An analysis of test case coverage balancing for fault localization

1 Reinaldo de Souza Júnior
Instituto de Informática - INF
Universidade Federal de Goiás - UFG
Goiânia - GO, Brazil
reinaldo.junior@discente.ufg.br

2 Plinio Sa Leitao-Junior
Instituto de Informática - INF
Universidade Federal de Goiás - UFG
Goiânia - GO, Brazil
plinio@inf.ufg.br

3 Thamer Horbylon Nascimento
Instituto Federal Goiano
Iporá - GO, Brazil
thamer.nascimento@ifgoiano.edu.br

***Abstract*—Context: Spectrum-based defect localization uses code coverage information obtained during testing to abstract patterns that help identify faulty statements. Since the number of test cases that reveal the presence of defects is typically lower than that which do not, such imbalance hinders the discovery of these patterns and, therefore, adds cost to defect localization. Objective: A better balanced test suite is necessary to achieve efficiency in software fault localization. This article explores the following questions: (1) Does balancing test cases reduce the effort involved in software fault localization? (2) If two or more test cases are redundant with respect to code coverage, does eliminating this redundancy improve the effectiveness of fault localization heuristics? Method: The Defects4J benchmark was used and the Smote technique (Synthetic Minority Over-sampling Technique) was applied to balance the test cases with and without removing redundant test cases with respect to code coverage. After balancing, the effectiveness of software fault localization was evaluated using some heuristics. Results: It was revealed that the removal of redundancy with respect to code coverage promoted the effectiveness of the fault localization heuristics when applied after the test case balancing procedure implemented by the Smote technique. Conclusions: The analyses presented in the article revealed the positive possibility of exploring research into the use of test case balancing techniques as a way of improving the effectiveness of fault localization heuristics.**

***Index Terms*—fault localization, test data generation, test case coverage, test case balancing.**

I. Introduction and Related Works

Fault Localization refers to the location of faulty software elements related to defects produced during the execution of test cases. This is a laborious and time-consuming task [9]. The main proposals for automating FL are based on executing test cases and then calculating a measure that translates the probability of each program element being faulty.

The coverage spectrum is defined as a set of data that expresses the test execution behavior in a given of software. In this sense, it makes it possible to visualize which software elements are executed by the test cases and how they may be related to the fault [11]. In general, the main proposals for automating the fault localization process are based on the execution of test cases to infer the probability of each program element being responsible for the faults(s) observed.

From the fault localization process, we can observe that the set of tests (test cases) are critical for evaluating the suspected values of the elements containing the fault. There are always two types of test cases: passed test cases and failed test cases. A sufficient number of passed and failed test cases are needed to carry out the process of calculating the suspect elements [19]. However, in practice, failed test cases are usually much smaller than approved ones, and this phenomenon of class imbalance inevitably leads to a negative impact on the effectiveness of fault localization. Previous research has therefore thoroughly evaluated the asymmetry of test cases [2] [10]. There are some studies related to improving the quality of test suites, such as:

- generating new test cases combined with existing test cases [2].
- applying data balancing to the test cases [20].
- item analysis of the characteristics of test cases that can deteriorate SBFL (Spectrum-based fault localization) metrics [17].
- creation of a specific metric to analyze test cases [14].

In this article, two investigations are conducted: **(1) Does balancing test cases reduce the effort of software fault localization? (2) If two or more test cases are redundant with respect to code coverage, does eliminating this redundancy promote the effectiveness of fault localization heuristics?**

Unlike what has been found in the literature, this article explores how balancing test cases, along with eliminating redundant test cases with respect to code coverage, can impact the effectiveness of software fault localization.

II. Background

A. Software Testing and Debugging

Software testing consists of the dynamic verification that a program provides expected behavior for a finite set of test cases, appropriately selected in the usually infinite execution domain. To do this, the software is executed with predefined inputs, verifying that the output obtained is in accord with the expected output. If this is not confirmed, it can be said that a fault has been identified. Testing is therefore a tool for achieving greater confidence in the operation of software. In addition to tests serving to indicate the presence of a fault, test execution information can also serve as support for locating and correcting the fault found [6].

