2018 ACM/IEEE 40th International Conference on Software Engineering: Companion Proceedings

Poster: Automated Program Repair with Canonical Constraints

Andrew Hill
North Carolina State University
Raleigh, North Carolina
ahill6@ncsu.edu

Corina S. Păsăreanu
Carnegie Mellon University CyLab
and NASA Ames
Mountain View, California
Corina.S.Pasareanu@nasa.gov

Kathryn T. Stolee
North Carolina State University
Raleigh, North Carolina
ktstolee@ncsu.edu

ABSTRACT

Automated program repair (APR) seeks to improve the speed and decrease the cost of repairing software bugs. Existing APR approaches use unit tests or constraint solving to find and validate program patches. We propose Canonical Search And Repair (CSAR), a program repair technique based on semantic search which uses a canonical form of the path conditions to characterize buggy and patch code and allows for easy storage and retrieval of software patches, without the need for expensive constraint solving. CSAR uses string metrics over the canonical forms to cheaply measure semantic distance between patches and buggy code and uses a classifier to identify situations in which test suite executions are unnecessary–and to provide a finer-grained means of differentiating between potential patches.

We evaluate CSAR on the IntroClass benchmark, and show that CSAR finds more correct patches (96% increase) than previous semantic search approaches, and more correct patches (34% increase) than other previous state-of-the-art in program repair.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**;

KEYWORDS

Symbolic Execution, SPF, Program Repair, Semantic Repair

ACM Reference format:
Andrew Hill, Corina S. Păsăreanu, and Kathryn T. Stolee. 2018. Poster: Automated Program Repair with Canonical Constraints. In *Proceedings of 40th International Conference on Software Engineering Companion, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18 Companion),* 3 pages.
https://doi.org/10.1145/3183440.3194999

1 INTRODUCTION

Given a faulty program and a test suite revealing the fault, automated program repair aims to fix the fault without human intervention. Current methods in Automated Program Repair (APR) fall into one of three categories: generate-and-validate (e.g., [6, 8, 14, 17]), synthesis-based (e.g., [1, 7, 9–12]), and reuse-based (e.g., [5, 18]).

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3194999

This work is closest to reuse-based program repair, which take advantage of high levels of code duplication to generate human-readable, high-quality patches by using databases of previously-written code (e.g., [5, 18]). These have shown promise in generating high-quality patches on defects which are often orthogonal to those the other approaches patch well [5]. However, at present ssFix [18] can perform only syntax matching and SearchRepair [5] (SR) requires expensive SMT queries to find semantic matches.

We propose Canonical Search And Repair (CSAR), a reuse-based APR approach which constructs human-readable, correct repairs via semantic matching without the need for expensive constraint solving when searching the database. Using the IntroClass benchmark, we show experimentally that CSAR can achieve high quality repairs comparable to existing approaches run on the same benchmark.

2 CANONICAL SEARCH AND REPAIR

A key benefit to SR is that it can repair code at a high granularity (i.e., 4-8 SLOC) [5]. However, SR struggles at runtime as using the constraint solver to check each potential patch is computationally expensive. CSAR preprocesses potential patch code into constraints, just like SR, but then transforms the constraints into a canonical form. Instead of invoking the solver at runtime, CSAR treats the canonical constraints like strings and performs less expensive string matching to identify similar, but not identical, code as potential patches. In essence, this turns a semantic search problem into a syntactic search problem, with the syntax as canonicalized constraints.

Prior to running unit tests, a simple decision tree classifier predicts when patches are likely to be incorrect. The classifier and speed of string metrics are important as the cost of running tests for patch validation can be prohibitive [3]. CSAR works as follows:

(1) Given buggy location identified using fault localization (e.g. Tarantula [4]), expand to a semantic region
(2) Symbolically execute the region to get Path Conditions (PCs)
(3) Convert PCs to normal form based on Green [16]
(4) Using string metrics, search a database of canonicalized PCs representing potential patches
(5) Use a classifier over string metric data to decide whether to run each patch against the unit tests or reject the patch without running unit tests
(6) Rename variables in patch code to match buggy context, run unit tests

This form automatically ensures matching of renamed variables (i.e. $x < y$ and $a < b$ both become $v0 < v1$), as well as rearranged inequalities (e.g. ints x < y and y > x both become x - y + 1 ≤ 0).[1]

To evaluate the repair technique, we used the IntroClass [2] benchmark. For Step (2), we used Java Symbolic Pathfinder [13],

[1]Which then becomes $v0 - v1 + 1 \leq 0$.

which necessitated a conversion of IntroClass into Java. In the process, many buggy methods were automatically repaired (e.g. garbage collection errors in C are automatically dealt with by Java's garbage collector), reducing the number of buggy programs.

For Step (4), the database was populated by 1,405 distinct potential patch templates. Each template was a randomly-generated (compilable) Java methods. Random generation was done at the semantic unit level, with programs consisting of a random number of `if` statements (possibly nested or including `else`), each of which contained a random number of conjuncts. Each of these conjuncts used random comparators, random variable names, and random combinations of variables, operators, and numbers. The consequents were also randomly generated.

For Step (5), Levenshtein (Edit) distance and Longest Common Subsequence (LCSub) were used.

3 RESULTS

In this work, we differentiate between *plausible* patches, which pass all the tests used during repair, and *correct* patches, which additionally pass the tests in a held-out test suite. The numbers reported in Table 1 represent plausible patches generated using black-box tests. Correctness was determined using white-box tests, all of which are provided with the IntroClass benchmark.

