2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)

Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair

Moumita Asad
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
bsse0731@iit.du.ac.bd

Kishan Kumar Ganguly
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
kkganguly@iit.du.ac.bd

Kazi Sakib
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
sakib@iit.du.ac.bd

***Abstract*—Patch prioritization means sorting candidate patches based on probability of correctness. It helps to minimize the bug fixing time and maximize the precision of an automated program repairing technique. Approaches in the literature use either syntactic or semantic similarity between faulty code and fixing element to prioritize patches. Unlike others, this paper aims at analyzing the impact of combining syntactic and semantic similarities on patch prioritization. As a pilot study, it uses genealogical and variable similarity to measure semantic similarity, and normalized longest common subsequence to capture syntactic similarity. For evaluating the approach, 22 replacement mutation bugs from IntroClassJava benchmark were used. The approach repairs all the 22 bugs and achieves a precision of 100%.**

***Index Terms*—patch prioritization, semantic similarity, syntactic similarity, automated program repair**

I. Introduction

Patch is the modifications applied to a program for fixing a bug. Automated program repair finds the correct patch based on a specification, e.g., test cases [1]. It works in three steps namely fault localization, patch generation and patch validation [2]. Fault localization identifies the faulty code where the bug resides. Patch generation modifies the faulty code to fix the bug. Patch validation checks whether the bug has been fixed or not. Since the solution space is infinite, numerous patches are generated [3]. In addition, an incorrect patch can be plausible - patch that passes all the test cases. These two problems hinder the ability to fix bugs at an affordable cost [3]. Therefore, patch prioritization is required for validating potentially correct patches before incorrect plausible ones.

Sorting candidate patches, based on its probability of correctness, is called patch prioritization [4]. It can help to minimize the bug fixing time and maximize the precision of a repairing technique. To prioritize patches, some information need to be incorporated, e.g., similarity between faulty code and fixing element (code used to fix the bug) or patterns derived from existing patches [3]. The information should limit the number of patches to be validated since patch validation is a time-consuming task [5]. Furthermore, the information should be able to improve the precision of a program repairing technique by executing the correct patch earlier.

Existing patch prioritization approaches [3], [5], [6], [7] use either syntactic or semantic similarity between faulty code and fixing element. To capture syntactic similarity, Elixir uses contextual and bug report similarities [5]. ssFix uses TF-IDF model to calculate syntax-similarity score [6]. To find top fixing elements syntactically similar to the faulty code, SimFix uses 3 metrics - structure, variable name and method name similarities [3]. To measure semantic similarity, CapGen uses 3 models based on genealogical structures (e.g., ancestors of an Abstract Syntax Tree (AST) node), accessed variables and semantic dependencies [7]. However, none of these approaches analyze the impact of incorporating strengths of both similarities to prioritize patches.

This paper presents a pilot study on patch prioritization. The technique takes a program, a set of test cases as input and generates a program passing all the test cases as output. The test cases must contain both positive test cases that show the expected functionality of the code and negative test cases that demonstrate the bug [8]. At first, the faulty AST node of type *Expression* is identified using GZoltar tool [9]. The technique works at expression level since finer granularity increases the probability of including the correct solution in the search space [7]. Next, patches are generated by replacing the faulty node with the fixing element. Following other repairing techniques, such as, [7], [8], the fixing elements are collected from the source file where the bug resides. To validate potentially correct patch earlier, generated patches are prioritized based on syntactic and semantic similarities. Finally, the correctness of a patch is validated by executing test cases.

To evaluate this approach, 22 out of 297 bugs from IntroClassJava [10] benchmark were used. These bugs were repaired by CapGen using replacement mutation (replacing the faulty node with fixing element) [7]. The formulated approach can repair all the 22 bugs without generating any plausible patch before the first correct solution. Thus, it achieves a precision of 100% in this context. It ranks the first correct patch higher compared to techniques using syntactic or semantic similarity in most of the cases.

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00050

II. Related Work

Existing patch prioritization techniques can be classified into two categories based on using the type of similarity between faulty code and fixing element. The first category [3], [5], [6] uses syntactic similarity which focuses on textual similarity. For example, similarity in variable names. The second category [7] uses semantic similarity between faulty code and fixing element. It focuses on code meaning, e.g., data type of variables [11].

