Combining SBFL with Mutation Testing: Challenges of a Practical Approach

Anna Derezińska[a] and Sofia Krutko
Warsaw University of Technology, Institute of Computer Science, Nowowiejska 15/19, Warsaw, Poland

Keywords: Fault Localization, SBFL, Mutation Testing, MBFL.

Abstract: Research on fault localization techniques focuses on their efficiency and cost reduction. Spectra-Based Fault Localization (SBFL) counts to the mostly used methods. Hybrid approaches have been shown to be beneficial. We discuss the challenges of a revised approach combining SBFL with mutation testing. Certain unconvinced issues of mutation testing have been identified, and a practical framework has been prepared to evaluate different variants of the approach. Building a new repository with faulty programs has been launched in order to study the approach not only on the commonly used legacy programs, but also on faulty programs of more contemporary versions of the Java language and unit tests.

1 INTRODUCTION

Fault Localization in software belongs to very laborious activities and, therefore, different methods have been studied to automate them and support code development and maintenance. Different approaches have been proposed based on program analysis, such as code slicing and its variants, machine learning-based techniques, including various kinds of neural network, statistical debugging, model-based techniques, and a family of methods based on test result spectra. An overview of the approaches to fault localization can be found in (Wong et al., 2016).

Spectra-Based Fault Localization (SBFL) is an approach which collects execution results of test cases: whether a program passed a test or failed. The results are related to program units and code coverage outcomes. This set of data is called the program *spectrum*. Later, the spectrum is used for the calculation of the metrics associated with the units of the program under test. These program units that have the highest metric values are the most suspicious, i.e. are considered as places of possible faults.

Mutation testing has been proposed to evaluate the quality of an existing test suite of a program and also to create new tests. An overview of different topics and achievements of this research area can be found in (Papadakis et al., 2019). Mutation testing is based on the idea of fault injection. One or more faults are injected into the program code. The modified program, called a *mutant*, is run against test cases. If the tests detect the fault(s), these tests are counted as good in revealing faults. Otherwise, additional tests could be developed, if possible. The main limitation of mutation testing is the high cost to apply it in practice. Mutation testing has also been used in some approaches to mutation-based fault localization (MBFL): (Papadakis and Le Traon, 2015), (Moon et al., 2014).

One of the ways to improve fault localization efficiency at a reasonable cost is to combine different approaches to fault localization. It has been studied, for example, SBFL combined with statistical debugging (Jiang et al., 2019), a variety of methods in (Zou et al., 2019), or SBFL followed by mutation testing (Cui, 2020), (Lobo de Oliveira et al., 2018), (Xu, Zou and Xue, 2020), (Dutta and Godboley, 2021).

This paper contributes to the latter approach. We have revisited the combination of SBFL with mutation testing and performed a basic case study. After discussing the results of the case study, we point out some questionable issues in test realization and interpretation, and practical tool support. As a result, a framework for the approach has been prepared that could assist in answering the identified research questions and supporting the evaluation of the code

[a] https://orcid.org/0000-0001-8792-203X

Derezińska, A. and Krutko, S.
Combining SBFL with Mutation Testing: Challenges of a Practical Approach.
DOI: 10.5220/0011093500003176
In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 544-552
ISBN: 978-989-758-568-5; ISSN: 2184-4895

of the newer technology than discussed in the previous research. Moreover, a new repository of contemporary Java programs with real faults has been launched.

The paper is structured as follows. The next Section gives an overview of SBFL methods and some mutation testing approaches. Related work is commented on in Section 3. In Section 4, we explain the main idea of the approach that joins SBFL and mutation testing illustrated with a case study. Then, we discuss the problems and present practical solutions to support the approach. Finally, Section 6 concludes the paper.

2 BACKGROUND

2.1 Fundamentals of Spectrum-Based Fault Localization

Spectrum-Based Fault Localization (SBFL) methods use a statistical approach to evaluate test results (Wong et al., 2016). For each test, a program spectrum contains its outcome (pass or fail) and information about code units covered during its execution. The goal of the analysis is to calculate the expected chance that a code unit includes a fault.

There have been proposed over thirty statistical formulas to compute the *suspiciousness* of each code unit. The summaries of selected formulas can be found, for example, in (Wong et al., 2016) and (Heiden et al., 2019). Based on the calculated metric, we can obtain a ranking of the program elements in descending order of suspiciousness.

