SCIENCE CHINA
Information Sciences

October 2019, Vol. 62 200103:1–200103:3
https://doi.org/10.1007/s11432-019-9947-6

• PERSPECTIVE •
Special Focus on Software Automation

Automated program repair: a step towards software automation

Abhik ROYCHOUDHURY[1*] & Yingfei XIONG[2,3*]

[1]*School of Computing, National University of Singapore, Singapore* 117417*, Singapore;*
[2]*Key Laboratory of High Confidence Software Technologies* (*Peking University*)*, Ministry of Education, Beijing* 100871*, China;*
[3]*Institute of Software, Department of Computer Science, Peking University, Beijing* 100871*, China*

Received 26 February 2019/Revised 7 May 2019/Accepted 23 May 2019/Published online 9 September 2019

Citation Roychoudhury A, Xiong Y F. Automated program repair: a step towards software automation. Sci China Inf Sci, 2019, 62(10): 200103, https://doi.org/10.1007/s11432-019-9947-6

Programming is seen as a problem solving activity, which combines precision with creativity. The program needs to be precise at least to the extent of passing given tests. At the same time, the programmer employs copious creativity in terms of problem solving strategies, algorithm design, data structure choice, or even choice of which libraries to invoke. The recent growth of machine learning techniques and the possibility of applying such techniques to large software repositories raise the question to what extent the various software engineering processes can be automated, which is known as the software automation problem [1].

It is known that for many software engineering projects up to 80% of the time is spent in debugging and fixing errors. This is an unfortunate narrative on the state-of-practice in software development, prompting practitioners to label the situation as a legacy crisis a decade back [2]. Since then, the scale of software has increased, and the use of third party code, or geographically distributed software development has also dramatically increased. It is indeed not an exaggeration to say that today's software systems are often not monolithic. Instead, they are assembled out of software components written by various geographically distributed teams, legacy software components, and third party software components purchased or acquired for free. In the absence of strong oversight, the challenges of debugging and fixing are exacerbated. This makes the prospect of automated program repair particularly attractive in future software development.

Classic automated repair techniques aim to modify a buggy program to meet a given correctness criterion; the correctness criterion is often given as a test-suite. Classic program repair typically proceeds with three steps: (i) fix localization, (ii) fix representation, and (iii) fix selection, as detailed below.

Step 1. Fix localization attempts to find program locations where the code may be changed to achieve the fix.

Step 2. A space of candidate patches is represented. The representation is often based on meta-level techniques, such as program transformation operations, and/or grammars to constrain the newly generated code pieces.

Step 3. The repair system selects a candidate patch from the space to satisfy the correctness requirement. Typical methods to perform the selection include heuristic search [3] and program synthesis with symbolic execution [4, 5].

In fixing program errors, a key issue is the enunciation of the correctness requirement. Because formal specifications of intended program behavior are typically unavailable, the correctness criteria driving program repair are given by test-suites. This presents a challenge for repair approaches because the generated patches can break tests not

* Corresponding author (email: abhik@comp.nus.edu.sg, xiongyf@pku.edu.cn)

appearing in the given test-suite, or even introduce new errors in the un-tested or under-tested program functionality. This problem is known as "weak specification", "weak test-suites", or "overfitting", and is often considered as one of the most important challenges faced by the program repair research [6, 7].

In this study, we discuss two possible directions to address the weak specification problem.

Repair with a reference implementation. Though a formal specification of program behavior is often unavailable, in many situations there exists a reference implementation of the program to demonstrate the desired behavior. In the case of an industrial standard for a program or a protocol, the organization responsible for the standard usually publishes a reference implementation to show how the standard should be implemented, and other companies try to be compatible with the reference implementation. For example, OpenJDK is the reference implementation for the Java programming language, and Oracle JDK and Jikes RVM are compatible with OpenJDK. Similarly, reference implementations usually exist for decoders/encoders for multimedia files, network protocols, encryption algorithms, etc.

