International Journal on Advanced Science Engineering Information Technology

Vol.12 (2022) No. 5
ISSN: 2088-5334

MT-Forward Algorithm for Prediction of Project Categories Based on Selection of Mutants

Sasa Ani Arnomo [a,*], Noraini Binti Ibrahim [b]

[a] *Department of Information System, Universitas Putera Batam, Batam, Indonesia*
[b] *Universiti Tun Hussein Onn Malaysia (UTHM), Johor, Malaysia*
Corresponding author: [*]sasa@puterabatam.ac.id

***Abstract*— Mutation testing is an effective technique for errors. Although effective, mutation testing has the main limitation that it is awfully expensive because it requires a series of tests on each mutant. Therefore, many researchers have focused on presenting various techniques to reduce the cost of mutation testing. In this study, the MT-Forward algorithm was developed. The concept of MT-Forward is mutation testing that connects the selection and prediction methods. As a result of the selection, the mutant operator can make a more efficient selection of the number of mutants obtained. The shrinkage of the selected mutants was from 1,019 live mutants to 749 live mutants as a priority. There are 15 features used in this study. The required features are obtained from the Locmetrix Test, Coverage Test, and Mutation Testing. The accuracy value of each method is almost the same, which distinguishes this study is the selection of important operators who prioritize improvement programs without reducing the level of accuracy. Where the PMS Method uses 43 Mutant Operators, PMT Method 22 Mutant Operators, while MT-Forward uses 7 mutant operators as priority mutant operators. The method used to evaluate the performance of the algorithm is 5-fold cross-validation. The accuracy of each method is PMS 57.14%, PMT Method 57.14%, and MT-Forward is 95.00%. MT-Forward states that combining mutant selection techniques and project category prediction is important for fixing faults.**

***Keywords*— Prediction; selection; mutants; mutation testing; MT-Forward.**

Manuscript received 22 Jan. 2021; revised 21 May 2021; accepted 8 Jul. 2021. Date of publication 31 Oct. 2022.
IJASEIT is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

I. INTRODUCTION

The quality of a software product is affected by the quality of software testing [1]–[3]. Several of the techniques used is the approach of making the test case [4] and automatic assessment [5], [6]. One of them is mutation testing. A program's large number of mutants causes a high computational cost for mutation testing [7]–[9]. Programmers will find it difficult to kill all mutants [10]. This motivates the research to try the mutant operator selection and predict it. Mutant prediction methods have been developed. In this study, the predictive mutation testing technique was evaluated from the reduction of the mutant operators. There are two types of MUCLIPSE mutant operators: a traditional operator and a class operator. There are many approaches to reducing the cost of mutation testing based on mutant reduction that is being considered and how to consider the number of trials to also be reduced [11], [12]. Many mutants can be tested in a project. So one of the solutions researchers apply is to create a clustering mutant or test case [13]. Previous research has also clustered projects for mutation testing.

Prediction is a systematic process of predicting something to be observed. The mutant prediction process requires an algorithmic classification process [14]. However, some of the applied methods can be developed further by looking at the tested features. The PMT method predicts mutants with AUC values with several features [15], whereas the PMS method predicts mutation testing based on mutation values. Mutation-Aware Fault Prediction performs predictions by taking the value from MCC. Optimization of different features is needed to increase the effectiveness or reduce the cost of mutation testing. Mutation clustering is more difficult to implement than selective mutation or mutant sampling because clustering removes unnecessary mutants [16].

Therefore, this study improves the optimization technique in predicting project categories by considering mutation testing features. The features used in the classification of mutation testing project categories are Project Categories (CP), Covered Line (CL), Coverage Method (CM), Covered Branch (CB), Coverage Complexity (CC), Covered Instruction (CI), Missed Line (ML), Missed Method (MM), Missed Branch (MB), Missed Instruction (MI), Missed

Complexity (MC), Line of Code (LOC), number of mutants killed selected (NSK), number of mutants alive selected (NSA), and Number of Mutants (NM). The algorithm developed is the MT-Forward algorithm that connects the concepts of selection and prediction. The selection stage is useful for reducing testing costs. In addition, prediction is applied to predict the project category on mutation testing. The project categories are divided into three, namely low, medium, and high status. It is based on the category status states the rate of mutants found. It also means determining how much to increase. Meanwhile, this prediction also takes into account the features associated with mutation testing.

