Attention Please: Consider Mockito when Evaluating Newly Proposed Automated Program Repair Techniques

Shangwen Wang[1,3], Ming Wen[2], Xiaoguang Mao[1,3], Deheng Yang[1,3]
[1]National University of Defense Technology, Changsha, China
[2]The Hong Kong University of Science and Technology, Hong Kong, China
[3]Hunan Key Laboratory of Software Engineering for Complex Systems, Changsha, China
{wangshangwen13, xgmao, yangdeheng13}@nudt.edu.cn; mwenaa@cse.ust.hk

ABSTRACT

Automated program repair (APR) has attracted widespread attention in recent years with substantial techniques being proposed. Meanwhile, a number of benchmarks have been established for evaluating the performances of APR techniques, among which Defects4J is one of the most widely used benchmark. However, bugs in Mockito, a project augmented in a later-version of Defects4J, do not receive much attention by recent researches. In this paper, we aim at investigating the necessity of considering Mockito bugs when evaluating APR techniques. Our findings show that: 1) Mockito bugs are not more complex for repairing compared with bugs from non-Mockito projects; 2) the bugs repaired by the state-of-the-art tools share the same repair patterns compared with those patterns required to repair Mockito bugs; however, 3) the state-of-the-art tools perform poorly on Mockito bugs (Nopol can only correctly fix one bug while SimFix and CapGen cannot fix any bug in Mockito even if all the buggy locations have been exposed). We conclude from these results that existing APR techniques may be overfitting to their evaluated subjects and we should consider Mockito, or even more bugs from other projects, when evaluating newly proposed APR techniques.

CCS CONCEPTS

• **Software and its engineering → Software reliability; Software testing and debugging**

KEYWORDS

Automated Program Repair; Defects4J; Mockito

ACM Reference format:

EASE' 19, April 15–17, 2019, Copenhagen, Denmark © 2019 Association for Computing Machinery. ACM ISBN 978-1-4503-7145-2/19/04…$15.00 https://doi.org/10.1145/3319008.3319349

1 INTRODUCTION

Automated program repair (APR) techniques aim at reducing the excessively high cost in fixing bugs and have shown to be promising in increasing the effectiveness of automated debugging [1]. Pioneered by GenProg [2], substantial techniques have been proposed recently [3 – 14].

To facilitate controlled experiments and fair evaluations for different APR techniques, researchers have built several publicly available benchmarks during recent years [15 – 17]. Defects4J [18], one of the most widely used benchmark, is a database containing Java bugs from real-world open source projects. Each bug in this benchmark is extracted from the project's associated version control histories through three steps, including identifying real bugs fixed by developers, reproducing those real bugs, and isolating them. As a result, each bug corresponds to two versions, buggy and fixed, and is accompanied by a comprehensive programmer-written test suite that can reveal the bug (at least one test triggers a failure on the buggy version). The test execution framework provided by Defects4J can facilitate the experiments of fault localization and program repair, making this database widely used by recent studies.

In the first release of Defects4J in June 2015, it contained 357 bugs from five open source projects, namely JFreechart, Closure compiler, Apache commons-lang, Apache commons-math, and Joda-Time, respectively. Subsequently, in October 2016, the original team added 38 new bugs from another project named Mockito into this database when releasing its version 1.1.0. Many APR techniques were proposed after that [4 – 14], however, none of them have evaluated the performance on these 38 bugs. In Table 1, we list nine APR techniques designed for Java with the projects they select for evaluating their performances. From the results, the newly augmented project, Mockito, does not receive much attention. This phenomenon motivates our study. Our intuition is that since these 38 bugs are extracted by the original methodology and integrated into the database by the professional developers, they must be able to make supplementary for this database from some aspects. It seems unreasonable to exclude them from the evaluation criteria of newly proposed APR techniques.

In this paper, we conduct an empirical study to investigate the importance of taking bugs from Mockito into consideration when evaluating APR techniques. Since our study will compare Mockito with other projects in Defects4J database frequently, we use terms Mockito and non-Mockito to distinguish them. Our study can be summarized in three phases: In the first phase, we do the statistics about the characteristics of human-written patches

from Mockito project and compare them with that of non-Mockito projects. Our aim is to investigate whether the patches from Mockito are more complex than patches from the non-Mockito projects and the result is negative. In the second phase, we examine whether patches from Mockito contain any unique repair patterns that cannot be observed among the patches from non-Mockito projects and the result is still negative. In the third phase, we choose two recently proposed techniques to evaluate their performances on Mockito and adopt the experimental results from a study which evaluated another tool on Mockito. Results indicate that the state-of-the-art APR tools perform poorly when evaluating on Mockito. To summarize: since bugs from Mockito are not more complex than bugs from non-Mockito projects and they do not require extra repair patterns, the poor performances of the state-of-the-art APR tools on this project drive us to conclude that we do need to take Mockito into consideration when evaluating newly proposed APR techniques. There may exist overfitting between existing APR techniques and their evaluated subjects.

