ESEC/FSE: U: Accelerating Redundancy-Based Program Repair via Code Representation Learning and Adaptive Patch Filtering

Chen Yang*
College of Intelligence and Computing, Tianjin University
Tianjin, China

Jiajun Jiang
College of Intelligence and Computing, Tianjin University
Tianjin, China

Junjie Chen
College of Intelligence and Computing, Tianjin University
Tianjin, China

ABSTRACT

Automated program repair (APR) has attracted extensive attention and many APR techniques have been proposed recently, in which redundancy-based techniques have achieved great success. However, they still suffer from the efficiency issue. One key problem is how to advance the generation and validation of correct patches. Traditional redundancy-based approaches often use simple methods, such as statistical-based methods, to measure code similarity. This could result in the inaccuracy of measuring code similarity, which may produce meaningless patches that hinder the generation and validation of correct patches. Recently, state-of-the-art studies demonstrate that neural models can better represent source code. Therefore, to solve this issue, we propose a novel method AccPR, which leverages neural network for code representation learning to measure code similarity accurately and employs adaptive patch filtering to accelerate redundancy-based APR. We have implemented a prototype of AccPR and integrated it with a state-of-the-art APR tool, SimFix. We conducted a preliminary study on the benchmark, Defects4J, where the average improvement of repairing efficiency is 47.85%, indicating AccPR is promising.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

representation learning, patch filtering, automated program repair

1 INTRODUCTION

Automated program repair (APR) has attracted extensive attention in recent years [6, 26]. Redundancy-based APR techniques [1, 24] (such as GenProg [21], CapGen [32], SCRepair [10], CRSearcher [29], ssFix [34], SimFix [13] and Refactory [9]) are one of the most important categories in this field and have achieved promising results on the number of real-world faults fixed. [12, 13, 34]. The basic idea of most redundancy-based methods is that by reusing existing code there is a hope that the correct patch will be generated. Specifically, they first search for a set of code snippets similar to a suspicious faulty code from a code base as references, and then generate a bunch of candidate patches to be validated one by one against a test suite according to the "generation-validation" model. Although redundancy-based APR can successfully fix a number of faults as demonstrated in existing studies [13, 23, 26, 34], they still suffer from the efficiency issue [22], meaning that they require a lot of time to repair a bug, as patch validation is a time-consuming process and they often waste much time validating incorrect patches before the correct one. For example, a state-of-the-art redundancy-based APR technique (SimFix [13]) sets a time budget of up to 5 hours for each fault [13]. APR efficiency, which represents the ability to accelerate debugging process and reduce the time-to-fix delays, has important influence on promoting these techniques into practice and significantly affects debugging performance. Therefore, it is important to accelerate redundancy-based APR techniques.

Through deeply investigating redundancy-based APR techniques, there are two major problems affecting their efficiency [20, 22]. First, the searched similar code snippets are not accurate for generating correct patches. Specifically, current redundancy-based APR techniques mainly depend on simple frequency or syntactic features (e.g., AST) to measure code similarity, which cannot capture semantic information and thus the measured similarity is not accurate. This leads to generating many incorrect patches and also wasting much time to validate them [28, 30, 35]. We call it *inaccurate similarity problem*. Second, the order of validating generated patches used currently leads to wasting much time to validate many similar but incorrect patches before the correct one. Due to the inaccurate problem mentioned above, existing APR techniques may generate many incorrect patches before the correct one. However, some patches produce quite similar or even the same modifications on the code snippet, so a lot of time is squandered validating these similar but incorrect patches. We call it *(patch) order problem*.

To boost the efficiency of redundancy-based APR techniques, we propose a novel method, called **AccPR**, to overcome the above problems. Regarding the inaccurate similarity problem, AccPR incorporates representation learning to extract deep semantic information from code snippets to improve the accuracy of similarity measurements, inspired by Dantas A et al.[2]. Here, we adopt ASTNN [36], a state-of-the-art code representation learning method, in AccPR. Regarding the order problem, although incorporating code representation learning could relieve this problem to some degree, by improving the accuracy of measuring code similarity we observed that there is a major obstacle for a better patch-validation order. Specifically, for a candidate code snippet, a lot of patches can be generated, including similar ones. If a patch has been validated to be incorrect, patches similar to it are quite likely to be incorrect as well, which has never been considered by existing techniques. Therefore, we design an adaptive patch filtering strategy to relieve the order problem. Specifically, according to the validation feedback, AccPR adaptively filters out those patches that are highly similar to confirmed incorrect ones to save time. In addition, regarding the set of generated patches for a suspicious faulty code snippet, AccPR ranks patches as the descending order of their similarity with the faulty code, since simple patches are more likely to be correct [19, 25].

