High-Quality Automated Program Repair

Manish Motwani
College of Information and Computer Sciences
University of Massachusetts Amherst, USA
mmotwani@cs.umass.edu

***Abstract*—Automatic program repair (APR) has recently gained attention because it proposes to fix software defects with no human intervention. To automatically fix defects, most APR tools use the developer-written tests to (a) localize the defect, and (b) generate and validate the automatically produced candidate patches based on the constraints imposed by the tests. While APR tools can produce patches that appear to fix the defect for 11–19% of the defects in real-world software, most of the patches produced are not *correct* or acceptable to developers because they overfit to the tests used during the repair process. This problem is known as the *patch overfitting* problem. To address this problem, I propose to equip APR tools with additional constraints derived from natural-language software artifacts such as bug reports and requirements specifications that describe the bug and intended software behavior but are not typically used by the APR tools. I hypothesize that patches produced by APR tools while using such additional constraints would be of higher quality. To test this hypothesis, I propose an automated and objective approach to evaluate the quality of patches, and propose two novel methods to improve the fault localization and developer-written test suites using natural-language software artifacts. Finally, I propose to use my patch evaluation methodology to analyze the effect of the improved fault localization and test suites on the quality of patches produced by APR tools for real-world defects.**

I. Research Problem and Hypothesis

Automated program repair (APR) tools aim to reduce the cost of manually fixing bugs by automatically producing patches [1]–[3]. APR tools have been successful enough to be used in industry [4], [5]. The goal of APR tools is to take a program and a suite of tests, some of which that program passes and some of which it fails, and to produce a patch that makes the program pass all the tests in that suite. Unfortunately, these patches can repair some functionality encoded by the tests, while simultaneously breaking other, undertested functionality [6]. Thus, *quality* of the resulting patches is a critical concern. Recent results suggest that the *patch overfitting* problem — patches pass a particular set of test cases supplied to the APR tool but fail to generalize to the desired specification — is common [6]–[9] and more than 50% of the patches produced by APR tools overfit to the tests used in the repair process [10]. This makes developers lose trust in the APR tools deterring their wide-scale adoption in practice [11]. The goal of my dissertation is to improve the quality of APR tools.

Most state-of-the-art APR tools use developer-written tests to localize the defect, and generate and validate the automatically produced candidate patches based on the constraints imposed by the tests. While test suites provide an easy-to-use (because they are executable) specification, software typically contains many more artifacts that describe the desired *correct* software behavior. Many of these software artifacts such as requirements specifications, code comments, and bug reports use natural-language text to describe the bug and intended software behavior, and are therefore not directly used by the APR tools. I hypothesize that *if I can derive executable constraints from such natural-language software artifacts and equip APR tools with these additional constraints, it could further constraint the search space of the candidate patches and would improve the quality of patches produced.* The central goal of my dissertation is to test this hypothesis and for that I divide my dissertation work into the following four thrusts:

1. **Evaluating patch quality (Section II-A).** The goal of this thrust is to develop an objective and scalable methodology to evaluate the quality of patches produced by APR tools. I plan to use this methodology to evaluate the quality of patches produced by APR tools for the real-world defects, and analyze how test suite characteristics correlate with patch quality.
2. **Improving fault localization (Section II-B).** The goal of this thrust is to develop an approach that uses multiple information sources such as bug reports and test suites to locate defective program elements. I plan to evaluate this approach to localize defects in large, real-world programs and study the effect of improved fault localization on patch quality.
3. **Improving test suites (Section II-C).** The goal of this thrust is to develop a technique to generate executable tests with oracles from natural language software specifications. I plan to evaluate this technique by generating tests from the publicly accessible and reliable specifications of real-world software and analyze the effectiveness of the generated tests. Further, I plan to study the effect of improved test suites on patch quality.
4. **Investigating the effect of improved fault localization and test suites on patch quality (Section II-D).** The goal of this thrust is to put together the improved fault localization (Section II-B) and developer-written test suites (Section II-C), and then to use the proposed patch evaluation methodology (Section II-A) to validate the proposed hypothesis using real-world defects.

