PyGGI 2.0: Language Independent Genetic Improvement Framework

Gabin An
KAIST
Daejeon, Korea
agb94@kaist.ac.kr

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

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

Shin Yoo
KAIST
Daejeon, Korea
shin.yoo@kaist.ac.kr

ABSTRACT

PyGGI is a research tool for Genetic Improvement (GI), that is designed to be versatile and easy to use. We present version 2.0 of PyGGI, the main feature of which is an XML-based intermediate program representation. It allows users to easily define GI operators and algorithms that can be reused with multiple target languages. Using the new version of PyGGI, we present two case studies. First, we conduct an Automated Program Repair (APR) experiment with the QuixBugs benchmark, one that contains defective programs in both Python and Java. Second, we replicate an existing work on runtime improvement through program specialisation for the MiniSAT satisfiability solver. PyGGI 2.0 was able to generate a patch for a bug not previously fixed by any APR tool. It was also able to achieve 14% runtime improvement in the case of MiniSAT. The presented results show the applicability and the expressiveness of the new version of PyGGI. A video of the tool demo is at: https://youtu.be/PxRUdlRDS40.

CCS CONCEPTS

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

KEYWORDS

Search-based Software Engineering, Genetic Improvement

ACM Reference Format:
Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3338906.3341184

1 INTRODUCTION

Genetic Improvement (GI) uses an automated search to find improved versions of existing software [18]. It has already led to significant breakthroughs with GI-improved code incorporated into production [12, 14]. For functional property improvement, such as correctness, Automated Program Repair (APR) techniques based on the GI paradigm have made significant advances during the last decade [9, 22, 23, 26]. For non-functional property improvement, topics such as execution speed improvement [13], automated problem specialisation [19], and energy consumption [7] have been studied, to name a few.

Many of the existing GI techniques involve making modifications to the program source code and observing their effects on properties under observation, such as test execution results, execution time, or power consumption. This, in turn, requires effective and efficient ways to define modifications, i.e., GI operators, with respect to specific program representations. A wide range of approaches exist in the literature, ranging from line-level modifications [2], BNF grammar-like modifications [13], C Intermediate Language (CIL) based Abstract Syntax Tree (AST) modifications [22], and a custom Java parser based AST modifications [24]. Most of these are coupled with a single target language, such as C via CIL [22] or Java via JavaParser [24], as modifications have to be defined syntactically. The grammar-based approach of Langdon and Harman [13] captures syntax information by translating the program into a specific notation, on which GI operates: modifications made to the program representation become source code modifications. While this approach is theoretically language independent, Langdon and Harman's tool only supports C and C++ programs, and the framework would require internal code changes and a dedicated translation tool to apply it to other programming languages.

PyGGI has been originally introduced as an easy to use GI framework that is written in, as well as targets, Python [1, 2]. The initial release supported both line-level and AST-level modifications such as swap, insertion, and deletion. The choice of Python as the implementation language was a conscious one. The dynamic typing and interpreted runtime make it well-suited for fast prototyping. The choice of Python as the target language, however, was partly forced upon by the limited range of parsers for other languages implemented in Python (Python as the target language was supported by the use of internal `ast` module).

This paper introduces version 2.0 of PyGGI, which supports a wider range of target languages, such as Java, C/C++, and C#, via the use of an XML-based representation of program source code. In our case studies, we utilise the `srcML` parser. The tree representation of `srcML` has been used to perform various program analysis tasks [3–5, 10]. By using the `srcML` as an intermediate representation, users of PyGGI can easily implement GI techniques for multiple languages, without having to deal with multiple parsers.

We show the capabilities of version 2.0 of PyGGI with two case studies. The first one is an APR experiment using QuixBugs [15, 25] that contains 40 defective programs translated to both Java and Python. We show that PyGGI can be used to write a single

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3341184

APR algorithm that works for both languages. The second one is a replication of MiniSAT program specialisation [20] (the original work used line-level modification). We show that PyGGI is capable of finding similar improvements.

