Genetic Improvement using Higher Order Mutation

Yue Jia, Fan Wu, Mark Harman and Jens Krinke
University College London
CREST, UK
{yue.jia, fan.wu.12, mark.harman, j.krinke}@ucl.ac.uk

ABSTRACT

This paper presents a brief outline of a higher-order mutation-based framework for Genetic Improvement (GI). We argue that search-based higher-order mutation testing can be used to implement a form of genetic programming (GP) to increase the search granularity and testability of GI.

Keywords

GI; SBSE; Higher Order Mutation

1. INTRODUCTION

Genetic Improvement (GI) seeks to automatically improve software systems by applying generic modifications to the program source code [7, 12, 14]. Given a human developed system as input, GI evolves new candidate implementations, which improve non-functional behaviours, while preserving the original functional requirements. Current research on GI has demonstrated many potential applications. For example, GI has been used to fix software bugs [1, 11], to dramatically speed up software systems [10, 14], to port a software system between different platforms [9], to transplant code features between multiple versions of a system [13], to grow new functionalities [4] and more recently the to improve memory [15] and energy usage [2].

The majority of GI work uses Genetic Programming (GP) to improve the programs under optimisation [1, 9, 10, 11, 12, 13, 14]. Early GI solutions attempted to apply strongly typed GP to evolve an entire program [1, 9, 14]. This GP approach uses a generic BNF grammar file that allows it to finely control the code generation. For example, the GP can evolve arbitrary new expressions by combining different variables and values with valid functions. However, such generic approaches also limit the scalability of GP-based GI. As a result only a set of small programs [1, 14] and a small part of a program [9] have been feasible for this kind of GI.

To scale up and cater for real world programs, later GI work used a so-called 'plastic surgery' GP approach [11, 10, 13]. Rather than evolving an entire program, this approach searches for a list of edits from the existing source code. To reduce search complexity, it uses a specialised grammar file that tracks the coarse syntactic information at the line-of-code or statement level. Typical changes generated are movements or replacements of different lines of code [10, 13]. Although this type of GP scales well and can be used to improve real world programs, the level optimisation is limited by the use of a specialised grammar file and the coarse level of genetic modifications.

In this paper, we propose to develop a GI framework using mutation testing [8]. We argue that recent advances in search-based higher-order mutation would allow GI to maintain a good level of scalability, while providing a fine-grained search granularity. Moreover, GI would also benefit from existing mutation-based test data generation frameworks with which, automated tests could be generated to improve the faithfulness of improved programs [5].

2. HIGHER ORDER MUTATION FOR GI

Mutation testing is an effective fault-based testing approach, which was first proposed in the 1970s [3]. It automatically seeds faults into the program under test to create a set of faulty version of the program, known as mutants. These mutants are used to assess the quality of given tests, as well as to provide a guideline for generating new tests. Recent evidence indicates that this approach is increasing in maturity and practical application [8].

The core fault seeding process uses source code manipulation techniques to create mutants. In the parlance of source code manipulation, each mutant is created by a source-to-source transformation of the original program. The transformation rules used in mutation testing are called mutation operators, designed to automatically modify the program thereby simulating a wide class of programmer changes [8]. This characteristic makes mutation testing a good alternative approach to evolve programs through GI.

Mutation testing can be classified into two types: first order and higher order. First order mutation generates mutants by introducing a single syntax change into the source code. This technique could be used for pre-sensitivity analysis at the beginning of the GI process [10]. Higher-order mutation applies multiple changes at multiple locations. Search-based higher-order mutation has been used to construct strong mutants than simulate subtle faults in real world programs [6]. We propose to use multi-objective search-based higher-order mutation testing to search for GI mutations that pass all the regression tests with improved non-functional properties.

