Toward Practical Automatic Program Repair*

Ali Ghanbari
University of Texas at Dallas, TX 75080, USA
ali.ghanbari@utdallas.edu

***Abstract*—Automated program repair (APR) reduces the burden of debugging by directly suggesting likely fixes for the bugs. We believe scalability, applicability, and accurate patch validation are among the main challenges for building practical APR techniques that the researchers in this area are dealing with. In this paper, we describe the steps that we are taking toward addressing these challenges.**

***Index Terms*—Program Repair, JVM Bytecode, Mutation Testing**

I. Introduction

Software debugging is a notoriously difficult activity that consumes over 50% of the development time/effort [1], [2]. So far, a large body of research has been dedicated to automatically localize [3] or fix software bugs [4]. Automated Program Repair (APR) [5] aims to directly fix software bugs with minimal human intervention which has been under intense research despite being a young research area [4], [6].

Based on the actions taken for fixing a bug, state-of-the-art APR techniques can be divided into two main classes: (1) techniques that monitor the dynamic execution of a program to find deviations from certain specifications, and *heal* the program by modifying its runtime state in case of any abnormal behavior [7], [8]; (2) *generate-and-validate* techniques that modify program code representations based on various rules/techniques, and use either tests or formal specifications as the oracle to validate each generated candidate patch for finding *plausible* patches (i.e., the patches that can pass all the tests/checks). Plausible patches are further checked to identify *correct* (or *genuine*) patches (i.e., the patches semantically equivalent to developer patches) [9]–[23].

Scalability, applicability, and accurate patch validation are often cited as the main challenges for building practical APR techniques that the researchers are dealing with [24]. Scalability refers to the ability of an APR technique in handling large, realistic programs. Applicability is the ability of the technique in handling different programming idioms, languages, or even different programming paradigms. Finally, patch validation refers to the process of classifying the patches generated by the APR tool into genuine and plausible patches.

In this paper, we describe our achievements in addressing each of the aforementioned challenges (§II) and discuss related work (§III), before presenting our plans for future work (§IV).

II. PraPR

We introduce a practical, general-purpose APR technique, named PraPR (**Pra**ctical **P**rogram **R**epair) [25], that is operating at the level of JVM bytecode [26]. PraPR is based on three classes of mutators that can be seen as a spectrum of mutators ranging from traditional mutators (e.g., changing `a>=b` into `a>b`) to simplistic program-fixing mutators that have been widely explored in the program repair literature [9], [14], [27]. In particular, we have adopted 18 mutators from traditional mutation testing [28], 12 replacement mutators (e.g., replacing field accesses or method invocations), and 14 mutators that are responsible for inserting checks in the vicinity of field dereferences and method calls. Table I illustrates two examples from each class of mutators wherein the white block contains examples from traditional mutators, light-gray block contains examples from augmented mutators that replace a field name or a method name with another, and the dark-gray part shows examples from augmented mutators that insert nullity checks before dereferences or virtual method calls and use default values [26] instead of triggering NullPointerException.

TABLE I: Mutators examples

ID	Mutator Illustration
AP	y=o.m(x)↪y=x
RV	return x↪return x+1
FR	int x=o.f1↪int x=o.f2
MR	int y=o.m1(x)↪int y=o.m2(x)
FG	int x=o.f↪int x=(o==null?0:o.f)
MG	int y=o.m(x)↪int y=(o==null?0:o.m(x))

PraPR, besides bringing a simple, yet effective, idea into the limelight, offers a 1-click APR tool publicly available on Maven Central Repo [29]. Being compatible with a variety of testing frameworks (e.g., JUnit, TestNG, and Spek), PraPR is readily applicable to arbitrary Java projects under Maven/Gradle build systems (not just Defects4J) and even projects in other JVM languages in a hassle-free manner, thereby allowing researchers replicate our experiments.

In what follows, we elaborate on the challenges mentioned in §I and discuss the contributions of design decisions made in the implementation of PraPR toward attaining APR goals.

A. Scalability

An important goal in constructing a practical, industrial-strength APR technique is to make it scalable to large, real-world programs. Such a technique should be able to produce genuine patches, otherwise it might mislead the developers rather than helping them [24], [30], [31].

