Refining Fitness Functions for Search-Based Automated Program Repair

Giovani Guizzo, Aymeric Blot, James Callan, Justyna Petke, Federica Sarro

Refining Fitness Functions for Search-Based Automated Program Repair

A Case Study with ARJA and ARJA-e

Giovani Guizzo, Aymeric Blot, James Callan, Justyna Petke, and Federica Sarro

University College London (UCL), London, United Kingdom
{g.guizzo, a.blot, james.callan.19, j.petke, f.sarro}@ucl.ac.uk

Abstract. Automated Program Repair (APR) strives to automatically fix faulty software without human-intervention. Search-based APR iteratively generates possible patches for a buggy software, guided by the execution of the patched program on a test suite (i.e., a set of test cases). Search-based approaches have generally only used Boolean test case results (i.e., pass or fail), but recently more fined-grained fitness evaluations have been investigated with promising yet unsettled results. Using the most recent extension of the very popular Defects4J bug dataset, we conduct an empirical study using ARJA and ARJA-e, two state-of-the-art search-based APR systems using a Boolean and a non-Boolean fitness function, respectively. We aim to both extend previous results using new bugs from Defects4J v2.0 and to settle whether refining the fitness function helps fixing bugs present in large software.
In our experiments using 151 non-deprecated and not previously evaluated bugs from Defects4J v2.0, ARJA was able to find patches for 6.6% ($\frac{10}{151}$) of bugs, whereas ARJA-e found patches for 8% ($\frac{12}{151}$) of bugs. We thus observe only a small advantage in using the refined fitness function. This contrasts with the previous work using Defects4J v1.0.1 where ARJA was able to find adequate patches for 26.3% ($\frac{59}{224}$) of the bugs and ARJA-e for 47.3% ($\frac{106}{224}$). These results may indicate a potential overfitting of the tools towards the previous version of the Defects4J dataset.

Keywords: Search-Based Automated Program Repair · Empirical Study · Software Engineering

1 Introduction

Automated Program Repair (APR) [6], as the name suggests, tries to automatically derive software patches, in order to fix bugs with none (or little) human intervention. Various approaches, based on symbolic execution, machine learning and search-based algorithms, have been proposed for APR [2,6]. In this work we focus on search-based APR.

Given a buggy program and related test suite, search-based APR approaches iteratively generate candidate patches by mutating the program. A fitness func-

tion compiles and tests each candidate patch to assess whether it produces expected outputs for all inputs in the test suite. If the test suite passes, then it is an indication that the generated patch was successful and it is deemed "adequate". This type of *Boolean* fitness function has been the approach most used so far, and only recently a more fine-grained fitness evaluation (i.e., non-Boolean), which uses not only the binary result of test cases but also their output in case of a failure, has been proposed [1,12].

Yuan and Banzhaf [12] have been the first to propose a more refined fitness function and included it in the ARJA-e tool. Their empirical evaluation, conducted on 224 real-world Java bugs from Defects4J [3] v1.0.1, showed that ARJA-e correctly fixed 39 bugs, achieving substantial performance improvements over the more traditional search-based APR tools using fitness functions based on Boolean test results. On the other hand, a more recent study conducted by Bian et al. [1] did not reveal any major significant difference when comparing the effectiveness and efficiency of such traditional vs. refined fitness functions.

In this challenge paper, we aim at unveiling whether there is indeed any difference in using traditional vs. refined fitness functions. We use a corpus of real-world software projects with potentially harder-to-fix bugs that has not been used in previous studies. This corpus consists of 151 non-deprecated bugs from 13 software programs from Defects4J v.2, which were not contained in earlier versions of this dataset. We evaluate and compare two well-known state-of-the-art search-based APR software: ARJA [11] and its extension, ARJA-e [12]. Both use Multi-Objective Evolutionary Algorithms (MOEAs) and differ in the implemented fitness functions. While ARJA uses the more traditional fitness function, based on number of failing test cases; ARJA-e uses the "smooth" non-Boolean fitness to gradually guide the APR process towards a passing test suite.