GECCO '15, July 11 - 15, 2015, Madrid, Spain
© 2015 ACM. ISBN 978-1-4503-3488-4/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739482.2768417

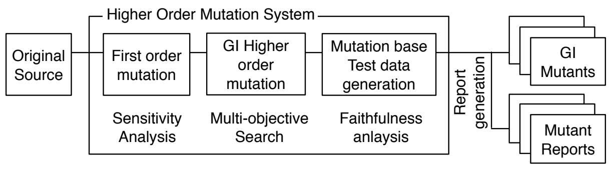

Figure 1: A higher-order mutation GI framework

The overall structure of the higher-order mutation-based GI framework is shown in Figure 1. This framework takes the program under optimisation as input, and applies traditional first order mutation to find locations that are sensitive to the non-functional properties under optimisation. This pre-analysis approach was first introduced by Langdon and Harman [10], to reduce the search space for GP. Their approach removes each line of code repeatedly, seeking changes that have a significant impact on non-functional properties. Our first-order mutation technique follows the same principle, but carries out the analysis at a finer grained level, including modifications to the variables within expressions.

The second step applies search-based higher-order mutation to find semantic-preserving mutants that could be useful for GI. It uses a vector to represent a higher order mutant, in which the indices represent the sensitive program points located in the previous analysis and the values represent the types of changes applied at each location. To search for higher order mutants preserving existing functional behaviours, one fitness function seeks to minimise the number of tests that capture the mutants. The search process could be implemented by reusing an existing higher-order mutation tool [6] with additional non-functional fitness functions, such as measuring the memory usage [15] or energy requirements [2]. As with 'plastic surgery' techniques [10, 11, 13], the higher-order mutation approach also searches for a list of changes. However, we believe this way will turn out to be flexible and provides a finer level of control in the code generation.

The framework applies a 'faithfulness' analysis after generation of candidate GI mutants. An improved program is faithful to a set of test data if it passes all of tests. Traditional GP-based approaches rely on a set of regression tests to check the faithfulness of the improved program. However, such regression tests might not be sufficient to thoroughly exercise the newly generated code. In the faithfulness analysis step, we attempt to apply additional mutation-based test data generation techniques [5] to find counter examples that kill the GI mutants. A GI mutant is killed, if a test input makes the evolved the program produce a different output to the original program, i.e. the original semantics have changed. This additional test data generation step would increase the faithfulness of the GI mutants, thereby providing additional confidence to the programmer.

Finally, for each candidate program generated, our approach creates a mutation report. The report summarises the types of mutation changes that have been applied to each variable or expression, based upon the mutation operators that have been used. This report will help to assist programmers to understand how such GI mutants can be used to improve the non-functional properties of their program. As the mutation operators are designed to mimic human syntactic changes, this form of report may prove to be more easily understandable than a report based upon line modifications.

The applicability of this approach depends on the number of GI mutants that still pass all tests. From a mutation testing point of view, the GI mutants are a subset of special mutants called equivalent mutants. Equivalent mutants are programs with syntactic differences, which nevertheless exhibit identical behaviour. Recent studies on equivalent mutants suggest that more than 23% of first order mutants are equivalent mutants on average[16]. Given that the number of mutants increases as the order of mutation increases, there are inevitably a large number of equivalent mutants produced by higher-order mutation. Thus there could be a sufficient number of equivalent mutants to be used by the higher-order mutation approach for GI.

3. CONCLUSIONS

Automatic software improvement is difficult for human developed systems. A good GI solution not only requires a generic and scalable way to modify programs, but also needs testing techniques to check for the faithfulness of the improved program. Higher-order mutation testing has proved to be a very effective source code manipulation approach for testing software. We therefore believe that this makes mutation testing a good candidate approach for GI.

4. REFERENCES

[1] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. *CEC*, pages 162–168, 2008.

[2] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *GECCO*, 2015.

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[4] M. Harman, Y. Jia, and W. Langdon. Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system. In *SSBSE Challenge*, pages 247–252, 2014.

[5] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *FSE*, 2011.

[6] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. *ASE*, 397–408, 2014.

[7] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *ASE*, pages 1–14, 2012.

[8] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *TSE*, 37(5):649–678, 2011.

[9] W. Langdon and M. Harman. Evolving a cuda kernel from an nvidia template. In *CEC*, pages 1–8, July 2010.

[10] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *TEC*, 19(1):118–135, Feb. 2015.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *TSE*, 38(1):54–72, Jan 2012.

[12] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *TEC*, 15(2):166–182, April 2011.

[13] J. Petke, M. Harman, W. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. *EuroGP*, 137–149, 2014.

[14] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *TEC*, 15(4):515–538, Aug. 2011.

[15] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In *GECCO*, 2015.

[16] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *ICSE*, pages 919–930, 2014.