2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00067

Fault Localization with Code Coverage Representation Learning

Yi Li
Department of Informatics
New Jersey Institute of Technology
New Jersey, USA
Email: yl622@njit.edu

Shaohua Wang*
Department of Informatics
New Jersey Institute of Technology
New Jersey, USA
Email: davidsw@njit.edu

Tien N. Nguyen
Computer Science Department
The University of Texas at Dallas
Texas, USA
Email: tien.n.nguyen@utdallas.edu

***Abstract*—In this paper, we propose DEEPRL4FL, a deep learning fault localization (FL) approach that locates the buggy code at the statement and method levels by treating FL as an image pattern recognition problem. DEEPRL4FL does so via novel code coverage representation learning (RL) and data dependencies RL for program statements. Those two types of RL on the dynamic information in a code coverage matrix are also combined with the code representation learning on the static information of the usual suspicious source code. This combination is inspired by crime scene investigation in which investigators analyze the crime scene (failed test cases and statements) and related persons (statements with dependencies), and at the same time, examine the usual suspects who have committed a similar crime in the past (similar buggy code in the training data).**

For the code coverage information, DEEPRL4FL first orders the test cases and marks error-exhibiting code statements, expecting that a model can recognize the patterns discriminating between faulty and non-faulty statements/methods. For dependencies among statements, the suspiciousness of a statement is seen taking into account the data dependencies to other statements in execution and data flows, in addition to the statement by itself. Finally, the vector representations for code coverage matrix, data dependencies among statements, and source code are combined and used as the input of a classifier built from a Convolution Neural Network to detect buggy statements/methods. Our empirical evaluation shows that DEEPRL4FL improves the top-1 results over the state-of-the-art statement-level FL baselines from 173.1% to 491.7%. It also improves the top-1 results over the existing method-level FL baselines from 15.0% to 206.3%.

***Index Terms*—fault localization, code coverage, representation learning, machine learning, deep learning**

I. INTRODUCTION

Finding and fixing software defects is an important process to ensure a high-quality software product. To reduce developers' effort, several *fault localization* (FL) approaches [49] have been proposed to help localize the source of a defect (also called a *bug* or *fault*). In the FL problem, given the execution of test cases, an FL tool identifies the set of *suspicious lines of code* with their associated suspiciousness scores [49]. The key input of an FL tool is the *code coverage matrix* in which the rows and columns correspond to the source code statements and test cases, respectively. Each cell is assigned with the value of 1 if the respective statement is executed in the respective test case, and with the value of 0, otherwise. In recent FL, several researchers also advocate for fault localization at method level [27]. FL at both levels are useful for developers.

Spectrum-based fault localization (SBFL) approaches [6], [20], [22] take the recorded lines of code that were covered by each of the given test cases, and assigned each line of code a suspiciousness score based on the code coverage matrix. Despite using different formulas to compute that score, the idea is that a line covered more in the failing test cases than in the passing ones is more suspicious than a line executed more in the passing ones. A key drawback of those approaches is that the same score is given to the lines that have been executed in both failing and passing test cases. An example is the statements that are part of a block statement and executed at the same nested level. Another example is the conditions of the condition statements, e.g., if, while, do, and switch.

To improve SBFL, *mutation-based fault localization* (MBFL) approaches [33], [37], [38] enhance the code coverage information by modifying a statement with mutation operators, and then collecting code coverages when executing the mutated programs with the test cases. They apply suspiciousness score formulas in the same manner as the spectrum-based FL approaches on the code coverage matrix for each original statement and its mutated ones. Despite the improvement, MBFL are not effective for the bugs that require the fixes that are more complex than a mutation (Section II).

Machine learning (ML) and *deep learning (DL)* have been used in fault localization. DeepFL [27] computes for each faulty method a vector with +200 scores in which each score is computed via a specific feature, e.g., a spectrum-based or mutation-based formula, or a code complexity metric. Despite its success, the accuracy of DeepFL is still limited. A reason could be that it uses various calculated scores from different formulas as a proxy to learn the suspiciousness of a faulty element, instead of fully exploiting the code coverage. Some formulas, such as the spectrum- and mutation-based formulas, inherently suffer from the issues as explained earlier with the statements covered by both failing and passing test cases.

