Towards Practical and Useful Automated Program Repair for Debugging

Qi Xin
qxin@whu.edu.cn
School of Computer Science, Wuhan University
Hubei Luojia Laboratory
China

Haojun Wu
haojunwu@whu.edu.cn
School of Computer Science, Wuhan University
China

Steven P. Reiss
spr@cs.brown.edu
Department of Computer Science, Brown University
USA

Jifeng Xuan*
jxuan@whu.edu.cn
School of Computer Science, Wuhan University
China

ABSTRACT

Current automated program repair (APR) techniques are far from being practical and useful enough to be considered for realistic debugging. They rely on unrealistic assumptions including the requirement of a comprehensive suite of test cases as the correctness criterion and frequent program re-execution for patch validation; they are not fast; and their ability of repairing the commonly arising complex bugs by fixing multiple locations of the program is very limited. We hope to substantially improve APR's practicality, effectiveness, and usefulness to help people debug. Towards this goal, we envision PracAPR, an interactive repair system that works in an Integrated Development Environment (IDE) to provide effective repair suggestions for debugging. PracAPR does not require a test suite or program re-execution. It assumes that the developer uses an IDE debugger and the program has suspended at a location where a problem is observed. It interacts with the developer to obtain a problem specification. Based on the specification, it performs test-free, flow-analysis-based fault localization, patch generation that combines large language model-based local repair and tailored strategy-driven global repair, and program re-execution-free patch validation based on simulated trace comparison to suggest repairs. By having PracAPR, we hope to take a significant step towards making APR useful and an everyday part of debugging.

*Corresponding author

1 MOTIVATION

Programs are rarely bug-free. Debugging is an indispensable activity in software development. It is however very costly, and can consume up to 50% of the programming time [6]. To reduce the cost of debugging and make it easier, researchers have proposed the concept of automated program repair (APR) [9, 11, 28, 52] whose goal is to automatically generate a patch that corrects a buggy program's misbehavior. For over a decade, more than 60 APR techniques have been developed [29, 36]. They have sought to achieve automated repair via various strategies generally classified as pattern-based (e.g., [17, 24, 40]), constraint-based (e.g., [26, 48]), search-based (e.g., [12, 43]), and learning-based [52].

Despite the promising potential, current APR techniques are far from being practical and useful enough to be integrated into an IDE for debugging. Three key challenges remain. First, current approaches are designed based on unrealistic assumptions. They assume the existence of a test suite serving as the correctness criterion and require frequent program re-execution for repair validation.

In a realistic debugging scenario, one cannot assume the existence of a (high-quality) test suite, especially in the initial development phase of the software. Studies have shown that developers do not write test suites containing a sufficient number of test cases or even do not write tests at all [4, 19]. As also noted by Koyuncu et al. [20], bugs are often reported without an available test suite revealing them. Surprisingly, the bug-revealing test cases for over 90% of the bugs in the Defects4J dataset [15] were introduced after the bug was identified.

While some techniques [2, 3, 8, 20, 41] have sought for test-free repair, they are restricted to handling specific types of bugs (e.g., the heap-property faults [41]) and potential issues flagged by static analyzers and are not designed to repair general semantic bugs that arise while debugging.

The reliance on frequent program re-execution also makes APR not practical. In a realistic debugging scenario, recreating the environment for the immediate failure caused by the bug can be difficult since the failure can be identified in a long run or in an interactive session. Moreover, frequent program re-execution makes APR not fast. Current approaches can take minutes (e.g., [14]) or even hours to repair one bug [25]. Studies showed that developers prefer not to wait for too long [30] for repair. One can also imagine that APR,

if integrated into an IDE to provide repair suggestions, is highly expected to be quick.

Second, while APR has made remarkable progress towards local repair by generating patches addressing a single location of the program, the repair ability is still weak. Our statistics based on the previous evaluation of existing tools shows that traditional non-learning-based approaches can only repair a small fraction (less than 23%) of the 150 single-hunk bugs in the Defects4J v1.2 dataset. For these bugs, the developer patches change only a single hunk of code. Learning-based approaches, especially those using a large language model (LLM) [10, 13, 16, 39, 45, 47], represent a significant improvement. However, a state-of-the-art approach Repilot [42] still failed to repair 86 (or 57.3%) of the single-location bugs. Moreover, existing techniques can generate spurious overfitting patches [21, 32, 38, 49], which are harmful and can adversely affect debugging [7, 30]. In summary, APR's weak ability of local repair can lower the developer's trust of APR in dealing with even simple bugs and make the developer unwilling to use APR for debugging.