Debugging is a two-step process that begins when a fault is identified by a negative test, is a test that reveals the existence

2836-3795/25/$31.00 ©2025 IEEE
DOI 10.1109/COMPSAC65507.2025.00316

of a fault [11]. The first step consists of determining the precise location of the fault in the program, while the second step consists of repairing it. Thus, debugging aims to reduce the number of faults in the software and consumes a significant portion of the time spent during the software development cycle [7].

B. Software Fault Localization

Fault localization (FL) is an important part of debugging that must precede the repair of any fault. Among the tasks inherent in debugging, fault localization is one of the most difficult and tedious [8] [16]. Locating faults directly affects the quality and cost of the product and refers to the activity of identifying the location of faults corresponding to faults that became known during the software testing stage.

The main proposals for automating FL are based on executing test cases and then calculating a measure that calculates the probability of each element of the program being faulty.

The traditional techniques, as classified by [18], are: Program Logging, Assertions, Breakpoints and Profiling. Advanced techniques, as classified in the publication, are based on: slices, code spectrum, statistics, program states, machine learning, data mining and other techniques. The aim of advanced techniques is to automate and reduce the human intervention involved, even partially, in the process of locating software faults. The need to develop such techniques is due to the increasing cost and volume of fault localization activities in the face of the growing complexity of software projects.

The automation of the FL process has been the subject of several studies in recent years [18]. With the aim of accurately indicating the location of the fault in faulty programs, some aspects stand out, such as fault localization based on spectrum coverage, two examples of which are the control flow spectrum and the data flow spectrum.

C. Control flow spectrum

The coverage spectrum is defined as a set of data that expresses the execution behavior of the tests in the respective program. In this sense, it allows the visualization of which program elements are executed by the test cases and how they may be related to the [16] fault. In general, the main proposals for automating the fault localization process are based on the execution of test cases to infer the probability of each program element being responsible for the fault(s) observed.

The control flow coverage spectrum (SBFL - Spectrum-based fault localization) is defined as an M × N matrix, obtained from M executions (test cases) that can cover N elements of the program. The success or fault of the execution can be represented as a separate column in the matrix. From the data in this coverage matrix, it is possible to measure the fault propensity for each element. Figure 1 represents the control flow spectrum matrix.

Fault- localization heuristics based on the control flow spectrum are equations that use the values of variables obtained from the control flow matrix to assign program elements a propensity value to be the faulty element. We will use the following notation to indicate these variables, which can be obtained for each program element:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ f_m \end{bmatrix}$$

Fig. 1. Control flow spectrum matrix

- **es**: Number of positive test cases that execute the observed element.
- **ns**: Number of positive test cases that do not execute the observed element.
- **ef** : Number of negative test cases that execute the observed element.
- **nf** : Number of negative test cases that do not execute the observed element.

The results obtained by heuristics can be used as a "guide" for locating faults. Various heuristics have been presented with the aim of improving accuracy. The Figure 2 shows some of these heuristics.

$$Ochiai = \frac{ef}{\sqrt{ef + nf \times (ef + es)}}$$

$$Tarantula = \frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf} + \frac{es}{es+ns}}$$

$$Jaccard = \frac{ef}{ef + nf + es}$$

$$OP2 = \frac{ef - es}{es + ns + 1}$$

$$Barinel = 1 - \frac{es}{es + ef}$$

$$Dstar = \frac{ef^2}{nf + es}$$

Fig. 2. Formulas - Heuristic

D. Data balancing

In many real data sets, the amount of data varies for different classes. This is common in applications where data from a subset of the classes appears more frequently than data from the other classes [1].

