Quick Repair of Semantic Errors for Debugging

Steven P. Reiss[a], Xuan Wei[b], Qi Xin[b*]
[a] Department of Computer Science, Brown University, USA
[b] School of Computer Science, Wuhan University, China
spr@cs.brown.edu, {isabel1015, qxin}@whu.edu.cn

***Abstract*—Current automatic program repair (APR) approaches typically require a high-quality test suite and can take considerable time. Developers need to fix program problems during development from within an IDE. They generally do not have a high-quality test suite or a lot of time to find and fix problems. We explore a new approach to APR that is more practical and can be used within an IDE. Our prototype system, ROSE, is invoked while debugging and does not require a test suite. Our initial evaluations confirm its effectiveness and utility.**

I. Introduction

Our goal is to make automatic program repair (APR) practical and widely used. We envision APR becoming part of an Integrated Development Environment (IDE) and being routinely used by programmers to fix errors during system development and testing.

Current APR techniques are not well-suited to be integrated into an IDE for debugging. They typically require an extensive, high-quality test suite to produce effective repairs. A high-quality test suite however is generally unavailable. As suggested by [1], [2], developers often do not write test suite containing a sufficient number of test cases if they write tests at all. Also, it has been shown that test-based APR techniques are inefficient [3]. A small number of APR techniques work without a test suite. However, they are still slow and are severely limited, either by being restricted to specific types of bugs [4], [5] or requiring other forms of specification such as bug reports [6].

We wanted to develop a system that would let developers use APR techniques to fix semantic errors much as how Eclipse's Quick Fix [7] and Visual Studio's Auto Correct [8] handle syntactic errors. Such a system needs to be reasonably fast and work for a variety of errors. As with Quick Fix, it does not have to be perfect, and should fail quickly when it cannot find a fix.

To demonstrate that such an approach is possible and a viable direction for APR research, we are developing a prototype system, ROSE (Repairing Overt Semantic Errors), that can be integrated into an IDE to provide semantic error correction suggestion. ROSE differs from existing APR techniques in that it does not require a test suite, that it does not focus on specific types of bugs, and that it uses non-test-based, efficient strategies to localize error and produce and validate patches. ROSE starts by interacting with the developer to obtain a quick description of the problem symptom showing what is wrong with the current program behavior. Based on this description, ROSE follows a generate-and-validate procedure to make repair suggestions by performing fault localization, patch generation, and patch validation.

For fault localization, it uses a flow analysis to build a dynamic backward slice. ROSE uses an combination of pattern-based, search-based, and deep-learning-based patch generation approaches to generate candidate patches for each location. It validates the patches and sorts them using a novel approach that compares full execution traces of the original program and the patched program, accounting for the problem symptom and various matching conditions. Finally, it presents the repairs as they are found in priority order.

We evaluated ROSE on two published benchmarks, QuixBugs [9] and a subset of the Defects4J errors [10]. We found that ROSE is competitive with existing repair techniques in terms of finding correct repairs, and could do so in well under a minute (often seconds). We also did a user study with 26 participants recruited for performing four debugging tasks. The results showed that ROSE is helpful for debugging.

ROSE is available as open source at https://github.com/StevenReiss/rose. A video showing ROSE in action can be seen at https://youtu.be/GqyTPUsqs2o.

II. Overview of ROSE

ROSE is designed to work in conjunction with an IDE and a debugger. It performs six steps to generate repair suggestions without test suite: *problem definition*, *fault localization*, *repair generation*, *base line execution generation*, *repair validation*, and *repair presentation*. We next discuss each step in turn.

Problem definition. ROSE assumes the developer is using the debugger and the program is suspended with an observed *problem* caused by a semantic *error*. The developer invokes ROSE at the line where the program stopped. ROSE starts by asking the developer to quickly specify the problem *symptoms*, which are the observed problems caused by the error (not the error itself). In the absence of test suite, the defined problem can help localize the error and validate patches.

To obtain problem symptoms, ROSE queries 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.