II. Materials and Method

A. Mutation Testing

The purpose of mutation testing is to evaluate the scope of the test. The general idea is to inject the false into the original program automatically, run the test again, and expect to kill the mutant [17]. The false code version based on the injection is called a mutant. Mutants are considered alive if the test still passes or the result is the same as the source code execution result but are considered alive if both have the same output [18], [19]. Then to detect a live mutant, a new test case must be made to find the false or mutant code completely equivalent to the original code. The mutation test uses a set of tests to assess effectiveness since the test set must kill all the mutants. A mutation test should be able to ensure that a test suite can detect and find the false of all program developers by comparing the output of the original code and the output of the code mutated against the same test case. The effectiveness of error detection of a test suite depends on the percentage of false that the test suite can detect [20]. There are several mutation testing terminologies such as killed mutants, alive mutants, mutant equivalent, and mutation operators [21]. The mutants that were killed were the mutants detected by the test [22]. Alive mutants are mutants not detected by the test. An equivalent mutant is a mutant that is semantically equivalent to the code under test [23], [24]. Mutation operators are operators that modify code in simple ways.

Applying mutation testing begins with building a mutant test program that creates a test suit. The following are the steps to run the mutation test. The first step is to insert incorrect code into the program source code by creating multiple versions called mutants. Each mutant made has a resemblance to the original program. Mutants containing false will affect the effectiveness of test cases. The second step is to test the case applied to the original and mutant programs. A test case is run to detect false in a program. The third step is to compare the results of the original and mutant programs based on the applied test case. The fourth step is mutant detection. The mutant is said to be killed by the test case if the original program and the mutant program produce different outputs [25]–[27]. Therefore, the test case is good enough to detect a change between the original and mutant programs. The fifth step is the detection of alive mutants. The mutant is said to be alive if the original program and the mutant program produce the same output. In such cases, the test cases are less effective because the goal is to kill all the mutants.

B. Feature in Project Category

Usually, software developers who create applications have a lot of code entered, which requires testing to ensure the app is running as expected. With mutation testing, it is possible to detect many mutants that must be corrected [28]. For this reason, software developers need clustering in project categories which shows which projects are prioritized. The features used are mutation scores [29]. In this study, the feature is not only a mutation score but also involves other features. These features are the number of killed mutants, SLOC-P, and SLOC-L.

The quality of a set of test cases can be determined by calculating the mutation score (MS), where the mutant detected is divided by the total number of mutants [30]–[33]. A mutation test is said to kill a mutant if it finds an output different from the original program. Killed mutants are collected and placed according to project data. The SLOC-P and SLOC-L features are obtained in the Locmetrix test. Line source (SLOC) is a metric that describes the program code size [34]. SLOC has two types, namely physical and logical. Physical SLOC (SLOC-P) is counting the number of lines in the program. Meanwhile, comment lines are not counted. Logical SLOC calculates the number of statements that can be executed to run a program.

C. The Classification Feature in Mutation Testing

Classification of mutation testing in project categories based on mutant operators after process selection. The feature used is project categories, covered instructions, covered line, covered methods, covered complexity, complexity branch, missed instructions, missed a line, missed methods, missed complexity, missed branch, LOC (Lines of Code), number of alive mutants selected, number of kill mutants selected, and number of mutants. In the Project Category (CP) feature, mutation testing projects need to be grouped. This is to make it easier for software developers. When several projects are tested for mutants, it is better to prioritize which ones will be fixed first. Covered Line (CL) is information the java code-covered tool provides, such as line counts in the class file. Class files are compiled first. The line will be counted; at least there are instructions run. Code coverage can be used for platform evaluation for java applications [35]. The coverage Method (CM) is the extent to which a specified coverage item has been performed by the test suite [36]. A method will be counted when there is at least one command executed. Sometimes not all methods are used in application sources. It requires a test on the coverage method [37]. Branch coverage (CB) covers the possible steps in the branching flow control structure followed. The coverage test will record if the expression is Boolean in the control structure and see if the expression matches when it executes [38]. Branch coverage is more impressive because it tends to go deeper into the code than the statement coverage technique. Branch coverage is a metric measuring the results of decisions that are subjected to testing. Information on branch coverage will have at least one branching done. Examples of instructions that are calculated in branch coverage are the if and switch statements. Coverage Complexity (CC) is a method's number of programming paths and is calculated from the minimum obtained. The complexity information in the class file can be computed even if there is no debugging information. The complexity value is closely

