New Operators for Non-functional Genetic Improvement

Justyna Petke
University College London, UK
j.petke@ucl.ac.uk

ABSTRACT

Genetic improvement uses automated search to find improved versions of existing software. Typically software is modified using either *delete*, *copy* or *replace* operations at the level of source code, its abstract syntax tree, binary or assembly representaion. Impressive improvements have been achieved through this approach, yet research in the use of other search operators is largely unexplored. We propose several ways for devising new search operators for improvement of non-functional properties using a genetic improvement apporach.

KEYWORDS

genetic improvement, GI, search operators, mutation operators, software mutational robustness

ACM Reference format:
Justyna Petke. 2017. New Operators for Non-functional Genetic Improvement. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 2 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082520

1 INTRODUCTION

Genetic improvement (GI) uses automated search to find improved versions of existing software. Even though this area of software engineering is relatively new, with the name being coined by Harman et al. in 2012 [5], it has already yielded impressive results. For example, Langdon and Harman's GI framework [8] found a version of a 50 000 line software system that is 70 times faster than the original, while Bruce et al. [3] reduced energy consumption of software by up to 25% using GI.

The typical GI process involves the use of genetic programming to modify code at the source code level or to make changes to its abstract syntax tree, binary or assembly representation. Other search approaches have been used less frequently, such as random search and hill climbing [1], although not in the context of improvement of non-functional software properties.

Perhaps due to the legacy of genetic programming, the most frequently used search operators for improvement of non-functional properties of software have been the *delete*, *copy* and *replace* mutation operators. Researchers in genetic improvement have been applying this now standard approach to report improvements on various existing software.

More fine-grained changes at the expression level have also been tried in the work on deep parameter tuning [17]. However, these are restricted to the standard operators used in mutation testing. With introduction of constraint-reasoning in the field of automated program repair, more complex statement-level changes have been applied [12–14]. These are, however, limited partly due to the expressiveness of the constraints in constraint solvers. Therefore, there is a need for greater analysis into which search operators lead to good solutions when using genetic improvement. An example idea was presented in a workshop paper by Wu et al., where they proposed the use of higher-order mutants [6].

We propose to take a step back and derive new ways of modifying code in a genetic improvement framework.

2 SOURCES OF NEW OPERATORS

We present three sources for deriving new mutation operators for the purpose of improvement of non-functional properties of software using genetic improvement.

2.1 Repository Mining

We propose to use existing solutions produced by software developers in devising new search operators. It has been recently shown [18] that a surprisingly large percentage of code snippets available on software developer fora, such as StackExchange[1], already contain executable code. We propose to mine changes made by software developers in open-source projects (GitHub repository[2] is one source of such code, containing over 38 million projects) with particular focus on improvement of the software property of interest, such as runtime efficiency. The results can then be used to devise new mutation operators in the form of templates. These could be then automatically transplanted into the original code during search, provided the required constraints are met. Restrictions on the data structures, for instance, might need to be considered to produce compilable software variants. Given that in previous work [15] significant runtime improvements have been achieved by transplanting code from other software variants, we propose to allow for the operators to reuse code not necessarily from the code to be improved itself.

In the automated program repair field, templates have already been used to fix bugs [7, 10, 11]. We could also draw from the latest approach proposed by Long et al. [11]. They created templates at the abstract syntax tree level. Therefore, the derived templates operate on a tree rather than source code level as is common in mutation testing.

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082520

[1] http://stackexchange.com/
[2] https://github.com/

2.2 Constraints

In order to increase the compilation chance of mutated code, profiling and grammars have been used to restrict the code change locations. Furthermore, as previously pointed out, most of the genetic improvement work to date reuses the same search, or in other words, mutation operators. Therefore, we propose to take inspiration from automated software repair work, in particular angelic debugging [13]. We propose to profile the code to find the most time consuming fragments and abstract program conditionals and variable assignments and derive value ranges for these such that all the training test data still pass. This is a novel idea; variation on the deep parameter tuning work with angelic execution replacing mutation testing operators.

