Dissecting Copy/Delete/Replace/Swap mutations: Insights from a GIN Case Study

Sherlock A. Licorish
Department of Information Sciences, University of Otago
Dunedin, New Zealand
sherlock.licorish@otago.ac.nz

Markus Wagner
School of Computer Science, The University of Adelaide
Adelaide, Australia
markus.wagner@adelaide.edu.au

ABSTRACT

Research studies are increasingly critical of publicly available code due to evidence of faults. This has led researchers to explore ways to improve such code, with static analysis and genetic code improvement previously singled out. Previous work has evaluated the feasibility of these techniques, using PMD (a static analysis tool) and GIN (a program repair tool) for enhancing Stack Overflow Java code snippets. Results reported in this regard pointed to the potential of these techniques, especially in terms of GIN's removal of PMD's performance faults from 58 programs. We use a contextual lens to explore these mutations in this study, to evaluate the promise of these techniques. The outcomes show that while the programs were syntactically correct after GIN's mutations (i.e., they compiled), many of GIN's mutations changed the semantics of the code, rendering its purpose questionable. However, certain code mutations tend to retain code semantics more than others. In addition, GIN's mutations at times affected PMD's parsing ability, potentially increasing false negatives. Overall, while these approaches may prove useful, full utility may not be claimed at this time. For enhancing the outcomes of these approaches, we outline ways to improve the utility of these techniques and multiple future research directions.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Search-based software engineering**.

ACM Reference Format:
Sherlock A. Licorish and Markus Wagner. 2022. Dissecting Copy/Delete/Replace/Swap mutations: Insights from a GIN Case Study. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3520304.3533970

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9268-6/22/07…$15.00
https://doi.org/10.1145/3520304.3533970

1 INTRODUCTION/MOTIVATION

On the premise that code-hosting websites such as Stack Overflow[1] and HackerRank[2] have become the cornerstone for software developers seeking solutions to their coding challenges [6], and because such code can at times possess faults [2, 3, 7], there have been efforts aimed at automating code improvement on such portals [5, 9]. In particular, Licorish and Wagner [5] use the PMD static analysis tool to detect performance faults for a sample of Stack Overflow Java code snippets, before performing mutations on these snippets using GIN, where, of 17,986 unique patches, PMD violations were removed from 770 patched versions. These authors also reported that 58 mutations (of 44 different snippets) no longer showed any performance issues, meaning that GIN mutations removed all performance violations.

While this result seems promising on the surface, Licorish and Wagner [5] cautioned that there is need for deeper contextual probing of the outcomes. These authors also noted that there were several issues to consider for such techniques to be proven viable, including the mitigation of false positives, parsing improvements and expansion of rules for static analysis techniques (such as those used by PMD), and improvement in sampling, code transformations and adapting to the expansion of rules for automated code improvement approaches (for tools like GIN). That notwithstanding, approaches that utilise such lightweight techniques to automatically enhance code (online or otherwise) could be useful for the search based software engineering community. We evaluate this promise in this study, where we have performed this contextual probe to validate the effectiveness of GIN mutations.

In this article, we dig into the results of the study by Licorish and Wagner [5], who investigated the effects of code mutations (using GIN [1]) on the output of a static checker (PMD [8]). Among their thousands of investigated mutations, one set stands out: the set of 58 mutations that result in compilable code that also no longer *allegedly* exhibits performance-related PMD errors. We have obtained that dataset via private communication, and we report in this article on the actual effects of the mutations. Both authors were involved in the systematic contextual examination of the mutations, exploring the code states before and after GIN's operations. The outcomes show that many of GIN's mutations changed the semantics of the code, affecting its intended purpose. In addition, while some mutations resulted in the code semantics being retained or mostly retained, at times GIN's operation affected PMD's parsing ability. This undermines the confidence in the false negative rate of PMD and it has implications for the adoption of these approaches.