Our results show that there is little benefit in using refined fitness functions, with ARJA producing 10 patches vs. 12 produced by ARJA-e. Moreover, the fix rates are much lower than those reported in the literature, pointing to overfitting to the previous Defects4J dataset.

2 Background

A handful of approaches has been proposed to refine fitness functions in search-based program repair, to not rely on just Boolean test case results. We present all of them below.

ARJA-e Unlike most other APR tools in the literature, ARJA, ARJA-e's predecessor, uses multi-objective optimisation search, considering both *patch size* (f_1) and *weighted failure rate* (f_2). Whilst the patch size simply is the number of edits in a given patch x, the weighted failure rate is defined by Equation 1, with T_{pos} and T_{neg} the sets of negative and positive sets and $w \in [0, 1]$ the weight parameter introducing bias against the latter.

$$f_2(x) = \frac{|\{t \in T_{pos} : x \text{ fails } t\}|}{|T_{pos}|} + w * \frac{|\{t \in T_{neg} : x \text{ fails } t\}|}{|T_{neg}|} \quad (1)$$

ARJA-e further extends ARJA with a more fine-grained fitness function, given by Equation 2. Instead of simply counting the number of failed test cases, the *error ratio* (h) of each executed test is calculated. This ratio is the mean *assertion distance* ($d(e)$) of each assertion executed by the test ($e \in E(x,t)$) for a particular patch. A specific normalised distance metric is used for each type of assertion, e.g., the Levenshtein distance for string objects, or absolute distance for numbers.

$$f_2(x) = \frac{\sum_{t \in T_{pos}} h(x,t)}{|T_{pos}|} + w * \frac{\sum_{t \in T_{neg}} h(x,t)}{|T_{pos}|} \; ; \quad h(x,t) = \frac{\sum_{e \in E(x,t)} d(e)}{|E(x,t)|} \quad (2)$$

2Phase 2Phase's [1] fitness function takes a hybrid approach between ARJA and ARJA-e. When two patches produce differing numbers of passing and failing tests, the one with the least failing/most passing will be considered better. Otherwise, the sum of their assertion distances on failing tests will determine which patch is preferred.

GenProgNS GenProgNS [10] is an implementation of GenProg [5] which replaces fitness evaluation based on the number of passing tests with the novelty of the solution. A Boolean vector b represents the outcome of each test case for a particular variant. For test case i, b_i is assigned 1 if the test case passes and 0 if it fails. The Hamming distance of these vectors is then used to calculate novelty and the most novel solutions are preferred. This approach prioritises exploration of the large search space over exploitation of the improvements found so far.

Checkpoints Checkpoints [9] implements a fitness function in which the values of variables that are present in test cases are tracked throughout execution. These variables' values are compared with those from the original test executions of the buggy program, and fitness is assigned based on their comparison. For failing test cases on a particular variant, the variant is considered more fit if it maintains the values of variables in tests which originally passed and changes variable values in the tests which originally failed.

3 Experimental Design

In this work we want to answer the following question:

How effective is search-based APR if refined fitness functions are used when compared with traditional fitness evaluation, which considers only Boolean test case results?

With this in mind we focus on comparing effectiveness and efficiency of ARJA and ARJA-e. ARJA is a mature tool and ARJA-e provides an extension with the fitness function change, thus making fitness comparison fair for our purposes.

Dataset Defects4J [3] is a prevalent and extremely widespread dataset of real Java bugs. Recently Defects4J has been updated with many new bugs for which very little is known in comparison. We use these new bugs to drive our investigation. To our knowledge, the new bugs introduced in Defects4J v2.0 have only been tackled with non-search-based approaches [7,13]. Table 1 shows

Table 1. Detail of Dejects4J between v1.0.1, v2.0, and depreciated bugs.