related to a class file's unit test. The instructions in the class file are counted as part of the covered instructions information (CI). The instructions must exist in a class file so they can be computed quickly even without debugging information. The code that has been executed is included in the calculation. Missed line (ML) is a calculation where instructions on the line are not executed. Missed methods (MM) are the number of methods that contain files but do not execute. The Missed Branch metric (MB) provides information about the branch (if and switch statement instructions) that were not executed. The covered branch is not being used, so it is entered into the missed branch metric. Missed instruction metrics (MI) provide information on the number of instructions that were not executed. Missed complexity (MC) indicates the number of test cases missing to cover the module completely. Line of Code (LOC) is a large software measurement technique by counting the number of lines of existing program code.

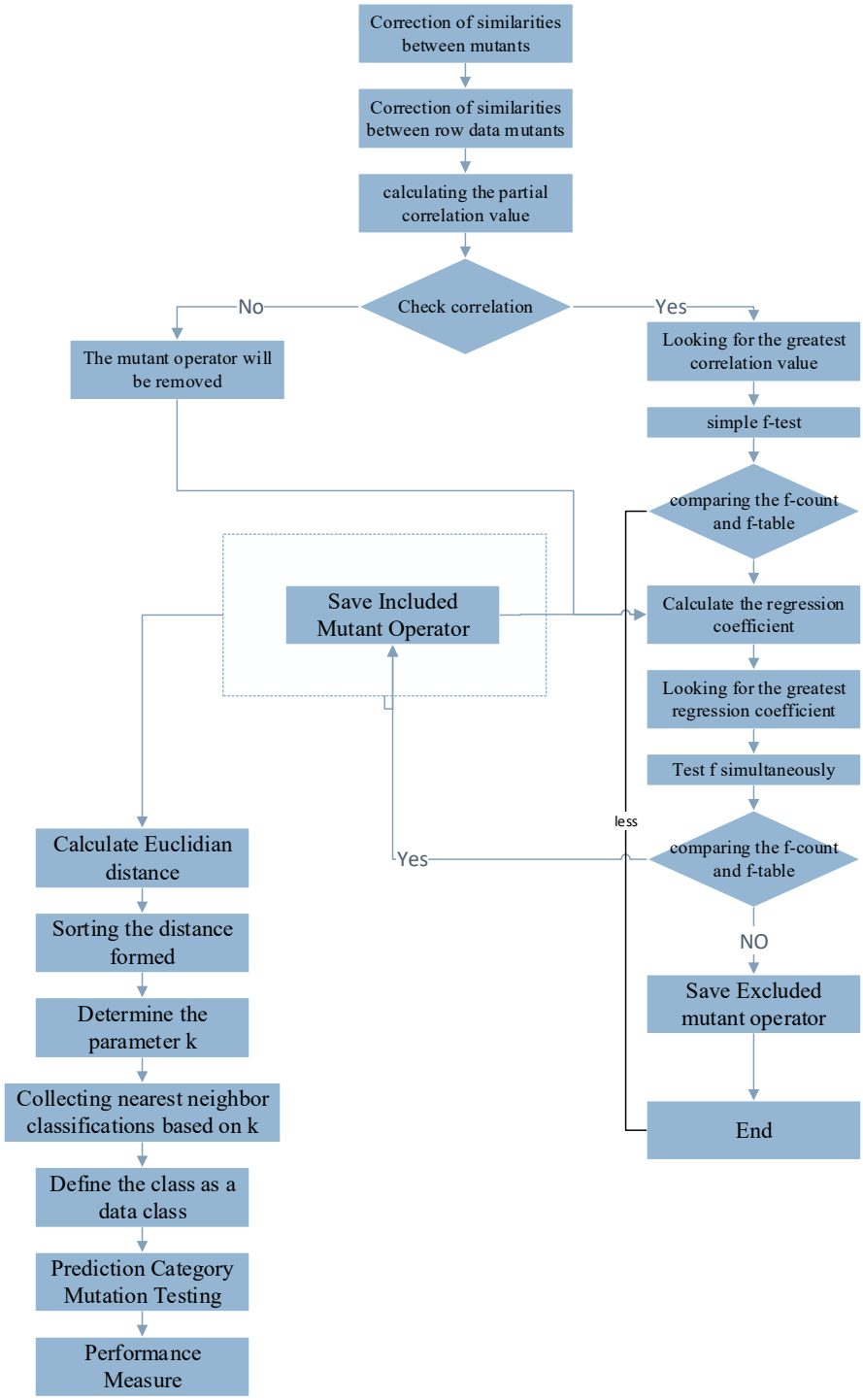

Fig. 1 MT-Forward Algorithm

The LOC method is one of the easiest traditional methods of measuring the quality of a software, although easy the LOC method is quite complicated when studied. LOC is a real proof of what software engineers are doing (in this context it proves how many lines of program a programmer commented there were). In general, better programming skills can use

fewer LOCs and sophisticated programming constructs [39]. The number of killed selected mutants (NSK) is obtained from mutation testing based on the selected mutant operator. There are many mutants killed in mutation testing. But not all mutants that were killed counted for NSK. However, NSK counts only the number of mutants killed from certain mutant types. This is adjusted to the reference mutant. The number of selected alive mutants (NSA) was identified during mutation testing. The NSA counts not all alive mutants. As with NSK, the number of living mutants is calculated based on the type of mutant that is the reference. Number of Mutants (NM) is a mutant that appears after mutation testing. Mutants were counted and grouped. Mutants use several types of operators. Mutation testing using MUCLIPSE has two types of mutant operators: traditional and class operators.

D. MT-Forward Algorithm