In summary, this paper makes the following contributions:

- A comprehensive comparison between Mockito and non-Mockito projects in Defects4J towards the patch characteristics and repair patterns;
- An experiment on the evaluation of the performances of SimFix and CapGen, two of the latest search-based APR tools, on bugs from Mockito.

Table 1. Details of Some Recently Released APR Tools

Tools	Source	Selected projects
ACS	ICSE'17	Chart, Math, Lang, Time
ssFix	ASE'17	Chart, Closure, Math, Lang, Time
JAID	ASE'17	Chart, Closure, Math, Lang, Time
SOFix	SANER'18	Chart, Math, Lang, Time
Probabilistic-Model	SANER'18	Chart, Closure, Math, Lang, Time
CapGen	ICSE'18	Chart, Math, Lang, Time
SketchFix	ICSE'18	Chart, Closure, Math, Lang, Time
Elixir	ICSE'18	Chart, Math, Lang, Time
SimFix	ISSTA'18	Chart, Closure, Math, Lang, Time

In the table, column "Source" denotes the conference the corresponding APR tool publishes in. The contents of this column are divided into two parts: the previous section represents the short name of this well-known international conference while the latter section represents the year. Note that these tools are arranged according to the order of their publication time.

2 RESEARCH QUESTIONS

RQ1: Are patches of Mockito bugs more complex than those of non-Mockito bugs in Defects4J? In order to find out if the bugs in Mockito are more difficult to fix, this problem analyzes these patches from a quantitative perspective. To answer this question, we select four features of the patches which can help to quantify the complexity and difficulty to fix a bug (i.e., patch size, number of chunks, number of modified files, and number of modified methods). We observe the distribution situations of these attributes of bugs from each project and perform a significant difference test.

RQ2: Are the state-of-the-art techniques capable to fix non-Mockito bugs whose repairs require the same repair patterns as Mockito bugs? This question, analyzing the repair patterns of Mockito bugs, is an extension of the previous one. To answer this question, we classify bugs in Mockito into several kinds according to their repair patterns and check if there is any bug in non-Mockito projects possessing the same repair pattern for each kind. Further, for each kind, we explore whether there are bugs in non-Mockito projects that have been successfully repaired before.

RQ3: What are the performances of the state-of-the-art APR techniques on Mockito? This question is the focus of our research and is directly related to our conclusion. Generally, APR tools can be classified into two categories, i.e., search-based and semantics-based. For evaluating the performances of the state-of-the-art tools on Mockito, we select two of the latest search-based tools, SimFix and CapGen, for performing the experiment and we also add the experimental results from another study [19], which presents the evaluation results on Mockito of a semantic-based tool, Nopol [20], into our analysis.

3 RESULTS AND ANALYSIS

3.1 RQ1: The Complexity of Patches

To characterize patches, a recent study [21] has performed detailed analysis of the patch characteristics in Defects4J. They further select six indicators for analyzing patch features (patch size, number of chunks, number of modified files, number of modified methods, spreading of chunks, and number of modified classes). In our study, we select the former four features for evaluating the patch complexity. We do not take the latter two into considerations since 1) spreading of chunks is not directly related to patch complexity, e.g., if two single lines of addition appear at the beginning and end of a class, it can spread more than three chunks of modification in the middle of this class, nevertheless, it is not obvious which situation is more difficult to be repaired; 2) number of modified classes is highly related to the number of modified files as shown in [21], making this indicator redundant. Along their ideas, we give our own explanations as follows.

It is widely known that there are three types of code changes: addition, deletion, and modification. Addition and deletion appear as lines of codes are added or deleted consecutively or separately in source code. Modification appears as sequences of removed lines are straight followed by added lines or vice-versa. The patch size is the sum of the number of lines of these three types of code change in the patch. In previous study [2], authors used patch size as an indicator when evaluating the repair results by stating that the less the patch size is, the more manageable the patch is. Thus, this indicator is a key point to the patch complexity.

