Regular Paper

Fault Localization with Virtual Coverage and Supervised Learning based on Execution Count Information

Takuma Ikeda†, Hitoshi Kiryu†, Satoshi Suda‡, Shinpei Ogata*, and Kozo Okano*

†Graduate School of Engineering, Shinshu University, Japan
‡Mitsubishi Electric Corporation, Japan
Suda.Satoshi@ay.MitsubishiElectric.co.jp
*Faculty of Engineering, Shinshu University, Japan
{ogata, okano}@cs.shinshu-u.ac.jp

Abstract - Automatic fault localization is a technique that helps to reduce the costly task of program debugging. Among the existing approaches, Spectrum-based fault localization shows promising results in terms of scalability. One Deep Neural Network (DNN)-based SFL approach that uses virtual coverage has been proposed. This approach uses a DNN model that classifies whether the test result of an input code coverage is Pass or Fail. Virtual coverage is code coverage that expresses that only certain code blocks are executed. The output value when the virtual coverage is input to a DNN model is treated as the suspiciousness score. We propose a new virtual coverage and a DNN model based on the number of executions. Our idea is that by using execution count-based virtual coverage, higher accuracy can be achieved than existing approaches. We evaluate our proposed approach using six projects available on Defects4j and Software Infrastructure Repository (SIR). As a result of the evaluation, we confirmed that our virtual coverages improve the accuracy by up to 4.2 points compared to the existing approach. We confirmed that our proposed model with our virtual coverages improve the accuracy by up to 5.6 points.

Keywords: Spectrum-based Fault Localization, Virtual Coverage, Deep Neural Network, Supervised Learning

1 INTRODUCTION

In software development, fault localization is a costly task. Testing and debugging are reported to account for up to 75% of the development cost [1]. Automatic fault localization is an effective technique to reduce the cost of program debugging. Among the existing methods, Spectrum-based fault localization (SFL) has shown promising results in terms of scalability, lightweight, and language-agnostic [2–4].

Ochiai [5] and Tarantula [6] are representative SFL techniques. These techniques calculate a failure suspicion score for each statement based on code coverages. Statements with high suspiciousness scores are considered highly suspicious of failure, and developers investigate statements with high suspiciousness scores as priority. The formulas for calculating the suspicion score for each statement in Ochiai and Tarantula are shown below.

$$\frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}} \quad (Ochiai) \tag{1}$$

$$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f} + \frac{e_p}{e_p+n_p}} \quad (Tarantula) \tag{2}$$

These formulas are calculated using the following values [7].

- e_f is the number of times the statement is executed in the Fail test case.
- e_p is the number of times the statement is executed in the Pass test case.
- n_f is the number of times the statement is not executed in the Fail test case.
- n_p is the number of times the statement is not executed in the Pass test case.

Ochiai and Tarantula calculate a failure suspiciousness score based on the frequency with which each statement is executed in the Fail and Pass test cases. Thus, the idea in the statistical SFL approach is that statements that are executed frequently in the Fail test case and infrequently in the Pass test case are suspicious.

In recent years, several deep learning-based approaches have been proposed for locating faults, and the learning capability of DNNs is effective in locating faults, showing better identification results than conventional SFL techniques (Ochiai, Tarantula) [8]. As one of the deep learning-based SFL approaches, the approach using virtual coverage has been proposed [8–11]. An overview of this approach [11] is shown in Fig. 1. The virtual coverage used in this approach is shown in Fig. 2. The approach in Fig. 1 takes test coverage as input and learns a Deep Neural Network (DNN) model that classifies whether the input test execution coverage is Pass or Fail. Next, the virtual coverage shown in Fig. 2 is input to a learned DNN model, and a DNN model outputs a score indicating the suspiciousness of failure for each code block.

The virtual coverage in Fig. 2 is generated from the test execution coverage and treats the statements commonly executed in all test cases as code blocks. Therefore, the size of

