Efficient 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

It is said ninety percent of faults that survive manufacturer's testing procedures are complex. That is, the corresponding bug fix contains multiple changes. Higher order mutation testing is used to study defect interactions and their impact on software testing for fault finding. We adopt a multi-objective Pareto optimal approach using Monte Carlo sampling, genetic algorithms and genetic programming to search for higher order mutants which are both hard-to-kill and realistic. The space of complex faults (higher order mutants) is much larger than that of traditional first order mutations which correspond to simple faults, nevertheless search based approaches make this scalable. The problems of non-determinism and efficiency are overcome. Easy to detect faults may become harder to detect when they interact and impossible to detect single faults may be brought to light when code contains two such faults. We use strong typing and BNF grammars in search based mutation testing to find examples of both in ancient heavily optimised every day C code.

Keywords: Pareto optimality, mutation testing, higher order mutation, SBSE, Monte Carlo, genetic algorithm, genetic programming, NSGA-II, strongly typed GP, grammar based GP, non-determinism, triangle, schedule, tcas, gzip

1. Introduction

Mutation Testing is a widely studied fault based software testing technique, in which simulated faults are deliberately inserted into a program under test in order to create a faulty version of the original called a mutant (Jia and Harman, 2009a; Offutt and Untch, 2001). 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. A test suite kills a mutant if any of its constituent test cases 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 (Kim et al., 2001; Ma et al., 2005; Tuya et al., 2006; Ferrari et al., 2008). The percentage of mutants killed by a test suite is known as the 'mutation score' for the test suite.

There are two ways in which Mutation Testing can be used. It can be used to measure the effectiveness of a test suite using the mutation score. Of course, we cannot expect to know all the real faults that might be found in a program; if we did, then we would not be testing it. This is where Mutation Testing has a role to play: we use mutants as a way of simulating real faults. This 'fault simulation mode' is the most widely studied mode of application of Mutation Testing and it is also our topic.

However, Mutation Testing has also been advocated as a technique for generating test data (DeMillo and Offutt, 1991; Ayari et al., 2007). In test data generation mode, test inputs are generated in order to kill the mutants seeking to achieve as high a mutation score as possible. In this way, killing mutants with test data can be seen as an attempt to meet a mutation-inspired test adequacy criterion; one that seeks to cover mutants rather than code.

Indeed, since mutants can be placed anywhere in the code, mutation coverage can be used as surrogate for almost any other form of structural coverage. For instance, by placing mutants at suitable locations in every branch, it is possible to simulate branch coverage using mutation coverage Offutt et al. (1996b). This makes mutation testing a highly generic and flexible approach to software testing.

A mutant is called a first order mutant if it results from a single change to the program under test, while it

is higher order if it is created by more than one change. Unfortunately, many of the mutants created by a single simple syntactic change (thereby creating a first order mutant) are unrealistic; they do not simulate real faults. Evidence to support this claim is provided by empirical results that indicate that in the vast majority of cases, about 90%, a modification made to fix one real fault needs several source code changes (Purushothaman and Perry, 2005). A single fault that is fixed by several changes can only be denoted by a mutant that is constructed by the insertion of several changes. Such a mutant is, by definition, a higher order mutant. We provide a more detailed critique of first order mutation testing and a motivation for the move to the higher order paradigm elsewhere (Jia and Harman, 2009b; Harman et al., 2010).

Our aim is to explore the relationship between a mutant's syntax and its semantics. For instance are larger syntax changes always worse than smaller ones or are there higher order mutants which are closer to the original program's semantics? We answer this by using search techniques. One objective is mutants with similar semantics (small semantic distance). The second is inspired by Occam's razor: we look for minimal syntactic changes. The hope is that by making smaller changes to the source code, the mutations will be more intelligible and so more useful. A Pareto optimal approach means each objective is treated separately when comparing solutions. Thus, a mutant that passes more of the test cases and is syntactically 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, a mutant that is worse on both objectives is not preferred. However, when one mutant is better on one objective but worse on the other, the two mutants are both considered to be incomparable (or "nondominant").

This is illustrated in Figures 1 and 10 which contain several "Pareto fronts", each of which contain several mutants. Mutants on the same Pareto front do not dominate one another, even though they pass different numbers of tests and lie at different distances from the original source. The entire Pareto front is kept and used to explore for further improvements.

We use a novel Multi Objective Pareto Optimal Genetic Programming (GP) approach (Poli et al., 2008) to explore the relationship between mutant syntax and mutant semantics. We use several real world programs as subjects. They are drawn from the Software-artifact Infrastructure Repository (Hutchins et al., 1994). These SIR benchmark programs include high quality test sets.

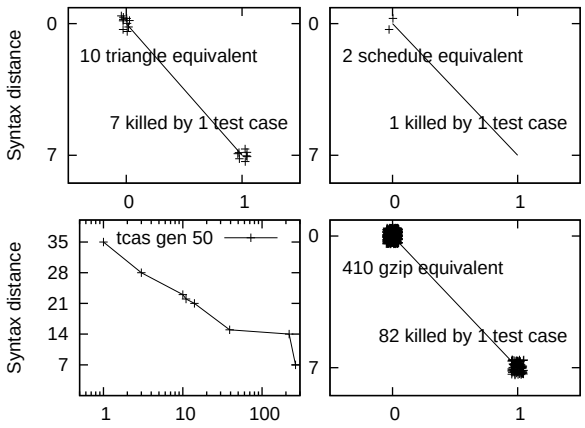

Figure 1: Pareto fronts. Top row: triangle and schedule first order mutants. Bottom row: tcas GP evolved mutants (see Figure 10) and gzip first order mutants. Equivalent mutants are not killed by any test cases. They are treated as special cases and given very poor scores (indicated by "0" on the y-axis). The smallest syntactic distance for a mutant is 7. Noise added (except tcas) to spread data.

The GP algorithm evolves mutant programs according to two fitness functions: semantic difference and syntactic difference. As we have extended mutation testing to allow more than one syntactic change to the source code, we still need to constrain the total change to the program. We anticipate that the number of changes will be small, but we do not know exactly how many. By using a multi-objective approach we can avoid imposing an arbitrary limit and give the optimisation process the freedom to find interesting mutants which tradeoff syntactic change against impact on the program's semantics.

Our syntactic distance sums the number of changes weighted by the actual difference. (Details will be given at the end of Section 2.) 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. For example, if `i` is known to be a non-negative integer then `if(i> 0)` and `if(i!=0)` are equivalent (since no external test can ever tell them a part). As equivalent mutants are indistiquishable, we do not want to burden the tester with complaints that the test suite does not kill them. (The problem of potential equivalent mutants is well known in mutation testing.) 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).

We focus on the set of relational operator mutants, since these control the logical control structure of the program under test. We use genetic programming, to-

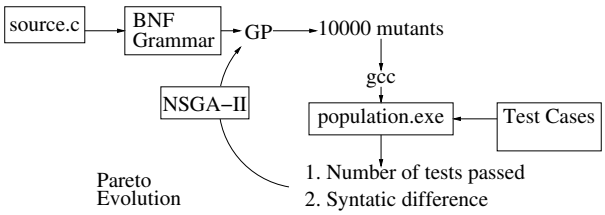

Figure 2: High order Multi-objective mutation testing. The BNF grammar tells GP where it can insert mutations into the original program `source.c`. Initially GP creates a population of random mutations, which are compiled and run against the test suite providing two objectives to NSGA-II. NSGA-II selects 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.

gether with Monte Carlo sampling, to explore the space of higher order mutants. We are particularly interested in cases where, within a class of first order mutants, a higher order mutant exists that is harder to kill than every one of the first order mutants. This is theoretically interesting because it demonstrates that there exist situations in which a tester is forced to move to the higher order paradigm in order to find tougher faults. It is also practically interesting, because it highlights situations in which known first order faults can combine in subtle ways to create partly masked faults that may have gone undetected.