Composed by the combination of addition, deletion, and modification of lines, a chunk is a sequence of continuous changes in a file. The number of chunks of a patch can provide insights on how a patch is spread through the source code and further give information about how complex the patch is: the more chunks means the more buggy points in the program, and thus the more complexly for fixing this bug. Patches with more chunks are more complex in logical structure than patches with fewer chunks. Several empirical studies have proved this perspective.

Similarly, the number of modified files and the number of modified methods are also two important indicators. The larger they are, the more program elements are involved in the patch, and the more complex the patch is.

The previous study [21] has figured out the statistics of these indicators of bugs from Defects4J using the same standard as we mentioned. We sum the data and make the distribution of the patches of each project about these four features as the box plots in Figure 1. Note that in each subgraph, the bar displayed at the rightmost is the distribution of this feature for the patches of non-Mockito projects (the other five projects in Defects4J except Mockito).

1) *Patch size:* In Mockito, the average value of patch sizes is around 7 (7.08, accurately) while the value for non-Mockito projects is 6.7. The median value is 4 and 25% of the patches change less than two lines, both these values the same as that of the non-Mockito projects.
2) *Number of chunks:* In Mockito, the average and median numbers are 3.5 and 3, both larger than that of the non-Mockito projects which are 2.6 and 2, respectively. The maximum of this feature is 20 and it occurs three times (one in Mockito, one in Lang, and one in Time).
3) *Number of modified files:* In Defects4J, 92.41% of the patches modify only one file [21]. Thus, the box in the figure becomes a line with the value of 1. Nevertheless, the average value of Mockito is 1.2, still slightly larger than that of the non-Mockito projects which is 1.08.
4) *Number of modified methods:* Although the first quartiles of Mockito and non-Mockito projects are both 1, the aver age of Mockito is 2.3, larger than that of non-Mockito projects (1.4). This time, the maximum of this feature is 20 and it only occurs once in Mockito-6.

From the results, we can see that the values of patch characteristics of Mockito are slightly larger than that of non-Mockito projects with respect to the four features. We further perform the significant difference test to check if the differences are significant and the result is positive. Thus, the Mockito patch features are subject to the same distribution as the patch features of non-Mockito projects.

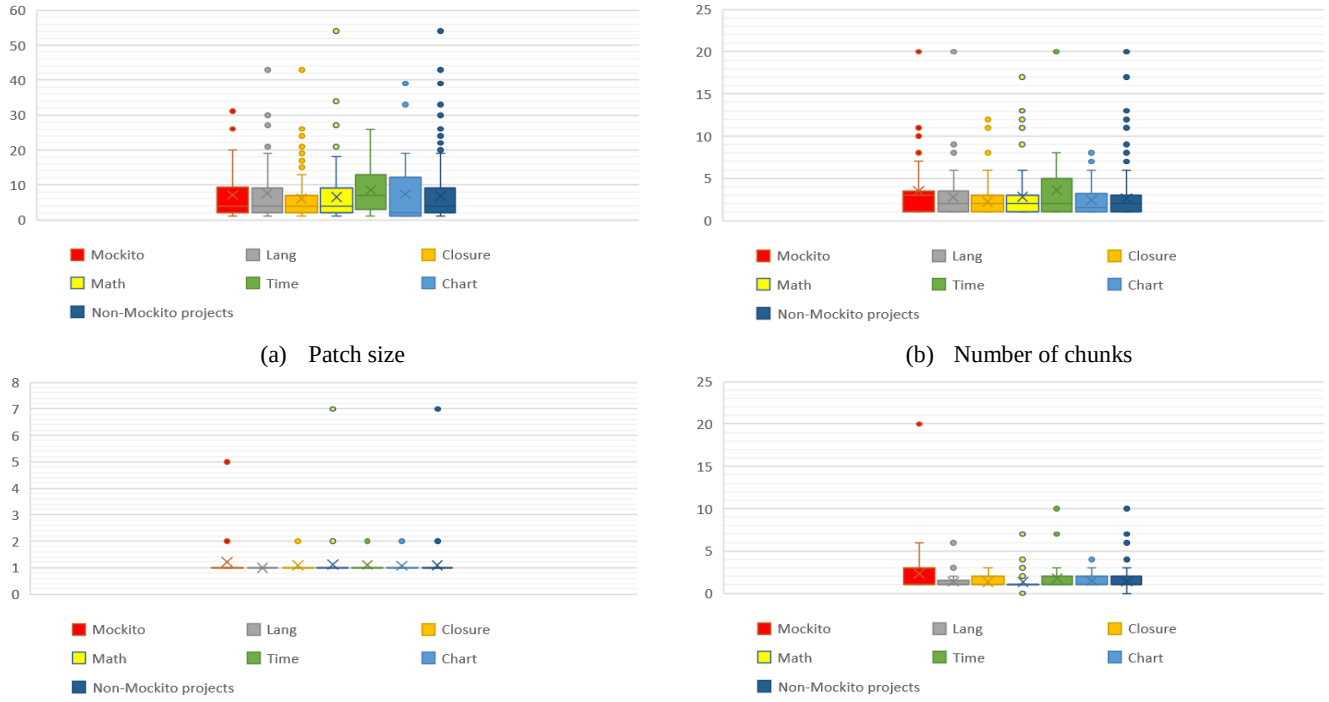
RQ1: Are patches of Mockito bugs more complex than that of non-Mockito bugs in Defects4J?
Findings: With respect to four measurements (patch size, number of chunks, number of modified files, and number of modified methods), the average values of Mockito are slightly larger than that of non-Mockito projects. However, these differences are not significant at the 10% significance level according to a test. Thus, **Mockito bugs are not more significantly complex for repairing than bugs from non-Mockito projects**.

