SmartDebug: An Interactive Debug Assistant for Java

Xinrui Guo
School of Software, Tsinghua University, China
Key Laboratory for Information System Security, Ministry of Education
Tsinghua National Laboratory for Information Science and Technology(TNList)
gxr12@mails.tsinghua.edu.cn

ABSTRACT

Debugging has long been recognized as one of the most labour- and time- consuming activities in software development. Recent research on automated debugging tries to facilitate this process by automatically generating patches for buggy programs so that they pass a predefined test suite. Despite the promising experimental results, several major obstacles emerge when we apply these techniques in active coding process. Inadequate test cases, multiple errors in one program and possible bug categories overlooked by existing fix generation search spaces impede these techniques to perform at their best.

To overcome these obstacles, we designed an interactive usage paradigm that allows a developer to characterize his or her judgments of program running state and utilize such information to guide the fix generation process. We implemented a prototype of this design, an Eclipse plugin called SmartDebug as a debug assistant for Java programs. Experimental results show that SmartDebug helped to debug 15 out of 25 buggy programs successfully. All programs contain less than 3 test cases. In 14 programs it accelerated the debugging process compared to pure human debugging, while one of which contains 2 buggy statements. This indicates that the proposed usage paradigm is capable of facilitating the debugging process in active coding process.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

automated debugging; generate and validate systems; user-interaction

1. PROBLEM AND MOTIVATION

It is commonly recognized that debugging is one of the most labour- and time-comsuming activities in software development. Recent research on automated debugging techniques try to generate patches that directly fix the program so that they pass a certain test suite [5, 4, 7, 9, 3]. State of the art techniques in general take three steps. (1) Localize the buggy statement or expression by test coverage information. (2) Generate fixes according to predefined search space, which usually includes a set of fix patterns or rules. (3)Validate these fixes and see if any one of them can make the program pass the test suite. These approaches follow the "generate-and-validate" paradigm.

However, these techniques have only been experimented on mature software systems in their maintenance phase. When we apply them to improving debugging efficiency in active coding practice where new code are created rapidly, several major obstacles emerge. First, existing techniques rely on test coverage data for accurate fault localization. While mature systems are provided with a regression test suite with ample tests, newly created code usually lacks such test suites. As a result the fix generator may waste time fixing statements that are actually correct, which slows down the fix generation progress. Second, it is very likely that multiple errors exist simultaneously while existing techniques focus mostly on single spot fix. Finally, since many bugs that pop up in coding activities are eliminated before the code is submitted to a code repository, they are consequently neglected by existing studies on bug patterns. As a result, search space of existing studies needs to be expanded accordingly so as to cover these bugs.

Based on the above analysis, we believe that it is necessary to design a usage paradigm which incorporates and adjusts existing automated fix generation techniques so that they adapt to debugging in active coding process. Due to the fact that it is common practice for programmers to trace and observe the program execution process during debugging, we provide a "checkpoint" mechanism for programmers to describe their judgments to the fix generation system. This information is utilized by the system for more efficient fix generation, thus forming interactivity between user and the fix generator.

2. BACKGROUND AND RELATED WORK

Automated fix generating systems has drawn much attention in recent years. Given a buggy program and a test suite, these systems try to generate patches that enable the program to pass the tests. Known systems include GenProg[5, 4], RSRepair[14], AE[16], Angelic debugging[2], SemFix[12], DirectFix[11], SPR[7], Prophet[9] for C programs and Par[6], NOPOL[3] for Java programs. Kali[15] is designed as a function remover to prove that several systems above do not actually fix bugs.

The fixing capability of these systems relies heavily on

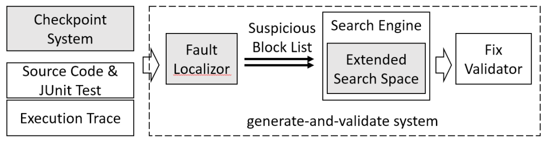

Figure 1: work flow of SmartDebug

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983971

predefined fix search space. K.Pan et al.[13] summarized a series of fix patterns at the granularity of AST elements in Java programs. F.Long et al.[8] systematically analyzed key characteristics of search space design and their influence in fix generation ability.

3. APPROACH AND UNIQUENESS

Figure 1 shows the basic work flow of SmartDebug. Similar with existing techniques, the center of SmartDebug is a "generate-and-validate" system. SmartDebug first localizes the bug by analyzing the coverage information of the test suite and measures the suspiciousness of each line of code. Then it generates possible fixes within a predefined search space following the suspiciousness rank of the code lines. Finally the generated fixes are validated by running the patched program against the test suite. However, to overcome the obstacles stated in Section 1, several innovations are made upon this work flow.

Checkpoint System. A "checkpoint" system is introduced for users to describe their judgments of the program running status. Each "checkpoint" is anchored to a single test case at a specific point of program execution and either claims the program to be running correctly (a "passing" checkpoint) or describes the correct values of the erroneous variables (a "failing" checkpoint). SmartDebug allows users to add checkpoints as they examine the program running process using a debugger.