The contributions[1] are:

1. This is the first paper to simultaneously explore the relationship between source code changes introduced by mutations and corresponding changes to the program's behaviour. It is also the first to use Pareto optimality in mutation testing (though this has been used in other forms of testing (Yoo and Harman, 2007; Lakhotia et al., 2007)). Pareto optimality, more normally associated with GAs (Fonseca and Fleming, 1993), has only been used a little in GP (Langdon, 1998, Sec. 3.9). This is also one of the few papers (other than Emer and Vergilio (2003)) to tackle mutation testing using a GP–based approach.
2. We confirm the intuition underlying the well–known Mutation Testing Coupling Hypothesis (DeMillo et al. (1978)): "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" (Offutt, 1992, p 6). Monte Carlo sampling of higher order mutants confirms the widely held belief that adding changes to a faulty program tends to make it more error–prone. However, as the results 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. Up till now the importance of non-deterministic faults has not been recognised in mutation testing (see Table 1). Inserting errors could lead to non-repeatable behaviour in programs coded in any language. For example, due to the introduction of timing problems. In C programs, the injection of faults may lead to variables, particularly local arrays, not being fully initialised. The program's subsequent behaviour may now depend on the exact values left in the storage which can vary between program runs.

   In one case this was non-deterministic, so that testing could detect the mutant approximately two times in three. To reliably detect (i.e. more than 95% of the time) even such a frequent error would entail running tests at least three times. Of course less frequent errors would require tests to be repeated more times. For example, if the chance of a test failing is 3% (rather than 67%) then the test has to be repeated 100 times to reliably detect the fault.
5. 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 three of the four programs studied) between those test cases that target wrong functionality compared to those that target mishandled exceptions.

[1] Some of these contributions were first made, albeit with respect to a smaller set of subject code, by the conference version of this paper Langdon et al. (2009). However, results relating to improved test efficiency and non–determinism are entirely novel to this extended journal version.

Table 1: Practical details for efficient mutation testing

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.
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

6. We include several methods for reducing machine resources during intensive testing, particularly when there are many program variants and test cases. See Figure 2 and Sections 6.1 and 8.3.

The next section describes the mutants, Section 3 outlines how our system works, whilst Section 4 describes how multiple changes are made to C source code. Sections 5–8 describe the four benchmarks (Triangle, schedule, tcas and gzip) and results obtained. Section 9 discusses threats to validity and potential future work. We conclude in Section 10.

2. 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 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 on 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 (2005).

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.

3. 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, Deb et al. (2002)) was used. It is a multi-objective evolutionary algorithm, which at 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 2 shows the relationships between GP, NSGA-II, the grammar and the existing testing regime. (Details will be given in Table 4.)

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.)

4. Strongly Typed Grammar Based GP for Mutation Testing

Initially the target source code is automatically analysed to create a BNF grammar tree which describes all its possible mutants. In the case of gzip, which is much bigger, additional optimisations were also used (see Section 8.3). 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> ::=  ">=" | ">"
```

For simplicity of integrating the grammar into the GP search process, BNF rules with two alternatives are supported. Thus 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>` (see Figure 7) 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 Langdon and Harrison (2008, 2009).

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.

Table 2: The 14 test cases for `triangle.c`.

Inputs			Expected Output	
0	0	0	4	not a triangle
1	0	0	4	not a triangle
1	1	0	4	not a triangle
1	1	1	3	Equilateral
2	2	1	2	Isosceles
1	1	2	4	not a triangle
2	1	2	2	Isosceles
1	2	1	4	not a triangle
2	1	1	4	not a triangle
3	2	2	2	Isosceles
3	2	1	4	not a triangle
4	3	2	1	Scalene
2	3	1	4	not a triangle
2	1	3	4	not a triangle

5. The triangle Benchmark

5.1. 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. (1978) translated from Fortran into C. See Figure 3. 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 3. follows that in `triangle.c`.)

The test set used by DeMillo et al. (1978) 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). 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 used CUTE (Sen et al., 2005) to ensure 1) all branches are covered 2) the test set reaches every Boolean sub-expression in the `if` statements. The final test set is given in Table 2.

5.2. 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 $6^{17} = 16.9267\ 10^{12}$.

```
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;
}
```

Figure 3: `triangle.c`. It contains 17 mutable comparisons: 7 `==`, 4 `>` and 6 `<=`. However there are, initially, no `<`, `>=` or `!=` comparisons.

Table 3: Equivalent and hardest to kill first and higher 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 (i.e. hardest to detect). Data are the numbers and the percentage of mutants of the same order.

Order	1^{st}	2^{nd}	3^{rd}	4^{th}
No.	85	3400	85 000	1 487 500
Equiv	14 16%	89 3%	340 0.4%	870 0.06%
1 test	18 21%	325 10%	2615 3.1%	12363 0.83%

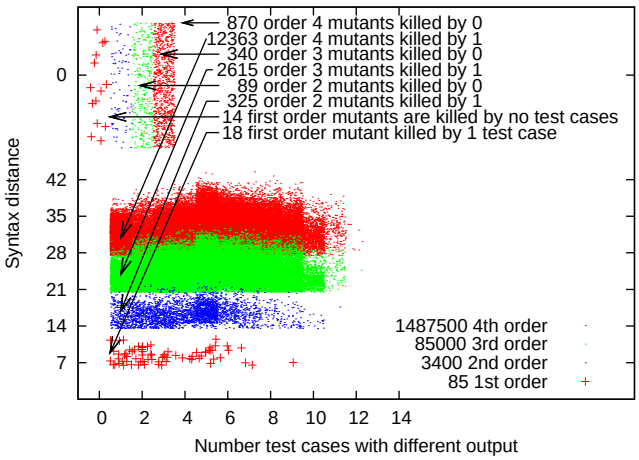

Figure 4: Fitness of all low order `triangle.c` mutants. (Noise added to spread data.) Remember even though potential equivalent mutants pass all the test cases they are given an infinitely bad score and hence are plotted at the top left.

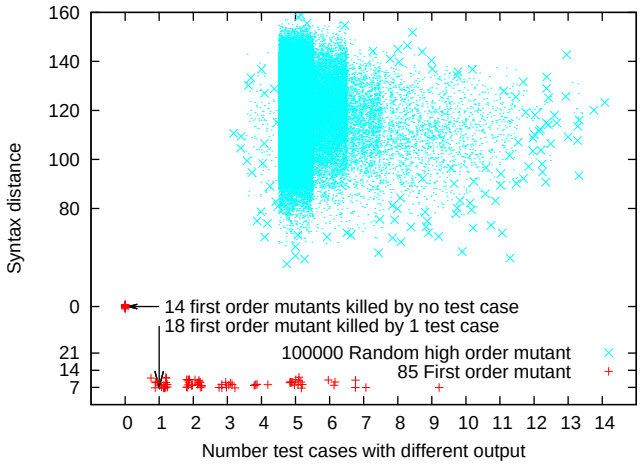

Figure 5: 100 000 random `triangle.c` mutants (from GP's initial population). Noise added to spread data.

In Figure 4 we plot the fitness of all the 1.5 million mutants up to order 4. Notice 1) the number of mutants grows exponentially with order. 2) The fraction of both potential equivalent mutants[2] and the hardest to detect mutants[3] falls rapidly with number of changes made, see Table 3. 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.

Five of our 14 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 5. These five tests are all those that check for normal operation. Whereas the other nine 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 3.

[2] Potential equivalent mutants are those where the mutation made no detected difference. They always pass all the test cases but we cannot be certain they are exactly equivalent to the original program.

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

6. The schedule Benchmark

6.1. Code and Test Suite, Robustness Improvements

`schedule.c` (SIR version 2.0) was down loaded from the Software-artifact Infrastructure Repository (Hutchins et al., 1994). 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, `printfs` were replaced by code to direct the scheduled process identifiers (`ints`) 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. (Checkpointing, perhaps inconjunction with the super mutant approach described in Section 8.3, might be another alternative.) 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 to ensure it 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. (`gzip`, Section 8, takes longer and therefore the limit was reduced to 200 + 10%.)

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.

The detection of infinite loops is in general undecidable. In practice it is often approached by applying an arbitrary timeout. (E.g. Weimer et al. (2009) abort their randomly created programs after five seconds and give them a zero score.) This has a number of problems.

Choosing an appropriate limit may not be straightforward. If the limit is too high, a single test case which causes a program to loop, may gobble excessive resources adversely affecting the whole testing process and possibly other uses of the computing system. If too short, a slow test case may be inadvertently marked as a failing a test case whereas it would have eventually passed the test.

If some test cases are very much longer than others (e.g. `gzip`, Section 8) a single time limit is not appropriate. Setting it long enough for the worst case means wasting too many resources should an indefinite loop form on any of the other cases. Choosing a timeout specific to each test cases runs the risks of a single timeout amplified n times.

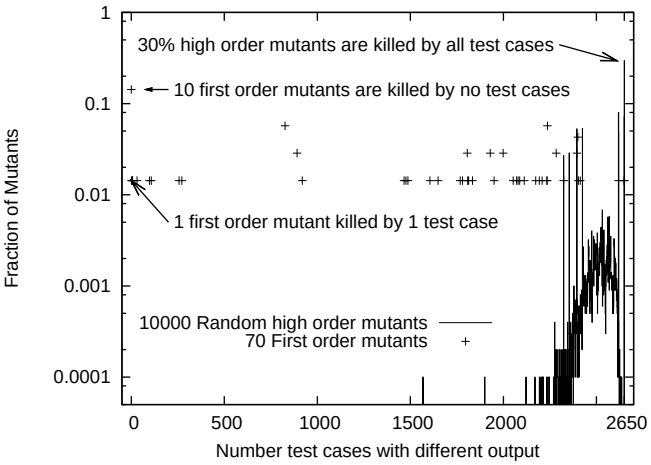

Figure 6: Fitness of all first order (+) and 10 000 random `schedule.c` mutants. Note log scale. More than 90% of high order mutants fail more than 90% of the test cases. On the other hand 10 of 70 first order mutants are not killed.

Elapsed time is notoriously variable. Even on a dedicated computer there are variations in run time. Hence a test suite which takes 4.99 seconds on average may sometimes pass 100% of its tests but appear to fail the last one every so often. This causes testing to be nondeterministic.

In our approach we don't seek to identify indefinite loops. We aim to tightly limit the resources used when testing mutants and report when a test causes a mutant to deflect from the original program behaviour. By instrumenting every potential loop (`for`, `while`, `goto`) during correct operation abberent behaviour of a mutant can be quickly spotted and the test efficiently aborted in a deterministic fashion.

6.2. 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 6. Ten of the 70 make no visible difference but one first order mutant fails a single test.

Figure 6 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.

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

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

tcas (SIR version 2.0) was down loaded from the Software-artifact Infrastructure Repository (Hutchins et al., 1994). It consists of 135 lines of C code (excluding comments and blank lines) with 40 branches (Zhong et al., 2008, Table 2) and 1608 test cases with up to 12 input parameters and one output. The supplied `main()` was modified 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.

`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. (They all compare `int` with `int`.) A BNF grammar (Figure 7) 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. (Randomly created individuals are 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 5 and 6) and confirmed by Figure 8, 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, see Table 4. Both worked and came to different solutions.

Table 4: Strongly Typed Grammar GP to find hard to kill tcas mutants

Primitives:	The function and terminal sets are defined by the BNF grammar (Figure 7). 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 2) 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 (Poli et al., 2008)
Parameters:	90% subtree crossover. 10% subtree mutation. Max tree depth 17 (no tree size limit)
Termination:	500 or 50 generations

7.2. 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 8. 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 (dots in Figure 8) shows there are 264 tcas tests which defeat 98.3% of random programs but 428 (Figure 9) which are passed by all 10 000 high order mutants.

Figure 9 plots the expected output from tcas for each test case. The three groups identified in the previous paragraph (see also Figure 8) are high lighted by sorting the tests by their effectiveness against random high order mutants. As with triangle and schedule, the most