MT-Forward is an algorithm that connects the Advanced Choice algorithm with classification. The selected classification is the k nearest neighbor algorithm. An algorithm is a sequence of several logical and systematic steps that are used to solve certain problems. Forward selection is part of the method for feature selection. The forward method is modeling from zero variables. Then the variables are entered one by one until certain criteria are met. After the mutant selection process is complete, it will be connected with the classification process. It uses the nearest k neighbor (KNN). So this method is called MT-Forward. The flow steps of the MT-Forward algorithm are shown in the flowchart as shown in Figure 1.

The MT-Forward algorithm starts from the similarity correction between mutants and the similarity correction between mutant data lines. Next is to calculate the partial correlation value first. Check for non-correlation values. If there is no correlation, the mutants will be eliminated. The mutant operator used is the one with the largest correlation value. The correlation equation used is as follows:

$$r = \frac{n\sum xy - (\sum x)(\sum y)}{\sqrt{\{n\sum x^2 - (\sum x)^2\}\{n\sum y^2 - (\sum y)^2\}}} \quad (1)$$

The x symbol shows the independent variable that is the selected feature. Meanwhile, the y symbol is the dependent variable, namely the project category. The simple f test uses the mutant operator. The next selection process is by comparing the f-count and the f-table. The mutant operator will enter the model if the value of f is greater than the f-table. The regression coefficient is calculated based on the remaining variables. This is done to find the mutant operator which has the largest coefficient. The next step is to test f simultaneously by the mutant operator and the selected mutant from the previous model. The process will be repeated until the value is smaller than the f-table. After the selection process, it enters the prediction process. Data originating from the selected operator will be processed. The f-test formula is shown:

$$F = \frac{R^2/(n-1)}{(1-R^2)/(n-k)} \quad (2)$$

R is the Correlation Coefficient. The n symbol is the amount of data. Meanwhile, k represents the number of independent variables. The next step in the MT-Forward algorithm is classification. The KNN algorithm starts by specifying the value of the k parameter. The Euclidian distance from the training data is calculated. The euclidean distance formula is shown:

$$d_{(i,j)} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad (3)$$

The x symbol is the data record. Meanwhile, y is the data centroid. Data is sorted from high to low values. The next process is to collect the nearest neighbors' classification based on k. The type of class determines the classification results. The final stage is to measure the performance of the model.

