Automatic Program Repair of Java Single Bugs using Two-level Mutation Operators

Cherry Oo, and Hnin Min Oo

Abstract—**Automatic program repair (APR) is one of the necessary software maintenance tasks because most software systems have errors that need to be fixed. APR techniques are considered as a search problem where the search space includes all potential repair candidates, with the aim of identifying the correct repair code in space. This paper proposes a repair approach that finds the correct repair code for object-oriented program bugs such as Java bugs in the minimized search space using the type of buggy statement and mutation system, MuJava. This approach consists of four main phases. First, program bugs are localized by prioritizing the statements based on their suspiciousness of containing bugs. Second, the mutation system is employed to mutate the program using two-level operators of the mutation system. Third, we extract the fixed candidates that are similar to the buggy statement type within the mutants and after receiving the ordered list of candidate patches, the last phase validates their correctness one by one using the test suite until a correct patch found. The experiment demonstrates that our strategy can effectively fix 19 out of 21 bugs from four real-world projects and achieves 90.48% accuracy.**

Index Terms—**patch generation, fault localization, mutation system, program repair, single bugs.**

I. Introduction

AUTOMATIC program repair (APR) is a relatively new field of research that is currently being explored using different strategies by many research groups. APR techniques were considered more difficult and complex than fault localization techniques. They automatically generate the patch to fix the bug based on the specified buggy program and related test cases. The patch can be used to guide developers or added to the program automatically to continuously enhance the quality of the program.

It is generally possible to divide existing approaches into two categories: search and semantics [1]. Search-based APR [2], [11]-[13] produces large populations of candidates for repair through source operations and finds the best among them. Semantics-based APR [9]-[10] utilizes symbolic execution and test suites to obtain requirements or constraints on the repaired techniques and utilizes program synthesis to produce repairs that fulfill the extracted constraints. Both categories rely mainly on the primary assumption that if the output causes the program to pass all the test cases provided, a program will be repaired correctly [1].

A key issue in patch generation systems is patch quality. As patches only validate for test cases, the program does not guarantee that the correct outputs are produced for other test cases. Recent work has shown that most patches approved by many systems are not generated and tested to generate accurate results beyond validation tests for test cases. This negative impact not only produces plausible patches but also emphasizes the importance of fixing patches that have no potential defects. The success of an APR technique is very important for a rich search space that contains the correct patches for the target bugs. Recent systems used the search space that contains significantly correct patches, so the search space including more successful patches is needed to make continued progress in this area. However, the ability of the technique to identify the correct patch in a larger, relevant space but the incorrect patch may also be complicated by these richer spaces. In the search space structure, Long and Rinard [17] describe a key trade-off for two reasons. Firstly, validation time increases due to more candidate patches and secondly, the test suite passes through more incorrect patches. The quality of the repair is defined as the correctness and maintenance of the repair, where it indicates how well a repaired program can retain the required functionality and how easy it can be understood and maintained. The time and steps needed to find potential repair are APR performance.

The motivations of automated software repair are to reduce the cost of fixing errors and increase the likelihood that the search space will contain correct fixes. To increase the probability of including the correct patch based on the second motivation, we propose a mutation-based system using the type of buggy statement and mutation system for the program repair of Java programs. In object-oriented programs, we assess our approach with real-world bugs from large programs and it can repair the single line bugs. As input, it requires a program and a set of test cases in which at least one test case fails. A search-based algorithm is used to find the correct patch consisting of a location, a buggy code, and a repair code. To evaluate and analyze our approach, we collect 21 bugs from four real-world projects: Closure Compiler, JFreeChart, Apache Commons Lang, and Apache Commons Math. Our results show that 19 of the 21 bugs can be fixed by our approach. The main contributions are:

- We particularly assess the most suspicious statement in real-world Java programs using a spectrum-based metric.
- We investigate a mutation-based program repair technique using the type of buggy statement and mutation system, called MuJava, to rapidly-produce relevant repair codes among the mutants.
- We evaluate our technique on the Defects4J dataset. The results show that correct patches can be produced with 90.48% high precision.

The remainder of this paper is structured as follows: some background information is described in Section 2 followed by our methodology for automated program repair in Section 3. In Section 4 and 5, we discuss our experiments. Then we mentioned some state of research in Section 6 and Section 7 concludes our study.

C. Oo and H. M. Oo are with University of Computer Studies, Mandalay, Myanmar, e-mail: cherryoo@ucsm.edu.mm.

II. Preliminary

Automatic program repair attempts to locate a program variant that meets an oracle by providing the source code and the oracle that can reveal its bugs. Assuming that the oracle is powerful enough, the repair process resolves these errors. Some repair techniques use formal specifications, but in most cases, existing test suites are used. A test suite is a specification based on input-output. Failure tests are used to expose bugs in these suites while passing test cases are used to prevent the destruction of existing and correct behaviors [2]. The search-based approach is a popular class of automated repair techniques and can be effective in dealing with these global optimal problems. It will automatically search for a program fix in the repair space [2].

The use of mutation techniques to repair programs is based on a mutation test that evaluates the performance of a test set by investigating whether a set can detect syntactical code changes in the program [37]. To create a new program called a mutant, a mutation operator is used. Each mutant represents a defective program version [38]. The idea of a mutation-based program repair is that a resulting mutant can't only be a real bug in a certain way, but also a repair for the buggy program. The APR with the mutation is aimed at finding the potential mutant with the desired correct program that has the same behavior. The behavior is generally described in practice by the execution of test cases [39]. Usually, a correct program does not cause a failure in a proper operation system, i.e. all test cases have been passed.

The techniques for search-based program repair are a popular class of APR techniques. Given a buggy program, which fails at least one test in a test suite, the repair process searches for fixing candidates from mutations in a program, which can pass all the tests with a certain set of repair templates. Our repair approach consists of four primary steps such as fault localization (step 1), mutant generation (step 2), fixed candidate extraction (step 3) and patch validation (step 4). The effectiveness of search-based program repair is restricted by the number to be successfully created of correct patches. There are two reasons. First, there is no correct patch in the search space. Second, the search space is enormous and thus it is impossible to create the correct patch.

A. *Fault Localization*

The fault localization step of the repair method is used to pick a small range of locations indicated as suspicious among the program to reduce the scope of the search space. They correspond to program statements for many existing automatic repair strategies [2]. The fault localization technique ranks a buggy statement highly suspicious and the bug can be fixed to generate possible mutants using the mutation operators, starting with the most suspicious statement. This process can be done without the need for human intervention automatically.

The first step in the process of fault localization is to calculate coverage information for the program to determine the set of buggy statements. The coverage value defines the statements that the program executes for each test within the suite. Each program statement is wrapped in a coverage statement, responsible for logging its execution and monitoring it [2]. The next step is to aggregate the data into a fault spectrum from this coverage information. The fault spectrum provides a concise description of which statements each test performed and whether the test result was a failure.