[1] https://stackoverflow.com/
[2] https://hackerrank.com/

Table 1: Based on the 44 original snippets, the 58 mutations remove 62 PMD errors; the 58 mutations are error-free. Examples of rule instantiations are shown as <...>.

rule	count	description
UseStringBufferForStringAppends	26	Prefer StringBuilder (non-synchronized) or StringBuffer (synchronized) over += for concatenating strings.
AvoidArrayLoops	15	System.arraycopy is more efficient.
AddEmptyString	6	Do not add empty strings.
AppendCharacterWithChar	5	Avoid appending characters as strings in StringBuffer.append.
InefficientStringBuffering	3	Avoid concatenating nonliterals in a StringBuffer/StringBuilder constructor or append().
InefficientEmptyStringCheck	2	String.trim().length() == 0 / String.trim().isEmpty() is an inefficient way to validate a blank String.
TooFewBranchesForASwitchStatement	2	A switch with less than three branches is inefficient, use an if statement instead.
AvoidInstantiatingObjectsInLoops	2	Avoid instantiating new objects inside loops.
ConsecutiveLiteralAppends	1	StringBuffer (or StringBuilder).append is called *<3>* consecutive times with literals.

Among our contributions in this paper, we outline ways to improve the utility of these techniques.

The remaining sections of this paper are organised as follows. We describe the studied dataset in Section 2. We next detail the methodology that is used for data annotation, after briefly summarising the previous procedures of Licorish and Wagner [5] in Section 3. We provide our observations in Section 4, before summarising our outcomes and outlining future opportunities in Section 5.

2 DATASET

The starting point of the study by Licorish and Wagner [5] are the 8010 Java code snippets from Stack Overflow that were provided by [7]. PMD finds performance issues in 1203 of these files. Following a sampling of the single-edit space with GIN, 58 single-edit mutations (of 44 different snippets) no longer show any performance issues and the code is compilable: one issue is removed in 54 cases, and two issues are removed in four cases.

Table 1 shows the performance-related errors of the original 44 code snippets: PMD's error name, the number of times that error has been observed, and a brief description of the error in English (as per PMD).

3 METHODOLOGY OF DATA ANNOTATION

In this article, we annotate the 58 mutations with a focus on whether or not a human would deem the mutation acceptable. This high-level characterisation is performed in two rounds of manual, inductive analyses:

- Round 1) Each author annotates 10 uniquely selected mutations. This is achieved by one author selecting the top 10 mutations, and the other author selecting the bottom 10. The goal: assess the utility of the mutation.
  (1) Describe the change to the semantics of the program.
  (2) Answer the question: "Are the semantics retained? Possible answers: yes/mostly/no"
- Intermediate step: both authors discuss the interpretations to align their protocols and establish agreement. For similar errors (e.g., those snippets where the AvoidArrayLoops error was removed by GIN), the authors discuss their feedback, and strive for convergence in the way the responses were reported. Overall, while the process did not lead to the formal computation of inter-rater agreement, we achieved 100% agreement in our responses to similar errors.
- Round 2) The remaining mutations are then annotated equally. Thus, altogether, each author analysed 29 mutations.

Previously, Licorish and Wagner [5] used Java code snippets extracted from Stack Overflow for 2014, 2015, and 2016 that were said to be compilable code from answers which were highly reused [6]. Various forms of preprocessing were performed on these snippets to make them suitable for program analysis [7], before PMD [8] and GIN [1] were used for experimentation.

PMD is a static analysis tool that has 324 Java-based rules (`rulesets/internal/all-java.xml`) organised in eight sets: Best Practices, Code Style, Design, Documentation, Error Prone, Multi-Threading, Performance, and Security. This tool accepts code as input for analysis, before returning a summary of the errors in the code under these eight dimensions [8].

GIN is an extensible and modifiable toolbox for search-based experimentation with code [1]. GIN automatically transforms, builds, and tests Java projects. GIN's RandomSampler randomly generates a patch (which is composed of a given number of individual edits) which is then applied to the code before testing. The RandomSampler does not perform a random walk or any iterated search via a sequence of patches, but it always takes the original file as the starting point for the application of the next patch, allowing the characterisation of neighbourhoods in the program space.