APR tools typically follow a three-step process: (1) identify the location of the defect (*fault localization*), (2) generate candidate patches by modifying defective location (*patch generation*), and (3) validate if the candidate patch fixes the defect (*patch validation*). The method used for each

of these steps can significantly affect the tool's success. Existing research in APR has mostly focused on devising novel patch generation algorithms (e.g., heuristic-based [12]–[16], constraint-based [17]–[20], and learning-based [21]–[23]) aimed to produce more correct patches. Recently, researchers have started investigating the effect of using different technologies, assumptions, and adaptations of fault localization techniques [24]–[29], and patch validation methodologies [30]–[35] on the performance of APR tools. Recent patch validation methodologies employ machine learning models to identify overfitted patches based on their source code features. Unlike existing methods, my proposed methods aim to improve the steps of the repair process by using information derived from natural-language software artifacts. The improved fault localization shall enable APR tools to produce patches for the correct defective locations while the improved test suites will increase the constraints imposed on candidate patches during the patch generation and validation steps, and therefore shall improve the quality of produced patches.

II. Research Contributions and Results So Far

This section describes the research contributions organized in terms of the four thrusts of the dissertation. I describe each contribution in the context of existing research work and present the results obtained so far.

A. Evaluating Patch Quality

To address the patch overfitting problem, we first need a method to evaluate the quality of the produced patches. Prior studies of quality of APR have either used manual inspection [7], [36], or have used automatically generated, independent, evaluation test suites not used during the repair process [6], [37]. The issue with manual inspection is that it cannot scale to evaluate hundreds of automatically produced patches and can be subject to subconscious bias, especially if the inspectors are authors of the tools being evaluated [38]. Contrastingly, using evaluation tests is inherently partial, as the generated tests may undertest the patched program. Existing studies that use evaluation tests focus on small programs and relatively-easy-to-fix defects [6], [37].

To address these issues, I proposed a methodology [10] that uses *high–quality* evaluation test suites to evaluate the quality of the produced patches. My methodology uses today's state-of-the-art test-suite generation techniques and overcomes their shortcomings to produce high-quality test suites. The automatically generated evaluation test suites used in my methodology cover 100% of all the developer-modified methods and at least 80% of all the developer-modified classes. My methodology ensures that the evaluation test suites do not undertest the patched program.

I performed a detailed study [10] to investigate the patch overfitting problem and identify the correlation between test suite characteristics and patch quality. I evaluated four representative APR tools (GenProg [39], TrpAutoRepair [40], Par [41], and SimFix [16]) on 357 real-world defects in 5 large, complex Java projects from the Defects4J benchmark [42]. My evaluation employed rigorous statistical analyses and controlled for confounding factors to increase the likelihood that my results generalize. I answered four research questions:

RQ1.1 How often and how much do the patches produced by APR tools overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification? Often. For the four tools I evaluated, only between 13.8% and 41.6% of the patches pass 100% of an independent test suite. Patches typically break more functionality than they repair.

RQ1.2 How do the coverage and size of the test suite used to produce the patch affect patch quality? Larger test suites produce slightly higher-quality patches, though, surprisingly, the effect is extremely small. Also surprisingly, higher-coverage test suites correlate with lower quality, but, again, the effect size is extremely small.

RQ1.3 How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit? The number of failing tests correlates with slightly higher quality patches.

RQ1.4 How does the test suite provenance (whether it is written by developers or generated automatically) influence patch quality? Test suite provenance has a significant effect on patch quality, although the effect may differ for different APR tools. In most cases, human-written tests lead to higher-quality patches.

> These results corroborate the patch overfitting problem in the state-of-the-art APR tools. Further, these results indicate that improving the quality of test suites used in the repair process can potentially improve the quality of automatically produced patches.

B. Improving Fault Localization

Fault localization (FL) is recently identified as a key aspect of APR that affects patch correctness [17], [24], [25], [28], [43], [44]. To identify the defective program elements, most APR tools use spectrum-based fault localization (SBFL), which uses test-execution coverage to compute the suspiciousness scores of the program's elements, such as classes, methods, and statements. The elements are ranked based on these scores and APR tools use top-ranked elements as candidate locations to patch defects. To the best of my knowledge, only two APR tools, R2Fix [45] and iFixR [27], use information retrieval-based fault localization (IRFL), which ranks suspicious program elements based on their similarity with the text in bug reports. Using SBFL and IRFL can be complementary. For example, iFixR patches defects that 16 SBFL APR tools cannot, and vice versa [27]. Recent studies also show that combining FL techniques that use different information sources (e.g., SBFL using test suites and IRFL using bug reports) can significantly outperform individual FL techniques in terms of localizing defects [46]–[48]. Based on these findings, I hypothesize that using combined SBFL and IRFL can enable APR tools to patch

all the defects that they can patch when using the underlying SBFL and IRFL alone, and perhaps some others. To the best of my knowledge, this is the first investigation of the effect of combined FL on APR.