From the fault localization process, we can see that the test suite is critical for evaluating the suspicious values of the elements. There are always two types of test cases: passed test cases and fail test cases. No matter which evaluation algorithms are used by fault localization techniques, a sufficient number of passed and fail test cases are required to carry out the process of calculating suspicious elements [19]. However, in practice, fail test cases are generally much smaller than pass test cases, and this class imbalance phenomenon

inevitably leads to a negative impact on the effectiveness of fault localization. Previous research has therefore thoroughly evaluated the asymmetry of [4] and [17] test cases.

The experiment in this article used the Smote (Synthetic Minority Over-sampling Technique) data balancing technique. It proposes an oversampling approach in which the minority class is oversampled by creating "synthetic" examples instead of oversampling with replacement. The minority class is oversampled by taking each sample from the minority class and introducing synthetic examples along the line segments connecting any/all of the k nearest neighbors of the minority class [12].

III. Experiment

A. Methodology

The experiment described in this article was conducted following the methodology.

1) Step I: the Defects4J benchmark was chosen as explained in topic III (B - Benchmarks).
2) Step II: a pre-processing step was conducted on the benchmark as explained in topic III (B - Benchmarks).
3) Step III: the benchmark test case balancing techniques were applied as explained in topic III (C - Baselines).
4) Step IV: the results were analyzed using the tables constructed and explained in topic IV (A - Analysis Perspective I and B - Analysis Perspective II).

B. Benchmarks

The Defects4J project (http://defects4j.org) and its respective programs: Math, Lang, Mockito and Time were chosen as the benchmark. This benchmark was chosen because it is part of the benchmark of the reference article [20] of this research. Table I gives an overview of the characteristics of the programs used.

The experiment in question followed some initial conditions, as a precondition for processing the experiment, as described below:

- only program versions with at least five test cases will be considered.
- will only consider program versions that have at least two positive test cases.
- will only consider program versions that have at least two negative test cases.
- only faults that have their element found in the control flow spectrum matrix will be considered.

After applying the pre-processing conditions, the number of processed versions of each program was reduced. Table II shows a comparison between the number of original versions of each program and the number of versions processed by the experiment proposed in this article.

For the complete extraction of the data from the experiment, here are the access URLs, together with an explanation of the files extracted:

- The URL https://fault-localization.cs.washington.edu/ contains the spectrum matrix file and the file containing the description of the matrix instructions/columns.
- The URL https://bitbucket.org/rjust/fault-localization-data/src/master/analysis/pipeline-scripts/buggy-lines/ has the files containing the instructions with the real defect.

C. Baselines

This article explores how balancing test cases can impact on the effectiveness of fault localization in software. The term balancing refers to the balance between the number of positive and negative test cases. In this sense, two forms of test case balancing were applied, as described below.

- application of the data balancing technique using the Smote technique, as a way of generating new negative test cases, so that we have a more balanced sample of test data.
- elimination of redundant test cases with respect to code coverage, along with the use of the Smote technique.

In order to make it easier to assemble the analysis tables and also to explain the experiment throughout this article, the following nomenclature was created and will be used in both analysis perspectives:

- **original**: test cases in their original version.
- **smoteoriginal**: test cases, applying the Smote data balancing technique.
- **smoteunique**: test cases, applying the following sequence of actions: eliminating the redundant test cases, applying the Smote data balancing technique and eliminating the redundant test cases again.

Was used to apply the Smote technique the following Python implementation package: "imblearn.over_sampling.SMOTE". The following parameters were used to create the Smote instance: random_state=42 and k_neighbors=the smallest number between the number of negative test cases subtracted from one and the number five, as recommended in [12].

In order to better demonstrate the results of the experiment described in this article, it was executed from two analysis perspectives: Analysis Perspective I and Analysis Perspective II, as will be detailed in topic IV (Results and Analysis).

D. Metrics

Given a fault localization technique T and a program P of size N with a single known faulty statement D, a numerical measure of the quality of the fault localization technique can be calculated as follows: (1) run the FL technique to calculate the ordered list of suspect elements; (2) assign n as the rank of D in the ordered list of suspect elements; (3) use a metric proposed in the literature to evaluate the effectiveness of a fault localization technique. [13].