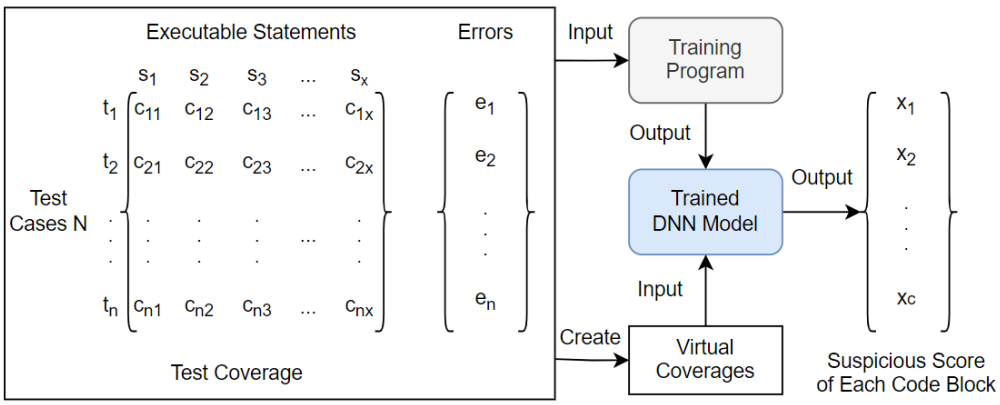

Figure 1: Existing Approach of DNN-based SFL

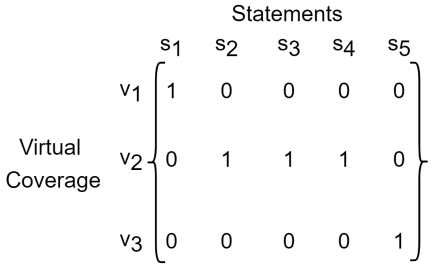

Figure 2: Virtual Coverage at Code-Block Granularity

some blocks can be too large for some programs, which has a negative impact on the accuracy of SFL. We propose a new virtual coverage that can be used with the existing approach shown in Fig. 1. Our proposed virtual coverage is created based on the number of executions (execution count reports) of each statement. Since it is based on the number of executions, it is possible to divide the source code into more code blocks than the existing virtual coverage shown in Fig. 2. Figure 3 shows the creation of existing virtual coverage and the creation of our proposed virtual coverage. In Fig. 3, t_1, t_2, and t_3 represent the three test cases, and $s_1, s_2, ...,$ and s_5 represent statements in the source code. The v_1, v_2, v_3, and v_4 represent the virtual coverages created in each approach. Execution count reports describe execution counts of each statement at test runtime. In Fig. 3, the existing approach divides source code into three code blocks (virtual coverages), and our approach divides source code into four code blocks.

We evaluated our proposed virtual coverage on the six projects (Math, Lang, Chart, Print_tokens, Print_tokens2, and Tot_info) available on Defects4j [12] and Software Infrastructure Repository (SIR) [13]. As a result, we confirmed that the proposed virtual coverage improves the accuracy by up to 4.2 points compared to the existing virtual coverage. Applying the Wilcoxon Signed-Rank Test to the experimental results, we confirmed that the proposed approach is significantly more accurate than the existing approach. In order to further improve the accuracy of our proposed virtual coverage, we also evaluated the proposed virtual coverage on DNN models trained on different training data from the existing approach. As a result, we confirmed that the accuracy is improved by up to 5.6 points compared to the existing approach.

The rest of the paper is organized as follows. Section 2 describes the background of this paper. Section 3 describes the proposed approach. Section 4 describes the experimental setup and Section 5 discusses the experimental results. We conclude in Section 6.

2 BACKGROUND

2.1 Statistical SFL Techniques

Tarantula [6] and Ochiai [5] are representative statistical SFL techniques. These techniques use test coverage collected during test execution to compute suspiciousness scores, which indicate the suspiciousness of failure for each statement. Several approaches [14–16] have been proposed to compute suspiciousness scores similar to Tarantula and Ochiai. Since various metrics have been proposed, the approach [17] to fuse these SFL techniques and calculate suspicion scores from multiple statistical SFL techniques has been proposed.

This paper discusses deep learning-based SFL approaches, which differ from statistical SFL approaches in the idea of fault localization. In this paper, statistical SFL techniques are not further discussed.

2.2 Deep Learning-based SFL Approaches

Existing function-level fault localization techniques [18, 19] use function coverage or statement coverage to compute the suspiciousness values. Murtaza et al. [18]'s approach uses decision trees to identify patterns of function calls related to failures. Sohn et al [19]. proposed the approach to rank faulty methods higher using genetic programming (GP) and linear rank-supported vector machines (SVM). These approaches are function grain SFL approaches, which are different in granularity from the statement granularity SFL approaches discussed in this paper.

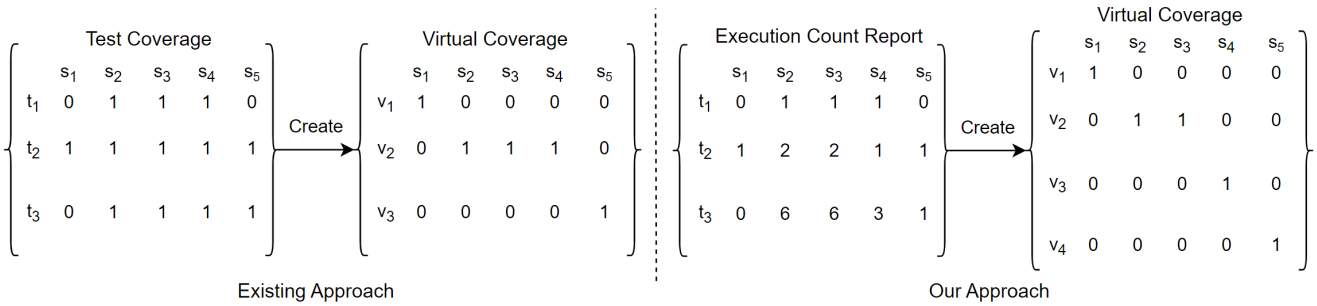

Figure 3: Virtual Coverage Creation in Each Approach

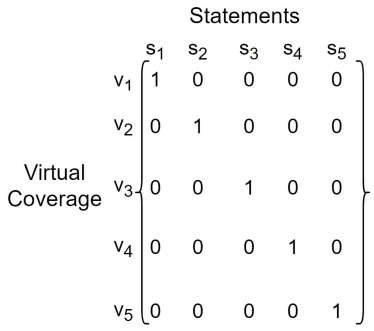

Figure 4: Virtual Coverage at Statement Granularity

As one of the latest techniques for dynamic analysis using machine learning, Li et al. proposed DeepRL4FL, which identifies buggy code by treating fault localization as an image pattern recognition problem [20]. Li et al.'s approach requires marking the statements that are faulty as training data. This is so that a model can distinguish between statements that are faulty and non-faulty at training. In our approach, we treat the label of whether each test case is Pass or Fail as training labels. Therefore, the training data used in Li et al.'s approach is more informative than our approach's training data.