3.2 RQ2: The Repair Patterns in Mockito

Repair pattern is a recurrent abstract structure in patches [22]. In the study [21], the repair patterns are established based on a Thematic Analysis (TA) process including identifying initial repair actions, combining repair actions re-appearing over many patches, and naming the themes.

Pioneered by Pattern-based Automatic program Repair (PAR) [3], there is a trend in APR of utilizing existent repair patterns for guiding the repair process [8, 9, 13]. Researchers have summarized more and more repair patterns with finer-grained granularity, from AST statement level to expression level, and made great progress in using these patterns for fixing bugs. In this question, we aim to check whether the repair patterns summarized in previous studies are enough for bugs in Mockito and whether there exists any bug that has been fixed utilizing the same pattern with Mockito bugs.

The previous study [21] categorizes patches in Defects4J benchmark into nine repair patterns. We do the statistic the repair patterns of bugs in Mockito from their results and illustrate them in Table 2. Note that to obtain information from a more detailed perspective, the authors of [21] create several sub-categories under each category (e.g., to distinguish the wrapping structure, the variants for Wraps-with include *if*, *if-else*, and *method call* as is shown in Table 2). In the column "Repair patterns", we demonstrate the main categories identified by the study [21] and we choose to demonstrate the sub-categories in the column "Detailed patterns" with the aim of investigating from a more detailed perspective. The column "Selected samples" shows some cases of bugs from non-Mockito projects which need the same repair pattern to fix. Due to the space limitation, we randomly select eight cases for each classification as the samples since the most common category (*Single line*) possesses 98 bugs (see the column "Total number of bugs"), making it impossible to list them all in the table. The selection is completely random without any bias and in these samples, the number of bugs of each project is calculated based on the proportion of its total bugs in this category. The column "Fixed bugs" at the right-most lists the successfully repaired bugs in the column "Selected samples". The APR techniques we take into consideration when collecting these information come from ACS, Elixir, ssFix, JAID, CapGen, Nopol, and SimFix. In this paper, to refer to Defects4J bugs, we use a simple notation with project name followed by bug id, e.g., Math-5. Note that in this table, a bug can be classified into several categories (e.g., Mockito-19 is classified into *Conditional block addition* and *Addition with return statement*), that is due to the multiple edits in the patch: every code change chunk in the patch is analyzed and classified and if a patch contains several chunks, it may be classified into several categories. From the results, bugs in Mockito contain up to 15 kinds of repair patterns with 5 of them (Mockito-10, 25, 27, 30, 31) being not classified. *Conditional block addition* is the most popular pat-tern among this project, including totally ten bugs. Several patterns like *Logic expression expansion* and *Constant change* contain only one instance and thus are not very popular in this project. For each classification, at least two cases from samples have been successfully repaired in the past. At the most, six of the eight instances have been fixed before (*Single line* and *Not classified*). Even the bugs which are not classified can be fixed by current tools. According to the afor-