*Jiajun Jiang and Junjie Chen are the advisors.

We have implemented a prototype of AccPR and integrated it with a state-of-the-art redundancy-based APR technique, SimFix [13]. Then, we conducted a preliminary study to investigate its performance on the benchmark, Defects4J [14]. Experimental results show that AccPR reduces the average repair time of SimFix from 11.34 to 5.91 minutes, an improvement of about 47.85% on the repairing efficiency, demonstrating its effectiveness.

To sum up, the major contributions of this work are as follows:

- We proposed a novel method to accelerate redundancy-based APR via incorporating representation learning to improve the similarity measurement and an adaptive filtering strategy to save validation time;
- We have implemented a prototype of AccPR, which has been integrated with a state-of-the-art redundancy-based APR tool, SimFix.
- We conducted a preliminary study on a benchmark Defects4J. The experimental results demonstrate the proposed method is indeed promising.

The remainder of this paper is structured as follows. Section 2 presents our approach. Section 3 describes the applications and effectiveness of AccPR. Related work, conclusion about our approach and future work are presented in Section 4 and Section 5 respectively.

2 APPROACH AND UNIQUENESS

AccPR follows the workflow of redundancy-based APR techniques. It is designed to solve the inaccurate similarity and patch order problems by optimizing the similarity measurement and applying an adaptive patch filtering strategy. Figure 1 presents the overview of AccRP, which mainly consists of three steps. First, when providing a candidate faulty code snippet, AccRP leverages code embedding techniques to aid the search of similar code, and then it employs a predefined strategy (e.g., SimFix) to generate patches according to the similar code. Finally, AccRP validates the correctness of candidate patches using the equipped test cases and filters those patches that are less likely to be correct by measuring patch similarity.

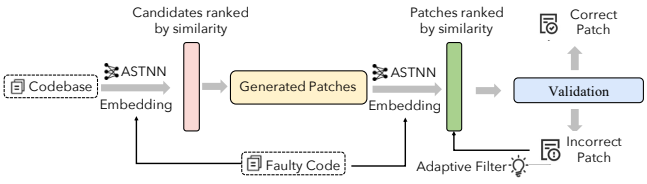

Figure 1: Overview of AccPR

Specifically, to solve the *inaccurate similarity problem*, AccPR leverages code representation learning to extract deep semantic information from code snippets to improve the accuracy of similarity measurements. That is, it employs the state-of-the-art code embedding technique, ASTNN [36], which has been evaluated to be effective. Regarding the *patch order problem*, there is a major obstacle for a better patch-validation order. Specifically, if a patch has been validated to be incorrect, patches similar to it are quite likely to be incorrect as well, which has never been considered by existing techniques. Therefore, AccPR designs an adaptive patch filtering strategy to relieve the order problem, which is the first time as far as we are aware.

In the following, we first introduce the similarity measurement using code representation in AccPR (Section 2.1), based on which we present our adaptive patch filtering process in Section 2.2.

2.1 Similarity Measurement

2.1.1 Code Representation. We incorporate ASTNN [36], a code representation model that can effectively extract semantic information of code snippets in AccPR. ASTNN is a novel AST-based neural network, which encodes the statement trees in an AST to vectors by capturing the lexical and syntactical knowledge. Based on the sequence of statement vectors, it then employs a bidirectional RNN to leverage the naturalness of statements and produce the vector representation of a code snippet. It is suitable to our scenario as the learned vectors can be used to measure code similarity. And since it is a general-purpose model based on ASTs, AccPR will not be limited to any specific programming languages or tools.