Finally, current APR cannot do effective global repair to tackle complex multi-location bugs whose fix requires changes for multiple locations of the program. Zhong and Su [53] found that multi-location bugs are common. At least 40% of real-bug fixes made by developers are used to tackle such bugs. This finding implies that if APR is not designed to support multi-location repair, it can have very limited usefulness. To enhance the usability of APR, previous techniques have attempted to address multi-location bugs using strategies such as genetic algorithms [22, 51], detection and update of evolutionary siblings [37], variational execution [44], deep learning [23], and iterative self-supervised training [50]. A key problem of these approaches is that most of the repaired multi-location bugs are actually multi-fault bugs exposed by multiple failures. A multi-fault bug can be decomposed into multiple single-fault bugs and is rare in realistic debugging, as developers typically deal with one failure (fault) at a time [18, 31]. To understand how existing approaches deal with single-fault multi-location bugs that commonly arise while debugging, we did a study and found that (1) about half (49.5%) of the multi-location bugs in the Defects4J v1.2 dataset are multi-fault, (2) the dataset has 118 single-fault multi-location bugs, and (3) current techniques [23, 37, 44, 46, 50, 51, 55] repaired at most 8 of them. This result shows that current APR's ability of repairing complex multi-location bugs is poor.

2   AN ENVISIONED REPAIR SYSTEM FOR 2030

We envision a repair system PracAPR that addresses the aforementioned challenges and provides quick repair suggestions for realistic debugging. Figure 1 shows an overview of PracAPR. To overcome the unrealistic-assumption drawback, PracAPR does not assume the existence of a test suite and does not require program re-execution. It works in conjunction with an IDE debugger and assumes that the program is stopped at a location where a problem is observed. PracAPR interacts with the developer to obtain a description of the problem and performs test-free fault localization, patch generation, and patch validation based on the description (the problem specification) to generate repair suggestions. Without a test suite, PracAPR performs flow-analysis-based fault localization while taking into account the problem symptom, current values from the

debugger, and the current runtime stack to compute a backward slice containing potential repair locations. Patch generation is done with local and global repair, which we will discuss later. Patch validation does not require program re-execution. Instead, PracAPR generates via a live programming mechanism simulated traces that reflect the real executions of the original and repaired programs and compares the traces to infer patch correctness. Finally, PracAPR presents the repair suggestions to the developer. The developer can choose to preview any of the repairs and further accept it to allow changes to be applied to the program.

For local repair, PracAPR uses an LLM-based approach, aiming to improve APR's repair ability in fixing more local bugs (that require changes of a single location of the program) and fixing them more precisely (by generating fewer bad patches). To this end, PracAPR uses an informative prompt that includes the buggy location, its context, the failure input and output, dynamic execution information (including for example the coverage and key program states), promising patterns and fix ingredients, and user guidance to help LLM accurately diagnose the problem and propose effective patches. PracAPR also allows conversational repair and refinement and re-fixing of the patches to improve the repair quality.

For global repair, PracAPR uses a variety of strategies distilled from our analysis of multi-location patches that led to a characterization of 8 types of partial patch relationships. The strategies include performing iterative repair to generate patches location by location to address bugs that require fixing different issues, performing simultaneous repair to address related issues, using local repair to tackle single-location-alike bugs, and using pattern-based methods to generate other common patches that involve for example adding a definition and use of a variable and inserting if-wrap code (i.e., an if-statement wrapping a code hunk).

3   ONGOING AND FUTURE WORK

We next discuss our ongoing and future work for realizing PracAPR.

3.1   Interactive Test-Free Repair Framework

We have developed an interactive test-free repair framework ROSE [33], which provides initial solutions for fault localization and patch validation without requiring a test suite and program re-execution.

ROSE works in the Eclipse-based Code Bubbles IDE [5] and allows easy integration of existing APR patch generators. ROSE assumes that the program is suspended at a location where unexpected behavior is observed. It interacts with the developer to obtain a problem description. The developer can specify that an exception is unexpected, a line should not be executed, or a variable should not hold a certain value. Based on the specification, ROSE performs test-free fault localization using an abstract-interpretation-based flow analysis to statically compute a partial backward slice containing potential repair locations. It invokes the patch generators that have been plugged into the framework to make patches for those locations. For patch validation without using test cases or allowing dynamic program re-execution, ROSE generates simulated traces based on a live-programming system SEEDE [35] for both the original and repaired executions and then compares the traces with respect to the problem to infer patch correctness. Finally, ROSE presents a limited number of prioritized patches. The developer can
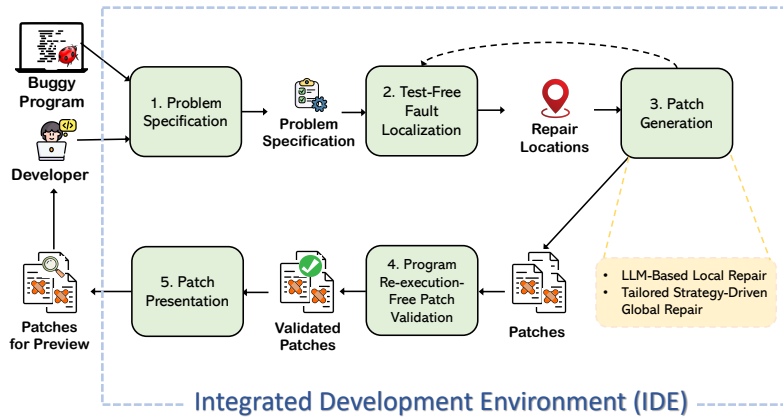
Figure 1: An overview of the PracAPR repair system.