```
<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>
```

Figure 7: 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 4. `XXX` is replaced by the individual mutant's identification before it is compiled.

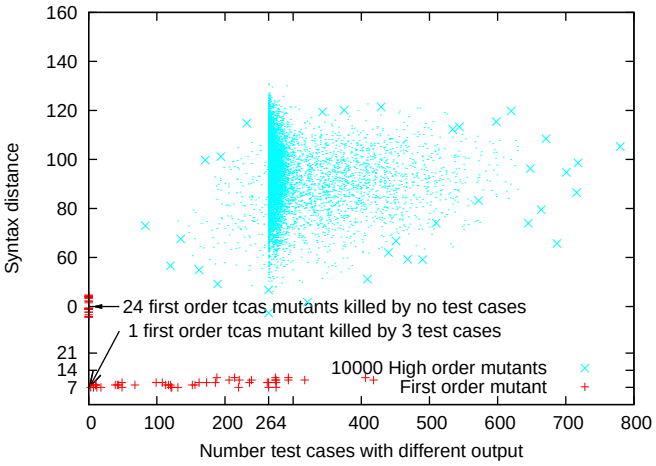

Figure 8: Fitness 70 first order tcas mutants (+) and 10 000 random high order tcas mutants (dots and ×).

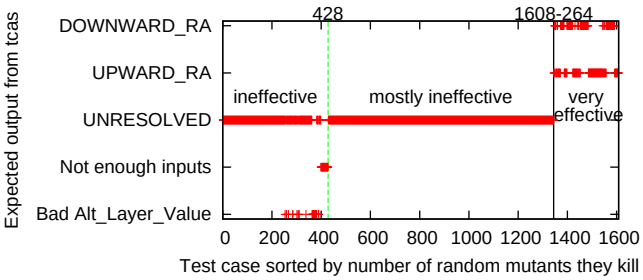

Figure 9: Expected value returned by tcas v. effectiveness of test against random mutants. The effectiveness of tcas tests falls into three main groups (highlighted by vertical lines) with little difference between members of the same group. On the right are 264 tests which check for threat of collision. They are failed by almost all high order mutants.

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 tests which check for `Bad Alt_Layer_Value` or `Not enough inputs` errors (lower two set of points in Figure 9). These are checked by outer level non-mutable code and so are unaffected by any mutations and consequently cannot detect any.

Most of the tcas test suite checks for UNRESOLVED. Of these tests, most are fairly poor against random mutations and find only about 1.67% of them, whereas the rest are totally ineffective and detect none.

7.3. Running the Pareto GP on tcas

Figure 10 shows NSGA-II progressively improved the initial random high order mutants. It both reduced

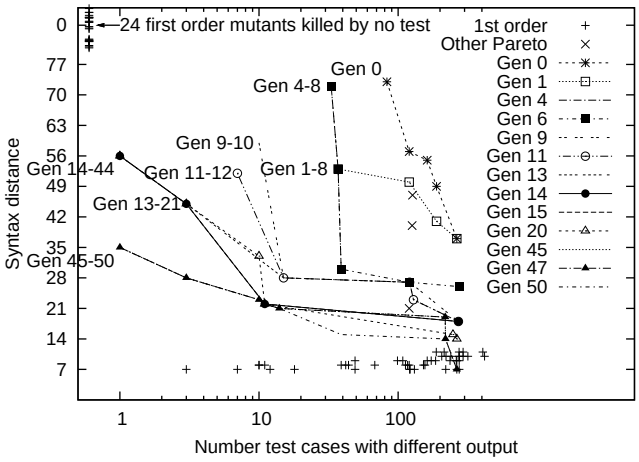

Figure 10: Evolution of tcas NSGA-II Pareto front with 10 000 mutants per generation.

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 (Appendix A.3) found several.

7.4. Hard to Detect tcas Mutants

At the left hand side of Figure 10 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.

Appendix A describes in detail three of the high order mutants discovered by genetic programming. In particular, it explains how the constituent first order tcas mutants knit together to form hard to detect high order mutants.

8. gzip

8.1. Robustness Improvements, Creating and Testing Mutants

The GNU compression program gzip and its test suite was down loaded from SIR (version 1.4).[4] SIR's version of gzip is 5680 lines of C spread over 104 functions. As well as being a real world program in daily use around the world, this makes it more than eight times the size of the earlier three benchmarks combined.

8.2. Mutation Testing Test Harness

8.2.1. Intercepting System Calls and I/O

`triangle.c` and tcas do not make use of files and even though the schedule test suite does contain several thousand files they are primarily a way of passing natural numbers. However files are fundamental to the normal operation of gzip. In fact gzip also makes extensive use of the file system, including recursively scanning directories and is in principle able to access any file. Fearful of losing our home directory, we placed gzip in a sandbox.[5] gzip was instrumented and the whole SIR test suite was run. Each test case's use of the file system was recorded. This included which files were opened and keeping the output generated. Including error messages (if any) and gzip's final status code. During mutation testing, each mutant is run on 211 test cases. (It proved impractical to run three, 38, 39 and 52, of the 214 test cases supplied by SIR. E.g. test 52 is not deterministic since results depend upon exact timings in an unpredictable fashion.) During mutation testing all file I/O and system calls are intercepted. When a mutant tries to open a file, the file name and mode of opening are compared with those used by the unmodified version of gzip. If they are different the mutant is deemed to have failed the test. Similarly its use of open files is compared. (This includes stdin, stdout and stderr, which are open by default.) When gzip tries to read from a file, its request is intercepted and data from an internal copy of the file is passed to it. When it tries to write, data is compared to a copy of the original output file. If any byte is different, the mutant has failed that test. In principle, system calls (such as changing file protection and reading shell environment variable POSIXLY_CORRECT) could also be checked however for simplicity they are simply discarded.

Notice as well as being secure, this can greatly speed up testing for several reasons. Often testing gzip is dominated by its access to the disk system. Typically each test cases uses two files, one for input and one for output. Nevertheless it is quite feasible for a modest personal computer to store the whole of SIR's gzip test suite in RAM. Using our approach files need only be read when the test harness starts. This overhead is

[4] We reported an error in the test suite to SIR which causes gzip to fail a test. SIR amended the test (SIR version 1.5) so that gzip passed it. However since the action of gzip is correct (it rejects the test's bad command line) and detecting incorrect usage is an important part of gzip's functionality we continue to use version 1.4.

[5] There are tales of mutation testing causing the loss of complete servers.

shared between all mutants being tested and the whole test suite. Since output files are not generated, there is 1) no disk I/O overhead for them 2) no disk space is required 3) there is no separate overhead of comparing output files generated by mutants with the correct output files 4) failing tests can be stopped immediately erroneous output is generated rather than having to run to completion. Despite additional overheads (described in Section 8.2.3) testing mutants within our framework is about twenty times faster than running the original SIR test script.

While not strictly necessary, the I/O interception code was enhanced to trap and treat as a test failing any mutant which repeatedly (i.e. more than 16 times) tries to read past the end of file. This is not an error as far as the C I/O system is concerned and will be trapped eventually by the limit on looping (described in Section 6.1). However, since the unmodified version of gzip stops reading files the first time it encounters EOF, detecting repeated attempts to read past the EOF gives an additional way of detecting mutants and has the benefit of reducing the time taken to test such mutants.

8.2.2. Memory Initialisation and Checks

To ensure mutants cannot interfere with each other before every test every variable is initialised. This is obviously required for global variables. However it turns out to be necessary for some local variables.

gzip is well coded and does not use uninitialised variables. But of course a mutant can fail to initialise a variable. Initially we thought this would simply be another source of error and the mutant would be be appropriately penalised. However in C the contents of uninitialised local variables is not defined and we found in practice it can vary between different runs, even when only one test is run for one mutant. (When using the gcc compiler, the option `-finit-local-zero` may be an effective workaround for this specific case of random runtime behaviour.) It appears the problem of nondeterminism in mutation testing has not been fully recognised. Of course it is well known that poor C code can misbehave in surprising ways.

