# A Quick Repair Facility for Debugging

Steven P. Reiss
Brown University
Providence, Rhode Island, USA
spr@cs.brown.edu

Qi Xin
Wuhan University
Wuhan, China
qxin@whu.edu.cn

## Abstract

We have developed ROSE, a tool to suggest quick-yet-effective repairs of semantic errors during debugging without a test suite. ROSE provides the functionality of automatic bug repair with the convenience of an auto-correction facility. It lets the developer describe the symptoms of a problem at a breakpoint during debugging. Then it does fault localization, repair generation, and repair validation without a test suite to present potential valid repairs to the developer.

## Keywords

Automatic Program Repair, Fault Localization, Repair Validation

**ACM Reference Format:**
Steven P. Reiss and Qi Xin. 2022. A Quick Repair Facility for Debugging. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/1122445.1122456

## 1 Introduction

Developers do software development using Integrated Development Environments (IDEs). Modern IDEs provide a variety of useful features including auto-correction for repairing simple errors such as Eclipse's Quick Fix. When a compiler syntax error arises, the developer can easily invoke Quick Fix to suggest repairs for that error, choose one, and make the correction. Quick Fix is not guaranteed to work in all cases but is widely used and liked by developers [7] because it can often find and repair errors quicker than the developer can.

Although useful, Quick Fix only tackles syntactic errors, those detected by a compiler. Developers also make semantic errors which are only exposed at run time and are much more difficult to repair [11]. This motivates our research – developing a technique that is similar in spirit to Quick Fix but targeting semantic errors.

Automatically repairing semantic errors is the goal of automated program repair (APR) [3]. Existing APR techniques cannot be applied easily to repairing semantic errors in an IDE because they are too slow for interactive use [5] and rely on a high-quality test suite for the code being repaired to define the problem, identify faults locations, and validate repairs [3].

In practice, high-quality test suites are rare. We scanned all Java projects in the large Merobase source repository. Over half have no tests at all. Only about 13% had as many tests per method as those used for evaluating APR. Moreover, running a large test suite for all potential repairs is time consuming, and what the developer is currently debugging might not be a formal test case. Addressing semantic error repair in an IDE requires novel techniques that can work efficiently without a test suite.

We have developed a prototype system, ROSE (Repairing Overt Semantic Errors), for repairing semantic errors while debugging Java programs in an IDE. ROSE assumes the developer has suspended the program in the debugger and has observed a semantic problem. ROSE first asks the developer to quickly describe the problem symptoms. Next, ROSE performs fault localization to identify potential responsible locations using a dynamic backward slice. For each identified location, ROSE uses a variety of approaches to suggest potential repairs. Next ROSE validates the potential repairs without a test suite by comparing the original execution to the repaired execution. It uses the comparison result to check if the identified problem was fixed and to avoid overfitted repairs [9]. Finally, ROSE presents the repairs to the developer as they are found in priority order. The developer can choose a repair to preview or have ROSE make the repair. Like Quick Fix, ROSE tries to suggest repairs quickly. It does not guarantee that all problems can be fixed automatically and focuses on simple repairs.

We evaluated our prototype of ROSE on two published error benchmarks, QuixBugs and a subset of Defects4J. Our result shows that ROSE was able to find repairs for a significant number of errors in only a few seconds. This result demonstrates the practicality of ROSE for interactive semantic error repair.

## 2 Overview of ROSE

Automated program repair (APR) techniques typically use a generate-and-validate approach based on a test suite. The techniques first perform fault localization to identify locations containing the error, generate potential repairs for these locations, and finally validate these repairs. ROSE uses a similar approach, but does so without a test suite. It is designed to work in conjunction with an IDE and a debugger. An overview of the process can be seen in Figure 1.

ROSE assumes the developer is using the debugger and the program is suspended with observed *problem symptoms* caused by a semantic *error* in the program. The developer invokes ROSE at the line where the program stopped. ROSE starts by asking the developer to quickly specify the problem symptoms. It does this by querying the debugger to get information about the stopping point and popping up a dialog. If the program is stopped due to an exception, ROSE assumes that the exception is the problem; if the program is stopped due to an assertion violation, it assumes that is the problem. Otherwise, the developer can indicate that a particular variable has the wrong value or that execution should not reach this line. The latter could arise if the code includes defensive checks for unexpected conditions.

*ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA*
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

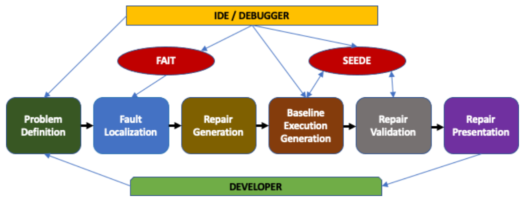

**Figure 1: Overview of ROSE (horizontal black arrows indicate sequencing; the other blue arrows represent data flow; the rounded boxes show ROSE's stages; the oval boxes show subsystems; and the rectangle boxes represent external environments).**

After the problem is defined, the developer asks ROSE to suggest *repairs*, code changes that will fix the problem causing the symptom. ROSE first does fault localization. Rather than using a test suite for this purpose, ROSE uses the location where the program is suspended along with the problem symptoms to identify potential lines to repair. It does this using program flow analysis to compute a partial backward dynamic slice from the stopping point, which it obtains from a full dynamic slice [1] using the problem symptoms, execution state, and execution distance to limit the result.

ROSE next generates potential repairs. Like Quick Fix, ROSE supports pluggable repair suggesters. It uses pattern-based, search-based, and learning-based suggesters to quickly find simple, viable repairs. For each proposed repair, these suggesters also provide a description of the repair and a syntactic priority approximating its likelihood of being correct.

The next step is to create a baseline execution that duplicates the original problem as a foundation for validating suggested repairs. If the execution leading to the problem is simple, for example, derived from a quick test case, then it can be used directly. However, in many cases, the execution is more complex and might have involved user or external events. In these cases, it is more efficient and practical for ROSE to consider only the execution of an error-related routine on the current call stack and everything it calls. ROSE uses SEEDE [8], a live-programming facility, to create a complete execution history of the identified routine. Finally, ROSE finds the current stopping point in the execution history.

ROSE next validates the suggested repairs using the baseline execution. ROSE does validation using a novel technique that does a step-by-step comparison between the baseline execution of the original program and the corresponding execution of the repaired program. This comparison checks if the problem was fixed, and attempts to find a repaired run that is close to the original to minimize overfitting [2]. It again uses SEEDE, making the proposed repair as a live update and getting the resultant execution history. The comparison yields a semantic priority score that is used in conjunction with the syntactic priority derived from repair generation to create a final priority score for the repair.

Finally, ROSE presents the potential repairs to the developer as in Quick Fix and previous interactive test-suite based APR tools (e.g., [4]). It limits this presentation to repairs that are likely to be correct by showing the repairs in priority order. The repairs are displayed as they are validated. In this way, the developer can preview or make a repair as soon as it is found.

## 3 Evaluation

We implemented a prototype of ROSE that worked with Eclipse and chose two widely used benchmarks for APR, QuixBugs [10] and a subset of Defects4J (v2.0.0) [6], to evaluate its repair efficacy. QuixBugs is a set of 40 student programs containing typical programming errors. Defects4J is a corpus of more complex development errors.

Our evaluation attempts to answer the question: *Is it possible to do program repair interactively while debugging without a test suite?* To do this we looked at how many errors can be repaired and how quickly can this be done. As ROSE resembles existing APR tools, we evaluated it along similar lines. We note that this is not a direct comparison as the inputs, outputs, and goals of ROSE differ from those of traditional program repair tools.

For each benchmark, we created an Eclipse project including the buggy code. We created a main program for each bug that effectively ran a failing test and used the failure to create a problem symptom, generally either an exception or an assertion failure. Then we asked ROSE to try to fix the problem and measured the time it took.

The results showed the effectiveness and practicality of the approach. For QuixBugs, ROSE found repairs for 17 of the 40 bugs with a median total time of ~5 seconds. Of the repairs generated, 14 were the top ranked repairs, two of them were ranked second, and one ranked fourth. For Defects4J, it found repairs for 16 of the 32 bugs with a median repair time of 29 seconds and a median time to find and report the correct repair of 7 seconds. The best prior results for QuixBugs was repairing 11 bugs with a median time of 14-76 seconds. For the Defects4J subset, prior APR results had fixed 4-18 bugs, with a median time generally over 4 minutes.

ROSE is available as open source from GitHub in repositories stevenreiss/{rose,seede,fait}. A video showing ROSE in action can be seen at https://youtu.be/GqyTPUsqs2o.

## References

[1] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN PLDI 1990*. ACM, 246–256.
[2] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification (CAV)*. Springer, 383–401.
[3] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* (2019), 56–65.
[4] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 266–276.
[5] 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 (ICSE)*.
[6] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* (2017), 1936–1964.
[7] Gail C Murphy, Mik Kersten, and Leah Findlater. 2006. How are Java software developers using the Eclipse IDE? *IEEE software* (2006), 76–83.
[8] 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 (ASE)*. 270–281.
[9] 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 the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*. 532–543.
[10] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software (JSS)* (2021), 1–15.
[11] Weiqin Zou, Xin Xia, Weiqiang Zhang, Zhenyu Chen, and David Lo. 2015. An empirical study of bug fixing rate. In *2015 IEEE 39th Annual Computer Software and Applications Conference*. IEEE, 254–263.