Stack-Based Genetic Improvement

Aymeric Blot
University College London
London, United Kingdom
a.blot@cs.ucl.ac.uk

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

ABSTRACT

Genetic improvement (GI) uses automated search to find improved versions of existing software. If originally GI directly evolved populations of software, most GI work nowadays use a solution representation based on a list of mutations. This representation however has some limitations, notably in how genetic material can be recombined. We introduce a novel stack-based representation and discuss its possible benefits.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**.

KEYWORDS

Genetic improvement (GI); Automated Program Repair (APR); Search-based software engineering (SBSE)

ACM Reference Format:
Aymeric Blot and Justyna Petke. 2020. Stack-Based Genetic Improvement. In *GI@ICSE '20: 8th Workshop on Genetic Improvement at the International Conference on Software Engineering, May 24, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Genetic improvement (GI) [8] uses automated search to find improved versions of existing software. The earliest GI work used genetic programming (GP) [4] to directly evolve populations of software [2]. Representation of software variants however quickly changed to a more convenient intermediary representation that most GI work nowadays prefer. This intermediary representation is based on the modifications between the original software and the mutated variant, usually through the sequence of mutations that are to be applied. One of its main advantages is that it is much more compact that the software itself, focusing on the changes at a level much closer to human understanding and thus ultimately making the changes more likely to be adapted into development [11]. However, it also has some limitations, notably in terms of genetic material recombination [6].

Stack-oriented programming is a powerful, but very rarely used paradigm. Used, for example, in the Forth, Push, or Postscript programming languages, this paradigm provides the main computation procedure with stacks to store data or code and retrieve them later in reverse order. The most popular use of stacks is probably the reverse Polish notation, which was at some point used in many hand-held calculators. Stacks have been used in general GP systems [7], and particularly efficiently in the PushGP [10] system for autoconstructive evolution.

In this paper, we introduce a novel representation for GI systems, based on new types of edits and the use of stacks. We aim to propose a representation that keeps the simplicity of the usual representation, includes the recombination flexibility of a decoupled representation [6], while providing many other possible advantages.

GI@ICSE '20, May 24, 2019, Seoul, South Korea

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 GENETIC IMPROVEMENT SETUP

The usual GI setup involves computing for the target software a list of possible modification points, either based on physical representation (e.g., lines of code) or syntactical data (e.g., using the software's abstract syntax tree—AST). Using this list of modification points, a software variant is then described using a sequence of *edits*, within which the most common are the *deletion*, the *replacement*, and the *insertion* of pieces of software.

`Delete(l)` removes the code currently at location `l`;
`Replace(l1, l2)` replaces the code currently at location `l1` by the code originally at location `l2`;
`Insert(l1, l2)` inserts, *before* or *after* location `l1`, the code originally at location `l2`.

For fitness evaluation, the actual software variant is obtained by successively applying to the original software every mutation in order. This intermediary representation enables very quick mutant generation as well as efficient mutations and crossovers between software variants. In particular, individual edits can very easily be added or removed to software variants while maintaining most of their original semantic.

A downside of such a representation is that it only facilitates manipulation of coupled genetic material (here, the type of edits and the location of modification points). Data composing edit sequences can be decoupled into three sub-spaces [5] (namely *operator*, *fault*, and *fix* spaces), leading to new types of mutations and crossovers [6]. However, the complexity of the induced representation makes it much harder to use, despite its benefits.

3 PROPOSAL

We propose a novel GI software variant representation, based on a new set of edits. The main difference with the usual representation is the use of one or more stacks when the edit sequence is used to obtain the actual mutated software.

3.1 New Edits

The usual set of edits (deletion, replacement, insertions) are replaced by a set of new operations, that rely on the addition to the edit a sequence of stacks, which will contain locations of mutation points. We describe here the most basic case in which a single stack is used; the use of multiple stacks is discussed later. We define three new types of edits: *copy*, *cut*, and *paste*.

`Copy(l)` puts the location `l` on top of the stack;
`Cut(l)` deletes the code currently at location `l` and then puts the location `l` on top of the stack;
`Paste(l)` retrieves the location at the top of the stack and uses it to replace the code at location `l`.

Similarly to the two versions of the insertion edit (insertion before or after), three versions of the paste edit can be defined: to replace at position `l`, or to insert before or after position `l`. Additionally, it might also be interesting to reuse the deletion edit.