The literature offers different metrics for evaluating the effectiveness of fault localization methods. This article will use the acc@n metric (acc@5, acc@10 and acc@20), which corresponds to the number of faults found among the n elements with the highest suspicion (higher measures are better). This metric calculates the accuracy of a given heuristic in the first n positions of the list of code elements ordered by the suspicions obtained [20].

TABLE I
GENERAL CHARACTERISTICS OF BENCHMARK PROJECTS

Project	Number of Versions	Total (KLOC)	Average (KLOC)	Statements	Test Pass	Test Fail	Test % Fail
Lang	65	22	0.34	52289	5971	124	2.03
Math	106	85	0.80	241521	18155	177	0.97
Mockito	38	6	0.16	67553	25380	120	0.47
Time	27	53	1.96	139056	68097	76	0.11

TABLE II
NUMBER OF VERSIONS FOR EACH PROJECT

Number of versions	Projects				
	Chart	Lang	Math	Mockito	Time
Number of original versions	26	65	106	38	27
Number of versions executed	10	16	26	19	11

E. Heuristics

The two proposed analysis perspectives used in this article were executed using some fault localization heuristics based on control flow spectra, as proposed in [13]. These heuristics are: Ochiai, Tarantula, Jaccard, OP2, Barinel and Dstar. The formula for these heuristics is described in Figure 2.

Machine learning (ML) based fault localization is an approach that has gained emphasis in recent years due to its ability to deal with the increasing complexity of software systems. Different from traditional techniques, which are based on heuristics or manual inspection, machine learning uses statistical models and algorithms to identify patterns in the data that indicate the presence of faults [3]. One promising direction is the integration of LLMs (Large Language Models), where the application of these large language models, such as GPT, for code analysis and fault identification is an emerging area that promises to bring significant advances. These models can understand the context of the code and suggest corrections in a more intuitive way. This type of approach could be applied to this article, as a way of evaluating whether even with balanced test cases the SBFL-based heuristics would remain competitive when compared to the results of applying LLMs.

IV. RESULTS AND ANALYSIS

A. Analysis Perspective I

In order to better present the results, Analysis Perspective I was created. The objective of this analysis perspective is to answer the question (1) proposed by this article: does balancing test cases reduce the effort involved in software fault localization? Therefore, an Table III will be presented with the results of the heuristics for each benchmark project, taking into consideration the test cases in their original form (original) and balanced with the Smote technique (smoteoriginal). By analyzing the results, it is possible to see how much the results of the heuristics are affected by the project used.

1) Heuristic analysis: The Dstar heuristic obtained the best results for the test cases without balancing (original) for the acc@n metric considering all the projects, except for the Lang project and the acc@10 metric where there was a tie between "original" and "smoteoriginal". No heuristic obtained better results than "smoteoriginal" when compared to "original" when evaluating all the projects involved in the experiment. Only the Ochiai heuristic obtained a better result (Lang; acc@10) for the test cases with balancing (smoteoriginal).

2) Analysis by Project: The Mockito project obtained the best results for the test cases without balancing (original) for the acc@n metric considering all the heuristics, except for the OP2 heuristic where there was a tie between "original" and "smoteoriginal". Only the Lang project obtained a better result (Ochiai; acc@10) for the test cases with balancing (smoteoriginal).

3) Analysis by coverage: In the great majority of the results, the test cases without balancing (original) had better results when compared to the test cases with balancing (smoteoriginal). In 52.08% of the results (values marked in bold and underlined) shown in Table III, the scenario with the unbalanced (original) test cases had better results.

B. Analysis Perspective II

In order to better present the results, Analysis Perspective II was created. The objective of this analysis perspective is to answer the question (2) If two or more test cases are redundant with respect to code coverage, does eliminating this redundancy improve the effectiveness of fault localization heuristics? Therefore, an Table IV will be presented with the results of the heuristics for each benchmark project, taking into consideration the test cases in their original form (original) and balanced with the Smote technique and the elimination of redundant test cases with respect to code coverage (smoteunique). By analyzing the results, it is possible to see how much the results of the heuristics are affected by the project used.