The most relevant researches to this paper are approaches [8, 9, 11] that use virtual coverage to locate faults. In these approaches, a DNN model is first trained that takes test execution coverages as input and classifies whether the test result is Pass or Fail. Next, virtual coverage is constructed that indicates that only certain a statement or a code block is executed. Several approaches have been proposed for constructing virtual coverage, including the approach [8] at the statement granularity and the approach [11] at the code block level. The statement granularity virtual coverage is shown in Fig. 4 and the code block granularity virtual coverage is shown in Fig. 2. By inputting the virtual coverage in Fig. 4 and 2 into a trained DNN model, a failure suspiciousness score is given for each virtual coverage. Since each virtual coverage is a coverage that represents only certain a statement or a code block executed, output values of a DNN model are treated as suspiciousness scores for each statement or code block. It is known experimentally that using virtual coverage at the code block granularity is more accurate for SFL than virtual coverage at the statement granularity [11].

Existing virtual coverage treats statements that are executed at least once in common in all test cases as code blocks. In this approach, the source code is divided into code blocks, which improves the accuracy to the limit of coverage-based SFL. Since the existing approach constructed virtual coverage on a coverage basis, there are possible cases where statements are aggregated in some blocks and the sizes of the blocks are too large. Since our approach constructs virtual coverage based on the number of executions, it is expected that large code blocks can be divided and the accuracy improved.

3 OUR APPROACH

An overview of the fault localization in our approach is shown in Fig. 5 and Fig. 6. Fault localization in our approach consists of the following steps.

1. Execute tests of the System Under Test (SUT).
2. Collect a Coverage Report for each test case at test run time and generate an execution count report.
3. Label the test results (Pass or Fail) in the generated execution count report.
4. A DNN model is trained using generated execution count reports and test result labels.
5. Create virtual coverage from the execution count report.
6. Virtual coverage is input to a trained DNN model and a suspiciousness score is given to each code block.
7. Rank the suspiciousness of each code block in descending order of the suspiciousness score.

Figure 5 shows Steps 3 and 4; Fig. 6 shows Steps 6 and 7. Figure 7 shows the DNN architectures. The coverage report in Step 2 describes information such as how many times each statement of the SUT is executed during each test case. An overview of the execution count report in Step 2 is shown in Fig. 8. The execution count report shown in Fig. 8 indicates how many times each statement of the SUT is executed in each test case. Code coverage is represented as 1 if a statement is executed at least once and 0 if it is not. On the other hand, in the execution count report, each statement is represented by the number of execution at test runtime.