One way to cope with nondeterministic mutants would be to run them multiple times and compare their results. But this raises several practical and philosophic problems. What is an acceptably low probability of a mutant failing before it can be assumed the error is unimportant? The more we demand that an error should occur at a low rate and the more confidence we demand of our estimate of the error rate, the larger will be the number of times we will have to test the mutant. This can easily lead to an explosion in run time. This appears to be an important area for further analysis. However, for the time being, we decided to take the option of forcing the mutants to be deterministic by forcing the initialisation of all variables even though this could be considered as treating mutants too leniently.

As described in Section 6.1 all calls to create or free heap memory were intercepted and directed to a predefined memory area. This is large enough for any test in the test suite. Again a mutant which tries to allocate more heap than allowed for is trapped and treated as failing that test case.

8.2.3. Array Index and Pointer Checking

Due to the size and complexity of gzip, it was not feasible to deal with every possible memory violation individually, instead we use the gcc -fbounds-checking patch.

The GNU compiler's distribution gcc 4.0.4 was obtained. (Version 4.0.4 was used because it is the most recent version for which there is a -fbounds-checking patch.) The patch was down loaded from SourceForge and applied to gcc.

-fbounds-checking is used in the normal way so that it signals an exception when it detects a mutant misusing data structures. The signal is intercepted by the test harness framework. The current mutant is judged to have failed the current test and the next test is started. Occasionally there is an unfortunate interaction between -fbounds-checking and the compiler optimisation -O3 switch which causes an internal error in the -fbounds-checking code. A mutant which triggers this has undoubtably failed. Unfortunately there is no mechanism to enable -fbounds-checking to reset its own data structures and so it is not possible for the process to continue testing. Therefore mutants are compiled without optimisation. This increases run time by about 40%.

For efficiency, in the test harness code which inserts mutants at run time, the run time bounds checks were disabled using BOUNDS_CHECKING_OFF_IN_EXPR. See also Section 8.3.

Two obscure cases where mutant misbehaviour was not spotted by -fbounds-checking were dealt with automatically as special cases.

8.2.4. System Traps

In addition to errors reported by -fbounds-checking, gzip mutants may encounter other errors which cause it to raise exceptions. Also it may detect errors which it handles gracefully by trying to shut gzip down. These are all intercepted by the test harness (via SIGABRT and SIGSEGV signal handlers). If any of these happen

the mutant is said to have failed the current test and the next started.

8.3. Speeding up Mutation Testing

gzip comprises 4795 lines of C code. This is eight times bigger than the three earlier benchmarks put together. However we use only 211 test cases (i.e. less than a 10^{th} the number used to test `schedule.c`). This much lower density of testing changes the balance of computational testing effort. Where in the three previous cases the time taken to create and compile the mutants was negligible it now took approximately half the CPU time. Therefore we adopted the mutant schema generation approach proposed by Untch et al. (1993).

Instead of compiling each C mutant, a super mutant was created. The super mutant contains every mutation plus code to selectively enable them. The NSGA II multi-objective Pareto approach (Sections 2 and 3 and Figure 2) is retained. However instead of manipulating C source code each generation, the population of high order mutants becomes a population of simple instructions to the super mutant. Each high order mutant now tells the super mutant which of the six possible comparisons to use at each of the possible mutation sites. Manipulating these fixed length individuals is straight forward and the super mutant need only be compiled once.

8.4. gzip Mutant Sites

gzip makes heavy use of include .h files and uses conditional compilation. To avoid mutants trying to do things like creating a SunOS version for our Linux computer, we fixed conditional compilation to the default Linux configuration. For simplicity duplicate and nested include files were removed so that all include statements appear only at the start of SIR's allfile.c source file.

gzip also makes heavy use of C macros. Several of these in allfile.c and gzip.h contain conditionals. Our mutation operator acts on macros and so a single change in a #define statement can be effective in many places in gzip. This is similar to what happens when a mutation is made to a function. The change could affect the program each time the function is called.

We use the gcc -E switch to process include files, perform the conditional compilation and remove comments. This reduces the source file and gzip include files from 8900 to 4795 lines containing 496 mutation sites. Fifteen occur between 2 and 47 times due to macro expansion.

In contrast to schedule and tcas where there are about ten times as many tests as lines of code, SIR's test suite provides on average only one test per 35 lines of code. Figure 11 shows the resulting very uneven distribution of number of times comparisons (i.e. potential mutation sites) are exercised. The maximum is more than 42 million times but the median is only 24 and there are 132 first order mutants which are not used by any of the tests.

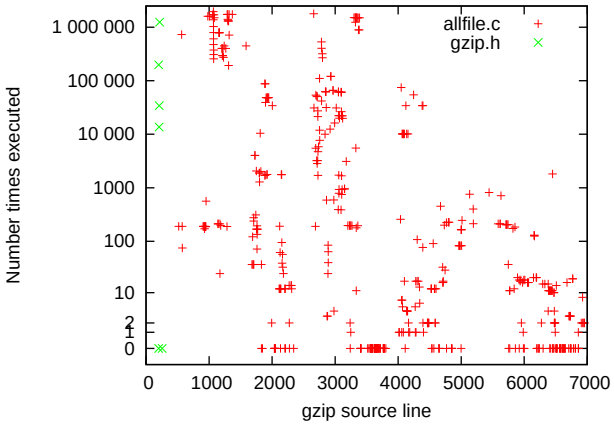

Figure 11: Number of times SIR gzip test suite uses each first order mutation site. Plotted by code location (horizontal axis).

8.5. gzip First Order Mutants

There are 496 comparisons each of which can be mutated to any of the six C comparison operations leading to a total of $5 \times 496 = 2480$ first order mutants. 660 of these are in code which is never executed by the 211 SIR test cases. Naturally these pass all the tests. Of the 1820 mutants which are exercised at least once 544 pass all the tests, 251 pass all but one, 129 pass all but two and so on. All the first order mutants are plotted in Figure 12. They also correspond to lines parallel to the left edge of Figure 13, with most of these mutant lying at the frontmost corner. As Figure 13 makes clear, there is a strong relationship between the number of tests which execute a potential mutation site and how hard it is to kill the corresponding first order mutations. Obviously a mutant can only fail a test case which exercises it, so every mutant lies on the same side of the diagonal. Figure 13 shows that first order mutants tend to lie on vertical and horizontal lines, which correspond to particular groups of test cases. Further, where these lines intersect on the diagonal is particularly popular. Points on the diagonal indicate that such first order gzip mutants fail every test that actually causes them to be exercised.

In general, how many times (provided its at least one) a mutation site is actually used during a test does not appear to be particularly important to whether a first order

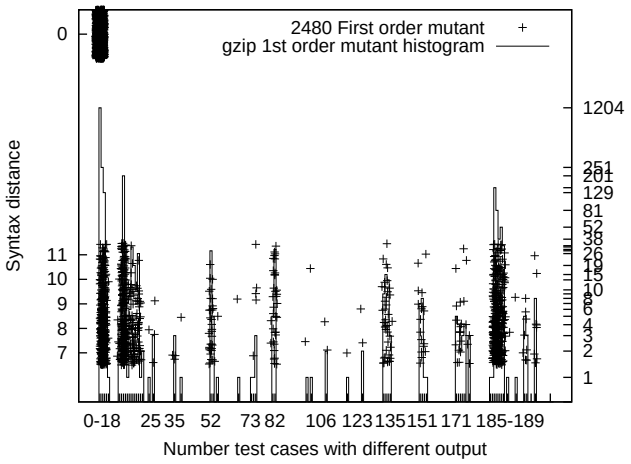

Figure 12: Difficulty of killing gzip first order mutants (+). (Noise added to spread data). Equivalent mutants are given poor fitness and therefore are plotted at the top left by "0". The solid line is the number of first order mutants killed by various numbers of the 211 SIR tests. (note non-linear vertical scale). 660 off the 1204 equivalent mutants, are equivalent (i.e. not killed) simply because they not run by any of the 211 SIR tests. 251 are killed by 1 test case, 129 by 2 test cases, and so on.