	# Bugs					# Bugs			
Project	v1.0.1	v2.0	Dep.	Total	Project	v1.0.1	v2.0	Dep.	Total
Chart	26	0	0	26	JacksonDatabind	0	112	0	112
Cli	0	39	1	39	JacksonXml	0	6	0	6
Closure	131	43	2	174	Jsoup	0	93	0	93
Codec	0	18	0	18	JxPath	0	22	0	22
Collections	0	4	24	4	Lang	64	0	1	64
Compress	0	47	0	47	Math	106	0	0	106
Csv	0	16	0	16	Mockito	0	38	0	38
Gson	0	18	0	18	Time	26	0	1	26
JacksonCore	0	26	0	26					
Total	391	444	29	835		←	←	←	←

the set of bugs in both versions Defects4J v1.0.1 and v2.0. Although there are 444 non-deprecated new bugs in the dataset, we ended up using a subset of 151 buggy project versions due to a few technical challenges. The list of all bugs and results of our experiments can be found at https://figshare.com/s/35ea3fd819e737ed806b.

Technical Challenges ARJA has a few undocumented requirements. ARJA is divided into two modules: the core module contains all the classes for the execution of the algorithms and generation of patches; the external module is used by the core module to instrument the code, run test cases, and capture code coverage. The default location of the external module is hard-coded into ARJA's source code, thus executing the tool within a working directory different from ARJA's root directory results in failures. This can be fixed by providing the path to the external module as an argument, but this is undocumented. This problem is aggravated because ARJA neither fails/crashes when it does not find the external project, nor does it output errors. Hence, it executes normally with missing functionalities, but the results seem to be successful with no generated patches. The second and most crucial challenge regards the resource files/scripts provided by Defects4J to checkout, compile, and to export metadata of projects. The supporting scripts sometimes fail due to various reasons. For example, we could not compile the Compress projects using Defects4J because using Java 8 (required by Defects4J) in combination with the Maven and Ant compilation scripts provided by Defects4J results in failures due to unsupported compilation of Java 1.4 sources (version of the project). Other examples of failures include: inclusion of nonexistent files and missing items in the classpaths, missing configuration files during checkout, and failure to replace placeholders (e.g. ${*project.root.dir*}) with actual paths. These problems can be fixed by manually (and laboriously) inspecting each of the 444 subjects' resource files. Finally, even with working projects, correct set-ups, and correct metadata, ARJA sometimes failed to in-

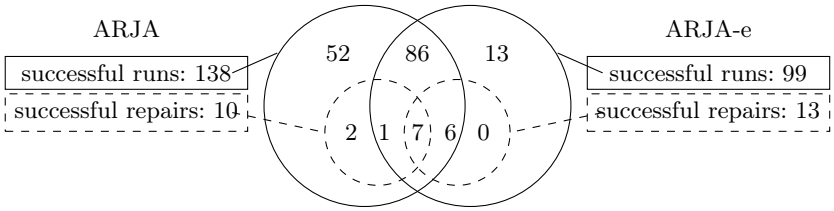

Fig. 1. Set of bugs with successful runs and repairs for ARJA and ARJA-e.

strument the code, localise faults, or it simply hanged. The 151 projects used in this paper are the only ones we could execute without any issues.

Experimental Setup We used the latest commits of ARJA[1] and ARJA-e[2]. Both tools were run to completion using their default recommended configuration without a specific time limit. Experiments were repeated ten times using independent random seeds, using a Sun Grid Engine cluster running CentOS Linux 7 nodes with Java v1.8.0_131, Perl v5.16, SVN v1.8.19, and Git v2.9.5.

4 Results and Discussion

Figure 1 shows the summary of our results. The solid circles represent bugs for which the tools were able to successfully run, whereas the dashed lines represent the subset of bugs for which the tools found test-suite adequate patches.

> Our results show that ARJA is able to find adequate patches for **6.62%** ($\frac{10}{151}$) of the subjects, or 10 out of 138 of the bugs if we consider only successful runs. ARJA-e is able to find adequate patches for **7.95%** ($\frac{12}{151}$) of the subjects, or 12 out of 99 of the bugs if we consider only successful runs.