III. Results and Discussion

A high Mutation Score means that many mutants detected on the mutation test are killed. So, the programmer just needs a little improvement in the source code being developed. A low Mutation Score means that a small number of killed mutants were detected on mutation testing. So, the programmer needs a lot of improvement on the developed source code. However, the project category clusters involve other attributes to differentiate levels. Apart from the mutation score, the attributes that are entered are the number of killed mutants, SLOC-P, and SLOC-L. Cluster models obtained after iterations are as follows:

TABLE I
ABSOLUTE COUNT CLUSTER COEFFICIENT

Index	Nominal Value	Absolute Count	Fraction
1	cluster_0	28	0.757
2	cluster_1	7	0.119
3	cluster_2	2	0.054

Based on project clustering in Table I, information was obtained that 28 projects were part of the low category. Medium status is 7 projects and in the high category is 2 projects. At the Clustering Evaluation stage, it was carried out using the Davies Bouldin Index (DBI). It determines the optimal number of clusters in the clustering process. The best number of clusters is the cluster that has the lowest DBI value compared to the number of other clusters. The most optimal value in the study amounted to 3 where the DBI value was 0.656. This means that the mutation value is high, medium, and low. MT-Forward starts with forwarding selection. Forward Selection is selecting variables based on correlation coefficients and regressing independent variables X one by one until perfect equations are obtained. The study of conducting a selection using the Forward Selection algorithm. The result obtained is 7 mutant operators from 21 mutant operators. The results of the forward selection obtained are as follows:

TABLE III
THE RESULT OF FORWARD SELECTION

Model		F	Sig.
1	Predictors: (Constant), AOIS	31.479	.000[a]
2	Predictors: (Constant), AOIS, COD	31.348	.000[b]
3	Predictors: (Constant), AOIS, COD, JID	40.692	.000[c]
4	Predictors: (Constant), AOIS, COD, JID, AORB	39.000	.000[d]
5	Predictors: (Constant), AOIS, COD, JID, AORB, COI	43.509	.000[e]
6	Predictors: (Constant), AOIS, COD, JID, AORB, COI, ASRS	52.463	.000[f]
7	Predictors: (Constant), AOIS, COD, JID, AORB, COI, ASRS, LOI	52.303	.000[g]

Table II shows that the mutant selection obtained is AOIS, COD, JID, AORB, COI, ASRS, LOI. Mutants selected operators were collected after obtaining the f test with a significant value below 0.05. Meanwhile, the mutants that were not selected were AORS, AOIU, ROR, COR, IOD, OAN, JTI, JTD, JSI. Unselected mutant operators were collected after the regression test with a significant value above 0.05. Shrinkage of the selected mutants was from 1,019 live mutants to 749 live mutants as priority. It is obtained after selecting operator mutants based on the forward selection method.

The features used in the classification of mutation testing project categories are Project categories (CP), Covered Line (CL), Coverage Method (CM), Covered Branch (CB), Coverage Complexity (CC), Covered Instruction (CI), Missed Line (ML), Missed Method (MM), Missed Branch (MB), Missed Instruction (MI), Missed Complexity (MC), Line of Code (LOC), number of mutants killed selected (NSK), number of mutants alive selected (NSA), and Number of Mutants (NM). The classification feature obtains the average value as shown in the following Figure 2.

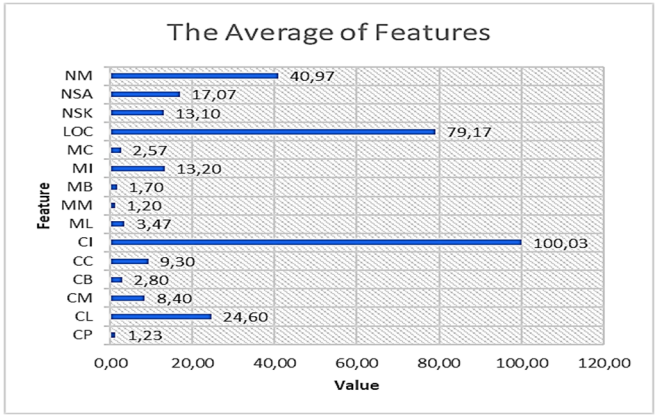

Fig. 2 The Average of Features

Mutant classification uses the KNN algorithm. The majority of classifications that emerge are category 1. Predictive data test results are classified into low mutations. This means that the project under study has a low mutation score, so the mutants live more than the dead ones. The classification method requires measurement so that it can be said to be the best method.