As an example, suspiciousness *s(j)* of *j-th* code unit is calculated with the Ochiai metric, one of the popular SBFL methods, using the following formula (Abreu, Zoeteweij, Golsteijn, and van Gemund, 2009):

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

where

- $a_{11}(j)$ – the number of failed test cases that cover unit *j*,
- $a_{10}(j)$ – the number of passed test cases that cover unit *j*,
- $a_{01}(j)$ – the number of failed test cases that do not cover unit *j*.

Following the notation used in unit testing frameworks, in SBFL research, a *failed* test case is one that has caused a program to fail, and a *passed* test case is one that was passed by the code. The latter is also called a *successful* test case (Wong et al., 2016). Depending on the approach, in practice, a code unit can be a code line, a statement, or a code block.

2.2 Mutation Testing in Fault Localization

Fault localization can also use ideas of mutation testing. In mutation testing, a mutant is a modified program. A type of program modification is specified by a mutation operator that typically reflects a programming fault to be injected. Mutation tools implement a set of mutation operators to inject faults and create a set of mutants from an original program. A mutant is killed if at least one of the tests detects the injected fault. If the tests do not reveal any difference in the program behavior, the mutant is said to be alive.

Mutation-based methods identify suspicious mutants and use them to find some faults that were not previously localized in the original program. In general, it is based on the following assumptions (Papadakis and La Traon, 2015):

- Mutants-faults located in the same program statements frequently exhibit a similar behavior,
- Mutants-faults located in diverse program statements exhibit different behaviors.

Evaluation of mutants should help detect which tests are sensitive to faults in a given statement. This could allow us to identify statements that are potentially responsible for failed tests.

A basic method of mutation-based fault localization is Metallaxis-FL (Papadakis and La Traon, 2015). In this method, a mutant is classified as killed if the test result is different from the test run on the original program, i.e. a failed test was changed to passed and a passed test was changed to a failed one. A mutant M1 is said to have the same behaviour as another mutant M2 if M1 and M2 are killed by the same test cases. The degree of similarity in the test cases that kill the mutants M1 and M2 defines the behaviour similarity of these mutants.

Therefore, by measuring the number of mutants killed by passing and failing test executions, one can have an indication of the suspiciousness of those mutants.

However, processing statements for which a mutant was created could be questionable. In Metallaxis-FL, if for a selected statement no mutant was generated, its suspiciousness is treated as the lowest and counted as zero. If for a statement more than one mutant was created, the highest suspiciousness is taken into account. A proposal of our modified approach to processing of results in MBFL has been discussed in Sect. 5.

SBFL methods could also be combined with mutation testing (Sect. 3.3).

3 RELATED WORK

3.1 Spectrum-Based Methods for Fault Localization

Many SBFL methods have been developed that use different formulas of suspiciousness. This raises the question of the adequacy and efficiency of these methods.

Nine SBFL methods have been evaluated in experiments on a set of eight C programs, namely: Jaccard, Tarantula, Ochiai, Sorensen-Dice, Anderberg, Simple-matching, Rogers and Tanimoto, Ochiai 2, Russel and Rao (Abreu, Zoeteweij, Golsteijn, and van Gemund, 2009). In these experiments, for each program, the Ochiai method gave the highest accuracy in fault localization or at least was as good as other methods. On average, the results were 4% better for Ochiai than for other methods.

Five SBFL methods (Tarantula, Ochiai, Op2, Barinel, oraz DStar) have been used in experiments conducted on programs with 3242 artificial faults and 323 real faults (Pearson, et al., 2017). The results showed the superiority of the Op2 formulas over other SBFL methods in the case of artificial faults. However, the experiments also demonstrated that the SBFL results of artificial faults are considerably different from those of real faults. All five methods gave similar results for real faults, but the best results were for DStar and Ochiai.

Research carried out by (Zou, Liang, Xiong, Ernst, and Zhang, 2019) aimed at comparison of different groups of fault localization techniques. SBFL was represented by two methods: Ochiai and DStar. In the top 10 elements with the highest suspicion, 44% of 357 faults were detected using Ochiai and 45% using DStar. In this case, the SBFL methods gave the best results compared to other fault localization methods examined in the paper.

One of the problems is the range of suspicious areas identified by SBFL that should be further examined. In experiments reported in (Heiden et al., 2019), while studding 10 of the most suspicious areas, only 40% of faults could be detected. Therefore, and additional post processing with other methods is recommended.