Previous research has found certain coefficients such as Ochiai [24]-[25], and Tarantula [26], as the best metric for the spectrum-based method. We use an existing technique of fault localization in our approach. Existing empirical studies [27]-[28] have shown that in object-oriented programs, Ochiai is more effective than other methods in identifying the root cause of defects. It is defined as follows [24].

$$S_j = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}} \qquad (1)$$

In the above equation, a_{11} is the number of failed runs involving part *j*. a_{10} is the number of passed runs involving part *j* and a_{01} is the number of failed runs which does not include part *j*. Given a buggy program *P* = S_1, S_2, . . . , $S_j$with *j* statements and executed by *i* test cases *T* = T_1, T_2, . . . , T_i. The test results of a_{11} test cases are recorded in the form of a matrix as program spectra information. The component in the i^{th} row and j^{th} column of the matrix refers to the spectral information of statement S_j, by test case T_i, with *1* indicating S_j, and *0* otherwise [29].

B. *Mutation System*

Mutation testing is usually done to evaluate the effectiveness of a particular test set by seeing if changes to the program syntax code can be recognized in the test suite. [14]. A mutation operator is introduced to the original program to generate a new program called a mutant. All mutants are the buggy version of the initial program [15]. The principle of using mutations for program repair is that the resulting mutants can represent "real defects" to some extent, then a buggy program mutant also produces a fix for the buggy program.

Mutation analysis employs well-defined mutation operators on syntactic structures to systematically modify the syntax or objects created from the syntax [30]. It is the process of executing program changes to create new programs, then executing test cases with new programs, and analyzing the results of program execution.

First, to generate a mutant P', mutant operator *m* is used for the program P. Applying *m* to P is likely to result not only in one such P' being generated but also in some similar yet different mutants. If the program contains multiple locations that can be applied, *m* will apply one at each location [18].

Given a test set *T*, it is possible to operate each mutant P' for each test case. If *T* has a test case (called *t*) such that *P(t)* / = $P'(t)$ (i.e., the output of P' is different from the output of P) then it is killed by *t*. That is, an error was found in P' because P' did not produce the expected output. Mutants that were not killed in any of *T* lived, meaning that no errors were found [18]. Since mutant P' can functionally be the same as the original program P, it cannot be killed by any test case. These mutants are referred to as equivalent mutants. This set can be assigned the mutation score, which is the percentage of non-equivalent mutants killed by the test case in the test set. If the mutation score is 100%, the test set is considered to be adequate [18].

C. Single Line Bugs and Statement Types

A bug line in a program is a region that appears weak and likely to cause problems or failure. So, we concentrate on single line bugs and evaluate manually to pick them from four projects in the dissection of Defects4J [22] for our repair strategy. Our approach considers three categories of statement types like expression statement, declaration statement, and control flow statement to extract the mutated statements that are possible as the fixed candidates from the mutants. Expression statements modify variable values, call methods, and object creation. Declaration statements declare variables and control-flow statements define the order of execution of statements. Java statements typically parse from top to bottom. However, the order can be interrupted with control-flow statements to enforce a branching or looping to allow a particular code section based on certain conditions to be implemented by the Java program. If block, return, and looping statements are control-flow statements.

III. Approach

A. Overview

In this subsection, we introduce an overview of our approach.

We propose a mutation-based program repair approach that uses the type of bug and a mutation system to increase the chance of correct fixes included in the search space. A buggy program and a set of test cases are required, but at least one test case fails. It potentially locates buggy locations, then it extracts the potential fixed candidates based on the type of buggy statements and mutation system, called MuJava. We use two-level mutation operators of MuJava to find the fixed candidates and rank the candidates using a variable model. After receiving the order lists, the proposed system validates the correctness of them using its test suite to define a repair. When all of the tests are successful, the candidate is considered a potential repair. An overview of the proposed system is given in Figure 1.

B. Defects4J Dataset

In this subsection, we introduce the Defects4J dataset. Defects4J is a bug dataset with 395 actual bugs that are designed to assist studies in software testing. Bugs in Defects4J have been collected from six Java open-source projects: JFreeChart, Joda Time, Mockito Testing Framework, Closure Compiler, Apache Commons Math and Apache Commons Lang. The buggy version and its related fixed version are supplied by Defects4J for each bug [22].

A bug line in a program is a region that appears weak and likely to cause problems or failure. So, we concentrated on single line bugs and evaluated manually to pick the bugs from four projects in the dissection of Defects4J. Table I represents a detailed description of the experiment benchmark. Table II shows the description of the bugs targeted by our approach. A bug index (Column 2) is called based on the following rule. Letter *F* indicates faults, *CM* refers to Commons Math; *CL* refers to Commons Lang; *FC* refers to JFreeChart and *CC* refers to Closure Compiler. For instance, *FCM* refers to a bug in the Commons Math project. In Math project, *FCM5*, *FCM6*, and *FCM7* come from a same buggy program

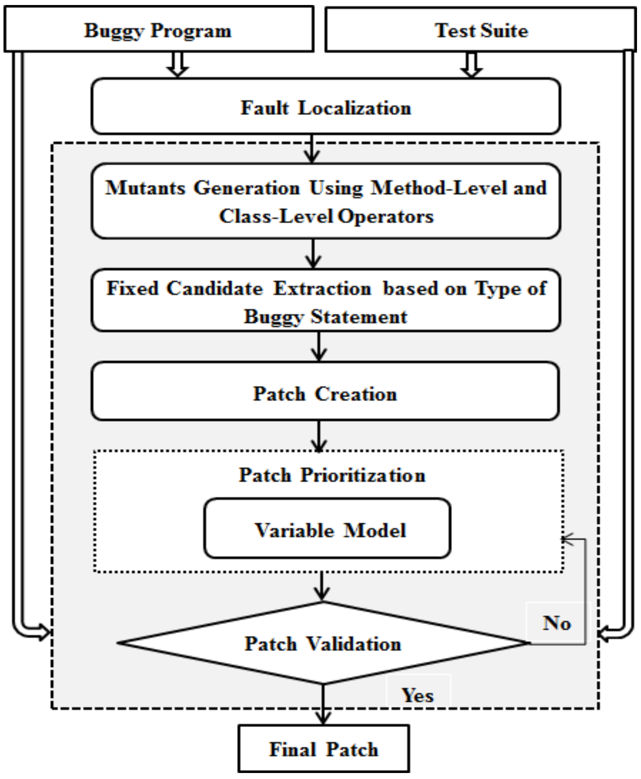

Buggy Program
Test Suite
Fault Localization
Mutants Generation Using Method-Level and Class-Level Operators
Fixed Candidate Extraction based on Type of Buggy Statement
Patch Creation
Patch Prioritization
Variable Model
Patch Validation
No
Yes
Final Patch

Fig. 1. Overview of the Proposed System

TABLE I
EXPERIMENT BENCHMARK DETAILS

Project	KLOC [1]	#Tests [2]	#Bugs [3]
Commons Math	85	3602	106
Commons Lang	22	2245	65
JFreeChart	96	2205	26
Closure Compiler	90	7927	133

[1] ”KLOC” is thousands of lines of code,
[2] ”#Tests” stands for the number of tests and
[3] ”#Bugs” stands for the number of bugs.

but their buggy line numbers are different. In Lang project, we consider as two single line bugs, named *FCL2* and *FCL3* contained in *FastDateFormat* class. Although they have the same bug ids, we separately fix these bugs as each single line bug. Column 4 displays the number of suspicious components of the targeted bugs. We define the functionality of the method for each method that contains the buggy code and record its Cyclomatic Complexity (Column 5). The number of linearly independent paths in the source code of a system is cyclomatic complexity. The complexity shows how well a method is tested and how difficult it is for developers to understand code [40]. As shown in Table II, the average of suspicious components and complexity are 268.6 and 5.9 and the median of them are 209 and 4 for 21 bugs, respectively.

