutation it is similar and was probably found through an intermediate cousin.

The 5^{th} order mutation contains the same last 5 changes as the 7^{th} order one. I.e. it is the same except for line 87. However, while the two high order mutations both fail just one test, it is a different test. In fact its a different one of the 23 tests where inputs 4 and 6 are equal. Note, removal of two equivalent mutations has actually changed the behaviour of tcas.

3) Third Order tough tcas Mutant: A separate GP run with a small population (100) but more generations found two more high order mutants which are defeated by a single test case. In generation 90 a fourth order mutant was found. This was replaced in generation 105 by a third order mutant. Again they are similar. The third order mutant is identical to the fourth except it does not include the mutation to line 117. For brevity we shall just described the third order mutant.

The mutant changes lines 101 in `Non_Crossing_Biased_Descend()` (once), 112 in `Own_Below_Threat()` and 117 in `Own_Above_Threat()` in the *same* way as described in Section VII-D1.

Although this mutant only contains three of the seven mutations described in Section VII-D1 it fails the same test (test 1400). (Which is different from that failed by the 5^{th} order, which differs by two of the same four changes.) Test 1400 (like all the other tcas tests) was taken from the `runall.sh` script provided by SIR. It runs tcas with 12 command line arguments: `tcas 601 1 0 502 200 502 0 599 400 0 0 1` The third order mutant yields 2 (DOWNWARD_RA) whereas the original program prints 0 (UNRESOLVED).

These results tend to suggest that if further testing effort were available it might be concentrated around lines 87, 101, 112, 117 and possibly 127. Note all twenty first order mutations to lines 87 and 101 are equivalent.

VIII. Conclusions

We have introduced a new form of mutation testing, reformulating traditional mutation testing as a multi objective search problem in which the goal is to seek higher order mutants that are hard to kill and syntactically similar to the original program under test. The approach uses higher order mutation testing, but subsumes traditional mutation testing since a first order mutant is also a special case of a higher order mutant (for which the order is simply 1).

Using the search based approach, we are able to specifically seek the mutants that denote more realistic complex faults that are also hard problems for testing. We have implemented this approach using genetic programming (GP) on a normal office personal computer running Linux and report results on three cases studies including two real world programs together with the Triangle 'benchmark' example. The results demonstrate st144that in a few minutes the Higher Order GP Mutation testing approach is able to find complex faults denoted by (non–equivalent) higher order mutants of real programs that cannot be denoted by *any* first order mutant and which are *harder to kill than any first order mutant*.

Acknowledgment

We would like to thank the Software-artifact Infrastructure Repository `sir.unl.edu` and the authors of NSGA `http://www.iitk.ac.in/kangal/`.

REFERENCES

[1] S. Kim, J. A. Clark, and J. A. Mcdermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," in *Mutation Testing for the New Century*, vol. 11, 2001, pp. 207–225.

[2] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon, "MuJava: an automated class mutation system," *Software Testing, Verification & Reliability*, vol. 15, no. 2, pp. 97–133, June 2005.

[3] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva, "SQLMutation: A tool to generate mutants of SQL database queries," in *Proceedings of the Second Workshop on Mutation Analysis*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 1.

[4] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, 2008, pp. 52–61.

[5] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, pp. 511–526, 2005.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[7] Yue Jia and M. Harman., "Constructing subtle faults using higher order mutation testing," in *SCAM'08,8th International Working Conference on Source Code Analysis and Manipulation*, 2008.

[8] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, (With contributions by J. R. Koza). [Online]. Available: http://www.gp-field-guide.org.uk

[9] Shin Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*, 2007.

[10] M. Harman, K. Lakhotia, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, 2007.

[11] C. M. Fonseca and P. J. Fleming, "Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization," in *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, S. Forrest, Ed. University of Illinois at Urbana-Champaign: Morgan Kaufmann, 17-21 July 1993, pp. 416–423.

[12] W. B. Langdon, *Genetic Programming and Data Structures*. Boston: Kluwer, 1998. http://www.cs.ucl.ac.uk/staff/W.Langdon/gpdata

[13] M. C. F. P. Emer and S. R. Vergilio, "Selection and evaluation of test data based on genetic programming," *Software Quality Journal*, vol. 11, no. 2, pp. 167–186, June 2003.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr 2002.

[15] W. B. Langdon and A. P. Harrison, "Evolving regular expressions for GeneChip probe performance prediction," Computing and Electronic Systems, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, UK, Tech. Rep. CES-483, 27 Apr. 2008. [Online]. Available: http://www.essex.ac.uk/dces/research/publications/technicalreports/2008/CES-483.pdf

[16] W. B. Langdon and A. P. Harrison, "Evolving DNA motifs to predict GeneChip probe performance," *Algorithms in Molecular Biology*, vol. 4, no. 6, 19 Mar. 2009. [Online]. Available: http://dx.doi.org/doi:10.1186/1748-7188-4-6

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of 16th International Conference on Software Engineering, ICSE-16*, May 1994, pp. 191–200.

[18] Hao Zhong, Lu Zhang, and Hong Mei, "An experimental study of four typical test suite reduction techniques," *Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008. [Online]. Available: http://dx.doi.org/doi:10.1016/j.infsof.2007.06.003