(a) Patch size

(b) Number of chunks

(c) Number of modified files

(d) Number of modified methods

Figure 1: Features distribution map of the Defects4J patches

Table 2. Details of the repair patterns of Mockito bugs

Repair patterns	Detailed patterns	Total number of bugs	Mockito bugs#	Selected samples	Fixed bugs
Conditional block	Conditional block addition	79	1,2,3,13,19,21, 23,33,36,37	Lang-39,45 Time-1,2,13 Math-36,51,54	Lang-39,45
	Addition with return statement	77	4,9,11,18,19, 21,22,23	Chart-14,15 Closure-33,60 Lang-5,40 Math-92,93	Chart-14,15 Closure-33 Math-93
Expression fix	Logic expression expansion	48	34	Chart-9 Closure-39,50 Lang-24,27,30 Math-32,37	Chart-9 Lang-24,27 Math-32
	Logic expression reduction	12	13	Chart-5 Lang-15 Time-2 Closure-7,23,31,35,89	Chart-5 Closure-31
	Logic expression modification	49	17	Chart-16 Closure-22,62,73 Lang-50 Math-47,94 Time-19	Closure-62,73 Lang-50 Math-94 Time-19
Wraps-with	Wraps-with if statement	24	14,15,16	Chart-15 Closure-96 Lang-3,46 Math-28,95 Time-3,27	Chart-15 Math-95
	Wraps-with if-else statement	46	11,12,17, 24,29,38	Closure-23,56,111 Lang-28,33 Math-47,97 Time-12	Lang-33 Time-12
	Wraps-with method call	14	4,14,28	Chart-12,13 Closure-16 Time-8 Math-23,26,35,105	Chart-12,13 Math-35,105
Wrong reference	Wrong reference variable	42	6,20	Chart-8,12 Closure-54 Time-4 Lang-4,60 Math-21,64,98	Chart-8,12 Lang-60 Time-4 Math-98
	Wrong reference method	31	6,32,35	Chart-13 Closure-4,45,109 Lang-26 Math-58,75 Time-26	Chart-13 Lang-26 Math-58,75
Missing null-check	Missing null check	25	4,23,29,38	Chart-15,25 Closure-20,30 Lang-32,33 Math-32 Time-2	Chart-15,25 Lang-33 Math-32
	Missing non-null check	32	33	Chart-25 Closure-17,22,76,98 Lang-12 Math-20 Time-21	Chart-25 Math-20
Constant change	Constant change	19	26	Closure-14,40,70 Lang-19 Math-22,104 Time-8,10	Closure-14,40,70 Math-22,104
Copy/Paste	Copy/Paste	48	4,6,35	Chart-7 Closure-4,6 Time-12 Lang-30,62 Math-37,76	Chart-7 Time-12
Single line	Single line	98	5,7,8,24,26, 28,29,34,38	Chart-1,24 Closure-62,67 Lang-51 Math-32,94 Time-16	Chart-1,24 Closure-62 Lang-51 Math-32,94
Not classified	Not classified	22	10,25,27,30,31	Chart-3 Closure-25 Lang-35 Math-8,61,90 Time-7,22	Chart-3 Lang-35 Time-7 Math-8,61,90

ementioned analysis, Mockito bugs do not contain unique repair pattern and the state-of-the-art APR techniques have achieved success on fixing bugs with the same repair patterns.

RQ2: Are the state-of-the-art techniques capable to fix non-Mockito bugs whose repairs require the same repair patterns as Mockito bugs?
Findings: Most of the repair patterns of Mockito bugs (33/38) can be classified into existing categories. In each category, a number of bugs in non-Mockito projects have been successfully fixed by existing techniques. Thus, the **state-of-the-art techniques are capable of fixing bugs from non-Mockito projects whose repairs require the same repair patterns as Mockito bugs.**

3.3 RQ3: The Performances of the State-of-the-art APR Techniques on Mockito

APR techniques can be generally divided into two categories (i.e., search-based and semantic-based). Search-based repair methods (also known as generate-and-validate methods) search within a huge population of candidate patches generated by applying mutation operators to a predefined fault space determined by Fault Location (FL) techniques and are widely recognized to be able to fix a wide range of bugs [9, 23]. Semantics-based repair methodology, on the contrary, utilizes semantic information generated by symbolic execution and constraint solving to synthesize patches. In this section, we aim to check the performances of the state-of-the-art APR tools on Mockito. To this end, we select two latest and open access search-based tools (SimFix and CapGen) for evaluating them on Mockito and we also take a reproduced study [19] about the semantic-based tool, Nopol, into consideration when analyzing. All the execution logs of our experiments can be found in the link[1].

