Multi Objective Higher Order Mutation Testing with GP

William B. Langdon, Mark Harman, Yue Jia

Department of Computer Science, CREST centre, King's College, London, WC2R 2LS, UK

Wi11iam.Langdon@kcl.ac.uk,Mark.Harman@kcl.ac.uk,Yue.Jia@kcl.ac.uk

ABSTRACT

Mutation testing is a powerful software engineering technique for fault finding. It works by injecting known faults (mutations) into software and seeing if the test suite finds them. It remains very expensive and the few valuable traditional mutants that resemble real faults are mixed in with many others that denote unrealistic faults. The expense and lack of realism inhibit industrial uptake of mutation testing. Genetic programming searches the space of complex faults to find realistic higher order mutants. Despite the much larger search space, we have found mutants composed of multiple changes to the C source code that challenge the tester and which cannot be represented in the first order space.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Verification

Usually each mutant is created by the insertion of a single simple fault. The faults are traditionally created by a small syntactic change, such as the replacement of one arithmetic or relational operator with another. If a test input can distinguish between a mutant and the original program, by causing each to produce a different output, then the test input is said to 'kill' the mutant. The percentage of mutants killed gives a measure of the effectiveness of a test suite. Mutation testing has also been used to assess other test coverage criteria, such as branch coverage and statement coverage and for test case generation.

Unfortunately, traditional mutation testing is very costly since there are many possible simple mutations which are uninteresting since they are readily killed by the simplest of test cases. This leads to wasted effort spent killing trivial mutants. The vast majority of hard-to-fix expensive faults are complex (i.e. higher order mutants).

We suggest mutation testing should be seeking faults which cause the mutated program to behave similarly to the original. I.e. *semantic* mutants rather than *syntactic* mutants. We explore the relationship between these two notions of similarity; syntactic and semantic using each as an objective in a multi-objective Pareto GP system. The syntactic difference is based on the changes made by the GP to the C source code and the number of them. The semantic difference is defined as the number of tests which kill the mutant. The fewer, the harder it would be to detect (and eventually fix) the fault. Equivalent mutants, where the changes to the code make no (detectable) difference, are usually uninteresting and are excluded by giving them a very poor fitness.

GECCO'09, July 8–12, 2009, Montreal, Canada
ACM 978-1-60558-325-9/09/07.

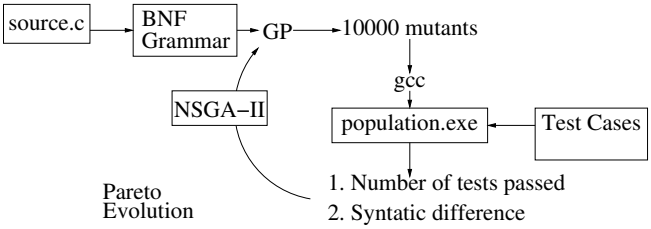

Figure 1: High order Multi-objective mutation testing. The BNF grammar tells GP where it can insert mutations into `source.c`. Initially GP creates a population of random mutations, which are compiled and run against the test suite. NSGA-II selects the mutants to retain and instructs the GP which mutants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

We study the set of mutation operators that replace one relational operator with another. These are interesting because they denote the ways in which one might alter the flow of control within a program. These may resemble subtle complex faults that programmers are likely to commit which might resemble such slightly anomalous control flow.

The target source code is automatically analysed to create a BNF grammar which describes all its possible mutants. Unlike most BNFs, the grammar consists mostly of terminals which regenerate the fixed portions of the source code. However all comparisons are replaced by a BNF rule with more than one production. By choosing between these alternative expansions, the GP generates a mutated program.

Monte Carlo sampling of higher order mutants confirms the well–known mutation testing coupling hypothesis. I.e. adding changes to a faulty program tends to make it more error–prone. However, there remain a non–trivial set of higher order mutants that are hard to kill.

We have found 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.

We show the exploration of the space of higher order mutants may reveal the structure of the test suite.

GP mutation testing is able to find complex faults denoted by higher order mutants of real programs that cannot be represented by any first order mutant and which are harder to kill than any first order mutant.