Different from a formal specification, a reference implementation specifies full execution behavior. Essentially the reference implementation acts as an informal specification of intended behavior. The program under repair does not have to be repaired to be fully identical to the reference implementation, but only the observable behavior should be equivalent after repair. To capture such behavior, one possibility is to generate test cases from the reference implementation. One could employ automated test generation methods such as symbolic execution to automatically generate test inputs. Dynamic symbolic execution engines such as KLEE [8] can compute the set of inputs driving execution along a program path as a logical formula called a path condition. Subsequently the path condition for a random input can be mutated to drive execution along other paths and thereby generate inputs traversing these paths. Symbolic execution can be used to systematically generate a comprehensive test-suite from the reference implementation, driving repair.

However, symbolic execution engines leverage constraint solving and SMT solvers as the backend, which leads to scalability challenges for symbolic execution. Several possibilities exist to address this problem. First, one can guide symbolic execution to reach specific targets as embodied by efforts such as [9]. This can help generate test cases stressing un-tested functionality and subsequently these tests can be used to guide program repair. Second, systematic grey-box fuzzing methods which attempt to achieve enhanced path coverage such as AFLFast [10] can be used. Grey-box fuzzing methods employ compile-time instrumentation followed by run-time detection of enhanced coverage of control flow artifacts. With the recent push on making grey-box fuzzing methods systematic, there exists an opportunity to generate abundant test-data for driving program repair. Last but not the least, symbolic execution techniques specifically designed for exposing the behavioral difference between two programs could be developed, such as the one by Mechtaev et al. [11].

Repair with big data. While obtaining the full specification of the program is difficult, estimating the likelihood of certain patches being correct in the represented space may be much easier for many types of bugs. For example, given a bug with an unexpected NullException, adding an "if" check to guard the statement is much more likely to be correct than deleting the respective statement. Therefore, another possible direction is to estimate the likelihood of the patches in the represented space, and select the most likely patch for the current context.

Towards this direction, recent program repair techniques often add an additional step compared with the classic techniques: fix prioritization. This step estimates the likelihood of the patches and prioritizes them. After adding this step, the goal of the fix selection step is to select the highest ranked patch that satisfies the (possibly weak) specification, such as passing a given test-suite.

Some existing techniques try to employ heuristic rules to rank the patches. These rules include minimizing the changes made by the patch with some measurement of change distance [12], anti-patterns [13], or checking if the execution of the tests change under some expected directions. However, heuristic rules are manually constructed by researchers, and by nature cannot cover all situations, especially when the probabilities of the patches depend on the local context of the project and the specific types of the bugs.

The availability of software big-data presents a unique opportunity for this problem. It remains an open question whether we can get useful guidance to estimate the likelihood of patches and correctly prioritize them via mining software big-data. By collecting a corpus of patches and training over the corpus, we may build a model to estimate the likelihood of patches. Identifying the most-likely patches with this model, we may repair bugs with a high probability of correctness. In this research direction, many challenges exist, and yet there are

many opportunities to address such challenges.

First, it is challenging to collect high-quality training data (patches for training). While patches can be found in the commit history of software projects, a commit may also add new functionalities, refactor code, or mix several purposes. The current approaches [14] use heuristics to identify bug-fixing commits, such as the number of modified code lines or keywords in commit message, but these heuristics all have limited precision and recall. A possible direction, as studied in a recent study [15], is to watch the development process and automatically identifies commits between a failing build and a passing build. Yet future work is still needed to identify which part of a big commit repairs the bug. Another opportunity is to learn from more sources beyond just patches. For example, existing approaches have utilized program source code [16] and QA web sites [17]. A remaining question is how to combine different sources to achieve the best performance.

Second, it is challenging to build a learning model for estimating the likelihood of patches. Existing approaches have applied classic machine learning models as well as deep learning to model the code [18, 19]. However, we still lack understanding on how different models perform at different situations. Furthermore, these models usually treat the likelihood estimation procedure as a black box, and cannot utilize the domain knowledge of the program, such as the semantics. These issues remain to be explored in future.