C. Two-level Mutation Operators

In our strategy, we use a scheme of mutations called MuJava [1] [5] to automatically produce the mutants with mutation operators of both class and method level. The mutation system modifies the program in small ways and is called a mutant for each mutated version. Mutants are based on clearly defined mutation operators that imitate

[1]https://cs.gmu.edu/ offutt/mujava/

TABLE II
DESCRIPTION OF TARGETED BUGS

Bug ID	Bug_Index	Bug Class	# Suspicious Component	# Test Case	Complexity
Math2	FCM1	HypergeometricDistribution	71	17	1
Math5	FCM2	Complex	215	138	5
Math22	FCM3	FDistribution	54	19	1
Math41	FCM4	Variance	113	15	7
Math43	FCM5	SummaryStatistics	205	15	4
Math43	FCM6	SummaryStatistics	205	15	4
Math43	FCM7	SummaryStatistics	205	15	4
Math80	FCM8	EigenDecompositionImpl	854	18	4
Math85	FCM9	UnivariateRealSolverUtils	34	9	6
Math96	FCM10	Complex	129	83	4
Lang22	FCL1	Fraction	269	25	12
Lang50	FCL2	FastDateFormat	468	15	4
Lang50	FCL3	FastDateFormat	468	15	4
Lang58	FCL4	NumberUtils	435	54	19
Lang60	FCL5	StrBuilder	661	74	9
Chart1	FFC1	AbstractCategoryItemRenderer	536	10	10
Chart16	FFC2	DefaultIntervalCategoryDataset	190	20	5
Chart24	FFC3	GrayPaintScale	38	5	1
Closure62	FCC1	LightweightMessageFormatter	72	12	9
Closure63	FCC2	LightweightMessageFormatter	72	12	9
Closure100	FCC3	CheckGlobalThis	50	44	2
Average			255	30	5.9
Median			205	15	4

typical errors in the programming (for example by misuse of the operator or name of the variable). Mutation operators alter expression by a substitute, delete and insert. They can produce more mutants by using larger and more diverse sets, and at the same time, they can also improve the ability to repair several types of bugs. Consequently, the choice of the mutation operator to use has a significant effect on the efficiency and effectiveness of the strategy. Indeed, the number of bugs that are fixed by the repair approaches depends on mutant operators choosing. A very effective strategy must be able to solve many types of failures, but greatly efficient strategies require minimal overhead. MuJava is to manage all possible syntactic modifications for object-oriented characteristics to develop mutation operators [4]-[5]. We apply two-level mutation operators (i.e., class-level and method-level) of MuJava to generate the mutants. MuJava provides six types of primitive operators including conditional and arithmetic operators. For some of them, MuJava provides short-cut operators [5], [7]-[8]. In our approach, we use all method-level operators of MuJava and ten class-level operators in the group of Java-specific features. Table III and Table IV show the method-level and class-level mutation operators respectively.

In class-level, EAM operator changes to other compatible accessor method names based on Java-specific features. This type of mistake happens because there will be many accessor methods with the same signature and very similar names in classes with various instance variables. This makes it easy to confuse programmers [4], [6]. For example, given a program that contains *getA()* and *getB()* functions and *point.getA();* is a statement within the program. MuJava generates the possible mutants when the program is mutated by MuJava using EAM operator, so *point.getB();* is one of the generated mutants.

The useful mutation operator can handle all the possible syntactic changes for a programming language. Generally, the mutation operators can be created by one of the ways like delete, insert and change a target syntactic element. The use of larger and more diverse sets of mutation operators may generate more mutants, but at the same time, it may also improve the ability to repair several types of bugs. Therefore, choosing the mutation operator to use has a major impact on the efficiency and effectiveness of the strategy. MuJava is to manage all possible syntactic modifications for object-oriented characteristics to develop mutation operators. This system monitors the selective method in planning for two-level mutation operators by manually investigating the buggy statement and human patches.

TABLE III
METHOD-LEVEL MUTATION OPERATORS

Operator	Description
AODU	Deletiion of unary arithmetic operator
AODS	Deletion of short-cut arithmetic operator
AOIS	Insertion of short-cut arithmetic operator
AOIU	Insertion of basic unary arithmetic operator
AORB	Replacement of basic binary arithmetic operator with alternative
AORS	Replacement of short-cut arithmetic operator
ASRS	Replacement of short-cut assignment operator
COD	Deletion of unary conditional operator
COI	Insertion of unary conditional operator
COR	Replacement of conditional operator
ROR	Replacement of entire predicate by true and false and relational operator with alternative relational operator
SOR	Replacement of shift operator
LOR	Replacement of logical operator
LOI	Insertion of unary logical operator
LOD	Deletion of logical operator
CDL	Deletion of constant
ODL	Each relational, arithmetic, bitwise, logical, and shift operator deletion from assignment and expression
SDL	Each deletion of the executable statement by commenting on it and replacement of entire predicate by true and false. It does not remove statement.
VDL	Deletion of variable deletion

TABLE IV
CLASS-LEVEL MUTATION OPERATORS

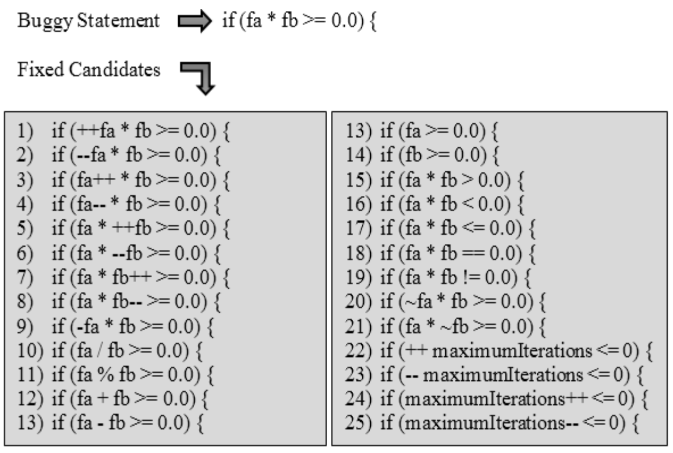
Operator	Description
JTI	Java-specific this keyword insertion
JTD	Java-specific this keyword deletion
JSI	Java-specific static modifier insertion
JSD	Java-specific static modifier deletion
JID	Java-specific member variable initialization deletion
JDC	Java-supported default constructor creation
EOA	Java-specific reference assignment and content assignment replacement
EOC	Java-specific reference comparison and content assignment replacement
EAM	Java-specific accessor method change
EMM	Java-specific modifier method change