PraPR does not need any kind of complicated computation (e.g., symbolic execution and constraint solving) that limits scalability. Thanks to this fact and the bytecode-level manipulation, PraPR with only single thread can already be over *an order of magnitude faster* than state-of-the-art SimFix [9], CapGen [14], JAID [19] (that reduces compilation overhead by bundling patches in meta-programs), and SketchFix [27] (that curtails compilation overhead via sketching [32]).

This Ph.D. thesis is being supervised by Dr. Lingming Zhang at the University of Texas at Dallas.

The speed of patch generation and validation makes it possible for PraPR to apply a larger set of mutators to exhaustively mutate every suspicious location in the buggy program which in turn enables the tool explore a larger search space in a reasonable amount of time. Our experiments show that PraPR successfully fixes 43/395 bugs from Defects4J V1.2.0 [33], significantly outperforming state-of-the-art APR techniques (e.g., the recent CapGen fixes only 22 bugs). We further applied PraPR on 192 additional bugs from Defects4J V1.4.0 [34] from which the tool successfully fixed 12 bugs. Meanwhile, CapGen produces genuine patches for only 2 bugs, while SimFix was unable to generate any plausible patches, in spite of exhausting its search space for most cases, and timed out (a 5-hour time limit) in 52 bugs. Furthermore, in the case of CapGen, we observed a sharp increase in the number of plausible but incorrect patches for Defects4J V1.4.0 bugs, while PraPR shows a decent level of consistency both in the number of fixed bugs and also false positives [25].

This indicates that simplistic bytecode-level mutation could be a viable approach for constructing a scalable APR tool.

B. Applicability

Program source code contains a wealth of information that researchers might exploit to develop more effective APR techniques. However, the process of mutation and/or extraction of fixing ingredients can be significantly different from one programming language to another. This makes APR techniques to be *hardwired* to work with a specific programming language. With the advent of more expressive, and less verbose, JVM-based programming languages such as Java 8 [35] (which adds many syntactic sugars to the older versions of the language), Kotlin [36], Scala [37], and Groovy [38] the need for applicability is especially pronounced for nowadays many real-world projects are written in a combination of these languages [39], so the APR techniques should be applicable in a uniform fashion.

PraPR works at the level of JVM bytecode that makes the tool JVM language agnostic and readily applicable to more than 6 popular programming languages [40]. We have applied PraPR on 118 Kotlin bugs from Defexts database [39], and the tool successfully fixed 14 bugs. To our knowledge, this is the first study on repairing Kotlin bugs. A similar ratio of fixed bugs for the Kotlin systems reduces the threats to external validity of our work, and shows that simplistic bytecode-level mutation alleviates the applicability challenge in the development of practical APR techniques.

C. Accurate Patch Validation

In a practical situation, we usually lack any kind of formal specifications. Thus, virtually every APR technique depends on test cases so as to verify the generated patches. However, since test cases are usually only partial specifications of the desired behavior of the system, we end up with a large number of plausible but incorrect patches (a.k.a. *overfitted patches* [41]). In the absence of an effective automatic classifier, the developer has to examine each and every one of the plausible patches to verify their correctness.

Lately, several techniques for identification of test case overfitted patches, ranging from manual [30], [42] to fully automatic [31], [41], [43], has been proposed. Unfortunately, none of the automatic techniques were applicable in our case; this is mainly due to two reasons: (1) PraPR makes tiny changes to the program that is difficult to be distinguished by the syntactic and semantic heurisitcs studied in [43]; (2) PraPR targets JVM-based languages, so the idea of fuzzing [31] is not suitable [43]. Furthermore, we realized that anti-patterns [30] are also not applicable in our research since it is highly dependent on the C programming language.

We have mined HD-Repair dataset [44] to find the frequency in which our mutators appear in real-world bug fix commits. We prioritize our mutators based on the frequency of their appearance in the dataset. After ranking the patches according to the Ochiai [45] suspiciousness values of the mutated locations, we break the ties with regard to the priority of the mutators. This results in ranking 30/43 patches in Top-1 position.