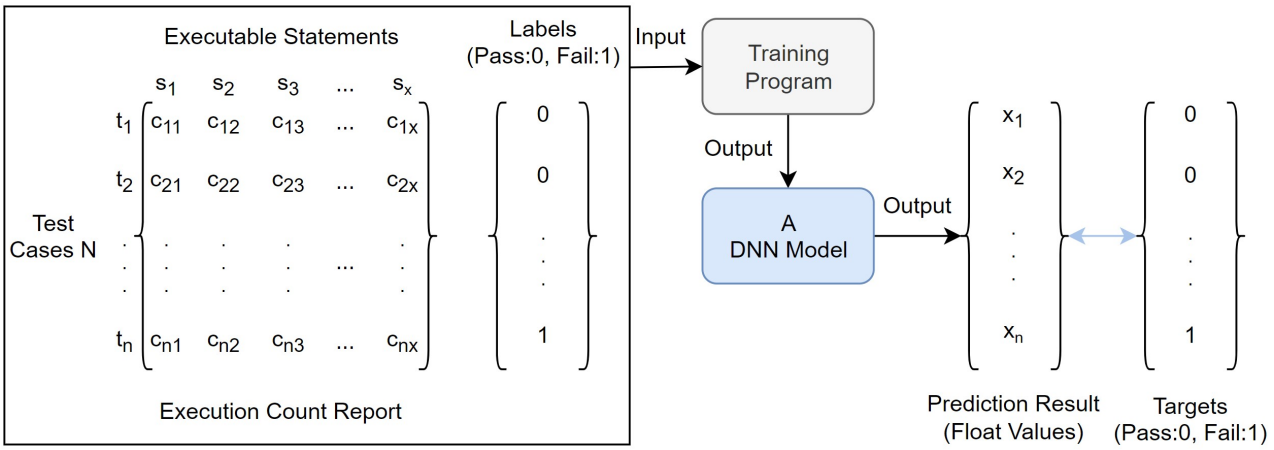

Figure 5: Training A DNN Model in Our Approach

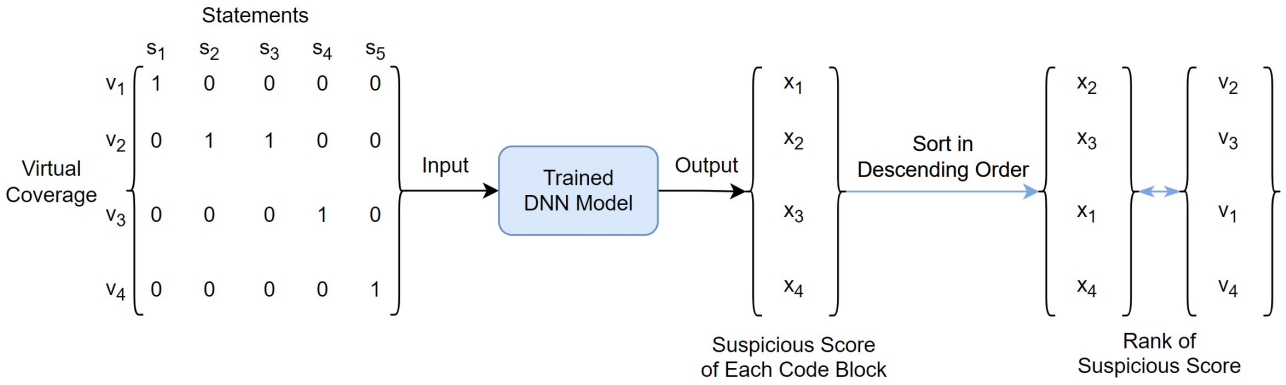

Figure 6: Calculating Suspicious Scores in Our Approach

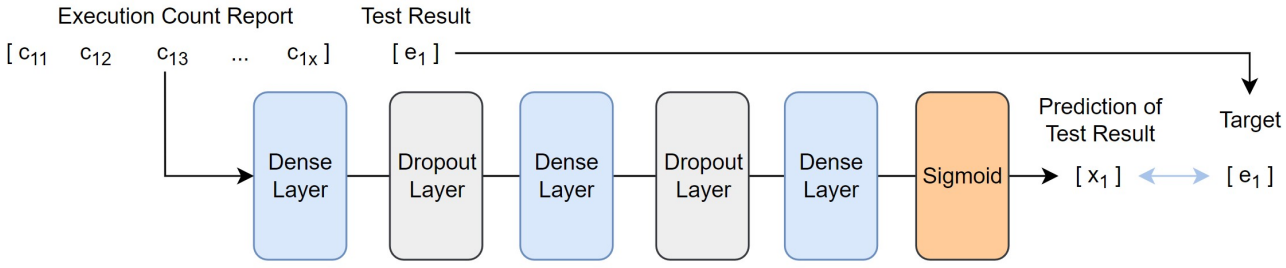

Figure 7: DNN Architecture in Our Approach

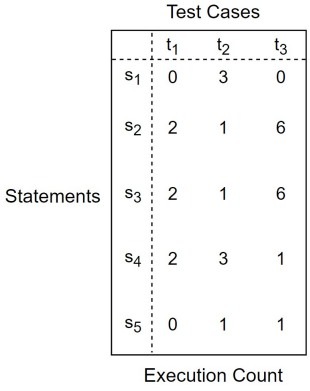

Figure 8: Execution Count Report

In step 4, the developer can use a DNN model that takes the execution count reports as input and classifies the test results. The output value of this DNN model is a float value between 0 and 1, which is regarded as the probability that the classification result is a Fail. A DNN model learns the difference in pattern between the execution count reports of Pass test and Fail test.

The next step is to create virtual coverages that will be input to a trained DNN model. Virtual coverage is code coverage that virtually represents that only a certain code block is executed. The developer creates a virtual coverage and inputs it to a DNN model. Since the output of a DNN model is considered to be the probability that the classification result is Fail, the output value of a DNN model when a virtual coverage containing the faulty statement is input is expected to be higher than other input values. Therefore, the output values of a DNN model in descending order are treated as the rank of the final suspiciousness score, and a suspiciousness rank is assigned to each code block.

The basic idea of SFL (Steps 6 and 7) is the same as the existing approach [11] and is not a new contribution of this paper. We propose a DNN-based virtual coverage fault localization approach based on execution count information. Our proposed approach is used in Steps 4 and 5. The details of our proposed approach are described below.

3.1 Training a DNN Model Using Execution Count Reports

Our approach is to train a DNN model based on execution count information. For this purpose, we use an execution count report. An overview of the execution count report is shown in Fig. 8, which describes the number of times each statement is executed at test runtime.

In our approach, we use execution count reports as training data for a DNN model. Using the execution count reports as training data is expected to improve the accuracy of SFL with our proposed new virtual coverage described in Section 3.2.

3.2 Create Virtual Coverage with Execution Count Reports

The following is a step-by-step description of how to create virtual coverage using execution count reports.

1. In each test case, statements that are adjacent and have the same number of executions shall be temporary blocks.

2. Statements contained in a common temporary block are defined as code block.

3. Create virtual coverage based on defined code blocks.

An overview of each Step is shown in Fig. 9. In Fig. 9, the dotted squares are temporary code blocks and the blue squares are the final code blocks to be defined. In Step 1, temporary code blocks are defined in each test case. In test case 1 (t_1), three temporary code blocks are defined, and in t_2, four temporary code blocks are defined. Each temporary code block is assigned a block id. The color of the dotted line in Step 1 indicates the id of the temporary code block. For example, statement 1 (s_1) belongs to the same temporary code block with the same id in all test cases.