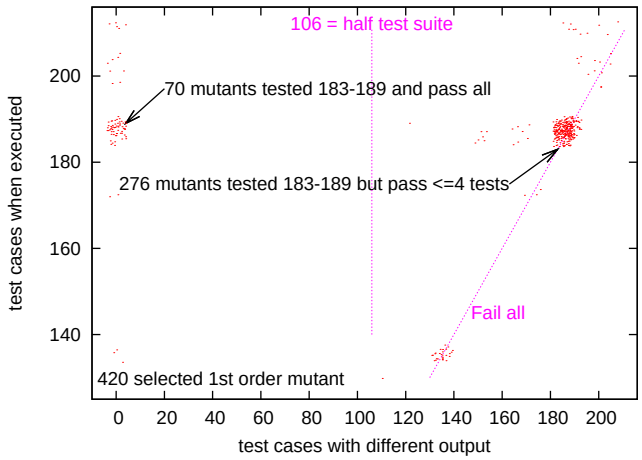

Figure 14: 420 first order gzip mutants located in code which is exercised by more than half the test cases and which are easy to kill. The 78 potential equivalent mutants are above 0. Noise used to spread data. (Dotted lines as those on Figure 13. See also Section 8.6.)

mutant is detected or not. This suggests, at least for gzip's test suite, repeated execution does not gain the tester much new information.

8.6. *Subtle Interactions Between gzip Mutants in Heavily Tested Code*

Since mutation testing is typically used after other less expensive forms of testing we decided to restrict ourselves to those parts of the gzip code which are more intensively tested by the SIR test suite. To be precise, those parts of gzip which are exercised at least once by at least half the test suite.

The last experiments use gzip as a test bed to investigate interactions between single mutations. That is, to investigate higher order mutations in well tested parts of gzip. In particular we look for interactions that show simple easy to detect faults can interact in subtle ways to yield hard to kill high order mutants.

We define an easy to detect first order mutant as one that fails more than half the test SIR suite (i.e. more than 106 test cases). These occupy 84 mutable locations in gzip. For completeness, we include all five possible mutations at each site, giving $5 \times 84 = 420$. These are shown in Figure 14. (These 420 first order mutations include 78 potential equivalent mutants, the remaining 342 fail more than 106 test cases.)

8.6.1. *Second Order Interactions of "Easy" Mutants*

We investigated all possible second order mutants formed by combining pairs of the 420 "easy" to kill first order mutations (including the potential equivalent mutants). Figure 15 shows the two components of our multiobjective fitness measure for all 87 150 of these second order mutants in well tested code.

There are four cases where a second order mutant is a lot more difficult to detect than its components (i.e. the difference is more than two test cases). In three of these two previously undetected mutants combine to create a second order mutant which fails almost all the tests. (Two fail 185 test cases and the other fails 188. All three are shown with stars in Figure 15.) The fourth example is arrowed in Figure 15. Let us first consider the two second order mutants which both fail 185 test cases.

Two Pairs of Equivalent Mutants Fail 185 Tests.
These two 2^{nd} order mutants are both mutations of the same two sites in the outer loop in `longest_match()`. `longest_match()` is responsible for compressing files. The two comparisons are in heavily optimised C code which is conditionally compiled without `-DUNALIGNED_OK`.

The 2^{nd} order mutants are identical except in one case the `!=` at the start of the outer loop is replaced by `<` and in the other `!=` is replaced by `>`.

The other component occurs 8 lines later. If the only change is `if (len > best_len)` is replaced by `if (len >= best_len)` then the first order mutant is equivalent. By itself, the change means that rather than using the first occurrence of the longest match the last

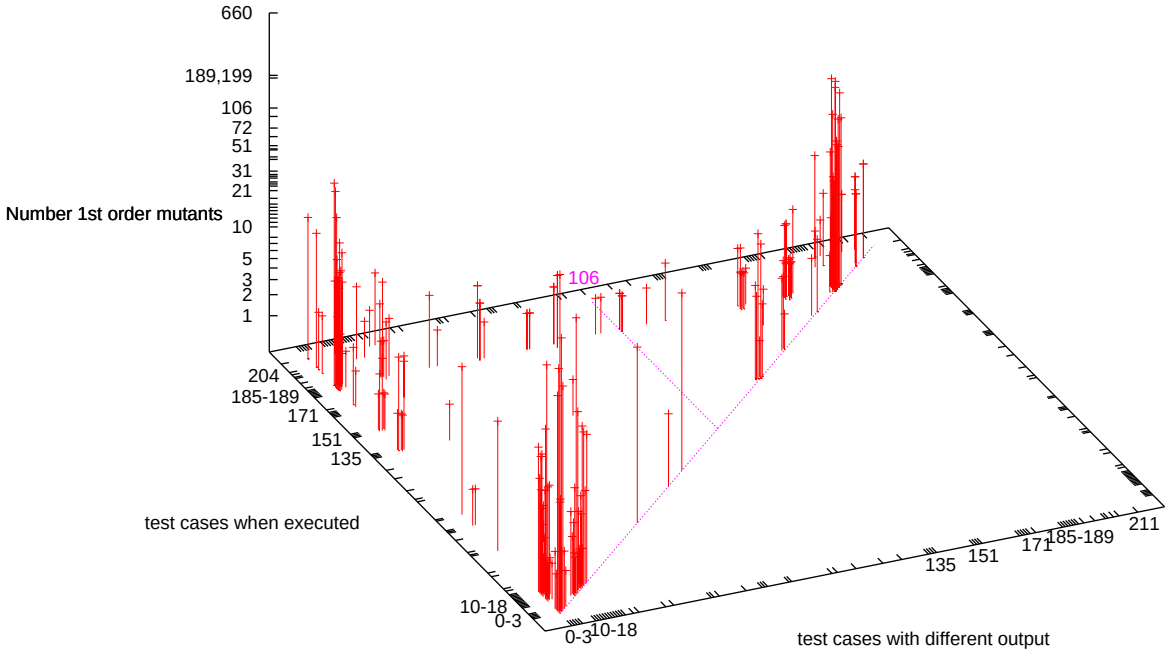

Figure 13: Difficulty of killing gzip first order mutants versus test cases. Height is number of such mutants (note non-linear vertical scale). Mutants can only be killed when they are run, hence all points lie in the upper triangle. (See Section 8.5.)

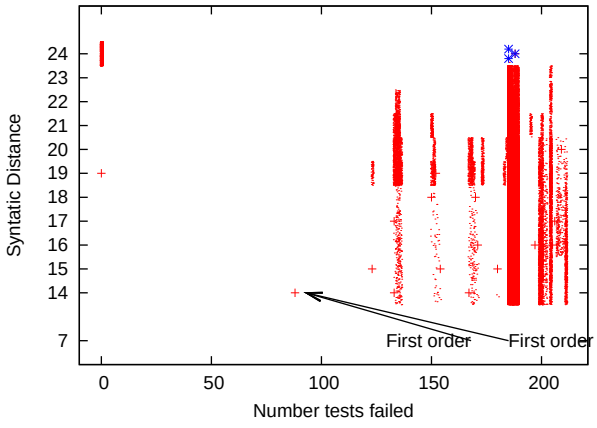

Figure 15: Distribution of fitness of all $\frac{84\times83}{2}5^2$=87 150 well tested second order gzip comparison mutants. Unit noise used to spread dots. Highlighted higher order mutant is harder to kill than its components. (*) Three non equivalent mutants 2^{nd} order mutants formed from two 1^{st} order potential equivalent mutants.

occurrence of it is used. Although the comments suggest this is an error, since the two strings are identical, it does not change the output of the SIR test cases.

Returning to the top of the outer loop, the `!=` is the first of four conditions which are designed to skip quickly to the next iteration of the loop if there is nothing to be done with this one. Most of the effects of replacing `!=` by `<` or `>` are masked by the following three conditions. So the mutation changes flow of control in only 7% of cases. However all of these are nullified by the `if (len > best_len)` statement eight lines later. Meaning both `<` and `>` first order mutants are equivalent. *However*, if it is also mutated most of the 7% of changes have an impact and cause incorrect matching and so gzip's output is wrong.

A Pair of Equivalent Mutants Fails 188 Tests.
The first mutation site is in the `while` condition of service routine `bi_reverse()`. The first order mutation is to replace `while (--len > 0)` by `while (--len != 0)`. This is clearly equivalent as long as `len` is initially positive (as it is intended to be).

The second mutation site is in the second `for` loop in `void` procedure `gen_codes()`. The mutation site `if (len == 0) continue` is intended to skip around empty tree nodes (when `fc.code == 0`). A first order mutation which replaces it with `if (len < 0) continue` will cause

`tree[n].fc.code` to be updated even when `len` is zero. An unmutated version of `bi_reverse()` will return and may change `tree[n].fc.code`. However `fc.code` is only used elsewhere by `send_bits()` and `send_bits()` effectively does nothing if `dl.len` is zero. Although `next_code[0]` is incremented `next_code[0]` is only used because of this mutation and so again has no external effect.

Thus we have two equivalent first order mutants. However if both errors are injected, the erroneous call of `bi_reverse()` with a length parameter of zero is no longer benign. If `bi_reverse()` returned, it would have no effect on the operation of `gen_codes()` but it does not. Instead it falls into an infinite loop, which is eventually trapped by our mutation testing framework.

2nd Order Mutant Harder to Kill.

The previous three second order mutations show cases where simple hidden errors are revealed by injecting a second error. The final example (arrowed in Figure 15) shows a case where two very easy to detect errors tend to conceal each other, making the combined second order mutation more difficult to detect than either component by itself. Both mutation sites are in the `for` loop of `scan_tree()`.

In the first mutation `if (count < min_count)` becomes `if (count <= min_count)`. This causes 89% of tests where the condition is executed to fail. The second site is 13 lines later.

In the second mutation `if (curlen == nextlen)` is replaced by `if (curlen != nextlen)`. This causes `max_count = 6, min_count = 3` to be replaced by `max_count = 7, min_count = 4` and vice-versa. Which in turn causes more than 99% of tests where it is used to fail.