TABLE V
NUMBER OF MUTANTS GENERATED BY MUJAVA

Bug_Index	Method-level Mutants	Class-level Mutants	Total
FCM1	856	75	931
FCM2	2395	40	2435
FCM3	683	40	723
FCM4	1315	11	1326
FCM5	544	510	1054
FCM6	544	510	1054
FCM7	544	510	1054
FCM8	16534	50	16584
FCM9	449	0	449
FCM10	1571	22	1593
FCL1	3177	104	3281
FCL2	14024	75	14099
FCL3	14024	75	14099
FCL4	4298	18	4316
FCL5	7903	18	7921
FFC1	1831	393	2224
FFC2	1269	56	1325
FFC3	344	15	359
FCC1	318	4	322
FCC2	318	4	322
FCC3	234	6	240

D. Mutant Generation using MuJava

MuJava is a Java-based mutation framework that creates mutants automatically in the case of traditional mutations and class-level mutations. It adapts the existing Mutant Schemata Generation (MSG) method for mutants that modify the behavior of the program and uses byte code translation for mutants that modify the structure of the program. The MSG method uses compile-time reflection and encodes all mutants for a program into a specially parameterized program, called a metamutant. Compile-time reflection, called OpenJava acts as an evaluation of the source of the program and builds the Java source for mutants. Structural mutants change elements of the structure of the program such as variables and method declarations. Behavioral and structural mutants are generated and performed by different engines and their effects are then combined. The number of mutants generated by MuJava is shown in Table V. MuJava made the largest number of mutants for bug index *FCM8* but for *FCM9*, it can't make any mutant at class-level. This is because it can't initialize the parse tree when many packages have been imported into the buggy program.



```
if (fa * fb >= 0.0) {

1)  if (++fa * fb >= 0.0) {
2)  if (--fa * fb >= 0.0) {
3)  if (fa++ * fb >= 0.0) {
4)  if (fa-- * fb >= 0.0) {
5)  if (fa * ++fb >= 0.0) {
6)  if (fa * --fb >= 0.0) {
7)  if (fa * fb++ >= 0.0) {
8)  if (fa * fb-- >= 0.0) {
9)  if (-fa * fb >= 0.0) {
10) if (fa / fb >= 0.0) {
11) if (fa % fb >= 0.0) {
12) if (fa + fb >= 0.0) {
13) if (fa - fb >= 0.0) {

13) if (fa >= 0.0) {
14) if (fb >= 0.0) {
15) if (fa * fb > 0.0) {
16) if (fa * fb < 0.0) {
17) if (fa * fb <= 0.0) {
18) if (fa * fb == 0.0) {
19) if (fa * fb != 0.0) {
20) if (~fa * fb >= 0.0) {
21) if (fa * ~fb >= 0.0) {
22) if (++ maximumIterations <= 0) {
23) if (-- maximumIterations <= 0) {
24) if (maximumIterations++ <= 0) {
25) if (maximumIterations-- <= 0) {
```

Fig. 2. Buggy Statement and Some of Fixed Candidates for Bug FCM9

E. Fixed Candidate Search

We extend the bug fixing strategy of Debroy and Wong [18] and it comes by combining the concepts of mutation and fault localization. They work Our approach generates all of the mutant versions that are generated by MuJava using two-level mutation operators of MuJava. These mutants may be the search area for repair candidates but we extract all mutated statements that are the same statement type with the most suspicious statement from the mutant programs. All extracted statements are repair candidates of our approach so that our search space is minimized by choosing the mutated statements that are the same type with the buggy statement.

To prevent the costly recompilation, we directly create all repair candidates in the JVM byte code level and choose the candidates based on three categories of statement types (explained in Section II) according to the type of buggy statement obtained by fault localization. Our approach represents the statements as abstract syntax tree nodes together with their line numbers and we consider three types of control flow statements such as if statement, for statement and return statement. For example, if the type of buggy statement is the expression statement, we extract all expression statements from the mutated statements. They are the fixed candidates for our approach. Table VI shows the number of fixed candidates extracted by our approach. As an example, Figure 2 illustrates the buggy statement and some of the fixed candidates extracted by our approach for *FCM9*.

We search the correct patches according to Algorithm 1. Firstly, our algorithm selects the most suspicious statement according to fault localization and all mutated statements from the mutants. Line 6 checks the type of buggy statement pointed out by fault localization and Line 7 extracts the mutated statements that are the same with the type of buggy statement. We assume all extracted statements as the fixed candidates of our approach. For all fixed candidates, we find the correct patch by passing through the two sets of test cases and collect if it passes through both sets of test cases i.e., Lines 8 to 20. Then, our approach generates a set of patches that contain a buggy location, a buggy statement and a fixed candidate in each patch.

TABLE VI
NUMBER OF FIXED CANDIDATES EXTRACTED BASED ON TYPE OF BUG STATEMENT

Bug_Index	Method-level Candidates	Class-level Candidates	Total
FCM1	261	32	293
FCM2	1126	12	1138
FCM3	321	0	321
FCM4	151	0	151
FCM5	116	16	132
FCM6	116	16	132
FCM7	116	16	132
FCM8	3765	1	3766
FCM9	135	0	135
FCM10	47	6	53
FCL1	1458	15	1473
FCL2	3786	0	3786
FCL3	3786	0	3786
FCL4	2607	0	2607
FCL5	744	0	744
FFC1	453	9	462
FFC2	775	39	814
FFC3	57	3	60
FCC1	119	0	119
FCC2	119	0	119
FCC3	3	0	3

Algorithm 1 Correct Patch Search Algorithm

```
1: Input : Failing test cases $T_f$ and passing test cases $T_p$, ranking list of fault localization $F_L$, mutants $M$ generated by MuJava and buggy program $P$.
2: $R \leftarrow \Phi$; // $R$ is a set of patch
3: $C \leftarrow \Phi$; // $C$ is a candidate set
4: Select the most suspicious statement $s$ on the top of $F_L$.
5: Select all mutated statements $s_m$ from $M$.
6: $t_p \leftarrow check(s)$; // check type of buggy statement
7: $C \leftarrow extract(t_p, s_m)$; // collect candidate statements that are the same type with $s$
8: for all $c_i \in C$ do
9:     $P \leftarrow substitute(s, c_i)$; // substitute with each candidate from $C$ into $P$
10:     for all $failing\ test\ case\ t_i \in T_f$ do
11:         if $t_i\ passed\ P$ then
12:             $R \leftarrow R \cup c_i$;
13:         end if
14:     end for
15:     for all $passing\ test\ case\ t_i \in T_p$ do
16:         if $t_i\ passed\ P$ then
17:             $R \leftarrow R \cup c_i$;
18:         end if
19:     end for
20: end for
21: Output : A set of correct patch R.
```

F. Patch Prioritization

We assume that the appropriate repair candidates will share high similarities with the buggy code. This is consistent with current empirical studies [13], [19]-[20]. To generate candidate patches, our approach prioritizes the fixed candidates by modeling the variables used within the source and target codes to produce the correct patch quickly. The similarity of variable names concerns whether the variable names between a buggy statement (target) and the mutated statements (sources) are similar. This idea was developed from an existing study called the context-aware patch generation technique [13]. It is used to rank the fixed ingredients by modeling the context information obtained from three different aspects according to the node types of abstract syntax tree (AST).