When evaluating the edit sequence to obtain the mutated software, it may happen than some of the operations require data from an empty stack. We propose three options. (1) It is possible to simply and safely ignore such operations; this means keeping in the representation inactive genetic material that could possibly be reactivated later through crossover or further mutation. (2) These operations can also be easily discarded, as are invalid genes [6]. (3) Finally, the empty stack can still produce some data, for example by returning a default value (e.g., an empty line/statement) or by keeping track of the last value retrieved from the stack.

The differences to the usual edit sequence representation may seem minimal. The usual four types of edits have been traded to either five or six operations, and obtaining the mutated software now requires the use of a stack. Edit sequences are necessarily at least slightly longer than beforehand, with replacements and insertions requiring both a copy and a paste operation. However, it is clear that this new representation is as least as expressive, as it can easily be converted back to the original one.

3.2 Decoupling and Recombination

One subtle difference between the two sets of edits is that the copy, cut, and paste (and possibly delete) edits all use a single argument. Using a decoupled representation [5] this would mean only two subspaces instead of three: the operator one, and a merged fault and fix one, inducing two sequences that will necessarily always have the same length. This consistent arity greatly simplifies decoupled mutations and crossovers [6] while avoiding any additional step to repair or discard invalid genes. In particular, any subset of operators could be recombined with any similar-sized subset of locations and necessarily result in a valid edit sequence.

3.3 Type-Based Stacks

Push [10] is a family of programming languages in which each data type (including code itself) is associated with a different stack. Our new representation can very easily benefit from such a scheme.

Indeed, modification points are intrinsically associated with pieces of software, e.g., lines of code, Boolean operators, or in general any sub-tree associated to the software AST. When all modifications points are homogeneous, e.g., only program statements or program conditions, then the GI system can ensure reasonable syntactic validity. In contrast, when multiple granularity levels are considered (e.g., simultaneous evolution of program statements and conditions [9]) operations—e.g., replacement, insertion; or here, paste—have to be consistent so to not result in invalid mutated software (e.g., an `if` condition being replaced by a full statement). While it is easy to generate consistent edits, maintaining validity after decoupling quickly becomes cumbersome as location types need to be taken in consideration. By using a different stack for each location types and enforcing that edits only use the stack associated to the type of their single argument, then operations are ensured to always use coherent data.

3.4 Content-Based Approaches

In addition or replacement to locations, edits in GI approaches may also directly include content data (e.g., [1, 3]). In that case, our stack-based representation can easily accommodate being modified to use additional stacks for content or mixed data.

4 CONCLUSION

Most GI work use edit sequences as solution representation for software variants. While very simple and easy to mutate and crossover, it has limitations in how genetic material can be recombined [6].

We presented a slightly modified solution representations, based on a set of new edits and the use of stacks. We expect this new stack-based solution representation to be straightforward to implement and at least as expressive and effective as the standard one, while keeping its simplicity and avoiding some of its current limitations. Lastly, we intend to challenge our claims through empirical analysis.

ACKNOWLEDGMENTS

This work is supported by UK EPSRC Fellowship EP/P023991/1.

REFERENCES

[1] Thomas Ackling, Bradley Alexander, and Ian Grunert. 2011. Evolving patches for software repair. In *Genetic and Evolutionary Computation Conference (GECCO 2011)*. ACM, 1427–1434.
[2] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation (CEC 2008)*. 162–168.
[3] Aymeric Blot. 2019. Fuzzy Edit Sequences in Genetic Improvement. In *International Workshop on Genetic Improvement (GI@ICSE 2019)*. ACM, 30–31.
[4] John R. Koza. 1992. *Genetic programming – On the programming of computers by means of natural selection*. MIT Press.
[5] Claire Le Goues, Westley Weimer, and Stephanie Forrest. 2012. Representations and operators for improving evolutionary software repair. In *Genetic and Evolutionary Computation Conference (GECCO 2012)*. ACM, 959–966.
[6] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. 2018. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering* 23, 5 (2018), 2980–3006.
[7] Timothy Perkis. 1994. Stack-Based Genetic Programming. In *International Conference on Evolutionary Computation (ICEC 1994)*. 148–153.
[8] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David Robert White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432.
[9] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2018. Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation. *IEEE Transactions on Software Engineering* 44, 6 (2018), 574–594.
[10] Lee Spector and Alan J. Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (2002), 7–40.
[11] Westley Weimer. 2006. Patches as better bug reports. In *Generative Programming and Component Engineering (GPCE 2006)*. ACM, 181–190.