Elixir, one of the first such approach, introduces 8 templates for generating candidate patches [5]. For example, checking array range and collection size. It uses 4 features including contextual and bug report similarities to prioritize patches. Contextual similarity measures the syntactic similarity between fixing element and surrounding code of the faulty location. Bug report similarity calculates the syntactic similarity between fixing element and bug report. For assigning different weights to these similarities, logistic regression model is used. The approach validates only the top 50 patches generated from each template.

ssFix searches for code that is syntactically similar to the faulty code, from a code database containing the faulty program and other projects [6]. The approach prioritizes code based on its syntax-similarity score calculated using TF-IDF.

SimFix uses 3 metrics - structure, variable name and method name similarities to capture syntactic similarity between faulty code and fixing element [3]. Structure similarity extracts a list of features (e.g., number of *If* statements) related to AST nodes. Variable name similarity tokenizes variable names (e.g., splitting studentID into student and ID) for calculating similarity using Dice coefficient [12]. Method name similarity follows the same process as variable name similarity. To generate patches, the approach selects top 100 fixing elements based on similarity score. To further limit the search space, only fixing elements found frequently in existing human-written patches are considered. However, these three approaches miss important information by not considering the semantic similarity between faulty code and fixing element. Hence, these yield low precisions 63.41%, 33.33% and 60.7% respectively [3].

To generate patches, CapGen defines 30 mutation operators, e.g., insert *Expression* statement under *If* statement [7]. These operators are derived from a dataset of 3000 real bugs from 700 open-source projects [13]. The approach uses 3 models based on genealogical structures, accessed variables and semantic dependencies to capture context similarities at AST node level. These models mainly focus on semantic similarities between faulty code and fixing element to prioritize patches. The precision of this approach is higher (84.00%). However, the technique could further be improved by considering syntactic similarity to reduce the search space [14].

The above discussion indicates that in spite of having limitations, syntactic or semantic similarities are effective in patch prioritization. However, the impact of combining the strengths of both similarities to prioritize patches has not been explored till now.

III. Methodology

It is expected that combining the strengths of syntactic and semantic similarities will improve patch prioritization. Fig. 1 shows two sample bug fixes from IntroClassJava (bug id = digits_d5059e2b_000) [10] and Defects4J (bug id = Math 70) [15] respectively. Here, the lines of code started with + and - indicate the added or deleted line respectively. It can be seen that inserted line (fixing element) and the deleted line (faulty code) are syntactically and semantically similar in both cases.

```
// faulty code
63: - digit.value = Math.abs (num.value) % 10;
// fixing element
63: + digit.value = Math.abs (((num.value) – (digit.value))/10) % 10;
```

(a) digits_d5059e2b_000 from IntroClassJava

```
// faulty code
72: - return solve (min, max);
// fixing element
72: + return solve (f, min, max);
```

(b) Math 70 from Defects4J

Fig. 1: Sample Bug Fixes

This paper devises a patch prioritization algorithm combining syntactic and semantic similarities for automated program repair, as shown in Fig. 2. It works in four steps namely fault localization, patch generation, patch prioritization and patch validation. The details of these steps are given below:

1) **Fault Localization:** This step identifies the faulty AST node of type *Expression*. It outputs line-number wise suspicious values (value indicating a line's probability of being faulty) of a program. These values are next mapped to AST nodes. If a node spans across multiple lines, it is assigned the average of the suspicious values of those lines.
2) **Patch Generation:** This step modifies the source code to generate patches. After identifying the faulty nodes, fixing elements are collected. Following existing technique [7], nodes from the faulty source file that have a category of *Expression* are used as fixing elements. The technique performs only replacement mutation since the faulty code is more likely to be syntactically and semantically similar to the fixing element for this kind of bugs [14]. To generate patches, it uses seven replacement mutation operators that works on *Expression* node and the corresponding probabilities proposed by Wen et al. [7]. To avoid generating semantically ill-formed program variants during mutation, the scope of the variables used by the fixing elements are checked.
3) **Patch Prioritization:** Having infinite solution space, the patch generation step produces numerous patches. To validate potentially correct patches earlier, the generated patches are prioritized, as shown in Fig. 2. Both syntactic and semantic similarities between faulty code and fixing element are used to prioritize patches. As an initial study,