Given an AST node, our approach extracts a set of variables (including local variables and fields) that are accessed by this node. We use JDT [2] packages to generate the tree nodes. The number of variables involved in AST nodes is used to represent the weight of the statements. Variable usages can provide the ranking of fix candidates with more similar variables usages compared with the buggy statement at the higher position. We use the Jaccard coefficient to measure the feasibility of two nodes.

$$f(S,T) = |S_i| * \frac{|S_i \cap T|}{|S_i \cup T|} \tag{2}$$

Two elements are the same only if they match both the type and the name. However, for nodes such as $SimpleName$ (i.e. variables), their names must be different if we replace them with another. Therefore, for such cases, we only require the data type to be the same.

IV. Empirical Evaluation

In this section, we presents the evaluation of our approach on Defects4J dataset.

Our approach is analyzed on 21 bugs from four real-world projects in Defects4J. Our experiment is intended to answer the following research questions:

RQ1: Are the output patches as correct as the patches written by the developer?

RQ2: Can our approach fix real bugs in large-scale Java programs?

RQ3: How are the fixing capabilities of mutation operators?

[2]https://www.eclipse.org/jdt/

A. Patch Correctness

Due to a weak test-suite is used as an oracle, passing the test suite may result in the test suite not being able to the adequate patch for specific bugs. This is a condition called patch overfitting [3], [16]. In this section, we manually assess the accuracy of the patches generated by our approach. The test-suite adequate patches are considered correct if they are the same or semantically equivalent to a patch manually written by the developer.

For **RQ1**, patches should be more than just passing test suites because test cases may not be sufficient to specify program behavior. In this article, we assume that if the patch is functionally equivalent to the manually written patch by developers is correct. We have followed Qi et al. [3] to manually investigate the correctness of each generated patch. The manual analysis involves understanding the role of patches in computing, understanding the realms (for example, testing mathematical functions in the Apache Commons Math project), and understanding the meaning of test cases as that assertion.

Table VII shows the patch evaluation of 21 bugs. Columns 2 and 3 represent the buggy code and the fixed code by our approach. Column 4 demonstrates the fixed code that is written manually by developers as discovered in the Defects4J benchmark anatomy [3]. Our manual patch correctness analysis is shown in column 5 (explained in RQ1). In this column, we illustrate that the patches produced by our technique for 8 out of 19 bugs are the same as the patches of developers.

Table VIII shows the performance on four projects of Defects4J. To check correctness, the suggested patches are manually compared with the actual developer patches in the Defects4J. The index of the bugs we say is described in column 2 and the number of correct patches is displayed in column 3. In columns 4 and 5, we show the result for the first correct patch obtained by patch prioritization and the time taken by our approach in seconds respectively.

Table IX shows the code coverage analysis of the outputs produced by the proposed system. To measure the quality and effectiveness, we analyze the coverage values of test cases, instruction, branch, line, and method of the repaired programs that are used the fixed code suggested by the proposed system using EclEmma Java code coverage tool [4]. Test coverage finds the area of a requirement not implemented by a set of test cases. It will include gathering information about which parts of a program are executed when running the test suite to determine which branches of conditional statements have been taken [42]. According to the previous studies [43]-[44], code coverage of 70-80% is a reasonable point for system tests of most projects with most coverage metrics. In Table IX, our approach gets reasonable coverage values except for *FFC1* in JFreeChart project.

B. Real-world Bug Fix

Our methodology focused on single line bugs from Defects4J benchmark [22], [36]. We use the Ochiai coefficient to detect the suspiciousness of the buggy statements. In our experiment, we compare three well-studied fault localization techniques: Ochiai [24], Tarantula [26] and Jaccard [41].

[3]http://program-repair.org/defects4j-dissection/#!/

[4]https://www.eclemma.org/

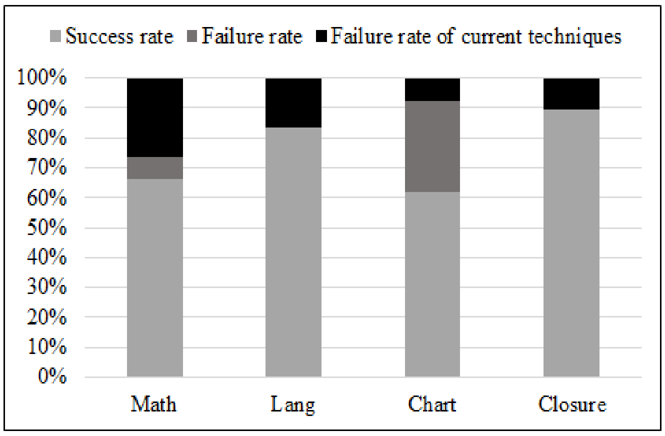


Fig. 3. Analysis on Repair Rate of Our Approach

Table XI presents the comparison among the fault localization techniques using the top-N rank that is widely used to evaluate techniques for fault localization. According to their ranking result, Ochiai obtains 52.38% in top-1 while Tarantula and Jaccard obtain 4.76% and 19.05% in top-1 respectively. For our approach, Ochiai is a reasonable choice.

For RQ2, our methodology reinforced the state of the art techniques by evaluating the overlap of repaired bugs between our approach and state of the art APR techniques including CapGen, jGenProg, Nopol, and ssFix. Our approach can fix 9 of the 21 bugs that have never been fixed by existing techniques. Figure 3 illustrated the analysis of the repair rate of our approach and existing techniques. Our approach can fix four bugs that are not possible with existing techniques in the Math project and can also fix three bugs in Lang, each one bug in Chart and Closure projects. When we calculate the failure rate of current techniques on our bugs, their rate is 36% in Math, 20% in Lang and 8% and 12% in Chart and Closure, respectively. Our success rate is 90% in Math, 67% in Chart and 100% in Lang and Closure projects, respectively. The failure rate of our approach is 10% in Math and 33% in Chart. There is no failure rate in Lang and Closure projects. In our experiments, we found that our approach can generate the correct patches for 19 bugs in Defects4J [22], [36], which are displayed in Table X. We achieve a precision rate of 90.48%. Our experiments demonstrate that our approach outperforms and complements existing techniques, beyond this quantitative performance.