choose a patch for a preview, which highlights the code before and after the repair with differences, and ask ROSE to make the repair. More details can be found in [33, 34].

We evaluated the effectiveness and utility of ROSE with a repair experiment and a user study. Our results showed that ROSE's test-free fault localization and patch validation are highly effective: the fault localization included the correct repair location for 89% of the bugs tested and the patch validation gave a top-5 rank for all correct repairs; that a ROSE-based tool can repair as many as 36/40 QuixBugs and 37/60 Defects4J bugs in only seconds; and that ROSE helped 44% more participants succeed in a debugging task and helped reduce the debugging time by about 16.5%. Overall, we believe that ROSE is a promising repair framework that can make debugging easier.

We plan to build PracAPR on top of ROSE, and we see two ways for improvement. First, we want to improve ROSE's user interaction for problem specification by exploring not only a better presentation of the failure information to facilitate understanding of the program semantics and the failure but also more forms of the specification (based on for example constraints and even natural language) to effectively guide fault localization and patch validation. Second, we want to investigate learning-based trace comparison while considering more information about the execution to enhance patch validation.

3.2 LLM-Based Local Repair

The LLM has demonstrated superior abilities in repairing software bugs [45, 47]. We believe that an LLM-based approach is promising in generating high-quality local patches (addressing single locations of the program for repair). Since ChatGPT is widely recognized as a prominent LLM for tackling various software engineering tasks, we consider a ChatGPT-based approach that serves as a key component of PracAPR to accurately infer the problem and provide a low number of promising patches for effective local repair.

We are conducting a study to understand the failure of ChatGPT-based approaches [45, 47] and motivate possible ways for improvement. We seek to answer three research questions: (1) What are the characteristics of the bugs that ChatGPT fails to repair? (2) What

```java
public TimeSeries createCopy(int start, int end)
        throws CloneNotSupportedException {
    if (start < 0) {
        throw new IllegalArgumentException("Requires start >= 0.")
        ;
    }
    if (end < start) {
        throw new IllegalArgumentException("Requires start <= end.
    ");
    }
    TimeSeries copy = (TimeSeries) super.clone();
    + copy.minY = Double.NaN;
    + copy.maxY = Double.NaN;
    copy.data = new java.util.ArrayList();
    if (this.data.size() > 0) {
        for (int index = start; index <= end; index++) {
            TimeSeriesDataItem item
                = (TimeSeriesDataItem) this.data.get(index);
            TimeSeriesDataItem clone = (TimeSeriesDataItem) item.
    clone();
            try {
                copy.add(clone);
            }
            catch (SeriesException e) {
                e.printStackTrace();
            }
        }
    }
    return copy;
}
```

Figure 2: Patch for the Chart_3 bug.

are the most common mistakes that ChatGPT makes? and (3) How to improve ChatGPT to repair more bugs?

Our current result shows that existing approaches are weak in that they use prompts that include only the buggy location, its limited context, and shallow information about the failure (including the input and the failing assertion). This is often insufficient for ChatGPT to understand the program semantics and the problem and can result in incorrect patches raising new problems.

Figure 2 shows for example the buggy method for the Defects4J Chart_3 bug and the patch (lines 10 and 11). To repair the bug, a state-of-the-art ChatGPT-based approach ChatRepair [47] uses a prompt that includes the code of the buggy method, the name of the failing test case `testCreateCopy3`, the failing assertion `assertEquals(101.0, s2.getMaxY(), EPSILON)`, and

the failure message `expected:<101.0> but was:<102.0>`. It does not however inform ChatGPT of the test input triggering the failure. Nor does it describe the behavior of the invoked method `add` (line 19) showing how `minY` and `maxY` are updated (key information for bug understanding) and provide the details about the failure execution. Due to insufficient knowledge of the failure, ChatGPT's problem diagnosis is shallow and inaccurate – it thought that there is a problem with the loop copying the data and did not seem to understand that it was the update of the `minY` and `maxY` values that causes the error. As a result, ChatGPT proposed a patch changing the loop condition (line 14), which is incorrect.