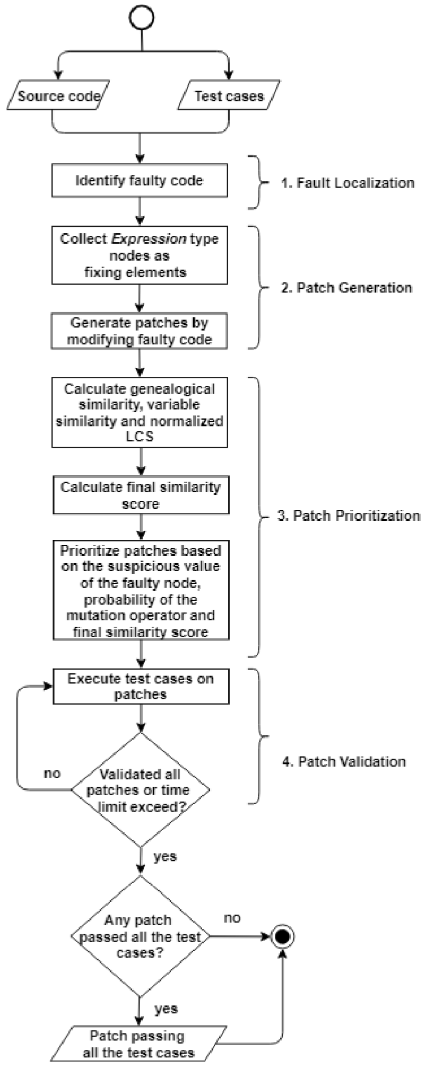

Fig. 2: Overview of the Technique

genealogical and variable similarity between faulty code and fixing element are used to measure semantic similarity [7]. On the other hand, a widely-used metric normalized Longest Common Subsequence (LCS) [14] is used to capture syntactic similarity [16].

- **Genealogical Similarity:** Genealogical structure indicates a node is frequently used under and together with which types of code elements [7]. To extract the genealogy contexts of a node, it's ancestor, as well as, sibling nodes are inspected. The ancestors of a node are traversed until a method declaration is found. For sibling nodes, nodes having a type *Expressions* or *Statements* within the same block of the specified node are extracted. Next, the type of each node is checked and the frequency of different types of nodes (e.g., number of *For* statements) are stored. Nodes of type *Block* are not considered since these provide insignificant context information [7]. The same process is repeated for the faulty node and the fixing element. Lastly, the genealogical similarity is measured using (1).

$$gs(\phi_{fn}, \phi_{fe}) = \frac{\sum_{t\in K} min(\phi_{fn}(t), \phi_{fe}(t))}{\sum_{t\in K} \phi_{fn}(t)} \quad (1)$$

where, (ϕ_{fn}) and (ϕ_{fe}) denote frequencies of different node types for faulty node and fixing element respectively. K represents a set of all distinct AST node types captured by ϕ_{fn}.

- **Variable Similarity:** Variables (local variables and class attributes) accessed by a node provides useful information as these are the primary components of a code element [7]. To measure variable similarity, two lists containing names and types of variables used by the faulty node (θ_{fn}) and the fixing element (θ_{fe}) are generated. Next, variable similarity is calculated using (2).

$$vs(\theta_{fn}, \theta_{fe}) = |\theta_{fe}| * \frac{|\theta_{fn} \cap \theta_{fe}|}{|\theta_{fn} \cup \theta_{fe}|} \quad (2)$$

Here, multiplication by $|\theta_{fe}|$ is done so that more priority can be given to complex elements (fixing elements with more variables) to avoid generating plausible patches [7]. Two variables are considered same if their names and types are exact match.

- **Normalized LCS:** LCS finds the common subsequence of maximum length by working at character-level [14]. The technique computes normalized LCS between faulty code (fn) and fixing element (fe) at AST node level using (3).

$$nl(\theta_{fn}, \theta_{fe}) = \frac{LCS(fn, fe)}{max(fn, fe)} \quad (3)$$

After calculating similarity metrics, the final similarity score is measured using (4):

$$simi(fn, fe) = \alpha * (gs * vs) + \beta * nl \quad (4)$$

where, α and β are weights assigned to semantic and syntactic similarities respectively. In this paper, the value of α and β are set based on experimentation (Section IV D). Alternatively, machine learning models (e.g., logistic regression [5]) can be used to calculate α and β. Lastly, rankings are calculated based on (5), which are used to prioritize patches.

$$rank = fl(fn) * freq(m) * simi(fn, fe) \quad (5)$$

where, $fl(fn)$ is the suspicious value of the faulty node and $freq(m)$ is the probability of the mutation operator.

4) **Patch Validation:** This step checks the correctness of a generated patch. Patches that are both syntactically and semantically similar to the faulty code are validated. To validate a patch, at first, the negative test cases are executed. If it passes, the positive test cases are executed. Patch validation continues until all the patches are checked or the predefined time-limit exceeds.