C. Bug Fixing Capabilities

Our approach finds the fixed candidates among all mutants generated by MuJava. Such mutations are produced with well-defined mutation operators, which are program transformations that introduce small artificial faults into the program under test systematically. For RQ3, we applied all method-level operators and ten class-level operators of MuJava in our approach. The bug fixing capabilities of these mutation operators are shown in Figure 4. From this figure, we note that ROR (the relational operator) can produce fixes for 43.14% of all the bugs that we have been able to make fixes for the use of the mutation operators. This is followed by COI (13.73%) which corresponds to the replacement of the conditional operator. AOIS, COR and EAM mutation

TABLE VII
BUGGY CODE, FIXED CODES BY OUR APPROACH AND DEVELOPERS

Bug_Index	Buggy Code	Fixed code by our approach	Fixed code by developers	Same/Difference
FCM1	return (double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize();	return (double) (getSampleSize() * getSupportUpperBound()) / (double) getPopulationSize();	return getSampleSize() * (getNumberOfSuccesses() / (double) getPopulationSize());	Difference
FCM2	return NaN;	–	return INF;	–
FCM3	return true;	return false;	return false;	Same
FCM4	for (int i = 0; i < weights.length; i++)	for (int i = begin; i != begin + length; i++)	for (int i = begin; i < begin + length; i++)	Difference
FCM5	if (!(meanImpl instanceof Mean))	if (meanImpl instanceof Mean)	if (meanImpl != mean)	Difference
FCM6	if (!(varianceImpl instanceof Variance))	if (varianceImpl instanceof Variance)	if (varianceImpl != variance)	Difference
FCM7	if (!(geoMeanImpl instanceof GeometricMean))	if (geoMeanImpl instanceof GeometricMean)	if (geoMeanImpl != geoMean)	Difference
FCM8	int j = 4 * n - 1;	int j = 4 * -n - 1;	int j = 4 * (n - 1);	Difference
FCM9	if (fa * fb >= 0.0)	if (fa * fb > 0.0)	if (fa * fb > 0.0)	Same
FCM10	ret = (Double.doubleToRawLongBits(real) == Double.doubleToRawLongBits(rhs.getReal())) && (Double.doubleToRawLongBits(imaginary) == Double.doubleToRawLongBits(rhs.getImaginary()));	ret = Double.doubleToRawLongBits(real) >= Double.doubleToRawLongBits(rhs.getReal()) && Double.doubleToRawLongBits(imaginary) == Double.doubleToRawLongBits(rhs.getImaginary());	ret = (real == rhs.real) && (imaginary == rhs.imaginary);	Difference
FCL1	if (Math.abs(u) <= 1 | | Math.abs(v) <= 1)	if (Math.abs(u) == 1 | | Math.abs(v) <= 1)	if (Math.abs(u) == 1 | | Math.abs(v) == 1)	Difference
FCL2	if (locale != null)	if (locale == null)	if (locale == null)	Same
FCL3	if (locale != null)	if (locale == null)	if (locale == null)	Same
FCL4	if (dec == null && exp == null && isDigits(numeric.substring(1)) && (numeric.charAt(0) == '-' | | Character.isDigit(numeric.charAt(0))))	if (dec == null && exp == null && isDigits(numeric.substring(1)) | | (numeric.charAt(0) == '-' | | Character.isDigit(numeric.charAt(0))))	if (dec == null && exp == null && (numeric.charAt(0) == '-' && isDigits(numeric.substring(1)) | | isDigits(numeric)))	Difference
FCL5	for (int i = 0; i < thisBuf.length; i++)	for (int i=0; i <= size; i++)	for (int i = 0; i < this.size; i++)	Same
FFC1	if (dataset != null)	if (dataset == null)	if (dataset == null)	Same
FFC2	if (categoryKeys.length != this.startData[0].length)	if (this.startData == null)	if (categoryKeys.length != getCategoryCount())	Difference
FFC3	if (categoryKeys.length != this.startData[0].length)	–	if (categoryKeys.length != getCategoryCount())	–
FCC1	if (excerpt.equals(LINE) && 0 <= charno && charno < sourceExcerpt.length())	if (excerpt.equals(LINE) && 0 <= charno && charno <= sourceExcerpt.length())	if (excerpt.equals(LINE)&& 0 <= charno && charno <= sourceExcerpt.length())	Same
FCC2	if (excerpt.equals(LINE) && 0 <= charno && charno < sourceExcerpt.length())	if (excerpt.equals(LINE) && 0 <= charno && charno <= sourceExcerpt.length())	if (excerpt.equals(LINE)&& 0 <= charno && charno <= sourceExcerpt.length())	Same
FCC3	return false;	return true;	return parent != null && NodeUtil.isGet(parent);	Difference

operators can only fix one bug each. This is extremely suggestive as operators such as ROR and COI are better for methods of repairing programs based on mutations.

D. Analysis on Variable Model

In this subsection, we analyze the variable model based on two similarity metrics for patch prioritization.

In the variable model, we used the overlap coefficient [45] as well as Jaccard to analyze the ranking of the patches. It measures the overlap between two sets. The measure is determined by dividing the size of the intersection by the smaller of the size of the two sets. We found that the ranking result of Jaccard is better than that of Overlap. Table XII shows the comparison between Jaccard and Overlap metrics for the variable model. Jaccard can locate the correct patches in top-1 for 14 bugs while Overlap can locate only for 7 bugs. So, Jaccard got 73.68% in top-1.

V. Discussion, Limitation, and Threats to Validity

A. Discussion

The experiments in Sections 4 demonstrate that our approach can efficiently fix many types of bugs in off-the-shelf Java programs. Our approach performs a mutation to generate patches, indicating that it can only target those bugs that currently require a single repair action. Our experiment was performed on the Defects4J benchmark [22], [36], which is a widely used dataset for studies into automatic program

TABLE VIII
PERFORMANCE OF OUR APPROACH ON DEFECTS4J

Project	Bug_Index	Correct Patch	Rank	Average Time (s)
Math	FCM1	1	34	63.69
Math	FCM3	1	1	26.69
Math	FCM4	1	23	44.57
Math	FCM5	4	1	53.26
Math	FCM6	4	1	55.07
Math	FCM7	4	1	55.46
Math	FCM8	5	1	302.61
Math	FCM9	5	1	45.89
Math	FCM10	1	1	135.12
Lang	FCL1	3	1	87.86
Lang	FCL2	4	1	115.00
Lang	FCL3	4	1	69.52
Lang	FCL4	1	1	108.81
Lang	FCL5	7	1	120.35
Chart	FFC1	2	1	83.86
Chart	FFC2	4	61	47.78
Closure	FCC1	1	1	58.59
Closure	FCC2	1	1	26.97
Closure	FCC3	2	1	47.56