3.2 Mutation-based Fault Localization

The mutation testing approach has been proposed for fault localization by (Papadakis and La Traon, 2012), and the idea extended to Metallaxis-FL (Papadakis and La Traon, 2015). Similar prerequisites were used in MUSE, a mutation-based approach using another metric to calculate fault localization (Moon et al., 2014). Direct mutation-based approaches are limited by the high execution cost. Therefore, it could be beneficial to pre-select a number of mutants, test cases, test runs, or their combinations.

Mutation testing ideas have also been used in program debugging and fixing of faulty programs (Debroy and Wong, 2010).

3.3 Combination of SBFL with Mutation-based Approaches

There are also several attempts to combine spectrum-based methods with mutation-based methods to locate faults.

In (Cui et al., 2020), a program is analyzed with an SBFL (Ochiai and DStar). Then, mutants of the program are generated and executed with MBFL. Finally, the n-top suspicious statements according to SBFL are re-ranked on the basis of the mutation results using the Metallaxis and MUSE techniques.

The FTMES approach (Lobo de Oliveira et al., 2018) has used only the set of failed test cases to execute mutants and avoided the execution of passed test cases, replacing the killing information with coverage data. It has been shown that the approach presented good solutions when the size of the failed test case set is smaller than the set of passed test cases.

In the hybrid approach presented by (Xu, Zou, and Xue, 2020), three different methods are combined. First, the standard SBFL is used and the corresponding program coverage is recorded. This gives a ranking of the suspiciousness of the statement. Then the k-top predicates (k = 10) are selected for mutation testing. Only failed test cases are run against mutants. Finally, the program slicing is applied.

The experiments on C programs reported in (Dutta and Godboley, 2021) start with calculation of a program spectrum with different SBFL methods (Tarantula, Barinel, Ochiai, and DStar). The programs are mutated and run against tests. The mutation results are combined with the averaged results of SBFL. An obstacle of this MBFL approach is the high cost of mutation testing of whole programs.

Most of the discussed works are limited to one fault in a program.

4 CASE STUDY

In this Section, we present a case study on an approach that combines SBFL with mutation testing.

4.1 A Subject of the Case Study

Case study experiments were carried out on a program which performs a method of data compression via textual substitution Lempel-Ziv-Storer-Szymanski (LZSS) (Storer and Szymanski, 1982), which is a modification of the basic SL77 algorithm. The program has been implemented in Java using the Spring Boot platform. The program can be run in two modes: coding and decoding. It can be called via a command line with appropriate options to control its execution.

The program architecture is divided into two main modules: encoder and decoder. These modules are supplemented with *Encoder* and *Decoder* interfaces. They are implemented by the classes *EncoderImpl* and *DecoderImpl.* The first class realises the whole functionality of the encoder module. The decoder module contains five additional helper classes.

The program has been developed with a set of twenty unit tests. The tests did not deal with the functionality of the whole application, but the case study was limited to a thorough testing of the *Encoder.encode()* and *Decoder.decode()* methods, as well as the selected helper methods. The tests give 100% class coverage, 95% method coverage, and 97% line coverage.

In order to evaluate the discussed approach, a fault has been injected into the *encode*() method of the *EncoderImpl* class on line 40. As a result, three unit tests detected this error and failed.

4.2 Spectrum-Based Fault Localization in the Case Study

In SBFL, suspiciousness degrees are assigned to code extracts according to a program spectrum. A program spectrum can be presented in a tabular form, as in Table 1. A code line is selected as a basic code unit, since most code coverage tools support line coverage. Four test cases are identified in columns. The eight rows correspond to selected code lines that include an executable code. The first column includes the line numbers. In the next columns, line coverages are given, where ‘1’ denotes a line covered by the test in this column, while ‘0’ means that a line was not covered by the test. The last row brings the results of the test execution: 1 – the program passed the test, 0 – the program failed for the test.

Table 1: Part of the spectrum of the LZSS program (*EncodedImpl* class).

Line	Test1	Test2	Test3	Test4
34	1	1	1	1
35	0	0	1	1
36	0	0	1	1
38	1	1	1	1
39	1	1	1	1
40	1	1	1	1
41	1	1	1	0
42	1	1	1	0
Test result	0	0	0	1

Based on the formula of the Ochiai method (Sect. 2.1), the suspiciousness of the program lines was calculated. The selected code lines in Table 1 were covered only by tests 1-4. The suspiciousness of these code lines is shown in Table 2.