Backed by our experimental results, we argue that ranking based on the frequencies of the bytecode-level mutators is generalizable to Java and Kotlin. Applying this technique in our experiments with the Kotlin systems also shows an improvement in the number of patches appearing in the Top-1 position. However, we emphasize that any reliable claim about the effectiveness of a new automatic patch validation technique deserves a more fundamental research, e.g., the findings of Gopinath et al. [46] suggest that mutator frequencies might be different for different programming languages.

III. Related Work

Ma et al. leveraged domain knowledge to fix cryptography misuses in Android apps at the level of bytecode [47]. Schulte et al. discusses the possibility to fix bugs through evolution of assembly code [48]. In their paper [49], Staples et al. introduce SABRE, an industrial-strength, semi-automatic framework for mitigating security problems by *wrapping* vulnerable programs at the level of JVM bytecode. PraPR is the first general-purpose APR technique at the bytecode level.

IV. Future Work

Despite the fast JVM-level patch generation and validation, the repair process can still be expensive for large-scale programs and PraPR can still fix only a small ratio of real-world bugs. We are pushing the envelope by improving PraPR in several directions: (1) reducing the number of candidate patches by leveraging and integrating with state-of-the-art fault localization [50]–[53]; (2) working on effective techniques to reorder the test executions [54], thereby improving the chances of dropping non-plausible patches sooner; and (3) runtime optimization for program executions.

References

[1] Undo Software, “Increasing software development productivity with reversible debugging,” https://tinyurl.com/y3qea8go, 2016, accessed June-12-2019.

[2] C. Boulder, “University of cambridge study: Failure to adopt reverse debugging costs global economy $41 billion annually,” https://tinyurl.com/y24ds5op, 2013, accessed Jun-8-2019.

[3] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE TSE*, pp. 707–740, 2016.

[4] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” *IEEE TSE*, 2017.

[5] M. Harman, “Automated patching techniques: the fix is in: technical perspective,” *CACM*, pp. 108–108, 2010.

[6] M. Monperrus, “Automatic software repair: A bibliography,” *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, Jan. 2018.

[7] F. Long, S. Sidiroglou-Douskos, and M. C. Rinard, “Automatic runtime error repair and containment via recovery shepherding,” in *PLDI*, 2014, pp. 227–238.

[8] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard, “Automatically patching errors in deployed software,” in *SOSP*, 2009, pp. 87–102.

[9] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, “Shaping program repair space with existing patches and similar code,” in *ISSTA*, 2018.

[10] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE TSE*, vol. 38, no. 1, pp. 54–72, 2012.

[11] V. Debroy and W. E. Wong, “Using mutation to automatically suggest fixes for faulty programs,” in *ICST*, April 2010, pp. 65–74.

[12] T. Ji, L. Chen, X. Mao, and X. Yi, “Automated program repair by using similar code containing fix ingredients,” in *COMPSAC*, 2016, pp. 197–202.

[13] S. H. Tan and A. Roychoudhury, “relifix: Automated repair of software regressions,” in *ICSE*, 2015, pp. 471–482.

[14] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, “Context-aware patch generation for better automated program repair,” in *ICSE*, 2018, pp. 1–11.

[15] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *FSE*, 2015, pp. 166–178.

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

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: program repair via semantic analysis,” in *ICSE*, 2013, pp. 772–781.

[18] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, “Nopol: Automatic repair of conditional statement bugs in java programs,” *IEEE TSE*, vol. 43, no. 1, pp. 34–55, 2017.

[19] L. Chen, Y. Pei, and C. A. Furia, “Contract-based program repair without the contracts,” in *ASE*, 2017, pp. 637–647.

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

[21] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, “Automated fixing of programs with contracts,” *IEEE TSE*, pp. 427–449, 2014.

[22] V. Dallmeier, A. Zeller, and B. Meyer, “Generating fixes from object behavior anomalies,” in *ASE*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 550–554.

[23] D. Gopinath, M. Z. Malik, and S. Khurshid, “Specification-based program repair using sat,” in *TACAS*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 173–188.

[24] X. B. D Le, “Overfitting in automated program repair: Challenges and solutions,” Ph.D. dissertation, Singapore Management University, 2018.