3.3.1 SimFix

SimFix is one of the latest search-based APR tools and has been evaluated on the non-Mockito projects in Defects4J [13]. The key novelty of SimFix is that it that takes the intersection of existing patches and source code into consideration to reduce search space. We first got the fault space information of suspicious buggy statements for each bug through the Ochiai algorithm implemented in GZoltar version 1.6.0 [24] and then reran SimFix to evaluate its performances on the Mockito bugs. All the experiments were conducted on a 64-bit Linux virtual machine with Ubuntu 15.10 operating system and 2GB RAM. The execution results of SimFix can be three types: *success* when a patch passes all test cases, *failed* when all the suspicious statements are executed but still no patch is found, and *timeout* when the execution exceeds the pre-defined time. The experiment results are illustrated in Table 3.

As is shown, none of Mockito bugs can be fixed by SimFix. From the execution logs, ten of these bugs failed due to timeout and the rest 28 bugs were due to incapable of finding a valid patch at all suspicious locations.

The failure of SimFix is mainly caused by two reasons. First is the lack of rich context information. Take Mockito-37 as an example, the human-written patch of this bug adds an method invocation (*reporter.cannotCallRealMethodOnInterface()*) under a conditional branch. However, this method invocation does not occur in any other place in this project, making SimFix incapable of generating a correct patch. Second is the course-grained donor snippet identification. Given a potential faulty location, SimFix expands it into a faulty code snippet and then locates a set of similar code snippets as donors for repair. The size of the code snippet can be as large as 10 lines. Consequently, if the correct fixing ingredient is only in a single line, SimFix may overlook it and fail to repair. For example, in Mockito-26, human-written patch turns a parameter in a method invocation from *0* to *0D*. The correct fixing ingredient is several lines before this statement but the objects which invocate the same function at these two locations are not the same, making *variable name similarity* (i.e., a metric which is used by SimFix to identify similar donor) rather low. Thus, SimFix considers these two code snippets as dissimilar, leading to the miss of possible correct patch of this bug.

3.3.2 CapGen

CapGen is another the state-of-the-art search-based APR tool that utilizes context information to prioritize patches. Empirically, its precision can reach 84% on the four projects of Defects4J [9].

We reran CapGen on Mockito bugs under the environment which is consistent with its original execution environment in [9]. The experiment results are illustrated in Table 4. CapGen generates patches for all of the 38 bugs, but none of them are plausible (i.e., pass both the failing test cases and the regression test cases).

CapGen fails mainly because: 1) it performs a single mutation to generate patches, making it impossible to fix bugs which need operations at several different places (i.e., multi-location bugs) and this kind of bug occupies a large proportion in Mockito project (65.8%, 25/38); and 2) it usually cannot find the correct fixing ingredients since its searching scope is restricted in a single file. For example, in Mockito-27, human-written patch replaces a parameter of an object instantiation with another one (*oldMockHandler.getMockSettings()*) under the class *MockUtil*. However, the correct fixing ingredient is in another file named *MockHandler*, making it impossible for being extracted by CapGen. This point is easy to understand because the previous analysis has indicated that there may be no fixing ingredient even if the search space is the whole project.

3.3.3 Nopol

Nopol is a semantic-based program repair tool utilizing angelic values and a Satisfiability Modulo Theory (SMT) solver for synthesizing conditional expressions [20]. A previous study has evaluated its performance on all the projects in Defects4J version 1.1.0 [19], thus, we adopt their results for our analysis here. Table 5 illustrates the empirical results of evaluating Nopol on Defects4J bugs.

It is shown in the results that Mockito is the lowest in terms of both the number of generated patches (2) and the repair rate (5%). Nopol totally generates patches for 103 Defects4J bugs and reaches an average repair rate of 26%. Note that these patches only pass all the test cases and are not manually checked. As a result, the repair rate here is consistent to the recall in other

[1] https://github.com/Kaka727/Mockito-Study

literature [4, 9] (i.e., the percentage of bugs for which Nopol generates a patch among all the bugs). We then manually analyzed these two patches and considered a patch correct if it is the same or semantically equivalent to human-written one provided in Defects4J, which is widely adopted by previous studies [4, 9, 12, 25]. According to our manual analysis, the patch for Mockito-29 is semantically equivalent to human-written one since they both wrap a statement with a conditional statement. However, the patch for Mockito-38 is not correct since it changes code at a wrong place and the generated code is partially redundant and to some extent unreadable with too many mathematic symbols. Thus, Nopol actually only fixes one bug in Mockito, achieving a poor performance of a repair rate at around 2.6%.