The patches of Licorish and Wagner [5] comprised one of the following eight: DeleteLine, ReplaceLine, CopyLine, and SwapLine; and DeleteStatement, ReplaceStatement, CopyStatement, and SwapStatement. These authors generated 10,000 patches with one line edit, and 10,000 patches with one statement edit, resulting in 17,986 patches (due to patches being randomly re-selected at times). As noted above, our goal in this work is to investigate Licorish and Wagner [5]'s outcomes, particularly for GIN's mutations that remove PMD's errors in the code of 58 mutations.

Note that a human annotation is necessary in this case, as no test cases are available for the original code snippets. While it would be possible to generate tests using test case generators, and to consider these then as the ground truth for a characterisation of the landscape, it is unclear if the tests would adequately capture the semantics. Hence, for this preliminary study, we rely on our programming expertise to annotate the data. To this end, we accept that under normal operation, GIN may strive for code correctness by repeated patch generation given the outcomes of test cases. With no test cases used here however, the mutations are not validated through automated means.

```
1 public class C66208{
2     public static String expand(String word) {
3         int stringLength = word.length();
4         StringBuffer buffer = new StringBuffer();
5         for (int i = 0; i < stringLength - 1; i++) {
6             buffer.append(word.substring(i, i + 1));
7             buffer.append("-");
8         }
9         buffer.append(word.substring(stringLength - 1,
              stringLength));
10        return buffer.toString();
11    }
12 }
```

Listing 1: Code snippet C66208 with error AppendCharacterWithChar, mutation DeleteStatement(64). The deleted statement is shown in red.

We report our observations from the manual, inductive analyses next.

4 OBSERVATIONS

We list the results of the annotation in Table 2. Immediately, we can make two observations. First, we can observe that 36 of the 58 mutations (i.e., the vast majority) are the result of DeleteStatement and DeleteLine operations. While this is in contrast to the study of Le Goues et al. [4], where deletions and replacements were almost equally frequent, this difference might be an artefact of our set of programs and of our small sample size. Second, we notice that the semantics are retained in only two cases, most of the semantics are retained in six cases, and the semantics undergo a major change (denoted as "semantics retained: no") in the remaining 50 cases. As a single example of a major, non-trivial change to the semantics (where we may see changes like "deletes entire for loop" as a trivial, semantics-changing case), we show in Listing 8 a case where the loop body was swapped with a statement outside of the loop.

When digging into our interpretation of the mutations, we can see that almost all *fixing* mutations remove the offending code and thus change the semantics. In a small number of cases, the code deletions might be deemed acceptable. The two cases of the CopyLine and CopyStatement mutations are curious (see Listing 6 for both), not because the two mutations result in the same code, but because we are of the opinion that PMD should still be reporting the performance-related issue AvoidArrayLoops.

5 SUMMARY AND FUTURE OPPORTUNITIES

When considering the 58 code snippets in our sample that all compile without PMD errors after mutations (i.e., they are without syntax errors), it appears like DeleteStatement and DeleteLine mutations result in fewer syntactic code anomalies than the other operations. This conclusion is drawn as, of the 1203 Java snippets that were repaired by Licorish and Wagner [5], only 58 code snippets compile with no performance issue (i.e., meaning that they had no syntactic errors). Of note is that a total of 38 mutations (62.1%) were of these DeleteStatement and DeleteLine types in Table 2. On the contrary, the six other types of mutations (i.e., ReplaceLine, CopyLine, SwapLine, ReplaceStatement, CopyStatement, and

```
1 public class C113624{
2     protected StringBuilder setOne(){
3         StringBuilder builder=new StringBuilder();
4         try{
5             builder.append("Cool"); // [1]
6             return builder.append("Return"); // [2]
7         } finally {
8             builder.append("+1"); //[3]
9         }
10    }
11 }
```