According to the original work of ARJA and ARJA-e using Defects4J v1.0.1, the tools found adequate patches for 26.33% ($\frac{59}{224}$) and 47.32% ($\frac{106}{224}$) of the bugs, respectively. The dissimilarity between our findings and the results of previous work [11,12] is striking: the differences between both tools is rather small (6.62% vs 7.95%) when compared to the difference found in their original papers (26.33% vs 47.32%). Furthermore, the low fixing rate suggests that the effectiveness of these tools can drop with the addition of unseen bugs, unveiling a possible overfitting (a known issue with APR tools [8,4]) to Defects4J v1.0.1, specially for ARJA-e with a steeper drop. On the other hand, our discoveries are aligned with the work of Bian et al. [1], who also found a negligible difference.

Finally, ARJA generated 6 383 patches, while ARJA-e a total of 8 703 patches. The medians were 97.5 and 63 per bug, respectively, meaning, in the worst case scenario, an engineer would have to analyse over 60 patches per bug to check for semantic correctness. This hinders the feasibility of both tools in practice due to the great amount of manual effort needed.

[1] https://github.com/yyxhdy/arja/tree/e7950328c05e3f7eb38e1af11efc31055af09d05
[2] https://github.com/yyxhdy/arja/tree/f24b777a7c53a390ff97ecfd66fbbdedd8f8b6b3

5 Conclusions

Search-based APR has been successfully used to generate test-suite adequate patches. The search over mutated program variants is guided by a fitness function that usually only considers Boolean test case results. In order to improve it's effectiveness, more refined fitness functions were proposed, that take *types of failures* into account. However, previous work is not unanimous on whether this more sophisticated fitness function actually is more effective.

In this work we ran two state-of-the-art tools, ARJA and ARJA-e, that differ in their fitness implementation, to compare whether the more refined fitness version in ARJA-e indeed helps the search. We ran our experiments on the newly added 151 bugs in the famous Defects4J set. Our results show that neither fitness is significantly better than another. Moreover, we also reveal that the two ARJA variants struggle to find test-suite adequate patches on the new dataset, having found significantly more on older benchmarks.

Acknowledgements: Funded by ERC 741278 and EPSRC EP/P023991/1.

References

1. Bian, Z., Blot, A., Petke, J.: Refining fitness functions for search-based program repair. In: ICSE Workshops (2021)
2. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: A survey. IEEE Trans. Softw. Eng. **45**(1), 34–67 (2019)
3. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: ISSTA. pp. 437–440 (2014)
4. Kechagia, M., Mechtaev, S., Sarro, F., Harman, M.: Evaluating automatic program repair capabilities to repair api misuses. IEEE Trans. Softw. Eng. (2021)
5. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
6. Monperrus, M.: The living review on automated program repair. Tech. Rep. hal-01956501, HAL (2018)
7. Motwani, M., Brun, Y.: Automatically repairing programs using both tests and bug reports. CoRR **abs/2011.08340** (2020)
8. Smith, E.K., Barr, E.T., Goues, C.L., Brun, Y.: Is the cure worse than the disease? Overfitting in automated program repair. In: SIGSOFT FSE. pp. 532–543 (2015)
9. de Souza, E.F., Le Goues, C., Camilo-Junior, C.G.: A novel fitness function for automated program repair based on source code checkpoints. In: GECCO. pp. 1443–1450 (2018)
10. Trujillo, L., Villanueva, O.M., Hernandez, D.E.: A novel approach for search-based program repair. IEEE Softw. **38**(4), 36–42 (2021)
11. Yuan, Y., Banzhaf, W.: ARJA: Automated repair of Java programs via multi-objective genetic programming. IEEE Trans. Softw. Eng. **46**(10), 1040–1067 (2020)
12. Yuan, Y., Banzhaf, W.: Toward better evolutionary program repair: An integrated approach. ACM Trans. Softw. Eng. Methodol. **29**(1), 5:1–5:53 (2020)
13. Zhu, Q., Sun, Z., Xiao, Y., Zhang, W., Yuan, K., Xiong, Y., Zhang, L.: A syntax-guided edit decoder for neural program repair. In: ESEC/FSE (2021)