Table 2: Suspiciousness of selected lines of the LZSS program (Ochiai SBFL).

Line	Failed tests	Passed tests	Suspiciousness
34	3	1	0.87
35	1	1	0.34
36	1	1	0.34
38	3	1	0.87
39	3	1	0.87
40	3	1	0.87
41	3	0	1
42	3	0	1

None of the tests of other classes has failed; hence, the suspiciousness of other classes of the decoder module equals zero. Therefore, after this first step in fault localization, other classes were excluded except the one that includes an injected fault.

The *EncoderImpl* class consists of four methods and includes 84 lines of executable code. Lines 41 and 42 were assigned the highest suspiciousness, value 1, as they were executed by failed tests only. Next, 38 lines get suspiciousness equal to 0.87, including line 40 that comprises the injected fault. Code lines with the three highest degrees of suspiciousness have been selected to be used in the further evaluation.

4.3 Mutation-based Fault Localization of the Case Study

After using the SBFL method, the code area under concern was bounded to 61 lines of code. To this area, mutation testing was applied. This method requires the creation of many mutants and the execution of tests several times. Therefore, any limitation of the number of suspicious code lines is worthwhile.

Mutants have been generated with (Pitest, 2021), currently the most efficient mutation testing tool of Java programs. The following Pitest mutation operators have been used:

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

The results of the mutation testing for selected lines are shown in Table 3. For each line, mutants are identified that introduced changes in this code area. In the column 'Mutant', the symbol '-' means that no such mutant was generated, while numbers 1, 2, 3 denote consecutive numbers of mutants that refer to this area. In the following columns, the results of tests 1-4 that are run against those mutants are given accordingly. The number '0' means that the program failed the test, '1' the program passed the test.

Table 3: Results of mutant testing for the selected code lines of the LZSS program.

Line	Mutant	Test1	Test2	Test3	Test4
34	1	0	0	0	1
34	2	0	0	0	1
34	3	0	0	0	1
35	-	-	-	-	-
36	1	0	0	0	1
38	-	-	-	-	-
39	1	0	0	0	1
40	1	1	1	1	0
40	2	0	0	0	1
41	1	0	0	0	1
42	-	-	-	-	-

When comparing the results of the mutant tests with the outcomes of the original program (Table 1), we can observe that only for mutant number 1 when referred to line 40 the results for all tests differ from the original ones. For all other mutants, all tests gave the same results as for the original program. Therefore, we could suspect that there is a fault in line 40, which corresponds to the fault introduced before the method evaluation.

4.4 Discussion

The application of an SBFL method as a first step resulted in the limitation of a considered area to 73% of the code under test. Therefore, the mutation testing approach could have been applied to a smaller part of a program and pointed directly at the suspicious area. Taking into account the labor intensity of mutation testing, it positively influences the cost of the approach.

It is also important to note that the considered area has been successfully limited to only one class, which in Java means that the search area is limited to one file. As was shown in (Parnin and Orso, 2011), programmers do not follow the ranked suspiciousness list linearly. They usually start with a file where the highest-ranked element is located and go through all the suspicious elements in the class, even if these elements have a lower ranking value. This means that, with respect to fault detection efficiency, the following situation would be beneficial: even if the highest-ranked element is not the faulty one, it should at least point to the file that includes some faults. This situation also occurred in the case study.

After the mutation test is applied to the selected area, the faulty line has been correctly identified. This could be counted as the success of the combination of two selected methods (Ochiai and Metallaxis). However, in general, this could not be true. In the case presented, the fault was correctly localized because a mutant created has changed an incorrect result to a correct one.

5 CHALLENGES OF A PRACTICAL APPROACH

Based on the experience of the case study, we have identified a few problematic points and addressed research questions. We also discuss tool support issues of the developed framework.

5.1 Obtaining Program Spectra

A crucial prerequisite for SBFL is high code coverage by tests. However, a simple code coverage is not sufficient. In the formulas of all SBFL methods, an important factor is the number of tests that failed and cover the *j*-code unit. In general, to practically apply SBFL methods and calculate statistical values, many test cases should be associated with a program. Moreover, multiple coverage is necessary, which means that many, or at least several, test cases should cover the same code elements.