TABLE IX
CODE COVERAGE ANALYSIS OF THE PROPOSED SYSTEM

Bug_Index	Coverage (%)				
	Test	Instruction	Branch	Line	Method
FCM1	72	99	96	96	95
FCM3	65	97	100	96	94
FCM4	82	84	70	87	76
FCM5	85	89	62	84	75
FCM6	85	89	62	84	75
FCM7	85	89	62	84	75
FCM8	91	89	81	89	89
FCM9	65	70	73	82	83
FCM10	80	96	92	97	100
FCL1	84	98	92	98	100
FCL2	76	68	67	72	65
FCL3	76	68	67	72	65
FCL4	85	97	90	98	100
FCL5	84	65	70	68	73
FFC1	85	25	26	28	45
FFC2	79	72	56	73	84
FCC1	93	88	60	87	63
FCC2	93	88	60	87	63
FCC3	82	93	85	94	100

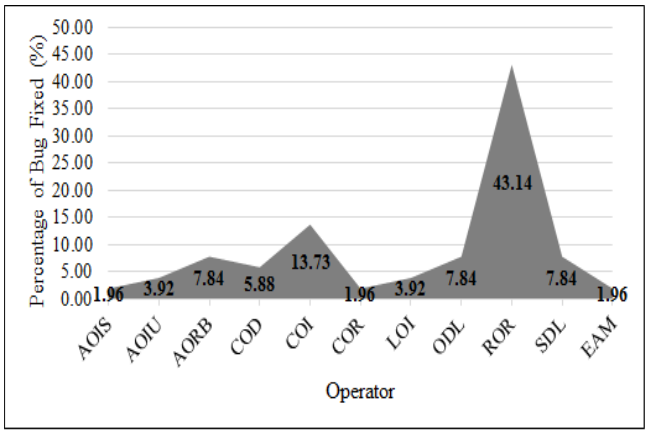


Fig. 4. Fault Fixing Capabilities of Mutation Operators

repair research. It comprises of six big projects developed by various developers and includes 395 bugs in total.

In the experiment, we reference the bug-related developer patch to determine the validity and accuracy of the patch produced by our approach manually. In specific, there are other techniques for defining it. While determining the significance or validity of some of the possible patches produced by our strategy and other methods is not simple, some patches may be valid and correct even if they are not syntactically equal or similar to developer patches.

B. *Limitation and Threats to Validity*

Like most previous test-suite based program repair work, our approach can only address buggy programs where there is a single bug. We cannot currently repair multiple bugs in Java programs. The repair efficiency of our approach significantly depends on how code searches are conducted within mutant programs. With a better code search, our approach can be more efficient and can produce more valid/correct patches.

Our approach depends on the mutation system, so if we can modify the mutation system to produce more mutants, we will be able to get more fixes. Our choice of subject programs is a threat that may limit the generalization of our results. However, we performed our experiments on four projects to alleviate this threat. We evaluated our approaches on 21 buggy programs from four real-world projects of the Defects4J benchmark [22], [36].

VI. Related Works

A. *Search-based Program Repair*

SearchRepair [12] is a system inspired by the code search, was proposed by Ke and colleagues. First, the code fragment is indexed as an SMT constraint, and then the fragment is combined with the required I/O pairs and fragments into a constraint problem at the time of repair. The system is designed by students to evaluate online courses in the Small C programs.

CapGen [13] is a context-aware repair technique, which works at the AST node level to improve the likelihoods of search space incorporating the correct patches. The context information is extracted for fixing ingredients by three different models, such as genealogy, variable and dependency. It used three types of mutation operators to rank the fixing ingredients. It operates in terms of AST nodes at a good granularity. At the expression level, CapGen considers context similarity, dependency similarity and name similarity between the fragment of buggy code and the ingredients. It produces patches from the top of the ranking through statements using the policy of brute force.

In 2014, Debroy et al. [18] proposed an automatic bug fixing technique by combining the concepts of mutation and fault localization. They use Tarantula to locate the bugs, and mutants are generated from the top of the ranking through statement and use policy of brute force. The mutation operators, such as logical, relational, arithmetic and assignment operators, were considered to replace with another operator of the same type, negating if/while conditions were met.

SsFix [19] conducted a syntactic code search and used existing bug repair code fragments to find new code in a code database whose syntax is related to bug contexts. It depends on the Apache Lucene search engine to gather similar code fragments, which have been developed for plain text but programs. The ingredients are then obtained by making a patch from a similar code chunk.

TABLE X
REPAIR RATES OF OUR APPROACH AND OTHER APPROACHES

Project	Our Approach	CapGen	jGenProg	Nopol	ssFix
Math	9	2	3	4	0
Lang	5	0	0	1	0
Chart	2	1	0	0	1
Closure	3	1	1	0	0
Total	19	3	4	6	1
Precision	**90.48**	14.29	19.05	28.57	4.76

TABLE XI
TOP-N COMPARISON AMONG THREE FAULT LOCALIZATION TECHNIQUES

Metric	Top-1	Top-3	Top-5	Top-10
Ochiai	52.38	66.67	66.67	85.71
Tarantula	4.76	4.76	4.76	4.76
Jaccard	19.05	19.05	19.05	19.05

TABLE XII
COMPARISON BETWEEN TWO METRICS FOR VARIABLE MODEL

Metric	Top-1	Top-3	Top-5	Top-10
Jaccard	73.68	73.68	73.68	73.68
Overlap	36.84	42.11	52.63	63.16

Jiang et al. [20] proposed a program repair technique, called SimFix which is considered the structure similarity, method name similarity and variable name similarity between the buggy statement and ingredient. But they remove ingredients that are less frequent in existing patches. From this point of view, SimFix and CapGen are two very similar approaches.

Wang and colleagues [21] proposed an automatic operator error debugging technique. It combines program repair with mutation analysis based on the location of the bug. Depending on their experiments, it can fix the programs correctly with specific errors.

In 2012, GenProg [23] seeks to repair programs without any specifications and to apply genetic programming without the guidance of historical patches to statement level mutation of existing source code. Originally designed for building repair approaches, Astor [34]-[35] was named jGenProg, a Java implementation of GenProg initially intended to repair C code. The concept is to create a program by using repair operators to reach a modified version without the bug.

B. Semantics-based Program Repair

Nopol [9] is a repair approach for conditional statement bugs. It can fix the programs by either adding a precondition (i.e. a guard) or modifying an existing IF condition to statement or block in the code. SMT synthesizes the condition that is modified or inserted. To facilitate repair, several test cases are modified. According to the empirical analysis of these bugs, it can effectively fix bugs with two types of conditions. It has been extended to repair infinite loops as well.

Based on symbolic execution and code synthesis, Nguyen et al. [10] proposed a repair technique called Semfix. They considered the bugs in boolean conditionals and the right-hand side of assignments. Angelic debugging [31] is used to discover the repair location, then synthesize the fixed expression with the input-output components. The same group proposed Angelix [32] to overcome this problem. It is a repair system such as Semfix [33], the symbolic execution stage was significantly designed to scale up to large programs and to gain more than one angelic value, which is the “angelic forest”.