To help ChatGPT understand the failure, we plan to use an augmented prompt that includes not only what ChatRepair uses in its prompt but also the code of the failing test case (including test input), the definition of related methods (including `add`), and the execution trace showing not only what lines are exercised in the failure run and their order but also the key program state (variable and field values). Using a prompt like this, ChatGPT successfully understands that the failure is related to "how the min and max y values are updated after copying a subset". This finally leads to a patch that correctly updates the min and max y values.

An augmented prompt, even with more failure and execution information, may not necessarily help ChatGPT figure out what is wrong. And even if ChatGPT precisely understands the problem, it may still fail to generate the correct patch tackling the problem in the right way. For example, ChatGPT may know that there is an invalid case where the `start` index is greater than `end` but can be unsure about how to process it – whether the program should throw an exception, return a special value, or do something else. One way to mitigate this problem is to solicit user feedback showing for example an exception is expected, a certain line should not be executed, or a variable should not hold a value.

In addition to using augmented prompts, we will also explore combining ChatGPT with traditional pattern-based and search-based methods (finding for example effective patterns and fix ingredients) to guide the repair, performing conversational repair highlighting the (negative) influence of the previous patches to allow ChatGPT to reflect on its mistakes for improvement, and conducting post-processing operations refining and re-fixing the patches to improve repair quality.

3.3 Global Repair Driven by Tailored Strategies

Existing global repair techniques have adopted various strategies for multi-location repair. The evaluation of these techniques is however based on the Defects4J bug dataset [15] and is severely misguided, as the dataset is filled with multi-fault bugs. Multi-fault bugs can be decomposed into independent single-fault bugs triggering different failures. Repairing multi-fault bugs by handling multiple failures simultaneously is practically uncommon for debugging, as the developer typically deals with one failure at a time [18, 31].

To understand existing approaches' abilities of repairing single-fault multi-location bugs, we proposed an approach to detect such bugs and found that there are 118 single-fault multi-location bugs in the Defects4J v1.2 dataset and that current approaches [23, 37, 44, 46, 50, 51, 55] repaired at most 8 bugs, suggesting weak repair abilities.

We aim to design a global repair approach that can effectively address single-fault multi-location bugs. Towards this goal, we went about analyzing the developer (ground-truth) patches for a sample of the single-fault multi-location bugs (about one third, or 75 in total). We wanted to understand why the repair needs to addresses multiple locations, what are the characteristics of the partial patches made at different locations, and furthermore what strategies to consider for patch generation based on the characteristics.

The analysis has led to a characterization of 8 partial patch relationships summarized below.

- **DU**: Partial patches with this relationship add the definition of variables, fields, packages, or methods and later use what has been defined for repair.
- **OA**: Partial patches with this relationship can be done in one repair action or operation by for example adding an if-statement wrapping a code hunk.
- **RIF**: This relationship indicates that the partial patches are used to address related issues that arise in different locations.
- **DIF**: This relationship indicates that the partial patches address different issues arising from different program parts that may implement the same functionality.
- **EOH**: Partial patches with this relationship can be considered as a single-hunk patch for reasons such as that there is only one partial patch that is semantically needed and the others are created only to improve readability.
- **SU**: In this relationship, some partial patches are created to do the setup work by updating a variable, field, or method while the others use what has been updated for repair.
- **ONPF**: In this relationship, some partial patches can fix the original problem and resolve the original failure. Unfortunately, they also raise new problems triggering new failures, which can be tackled by other partial patches.
- **FU**: Some partial patches serve as the primary changes for correcting the misbehavior of the program. Others are needed to undo the negative influence brought by the previous changes.

As the next step, we plan to design specialized repair strategies based on the relationships and develop a global repair approach that uses these strategies to obtain guided exploration for effective multi-location repair. An approach that we envision to have uses the LLM-based method discussed in Section 3.2 to generate single-location patches. It performs iterative repair via repeated single-location-based fault localization and patch generation to generate patches of the DIF, ONPF, and FU relationships. Unlike existing approaches [50, 51], our approach considers a variety of program syntactic and semantic features and execution information to infer promising patches for further evolution. To generate the RIF patch, our approach reuses a simultaneous strategy [37] that identifies locations for co-evolution and applies similar changes to those locations. The approach relies on local repair to address EOH and uses pattern-based methods to generate DU, SU, and OA patches.