The SBFL evaluation in a case study was based on the Ochiai algorithm, counted as one of the more efficient methods in several experiments. However, it has not yet been concluded which method is the best. Moreover, in the framework, it could be a good idea to use several SBFL methods and combine their results to select suspicious areas that could be further evaluated in mutation testing.

The spectrum of the LZSS programs was generated with the support of the Intelij Idea Code Coverage Agent. Code coverage could be obtained, for example, through CodeCoverage Agent or JaCoCo. However, commonly used coverage tools report on line coverage but do not provide the information necessary to calculate spectra, that is, which test covered a single line. Therefore, in the case study, the test cases were executed separately and the final information was merged using those separate test results.

Some tools that support spectra-based analysis obtain information from unit tests of Junit3 and Junit4 (GZoltar, 2022), but not from Junit5. In the developed framework, JaCoCo has been incorporated, and the process has been automated to acquire all the spectra also from the currently used Junit5 library.

5.2 Processing of Failed Tests in Mutation Testing

In Metallaxis, the suspiciousness of a statement is 0 if none of the failed tests has been changed to a passed one. This makes the approach hardly useful in many cases. If a method returns a type of limited set of values (like a Boolean or enum), there is a chance to get a correct test result after introducing a mutation that compensates for an existing fault. However, if a result type returned by a method or its behavior is more complex, then it is less probable that a combination of two faults, i.e. an original with the one injected by mutation, give a correct program result. If a fault were in such an expression, the tests would fail.

Therefore, in the framework, the number of tests changed from pass to fail should be counted in suspiciousness evaluation, even though the non-failure test was changed to the introduction of a passed due to a mutation introduction in this expression.

After the execution of the tests, the mutation tool returns the state of execution of a mutant, ‘killed’ or ‘live’, and the information which tests have killed the mutant. This could be used to automate the fault localization process. However, Pitest does not allow us to run mutants against tests that fail in an original program. This could be logical for the typical application of mutation testing, but is a limitation if we want to use the results for fault localization purposes. Moreover, it could be possible that a mutant run with a failed test gives a positive result.

In the Pitest case study, the results were collected for positive tests. Failed tests have been processed by running mutants separately and collecting their results. To automate the process for all types of tests, the functionality of Pitest could be extended.

5.3 Differentiating Failed Tests in Mutation Testing

Another problem is differentiating between various types of failed test results. In mutation testing, we are primarily interested in whether or not a mutant was killed by any test. In the proposed approach, different values obtained from failed tests could be distinguished. In the determination of the final suspiciousness, we could take into account tests that change a program result in the following way:

- from pass to fail,
- from fail to pass,
- from fail to fail with the same result (“fail invariant”),
- from fail to fail with another result (“fail diversely”).

Therefore, a mutation testing platform should also process test outcomes and, then, compare. Analyzing these data could be beneficial, as the influence of the results of a faulty program on the efficiency of the fault location remains an open research question.

Research question 1: How will counting the tests that *failed with another result because of mutation* as tests that *changed from fail to pass* affect fault localization efficiency?

5.4 Mutation Efficiency

An important aspect of mutation testing is its efficiency. To achieve substantial savings in time and space, the platform should provide additional functionalities. The first one is the possibility of precisely identifying the units (lines) of code that should be mutated. The bigger the program, the more costly the addition of a single mutant can be. Hence, the result retrieved from SBFL should be used as a direct line-by-line input for a mutation engine.

Secondly, the optimisation functionality already provided by Pitest should be used. This optimisation excludes from running the tests that do not cover a code unit where a mutation is located. Therefore, the functionality provided by Pitest should be modified, as it is impossible to collect complete coverage for a failed test. That is why, in addition to all tests that pass on the original program and cover the mutant, failed tests should be run for each mutant.

5.5 Re-ranking Code Units for Which No Mutants Were Generated

As mentioned above, the MBFL approach ranks all code units without mutations as code with 0 suspiciousness. However, in the case of combining spectrum- and mutation-based fault localization, these units still have a ranking from the SBFL technique. This might result in the actual faulty statement being pushed to the bottom of the final ranking if there are no available mutation operations for the statement. We have not found any research on whether reusing the SBFL suspiciousness ranking in the final ranking could eliminate these scenarios and improve the overall effectiveness of the combined techniques. This open issue has to be verified.

Research Question 2: Will keeping the SBFL score as the final score when no mutations were created affect the effectiveness of spectrum-mutation-based fault localization?