IV. EXPERIMENT

This section presents implementation details, evaluation criteria, experiment setting and result analysis of the work.

A. Implementation

The devised technique is implemented in Java. It uses Eclipse JDT parser for manupulating AST. To localize fault, GZoltar tool (version 1.6.1) [9] is used with Ochiai algorithm [17]. GZoltar is widely used for localizing fault in automated program repair [6], [7]. It takes the class files of the source code and test cases as input. It provides line-number wise suspicious values of a program as output.

B. Evaluation

As an initial study, the approach was evaluated using 22 bugs from the IntroClassJava benchmark [10], fixed by CapGen applying replacement mutation [7]. IntroClassJava contains 297 bugs from 6 projects, as shown in Table I.

TABLE I: Projects of IntroClassJava Benchmark

Name	Description	Number of Faulty Programs
checksum	calculates checksum of a string	11
digits	reverses of a number	75
grade	calculates grade from percentage	89
median	calculates median of 3 numbers	57
smallest	finds minimum of 4 numbers	52
syllables	counts number of vowels in a string	13

For evaluation, the following metrics are inspected:

- **Number of bugs fixed:** If an approach fixes more bugs, it is considered more effective [13].
- **Precision:** If an approach can rank correct solutions prior to incorrect plausible one, it achieves higher precision. If precision is high, developers do not have to manually analyze the solutions generated by the technique [7].
- **Rank of the first correct solution:** The approach that ranks the first correct solution higher, the better [13].

C. Experiment Settings

In IntroClassJava, the timeout parameter for all the test cases was set to 1000 milliseconds. However, the time-limit exceeded for some of the generated patches. Hence, the timeout parameter was set to 2000 milliseconds for this experiement. It was run on a Ubuntu server with Intel Xeon E5-2690 Core CPU @3.0GHz and 64GB physical memory. Following existing techniques, the time limit for each bug was set to 90 minutes [5], [7]. To check the correctness of a generated patch, publicly available CapGen results was used [1].

D. Result Analysis

The approach repairs all the 22 bugs. For the repaired bugs, no plausible patch is generated before the first correct solution. Thus, the approach achieves a precision of 100%. Table II shows the bugs for which the devised approach performs at least as good result as semantic similarity and better than syntactic similarity. Column one shows the bug IDs. Column two and three show the result obtained using only semantic similarity and syntactic similarity respectively. It can be viewed that semantic similarity can rank the first correct patch higher and minimize the number of generated patches compared to syntactic similarity. The reason is semantic similarity focuses on code meaning rather than textual similarity. Column four shows the result of combining both similarities. The approach was run using various weights of α and β. The results are publicly available at: https://github.com/mou23/Impact-Analysis-of-Syntactic-and-Semantic-Similarities-on-Patch-Prioritization. Due to lack of space, 5 combinations of α and β are shown. For all weights of α and β, the combination of semantic and syntactic similarities achieves lower median rank than semantic or syntactic similarity. It indicates that the combination of both similarities is better in ranking the correct patch higher than semantic or syntactic similarity. For example, the first correct solution of bug grade_1b31fa5c_003 was ranked 166 by the combination of both similarities (α = 0.5 and β = 0.5), whereas it was ranked 177 and 252 by semantic and syntactic similarity respectively. This is because the combination of two similarities incorporates information from both textual similarity and code meaning.

Among the 22 bugs, there are 6 cases when the combination of both similarities results in declined performance than semantic similarity (Table III). For these bugs, semantic similarity replaced a large conditional expression with a smaller one. For example, replacing (a.value < b.value) && (a.value < c.value) && (a.value < d.value) with (a.value < d.value). Normalized LCS does not perform well in such cases since the character level difference is high.

V. CONCLUSION

In this paper, a patch prioritization algorithm combining syntactic and semantic similarity metrics for automated program repair is formulated. As an initial study, primitive metric normalized LCS is used to capture syntactic similarity. Genealogical and variable similarities are used to measure semantic similarity. The approach has been validated using 22 replacement mutation bugs from IntroClassJava benchmark [10]. It achieves a precision of 100% in solving those bugs. Results further show that the technique is better in ranking the correct patch earlier than semantic or syntactic similarity in most cases. Normalized LCS does not perform well when the character level difference between faulty code and fixing element is high. In future, more appropriate metrics for measuring syntactic similarity in the context of automated program repair will be identified. In addition, a mathematical model will be derived to generalize the impact of each metric.