Existing approaches [46]–[50] to combine multiple FL techniques, are based on *learning to rank* [51], supervised deep machine learning techniques. The performance and generalizability of such approaches depend heavily on the dataset and features used for training the machine learning model. I proposed to use an unsupervised approach that requires no training. To combine FL techniques, I developed Rank Aggregagtion-based FL (RAFL) [52], a novel approach that uses rank aggregation algorithms [53] to combine the top-k ranked statements produced using different FL techniques. RAFL measures the similarity of the two ranked lists using the Spearman footrule distance [54] and runs the cross-entropy Monte Carlo algorithm [55] to produce a super list of top-k statements while maximizing the similarity to the individual lists. RAFL can combine FL results obtained using any set of techniques; in my dissertation, I specifically focus on combining SBFL and IRFL, as these two are used in APR.

Existing IRFL techniques [56]–[60] are not well suited for APR because they localize defects at the file or method level, whereas APR tools need statement-level granularity. I developed Blues [52], a statement-level IRFL technique based on BLUiR [57], an existing file-level IRFL technique. Blues considers the abstract syntax tree (AST) representations of program statements as a collection of documents, and bug report as a query, and uses a structured information retrieval technique to rank the statements based on their similarity with the bug report. Blues is the first unsupervised statement-level IRFL technique. The prior statement-level IRFL technique, D&C [61] used by iFixR [27], requires supervised training.

I implemented an SBFL technique using GZoltar v1.7.2, and the Ochiai ranking strategy, which is one of the most effective ranking strategies in object-oriented programs [46], [50]. I evaluated this SBFL technique, Blues, and their RAFL-enabled combination SBIR, on 818 real-world defects from 17 large Java projects in the Defects4J v2.0 benchmark [62].

To test if the combined FL improves the quality of patches, I used SimFix [16], a state-of-the-art APR tool. I chose SimFix because a recent study [27] found that it outperforms a suite of 16 other APR tools, including iFixR, kPAR [43], AVATAR [63], and LSRepair [64], as well as others. I ran SimFix on 818 defects in Defects4J v2.0 for which bug reports are available using my SBFL implementation, Blues, and SBIR. To evaluate the correctness of patches, I used my patch evaluation methodology (recall Section II-A). I answered three research questions:

RQ2.1 Does SBIR localize more defects than the underlying SBFL and Blues techniques? Yes, SBIR outperforms SBFL and Blues, for all sizes of suspicious statement lists investigated (1, 25, 50, 100). For example, SBIR correctly identifies a buggy statement as the most suspicious for 148 of the 818 (18.1%) defects, whereas SBFL does so for 89 (10.9%) and Blues for 25 (3.1%).

RQ2.2 Does using SBIR in APR patch more defects? Using SBIR enables SimFix to patch marginally more defects (112 out of 818) than using SBFL (110) and significantly more than using Blues (55). With SBIR, SimFix produces patches for most of the defects patched using SBFL or Blues, as well as 3 new defects that could not be patched previously. Further, with SBIR, SimFix can produce all but one (29 out of 30) of the correct patches it produces with SBFL, and all (16 out of 16) of the correct patches with Blues. Additionally, SimFix with my FL implementations patches 10 defects that none of 14 state-of-the-art APR tools patch [43]. Finally, using Blues, SimFix significantly outperforms iFixR, the state-of-the-art IRFL-based APR tool [27], patching 19 out of 156 defects (7 correctly) while iFixR patches only 4 defects (3 correctly).

RQ2.3 How does the patch quality vary across the new and old versions of Defects4J benchmark? Past APR evaluations fail to generalize to new defects. For example, SimFix correctly patches 3–6% (6% when using SBFL, 3% Blues, 6% SBIR) of the defects in the older version of Defects4J, but only 1–2% (2% SBFL, 1% Blues, 2% SBIR) of the *new* defects. Of the patches SimFix produces for the old defects, 39–40% are correct; for the new defects, only 13–19% are correct.

> These results show that improving FL can enable APR tools to patch new defects without requiring any changes to their core patch generation and validation algorithms. A recent study [65] shows that the quality of patches produced by some APR tools is more sensitive to the accuracy of FL results they use than others. Hence, I plan to extend my evaluation of using improved fault localization to more sensitive APR tools.