Table 1 gives the number of *plausible* patches generated by each technique and the number of buggy programs attempted in parentheses, as reported in prior work [5]. For example, with SR and *grade*, plausible patches were found for five programs out of 226 attempted. Methods without parentheses attempted to repair all bugs (see *total* column); GenProg (GP) attempted to repair all 52 *checksum* bugs and found plausible patches for eight. Note that SR and CSAR rely on the ability to symbolically execute patch code, which may not be possible for all buggy methods.

On the IntroClass benchmark, 35.8% of AE, 31.3% of GenProg, 27.9% of TrpAutoRepair, 2.7% of SearchRepair, and 0.0% of CSAR are plausible, but not correct. That is, they pass the tests used to generate the patch, but do not pass all the held-out tests [5, 15]. Table 2 attempts to visualize the percentage of *correct* patches as a fraction of all bugs attempted for each technique. However, the split of plausible/correct between the programs was not reported. For non-CSAR patches, the number of plausible patches is multiplied by the overall percentage of plausible patches which are correct, rounded up.[2] For example, with GenProg and *checksum*, $8/52 = .1538$, but since 31.3% are incorrect, $(1 - .313) * .1538 = .106$, which rounds up to .11. For CSAR, all the plausible patches are also correct. Because the plausible/correct split was not reported by category, Table 2 should be taken only as a rough indication of relative performance.

4 DISCUSSION AND LIMITATIONS

In Table 1, the number of patches given for all methods except CSAR are *plausible* patches, while those given for CSAR are the number of *correct* patches. We point out that CSAR is accomplishing similar numbers of patches from fewer bugs. On *grade*, CSAR's performance is more than an order of magnitude better than the nearest alternative. Considering how effective this approach is at the *grade* problems, it is somewhat surprising that in terms of percentage of possible patches obtained, CSAR performs only as well as the best methods at *median*, and improves *smallest* by only 10-15%. This is likely due to the relative density of semantically similar incorrect patches in the database. It is likely that CSAR's performance on *grade* is related to the fact that *grade* requires more extensive logical branching than the other programs.

CSAR is limited by the capabilities of symbolic execution and the contents of its database. If a patch template does not exist in the search space, it will not be found. Likewise, if collecting PCs for buggy code requires constraints for a structure which current symbolic execution cannot handle (e.g. complex class structures), CSAR will be ineffective. Multiple line fixes are tested, but this work only tests single-location fixes. Finally, the size and composition of the database will influence the speed of execution; comparisons for this work (i.e. steps (4) and (5)) never exceeded five seconds on a personal computer with no database optimizations of any kind.

[2] Recent work has suggested that the true number of correct patches from non-reuse approaches may be even lower than previously reported [19]

Table 1: Number of plausible patches generated for the IntroClass benchmark. The number in parentheses represents the number of programs on which patching was attempted, else the value in the Total column is assumed.

Program	SR	AE	GP	TAR	JFix	CSAR	Total
checksum	0	0	8	0	-	-	52
digits	0	17	30	19	-	-	204
grade	5(226)	2	2	2	-	73(75)	228
median	68(168)	58	108	93	-	49(148)	204
smallest	73(155)	71	120	119	16(47)	92(131)	163
syllables	4(109)	11	19	14	-	-	129

SR - SearchRepair, GP - GenProg, TAR - TrpAutoRepair

Table 2: Estimated correct fixes for IntroClass, as a % of total attempts. Correctness is determined based on passing all the tests in a held-out test suite

Program	SR	AE	GP	TAR	JFix	CSAR
checksum	0	0	.11	0	-	-
digits	0	.06	.11	.07	-	-
grade	.02	.01	.01	.01	-	.97
median	.40	.19	.37	.33	-	.33
smallest	.48	.44	.51	.53	.34	.70
syllables	.04	.06	.11	.08	-	-

5 CONCLUSION

This paper presents CSAR, a code-reuse technique which uses semantic search to improve on previous code-reuse approaches and current G&V and synthesis techniques on a subset of IntroClass bugs. We showed that CSAR works on small programs. A next step is to expand this to more complex programs.

ACKNOWLEDGMENTS

This work was supported in part by Google Summer of Code and NSF #1645136, #1549161, and #1329278.

REFERENCES

[1] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. 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*. ACM, 30–39.

[2] Thomas Durieux and Martin Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs.* Ph.D. Dissertation. Universite Lille 1.

[3] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 235–245.

[4] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 273–282.

[5] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. 2015. Repairing Programs with Semantic Code Search (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 295–306. https://doi.org/10.1109/ASE.2015.60

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

[7] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 376–379.

[8] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[9] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178.

[10] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 448–458.

[11] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 691–701.

[12] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 772–781.

[13] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* (2013), 1–35. https://doi.org/10.1007/s10515-013-0122-2

[14] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient automated program repair through fault-recorded testing prioritization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 180–189.

[15] 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 (ESEC/FSE 2015)*. ACM, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

[16] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. 58. https://doi.org/10.1145/2393596.2393665

[17] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 356–366.

[18] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 660–670.

[19] Luciano Zemín, Simón Gutiérrez Brida, Ariel Godio, César Cornejo, Renzo Degiovanni, Germán Regis, Nazareno Aguirre, and Marcelo Frias. 2017. An analysis of the suitability of test-based patch acceptance criteria. In *Proceedings of the 10th International Workshop on Search-Based Software Testing*. IEEE Press, 14–20.