Third, it is challenging to identify the most probable patch that meets the weak specification. Though efficient methods exist to quickly identify patches passing the tests in a prioritized space [20, 21], it is often impossible to enumerate all possible patches and sort them by priority. Recent study [18] proposed to decompose a patch into a series of search steps, and instead of estimating the likelihood of patches, the likelihood of choices at each step is estimated. Future work needs to be done to understand how this method can be combined with semantic repair [4].

Acknowledgements This work was partially supported by Singapore's National Cybersecurity R&D Program (Grant No. NRF2014NCR-NCR001-21), National Key Research and Development Program of China (Grant No. 2017YFB1001803), and National Natural Science Foundation of China (Grant Nos. 61672045, 61529201).

References

1 Mei H, Zhang L. Can big data bring a break-through for software automation? Sci China Inf Sci, 2018, 61: 056101
2 Seacord R, Plakosh D, Lewis G. Modernizing Legacy Systems: Software Technologies, Engineering Processes and Business Practices. Boston: Addison Wesley, 2003
3 Weimer W, Nguyen T V, Goues C L, et al. Automatically finding patches using genetic programming. In: Proceedings of ICSE, 2009. 364–374
4 Nguyen H D T, Qi D W, Roychoudhury A, et al. SemFix: program repair via semantic analysis. In: Proceedings of ICSE, 2013. 772–781
5 Mechtaev S, Griggio A, Cimatti A, et al. Symbolic execution with existential second-order constraints. In: Proceedings of ESEC/FSE, 2018. 389–399
6 Qi Z C, Long F, Achour S, et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of ISSTA, 2015. 24–36
7 Smith E K, Barr E, Goues C L, et al. Is the cure worse than the disease? overfitting in automated program repair. In: Proceedings of FSE, 2015. 532–543
8 Cadar C, Dunbar D, Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI, 2008. 209–224
9 Marinescu P, Cadar C. Katch: high-coverage testing of software patches. In: Proceedings of ESEC-FSE, 2019. 235–245
10 Böhme M, Pham V T, Roychoudhury A. Coverage based greybox fuzzing as a markov chain. In: Proceedings of CCS, 2016. 489–506
11 Mechtaev S, Nguyen M D, Noller Y, et al. Semantic program repair using a reference implementation. In: Proceedings of ICSE, 2018. 129–139
12 Mechtaev S, Yi J, Roychoudhury A. Directfix: looking for simple program repairs. In: Proceedings of ICSE, 2015. 448–458
13 Tan S H, Yoshida H, Prasad M, et al. Anti-patterns in search-based program repair. In: Proceedings of FSE, 2016. 727–738
14 Just R, Jalali D, Ernst M D. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of ISSTA, 2014. 437–440
15 Dmeiri N, Tomassi D, Wang Y, et al. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In: Proceedings of ICSE, 2019. 339–349
16 Xiong Y F, Wang J, Yan R F, et al. Precise condition synthesis for program repair. In: Proceedings of ICSE, 2017. 416–426
17 Gao Q, Zhang H S, Wang J, et al. Fixing recurring crash bugs via analyzing Q&A sites (T). In: Proceedings of ASE, 2015. 307–318
18 Xiong Y, Wang B, Fu G R, et al. Learning to synthesize. In: Proceedings of GI, 2018. 37–44
19 Gupta R, Pal S, Kanade A, et al. DeepFix: fixing common C language errors by deep learning. In: Proceedings of AAAI, 2017
20 Mechtaev S, Gao X, Tan S H, et al. Test-equivalence analysis for automatic patch generation. ACM Trans Softw Eng Methodol, 2018, 27: 15
21 Wang B, Xiong Y F, Shi Y Q W, et al. Faster mutation analysis via equivalence modulo states. In: Proceedings of ISSTA, 2017. 295–306