[25] A. Ghanbari, S. Benton, and L. Zhang, “Practical program repair via bytecode mutation,” in *ISSTA*, 2019, pp. 19–30.

[26] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 2014.

[27] J. Hua, M. Zhang, K. Wang, and S. Khurshid, “Towards practical program repair with on-demand candidate generation,” in *ICSE*, 2018, pp. 12–23.

[28] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE TSE*, pp. 649–678, 2011.

[29] A. Ghanbari, S. Benton, and L. Zhang, “Prapr website,” https://github.com/prapr/prapr, 2019, accessed June-12-2019.

[30] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, “Anti-patterns in search-based program repair,” in *FSE*, 2016, pp. 727–738.

[31] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, “Better test cases for better automated program repair,” in *FSE*, 2017, pp. 831–841.

[32] A. Solar-Lezama, “Program sketching,” *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 475–495, 2013.

[33] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *ISSTA*. New York, NY, USA: ACM, 2014, pp. 437–440.

[34] Greg4cr, “Defects4j – version 1.4.0,” https://github.com/Greg4cr/defects4j/tree/additional-faults-1.4, 2018, accessed June-11-2019.

[35] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*, 2014.

[36] JetBrains, “Kotlin language documentation,” http://kotlinlang.org/, 2018, accessed June-12-2019.

[37] M. Odersky, “The scala language specification,” http://www.scala-lang.org, 2014, accessed June-12-2019.

[38] A. S. Foundation, “Groovy programming language,” http://groovy-lang.org/, 2019, accessed June-12-2019.

[39] S. Benton, A. Ghanbari, and L. Zhang, “Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages,” in *ICSE*, 2019, pp. 47–50.

[40] ”Wikipedia”, “List of JVM Languages,” https://tinyurl.com/cgy8pqv, 2019, accessed May-19-2019.

[41] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, “Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system,” *Empirical Software Engineering*, pp. 1–35, 2018.

[42] Q. Xin and S. P. Reiss, “Identifying test-suite-overfitted patches through test case generation,” in *ISSTA*, 2017, pp. 226–236.

[43] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, “Identifying patch correctness in test-based program repair,” in *ICSE*, 2018, pp. 789–799.

[44] X. B. D. Le, D. Lo, and C. Le Goues, “History driven program repair,” in *SANER*, vol. 1. IEEE, 2016, pp. 213–224.

[45] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “On the accuracy of spectrum-based fault localization,” in *TAICPART-MUTATION*. IEEE, 2007, pp. 89–98.

[46] R. Gopinath, C. Jensen, and A. Groce, “Mutations: How close are they to real faults?” in *ISSRE*. IEEE, 2014, pp. 189–200.

[47] S. Ma, D. Lo, T. Li, and R. H. Deng, “Cdrep: Automatic repair of cryptographic misuses in android applications,” in *Asia CCS*, 2016, pp. 711–722.

[48] E. Schulte, S. Forrest, and W. Weimer, “Automated program repair through the evolution of assembly code,” in *ASE*, 2010, pp. 313–316.

[49] J. Staples, C. Endicott, L. Krause, P. Pal, P. Samouelian, R. Schantz, and A. Wellman, “A semi-autonomic bytecode repair framework,” *IEEE Software*, vol. 36, no. 2, pp. 97–102, 2019.

[50] X. Li, W. Li, Y. Zhang, and L. Zhang, “Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization,” in *ISSTA*, 2019, pp. 169–180.

[51] J. Sohn and S. Yoo, “Fluccs: using code and change metrics to improve fault localization,” in *ISSTA*. ACM, 2017, pp. 273–283.

[52] M. Zhang, X. Li, L. Zhang, and S. Khurshid, “Boosting spectrum-based fault localization using pagerank,” in *ISSTA*, 2017, pp. 261–272.

[53] X. Li and L. Zhang, “Transforming programs and tests in tandem for fault localization,” *OOPSLA*, pp. 92:1–92:30, 2017.

[54] L. Zhang, D. Marinov, and S. Khurshid, “Faster mutation testing inspired by test prioritization and reduction,” in *ISSTA*, 2013, pp. 235–245.