We propose DEEPRL4FL, a fault localization approach for buggy statements/methods that exploits the image classification and pattern recognition capability of the Convolution Neural Network (CNN) [24] to apply on the code coverage (CC) matrix. Instead of summarizing each row in that matrix

*Corresponding Author

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00067

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

with a suspiciousness score, we use its full details. Importantly, we enhance the matrix to facilitate the application of the CNN model in *recognizing the key characteristics in the matrix* to discriminate more easily between faulty and non-faulty statements/methods. Toward that end, we order the columns (test cases) of the CC matrix so that the *test cases with the non-zero values on nearby statements are close together*. Specifically, the first test case covers the most statements. The next test case shares with the previous one as many executed statements as possible. We expect that *the CNN model with its capability to learn the relationships among nearby cells via a small filter* can recognize the visual characteristic features to discriminate faulty and non-faulty statements/methods.

Inspired by the method in crime scene investigation, we use three sources of information for FL: 1) code coverage matrix with failed test cases (the crime scene and victims), 2) similar buggy code in the history (*usual suspects* who have committed a similar crime in the past), and 3) the statements with data dependencies (related persons). First, the evidences at the crime scene are always examined. For an analogy, the CC matrix for the occurrence of the fault is analyzed. Second, an investigator also makes a connection from the crime scene to *the usual suspects*. This is analogous to the modeling of the code of the faults that have been encountered in the training dataset. The idea is that if the persons (analogous to the code) who have committed the crimes with similar modus operandi (M.O.) in the past are likely the suspects (code with high suspiciousness) in the current investigation.

Third, in addition to the crime scene, the investigator also looks at the relationships between the victim or the things happening at the scene and other related persons. Thus, in addition to the statement itself, its suspiciousness is viewed taking into account the data dependencies to other statements in execution flows and data flows. The idea is that some statements, even far away from the buggy line, could have impacts or exhibit the consequences of the buggy line when they are data-dependent. Thus, for a test, we first identify the error-exhibiting (EE) line (defined as the line where the program crashed or exhibited an incorrect value(s)/behavior(s)). That is, if the program crashes, the error-exhibiting line is listed. If there is no crash and an assertion fails, assertion statement is EE line. EE line is usually specified in a test execution. To identify the related statements, from the EE line, we consider the execution order. However, if the statements are in the same block of code (i.e., being executed sequentially), we also consider the data dependencies among them and with the EE line. Finally, all three sources of information are encoded into vector/matrix representations, which are used as input to the CNN model to act as a classifier to decide whether a statement/method as a faulty or not.

We conducted several experiments to evaluate DEEPRL4FL on Defects4J benchmark [1]. Our empirical results show that DEEPRL4FL locates 245 faults and 71 faults at the method level and the statement level, respectively, using only top-1 candidate (i.e., the first ranked element is faulty). It can improve the top-1 results of the state-of-the-art *statement-level* FL baselines by 317.7%, 273.7%, 173.1%, 195.8%,

```
1 public static String join(Object[] array, char separator,
2   int startIndex, int endIndex) {
3   if (array == null) {
4     return null;
5   }
6   int noOfItems = (endIndex - startIndex);
7   if (noOfItems <= 0) {
8     return EMPTY;
9   }
10 -StringBuilder buf = new StringBuilder((array[startIndex]
11   == null? 16 : array[startIndex].toString().length())+1);
12 + StringBuilder buf = new StringBuilder(noOfItems * 16);
13   for (int i = startIndex; i < endIndex; i++) {
14     if (i > startIndex) {
15       buf.append(separator);
16     }
17     if (array[i] != null) {
18       buf.append(array[i]);
19     }
20   }
21   return buf.toString();
22 }
```

Fig. 1: An Example of a Buggy Statement

and 491.7% when comparing with Ochiai [6], Dstar [48], Muse [33], Metallaxis [38], and RBF-Neural-Network-based FL (RBF) [47], respectively. DEEPRL4FL also improves the top-1 results of the existing *method-level* FL baselines, MULTRIC [52], FLUCCS [43], TraPT [28], and DeepFL [27], by 206.3%, 53.1%, 57.1%, and 15.0%, respectively. Our results show that three sources of information in DEEPRL4FL positively contribute to its high accuracy.