We believe that PyGGI 2.0 will contribute towards faster uptake and popularisation of GI techniques. With the new XML engine, the framework allows for quick experimentation among multiple programming languages. PyGGI 2.0 is publicly available at https://github.com/coinse/pyggi.

2 DESIGN OF PyGGI 2.0

With this new version, PyGGI focuses on flexibility, versatility and expressiveness. Its core structure has been upgraded, most of its components being extracted and generalised, in order to further support future extensions and variations for particular applications. In addition, PyGGI 2.0 now provides support for XML files as a way to handle multiple programming languages. In this section, we first discuss the architecture of PyGGI 2.0, then introduce its notion of *engines*, before finally describing how XML is used as an intermediary source code representation.

2.1 From PyGGI 1.1 to PyGGI 2.0

The initial version of PyGGI [1] only targeted language-agnostic source code lexical modifications, i.e., it only considered mutation of full raw lines of code. PyGGI 1.1 [2] introduced support for the second type of mutations, targeting Python lines of code, thus enabling an empirical study comparing lexical and syntactic mutations. However, PyGGI 1.1 was built directly on top of the first, purposely very simple and straightforward, version of PyGGI. Granularity level was also strongly tied to the choice of the specific parser used. Consequently, its codebase was monolithic, with intertwined components sharing multiple responsibilities, and overall not adapted to further extensions.

If PyGGI 1.1 was an easy gateway for practitioners to try and use GI, PyGGI 2.0 aims to also provide researchers with a cleaner and more robust environment to try out new ideas, implement new functionalities, and perform experiments. In particular, GI components are generalised and abstracted so that concepts can be more easily compared over multiple types of granularity levels or types of source code targeted.

While PyGGI 1.1 implementation was contained within a single Python module —`pyggi`— PyGGI 2.0 makes use of Python submodules to further structure its codebase, described hereafter:

- **`pyggi/base`** is the main submodule of PyGGI 2.0; it defines the base classes of programs, engines (introduced in the following section), patches, edits, and algorithms.
- **`pyggi/algorithms`** contains the local search of PyGGI 1.1. More algorithms are planned to be integrated in the near future.
- **`pyggi/utils`** includes general helpers.

In addition, code pertaining to the two granularity levels of PyGGI 1.1 have been relocated into the two following submodules:

- **`pyggi/line`** defines array-based program representations and mutations. It includes PyGGI 1.1's line-based representation.
- **`pyggi/tree`** defines tree-based program representations and mutations. It includes PyGGI 1.1's representation of Python statements, together with the new XML representation.

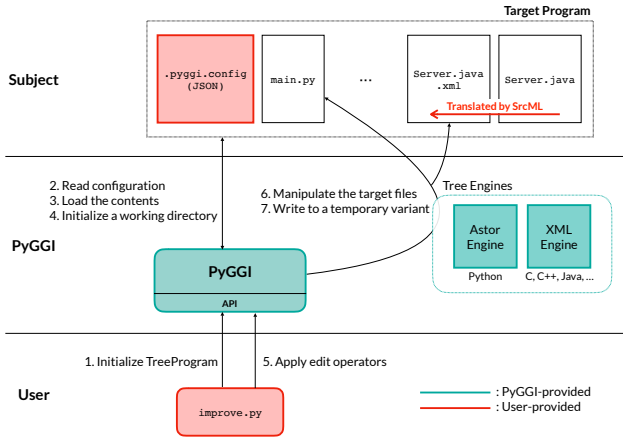

Figure 1: Workflow of PyGGI 2.0 for tree-based programs

2.2 File-Specific Engines

Together with the structural refactoring, the other main feature of PyGGI 2.0 architecture is the introduction of *engines*. While multiple files could be considered at the same time, in PyGGI 1.1 granularity was a global property, i.e., all files of the targeted source code had to share the same granularity level. In PyGGI 2.0 this constraint is lifted as different parts of the same source code can now be managed by different *engines*. Engines define both the representation of a single source code file —how to parse the initial contents of the file together with their modification points and how to convert back to text format— and the available atomic operations that can be performed on it, e.g., deletion, replacement, or insertion.