The first three lines within the `scan_tree()` `for` loop search for identical values of `tree.Len` and so in almost all cases the second mutation site is only executed when `curlen` is not equal to `nextlen`. So without either mutation the second comparison typically sets `min_count` to four. Therefore in the following iterations of the `for` loop the first mutation site, `if (count < min_count)`, is equivalent to `if (count < 4)`. Now if both mutations are applied, the second one has the effect of setting `min_count` to three. Thus the first one now has the effect of `if (count <= 3)`. Which is clearly equivalent to the case when neither fault is injected. Given a fortuitous sequence of `tree.Len` the two mutations can mask each other. This means the second order mutation passes slightly more than half of the SIR test suite, whereas its components cause many tests to fail.

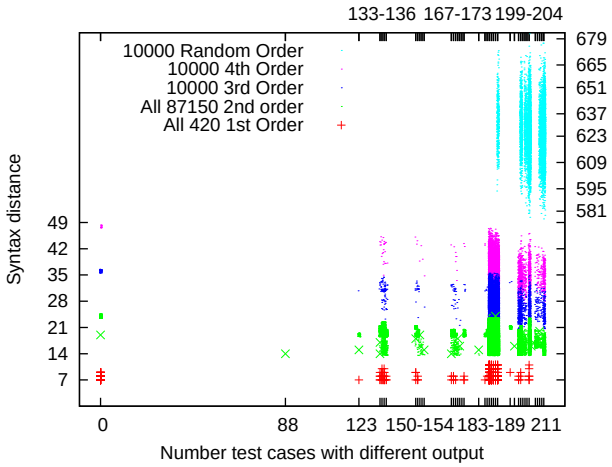

Figure 16: Fitness distributions produced by Monte Carlo sampling of high order mutants produced by combining easy to kill gzip comparison mutants. Unit noise used to spread dots. (Syntactic distance of 70^{th} order mutants rescaled, see right hand scale.)

Table 5: Equivalent first and higher order gzip mutants. Last row (Equiv) gives percentage of well tested gzip mutants that pass all test cases. (None fail just one test.)

Order	1^{st}	2^{nd}	3^{rd}	4^{th}	70^{th}
No.	420	87 150	$1.2\ 10^6$	$1.2\ 10^9$	$2.6\ 10^{64}$
Equiv	19%	3%	0.5%	0.1%	0

8.6.2. *Higher Order Interactions of "Easy" Mutants*

Figure 16 shows the distribution of fitnesses of higher order mutants created by combinations of the well tested first order gzip mutants selected in Section 8.6. We see similar behaviour to the other example programs. The fraction of equivalent mutants falls rapidly with mutation order (see Table 5) and the test suite is highly effective against random high order mutants.

90% of the SIR test cases kill all 10 000 70^{th} order mutants. 4% kill more than 90% and the remaining 7 kill at least 50%. These seven test gzip's ability to produce help text and similar messages. The routines responsible for these messages are called directly from the `main()` routine or each other and do not contain any comparisons and so will never be mutated themselves. They could only be disrupted by the three potential mutation locations in `main()` before them. Hence their comparative ineffectiveness against random injection of multiple errors. However they are cheap to use and effective against about half of all high order mutants.

8.6.3. Searching for Second Order Interactions of "Easy" Mutants

In terms of a search problem, the second order mutant (described in Section 8.6.1) is very hard to find. If we look at the complete search neighbourhood formed by third order mutants which share one or more components with it, they either all fail at least 167 tests or they have identical fitness. This is important because it means that there is no sign of a gradient in the fitness landscape leading to hard to kill third order mutants. Unfortunately even with a population as large as 1000, GA runs seldom found non-equivalent high order combinations of easy to kill first order gzip mutants which failed less than 188 test cases (best 133).

9. Threats to Validity

Four experiments were conducted on higher order mutation testing using multi-objective search. Although these experiments were carefully designed to be as fair as possible, there are a number of threats to the validity. These include: the choice of mutation operators, the choice of test sets and the mutant killing condition.

The choice of mutation operators is the first threat. To reduce the enormous number of mutants generated, a common mutation cost reduction technique, Selective Mutation (Offutt et al. (1996a)) was used. Our experiments were based on the suggestion that complex faults are likely to involve an element of anomalous control flow. The relational replacement operator was therefore chosen to generate mutants because it directly alters the control flow in the mutants and because it alters their data flow indirectly. In the future this threat can be reduced by applying more operators from Offutt et al. (1996a)'s five selective mutation operators.

The quality of the test sets is another potential threat. Since the semantic distance is computed based on the number of failed test data, low quality test sets might affect the result. To avoid this threat, all test sets in our experiments (except gzip) are branch coverage adequate. While gzip's tests were supplied by SIR (Hutchins et al., 1994). However with a different branch coverage adequate test set, the distribution of results might be affected a little. This threat might be overcome by combining higher order mutation testing with the co-evolutionary mutation testing approach of Adamopoulos et al. (2004). This might allow us to co-evolve test sets able to kill the co-evolving mutants dynamically. (We shall discuss co-evolution more fully shortly).

The last threat is the mutation killing condition. To successfully apply mutation testing to these real world programs, a number of new weak mutation techniques (see Table 1) were introduced. These approaches can detect changes to either the internal or the external behaviour of mutants. However, as with other weak mutant approaches (Offutt and Lee (1994)), a few mutants killed by by our approach might have survived strong mutation testing (where only external changes in behaviour kill a mutant). As with the first threat, this can only increase (or leave unchanged) the number of "interesting" hard to kill mutants found. So our results can be considered to be a lower bound on the number hard to kill higher order mutants to be found.

Mutation testing is widely regarded as expensive. This is in part true, however we hope we have explained how search based optimisation methods make it practical. There are many more high order mutants than first order ones. This has lead to the misimpression that (as ordinary mutation testing is expensive) higher order mutation testing is impossible. The purpose of many modern optimisation techniques is to avoid enumeration of all possibilities and instead to concentrate search on the most fruitful areas. We have considered high order mutants of four programs of increasing complexity and hopefully convinced you that search heuristics can make higher order mutation testing feasible. Nevertheless there are "No Free Lunch" theorems which suggest that we cannot always guarantee modern (or indeed any) search heuristics will always be successful. However we have been able to carry out high order mutation testing on real programs code in C on an everyday office personal computer.

One idea for the future is to integrate mutation testing with test case generation. There have already been thoughts in this direction (DeMillo and Offutt, 1991; Ayari et al., 2007). However a co-evolutionary framework, such as that proposed in Bongard and Lipson (2005) could put this on an automated basis. This might take the form of a population of test cases and a population of high order mutants of the software under test. The two populations compete against each other. The genetic fitness of a high order mutants being given by how well it conceals itself from a test suite drawn from the population of tests. Whereas the fitness of a test is given by how many mutants it kills. The fitter members of each populations are retained and the less successful ones discarded. The selected individuals (both test cases and higher order mutants) are subject to random changes and recombination to create new variants. The fitnesses of the new variants in the two new populations are measured in the same way that their parents were. The idea is that by using computer resources