C. Improving Test Suites

The developer-written tests are often inadequate [66] yet they are used by most APR tools because the tests are readily available and are machine-processable. Tests consist of two parts, an input to trigger a behavior and an oracle that indicates the expected behavior. While the state-of-the-art automated test generation techniques (e.g., Randoop [67], EvoSuite [68]) can effectively generate test inputs, they require a reference implementation to compute oracles for the generated inputs. In practice, a correct reference implementation may not be available, thus, limiting the use of such test generation techniques to improve the quality of APR tools. To address this, I analyzed other software artifacts from which I can derive the intended software behavior and improve the developer-written tests, which are used by APR tools. While formal, mathematical specifications that can be used automatically by computers are rare, developers do write natural language (NL) specifications, often structured (e.g., JavaDoc comments), as part of software requirements specification documents. Hence, in this thrust, I tackle the problem of automatically generating tests from such structured NL specifications to verify that the software does what the specifications say it should.

I developed Swami [69], a technique to automatically generate executable tests from structured NL specifications. I scoped my work by focusing on exceptional and boundary behavior, precisely the important-in-the-field behavior developers often undertest [70], [71]. Swami uses regular expressions to identify what sections of the specification document encode testable behavior. Swami then applies a series of four regular-expression-based rules to extract information about the syntax for the methods to be tested, the relevant variable assignments, and the conditionals that lead to visible oracle behavior, such as return statements or exception throwing statements. Swami then backtracks from the visible-behavior statements to recursively fill in the variable value assignments according to the specification, resulting in a test template encoding the oracle, parameterized by test inputs. Swami then generates random, heuristic-driven test inputs to produce executable tests.

Swami complements prior work (e.g, [67], [68]) on automatically generating test inputs for regression tests or manually-written oracles by automatically extracting oracles from NL specifications. The closest work to Swami is Toradacu [70] and Jdoctor [72], which focus on extracting oracles for exceptional behavior, and @tComment [73], which focuses on extracting preconditions related to nullness of parameters. These techniques are limited to using Javadoc comments, which are simpler than the specifications Swami tackles. Swami builds on these techniques, expanding the rule-based NL processing techniques to apply to more complex NL. Additionally, unlike those techniques, Swami generates oracles for boundary conditions along with exceptional behavior.

I evaluated Swami using ECMA-262, the official specification of the JavaScript programming language [74], and two well known JavaScript implementations: Java Rhino and C++ Node.js. I answered three research questions:

RQ3.1 How precise are Swami-generated tests? Of the tests Swami generates, 60.3% are innocuous — they can never fail. Of the remaining tests, 98.4% are precise to the specification and only 1.6% might raise false alarms.

RQ3.2 Do Swami-generated tests cover behavior missed by developer-written tests? Swami-generated tests identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification. Further, Swami generated tests for behavior uncovered by developer-written tests for 12 Rhino methods. The average statement coverage for these methods improved by 15.2% and the average branch coverage improved by 19.3%.

RQ3.3 Do Swami-generated tests cover behavior missed by state-of-the-art automated test generation tools? Comparing Swami tests to EvoSuite tests revealed that most of the EvoSuite tests that cover exceptional behavior were false alarms, whereas 98.4% of the Swami tests were precise to the specification and can only result in true alarms. Augmenting EvoSuite tests using Swami increased the statement coverage of 47 classes by, on average, 19.5%. Swami also produced fewer false alarms than Toradacu and Jdoctor, and, unlike those tools, generated tests for missing features.

To study the effect of improved test suites on patch quality, I plan to create a dataset of defects for which Swami can be used to improve developer-written tests. I will then perform controlled experiments on that dataset using APR tools to patch those defects by using the original and developer-written tests augmented with Swami tests in the repair process.

D. Investigating the effect of improved fault localization and test suites on patch quality.

This thrust aims to put together the improved fault localization (FL) (Section II-B) and test suites (Section II-C), and then use the proposed patch evaluation methodology (Section II-A) to measure the repair success. I will first create a defect dataset on which I can run different FL techniques, as well as improve test suites using Swami. Next, I will select a representative set of APR tools that are more sensitive to FL accuracy and use them to patch the defects. Finally, I will study the individual as well as the combined effect of improving FL and test suites on the patch quality. I will answer three research questions:

RQ4.1 Does improving fault localization using SBIR improves patch quality?

RQ4.2 Does improving test suites using Swami improves patch quality?