We also evaluated DEEPRL4FL on ManyBugs [25], a benchmark of C code with 9 projects. The results are consistent with the ones on Java code. DEEPRL4FL localizes 27 faulty statements and 98 faulty methods using only top-1 results.

The contributions of this paper are listed as follows:

1. Novel code coverage representation. Our representation enables fully exploiting test coverage matrix and taking advantage of the CNN model in image recognition to localize faults.

2. DEEPRL4FL: Novel DL-based fault localization approach. Test case ordering and three sources of information allow treating FL as a pattern recognition. Without ordering and statement dependencies, the CNN model will not work well.

3. Extensive empirical evaluation. We evaluated our model against the most recent FL models at the statement and method levels, in both within-project and cross-project settings, and for both C and Java. Our replication package is available at [5].

II. MOTIVATING EXAMPLES

Fig. 1 shows a real-world example of a bug in Defects4J [1]. The bug occurs at line 10 in which the length of the string to be built via *StringBuilder* was not set correctly. A developer fixed the bug by modifying lines 10–11 into line 12.

To localize the buggy line, there exist three categories of approaches. The first one is spectrum-based fault localization (SBFL). The key idea in SBFL is that in a test dataset, *a line executed more in the failing test cases than in the passing ones is considered as more suspicious than a line executed more in the passing ones*. A summary of the CC matrix for this bug is shown in Fig. 2a. The lines 3, 6–7, and 10–11 in Fig. 1 are

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

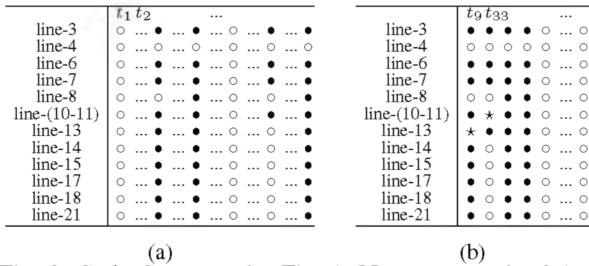

Fig. 2: Code Coverage for Fig. 1 (Note: ○, ●, ⋆ for 0,1,-1)

executed in both passing and failing test cases, and as a result, given *the same suspiciousness scores*. Thus, SBFL is ineffective to detect the buggy line 10 and this buggy method.

The second category is mutation-based fault localization (MBFL). A MBFL approach (e.g., Metallaxis [38]) modifies a statement using mutation operators. After collecting code coverage information for each statement regarding to multiple mutations, it computes the suspiciousness score for each statement using a spectrum-based formula (e.g., Ochiai [6]) on the CC matrix for each original statement and for its mutated ones. However, the fix for the buggy line 10 requires more complex code transformations than a mutation. Thus, an MBFL approach cannot detect this buggy line and buggy method.

The third category is deep learning and machine learning-based FL approaches [27], [47]. Specifically, Wong *el al.* [47] use a backpropagation neural network on code coverage for each statement. Since the lines 3, 6–7, and 10–11 are executed in both passing and failing test cases, the model cannot learn to distinguish them to detect the buggy line 10. DeepFL [27], uses multilayer perceptron (MLP) on a matrix in which each row corresponds to a statement, while each column is a suspiciousness score computed by a spectrum-based formula, or a code complexity metric. In our experiment (Section IX-C1), DeepFL could not detect the buggy line 10. Despite combining several scores, the aforementioned lines are given the same suspiciousness scores by each spectrum-based formula.

Observation 1. The state-of-the-art spectrum-based [22], [31], mutation-based [33], [37], [38], and deep learning-based FL approaches [27] do not consider the full details of the CC matrix. Instead, they summarize each statement/row with a suspiciousness score, thus limiting their capabilities.