1) Heuristic analysis: The Tarantula and Barinel heuristics obtained the best results for the test cases with balancing (smoteunique) for the acc@n metric considering all the projects, except for the Lang project and the acc@5 metric where there was a tie between "original" and "smoteunique". The Dstar heuristic did not show good results for the test cases with smoteunique, there was a tie between the results presented by "original" and "smoteunique".

2) Analysis by Project: The Mockito project obtained the best results for the test cases with balancing (smoteunique) for

TABLE III
COVERAGE FOR PROJECT AND HEURISTIC (ANALYSIS PERSPECTIVE I)

Heuristic	Statement Coverage	Lang		Math		Mockito		Time	
		acc@5	acc@10	acc@5	acc@10	acc@5	acc@10	acc@5	acc@10
OCHIAI	original	**0.06**	0.18	0.07	**0.15**	**0.21**	**0.21**	**0.18**	**0.36**
	smoteoriginal	0.00	**0.25**	0.07	0.11	0.15	0.15	0.09	0.18
TARANTULA	original	**0.06**	0.18	0.07	0.15	**0.26**	**0.31**	0.18	0.27
	smoteoriginal	0.00	0.18	0.07	0.15	0.21	0.21	0.18	0.27
JACCARD	original	**0.06**	0.25	0.07	**0.15**	**0.21**	**0.26**	**0.18**	**0.36**
	smoteoriginal	0.00	0.25	0.07	0.11	0.15	0.15	0.09	0.18
OP2	original	0.00	0.25	0.07	0.11	0.21	0.21	0.09	0.18
	smoteoriginal	0.00	0.25	0.07	0.11	0.21	0.21	0.09	0.18
BARINEL	original	**0.06**	0.18	0.07	0.15	**0.26**	**0.31**	0.18	0.27
	smoteoriginal	0.00	0.18	0.07	0.15	0.21	0.21	0.18	0.27
DSTAR	original	**0.12**	0.31	**0.11**	**0.19**	**0.21**	**0.21**	**0.18**	**0.36**
	smoteoriginal	0.06	0.31	0.03	0.07	0.15	0.15	0.09	0.27

the acc@n metric considering all the heuristics, except for the Jaccard heuristic and the acc@10 metric.

3) Analysis by coverage: In the large majority of results, the test cases with balancing (smoteunique) had better results than the test cases without balancing (original). In 62.50% of the results (values marked in bold and underlined) shown in Table IV, the scenario with the balanced test cases had better results.

C. Analysis Perspective I and Analysis Perspective II

Considering the two investigations conducted in this article, we can carry out the following analyses of the issues investigated:

1) **Does balancing test cases reduce the effort involved in software fault localization?** Balancing the test cases only by applying the Smote technique (smoteoriginal) did not result in effective software fault localization and consequently did not reduce the fault localization effort. This was because applying the Smote technique generated many negative test cases that were redundant in terms of code coverage.
2) **If two or more test cases are redundant with respect to code coverage, does eliminating this redundancy improve the effectiveness of fault localization heuristics?** Balancing the test cases by applying the Smote technique and eliminating redundant negative test cases with respect to code coverage (smoteunique) contributed to the effectiveness of software fault localization. This was due to the elimination of redundant negative test cases, thus improving the balance of test cases.

The generation of redundant test cases in terms of code coverage after balancing the data can be better seen in Table V, by verifying that the "smoteoriginal" values are greater than "smoteunique" considering the "%fail" column, i.e. there are a large number of duplicate negative test cases present in "smoteoriginal", caused by the balancing of the test cases applied. Specifically, version 4 of the Mockito project (Table VI - values marked in bold and underlined), shows that the test cases in their normal (original) state, had 4 negative test cases and by applying the Smote balancing technique, 1303 new negative test cases were produced (smoteoriginal) which were added to the 4 existing test cases. Of these 1307 negative test cases, only 8 are non-redundant (smoteunique) in terms of code coverage.