PyGGI 2.0 provides two types of engines naturally associated to the two granularity levels of PyGGI and the two submodules `pyggi/line` and `pyggi/tree`. Each of the two submodules defines an abstract interface enabling edits to be shared between engines of the same type. In total PyGGI 2.0 provides three concrete engines, one under `pyggi/line` for general line-based operations, and two under `pyggi/tree` for Python statements and XML trees. Figure 1 details PyGGI 2.0's usual workflow for a tree-based program.

Engines enable granularity level to be dissociated from the concrete source code parser. This means, for example, that any experiment on a specific language (e.g., Python) can easily be replicated on another (e.g., `C++`) as long as both parsers implement the same granularity level abstract interface. In practice, the XML engine provides a shared representation at very high granularity for source code, greatly improving PyGGI's scope for potential experiments.

2.3 XML Integration

The two modes of PyGGI 1.1 enabled it to either consider language-agnostic files at the line granularity level, or Python files at the statement level. The third engine of PyGGI 2.0 introduces handling of XML files, and enables it to easily tackle `C`, `C++`, `C#` and Java files at various granularity levels through the use of the `srcML`[1] parser.

[1] https://www.srcml.org/

Listing 1: C++ code, srcML translation, modification points

```
if ( j > i ) {
  x = j;
}
-------------------------------------------------
<stmt>if <condition>( j &gt; i )</condition> {
  <stmt>x = j;</stmt>
}</stmt>
-------------------------------------------------
% /stmt[1]
<stmt>if <condition>( j &gt; i )</condition> {
  <stmt>x = j;</stmt>
}</stmt>
% /stmt[1]/condition[1]
<condition>( j &gt; i )</condition>
% /stmt[1]/stmt[1]
<stmt>x = j;</stmt>
```

Listing 1 shows how source code can be translated to XML. Note that srcML outputs a highly detailed XML tree, which is here simplified to a much simpler format for the sake of keeping a reasonable search space. For example, the statement "x = j;" would actually be converted into the very detailed following XML fragment: "<expr_stmt><expr><name>x</name> <operator>=</operator> <name>j</name></expr>;</expr_stmt>", allowing the consideration of much more precise and specific edits. XML also provides XPath as a very convenient way of traversing the tree of nodes, with, for example, the path "/stmt[1]/condition[1]" corresponding to the first "condition" child of the first "stmt" child of the root node. This is similar to the strategy already used in the Python engine to access specific statements.

3 EXPERIMENTAL DESIGN

In order to show how PyGGI 2.0 can be used for program improvement, we present two case studies. The first one is concerned with the improvement of a functional property (repair), while the other is focused on non-functional property improvement (runtime efficiency). We also target 3 programming languages: Python, Java, and C. In this section we outline our experimental design.

3.1 Automated Program Repair

3.1.1 Dataset. We evaluate PyGGI 2.0 on the QuixBugs benchmark [15, 25], which consists of 40 defective programs translated into both Python and Java. As only 31 of the programs have a test suite, we target those programs as our repair subjects. Furthermore, for 11 of 31 defective programs failing on all test cases, we tried additionally to generate passing test cases since this may make it difficult to distinguish the original faulty program from even worse programs. To do so, we repeatedly mutated the initial failing test inputs until finding passing test inputs that satisfy the described input precondition and yield the same output on both correct and defective programs. As a result, we succeeded to generate such passing test cases for 8 out of 11 programs, and the new test cases are merged into the benchmark's master branch. All defective Java programs are translated to XML files using srcML Beta v0.9.5.