5.6 Evaluation of the Approach with Contemporary Software

FL approaches are usually evaluated on programs that include artificial or real faults. In (Pearson et al., 2017), the same localization methods have been compared using both types of faults. It was observed that the results of artificial faults were not adequate for evaluating programs with real faults. Therefore, most of the research focused on programs with real faults, eg., those collected in the Defects4J repository (Just, Jalalj, and Ernst, 2014), (Defects4J, 2021).

To overcome certain limitations of Defects4J, other sets of Java programs have been collected, e.g. Bugs.jar (Saha et al., 2018) and BEARS Benchmark (Madeiral et al., 2019).

However, the programs in these repositories are written in Java 8 or earlier versions, and their unit tests are in Junit 3 or Junit 4. In 2021, the 17th version of Java was published. Although previous Java versions are still commonly used, programs in Java before version 8 could be treated as legacy code. Since 2017, unit tests can be written with the Junit5 library. Therefore, a collection of faulty programs has been launched in a new BugsRepo repository to evaluate the approach also on some programs developed using contemporary technology.

Research Question 3: Will evaluation of real faults and newer versions of Java and Junit confirm the results of previous research?

5.7 Framework Design

To address the above-mentioned issues, a new spectrum-mutation-based fault localization framework has been developed. The framework consists of two parts: spectrum processing and mutation processing. The spectrum module uses the JaCoCo Java agent to instrument code classes. It discovers the Junit5 tests, runs them sequentially, and collects after each test the coverage data from the agent. Based on the results, the SBFL suspiciousness ranking is calculated for a range of different metrics. Next, the top N suspicious code units (code lines) are passed to the mutation module.

Mutation testing is performed with a modified Pitest. This mutation tool has been extended with all the required functionalities mentioned in Sections 5.2-5.4. During mutation testing, only the most SBFL-suspicious units are taken into account.

Then, the final suspiciousness ranking is produced.

6 CONCLUSIONS

Selected aspects of a process that combines SBFL with MBFL have been presented. In the case study, the suspicious code was considerably limited by SBFL, and using MBFL to this area only, a previously injected fault was found. To generalize the idea, new recommendations are provided for processing the test results of both approaches, which could improve the efficiency and performance of fault detection. Most of the suggestions refer to interpretation of failed tests in mutation testing. We have formulated three research questions that need further investigation.

A platform has been prepared to effectively support the approach and further research on the selection of different variants of SBFL metrics and the processing of mutation testing results. It would allow us to answer the identified research questions.

We have planned to carry out experiments with real faults using not only legacy projects, as typically in research, but also more contemporary projects in terms of the Java language and Junit library. A new bug repository will help to evaluate the known results, for older Java and Junit versions, on newer applications. Moreover, this supports the development of tools that are compatible with newer technologies. Without current easy-to-use tools, fault localization can hardly be used in real life projects.

In the future, the platform could also be combined with other fault localization approaches (Wong et al., 2016), (Zakari et al., 2020), as well as fault prediction methods (Caulo,2019), (Catal, 2011).

REFERENCES

Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C., 2009. A practical evaluation of spectrum-based fault localization. *The Journal of Systems and Software*, 82(11), 1780-1792. doi:10.1016/j.jss.2009.06.035.

Caulo, M., 2019. A taxonomy of metrics for software fault prediction. *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019, pp. 1144–1147. doi:10.1145/3338906.3341462.

Catal, C., 2011. Software fault prediction: A literature review and current trends. *Expert Systems with Applications*, vol. 38, no. 4, pp. 4626-4636. doi: 10.1016/j.eswa.2010.10.024.

Cui, Z., Jia, M., Chen, X., Zheng, L., Liu, X., 2020. Improving software fault localization by combining spectrum and mutation. *IEEE Access* vol 8, 172296-172307. doi:10.1109/ACCESS.2020.3025460.

Debroy V., Wong, W.E., 2010. Using mutation to automatically suggest fixes for faulty programs. In: *Third International Conference on Software Testing, Verification and Validation*, pp. 65–74. doi:10.1109/ICST.2010.66.

Defects4j on GitHub. [Online] [Accessed 29 Dec 2021] Available from https://github.com/rjust/defects4j.

Dutta A., Godboley S., 2021. MSFL: A model for fault localization using mutation-spectra technique. In: *LASD'2021, Lean and Agile Software Development. LNBIP*, vol 408. pp 156-173, Springer, Cham. doi: 10.1007/978-3-030-67084-9_10.