the two populations will continuously improve against each other, leading to very high quality test suites and so very high quality software. In practice such virtuous circles of co-evolutionary improvement have proved tricky, which is why approaches such as Bongard and Lipson (2005) are attractive.

In Section 8.2.2 we talked about nondeterminism caused by uninitialised data values, etc. We have ensured that data values are initialised, either to the value given in the original (unmutated) source or to zero. An interesting test possibility which might be considered in future would be to introduce setting variables to less benign values initially. For example, in C, int values could be set to 0, -1, INT_MAX, -INT_MAX, -2147483648, as well as randomly chosen values. floats could be set to 0, -1, FLT_MAX, -FLT_MAX, inf, -inf, -0, NaN, as well as randomly chosen values. Obviously in well written code which does not rely on initialised variables, this would not be as useful as an isolated first order mutation however it may be an additional tool particularly with higher order mutations.

10. Conclusions

We have reformulated 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 merely a special case of a higher order mutant (for which the order is simply one). We have shown how such a multi-objective approach can be used to investigate the relationship between the syntax of a mutant and its impact on the host program's semantics. In real ancient heavily optimised C code, not only have we found combinations of simple faults (first order mutants) that are harder to kill than the first order mutants but also the reverse; combinations of simple faults which to some extent actually mask each other.

We have implemented this approach using a combination of genetic programming (GP), Genetic Algorithms and Monte Carlo sampling. Through several efficiency improvements, reported above, the results were obtained using a standard office personal computer. On the 2.66GHz 2Gbyte Linux PC, the smaller GP tcas runs take a few minutes but it took more than 5 days to collect the data behind Figure 15. The results demonstrate that 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*.

Section 8.2.2 reported the problem of nondeterminism in mutation testing for the first time. In Section 8.2.2 the problems associated with a mutant behaving differently between different runs were discussed in the context of the mutant causing a local variable not to be initialised and hence to take essentially random values on different runs. This in turn could cause the mutant to pass tests on some runs and fail the same tests (i.e. be killed) on other runs. There are other possible reasons why (mutated) code may behave differently on different runs. For example, interactions with the operating system, the file system and the environment. As we said on page 2427, trying to detect random faults, even when they are probable, by repeatedly running tests, quickly leads to a large increase in testing effort.

Although very high order mutants are easy to detect they give information about the test suite. If a test kills almost all high order mutants this tends to suggest the test covers normal operation of the program. Conversely those tests which pass some high order mutants may be covering more peripheral areas of the program under test, such as: exceptional cases or even erroneous inputs, command line verification or generation of help messages.

While the number of possible mutations is bound to rise rapidly with their order, triangle.c shows that (at least up to fourth order) the number that are hard to kill also rises as more first order mutations are combined. However as a fraction of the total they fall. This suggests (all other things being equal) search for interesting high order mutants is harder as higher orders are considered.

Although DeMillo et al. (1978, p 39) suggested "well under 1 percent" of mutants are equivalent, the problem of potential equivalent mutants is now recognised. Indeed we find 16%, 14%, 34% and 49% (for triangle, schedule, tcas and gzip, respectively) of first order mutants are potential equivalent mutants. Admittedly the fraction is swollen for gzip by the large number of mutants not even executed by the SIR test suite. Even if we include these, the proportion is still 30%. Detailed analysis of these hundreds of first order mutants to show which are truly equivalent and which are equivalent only as far as we are able to test the code, is not feasible. This confirms the expectation that for practical purposes the fraction of equivalent mutants is considerably higher than originally suggested.

We have provided additional support for the coupling hypothesis. Remember it says if a test suite is good

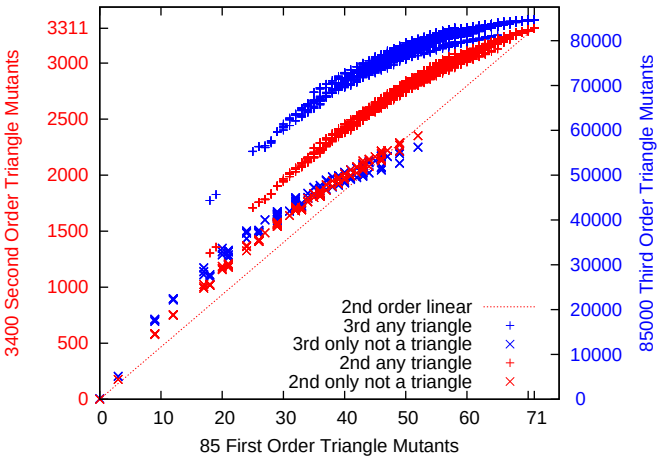

Figure 17: Demonstration of the coupling hypothesis. 16 383 test suites were generated by selectively including parts of the triangle test suite given in Table 2. Test suites that kill more first order mutants tend to kill more second order mutants (red) and third order mutants (blue). There is a near linear relationship, but most test suites are proportionately more effective against higher order triangle mutants. However test suites where all tests have an expected output of "not a triangle" (×) are closer to the proportionate response. The third row of Table 3 gives the number of potential equivalent mutants.

enough to detect all first order mutants, it will detect "a large percentage" of higher order mutants. In all four cases we saw the test suite was better at killing higher order mutants than first order mutants. Indeed their effectiveness increased with increasing complexity of the mutants. (See also Figure 17). This is even true of gzip, where we know SIR's test suite does not kill all first order mutants. Nevertheless there remains a significant number (albeit a small fraction) of interesting higher order mutants which show non-linear interactions between their first order component. Purushothaman and Perry (2005)'s empirical evidence that most bugs require multiple changes to correct, suggests that in many cases the "competent programmer hypothesis" (DeMillo et al., 1978) only holds for a small fraction of errors. Perhaps the simpler ones. Perhaps errors cause by poor specification or initially omitted requirements or incomplete data structures designs, tend to require multiple changes to the source code to rectify.

As would be expected, mutation testing highlights problems with testing. For example, in gzip it shows large amounts of the code are not being covered at all but others are exercised repeatedly. While these are in core areas of the code, it is doubtful that testing the same components millions of times gives more confidence in the program and perhaps the effort could be better be used elsewhere. (The idea of using mutation testing in combination with test generation tools was discussed in the previous section.) In the case of tcas, several areas of convoluted logic were highlighted by higher order mutation as being potentially worthy of further validation effort.

The SIR test suite for gzip is much smaller than those available for the other programs studied. Nevertheless, several fault pairs were found, which, although individually easy to detect, combine to partially mask each other, thereby producing a high order mutant which is harder to kill than either of its components. Search also revealed cases where two first order equivalent mutants interact to produce a serious fault. These are examples where faults, which *cannot be detected* by testing, are benign until the two of them are brought together.