We envision to have a suite of specialized patch generators. Once a failure occurs, one would not easily know which generators to use for repair but can run all the generators in parallel to get all the patches. This can be further improved via a trained multi-classifier [1, 27] to select the most suitable generators or an ensemble approach (e.g., [54]) for generator prioritization.

REFERENCES

[1] Aldeida Aleti and Matias Martinez. 2021. E-APR: Mapping the effectiveness of automated program repair techniques. *Empirical Software Engineering* 26 (2021), 1–30.

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[3] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. 2019. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering*. 613–624.

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint Meeting on the Foundations of Software Engineering*. 179–190.

[5] Andrew Bragdon, Steven P Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 455–464.

[6] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* 229 (2013).

[7] Hadeel Eladawy, Claire Le Goues, and Yuriy Brun. 2024. Automated Program Repair, What Is It Good For? Not Absolutely Nothing!. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 868–868.

[8] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[9] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[10] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.

[11] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A survey on automated program repair techniques. *arXiv preprint arXiv:2303.18184* (2023).

[12] 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 ACM 27th SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 298–309.

[13] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020* (2023).

[14] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE/ACM, 1161–1173.

[15] 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 ACM 23rd SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 437–440.

[16] Sungmin Kang and Shin Yoo. 2022. Language models can prioritize patches for practical program patching. In *Proceedings of the Third International Workshop on Automated Program Repair*. 8–15.

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.

[18] Amy J Ko and Brad A Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*. 301–310.

[19] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. 2013. An empirical study of adoption of software testing in open source projects. In *Proceedings of 13th International Conference on Quality Software*. 103–112.

[20] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug report driven program repair. In *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering*. 314–325.

[21] Xuan-Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th international conference on software engineering*. 163–163.

[22] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (2011), 54–72.

[23] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. DEAR: A novel deep learning-based approach for automated program repair. In *Proceedings of IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 25–27.

[24] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of ACM 28th SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 31–42.

[25] 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. In *Proceedings of International Conference on Software Engineering*. 615–627.

[26] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 691–701.

[27] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred convolutional neural networks. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 1169–1180.

[28] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[29] Martin Monperrus. 2018. *The Living review on automated program repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.

[30] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 2228–2240.

[31] Alexandre Perez, Rui Abreu, and Marcelo d'Amorim. 2017. Prevalence of single-fault fixes and its impact on fault localization. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 12–22.

[32] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of ACM 24th SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 24–36.

[33] Steven P Reiss, Xuan Wei, and Qi Xin. 2023. Quick Repair of Semantic Errors for Debugging. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 9–10.

[34] Steven P Reiss and Qi Xin. 2022. A Quick Repair Facility for Debugging. *arXiv preprint arXiv:2202.05577* (2022).

[35] Steven P Reiss, Qi Xin, and Jeff Huang. 2018. SEEDE: simultaneous execution and editing in a development environment. In *Proceedings of 33rd IEEE/ACM International Conference on Automated Software Engineering*. 270–281.

[36] RepairTools 2024. *Program Repair Tools*. https://program-repair.org/tools.html

[37] Seemanta Saha, Ripon k. Saha, and Mukul r. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE/ACM, 13–24.

[38] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of ACM 10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 532–543.

[39] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[40] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 727–738.

[41] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*. 151–162.

[42] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. *arXiv preprint arXiv:2309.00608* (2023).

[43] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of IEEE/ACM 40th International Conference on Software Engineering*. 1–11.

[44] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of ACM 29th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 354–366.

[45] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery.

[46] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[47] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint*

arXiv:2304.00385 (2023).

[48] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[49] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. 2022. Attention: Not just another dataset for patch-correctness checking. *arXiv preprint arXiv:2207.06590* (2022).

[50] He Ye and Martin Monperrus. 2023. ITER: Iterative Neural Repair for Multi-Location Patches. *arXiv preprint arXiv:2304.12015* (2023).

[51] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward better evolutionary program repair: An integrated approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–53.

[52] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *arXiv preprint*

arXiv:2301.03270 (2023).

[53] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of IEEE/ACM 37th International Conference on Software Engineering (ICSE)*, Vol. 1. IEEE/ACM, 913–923.

[54] Wenkang Zhong, Chuanyi Li, Kui Liu, Tongtong Xu, Tegawendé F Bissyandé, Jidong Ge, Bin Luo, and Vincent Ng. 2023. Practical Program Repair via Preference-based Ensemble Strategy. *arXiv preprint arXiv:2309.08211 (to appear in ICSE'24)* (2023).

[55] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of ACM 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 341–353.