V. CONCLUSION AND FUTURE WORK

This article explored how the balancing of test cases can impact on the effectiveness of software fault localization, through the use of the defects4j benchmark, used in many scientific experiments on fault localization based on code coverage spectrum. This research showed that the removal of redundancy with respect to code coverage, when applied together with the test case balancing procedure (application of balancing with the Smote technique), improved the effectiveness of the fault localization heuristics, on the other hand, if the test case balancing was carried out in its pure form (only the application of balancing with the Smote technique) the results obtained were not favorable. These results open up an important area for research into the use of balancing techniques for test cases, so that the chosen technique can produce new test cases that differ from each other. It is believed that the generating these new test cases, as a result of the test case balancing process, could further improve the effectiveness of fault localization heuristics.

As a proposal for future work to strengthen the research in this article, here are some suggestions: (1) Using a real-world case study would strengthen the contributions and practical relevance of this article; (2) As a way of better proving the effectiveness of balancing with SMOTE together with the elimination of test case redundancies, it is suggested that new baselines be created, such as: simple duplication of test cases and generation of synthetic test cases by Artificial Intelligence (AI) and (3) The interpretation that a better classification of heuristics does not always translate into less human debugging effort, therefore, suggests a more in-depth study of this hypothesis.

TABLE IV
COVERAGE FOR PROJECT AND HEURISTIC (ANALYSIS PERSPECTIVE II)

Heuristic	Statement Coverage	Lang		Math		Mockito		Time	
		acc@5	acc@10	acc@5	acc@10	acc@5	acc@10	acc@5	acc@10
OCHIAI	original	0.06	0.18	0.07	0.15	0.21	0.21	0.18	**0.36**
	smoteunique	0.06	**0.20**	**0.12**	**0.20**	**0.27**	**0.27**	0.18	0.27
TARANTULA	original	0.06	0.18	0.07	0.15	0.26	0.31	0.18	0.27
	smoteunique	0.06	**0.26**	**0.08**	**0.16**	**0.27**	**0.33**	**0.27**	**0.36**
JACCARD	original	0.06	0.25	0.07	0.15	0.21	**0.26**	0.18	**0.36**
	smoteunique	0.06	**0.26**	**0.12**	**0.16**	**0.22**	0.22	0.18	0.27
OP2	original	0.00	**0.25**	0.07	0.11	0.21	0.21	0.09	0.18
	smoteunique	0.00	0.20	**0.08**	**0.12**	**0.22**	**0.22**	0.09	0.18
BARINEL	original	0.06	0.18	0.07	0.15	0.26	0.31	0.18	0.27
	smoteunique	0.06	**0.26**	**0.08**	**0.16**	**0.27**	**0.33**	**0.27**	**0.36**
DSTAR	original	**0.12**	0.31	**0.11**	**0.19**	0.21	0.21	0.18	0.36
	smoteunique	0.06	**0.33**	0.08	0.12	**0.22**	**0.22**	0.18	0.36

TABLE V
BALANCING FOR PROJECT

Statement Coverage	Lang			Math			Mockito			Time		
	pass	fail	%fail	pass	fail	%fail	pass	fail	%fail	pass	fail	%fail
original	130.31	4.06	3.02	218.96	3.58	1.61	772.16	5.32	0.68	2245.82	4.73	0.21
smoteoriginal	130.31	130.31	50.00	218.96	218.96	50.00	772.16	772.16	50.00	2245.82	2245.82	50.00
smoteunique	103.67	6.20	5.64	117.04	5.17	4.23	616.00	11.83	1.88	1585.55	9.36	0.59

TABLE VI
FAIL TEST CASE FOR VERSIONS (PROJECT MOCKITO)

Coverage	Version																		
	#1	#2	#3	#4	#6	#9	#11	#12	#20	#24	#25	#26	#27	#33	#34	#35	#36	#37	#38
original	26	3	9	**4**	7	3	2	10	8	2	6	5	2	2	2	4	2	2	2
smoteoriginal	938	27	956	**1307**	1082	1101	900	10	1295	1051	1025	778	1073	688	609	628	802	301	100
somoteunique	90	5	32	**8**	9	4	0	3	16	2	9	12	3	3	3	5	3	3	3

ACKNOWLEDGMENTS

The authors would like to acknowledge the support of CAPES – Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Financing Code #001, CAPES/PDPG n. 30/2022 – Programa Emergencial de Solidariedade Acadêmica, and CAPES/PDPG n.17/2023 – Políticas Afirmativas e Diversidade.