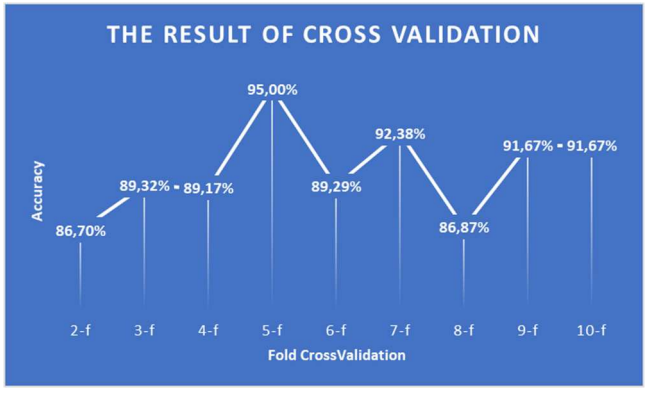

Fig. 3 Accuracy of Cross Validation

The method used is the confusion matrix. The confusion matrix results are compared with the results of other classification confusion matrices. Accuracy is defined as the level of closeness between the predicted value and the actual value. The accuracy value is taken from the correct prediction ratio (positive and negative) with the whole data. The KNN algorithm predicts project categories based on features. The k-optimal search using the k-Fold Cross Validation method can be seen in the following figure 2.

The number of folds that can be maximized in the MT-Forward method is k-fold = 5 with an accuracy of 95.00%. The method that has been generated is compared with other methods. Among the methods being compared are the PMT method and the PMS method. The results obtained are as follows:

TABLE III
COMPARE ALGORITHM

	Kevin Jalbert (PMS Method)	Zhang Jie (PMT Method)	Sasa Ani Arnomo (MT-Forward)
Year	2012	2018	2020
Mutation Operator	43 Operator	22 Operator (14 pit and 8 major)	21 Operator
Reduction	-	-	7 Operator
Feature	4 Feature	14 Feature	15 Feature
Classification	SVM	Random forest	KNN
Accuracy	57.14%	57.14%	95%

The use of many operators to detect mutants will be increasingly influential, with much testing that must be done. The repair will also require a long time for any errors in programming. Table 3 above shows a comparison of methods for predicting mutation testing. The accuracy value of each method is almost the same which distinguishes this research is choosing an important operator that is a priority. Although reduced, it does not reduce the level of prediction accuracy in mutation testing. Where PMS Method uses 43 Mutant Operators, PMT Method 22 Mutant Operators while MT-Forward uses 7 mutant operators as priority mutant operators.

IV. CONCLUSION

A high Mutation Score means that many mutants detected on the mutation test are killed. So, the programmer just needs a little improvement on the source code being developed. A low Mutation Score means that a small number of killed mutants were detected on mutation testing. So, the programmer needs much improvement on the developed source code. However, the project category clusters involve other attributes to differentiate levels. Where apart from the mutation score, the attributes that are entered are the number of killed mutants, SLOC-P, and SLOC-L. Based on clustering, most of the mutation testing project categories were obtained at cluster_0 (Low). There are 28 low level projects, 7 medium level projects and 2 high level projects. The mutant selection and project category clusters were completed and then predicted. The accuracy value of each method is almost the same which distinguishes this research is choosing an important operator that is a priority. Although reduced, it does not reduce the level of prediction accuracy in mutation testing. Where the PMS Method uses 43 Mutant Operators, the PMT Method is 22, while Mutant Operators MT-Forward uses 7 mutant operators as priority mutant

operators. While the accuracy of each method is PMS 57.14%, PMT Method 57.14%, and MT-Forward are 95.00%

ACKNOWLEDGMENT

This research was supported by Universiti Tun Hussein Onn Malaysia. The institution provides facilities in the form of a computer laboratory that helps provide complete software.

REFERENCES

[1] A. Mustafa, W. M. N. Wan-Kadir, and N. Ibrahim, "Comparative evaluation of the state-of-art requirements-based test case generation approaches," *Int. J. Adv. Sci. Eng. Inf. Technol.*, vol. 7, no. 4–2 Special Issue, pp. 1567–1573, 2017.

[2] F. F. Ismail, R. Razali, and Z. Mansor, "Considerations for cost estimation of software testing outsourcing projects," *Int. J. Adv. Sci. Eng. Inf. Technol.*, vol. 9, no. 1, pp. 142–152, 2019.

[3] A. Aghamohammadi, S. H. Mirian-Hosseinabadi, and S. Jalali, "Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness," *Inf. Softw. Technol.*, vol. 129, no. September 2020, p. 106426, 2021.