Listing 2: Code snippet C113624 with error ConsecutiveLiteralAppends, mutation DeleteLine(5). The deleted line is shown in red.

```
1  public class C204957{
2  public static String convert(String in) {
3    String[] strs = in.split("\\.*000\\.*");
4    StringBuilder sb = new StringBuilder();
5    for (int i = strs.length - 1; i >= 0; --i) {
6      sb.append(strs[i]);
7      if (i > 0 && strs[i - 1].length() > 0) {
8        sb.append(".");
9      }
10   }
11   return sb.toString();
12 }
13
14 public static void main(String[] args) {
15   System.out.println(convert("010.011.100.000.111"));
16   System.out.println(convert("
         010.011.100.000.111.001.011.000.101.110"));
17   System.out.println(convert("010.011.100.111"));
18   System.out.println(convert("000.010.011.100.111"));
19   System.out.println(convert("010.011.100.111.000"));
20   System.out.println(convert("000.010.011.100.111.000"));
21 }
22 }
```

Listing 3: Code snippet C204957 with error AppendCharacterWithChar, mutation DeleteLine(8). The deleted line is shown in red.

SwapStatement) only accounted for 20 code snippets that could compile without performance issues. This preliminary observation warrants further investigation to rule out randomness.

While GIN mutations tend to change the semantics of the code, code semantics are at least mostly retained in at least one code snippet for six of the eight operations. Thus, while GIN mutations fix performance issues, such fixes tend to come at the expense of changes in code semantics. Notwithstanding that our analysis did not involve the use of test cases, where GIN may have repeated some of the mutations in search of larger patches for fixing PMD issues, the outcomes here demonstrate the *need for deeper contextual probing of automated program repair outcomes*, particularly given that test coverage can be biased by developers interest in only testing particular coding paths (and not all possible paths). Under such conditions, mutations may appear plausible, also resulting in

Table 2: Details on the 58 mutations. For the cases where our assessment is that the semantics are not retained in the mutation ("NO"), we list only representative or interesting cases.

edit type	count	semantics retained YES/MOSTLY/NO	A	B	C	D	E	F	G	H	I	notes on semantics retention (with the level of semantics preservation in bold)
DeleteStatement	25	-/1/24	9	4	4	4	2	1	1	2	-	**MOSTLY:** Listing 1: "the concatenation with a dash is deleted, which can cause confusion if subsequent processing depends on the dashes as a separator" **NO:** "core intelligence of the class is removed", "removes the statement to copy array elements from one array to another", "removes the entire body of the method", "removes an "if" block responsible for core functionality", "deletes the for loop", "deletes the inner for loop"
DeleteLine	11	-/3/8	4	2	1	1	1	1	-	-	1	**MOSTLY:** Listing 2: "removes one StringBuilder.append which improves the performance of the code, thus the first half of the string is lost... not a serious change in the semantics" **MOSTLY:** Listing 3: "removes code aimed at adding a period (".") to a string... could create confusion if the string was used later where the period was necessary" **MOSTLY:** Listing 4: "removes the statement responsible for output operation. While this would render the code semantically incorrect code, this is not a major change when compared to others" **NO:** "deleting the offending String append", "removes the code aimed at creating the string buffer", "removes the core of the class"
SwapStatement	9	-/-/9	6	3	-	-	-	-	1	-	-	**NO:** "swaps the core of the method, removing an entire "if" block.", "swaps the while loop", "swaps out the body of the method", "an entire for loop of 14 lines was replaced with one of the contained statements", "swaps loop body with another statment" (Listing 8)
ReplaceStatement	8	-/1/7	6	2	1	-	-	-	-	-	-	**MOSTLY:** Listing 5: "replaces loop body with one of the statements from the loop body" **NO:** "replaces a long "if" body with a return", "overwrites the offending string concatenation with a System.out.println"
SwapLine	2	-/-/2	1	1	-	-	-	-	-	-	-	**NO:** "the mutation causes the results to be written before the merge, and also changes the program operation by nesting multiple while loops that overwrites the arrays"
CopyLine	1	1/-/-	-	1	-	-	-	-	-	-	-	**YES:** Listing 6: we consider this a false positive as PMD should (in our opinion) still be reporting the error
CopyStatement	1	1/-/-	-	1	-	-	-	-	-	-	-	**YES:** Listing 6: we consider this a false positive as PMD should (in our opinion) still be reporting the error
ReplaceLine	1	-/1/-	-	1	-	-	-	-	-	-	-	**MOSTLY:** Listing 7: "the replacement results in the deletion of an increment, however, as it is still an array loop, we find it curious that PMD thinks the problem has gone away"