Acknowledgments

We would like to thank the Software-artifact Infrastructure Repository `sir.unl.edu`, the authors of NSGA `http://www.iitk.ac.in/kangal/` and William Bader. Finally our thanks to Dan Hoffman for reporting an error in our original `triangle.c` code.

Appendix A. Tough High Order tcas Mutants

Appendix A.1. Seventh Order tough tcas Mutant

The mutant changes line 87 in function `Non_Crossing_Biased_Climb()` (twice), line 101 in `Non_Crossing_Biased_Descend()` (twice), line 112 in `Own_Below_Threat()`, line 117 in `Own_Above_Threat()` and line 127 in `alt_sep_test()`.

line 127, input 10 v. `NO_INTENT` $= \quad \rightsquigarrow \quad \leq$.
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 relatively easy to detect. However its effect is totally masked by the rest of tcas. This means that as an isolated first order mutant this change to line 127 is not detected by the test suite.

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.

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.

Appendix A.2. Fifth Order tough tcas Mutant

Referring back to the left hand side of Figure 10 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 (described in Appendix A.1) 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. That is, 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.

Appendix A.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 line 101 in `Non_Crossing_Biased_Descend()` (once), line 112 in `Own_Below_Threat()` and line 117 in `Own_Above_Threat()` in the *same* way as described in Appendix A.1.

Although this mutant only contains three of the seven mutations described in Appendix A.1 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.

References

Adamopoulos, K., Harman, M., Hierons, R. M., 26-30 June 2004. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: Deb, K., et al. (Eds.), Genetic and Evolutionary Computation – GECCO-2004, Part II. LNCS 3103. Springer, Seattle, WA, USA, pp. 1338–1349. URL `http://link.springer.de/link/service/series/0558/bibs/3103/31031338.htm`

Ayari, K., Bouktif, S., Antoniol, G., 7-11 July 2007. Automatic mutation test input data generation via ant colony. In: Thierens, D., et al. (Eds.), GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation. Vol. 1. ACM Press, London, pp. 1074–1081. URL `http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1074.pdf`

Bongard, J. C., Lipson, H., 2005. Nonlinear system identification using coevolution of models and tests. IEEE Transaction on Evolutionary Computation 9 (4), 361–384.

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., Apr 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6 (2), 182–197.

DeMillo, R. A., Lipton, R. J., Sayward, F. G., April 1978. Hints on test data selection: Help for the practicing programmer. Computer 11 (4), 34–41.

DeMillo, R. A., Offutt, A. J., 1991. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17 (9), 900–910.

Emer, M. C. F. P., Vergilio, S. R., Jun. 2003. Selection and evaluation of test data based on genetic programming. Software Quality Journal 11 (2), 167–186.

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

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

Harman, M., Jia, Y., Langdon, W. B., 6 Apr. 2010. A manifesto for higher order mutation testing. In: du Bousquet, L., Bradbury, J., Fraser, G. (Eds.), Mutation 2010. IEEE Computer Society, Paris, pp. 80–89, keynote.
URL http://www.dcs.kcl.ac.uk/pg/jiayue/publications/papers/HarmanJL10.pdf

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

Jia, Y., Harman, M., September 2009a. An Analysis and Survey of the Development of Mutation Testing. Technical Report TR-09-06, CREST Centre, King's College London, London, UK.

Jia, Y., Harman, M., 2009b. Higher order mutation testing. Journal of Information and Software Technology 51 (10), 1379–1393.

Kim, S., Clark, J. A., Mcdermid, J. A., 2001. Investigating the effectiveness of object-oriented testing strategies using the mutation method. In: Mutation Testing for the New Century. Vol. 11. pp. 207–225.

Lakhotia, K., Harman, M., McMinn, P., 7-11 July 2007. A multi-objective approach to search-based test data generation. In: Thierens, et al. (Eds.), GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation. Vol. 1. ACM Press, London, pp. 1098–1105.
URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1098.pdf

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

Langdon, W. B., Harman, M., Jia, Y., 4-6 Sep. 2009. Multi objective mutation testing with genetic programming. In: Bottaci, L., Kapfhammer, G., Walkinshaw, N. (Eds.), TAIC-PART. IEEE, Windsor, UK, pp. 21–29.
URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2009_TAICPART.pdf

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

Langdon, W. B., Harrison, A. P., 19 Mar. 2009. Evolving DNA motifs to predict GeneChip probe performance. Algorithms in Molecular Biology 4 (6).
URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_amb.pdf

Ma, Y.-S., Offutt, J., Kwon, Y. R., June 2005. MuJava: an automated class mutation system. Software Testing, Verification & Reliability 15 (2), 97–133.

Offutt, A., Lee, S., May 1994. An empirical evaluation of weak mutation. IEEE Transactions on Software Engineering 20 (5), 337–344.

Offutt, A. J., January 1992. Investigations of the software testing coupling effect. ACM Transactions on Software Engineering and Methodology 1 (1), 5–20.

Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., Zapf, C., April 1996a. An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology 5 (2), 99–118.

Offutt, A. J., Pan, J., Tewary, K., Zhang, T., 1996b. An experimental evaluation of data flow and mutation testing. Software - Practice and Experience 26 (2), 165–176.

Offutt, A. J., Untch, R. H., 6-7 October 2001. Mutation 2000: Uniting the Orthogonal. In: Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00). San Jose, California, pp. 34–44, published in book form, as *Mutation Testing for the New Century*.

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

Purushothaman, R., Perry, D. E., 2005. Toward understanding the rhetoric of small source code changes. IEEE Transactions on Software Engineering 31, 511–526.

Sen, K., Marinov, D., Agha, G., 2005. CUTE: a concolic unit testing engine for C. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. ACM, New York, NY, USA, pp. 263–272.

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

Untch, R. H., Offutt, A. J., Harrold, M. J., 1993. Mutation analysis using mutant schemata. In: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis. Cambridge, Massachusetts, pp. 139–148.

Weimer, W., Nguyen, T., Le Goues, C., Forrest, S., May 16-24 2009. Automatically finding patches using genetic programming. In: Fickas, S. (Ed.), International Conference on Software Engineering (ICSE) 2009. Vancouver, pp. 364–374.
URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/Weimer_2009_ICES.html

Yoo, S., Harman, M., 2007. Pareto efficient multi-objective test case selection. In: ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis. ACM, New York, NY, USA, pp. 140–150.

Zhong, H., Zhang, L., Mei, H., 2008. An experimental study of four typical test suite reduction techniques. Information and Software Technology 50 (6), 534–546.
URL http://www.sciencedirect.com/science/article/B6V0B-4P2J035-1/2/d5014b52df76dbb547b94b0cf387eab5