RQ3: What are the performances of the state-of-the-art APR tools on Mockito?
Findings: Both search-based APR approaches and semantic-based APR approaches perform poorly, with extremely low repair rate, on Mockito bugs. Compared with its performance on non-Mockito projects, Nopol have lower repair rate on this project and can only fix one bug in fact. SimFix and CapGen even cannot fix any bug in this project. Thus, **the performances of the state-of-the-art APR tools on Mockito are poor**.

Discussion: According to the findings of our RQ1-3, Mockito bugs are not more complex with respect to the four patch characteristics (i.e., patch size, number of chunks, number of modified files, and number of modified methods); the state-of-the-art APR tools have achieved success on bugs whose repairs require the same repair patterns with Mockito bugs; three representative APR tools (one is semantic-based and two are search-based) achieve poor performances on this project. These are the bases of our argument. It is unreasonable to exclude Mockito from the verification set since the experimental results show that existing APR techniques may be overfitting to non-Mockito projects. Therefore, we recommend that **Mockito should be considered when evaluating newly proposed APR techniques to avoid potential bias from the evaluation process and judge the progress in repairing ability.**

4 LIMITATIONS

The results of this study may not apply to other APR tools since APR is a hot topic in Software Engineering (SE) and there are lots of tools being proposed in main software conferences and journals each year, but we only analyzed three of them. However, our evaluating subjects are representative since SimFix possesses the highest recall (34/357, 9.52%) and CapGen possesses the highest precision (21/25, 84%) on Defects4J among the state-of-the-art APR tools. Nopol generates the most patches for Defects4J bugs among semantic-based tools and is a widely used semantic-based tool in recent empirical studies [25, 26]. Therefore, it is reasonable to speculate that other approaches may demonstrate similar results. Besides, we deprecated other tools for several reasons: HDRepair, SOFix, ProbabilisticModel, Elixir and SketchFix do not release the source code of their tools for reproducible experiments. JFix announces to design a plugin for Eclipse but does not provide the download address (the download link provided in JFix's homepage is invalid). ACS announces in its homepage that it can no longer execute on new bugs due to the interface change in GitHub. ssFix needs a code search phase and the database is stored in a server in Brown university in the USA, making this process much slower for overseas users like us (we get the information after connecting with the authors). Thus, evaluating more APR tools on Mockito project can be future work.

Table 3. Experiment results of SimFix

Execution results	Bug ID
Timeout	1, 2, 3, 6, 9, 12, 23, 26, 33, 35
Failed	4, 5, 7, 8, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 27, 28, 29, 30, 31, 32, 34, 36, 37, 38

Table 4. Experiment results of CapGen

Execution results	#Bug
Generated	38
Plausible	0
Correct	0

Table 5. Experiment results of Nopol

Projects	#Patches	#Bugs	Repair Rate
Chart	9	26	37%
Closure	56	133	42%
Math	24	106	22%
Lang	4	65	6%
Time	8	27	29%
Mockito	2	38	5%
Total	103	395	26%

5 RELATED WORK

During the years, developers have created several benchmarks for reproducible experiments on APR tools. The iBugs project [15] which contains 223 Java bugs with an exposing test case was initially created for fault localization. The software-artifact infrastructure repository (SIR) [16] can be considered as the first to provide a database of real bugs but most of its bugs are hand-seeded or obtained from mutation. Recently, a multilingual program repair benchmark named QuixBugs [17] is designed with 40 programs in both Python and Java, each with a bug on one line. Although many benchmarks are created, our evaluation object, Defects4J, is the most widely-used one for Java language in recent studies [4-12].

There are also some experiments about evaluating previous tools on newly released benchmark. Martinez et al. [24] reimplement GenProg and Kali in Java language and evaluate them with Nopol on Defects4J benchmark. A recent technique report [19] evaluates GenProg on Defects4J version 1.1.0

including Mockito bugs and thus its experiment results are used by us for analysis. Our study is the first to conduct experiments to present results for SimFix and CapGen on Mockito bugs.