PMD errors: A=UseStringBufferForStringAppends, B=AvoidArrayLoops, C=AddEmptyString, D=AppendCharacterWithChar, E=InefficientStringBuffering, F=InefficientEmptyStringCheck, G=TooFewBranchesForASwitchStatement, H=AvoidInstantiatingObjectsInLoops, I=ConsecutiveLiteralAppends

no syntax-related faults, but the mutated code may not be useful (in terms of its intended purpose).

Findings here also have implications for sampling and code transformations during mutation. For instance, we observe that of 11 code snippets that were mutated by DeleteLine, three mostly retained their semantics. While this edit operation is arguably simple, our evidence suggests that *removing offending code can be an effective program repair strategy*. If we are able to learn when to use this strategy (e.g., through the use of some pattern recognition approach), thereby removing the randomness of the mutations, we may be able to effectively improve faulty code online (the goal of Licorish and Wagner [5]).

Of note is that PMD parsing seems at times to be confused by GIN's mutations. For instance, Listing 6 shows that the CopyLine mutation result in the same code, where we anticipated that PMD should still be reporting the performance-related issue AvoidArrayLoops. However, the new code passes without any PMD warning. This points to scope for improving PMD parsing in the way the abstract syntax tree (AST) is processed.

```
1 public class C264051{
2 public static int gcd(int a, int b) {
3     if (b == 0) {
4         return a;
5     } else {
6         return gcd(b, a % b);
7     }
8 }
9
10 public static int pairwisePrimes(int k) {
11   int numWays = 0;
12   for (int a = 1; a < k; a++) {
13     for (int b = a + 1; b < k; b++) {
14       for (int c = b + 1; c < k; c++) {
15         if ((a + b + c == k) && gcd(a, b) == 1 && gcd(a,
               c) == 1 && gcd(b, c) == 1) {
16           System.out.println("" + a + "+" + b + "+" + c);
17           numWays++;
18         }
19       }
20     }
21   }
22   return numWays;
23 }
24 }
```

Listing 4: Code snippet C264051 with error AddEmptyString, mutation DeleteLine(16). The deleted line is shown in red.

```
1 public class C83902{
2     public int[] getSubArray(int[] array, int index, int
          size) {
3         int[] subArray = new int[size];
4         int subArrayIndex = 0;
5         for (int i = index; i < index + size; i++) {
6             subArray[subArrayIndex] = array[i];
7             subArrayIndex++;
8         }
9         return subArray;
10    }
11 }
```

Listing 5: Code snippet C83902 with error AvoidArrayLoops, mutation ReplaceStatement(55,54). The deleted code is shown in red.

```
1 public class C315640{
2 private static Integer[] toIntegerArray(int[] array){
3     Integer[] finalArray = new Integer[array.length];
4     for (int i=0; i<array.length; i++){
5         finalArray[i] = array[i];
6         finalArray[i] = array[i];
7     }
8     return finalArray;
9 }
10 }
```

Listing 6: Code snippet C315640 with error AvoidArrayLoops, mutations CopyLine(5,5) and CopyStatement(47,46). The introduced line is shown in blue.

```
1 public class C83902{
2 public int[] getSubArray(int[] array, int index, int
      size) {
3   int[] subArray = new int[size];
4   int subArrayIndex = 0;
5   for (int i = index; i < index + size; i++) {
6     subArray[subArrayIndex] = array[i];
7     subArrayIndex++;
8     subArray[subArrayIndex] = array[i];
9   }
10  return subArray;
11 }
12 }
```