RQ4.3 Does improving both fault localization and test suites improve patch quality?

III. Expected Contributions and Timeline

My dissertation will make the following contributions that will extend the state-of-the-art of APR along with other closely related fields. I plan to release all of the data and tools developed as part of my dissertation as freely available and reusable open-source software.

1) An APR quality evaluation framework that provides an automated and objective methodology to evaluate patch quality and allows APR tools to use multiple fault localization (FL) techniques in the repair process.
2) JaRFly[1], the Java Repair Framework, which simplifies the implementation of Java techniques for genetic improvement including but not limited to genetic improvement techniques for APR.
3) RAFL, an unsupervised learning-based approach to improve the FL accuracy by combining results of multiple FL techniques that use different information sources.
4) Blues, the first unsupervised learning-based statement-level IRFL technique that can be used by the APR tools to localize and patch defects using bug reports.
5) Swami[2], a novel approach to generate tests with oracles from structured natural language software specifications for exceptional behavior and boundary conditions.

I am currently working on the fourth thrust (Section II-D) and I plan to finish all of my dissertation work by late 2021.

[1]http://jarfly.cs.umass.edu

[2]http://swami.cs.umass.edu

REFERENCES

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. on Software Eng.*, vol. 45, no. 01, pp. 34–67, 2019.

[2] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019. [Online]. Available: http://doi.acm.org/10.1145/3318162

[3] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3105906

[4] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages (PACMPL) Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) issue*, vol. 3, October 2019.

[5] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *ACM/IEEE Int. Conference on Software Engineering*, 2019, pp. 269–278.

[6] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *ESEC/FSE*, 2015, pp. 532–543.

[7] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Baltimore, MD, USA, 2015, p. 2436.

[8] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, Buenos Aires, Argentina, 2016, pp. 702–713.

[9] X. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2018, pp. 163–163.

[10] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of automated program repair on real-world defects," *IEEE Transactions on Software Engineering*, 2020.

[11] G. M. Alarcon, C. Walter, A. M. Gibson, R. F. Gamble, A. Capiola, S. A. Jessup, and T. J. Ryan, "Would you fix this code for me? effects of repair source and commenting on trust in code repair," *Systems*, vol. 8, no. 1, p. 8, 2020.

[12] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, pp. 54–72, 2012.

[13] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

[14] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in C," in *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Paderborn, Germany, Sep. 2017, pp. 752–762.

[15] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, Jun. 2018, pp. 1–11.

[16] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, The Netherlands, Jul. 2018, pp. 298–309.

[17] A. Afzal, M. Motwani, K. Stolee, Y. Brun, and C. Le Goues, "SOSRepair: Expressive semantic search for real-world program repair," *IEEE Transactions on Software Engineering*, 2019.

[18] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, USA, Jun. 2018, pp. 481–495.

[19] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, USA, Jun. 2018, pp. 465–480.

[20] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, 2018, pp. 129–139.

[21] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. on Software Engineering*, 2019.

[22] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "DeepFix: Fixing common C language errors by deep learning," in *National Conference on Artificial Intelligence (AAAI)*, San Francisco, CA, USA, Feb. 2017, pp. 1345–1351.

[23] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: Effective object oriented program repair," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, IL, USA, Nov. 2017, pp. 648–659.

[24] F. Y. Assiri and J. M. Bieman, "Fault localization for automated program repair: Effectiveness, performance, repair correctness," *Software Quality Journal*, vol. 25, no. 1, pp. 171–199, 2017.

[25] D. Yang, Y. Qi, and X. Mao, "Evaluating the strategies of statement selection in automated program repair," in *International Conference on Software Analysis, Testing, and Evolution*. Springer, 2018, pp. 33–48.

[26] S. Sun, J. Guo, R. Zhao, and Z. Li, "Search-based efficient automated program repair using mutation and fault localization," in *Annual Computer Software and Applications Conference*, vol. 1, 2018, pp. 174–183.

[27] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. L. Traon, "iFixR: Bug report driven program repair," in *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2019, pp. 314–325.

[28] J. Jiang, Y. Xiong, and X. Xia, "A manual inspection of defects4j bugs and its implications for automatic program repair," *Science China Information Sciences*, vol. 62, no. 10, p. 200102, 2019.

[29] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event, USA, 2020, pp. 75–87.

[30] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Paderborn, Germany, 2017, pp. 831–841.