Table 1: Number of unique QuixBugs patches

	Python		Java	
	Line	Statement	Line	Statement
lis	2	3	3	4
wrap	0	4	0	0
quicksort	0	0	0	3
sieve	0	0	0	3

3.1.2 Experimental Setting. The experiment is conducted at both *line* and *statement* granularity level, with three modification operators *deletion*, *replacement*, and *insertion*. For the Java programs translated to XML files by srcML, we targeted only srcML elements classified as statements[2] along with "decl_stmt" and "expr_stmt". To evaluate each candidate patch, we use a simple fitness function defined by the number of failing test cases (including test cases that timed out), and a basic descent first hill climbing algorithm is employed as a search algorithm. In each step, either a random edit is added to the current best patch or one of the existing edits is removed from the best patch to generate neighbouring solutions. The time limit for test suite execution is set to 10 seconds, and each run of the hill climbing search is given the fitness evaluation budget of 500 steps: the stopping criterion is either when the budget expires, or when a plausible patch is found. We execute the repair experiment 20 times for each fault.

3.1.3 Results. The hill climbing algorithm is able to generate 22 plausible (test-suite adequate) patches for four programs among the 31 defective programs, and the number of unique patches is reported in Table 1. Both Python and Java versions of *lis* are repaired in both granularity levels, whereas the other three programs are repaired in only one combination of language and granularity.

Interestingly, the program *sieve* have not been repaired by any repair system in previous work [25]. The three plausible patches of *seive*, which are semantically, but not syntactically, equivalent, consist of more than one atomic operation, while the patches for the other programs are composed of only one operation. This patch shows that the simple hill climbing algorithm can gradually find multi-edit patches when an appropriate partial repair is generated. Overall, the results show that PyGGI 2.0 can be used to implement program repair systems in different programs languages, Python and Java, and also at different granularity levels.

3.2 Running time Improvement

3.2.1 Dataset. As for our second case study, we consider a running time optimisation scenario specialising the MiniSAT [8] SAT solver, building on previous GI work [20, 21]. In particular, we use an existing instrumentalised MiniSAT source code —based on MiniSAT2-070721— from which we translate the main solving algorithm ("Solver.C") using srcML, and a benchmark of 130 combinatorial interaction testing (CIT) SAT instances.

3.2.2 Experimental Setting. We operate on both statements and Boolean conditions. Most of the tags of the MiniSAT XML tree are ignored, as we only consider statement ones (e.g., "break", "continue", "decl_stmt", "do"), together with the "condition" tags

[2] See the *Statements* row at https://slides.com/collard/srcmloverview#/10

Table 2: MiniSAT evolved mutants

Mutant	Lines of code	Time (sec)
baseline	28398038591 (100.0%)	67.49 (100.0%)
seed 0	24247029088 (85.4%)	67.36 (99.8%)
seed 3	28094544573 (98.9%)	67.23 (99.6%)
seed 4	23327239091 (82.1%)	72.01 (106.7%)
seed 5	22496801475 (79.2%)	62.36 (92.4%)
seed 6	25050800206 (88.2%)	63.51 (94.1%)
seed 7	20066013444 (70.7%)	58.66 (86.9%)
seed 9	18197820457 (64.1%)	58.04 (86.0%)
seed 22	26562843149 (93.5%)	76.15 (112.8%)
seed 26	23229870424 (81.8%)	65.65 (97.3%)

of "`do`", "`for`", "`if`", and "`while`" statements. As in the previous case study, we consider *deletion*, *replacement*, *insertion* of either statements or conditions. Mixed mutations (e.g., replacement of a statement by a Boolean condition) are forbidden. Finally, Boolean conditions such as "`<condition>foo</condition>`" are automatically rewritten as "`<condition>(foo)||</condition>0`" so that deletion and insertion of conditions work as expected.