2.1.2 Similarity Measurement. We apply learned embeddings to capture the similarity between code snippets. Given two code snippets m and n, we first leverage ASTNN to embed them into vectors, and then employ *1-Norm* to compute their similarity, which is defined as:

$$Simi(m, n) = ||astnn(m) - astnn(n)||$$

In this formula $astnn(*)$ refers to the embedding result of the given code snippet and $Simi(*, *)$ refers to the similarity score of the two given code snippets.

2.1.3 Candidate Code Snippets Ranking. Given a suspicious faulty code snippet n, AccPR identifies a set of similar code snippets $\mathbb{M}$ as candidates for patch generation. That is, for each $m \in \mathbb{M}$, we compute its similarity with n by $Simi(n, m)$. Then, AccPR ranks all candidate code snippets as the descending order of these similarity results, since the code snippets having higher similarities with the faulty code are more likely to generate the correct patch [13, 18, 25]. Then we extract modifications from the candidate code snippets and the suspicious faulty code to generate candidate patches.

2.2 Adaptive Patch Filtering

Given a set of similar code snippets, multiple patches $\mathbb{P}$ shall be generated for a faulty snippet n. The APR tool will then validate these candidate patches one by one, which is a rather time-consuming process and may waste much time validating many similar but incorrect patches before the correct one. To save this time overhead, AccPR performs a multi-level patch prioritization and filtering strategy according to the following rules to make the correct patch be validated as early as possible.

R1 (**Similarity**): Patches that have higher similarities with the faulty code (i.e., $Simi(p, n)$ for each $p \in \mathbb{P}$) are ranked higher since the patches with fewer, simpler modifications are more likely to be correct patches.

R2 (**Adaptive Filtering**): Patches that are similar to a known incorrect patch under a given threshold will be filtered since they are highly likely to be incorrect as well.

R1 is inspired by syntactic distance proposed by Mechtaev et al. [19, 25]: the patches with fewer, simpler modifications are ranked higher.

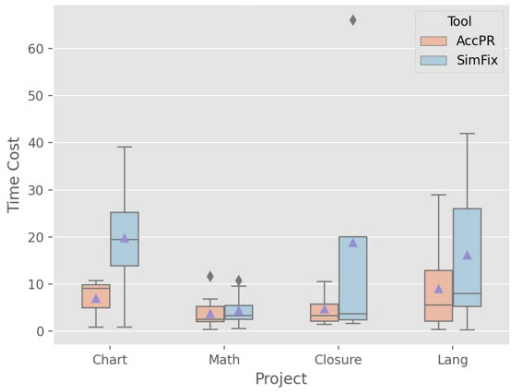

Figure 2: Distribution of time cost per each project

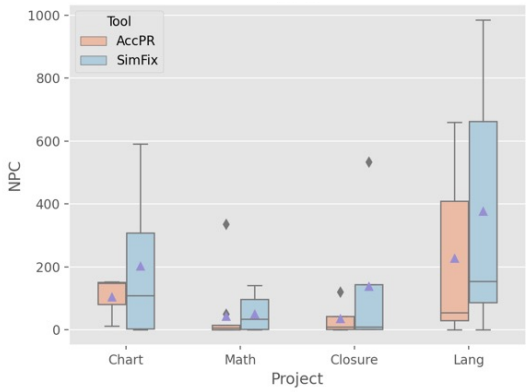

Figure 3: Distribution of NPC per each project

While R2 targets to filter out potential incorrect patches as early as possible with similar inconsequential modifications. AccPR validates patches according to the ordered list of candidate patches, which is adaptively updated by applying the above two strategies online.

3 RESULTS AND CONTRIBUTIONS

To investigate the performance of AccPR, we implemented a prototype of AccPR and integrated it with a state-of-the-art redundancy-based APR tool, i.e., SimFix, and conducted our experiments.

3.1 Experiment Setup

We evaluated AccPR on Defects4J [14](v1.2),which is a commonly-used benchmark for automatic program repair research. In the experiment, we will compare the results of AccRP with the state-of-the-art redundancy-based program repair tool, SimFix [13]. The experiment was conducted on a laptop with Ubuntu 7.5.0 and Oracle JDK 1.8.