REFERENCES

[1] Carvalho A. C. P. L. F., Almeida T. A., Gama, João, Lorena, Ana Carolina, Faveli, Katti, Inteligência Artificial - Uma Abordagem de Aprendizado de Máquina, Editora LTC, 2021.

[2] Campos J.; Abreu R.; Fraser G.; Amorim M., Entropy-based test generation for improved fault localization, IEEE, 2013.

[3] Cetiner M.; Sahingoz O. K., A comparative analysis for machine learning based software defect prediction systems, IEEE 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 2020.

[4] Cheng Gong, Zheng Zheng, Wei Li, Peng Hao, Effects of class imbalance in test suites: an empirical study of spectrum-based fault localization, 36th IEEE Annual Computer Software and Applications Conference Workshops, 2012.

[5] DANIEL P., SIM K. Y., Noise reduction for spectrum-based fault localization, International Journal of Control and Automation, 2013.

[6] Delamaro, M. E. and Maldonado, J. C. and Jino, M., Introdução ao teste de software, Elsevier Editora Ltda, 2007.

[7] Hailpern B., Santhanam P., Software debugging, testing, and verification, BM Systems Journal, 2002.

[8] Jones J. A., Harrold M. J., Stasko J., Visualization of test information to assist fault localization, Internatioal Conference on Software Engineering, ICSE, 2002.

[9] Leitao-Junior P. S., Freitas D. M., Vergilio S. R., Camilo-Junior C. G., Harrison R., Search-based fault localisation: A systematic mapping study. Information and Software Technology, 2020.

[10] Long Zhanga, Lanfei Yana, Zhenyu Zhanga, Jian Zhanga, Chand W.K., Zheng Zheng, A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization, Journal of Systems and Software, 2017.

[11] Myers G. J., Sandler C., The Art of Software Testing, Sons, Inc., 2004.

[12] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, SMOTE: Synthetic Minority Over-sampling Technique, Journal of Artificial Intelligence Research, 2002.

[13] Pearson S., Campos J., Justt R., Fraser G., Abreu R., Ernst M. D., Pang D., Keller B., Evaluating and improving fault localization, CM - 39 International Conference on Software Engineering, 2017.

[14] Perez A.; Abreu R.; Van Deursen A., A test-suite diagnosability metric for spectrum-based fault localization approaches. IEE/ACM 39th Internacional Conference on Software Engineering, 2017.

[15] Ribeiro H. L., On the use of control and data-flow in fault localization, Dissertação de Mestrado - Universidade de São Paulo, 2016.

[16] Silva-Junior D., Leitao-Junior P. S., Localização de Defeitos Evolucionária Baseada em Fluxo de Dados, Dissertação de Mestrado (Instituto de Informática - Universidade Federal de Goiás), 2020.

[17] Wang X., Jiang S., Gao P., Ju X., Wang R. and Zhang Y., Cost-effective testing based fault localization with distance based test-suite reduction, Frontiers of Computer Science, 2017.

[18] Wong W. E., Gao R., Li Y., Abreu R., Wotawa F., A survey on software fault localization, IEEE Transactions on Software Engineering, 2016.

[19] Zhang Z., Xue J., Yang D. and Mao X., Contextaug: model-domain failing test augmentation with contextual information, Frontiers of Computer Science, 2023.

[20] Zhuo Zhang, Yan Lei, Xiaoguang M., Meng Y, Xin X., Improving fault localization using model-domain synthesized failing test generation. IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022.