Following the previous work [20, 21], in order to have a deterministic fitness function, during training we count the number of statements of "`Solver.C`" executed as a proxy for runtime. This metric is easily obtained by prefixing a global counter increment before all single-line statements and at the beginning of every "`do`", "`for`", and "`while`" statements.

Finally, as GI search process we use PyGGI 2.0's local search with a budget of 2000 steps. Previous work used a genetic programming approach with 5 instances selected in each generation from 5 bins (based on instance difficulty and satisfiability), containing overall 74 instances. Since we do not change instances during the search, we increase the size of the training set to 15, in order to avoid overfitting. Each mutant is first compiled, then executed on 15 instances selected at random at the beginning of the search from the training set. Mutants failing to solve all 15 instances are immediately discarded. Training is performed 30 times, with different random seeds. Performance of the 30 final mutants is then reassessed using the second test set of 56 SAT instances (used in previous work).

3.2.3 Results. Table 2 shows the assessment of 9 of the 13 final mutants that were able to correctly solve every of the 56 previously unseen test instances, averaged over 30 executions. As for the 21 other mutants, 4 correctly solved every instance but required noticeably more time than the baseline (between 100 and 200 seconds), 10 incorrectly classified at least one instance, 5 were discarded after spending more than 120 seconds on a single instance, and finally 2 experienced errors during execution.

The best mutant —seed 9— reduced to only 64.1% the cumulative amount of statements executed by the baseline (the empty patch, i.e., the original source code) on all 56 test instances. Improvements in fitness mostly translate to improvement in running time, with the best mutant clearing the test benchmark in 58.04 seconds, compared to the 67.49 seconds of the baseline, improving it by 14%.

Furthermore, analysis of mutant 9 highlighted a mutation which applied on its own yielded a 19.4% speed-up in running time. This mutation inserts a line manipulating variable activity levels, thus rebalancing the priority queue for variable assignment during search. This mutation is different from the one-line "good change" modification found in previous work [20, 21]. Interestingly these two mutations are compatible, leading to a mutant clearing the test benchmark without error in only 49.44 seconds (26.8% speed-up).

4 RELATED WORK

The area of Genetic Improvement (GI) arose as a separate field of research only in the last few years [18]. GI tools can be divided into two categories: those that deal with the improvement of functional and non-functional program properties.

In the first category program repair tools[3], such as GenProg [11], have gathered a lot of attention and led to the development of the field of Automated Program Repair (APR). Within the field, however, currently only the ASTOR [17] framework allows for comparison of different repair approaches. Another functional property for improvement tackled by GI is the addition of a new feature [16].

With regards to improvement of running time, memory or energy consumption, there is a plethora of GI frameworks available that target a specific programming language [21]. However, a lot of these tools are not available, and, aside from one exception, have not been designed to be general GI frameworks. The closest in the objectives of PyGGI is the Gin toolbox [6, 24]., which targets Java.

There also exist a few code manipulation frameworks that came from the field of GI. Among these, the Software Evolution Library (SEL)[4] is worth mentioning, as it aims to manipulate multiple programming languages. However, it's been written in Lisp and requires a substantial learning overhead. PyGGI, on the other hand, aims to be a light-weight framework for work in GI.

5 CONCLUSIONS

We present PyGGI 2.0, a Python General Genetic Improvement framework, that allows for quick experimentation in GI for multiple programming languages. This is achieved by the use of XML representation incorporated in version 2.0 of the tool. We conducted two experiments, showing two usage scenarios of PyGGI 2.0: for the purpose of improvement of functional (repair) and non-functional (runtime efficiency) properties of software. We show that PyGGI 2.0 can find 22 patches for four programs from the QuixBugs benchmark, including a fix not previously produced by an APR tool. We were able to find these both in the Python and Java implementations of the subject programs. Moreover, we show that PyGGI 2.0 can also find efficiency improvements of up to 14% in the MiniSAT solver when specialising for a particular application domain, finding additional improvements to previous work. We thus demonstrate that PyGGI 2.0 is a useful tool for GI research, facilitating quick comparisons between different programming languages.

