2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Towards Addressing the Patch Overfitting Problem Extended Abstract

Qi Xin
Advisor: Steven P. Reiss
Department of Computer Science, Brown University
Providence, RI, USA
qx5@cs.brown.edu

***Abstract*—Current automatic program repair techniques often produce overfitting patches. Such a patch passes the test suite but does not actually repair the bug. In this paper, we propose two techniques to address the patch overfitting problem. First, we propose an automatic repair technique that performs syntactic code search to leverage bug-related code from a code database to produce patches that are likely to be correct. Due to the weak and incomplete program specification encoded in the test suite, a patch is still possible to be overfitting. We next propose a patch testing technique which generates test inputs uncovering the semantic differences between a patch and its original faulty program, tests if the patch is overfitting, and if so, generates test cases. Such overfitting-indicative test cases could be added to the test suite to make it stronger.**

I. Introduction

Given a faulty program and a fault-exposing test suite, an automatic program repair technique [1]–[9] aims to produce a patch that passes the test suite and is correct in general. Studies have shown that automatic repair techniques often produce overfitting patches [10], [11]. As [10] shows, the majority of patches generated by GenProg [1], AE [2] and RSRepair [3] are incorrect. Within a 12-hour time limit, the state-of-the-art repair techniques SPR [4] and Prophet [5] generated patches for less than 60% bugs with more than 60% of the first found patches being incorrect. A recent study [12] suggests a repair technique to leverage information beyond the test suite to create a *targeted* search space where a correct patch could be effectively identified. One idea is to leverage existing code that is likely to be correct from a code database. SearchRepair [7] and Code Phage [13] (or CP) are two representatives of this idea which try to find fix code through semantic code search. However, semantic code search is often expensive and time-consuming, and scalability seems to be a concern.

We propose a novel repair technique ssFix which performs syntactic code search to find code fragments that are similar to the context of the bug and leverages the syntactic differences between such a code fragment (as the *candidate*) and the bug (with its context) to produce patches. For a similar candidate, the syntactic differences are small. If the candidate contains the fix, the search space of patches would be targeted, and a correct patch can be effectively identified.

In addition to the search space problem, the quality of the test suite itself is another important reason why an overfitting patch occurs. Unlike a formal specification, the correctness encoded in a test suite is often weak and incomplete. To address the problem, we propose a patch testing technique DiffTGen that identifies a patch to be overfitting with new test cases generated. Given a faulty program and a patch, DiffTGen employs a test generator to generate new test inputs uncovering the semantic differences between the two programs. DiffTGen tests the patch based on the semantic differences (for which it needs an oracle) and produces a test case showing the patch is overfitting. Such a test case, if generated, could be added to the original test suite to make it stronger.

The two techniques can be combined to form a repair system which repeats calling ssFix and DiffTGen for patch generation and patch testing. Our hypothesis is that the repair system can effectively mitigate the patch overfitting problem.

II. The Two Techniques

In this section, we elaborate on the two techniques proposed.

A. Automatic Program Repair Technique

The automatic repair technique ssFix works in four stages: *fault localization*, *code search*, *patch generation* and *patch validation* to produce a patch. In the first stage, ssFix employs a fault localization technique (we use GZoltar [14]) to identify a suspicious statement that is likely to be faulty to work on. In the second stage, ssFix generates a code chunk for the statement including the statement itself and its local context as a buggy chunk, or $bchunk$. ssFix extracts structural and conceptual tokens from the $bchunk$'s textual content and employs Lucene [15] to search for code chunks containing "similar" tokens using the tf-idf vector space model from a code database consisting of the local code project and an external large code repository. ssFix considers such similar code chunks as the candidates, or $cchunk$s, and tries each to produce patches for $bchunk$. In the third stage, ssFix leverages a $cchunk$ to produce patches for $bchunk$ in three steps: *candidate translation*, *component matching* and *modification*. In the first step, ssFix translates $cchunk$ by unifying the identifier names in the chunk with those in $bchunk$. In the second step, ssFix matches the components (statements and expressions) between the chunks to establish their correlation and thereby to identify the syntactic differences which may suggest the correct repair. In the third step, ssFix uses three types of modifications: replacement, insertion and deletion, to

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.42

produce patches based on the differences. In the final stage, ssFix validates the generated patches against the test suite and reports the first patch, if any, that passes the test suite.