3.2 Research Questions and Results

In this preliminary study, we only focus on faults that have been fixed by previous APR tools, and show the results of faults repaired by either SimFix or AccPR. We employ two metrics to evaluate AccPR's effectiveness:

1) *Time Cost*: time for online patch generation, patch embedding and validation (i.e., fault localization and donor code searching are excluded);
2) *NPC*: the **N**umber of **P**atch **C**andidates validated before the correct patch[22].

By studying the time cost, we can visualise the effectiveness of AccPR in improving the repair efficiency of the APR tool. The NPC metric can reflect the accuracy of patch generation and thus reflect the effectiveness of the more accurate similarity measurement. These two metrics can be corroborated, as intuitively the larger the NPC, the more time is consumed to validate the patches and vice versa. Based on the two evaluation metrics, we aim to investigate the following research questions:

3.2.1 RQ1: How does AccRP perform on the time cost metric? Table 1 presents the detailed results in our experiment, where we list the time cost for each individual bug. Also, we have visualized the distribution of time cost according to different projects in Figure 2. According to the results, AccPR significantly improves the repair efficiency of SimFix, i.e., requiring less time to repair a bug. AccPR achieves faster repairs on 22 of the 27 bugs, counting for about 80%. On the bug of Closure-62, the improvement in time cost was even as high as 84%. More concretely, the average repair time is reduced from 11.34 to 5.91 minutes, achieving 47.85% repair time reduction on average. In summary, these results show that AccPR can effectively reduce the time required for redundancy-based APR tools and significantly improve their repairing efficiency.

3.2.2 RQ2: How does AccRP perform on the NPC metric? Figure 3 and Table 1 show the detailed results on the NPC metric. Similar to the results on the time cost metric, AccPR achieves a significant improvement on the NPC metric as well. Specifically, the number of patches validated before the correct one is reduced by 46.48% on average on 23 of the 27 bugs, counting for about 85%, which means that AccPR wastes far less time generating and validating incorrect

Table 1: Detailed experimental results

Bug	Time	NPC	Bug	Time	NPC
Ch1	**10.76** (18.32)	**149** (213)	Cl57	**4.22** (4.69)	18 (**15**)
Ch3	Fail (**39.21**)	-	Cl62	**10.55** (66.11)	**121** (534)
Ch7	**9.08** (20.61)	**153** (591)	Cl73	**1.49** (1.65)	**1** (2)
Ch20	0.93 (**0.87**)	17 (**5**)	Cl115	**2.41** (2.78)	**2** (3)
M5	**2.46** (2.79)	**1** (**1**)	L16	**7.62** (Fail)	-
M50	**2.28** (2.50)	**1** (**1**)	L27	**15.56** (26.27)	**569** (985)
M53	6.89 (**4.51**)	14 (**2**)	L33	**1.23** (2.72)	**22** (62)
M57	**2.02** (5.81)	**4** (97)	L39	**28.94** (42.00)	**659** (977)
M59	**2.71** (3.28)	**9** (34)	L41	**2.53** (8.07)	**39** (154)
M63	**2.66** (3.35)	**16** (41)	L50	**12.04** (25.89)	**250** (348)
M70	**0.36** (0.58)	**1** (**1**)	L58	0.35 (**0.32**)	**1** (**1**)
M71	**6.11** (10.80)	**50** (142)	L60	**3.63** (7.82)	**55** (112)
M75	**0.45** (0.89)	**1** (8)	T7	**8.83** (Fail)	
M79	11.64 (**9.58**)	336 (**127**)			
Avg.	Time: **5.91** (11.34)		NPC: **99** (186)		

Abbreviation- Ch: Chart, M: Math, Cl: Closure, L: Lang, T: Time.
X(Y)- X is the result of AccPR, while Y is the result of SimFix.

patches before finding the correct one. This result also shows that AccPR can pick out more compliant candidate code snippets to generate higher quality patches. This, combined with the adaptive filtering strategy for candidate patch list, allows the correct patch to be discovered as early as possible.