[4] A. Usman, N. Ibrahim, and I. A. Salihu, "TEGDroid: Test case generation approach for android apps considering context and GUI events," *Int. J. Adv. Sci. Eng. Inf. Technol.*, vol. 10, no. 1, pp. 16–23, 2020.

[5] R. Romli, S. Sarker, M. Omar, and M. Mahmod, "Automated test cases and test data generation for dynamic structural testing in automatic programming assessment using MC/DC," *Int. J. Adv. Sci. Eng. Inf. Technol.*, vol. 10, no. 1, pp. 120–127, 2020.

[6] P. Ma, H. Cheng, J. Zhang, and J. Xuan, "Can this fault be detected: A study on fault detection via automated test generation," *J. Syst. Softw.*, vol. 170, p. 110769, 2020.

[7] J. M. Zhang, L. Zhang, D. Hao, L. Zhang, and M. Harman, "An empirical comparison of mutant selection assessment metrics," *Proc. - 2019 IEEE 12th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2019*, pp. 90–101, 2019.

[8] M. A. Guimaraes, L. Fernandes, M. Ribeiro, M. D'Amorim, and R. Gheyi, "Optimizing Mutation Testing by Discovering Dynamic Mutant Subsumption Relations," *Proc. - 2020 IEEE 13th Int. Conf. Softw. Testing, Verif. Validation, ICST 2020*, pp. 198–208, 2020.

[9] E. Guerra, J. Sánchez Cuadrado, and J. De Lara, "Towards Effective Mutation Testing for ATL," *Proc. - 2019 ACM/IEEE 22nd Int. Conf. Model Driven Eng. Lang. Syst. Model. 2019*, no. label 1, pp. 78–88, 2019.

[10] X. Dang, X. Yao, D. Gong, T. Tian, and B. Sun, "Multi-Task Optimization-Based Test Data Generation for Mutation Testing via Relevance of Mutant Branch and Input Variable," *IEEE Access*, vol. 8, pp. 144401–144412, 2020.

[11] A. V. Pizzoleto, F. C. Ferrari, L. D. Dallilo, and J. Offutt, "SiMut: Exploring Program Similarity to Support the Cost Reduction of Mutation Testing," *Proc. - 2020 IEEE 13th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2020*, pp. 264–273, 2020.

[12] Z. Cui, M. Jia, X. Chen, L. Zheng, and X. Liu, "Improving software fault localization by combining spectrum and mutation," *IEEE Access*, vol. 8, pp. 172296–172307, 2020.

[13] N. Chetouane, F. Wotawa, H. Felbinger, and M. Nica, "On Using k-means Clustering for Test Suite Reduction," *Proc. - 2020 IEEE 13th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2020*, pp. 380–385, 2020.

[14] A. Derezińska, "Soft Computing in Computer and Information Science," vol. 342, 2015.

[15] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive Mutation Testing," *IEEE Trans. Softw. Eng.*, vol. 14, no. 8, 2018.

[16] A. Derezińska, "A quality estimation of mutation clustering in C# programs," *Adv. Intell. Syst. Comput.*, vol. 224, pp. 119–129, 2013.

[17] M. Al-Hajjaji, J. Krüger, F. Benduhn, T. Leich, and G. Saake, "Efficient Mutation Testing in Configurable Systems," *Proc. - 2017 IEEE/ACM 2nd Int. Work. Var. Complex. Softw. Des. VACE 2017*, pp. 2–8, 2017.

[18] Dana H Halabi & Adnan Shaout, "Mutation Testing Tools for Java Programs – a Survey," *Int. J. Comput. Sci. Eng.* , vol. 5, no. 4, pp. 11–22, 2016.

[19] A. Duque-Torres, N. Doliashvili, D. Pfahl, and R. Ramler, "Predicting Survived and Killed Mutants," *Proc. - 2020 IEEE 13th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2020*, pp. 274–283, 2020.

[20] L. Madeyski and N. Radyk, "Judy – a mutation testing tool for Java," *IET Softw.*, vol. 4, no. 1, pp. 32–42, 2010.

[21] S. A. Arnomo, N. B. Ibrahim, and A. Maslan, "Mutation score prediction based on traditional operator of MUJAVA using logistic regression," *J. Adv. Res. Dyn. Control Syst.*, vol. 11, no. 8 Special Issue, pp. 1230–1238, 2019.

