Fuzzy Edit Sequences in Genetic Improvement

Aymeric Blot

Fuzzy Edit Sequences in Genetic Improvement

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

***Abstract*—Genetic improvement uses automated search to find improved versions of existing software. Edit sequences have been proposed as a very convenient way to represent code modifications, focusing on the changes themselves rather than duplicating the entire program. However, edits are usually defined in terms of practical operations rather than in terms of semantic changes; indeed, crossover and other edit sequence mutations usually never guarantee semantic preservation. We propose several changes to usual edit sequences, specifically augmenting edits with content data and using fuzzy matching, in an attempt to improve semantic preservation.**

***Keywords*-GI; genetic improvement; SBSE; search-based software engineering; fuzzy matching**

I. Introduction

Genetic improvement (GI) [1], [2] uses automated search in order to improve existing software. Because it is expected that, in the search space of all possible programs the evolved ones stay to some extent close to the original ones (see the "plastic surgery hypothesis" [3]), many GI work use representations based on the sequence of modifications applied to the original software rather than evolving it as a whole [4]–[8].

Edit sequences have many advantages. They are a sparse representation of the mutated programs and are very close to the human understanding of which changes are performed, making them more likely to be adapted into development [9]. Furthermore, they are easily combined, thus greatly facilitating crossover between different mutants. In addition, they are abstract enough so that they can be used with both linear representations (e.g., with the grammars of [6], [8]) and AST-based representations (e.g., in GenProg [5]).

Edits can be formulated using three pieces of information: (1) the type of modification (e.g., is the operation a *deletion*, an *insertion*, a *replacement*, a *swap*), (2) the place at which the edit takes place (e.g., which line of code has to be deleted), and (3) some new content if applicable (e.g., the content being inserted). The three associated search spaces are sometimes called *operation*, *fault location*, and *fix code* [10], [11]. Information about fault location and fix code are usually given through unique identifiers of modification points in the original source code. For example, following the notation used in [7], the edit "`i(1,2)`" (i.e., the triple "`(i,1,2)`") will represent the insertion at location "`1`" of the content originally at location "`2`", while "`d(3)`" will represent a deletion at location "`3`".

II. Motivation

As a motivating example, we consider the transformation of Listing 1 into Listing 2, in which two modifications are to be found: (1) as illustrated in Listing 3, the call to "`foo()`" should be moved from its original location "`a`" to the location "`b`"; and (2) as illustrated in Listing 4, the call to "`foo()`" should be preceded by a call to "`setup_foo()`" (already present somewhere in the source code, here at location "`z`"). The ideal edit sequence "`i(b,a)d(a)i(b,z)`" unfortunately cannot be directly obtained from the existing edits, because the insertion "`i(a,z)`" has to be modified into "`i(b,z)`" to follow the reinsertion of line "`foo()`" to its new location.

The idea behind our proposal is that a practitioner, being given two conflicting patches such as the ones of Listing 3 and Listing 4, may *presumably* be able to understand the semantic of the edit "`i(a,z)`" as "insert the line "`setup_foo()`" before the line "`foo()`"" rather than "insert the content of line "`z`" before line "`a`"", and *naturally* try to insert it as in Listing 2. In practice, we can expect to guide the GI process into automatically considering the variant "`i(b,z)`" of the edit "`i(a,z)`" following the semantic change induced by "`i(b,a)d(a)`"—that is, the content of location "`a`" being moved to location "`b`".

III. Proposal

It has already been shown, using crossover on decoupled edit sequences [7], that re-using knowledge already present in existing sequences (here, over all possible modification points only "`a`", "`b`", and "`z`" were used) can positively influence the creation of new edits. Furthermore, a lot of successful GI work already have demonstrated the relevance of relying on code semantic [2].

In the following, we propose to use the content relevant at the time of creation of an edit as a marker of the initial edit semantic, in order to generate new variants of the edit when this semantic is modified.