3.2.3 RQ3: How does AccRP perform on the number of bugs that can be successfully repaired? Figure 4 shows the results on the number of bugs that can be successfully repaired. Particularly, AccPR successfully repaired two more bugs that SimFix failed to repair as SimFix ranks the correct patch too far behind. On the contrary, benefiting from the more accurate similarity measurement for better patch generation and the adaptive patch filtering strategy, AccPR can rank the correct patch higher and therefore make successful fixes. Unfortunately, AccPR failed to repair Chart-3 because the desired patch was mistakenly filtered out due to an incorrect similar patch that was validated earlier, which could be solved by a better patch generation or filtering strategy and is worth studying more.

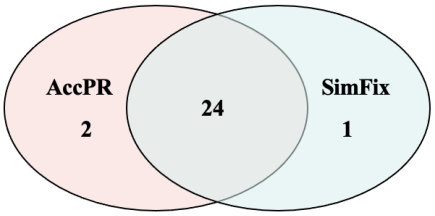

Figure 4: Overlaps of fixed bugs

3.2.4 RQ4: Will the extra time overhead associated with the patch embedding process reduce the efficiency of AccPR?. Extra time overheads will be introduced in our patch embedding process since representing code snippets is a time-consuming operation. That's the reason why AccPR did not yield better results on bugs that could have been fixed quickly by SimFix (i.e.,Ch20, M53, M79 and L58 shown in Table 1), as the room for improvement was already limited and the extra time overhead would be non-negligible on these bugs. However, AccPR can still achieve better results on bugs that would otherwise take a long time to fix, because there is still much room for improvement on the inaccurate similarity and patch order problem and the extra time overhead could be ignored compared to the improvement brought by the patch embedding process. In summary, the initial promising result on most cases demonstrates the effectiveness of AccPR.

The experimental results show that AccPR is promising to accelerate existing redundancy-based APR techniques by overcoming the *inaccurate similarity* and *patch order* problems.

4 RELATED WORK

4.1 Automated Program Repair

Automated program repair(APR) is becoming an increasingly popular area of research in recent years. Many promising approaches have been proposed and redundancy-based APR techniques are one important category in this field. The redundancy assumption has been leveraged extensively by these program repair approaches which mainly depends on similar code snippets(called donor code) to generate patches. For example, ssFix [34] reuses similar code in a fine-grained granularity via a differencing algorithm. Another related work is GenProg [21, 31], which applies genetic programming to mutate existing source code for patch generation. SearchRepair [16] considers existing code reusing as well. And all such redundancy-based APR methods based on similar code reusing can be combined with our approach.

4.2 Similar Code Identification

SimFix depends similar code searching in a project, consulting the work on code clone detection [15, 17]. In particular, the structure similarity used in SimFix inspired by DECKARD [11], a fast clone detection approach. Many existing techniques dedicate to the identification of the differences between two code snippets and the generation of the transformations from one to the other. ChangeDistiller [5] is a widely-used approach for source code transformation at AST level. GumTree [3] improves ChangeDistiller by removing the assumption that leaf nodes contains a significant amount of text. Recently deep learning based approaches have drawn much attention to learn the representation of source code. TBCNN [27] uses custom convolutional neural network on ASTs to learn vector representations of code snippets. A transformer-based embedding model is used in code search to map source code and natural language descriptions and learn the distributed representation of source code [4]. And GraphCodeBERT [8] improves it by leveraging the data flow information and combining transformer with a graph neural network. It is convenient to measure code similarity by using the vector representations obtained from these learning based methods. For example, DeepSim [37] encodes code control flow and data flow into a semantic matrix for measuring code functional similarity.

4.3 Deep Learning in Software Engineering

Many deep learning applications in software engineering have been emerging in recent years. DeepAPI [7] uses a sequence-to-sequence neural network to learn representations of natural language queries and predict relevant API sequences. And Recently, White et al. [33] introduced deep learning into automated program repair and proposed a neural network based APR technique DeepRepair.

5 CONCLUSION AND FUTURE WORK

To accelerate redundancy-based APR techniques, we proposed a novel method (named AccPR) by leveraging code representation learning for better similarity measurement and designing a novel adaptive patch filtering strategy. To evaluate its effectiveness, we implemented a prototype of it and integrated it with SimFix. The initial experimental results on Defects4J demonstrate the effectiveness of AccPR. In the future, we will further improve AccPR by exploring more advanced code representation learning methods and evaluate it on a wider range of benchmarks and APR tools.