To address that, we aim to exploit the full details of the CC matrix via the use of the CNN model [24], which has been shown to be effective in image pattern recognition. However, there is a challenge: if we do not enforce an order on the test cases (columns), we might end up with a CC matrix with the dark cells (the values of 1) that are far apart (Fig. 2a). Note that *the CNN model is effective to learn the relationships among the nearby cells in a matrix with its small sliding window (called filter)* [24]. Thus, *we need to enforce an order on the test cases, i.e., the columns of the CC matrix so that the values of 1 on the same or nearby rows get to be close to one another*. For example, if we enforce an order with the mentioned strategy (we will explain the detailed algorithm later) for the running example, we will have the matrix in Fig. 2b. That is, the results

```
1  public int Compute(int x, int y, int z){
2     int i = x + 1;
3     int j = x + y;
4     int m = 5;
5  -  if (i < y + 4)
6  +  if (i < y + 7)
7        if (j > 5 & z > j){
8           m = m + z;
9        } else {
10          m = m + j;
11       }
12    } else {
13       m = m + i;
14    }
15    i = m + 1;
16    return m;
17 }
```

Fig. 3: A Buggy Statement and Interdependent Statements

for the test cases 9, 33, etc. in the test dataset of Defects4J for this example are shown in the leftmost columns. We expect that *the CNN model with its sliding window is more effective in the resulting matrix after the ordering due to the nearby dark cells on the left side*. The empirical study on the impact of such ordering will be explained in Section IX.

Let us consider another example in Fig. 3. The bug occurs at line 5 and is fixed in line 6. The program fails in two test cases: 1) x=5, y=0, z=1, and 2) x=7, y=1, z=9. In this example, the lines 2, 3, 4, 5, 15, and 16 are all executed in both passing and failing test cases. Thus, the spectrum-based, mutation-based approaches, and DeepFL give them the same suspiciousness scores, and do not detect the buggy line 5 and this buggy method. The line 16 returns the unexpected results for the two failing test cases. In fact, the spectrum-based and mutation-based approaches locate line 16 as the buggy line. However, the actual error occurs at line 5, steering the execution to the incorrect branch of the if statement. This implies that *while the source of the bug is at line 5, the error exhibits at line 16, which is far apart from line 5, yet has a dependency with it*. However, the line 15, immediate preceding of line 16, does not contribute to the incorrect result at line 16.

Observation 2. We observe that the line that exhibits erroneous behavior (e.g., line 16) might not be the buggy line (line 5). However, the buggy line 5 has a dependency with the line 16. Thus, *identifying the key line exhibiting the erroneous behavior is crucial for FL*. We also observe that the lines with program dependencies with one another are in fact more valuable in helping localize the buggy line than the lines without such dependencies. Thus, *while considering the execution order of statements, an FL approach should consider the statements with program dependencies as well.*

III. Exploratory Study

Inspiring by the above observations, to further study the impact of ordering of the columns (i.e, the test cases) of the code coverage matrix, we conduct an exploratory experiment with the Convolution Neural Network (CNN) model. Specifically, we choose a simple CNN model having 2D convolutional layer and 15 convolutional cores with the size of $3 * 3$. In

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

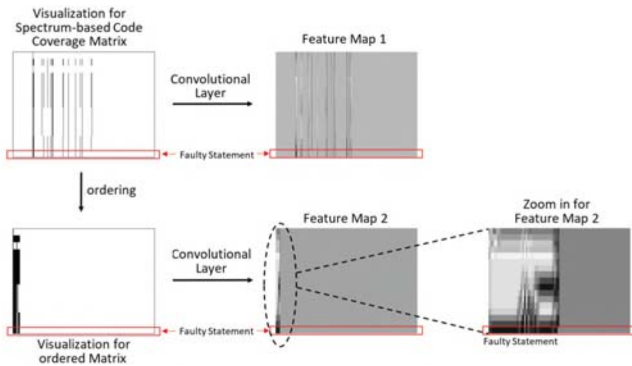

Fig. 4: A Feature Map after Ordering of Test Cases

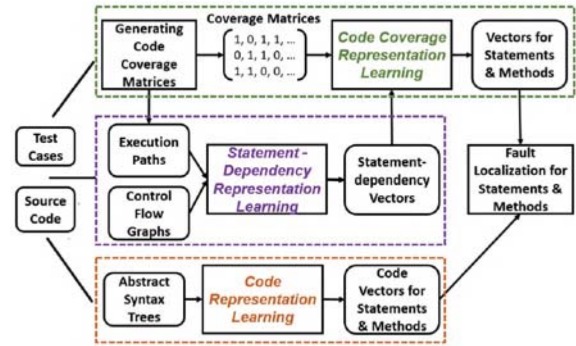

Fig. 5: DEEPRL4FL's Architecture

this experiment, we use all the bugs in the Defects4J dataset (will be explained in Table I). As for training and testing, we use the leave-one-out strategy on the entire Defects4J dataset. That is, when we perform testing on a fault, we use all other faults in the dataset as the training data to train the model. As for the ordering in the code coverage matrix, the first test case is the one covering as many statements as possible, and the subsequent test case is the one that runs through as many same statements as the previously selected test cases (will be detailed in Section V-C). To encode the pass/fail information, we detected the error-exhibiting lines (EE) (see Section V-B) and marked them with -1 values.