PAR [11] is a program repair for Java bugs. It is based on templates for repairing and a repair template of PAR is a common way to fix a common bug. For example, access to a null point is a common bug, and a common fix is to add a null-ness check just before the undesirable access. It is the template used to test the null pointer exception. The templates are randomly applied and tested.

VII. Conclusion

In this paper, we proposed a search-based program repair technique to detect the bug and fix it based on the type of buggy statement and mutation system called MuJava as quickly as possible. We used both two-level mutation operators of MuJava to find the fixed candidates and prioritized the candidate patches by modeling the variables of the buggy code and fixed candidates. We considered modifying the source codes for its impact on the overall structure of the software system with little consideration. We concentrated on single line bugs, and we evaluated our approach on four real-world projects of the Defects4J dataset. Our approach can produce the correct patch for 19 bugs of 21 bugs, where 9 bugs have never been fixed by existing techniques. We have a plan to evaluate our approach in the future to address more real-world bugs and more types of mutation operators.

References

[1] X. B. D. Le, F. Thung, D. Lo and C. Le Goues, “Overfitting in semantics-based automated program repair,” *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007-3033, 2018.
[2] C. S. Timperley, “Advanced Techniques for Search-Based Program Repair,” *Doctoral dissertation, University of York*, 2017.
[3] Z. Qi, F. Long, S. Achour and M. Rinard, “An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems,” *In Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 24-36, 2015.
[4] Y. S. Ma and J. Offutt, “Description of class mutation mutation operators for java,” *Electronics and Telecommunications Research Institute, Korea*, 2005.
[5] Y. S. Ma, J. Offutt and Y. R. Kwon, “MuJava: a mutation system for Java,” *In Proceedings of the 28th international conference on Software engineering*, pp. 827-830, 2006.
[6] J. Offutt, Y. S. Ma and Y. R. Kwon, “The class-level mutants of MuJava,” *In Proceedings of the 2006 international workshop on Automation of software test*, pp. 78-84, 2006.
[7] Y. S. Ma and J. Offutt, “Description of muJava’s Method-level Mutation Operators”, 2016.
[8] Y. S. Ma and J. Offutt, “Description of method-level mutation operators for java,” *Electronics and Telecommunications Research Institute, Korea, Tech. Rep*, 2005.

[9] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34-55, 2017.

[10] H. D. T. Nguyen, D. Qi, A. Roychoudhury and S. Chandra, "Semfix: Program repair via semantic analysis," *In 2013 35th International Conference on Software Engineering (ICSE)*, pp. 772-781, 2013.

[11] 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*, pp. 802-811, 2013.

[12] Y. Ke, K. T. Stolee, C. Le Goues and Y. Brun, "Repairing programs with semantic code search (t)," *In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 295-306, 2015.

[13] M. Wen, J. Chen, R. Wu, D. Hao and S. C. Cheung, "Context-aware patch generation for better automated program repair," *In Proceedings of the 40th International Conference on Software Engineering*, pp. 1-11, 2018.

[14] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733-752, 2006.

[15] J. H. Andrews, L. C. Briand, Y. Labiche and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608-624, 2006.

[16] E. K. Smith, E. T. Barr, C. Le Goues and Y. Brun, "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*, pp. 532-543, 2015.

[17] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," *In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 702-713, 2016.

[18] V. Debroy and W. E. Wong, "Combining mutation and fault localization for automated program debugging," *Journal of Systems and Software*, vol. 90, pp. 45-60, 2014.

[19] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair." *In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 660-670, 2017.

[20] J. Jiang, Y. Xiong, H. Zhang, Q. Gao and X. Chen, "Shaping program repair space with existing patches and similar code," *In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 298-309, 2018.

[21] T. Wang, J. Xu, X. Su, C. Li and Y. Chi, "Automatic debugging of operator errors based on efficient mutation analysis," *Multimedia Tools and Applications*, pp. 1-18, 2018.

[22] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from Defects4J," *In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 130-140, 2018.

[23] C. Le Goues, T. Nguyen, S. Forrest and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54-72, 2011.

[24] R. Abreu, P. Zoeteweij and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," *In Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, pp. 39-46, 2006.

[25] R. Abreu, P. Zoeteweij and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," *In Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pp. 89-98, 2007.

[26] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273-282, 2005.

[27] F. Steimann, M. Frenkel and R. Abreu, "Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-Based Fault Locators," *In Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 314-324, 2013.

[28] J. Xuan and M. Monperrus, "Learning to Combine Multiple Ranking Metrics for Fault Localization," *In 2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 191-200, 2014.

[29] X. Xie, T. Y. Chen, F. C. Kuo and B. Xu, "A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 1-40, 2013.

[30] P. R. Mateo, M. P. Usaola and J. Offutt, "Mutation at the Multi-Class and System Levels," *Science of Computer Programming*, vol. 78, no. 4, pp. 364-387, 2013.

[31] S. Chandra, E. Torlak, S. Barman and R. Bodik, "Angelic Debugging," *In Proceedings of the 33rd International Conference on Software Engineering*, pp. 121-130, 2011.

[32] S. Mechtaev, J. Yi and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," *In Proceedings of the 38th international conference on software engineering*, pp. 691-701, 2016.

[33] H. D.T. Nguyen, D. Qi, A. Roychoudhury and S. Chandra, "Semfix: Program Repair via Semantic Analysis," *In 2013 35th International Conference on Software Engineering (ICSE)*, pp. 772-781, 2013.

[34] M. Martinez and M. Monperrus, "Astor: Exploring the Design Space of Generate-and-Validate Program Repair beyond GenProg," *Journal of Systems and Software*, vol. 151, pp. 65-80, 2019.

[35] M. Martinez and M. Monperrus, "Astor: A Program Repair Library for Java," *In Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 441-444, 2016.

[36] R. Just, D. Jalali and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java rograms," *In Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437-440, 2014.

[37] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733-752, 2006.

[38] J. H. Andrews, L. C. Briand, Y. Labiche and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608-624, 2006.

[39] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme and A. Roychoudhury, "A correlation study between automated program repair and test-suite metrics," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2948-2979, 2018.

[40] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, vol. 4, pp. 308-320, 1976.

[41] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," *In Proceedings International Conference on Dependable Systems and Networks, IEEE*, pp. 595-604, 2002.

[42] D. Graham, E. Van Veenendaal and I. Evans, "Foundations of software testing: ISTQB certification," *Cengage Learning EMEA*, 2008.

[43] Q. Yang, J. J. Li and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589-597, 2009.

[44] P. Piwowarski, M. Ohba and J. Caruso, "Coverage measurement experience during function test," *In Proceedings of 1993 15th international conference on software engineering, IEEE*, pp. 287-301, 1993.

[45] M. K. Vijaymeena and K. Kavitha, "A survey on similarity measures in text mining," *Machine Learning and Applications: An International Journal*, vol. 3, no. 2, pp. 19-28, 2016.