We evaluated ssFix on a subset of the Defects4J bug dataset [16] containing 93 simple bugs whose fixes require local modifications within a method and empirically found that ssFix is more effective than three comparing techniques for Java: jGenProg (the Java version of GenProg), HDRepair [9] and NoPol [8] in producing more correct patches with the highest non-overfitting rates ($\#CorrectPatches/\#PlausiblePatches$). ssFix produced correct patches for 7 bugs from the Closure-Compiler project (for which manual fault localization was performed) and 15 bugs from the other projects with the non-overfitting rates being 70% and 71.4% respectively.

B. Patch Testing Technique

The patch testing technique DiffTGen accepts as input a faulty program fp, a patched program pp, the syntactic differences Δ between the two programs, and an oracle as input. As output, it either produces a test case showing pp is overfitting or nothing if no such test case could be found. DiffTGen works in three stages to generate a test case: *Test Target Generation*, *Test Method Generation* and *Test Case Generation*.

In the first stage, DiffTGen produces a test target program tp based on fp, pp and Δ with the target statements, or $tstmt$s, in tp specified. For each $\delta \in \Delta$, DiffTGen specifies a $tstmt$ in tp. tp is the actual program on which an test generator works to generate test inputs. A test input that is generated later has to exercise some $tstmt$ in tp to be able to uncover any semantic difference between fp and pp. tp preserves the semantics of pp and is initially a copy of pp. Depending on a specific δ, $tstmt$ could be as simple as a statement (e.g., a modified assignment) in tp or a simple, dummy statement DiffTGen inserts in tp (e.g., for an assignment deleted in pp, a dummy statement is inserted in tp at the deleting location). When δ is related to an if-statement which is common [17], [18], DiffTGen may produce a "forking" if-statement inserted in tp with a dummy statement as its then-branch being $tstmt$. A test input exercising $tstmt$ would yield different branch-taking behaviors related to a modified if-statement between fp and pp. In the second stage, DiffTGen employs a test generator (we use EvoSuite [19]) to generate test methods (as test inputs) that can exercise some $tstmt$ in tp. DiffTGen then creates instrumented versions of fp and pp as fp' and pp'. For any test method, it runs fp' and pp' against the test method to obtain outputs and compares them to check whether any semantic difference exists. If so, in the third stage, DiffTGen finds out which values are different and asks the oracle which is correct. If the output of pp is incorrect and the oracle could provide the correct output, DiffTGen produces a test case showing the patch is overfitting (DiffTGen produces another instrumented version of fp as fp'' that is semantics preserving. For testing, one needs to run fp'' against the test case.)

We empirically evaluated DiffTGen on 89 patches (including 79 patches that are likely to be incorrect) for the Defects4J bugs generated by four automatic repair techniques jGenProg, jKali (the Java version of Kali [10]), NoPol and HDRepair. DiffTGen identified 39 patches to be overfitting with test cases generated (we used the fixed version of a bug from the Defects4J dataset as the oracle). The average running time is about 7 minutes. We further configured DiffTGen with each repair technique for patch generation. Our results show that with the assistance of DiffTGen, the repair techniques avoided generating any overfitting patches for 36 of the 39 bugs. There were correct patches (as the first found ones) generated for the bug *Math_50* using HDRepair and DiffTGen.

III. Conclusion

To address the patch overfitting problem, we proposed an automatic repair technique ssFix and a patch testing technique DiffTGen. We demonstrated the effectiveness of ssFix and DiffTGen and the feasibility of combining DiffTGen with a repair technique (as a repair system) to enhance its performance.

References

[1] C. L. 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 *ICSE*, 2012, pp. 3–13.

[2] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: models and first results," in *ASE*, 2013, pp. 356–366.

[3] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, 2014, pp. 254–265.

[4] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *ESEC/FSE*, 2015, pp. 166–178.

[5] ——, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

[6] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016, pp. 691–701.

[7] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *ASE*, 2015, pp. 295–306.

[8] J. Xuan, M. Martinez, F. Demarco, M. Clment, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, 2016.

[9] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016, pp. 213–224.

[10] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ISSTA*, 2015, pp. 24–36.

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

[12] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *ICSE*, 2016, pp. 702–713.

[13] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *PLDI*, 2015, pp. 43–54.

[14] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *ASE*, 2012, pp. 378–381.

[15] "Apache lucene," https://lucene.apache.org.

[16] 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.

[17] K. Pan, S. Kim, and E. J. Whitehead Jr, "Toward an understanding of bug fix patterns," *ESE*, pp. 286–315, 2009.

[18] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *ESE*, vol. 20, no. 1, pp. 176–205, 2015.

[19] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *ESEC/FSE*, 2011, pp. 416–419.