Listing 7: Code snippet C83902 with error AvoidArrayLoops, mutation ReplaceLine(6,7). The removed code is shown in red and the introduced code is shown in blue.

```
  // original
1 public class C330977{
2     public void printStrings(String a, int b) {
3         String printString = "";
4         for (int i = 0; i<b; i++) {
5             printString = printString+" "+a;
6         }
7         System.out.println(printString);
8     }
9 }
  // after the mutation
1 public class C330977 {
2     public void printStrings(String a, int b) {
3         String printString = "";
4         for (int i = 0; i < b; i++) System.out.println(
              printString);
5         {
6             printString = printString + " " + a;
7         }
8     }
9 }
```

Listing 8: Code snippet C330977 with error UseStringBufferForStringAppends, mutation SwapStatement(36,48). The removed code is shown in red and the introduced code is shown in blue.

Licorish and Wagner [5]'s earlier work singled out the need to mitigate potential false positives when static analysis and automated code improvement techniques are used. Neglecting false positives could particularly weaken the pipeline as mutations should not be performed when there is no need for code improvement. On the contrary, the opposite may result when there are false negative, alluding to the need to enhance the fitness of PMD, and static analysis more generally. In fact, our outcomes here cut to the core of other work that has reported widespread faults in code publicly available online [2, 3, 7], alluding to the need for focused scrutiny of static analysis routines. We thus reiterate here that the *mitigation of false positives and negatives as well as parsing improvements are essential to validate static analysis techniques.*

Our code and data publicly available at https://github.com/markuswagnergithub/combining_sa_and_gi.

ACKNOWLEDGMENTS

Sherlock acknowledges support by the New Zealand SfTI project UOOX2001, and thank Dr Caitlin Owen and Associate Professor Tony Savarimuthu for their earlier involvement and advice on the study of Stack Overflow code quality. Markus acknowledges support by the ARC projects DP200102364 and DP210102670, and by gifts from Facebook and Google.

REFERENCES

[1] 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 *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 985–993. https://doi.org/10.1145/3321707.3321841

[2] Georgios Digkas, Nikolaos Nikolaidis, Apostolos Ampatzoglou, and Alexander Chatzigeorgiou. 2019. Reusing code from stackoverflow: the effect on technical debt. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 87–91.

[3] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 121–136. https://doi.org/10.1109/SP.2017.31

[4] Claire Le Goues, Westley Weimer, and Stephanie Forrest. 2012. Representations and Operators for Improving Evolutionary Software Repair. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 959–966. https://doi.org/10.1145/2330163.2330296

[5] Sherlock A. Licorish and Markus Wagner. 2022. On the Utility of Marrying GIN and PMD for Improving Stack Overflow Code Snippets. *CoRR* abs/2202.01490 (2022). arXiv:2202.01490 https://arxiv.org/abs/2202.01490

[6] Adriaan Lotter, Sherlock A. Licorish, Bastin Tony Roy Savarimuthu, and Sarah Meldrum. 2018. Code Reuse in Stack Overflow and Popular Open Source Java Projects. In *25th Australasian Software Engineering Conference (ASWEC)*. 141–150. https://doi.org/10.1109/ASWEC.2018.00027

[7] Sarah Meldrum, Sherlock A. Licorish, Caitlin A. Owen, and Bastin Tony Roy Savarimuthu. 2020. Understanding stack overflow code quality: A recommendation of caution. *Science of Computer Programming* 199 (2020), 102516. https://doi.org/10.1016/j.scico.2020.102516

[8] PMD. 2021. PMD 6.41.0. https://pmd.github.io/latest/pmd_rules_java.html [Online; accessed on 22 January 2022].

[9] Brittany Reid, Christoph Treude, and Markus Wagner. 2020. *Optimising the Fit of Stack Overflow Code Snippets into Existing Code*. ACM, 1945–1953. https://doi.org/10.1145/3377929.3398087