Benefits of this design are twofolds. First, as programmers understand their own code, they may narrow down the scope of the bug effortlessly, thus substantially reducing the time cost for fix generation. Secondly, SmartDebug allows users to set multiple checkpoints simultaneously. Through management of fix goals in each fix session, it tries to resolve one failing checkpoint at a time so that multiple bugs can be fixed iteratively. This complements to existing techniques which mainly focus on single point fixes.

Adaptive Fault Localization. We adopt the Ochiai metric in fault localization process based on empirical study [1]. For every basic block b covered in the test execution process, we count the number of test cases that hit the block and failed as a_{hf}, missed the block and passed as a_{mp} and finally missed the block and failed as a_{mf}. Then the suspicious value of the block is calculated as follows:

$$Suspicious(b) = \frac{a_{hf}}{\sqrt{(a_{mf} + a_{hf})(a_{hf} + a_{mp})}}$$

According to this value all basic blocks are ranked into a candidate list with the most suspicious block at the top.

Though research have demonstrated the effectiveness citation of Ochiai, we find that inadequate test cases causes unsatisfactory localization accuracy. Therefore we utilize the input of the checkpoint system to refine the localization results. For each test case that contains at least one passing checkpoint, we split the execution trace at the last passing checkpoint into two traces, then recalculate the suspiciousness rank and finally filter out only the code lines between the last passing checkpoint and the first failing checkpoint. This greatly narrows down the scope of the error and improves accuracy of fault localization.

Fix Generation Search Space. Referring to existing studies [8, 13, 6, 10], our search space includes fixes on general if-conditions, inserting null-checkers and cast-checkers, inserting (conditional) `break`/`continue`/`return` statements, modifying method parameters and method names, general fixes of expressions.

However, previous studies on search spaces are basically based on analysis of differences between historical versions in large software repositories. In practice bugs that have once occurred and manifest easily are already resolved before submission to code repositories. By this observation, we introduced a series of structural fixes to complement the existing search space. These fixes include (1)changing order of execution of statements, (2)moving statements between branches, (3)moving statements in and out branches, (4)moving statements in and out of a loop.

4. RESULTS AND CONTRIBUTIONS

We implemented this design as an Eclipse plugin named SmartDebug for Java programs. To evaluate how SmartDebug is able to improve debug efficiency in active coding process, we first collected bugs that appear in the actual coding process before the code is committed to a repository. We recorded the editing history of 25 1st year graduate students majoring in software engineering working on 3 coding tasks in a programming exam. Of all the later recovered buggy versions of the programs, 25 versions were selected as a test bench in our experiment as they implemented most of the program correctly but still needed further correction.

To see if SmartDebug can really improve debug efficiency, we invited students of the same level to debug these buggy versions and recorded the time they needed with and without SmartDebug. Results presented in Table 1 show that SmartDebug is able to assist in generating correct fix suggestions for 15 of the buggy versions in acceptable time. On 14 versions of programs SmartDebug beats human in the debugging speed. Generally SmartDebug generates at least one correct fix suggestion in less than 5 minutes. As time is limited in the programming exam, all versions contain less than 3 test cases. Fault localization is thus greatly enhanced with help of the checkpoint system.

The bugs in the collected programs include mis-usage of local variables, condition expression errors, wrong initialization of loop variables which are covered in search space of existing work. However version No.6 requires structural modification that is first proposed in this paper. One possible reason is that existing work focus on fixing bugs from mature code repositories. Structural errors are generally very easy to be discovered before commitment. However they may not be easy to fix, as in our case the participant took 730 seconds to solve the bug in version No.6, 7 times longer than with the help of SmartDebug. We believe that observing and analyzing characteristics of bugs that appear in intensive coding process may bring new knowledge to existing research. Also, version No.2 contains two bugs in one program. However as the two bugs appeared sequentially in program computation, by using the checkpoint system the participant is able to resolve the two bugs iteratively.

To sum up, major contributions of this paper include:(1) A novel interactive design paradigm of incorporating automated fix generation techniques to improve debug efficiency in active coding practice. (2) An expanded search space for fix generation systems. (3) An Eclipse plug-in prototype "SmartDebug" as an interactive debug assistant. (4) Experimental comparison of debugging with and without assistance of SmartDebug to show that automated fix generation techniques can actually facilitate debugging activities in the coding process. To the best of our knowledge, this is the first quantitative proof of the capability in accelerating debug process of "generate-and-validate" systems.

Table 1: SmartDebug(SD) v.s. Human(H)

No.	SD(s)	H(s)	No.	SD(s)	H(s)
1	86	1020	9	183	476
2	282	300	10	154	773
3	95	691	11	70	292
4	**1055**	**694**	12	215	829
5	260	423	13	22	515
6	93	730	14	33	840
7	309	410	15	228	600
8	198	341			

5. REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780 – 1792, 2009. SI: {TAIC} {PART} 2007 and {MUTATION} 2007.

[2] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 121–130. IEEE, 2011.

[3] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM, 2014.

[4] 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 *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13, June 2012.

[5] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan 2012.

[6] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[7] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[8] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 702–713, New York, NY, USA, 2016. ACM.

[9] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312. ACM, 2016.

[10] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[11] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 448–458. IEEE, 2015.

[12] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[13] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[14] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, New York, NY, USA, 2014. ACM.

[15] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems (supplementary material). 2015.

[16] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366, Nov 2013.