ACKNOWLEDGMENTS

Special thanks to Junjie Chen and Jiajun Jiang for their supervision and help.

REFERENCES

[1] Zimin Chen and Martin Monperrus. 2018. The remarkable role of similarity in redundancy-based program repair. *arXiv preprint arXiv:1811.05703* (2018).

[2] Altino Dantas, Eduardo F. de Souza, Jerffeson Souza, and Celso G. Camilo-Junior. 2019. Code Naturalness to Assist Search Space Exploration in Search-Based Program Repair Methods. In *Search-Based Software Engineering*, Shiva Nejati and Gregory Gay (Eds.). Springer International Publishing, Cham, 164–170.

[3] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. 2014. Fine-grained and Accurate Source Code Differencing. *ACM* (2014), 313–324.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *EMNLP*.

[5] B. Fluri, M Wü, M. Pinzger, and H. C. Gall. 2007. change distilling: tree differencing for fine- grained source code change extraction. *IEEE Trans. Softw. Eng.* (nov 2007), 725–743.

[6] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[7] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 631–642.

[8] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. Graphcodebert: Pre-training code representations with data flow. In *ICLR*.

[9] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 388–398. https://doi.org/10.1109/ASE.2019.00044

[10] Tao Ji, Liqian Chen, Xiaoguang Mao, and Xin Yi. 2016. Automated Program Repair by Using Similar Code Containing Fix Ingredients. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 197–202. https://doi.org/10.1109/COMPSAC.2016.69

[11] Jiang, LX, Misherghi, G, Su, ZD, Glondu, and S. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. *PROC INT CONF SOFTW ENG* (2007).

[12] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 255–266. https://doi.org/10.1109/ASE.2019.00033

[13] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 298–309.

[14] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, 437–440.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans Softw Eng. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[16] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2016. Repairing Programs with Semantic Code Search (T). In *IEEE/ACM International Conference on Automated Software Engineering*.

[17] R. Koschke, R. Falke, and P. Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. In *2006 13th Working Conference on Reverse Engineering*.

[18] A. Koyuncu, K. Liu, Tegawendé F Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024.

[19] Xbd Le, D. H. Chu, D. Lo, C. L. Goues, and W. Visser. 2017. S3: Syntax-and Semantic-Guided Repair Synthesis via Programming by Examples. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[20] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software quality journal* 21, 3 (2013), 421–443.

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[22] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs *(ICSE '20)*. New York, NY, USA, 615–627. https://doi.org/10.1145/3377811.3380338

[23] M. Martinez and M. Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *International Symposium on Software Testing and Analysis*.

[24] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches *(ICSE Companion 2014)*. New York, NY, USA, 492–495. https://doi.org/10.1145/2591062.2591114

[25] S. Mechtaev, J. Yi, and A. Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*.

[26] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[27] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. 2014. Convolutional Neural Networks over Tree Structures for Programming Language Processing. (2014).

[28] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems *(ISSTA 2015)*. 24–36. https://doi.org/10.1145/2771783.2771791

[29] Yingyi Wang, Yuting Chen, Beijun Shen, and Hao Zhong. 2017. CRSearcher: Searching Code Database for Repairing Bugs. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware* (Shanghai, China) *(Internetware'17)*. Association for Computing Machinery, New York, NY, USA, Article 16, 6 pages. https://doi.org/10.1145/3131704.3131720

[30] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 356–366.

[31] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.

[32] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1–11. https://doi.org/10.1145/3180155.3180233

[33] M. White, M. Tufano, M. Martinez, M. Monperrus, and D Poshyvanyk. 2017. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. (2017).

[34] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 660–670. https://doi.org/10.1109/ASE.2017.8115676

[35] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair *(ICSE '18)*. 789–799. https://doi.org/10.1145/3180155.3180182

[36] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.

[37] Gang Zhao and Jeff Huang. 2018. DeepSim: deep learning code functional similarity. In *the 2018 26th ACM Joint Meeting*.