In Step 2, statements that belong to the temporary code block with the same id in all test cases are defined as the final code block. Since s_1 belongs to the temporary code block with the same id (color) in all test cases, we define it as the final code block (blue square). Next, since s_2, and s_3 belong to the same id (color) in all test cases, they are defined as the final code block. At this time, the temporary code block id of s_4 in t_1 is changed to the same id (color) as the next statement, s_5. Next, in Step 2-2, s_4 is defined as the final code block, and the temporary code block id of s_5 in t_1, t_3 is changed to the next temporary code block id (yellow). Finally, in Step 2-3, s_5 is defined as the final code block, and four code blocks are defined in the execution count report shown in Fig. 9.

Our proposed virtual coverage treats statements that are common execution count patterns in all test cases as code blocks. Since code blocks can be created according to the actual execution patterns, source code can be divided into more code blocks than the existing approach. The difference between our proposed code block and the existing approach [11] is shown in Fig. 10.

In Fig. 10, our approach creates four code blocks, while the coverage-based existing approach creates three code blocks. By creating code blocks according to the pattern of execution counts, code blocks can be created with a finer granularity than the existing approach.

The virtual coverage is created from the code blocks and the fault location is identified (Steps 6 and 7). We believe that we can achieve higher accuracy than the existing approach by creating virtual coverage with a finer granularity than the existing approach, and by training a DNN model with data that is more suitable for the proposed virtual coverage.

4 EVALUATION EXPERIMENT

4.1 Research Questions

In this paper, the following research questions are investigated in the evaluation experiment.

RQ1. Comparison of fault localization accuracy with existing approach [11] when virtual coverages created based on

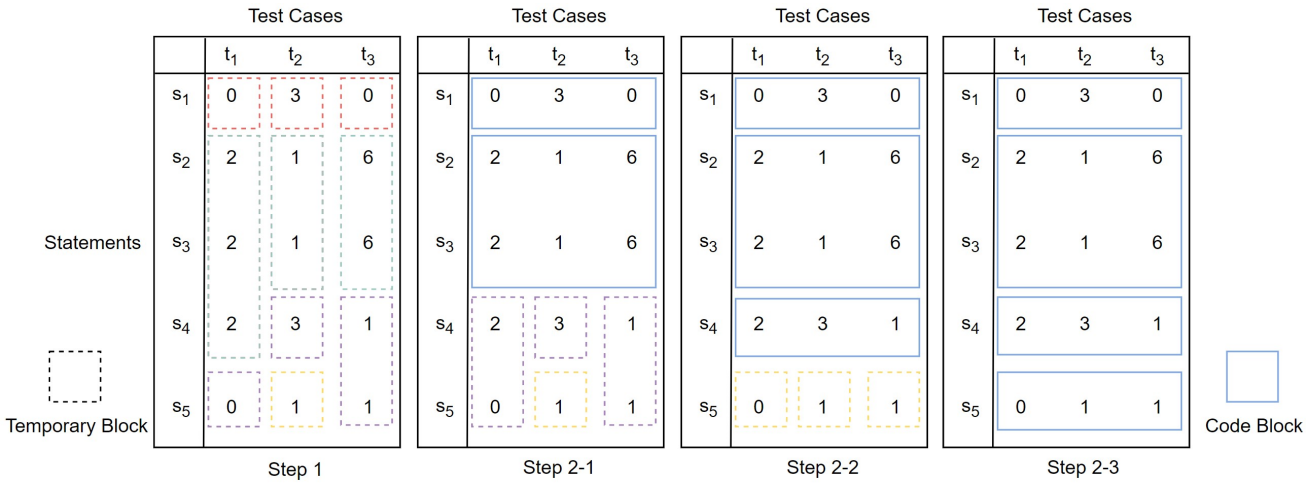

Figure 9: Steps in The Creation of Our Proposed Code Block

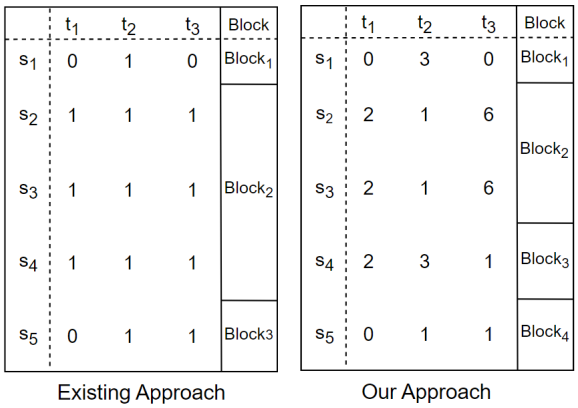

Figure 10: Differences in Code Blocks

the number of executions are input to a DNN model trained with coverages.

In this paper, we propose a new approach for creating a new virtual coverage. We evaluate the effectiveness of our proposed virtual coverage by comparing its fault localization accuracy with existing coverage-based virtual coverage.