ACKNOWLEDGEMENT

Gabin An and Shin Yoo are supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068179). Aymeric Blot and Justyna Petke are supported by UK EPSRC Fellowship EP/P023991/1.

[3] http://program-repair.org/tools.html
[4] https://grammatech.github.io/sel/

REFERENCES

[1] Gabin An, Jinhan Kim, Seongmin Lee, and Shin Yoo. 2017. PyGGI: Python General framework for Genetic Improvement. In *Proceedings of Korea Software Congress (KSC 2017)*.

[2] Gabin An, Jinhan Kim, and Shin Yoo. 2018. Comparing Line and AST Granularity Level for Program Repair using PyGGI. In *Proceedings of the 4th Genetic Improvement Workshop (GI@ICSE 2018)*. 19–26.

[3] Gabriele Bavota and Barbara Russo. 2016. A Large-scale Empirical Study on Self-admitted Technical Debt. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/2901739.2901742

[4] Dave Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shin Yoo. 2017. Tree-Oriented vs. Line-Oriented Observation-Based Slicing. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 21–30.

[5] David Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shin Yoo. 2018. A Comparison of Tree- and Line-Oriented Observational Slicing. *Empirical Software Engineering* to appear (2018).

[6] Alexander E.I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White. 2019. Gin: Genetic Improvement Research Made Easy. In *Proceedings of the 2019 Annual Conference on Genetic and Evolutionary Computation (GECCO '19)*. ACM, 8.

[7] Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing Energy Consumption Using Genetic Improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. ACM, New York, NY, USA, 1327–1334.

[8] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003) (Lecture Notes in Computer Science)*, Vol. 2919. Springer, 502–518.

[9] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*. ACM, New York, NY, USA, 947–954.

[10] Daniel M. German, Bram Adams, and Kate Stewart. 2019. cregit: Token-level blame information in git version control repositories. *Empirical Software Engineering* (2019). https://doi.org/10.1007/s10664-019-09704-x

[11] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. 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*. 3–13.

[12] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. 2017. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, Peter A. N. Bosman (Ed.). ACM, 1513–1520. https://doi.org/10.1145/3067695.3082517

[13] W.B. Langdon and M. Harman. 2015. Optimizing Existing Software With Genetic Programming. *Transactions on Evolutionary Computation* 19, 1 (2015), 118–135.

[14] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.). ACM, 1063–1070. https://doi.org/10.1145/2739480.2754652

[15] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017)*. ACM, New York, NY, USA, 55–56. https://doi.org/10.1145/3135932.3135941

[16] Alexandru Marginean, EarlT. Barr, Mark Harman, and Yue Jia. 2015. Automated Transplantation of Call Graph and Layout Features into Kate. In *Search-Based Software Engineering*, Márcio Barros and Yvan Labiche (Eds.). Lecture Notes in Computer Science, Vol. 9275. Springer International Publishing, 262–268. https://doi.org/10.1007/978-3-319-22183-0_21

[17] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*. https://doi.org/10.1145/2931037.2948705

[18] J. Petke, S. Haraldsson, M. Harman, w. langdon, D. White, and J. Woodward. 2017. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* PP, 99 (2017), 1–1. https://doi.org/10.1109/TEVC.2017.2693219

[19] 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 *Genetic Programming*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo García-Sánchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–149.

[20] 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 *17th European Conference on Genetic Programming (LNCS)*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim (Eds.), Vol. 8599. Springer, 137–149.

[21] 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.

[22] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09)*. IEEE, Vancouver, Canada, 364–374.

[23] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[24] David R White. 2017. GI in no time. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 1549–1550.

[25] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A comprehensive study of automatic program repair on the QuixBugs benchmark. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 1–10.

[26] Y. Yuan and W. Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 1, 1 (2018), 1–1.