A. Content-first Edits

In the literature [5]–[8], [10], [11], edits are traditionally based on location first, and content second: in the edit "`i(a,b)`", "`a`" and "`b`" refer to modifications points usually in the original source code, and only through them the content at these locations. Instead, we propose to base edits on content first and location second. That is, given f a *lookup* function, that associates content with a location,

Listing (1) Original code

```
   ...
a: foo();
   ...
b: bar();
   ...
z: setup_foo();
   ...
```

Listing (2) Ideal code

```
   ...
a: // empty line
   ...
b: setup_foo();
   foo();
   bar();
   ...
z: setup_foo();
   ...
```

Listing (3) Mutant 1

```
// edits: i(b,a)d(a)
   ...
a: // empty line
   ...
b: foo();
   bar();
   ...
z: setup_foo();
   ...
```

Listing (4) Mutant 2

```
// edit: i(a,z)
   ...
a: setup_foo();
   foo();
   ...
b: bar();
   ...
z: setup_foo();
   ...
```

$\alpha = f(\texttt{a})$ and $\beta = f(\texttt{b})$ the contents at locations "a" and "b" at the time of the edit creation, we would like to use "op(α,β)" rather than using "op(a,b)" and perform lookup only when the edit is actually applied. Unfortunately, because the lookup function f cannot be inverted (while locations are unique, content is not), we propose to use both content and location, i.e., "op((α,a),(β,b))". For example, the edit "i(a,z)" of Listing 4 could be replaced by "i(("foo();",a),("setup_foo();",z))".

Note that there is no reason to actually store content data inside edits (e.g., α literally being the code fragment "foo();"), which unnecessarily enlarges edit size. Indeed, α and β can similarly simply correspond to content through identifiers to a separate bank of genetic material.

B. Fuzzy Matching

In our motivating example, this means that when appending the insertion of Listing 4 at the end of the edit sequence of Listing 3, the GI process now has the opportunity to realise that the line "foo()" has changed location. In the general case, when interpreting the edit "op((α,a),(β,b))", a check should be made to verify if the two matchings "(α,a)" and "(β,b)" are still valid. If not, then fuzzy matching can provide new plausible alternative variants of the edit, by searching for similar content at the same location (i.e., "(α',a)", with α' some content related to α) or for the same content at similar locations (i.e., "(α,a')", with "a'" a nearby location). Possible similarity metrics for content include string or tree edit distances. As for locations, similarity could be defined in terms of restriction to the context of the original location (e.g., the same method body).

This fuzzy matching can be performed every time the context of edits changes—i.e., whenever an edit in the middle of an edit sequence is inserted, modified, or deleted, or whenever a crossover is performed. When a conflict arises, and one or multiple plausible matches are found, they can then be used in order to help the GI process with additional diversity. If no sufficiently plausible match is found, then it might provide a clue to discard the edit. In any case, it can be expected that falling back to the current approach (i.e., applying edits without regard to the original content) should still be considered to avoid losing potentially useful edits.

IV. Conclusion

Edit sequences have been proven to be a very convenient and versatile solution representation for a lot of genetic improvement work. However, edit sequence implementations usually only focus on practical modifications [5]–[8], [10], [11] and often overlook semantics in the general GI literature [2].

We proposed to augment individual edits by the inclusion of content data. We expect that content data may be used to track semantic changes, which then, through fuzzy matching, may lead to the generation of new edits beneficial to the overall GI process.

References

[1] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 4, pp. 515–538, 2011.

[2] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, 2018.

[3] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*. ACM, 2014, pp. 306–317.

[4] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proceedings of the 13th Genetic and Evolutionary Computation Conference, GECCO 2011*. ACM, 2011, pp. 1427–1434.

[5] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[6] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, 2015.

[7] V. P. L. Oliveira, E. F. de Souza, C. Le Goues, and C. G. Camilo-Junior, "Improved representation and genetic operators for linear genetic programming for automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2980–3006, 2018.

[8] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Specialising software for different downstream applications using genetic improvement and code transplantation," *IEEE Transactions on Software Engineering*, vol. 44, no. 6, pp. 574–594, 2018.

[9] W. Weimer, "Patches as better bug reports," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006*. ACM, 2006, pp. 181–190.

[10] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012*. IEEE, 2012, pp. 3–13.

[11] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proceedings of the 14th Genetic and Evolutionary Computation Conference, GECCO 2012*. ACM, 2012, pp. 959–966.