6 CONCLUSION

While Defects4J database is widely used in evaluating the performances of APR tools, the Mockito project has not attracted enough attention. In this paper, we studied the importance of taking bugs from Mockito into consideration when evaluating APR techniques. We investigated the characteristics as well as the repair patterns of patches in Mockito. We conducted experiments on two of the latest search-based tools (SimFix and CapGen) on this project and took the experimental results of another semantic-based tool (Nopol) into consideration. Results show that the state-of-the-art tools achieve poor performances on Mockito bugs. We thus reached the conclusion that indeed we should incorporate Mockito into the evaluation criteria.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant 61672529.

REFERENCES

[1] Britton T, Jeng L, Carver G, et al. Reversible debugging software[J]. Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep, 2013.

[2] Weimer W, Nguyen T V, Le Goues C, et al. Automatically finding patches using genetic programming[C]//Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, 2009: 364-374.

[3] Kim D, Nam J , Song J , et al . Automatic patch generation learned from human-written patches[C]// 2013 35th International Conference on Software Engineering (ICSE). IEEE Computer Society, 2013.

[4] Xiong Y, Wang J, Yan R, et al. Precise condition synthesis for program repair[C]// Proceedings of the 39th International Conference on Software Engineering. IEEE Press, 2017: 416-426.

[5] Xin Q, Reiss S P. Leveraging syntax-related code for automated program repair[C]//Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, 2017: 660-670.

[6] Chen L, Pei Y, Furia C A. Contract-based program repair without the contracts[C]//Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on. IEEE, 2017: 637-647.

[7] Liu X, Zhong H. Mining stackoverflow for program repair[C]//2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018: 118-129.

[8] Soto M, Le Goues C. Using a probabilistic model to predict bug fixes[C]//2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018: 221-231.

[9] Wen M, Chen J, Wu R, et al. Context-Aware Patch Generation for Better Automated Program Repair[C]// Proceedings of the 40th International Conference on Software Engineering. ACM, 2018.

[10] Hua J, Zhang M, Wang K, et al. Towards practical program repair with on-demand candidate generation[C]//Proceedings of the 40th International Conference on Software Engineering. ACM, 2018: 12-23.

[11] Saha R K, Yoshida H, Prasad M R, et al. Elixir: an automated repair tool for Java programs[C]//Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. ACM, 2018: 77-80.

[12] Jiang J, Xiong Y, Zhang H, et al. Shaping Program Repair Space with Existing Patches and Similar Code[C]// The International Symposium on Software Testing and Analysis. 2018.

[13] Le X B D, Chu D H, Lo D, et al. JFIX: semantics-based repair of Java programs via symbolic PathFinder[C]//Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2017.

[14] Durieux T, Cornu B, Seinturier L, et al. Dynamic patch generation for null pointer exceptions using metaprogramming[C]//Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. IEEE, 2017: 349-358.

[15] Dallmeier V , Zimmermann T. Extraction of bug localization benchmarks from history[C]// IEEE/ACM International Conference on Automated Soft-ware Engineering. ACM, 2007.

[16] Do H , Elbaum S , Rothermel G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact[J]. Empirical Software Engineering, 2005, 10(4):405-435.

[17] Lin D, Koppel J, Chen A, et al. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge[C]// Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. ACM, 2017: 55-56.

[18] Just R, Jalali D, Ernst M D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs[C]// Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, 2014.

[19] Durieux T, Danglot B, Yu Z, et al. The patches of the nopol automatic repair system on the bugs of defects4j version 1.1.0[D]. Université Lille 1-Sciences et Technologies, 2017.

[20] Xuan J, Martinez M, Demarco F, et al. Nopol: Automatic repair of conditional statement bugs in java programs[J]. IEEE Transactions on Software Engineering, 2017, 43(1): 34-55.

[21] Sobreira V, Durieux T, Madeiral F, et al. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J[C]//2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018.

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

[23] Wen M, Chen J, Wu R, et al. An empirical analysis of the influence of fault space on search-based automated program repair[J]. arXiv preprint arXiv:1707.05172, 2017.

[24] Pearson S, Campos J, Just R, et al. Evaluating and improving fault localization[C]//Proceedings of the 39th International Conference on Software Engineering. IEEE Press, 2017: 609-620.

[25] Martinez M, Durieux T, Sommerard R, et al. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset[J]. Empirical Software Engineering, 2017, 22(4): 1936-1964.

[26] Wang S, Mao X, Niu N, et al. Multi-Location Program Repair Strategies Learned from Successful Experience [J]. arXiv preprint arXiv:1810.12556, 2018.