The TopN % is used as a measure of the accuracy of fault localization, where the TopN % represents that a bug is classified into the top N % of the total, and a smaller value indicates a better fault localization performance. In the case of multiple bugs, the largest TopN % is used.

RQ2. Comparison of fault localization accuracy with existing DNN model when virtual coverage created based on the number of executions is input.

In our approach (Fig. 5), we use execution count reports as training data to enhance the effectiveness of our proposed virtual coverage. We input the proposed virtual coverage into the existing DNN model and our DNN model, and compare the accuracy of fault localization.

4.2 Subject Programs

We conducted an evaluation experiment using bugs and their fixes provided by Defects4j [12] and Software Infrastructure Repository (SIR) [13]. Defects4J is a database of actual bugs in Java projects. Lang, Math, and Chart from Defects4j and Print_tokens, Print_tokens2, and Tot_info from SIR are selected as the projects for the experiments. We use bugs that meet the following conditions for our experiments.

- Bug fixes only with code addition are excluded. If the bug is fixed by code addition only, the original buggy source code does not have any defects to be pointed out.
- The fault statement is executed at least once in each of the fail and pass tests: the SFL approaches require the bug to be executed in those tests.

Execution coverage and execution count reports are collected using OpenClover [21] and gcov [22]. We use OpenClover to collect coverages of Defects4j's projects, and we use gcov to collect coverages of SIR's projects. Because execution is not recorded for class member variable definitions, etc., due to OpenClover's specifications, we excluded parts of the program that are not recorded as execution coverage from the fault set. The number of lines (LOC) and number of tests for the experimental program are shown in Table 1.

Table 1: Details of Target Programs

Project	Number of Versions	LOC	Number of Tests
Lang	30	58389	54987
Math	29	23623	83364
Chart	15	10094	27036
Print_tokens	7	336	4130
Print_tokens2	9	343	2064
Tot_info	23	268	1052

Figure 11: Results in RQ1 of Defects4J

4.3 Setup for a DNN Model

In performing supervised training, the weight parameters of Dense Layers are initialized with random values. The size of the hidden layer of each Dense Layer is changed according to the size of the coverage size to be trained. The first Dense Layer is the minimum size of 100, and the second Dense Layer is the minimum size of 20 hidden layers. The third Dense Layer is the Output Layer, which is a hidden layer of size 1. The ReLu and Sigmoid functions are used as activation functions. In addition, a Dropout Layer is used to suppress over-learning. Adam optimizer [23] is used, and the learning rate is set to $1.0e-3$. TensorFlow (ver. 2.12.0) is used as the learning framework.

5 RESULTS AND DISCUSSIONS

5.1 RQ1: Result

Figure 11 shows the experimental results for RQ1 of Defects4J. The horizontal axis in Fig. 11 represents the TopN % and indicates the amount of source code examined by the developer. The vertical axis shows the percentage of identified faults, where 100 % on the vertical axis means that all faults are identified. For example, a vertical axis plot with a Top 50% is more than 90%, indicating that more than 90% of the faults are identified by investigating half of the source code. There are two plots in Fig. 11: the gray plot shows the accuracy of the existing approach, and the orange plot shows the accuracy of the proposed virtual coverage.

As a result, except for the 25% and 35% TopN % plots, the proposed virtual coverage has a larger value in the vertical axis plot. This means that for each TopN %, the number of faults that can be identified is higher than with the existing approach.

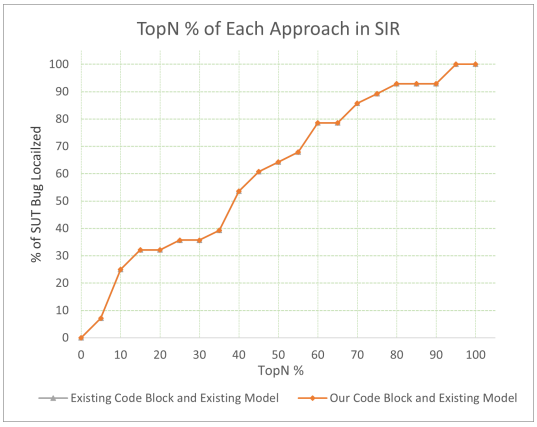

Figure 12: Results in RQ1 of SIR

Table 2: Test Results in RQ1

Subject	1-tailed (left)	1-tailed (right)
Defects4J	3.534E-02	9.647E-01
SIR	5.671E-02	9.433E-01

Figure 12 shows the experimental results for RQ1 of SIR. In Fig. 12, there is no difference in accuracy between the existing and proposed approaches in SIR. Figure 12 shows the accuracy at a granularity of Top 5%, so there is no difference at all, but at a finer granularity, the proposed approach is slightly more accurate ($< 5\ points$) than the existing approach.

Table 2 shows the results of adapting the Wilcoxon Signed-Rank Test to the experimental results. The Wilcoxon Signed-Rank Test is a nonparametric test that tests for differences between two corresponding groups. The null hypothesis indicates that there is no significant difference between the two groups, while the alternative hypothesis indicates that there is a significant difference between the two groups. The 1-tailed test and the alternative hypotheses are listed below.

- **1-tailed (left):** The proposed approach has a smaller TopN % value than the existing approach.
- **1-tailed (right):** The proposed approach has a larger TopN % value than the existing approach.