We conduct two executions with two different inputs for the CNN model. In the first one, for training, we use the original spectrum-based CC matrix as the input. The output is a matrix with the same size as the input CC matrix, however, the row corresponding to the buggy statements/lines are marked with all the 1 values and all other rows are marked with the zeros. For the second execution, we use the CC matrix after ordering. The output is the same as in the first execution. For testing, we use the trained CNN model to run on the buggy methods under test. We examine the output of that execution. The CNN model generates 15 feature maps as the output from the 15 different convolutional cores. The feature maps of a CNN capture the result of applying the CNN filters to an input matrix. That is, at each layer, the feature map is the output of that layer. By visualizing a feature map for a specific input image, i.e., an CC matrix, we aim to gain some understanding of what features the CNN model can detect.

We randomly select 10 faults as the testing data. In two of them, the result of the CNN model indicates the correct buggy statement for the fault. Fig. 4 shows the result for one of the faults. We visualized the code coverage matrices and feature maps as gray-scale images. In the code coverage matrices on the left, rows represent statements from the top to the bottom, and columns represent test cases. The buggy statement/line is marked with a red rectangle. As seen, after ordering, the left side of the CC matrix becomes darker. The white part, which represents the zero values, corresponds to the test cases that do not go through the statements in this buggy method.

For the feature maps corresponding to before and after ordering, the rows also correspond to the statements and the columns represent the test cases. We examine all 15 feature maps when running the CNN model on an input. Among the 15 feature maps for the case of ordering, we found one feature map (feature map 2: the bottom right image) contains the darker spot at the buggy statement/line compared to the lighter spots for the non-buggy statements/lines. We examine all 15 feature maps for the case of the original CC matrix and visualize the corresponding feature map (feature map 1: the upper right image). As seen in the red rectangle, there is no dark line/spot around the buggy statement. In brief, with the ordering of the columns in the CC matrix, we make the CNN model recognize visual characteristics corresponding to the buggy statement and distinguish it from the non-buggy ones. This motivates us to integrate the ordering of the columns in the CC matrix for code coverage representation learning.

IV. APPROACH OVERVIEW

Inspired by the crime scene investigation method, we explore three aforementioned sources of information. Correspondingly, DEEPRL4FL has three representation learning processes: code coverage representation learning (crime scene), statements dependency representation learning (relations), and source code representation learning (usual suspects) (Fig. 5).

1) **Code Coverage Representation Learning**: This learning is dedicated to the "crime scene" analysis of the bug. This process has two parts. First, to help the CNN model recognize the patterns, we take *the given (un-ordered) set of test cases* and *perform an ordering algorithm* to arrange the columns of the CC matrix. The strategy of ordering is to enable the values of 1 to be closer to form darker spots in the left side of the matrix, expecting that the CNN model can work effectively to recognize nearby cells to distinguish the buggy and non-buggy statements (see exploratory study and empirical evaluation).

Second, we also perform the analysis on the output of test cases to locate the *error-exhibiting (EE)* lines (Observation 2). If the execution of a test crashes, the line information is always available. Even if there is no crash, the test fails, the program often explicitly lists the lines of code that exhibit the incorrect results/behaviors. We use such information to locate the EE line in the buggy source code corresponding to each test

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

case. Finally, the results of individual test cases are encoded as follows. The cells in the matrix corresponding to the EE lines of test cases will be marked with -1 values (see the stars in Fig. 2b). Thus, if a column has a value of -1 at a row, the corresponding test case is a *failing* one. The values of 1 and 0 represent the coverage or non-coverage of the test case to a statement. Thus, a column has no value of -1 (all the values are 0 or 1), the corresponding test case is *passing*. The resulting matrix is called the *enhanced CC matrix* (ECC).