The power of genetic improvement lies in potentially making arbitrary changes to the code. Therefore, we propose to combine templates derived from the repository mining stage and new angelic parameters with traditional genetic improvement operators to widen the pool of software modifications. We propose to investigate the efficacy of each of these techniques by measuring the effect of each type of changes on the fitness of modified software variants. Furthermore, we propose to test various search operator combinations in a systematic way.

2.3 Software Diversity

Throughout the genetic improvement process thousands (if not more) of software variants are being produced. Even though changes that are being made to the code are mostly random, high compilation rates are still being reported (e.g., 70-80% by Petke et al. [15]). There have been several studies showing that software is more robust than initially thought [9, 16]. Schulte et al. [16] defined the term software "mutational robustness" in terms of changes to computer code. They also defined a "neutral mutation" to be a random change that does not change program's behaviour as defined by program specification and its test suite. Therefore, those neutral mutants can still change the code semantics.

The concept of software mutational robustness is closely related to the goals of n-version programming [4] that aims to create diverse software variants that exhibit the same behaviour. Software diversity is useful for increasing its resilience against repeated software attacks [2, 4]. The genetic improvement approach can thus be used to create several software variants for this purpose. Schulte et al. [16] conducted several empirical studies, showing that 37% of software mutations were neutral, that is, 37% of the created software variants produced the same behaviour modulo test suite.

Insights from studies on mutational robustness can be utilised in the genetic improvement of non-functional software properties as follows. Given that high percentage of mutants created in such studies have no effect on program behaviour (modulo test suite), these are great candidates for finding a software version that improves upon some non-functional property. Therefore, one could run different versions of the same software within different environments, picking the best version in terms of, for example, runtime efficiency for that environment.

Furthermore, analysis of the various software mutants can yield to new search operators in the form of parameters that could be tuned. One obvious example is loop perforation. However, given the randomness of changes produced by GI, by analysing the different neutral mutants, one could identify structures that are more amendable to mutation. Therefore, not only new operators could be derived for the purpose of non-functional property optimisation, but the search itself could be more targeted towards more "robust" parts of the code.

3 CONCLUSIONS

Genetic improvement techniques have been successfully used to optimise various non-functional properties of software, ranging from runtime [8] through energy consumption [3] to memory consumption [17]. We propose to extend the standard set of search operators (*delete*, *copy* and *replace*) with new ways of mutating code. We proposed three ways of deriving such operators. We hope that we will inspire researchers in genetic improvement to pursue novel ways of modifying software and thus lead to further impressive empirical results and further growth of this exciting field of research.

REFERENCES

[1] Andrea Arcuri. 2011. Evolutionary repair of faulty software. *Appl. Soft Comput.* 11, 4 (2011), 3494–3514.

[2] Benoit Baudry and Martin Monperrus. 2015. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Comput. Surv.* 48, 1 (2015), 16:1–16:26.

[3] Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing Energy Consumption Using Genetic Improvement. In *GECCO*. 1327–1334.

[4] Benjamin Cox and David Evans. 2006. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006.*

[5] Mark Harman, William B. Langdon, Yue Jia, David Robert White, Andrea Arcuri, and John A. Clark. 2012. The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012.* 1–14.

[6] Yue Jia, Fan Wu, Mark Harman, and Jens Krinke. 2015. Genetic Improvement using Higher Order Mutation. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings.* 803–804.

[7] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013.* 802–811.

[8] William B. Langdon and Mark Harman. 2015. Optimizing Existing Software With Genetic Programming. *TEVC* 19, 1 (2015), 118–135.

[9] William B. Langdon and Justyna Petke. 2017. Software is Not Fragile. In *CSDC'15.*

[10] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1.* 213–224.

[11] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms and Search Spaces for Automatic Patch Generation Systems. http://hdl.handle.net/1721.1/103556. (2017). Technical Report.

[12] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *ESEC/FSE*. 166–178.

[13] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *ICSE*. 691–701.

[14] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *ICSE*. 772–781.

[15] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *EuroGP*. 137–149.

[16] Eric M. Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *GPEM* 15, 3 (2014), 281–312.

[17] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep Parameter Optimisation. In *GECCO*. 1375–1382.

[18] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of Stack Overflow code snippets. In *MSR*. 391–402.