Since the value of TopN % is the amount of code investigated by the developer to identify the fault location, a smaller value indicates a high accuracy. Therefore, if the left-tailed Wilcoxon Signed-Rank test is accepted, the proposed approach is significantly better than the existing approach in fault localization performance.

The test results of Defects4J show that $p = 0.035 < 0.05 = \alpha$, so the proposed virtual coverage is significantly more accurate than the existing approach. However, the test results of SIR show that $p = 0.057 > 0.05 = \alpha$, so the proposed virtual coverage is not significantly more accurate than the existing approach.

Table 3: Percentage Increase of The Number of Code Blocks in the Defects4J Project

Project	Bugs with Improved Accuracy	Bugs with Decreased Accuracy
Lang	52.36 %	98.54 %
Math	65.34 %	73.01 %
Chart	21.00 %	32.69 %

Table 4: Average Number of Source Code Divisions in Each Project

Project	Existing Approach	Our Approach
Lang	143.86	294.93
Math	80.68	120.93
Chart	65.93	78.33
Print_tokens	120.25	120.25
Print_tokens2	140.63	140.63
Tot_info	72.72	74.00

5.2 RQ1: Discussions

In our approach, the source code is divided into many more code blocks than the existing approach. We consider that defining code blocks with a finer granularity than the existing approach according to the actual execution pattern (number of executions) contributed to the improved accuracy shown in Fig. 11.

On the other hand, for some bugs, the proposed approach showed lower accuracy than the existing approach. Table 3 shows the percentage of increased number of code blocks for each project when the proposed approach achieved higher accuracy than the existing approach, and the proposed approach showed lower accuracy than the existing approach. In Table 3, for all three projects, bugs (Version) with improved accuracy over the existing approach show a smaller percentage increase in the number of code blocks than those with decreased accuracy. Existing research [11] has reported that virtual coverage using code blocks is more accurate than a statement-based virtual coverage approach. Therefore, our approach can define code blocks at a finer granularity than existing virtual coverage, but we consider that the accuracy decreases when the granularity is too fine. In particular, if the granularity of the code blocks defined by our approach is as fine as the statement level, the accuracy is expected to decrease. Future work is needed to determine the level of granularity that will improve accuracy the most.

The results in Fig. 12 show that the proposed approach has no difference in accuracy compared to the existing approach in the SIR project. Table 4 shows the average number of source code divisions (average number of code blocks) for the existing and proposed approaches in each project. In Table 4, the three Defects4j projects (Lang, Math, and Chart) show a difference in the number of source code divisions between the existing and proposed approaches. On the other hand, in the three projects of SIR (Print_tokens, Print_tokens2, Tot_info), there is almost no difference in the number of source code divisions. Therefore, it is considered that the proposed virtual coverage by itself cannot achieve higher accuracy than the existing approach for programs where the number of source code divisions is almost the same as the existing approach.

Figure 13: Results in RQ2 of Defects4J

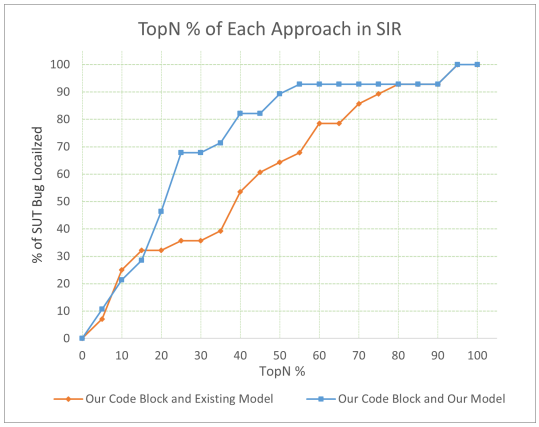

Figure 14: Results in RQ2 of SIR

5.3 RQ2: Result

Figure 13 shows the experimental results for RQ2 of Defects4J. The orange plots in Fig. 13 show the fault localization accuracy when our virtual coverage is input to an existing DNN model. The blue plot shows the accuracy when our proposed virtual coverage is input to a DNN model trained with the execution count reports. The values shown in the two plots are almost identical, and there is no improvement in fault localization performance using our proposed DNN model.

Figure 14 shows the experimental results for RQ2 of SIR. The proposed approach identifies more faults than the existing approach in SIR.

Table 5 shows the results of adapting the Wilcoxon Signed-Rank Test to the experimental results in RQ2. The test results of Defects4J show that $p = 0.421 > 0.05 = \alpha$, so the proposed DNN model is not significantly more accurate than the existing approach. The test results of SIR show that

Table 5: Test Results in RQ2

Subject	1-tailed (left)	1-tailed (right)
Defects4J	4.205E-01	5.795E-01
SIR	1.818E-03	9.982E-01

$p = 0.00182 < 0.05 = \alpha$, the proposed DNN model is significantly more accurate than the existing approach in SIR projects.

5.4 RQ2: Discussions

In Fig. 13, our proposed DNN model did not improve the accuracy compared to the existing DNN model. In Defects4j, we consider that the accuracy based on our proposed virtual coverage (orange plot) is the maximum performance of the SFL method [8] and that the accuracy cannot be improved any further.