GZoltar – Java library for automatic debugging. [Online] [Accessed 25 Jan 2022] Available from https:/github.com/GZoltar/gzoltar.

Heiden, S., Grunske, L., Kehrer, T., Keller, F., van Hoorn, A., Filieri, A., Lo, D., 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Journal of Software: Practice and Experience*. 49(8) pp. 1197-1224. doi:10.1002/spe.2703.

Jiang, J., Wang, R., Xiong, Y., Chen, X., Zhang, L., 2019. Combining spectrum-based fault localization and statistical debugging: an empirical study. In: *ASE'19, 34th IEEE/ACM International Conference on Automated Software Engineering.* pp. 502-514. IEEE Comp. Soc. doi.10.1109/ASE.2019.00054.

Just, R., Jalali, D., Ernst, M. D., 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In: *ISSTA'2014, International Symposium on Software Testing and Analysis.* doi:10.1145/2610384.2628055.

Lobo de Oliveira, A.A., Camilo-Junior, C. G., Noronha de Andrade Freitas, E., Rizzo Vincenzi, A. M., 2018. FTMES: A failed-test-oriented mutant execution strategy for mutation-based fault localization. In: *IEEE 29th International Symposium on Software Reliability Engineering*, pp. 155-165. doi: 10.1109/ISSRE.2018.00026.

Madeiral, F., Urli, S., de Almeida Maia, M., Monperrus, M., 2019. BEARS: An extensible Java bug benchmark for automatic program repair studies. In: *SANER'2019, IEEE 26th Conference on Software Analysis, Evolution and Reengineering.* pp. 468-478. doi: 10.1109/SANER.2019.8667991.

Moon, S., Kim, Y., Kim, M., Yoo, Y., 2014. Ask the mutants: mutating faulty programs for fault localization. In: Proceedings of IEEE International Conference on Software Testing, pp. 153–162. doi: 10.1109/ICST.2014.28.

Papadakis, M., Kintis, M., Zhang, Jie, Jia, Y., Le Traon, Y., and Harman, M., 2019. Chapter Six - Mutation testing advances: an analysis and survey. *Advances in Computers*. 112, pp. 275-378. Elsevier. doi:10.1016/bs.adcom.2018.03.015.

Papadakis M, Le Traon Y. 2012. Using mutants to locate "unknown" faults. In: IEEE *Fifth International Conference on Software Testing, Verification and Validation*, pp. 691–700. doi:10.1109/ICST.2012.159.

Papadakis, M., Le Traon, Y. 2015. Metallaxis-FL: Mutation-based Fault Localization. *Software Testing, Verification and Reliability* 25, pp. 605-628. doi:10.1002/stvr.1509.

Parnin C, Orso A., 2011. Are automated debugging techniques actually helping programmers? In: *ISSTA'2011, Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp.199-209. doi:10.1145/2001420.2001445.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: IEEE/ACM *39th International Conference on Software Engineering* (ICSE), pp. 609-620. IEEE doi:10.1109/ICSE.2017.62.

Pitest.org.[Online] [Accessed 29 Dec 2021] Available from: https://pitest.org/.

Saha, R., Lyu, Y., Lam, W., Yoshida, H., Prasad, M., 2018. Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. In: *IEEE/ACM 15th International Conference on Mining Software Repositories (MSR*), pp. 10-13. doi:10.1145/3196398.3196473.

Storer, J.A., Szymanski, T.G., 1982. Data compression via textual substitution. *Journal of the ACM.* 29(4) pp. 928-951. doi:10.1145/322344.322346

Wong, E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42(8), pp. 707-740. doi:0.1109/TSE.2016.2521368.

Zakari, A., Lee, S.P., Abreu, R., Ahmed, B.H., Rasheed, R.A., 2020. Multiple fault localization of software programs: a systematic literature review. *Information and Software Technology*, 124, 106312. doi:10.1016/j.infsof.2020.106312.

Zou, D., Liang, J., Xiong, Y., Ernst, M. D. Zhang, L. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2), 332-347. doi:10.1109/TSE.2019.2892102.

Xu, X., Zou, C., Xue, J., 2020. Every mutation should be rewarded: Boosting fault localization with mutated predicates. In: *International Conference on Software Maintenance and Evolution.* IEEE pp.196-207.