* Corresponding author

2023 IEEE/ACM International Workshop on Automated Program Repair (APR) | 979-8-3503-0214-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/APR59189.2023.00008

979-8-3503-0214-1/23/$31.00 ©2023 IEEE
DOI 10.1109/APR59189.2023.00008

Fault localization. After the problem is defined, the developer asks ROSE to suggest *repairs*, code changes that will fix the original problem. ROSE first does fault localization to identify where the error might be. In the absence of test suite, ROSE uses an abstract interpretation-based flow analysis to statically compute a partial backward dynamic slice from the stopping point to identify potential lines to repair. The slice is partial because ROSE takes into account the problem definition and the current execution environment to create an accurate slice for this particular situation, and then limits the slice based on execution distance from the stopping point.

Repair generation. ROSE next generates potential repairs for each identified location. ROSE supports pluggable repair suggesters. It currently provides pattern-based, search-based, and learning-based suggesters to quickly find simple, viable repairs. In addition to generating a repair, ROSE also provides a description of the repair and a syntactic priority approximating its likelihood of being correct.

Baseline execution generation. The next step is to create a baseline execution that duplicates the original problem as a foundation for validating suggested repairs. This is represented as a full program trace including both control and data flow created by a live-programming facility without rerunning the system. In general, a full problem-duplicating execution might have involved user or external events and can be too complex to obtain. To ensure efficient and practical repair validation, ROSE considers only the execution of an error-related routine on the current call stack and everything it calls. ROSE identifies a suitable error-related routine for which a partial problem-duplicating execution including the potential fault locations would be relatively easy to create, obtains the complete execution history of that routine, and finally finds the current stopping point in the execution history to obtain the partial execution.

Repair validation. ROSE next validates the suggested repairs using the baseline execution. This is done by comparing the full trace of the baseline execution with the corresponding simulated trace of each repaired program. ROSE takes into account the problem symptom and a variety of matching situations to compute a semantic priority score approximating the likelihood of the problem being fixed and the repair being non-overfitting. The semantic priority score is used in conjunction with the syntactic priority derived from repair generation to create a final priority score for the repair.

Repair presentation. Finally, ROSE presents the potential repairs to the developer. It limits this presentation to repairs that are likely 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.

III. Evaluation

We evaluated ROSE on all 40 QuixBugs errors and a set of 32 Defects4J (v2.0.0) errors that are relevant with ROSE's assumptions on repair simplicity. These are errors whose repairs involve one-line changes and are close (execution-wise) to the stopping point based on a predefined threshold. For each error, we created an Eclipse project including the buggy code, a main program that effectively ran a failing test, and a problem symptom based on the failure. Then we asked ROSE to try to fix the problem and measured the time it took.

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 time to find and report the correct repair of ~7 seconds. For 12 of the bugs, the correct repair was top ranked. Two other correct repairs were second ranked, and then one correct repair was third ranked and one was fourth. 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.

We also did a user study to evaluate ROSE's utility. We recruited 26 student participants, assigned them to two groups to perform 4 debugging tasks with and without ROSE, and compared their performance. These 4 tasks are created based on errors selected from the benchmarks. Our results showed that ROSE helped 44.6% more participants find the correct repair and that participants who did not use ROSE spent 56.9% more time for debugging. The feedback given by participants showed that they find ROSE useful and they like ROSE.

Overall, our research shows that APR within an IDE without a test suite (or even a test case) is both possible and practical. We plan continuing research to expand ROSE and to investigate how such an approach could become widely used.

Acknowledgement: This work was partially supported by the National Natural Science Foundation of China under the grant numbers 62202344 and 62141221.

References

[1] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *ICQS*, 2013, pp. 103–112.

[2] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *FSE*, 2015, pp. 179–190.

[3] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, "On the efficiency of test suite based program repair," in *ICSE*, 2020, pp. 615–627.

[4] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *TOSEM*, pp. 1–27, 2021.

[5] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *ICSE*, 2018, pp. 151–162.

[6] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "iFixR: Bug report driven program repair," in *FSE*, 2019, pp. 314–325.

[7] (2023) Eclipse's Quick Fix. [Online]. Available: https://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F

[8] (2023) Visual Studio's Auto Correct. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=sygene.auto-correct

[9] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *SIGPLAN Companion*, 2017, pp. 55–56.

[10] 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.