Figure 14, the experimental results in SIR, shows that using our DNN model significantly improves the accuracy compared to the existing DNN model. In addition, Table 5 shows that accuracy is significantly improved by using our DNN model. Our proposed virtual coverage is created based on the number of executions of each statement. We consider that training the DNN model with the information used to create our virtual coverage contributed to the improvement in accuracy shown in Fig. 14.

Defects4j showed no improvement in accuracy with our DNN model, while SIR showed a significant improvement (Table 5). It is a future challenge to investigate for which programs our DNN model is effective.

6 CONCLUSION

In this paper, we propose a new virtual coverage to be used for existing DNN-based SFL approaches. Our proposed virtual coverage is created based on the number of executions and is able to divide the source code with finer granularity than the existing virtual coverage. In order to improve the accuracy of fault localization using the proposed virtual coverage, we also proposed a DNN model using execution count reports as training data. Since the proposed virtual coverage is created based on the execution count reports, we consider that training a DNN model with the execution count reports enhances the effectiveness of our proposed virtual coverage.

Our approach is evaluated with six different projects available on Defects4j and SIR. Experimental results show that our proposed virtual coverage is more accurate than existing virtual coverage. We also obtained the prospect of further improving the accuracy by using our proposed DNN model.

Further evaluation of our approach in broader and larger SUTs is needed and is a topic for future work. In the future, we intend to improve a DNN model in our approach to identify faults that cannot be addressed (e.g., faults for which the only bug fix is to add code, performance bugs, etc.).

ACKNOWLEDGMENT

Part of this work is supported by fund from Mitsubishi Electric Corp. The research is also being partially conducted as Grant-in-Aid for Scientific Research C (21K11826).

REFERENCES

[1] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," (2002).

[2] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," (2016).

[3] A. Perez and R. Abreu, "A qualitative reasoning approach to spectrum-based fault localization," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, (2018), pp. 372–373.

[4] Q. I. Sarhan and A. Beszédes, "A survey of challenges in spectrum-based software fault localization," *IEEE Access*, vol. 10, pp. 10 618–10 639, (2022).

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

[6] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, (2002), pp. 467–477.

[7] G. Laghari, K. Dahri, and S. Demeyer, "Comparing spectrum based fault localisation against test-to-code traceability links," in *2018 International Conference on Frontiers of Information Technology (FIT)*, (2018), pp. 152–157.

[8] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, "A study of effectiveness of deep learning in locating real faults," *Information and Software Technology*, vol. 131, p. 106486, (2021).

[9] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an rbf neural network," *IEEE Transactions on Reliability*, vol. 61, no. 1, pp. 149–169, (2012).

[10] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," *J. Softw. Evol. Process*, vol. 33, no. 3, mar (2021).

[11] H. Kiryu, S. Ogata, and K. Okano, "Improve measuring suspiciousness of bugs in spectrum-based fault localization with deep learning," in *Proceedings of International Workshop on Informatics*, ser. IWIN '22. Kii-Katsuura, Japan: Informatics Laboratory, (2022), pp. 3–8.

[12] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, (2014), pp. 437–440.

[13] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, pp. 405–435, 10 (2005).

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

[15] W. Masri, "Fault localization based on information flow coverage," *Software Testing, Verification and Reliability*, vol. 20, no. 2, pp. 121–147, (2010).
[16] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, (2014).
[17] Lucia, D. Lo, and X. Xia, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, (2014), pp. 127–138.
[18] S. Murtaza, N. Madhavji, M. Gittens, and A. Hamou-Lhadj, "Identifying recurring faulty functions in field traces of a large industrial software system," *Reliability, IEEE Transactions on*, vol. 64, pp. 269–283, 03 (2015).
[19] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, (2017), pp. 273–283.
[20] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, (2021), pp. 661–673.
[21] "OpenClover," https://openclover.org/.
[22] "gcov - A Test Coverage Program," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., (2015).

(Received: November 15, 2023)
(Accepted: January 19, 2024)

Takuma Ikeda is a graduate student of Shinshu University. His areas of interest include fault localization.

Hitoshi Kiryu is a graduate student of Shinshu University. His areas of interest include formal verification.

Satoshi Suda received his M.S. degree in mathematics from Osaka University, Osaka, Japan, in 2016. He joined Mitsubishi Electric Corp.Currently he is a researcher of Solution Engineering Dept. at Advanced Technology R&D Center.

Shinpei Ogata is an Associate Professor at Shinshu University, Japan. He received his BE, ME, and PhD from Shibaura Institute of Technology in 2007, 2009, and 2012 respectively. From 2012 to 2020, he was an Assistant Professor, and since 2020, he has been an Associate Professor, in Shinshu University. He is a member of IEEE, ACM, IEICE, IPSJ, and JSSST. His current research interests include model-driven engineering for information system development.

Kozo Okano received his BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. He was an Assistant Professor and an Associate Professor of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2020, he has been a Professor at the Department of Electrical and Computer Engineering, Shinshu University. Since 2023, he has been the Director of Center for Data Science and Artificial Intelligence. His current research interests include formal methods for software and information system design, and applying deep learning to Software Engineering. He is a member of IEEE, IEICE, and IPSJ.