[31] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, Jun. 2018, pp. 789–799.

[32] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system," *Empirical Software Engineering*, vol. 24, no. 1, pp. 33–67, 2019.

[33] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019, pp. 8–18.

[34] H. Tian, K. Liu, A. K. Kaboreé, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," *arXiv preprint arXiv:2008.02944*, 2020.

[35] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[36] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset," *EMSE*, vol. 22, no. 4, pp. 1936–1964, April 2017.

[37] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," in *IEEE International Workshop on Intelligent Bug Fixing (IBF)*, Hangzhou, China, Feb. 2019, pp. 1–10.

[38] X. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. S. Pasareanu, "On reliability of patch correctness assessment," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, May 2019.

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

[40] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *ICSM*, Sep. 2013, pp. 180–189.

[41] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486893

[42] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *ISSTA*, 2014, pp. 437–440.

[43] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *IEEE International Conference on Software Testing, Verification, and Validation*, Xian, China, 2019, pp. 102–113.

[44] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "An empirical analysis of the influence of fault space on search-based automated program repair," *arXiv preprint arXiv:1707.05172*, 2017.

[45] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2Fix: Automatically generating bug fixes from bug reports," in *IEEE International Conference on Software Testing, Verification and Validation*, 2013, pp. 282–291.

[46] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[47] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[48] T. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[49] J. Sohn and S. Yoo, "FLUCCS: Using code and change metrics to improve fault localization," in *International Symposium on Software Testing and Analysis*, Santa Barbara, CA, USA, 2017, pp. 273–283.

[50] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *IEEE International Conference on Software Maintenance and Evolution*, Victoria, BC, 2014, pp. 191–200.

[51] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," in *ACM International Conference on Machine Learning*, 2005, pp. 89–96.

[52] M. Motwani and Y. Brun, "Automatically repairing programs using both tests and bug reports," *arXiv:2011.08340*, 2020. [Online]. Available: http://arxiv.org/abs/2011.08340

[53] S. Lin, "Rank aggregation methods," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 5, pp. 555–570, 2010.

[54] F. J. Brandenburg, A. Gleißner, and A. Hofmeier, "The nearest neighbor spearman footrule distance for bucket, interval, and partial orders," *Jour. of Combinatorial Optimization*, vol. 26, no. 2, pp. 310–332, 2013.

[55] R. Y. Rubinstein and D. P. Kroese, *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Springer Science & Business Media, 2013.

[56] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[57] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Palo Alto, CA, USA, 2013, pp. 345–355.

[58] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *IEEE International Conference on Software Maintenance and Evolution*, Victoria, BC, 2014, pp. 181–190.

[59] K. C. Youm, J. Ahn, J. Kim, and E. Lee, "Bug localization based on code change histories and bug reports," in *IEEE Asia-Pacific Software Engineering Conference*, New Delhi, India, 2015, pp. 190–197.

[60] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *IEEE/ACM International Conference on Automated Software Engineering*, Singapore, 2016, pp. 262–273.

[61] A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. L. Traon, "D&C: A divide-and-conquer approach to IR-based bug localization," *ArXiv*, vol. abs/1902.02703, 2019.

[62] G. Gay and R. Just, "Defects4J as a challenge case for the search-based software engineering community," in *Proceedings of the International Symposium on Search-Based Software Engineering (SSBSE)*, Oct. 2020.

[63] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *Int. Conf. on Software Analysis, Evolution and Reengineering*, 2019, pp. 1–12.

[64] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "LSRepair: Live search of fix ingredients for automated program repair," in *Asia-Pacific Software Engineering Conference*, 2018, pp. 658–662.

[65] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2020.

[66] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel, "How well do professional developers test with code coverage visualizations? an empirical study," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005, pp. 53–60.

[67] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, Montreal, QC, Canada, 2007, pp. 815–816. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297902

[68] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, February 2013.

[69] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 188–199.

[70] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrücken, Genmany, July 2016, pp. 213–224.

[71] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," in *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, Vancouver, BC, Canada, 2004, pp. 419–431.

[72] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, Netherlands, 2018, pp. 242–253.

[73] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *International Conference on Software Testing, Verification, and Validation (ICST)*, Montreal, QC, Canada, 2012, pp. 260–269.

[74] A. Wirfs-Brock and B. Terlson, "ECMA-262, ECMAScript 2017 language specification, 8th edition," https://www.ecma-international.org/ecma-262/8.0, 2017.