[22] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting Trivial Mutant Equivalences via Compiler Optimisations," *IEEE Trans. Softw. Eng.*, vol. 44, no. 4, pp. 308–333, 2018.

[23] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Employing Dynamic Symbolic Execution for Equivalent Mutant Detection," *IEEE Access*, vol. 7, pp. 163767–163777, 2019.

[24] X. Dang, X. Yao, D. Gong, and T. Tian, "Efficiently Generating Test Data to Kill Stubborn Mutants by Dynamically Reducing the Search Domain," *IEEE Trans. Reliab.*, vol. 69, no. 1, pp. 334–348, 2020.

[25] H. Gu, J. Zhang, M. Chen, T. Wei, L. Lei, and F. Xie, "Specification-Driven Conformance Checking for Virtual/Silicon Devices Using Mutation Testing," *IEEE Trans. Comput.*, vol. 70, no. 3, pp. 400–413, 2021.

[26] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, 2018.

[27] P. Delgado-Pérez and F. Chicano, "An experimental and practical study on the equivalent mutant connection: An evolutionary approach," *Inf. Softw. Technol.*, vol. 124, no. April, 2020.

[28] S. A. Arnomo and N. Binti Ibrahim, "Priority path for mutant repairs on mutation testing," *Proc. ICAITI 2019 - 2nd Int. Conf. Appl. Inf. Technol. Innov. Explor. Futur. Technol. Appl. Inf. Technol. Innov.*, pp. 71–76, 2019.

[29] K. Jalbert and J. S. Bradbury, "Predicting mutation score using source code and test suite metrics," *2012 1st Int. Work. Realiz. AI Synerg. Softw. Eng. RAISE 2012 - Proc.*, pp. 42–46, 2012.

[30] M. R. Naeem, T. Lin, H. Naeem, F. Ullah, and S. Saeed, "Scalable Mutation Testing Using Predictive Analysis of Deep Learning Model," *IEEE Access*, vol. 7, pp. 158264–158283, 2019.

[31] M. Deon Bordignon and R. A. Silva, "Mutation Operators for Concurrent Programs in Elixir," *21st IEEE Latin-American Test Symp. LATS 2020*, 2020.

[32] X. Yao, G. Zhang, F. Pan, D. Gong, and C. Wei, "Orderly Generation of Test Data via Sorting Mutant Branches Based on Their Dominance Degrees for Weak Mutation Testing," *IEEE Trans. Softw. Eng.*, vol. 5589, no. c, pp. 1–17, 2020.

[33] G. Grano, F. Palomba, and H. C. Gall, "Lightweight Assessment of Test-Case Effectiveness Using Source-Code-Quality Indicators," *IEEE Trans. Softw. Eng.*, vol. 47, no. 4, pp. 758–774, 2019.

[34] J. Zhang *et al.*, "A Study of Programming Languages and Their Bug Resolution Characteristics," *IEEE Trans. Softw. Eng.*, vol. PP, no. X, p. 1, 2019.

[35] L. Villalobos-Arias, C. Quesada-López, A. Martínez, and M. Jenkins, "Evaluation of a model-based testing platform for Java applications," *IET Softw.*, vol. 14, no. 2, pp. 115–128, 2020.

[36] J. Lee, S. Kang, and P. Jung, "Test coverage criteria for software product line testing: Systematic literature review," *Inf. Softw. Technol.*, vol. 122, no. January, p. 106272, 2020.

[37] S. Yang, S. Huang, and Z. Hui, "Theoretical Analysis and Empirical Evaluation of Coverage Indictors for Closed Source APP Testing," *IEEE Access*, vol. 7, pp. 162323–162332, 2019.

[38] C. Lu, J. Zhong, Y. Xue, L. Feng, and J. Zhang, "Ant Colony System with Sorting-Based Local Search for Coverage-Based Test Case Prioritization," *IEEE Trans. Reliab.*, vol. 69, no. 3, pp. 1004–1020, 2020.

[39] K. Amanullah and T. Bell, "Evaluating the Use of Remixing in Scratch Projects Based on Repertoire, Lines of Code (LOC), and Elementary Patterns," *Proc. - Front. Educ. Conf. FIE*, vol. 2019-Octob, pp. 1–8, 2019.