VI. ACKNOWLEDGEMENT

This research is supported by Bangladesh Research and Education Network (BdREN).

[1] https://github.com/justinwm/CapGen

TABLE II: Improved Performance of the Technique on IntroClassJava Benchmark

Bug ID	Only Semantic Similarity		Only Syntactic Similarity		Combination of Syntactic and Semantic Similarity					
	Total Patches	Rank	Total Patches	Rank	Total Patches	Rank ($\alpha = 0.9$ and $\beta = 0.1$)	Rank ($\alpha = 0.7$ and $\beta = 0.3$)	Rank ($\alpha = 0.5$ and $\beta = 0.5$)	Rank ($\alpha = 0.3$ and $\beta = 0.7$)	Rank ($\alpha = 0.1$ and $\beta = 0.9$)
digits_6e464f2b_003	189	115	491	269	189	109	102	102	112	99
digits_c9d718f3_001	157	141	502	364	157	141	141	141	139	127
digits_d5059e2b_000	155	122	388	227	155	118	115	114	116	107
grade_1b31fa5c_003	246	177	614	252	246	170	170	166	152	113
median_0cdfa335_003	273	202	557	312	273	205	191	191	176	140
median_89b1a701_003	154	112	355	138	154	112	104	104	73	49
median_89b1a701_007	190	148	391	138	190	148	136	136	88	51
median_89b1a701_010	629	279	952	179	629	279	247	173	110	92
median_fe9d5fb9_000	230	124	625	3	230	111	111	111	105	101
median_fe9d5fb9_002	230	124	625	3	230	111	101	100	100	99
smallest_15cb07a7_007	299	198	688	485	299	198	198	196	215	236
smallest_36d8008b_003	470	236	841	587	470	236	254	291	360	399
smallest_48b82975_001	470	236	838	584	470	236	254	291	360	399
smallest_68eb0bb0_000	383	203	727	504	383	203	221	258	310	335
smallest_97f6b152_003	374	225	830	612	374	225	225	225	279	292
smallest_818f8cf4_003	561	406	1085	27	561	402	298	303	297	248
Median Rank		**187.5**		**260.5**		**184**	**180.5**	**169.5**	**145.5**	**120**

[1] **Total Patches** denotes the total number of patches generated. **Rank** indicates rank of the first correct patch.

TABLE III: Declined Performance of the Technique on IntroClassJava Benchmark

Bug ID	Only Semantic Similarity		Only Syntactic Similarity		Combination of Syntactic and Semantic Similarity					
	Total Patches	Rank	Total Patches	Rank	Total Patches	Rank ($\alpha = 0.9$ and $\beta = 0.1$)	Rank ($\alpha = 0.7$ and $\beta = 0.3$)	Rank ($\alpha = 0.5$ and $\beta = 0.5$)	Rank ($\alpha = 0.3$ and $\beta = 0.7$)	Rank ($\alpha = 0.1$ and $\beta = 0.9$)
grade_b1924d63_001	385	161	714	333	385	162	162	181	195	183
grade_b1924d63_003	385	161	700	362	385	162	162	181	195	183
smallest_3b2376ab_008	327	169	759	555	327	177	194	214	284	267
smallest_dedc2a7c_000	345	127	582	447	345	134	173	209	278	282
smallest_ea67b841_003	424	156	748	567	424	163	202	238	320	338
smallest_f8d57dea_000	327	169	686	488	327	177	194	214	284	267
Median Rank		**161**		**467.5**		**162.5**	**183.5**	**211.5**	**281**	**267**

References

[1] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[2] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)-Volume 1*, pp. 471–482, IEEE Press, 2015.

[3] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 298–309, ACM, 2018.

[4] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 789–799, ACM, 2018.

[5] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 648–659, IEEE, 2017.

[6] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 660–670, IEEE, 2017.

[7] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 1–11, ACM, 2018.

[8] 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, 2011.

[9] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 378–381, ACM, 2012.

[10] T. Durieux and M. Monperrus, *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*. PhD thesis, Universite Lille 1, 2016.

[11] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 532–542, ACM, 2013.

[12] V. Thada and V. Jaglan, "Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm," *International Journal of Innovations in Engineering and Technology*, vol. 2, no. 4, pp. 202–205, 2013.

[13] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 213–224, IEEE, 2016.

[14] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," *Computing Research Repository (CoRR)*, vol. abs/1811.05703, 2018.

[15] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 437–440, ACM, 2014.

[16] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2464–2519, 2018.

[17] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, 2017.