2) **Dependency Representation Learning***:* The suspiciousness of a statement is seen taking into account the data dependencies to other statements in the execution flows and data flows, in addition to the statement itself (Observation 2). Specifically, we consider both the execution orders and data dependencies among the statements. For example, if the statements are executed sequentially in the same nested level as part of a block statement, data dependencies will help the model in FL as shown in Section II. Additionally encoding the statements with such dependencies has the same effect as putting together the rows corresponding to the dependent statements in the CC matrix. In our example, in addition to the entire matrix in Fig. 3, we also encode the data dependencies among statements (i.e., in the same spirit with the case of putting closer the rows 2, 3, 4, 5, 13, 15, and 16), and feed them into the CNN model. In our tool, we collect execution paths and data flow graph for each test case.

3) **Source Code Representation Learning***:* For each buggy code in the training data, we choose to represent the code structure by the long paths that are adapted from a prior work [10], [29]. A long path is a path that starts from a leaf node, ends at another leaf node, and passes through the root node of the AST. The AST structure can be captured and represented via the paths with certain lengths across the AST nodes [10]. After this, we have the vectors for the buggy code. Finally, all the representation vectors are used as the inputs of the CNN model, which is part of the FL module in Fig. 5.

V. Code Coverage Representation Learning

A. Generating Code Coverage Matrices

As in prior FL studies [6], [7], [30], we obtain a code coverage matrix for each method of a given project and error messages of the failing test cases using GZoltar [2], a tool for code coverage analysis. We further modify GZoltar to record the actual execution path of statements within a method during the execution of a test case. For example, for the method in Fig. 1, the execution path of running the first selected test case is $line_3 \Rightarrow line_6 \Rightarrow line_7 \Rightarrow line_(10-11) \Rightarrow line_13 \Rightarrow \underbrace{line_14 \Rightarrow line_15 \Rightarrow line_17 \Rightarrow line_18 \Rightarrow ...}_{Statements\ repeated\ in\ the\ for\ loop} \Rightarrow line_21$.

We also use mutation to generate more coverage information. First, we apply the same mutators as in DeepFL [27] to mutate each statement within a method using the mutation tool PIT-1.1.5 [4]. To generate a mutation-based matrix, we apply one mutator to mutate a statement and use GZoltar to record the execution. Thus, given n mutators that can be applicable to a statement, we generate n new versions of the given method.

```
java.lang.IllegalArgumentException: Color parameter outside of expected range: Red Green Blue
    at java.awt.Color.testColorValueRange(Color.java:310)
    at java.awt.Color.<init>(Color.java:395)
    at java.awt.Color.<init>(Color.java:369)
    at org.jfree.chart.renderer.GrayPaintScale.getPaint(GrayPaintScale.java:128)
    at org.jfree.chart.renderer.junit.GrayPaintScaleTests.testGetPaint(GrayPaintScaleTests.java:107)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at junit.framework.TestCase.runTest(TestCase.java:176)
    at junit.framework.TestCase.runBare(TestCase.java:141)
    at junit.framework.TestResult$1.protect(TestResult.java:122)
```

Fig. 6: Error Message Example

If it has m statements, we generate $n * m$ matrices for the method. We refer the mutation-generated $n * m$ matrices as **mutation-based matrices** and for clarification, we refer the non-mutator-generated matrix as the **spectrum-based matrix**.

B. Identifying Error-Exhibiting Lines

A cell in the CC matrix can have three values: {1,0,-1}. While the values of 1 and 0 indicate passing, the values of (-1) indicate failing. We obtain -1 for an error-exhibiting statement or crashed statement from the error messages of failing test cases. An error message shows the names of classes, methods, and line numbers exhibiting an error. We directly use the line numbers, method and class names to assign -1s to the statements in the matrix. Fig. 6 shows an example of the error message containing a stack trace produced by running a test case on the project *Chart* with the bug *Chart-24*. Because the current method under investigation is getPaint, our algorithm searches for that method in the stack trace to derive the EE statement at the line 128 of the file GrayPaintScale.java (which contains the method getPaint). Each failing test case has only one EE statement for the current method under study.

C. Test Case Ordering Algorithm

Algorithm 1 takes the set of test cases S and enforces an order on S. The strategy is to move the values of 1 and -1 closer to one another in the left side. First, if there exist failing test cases, i.e., test cases with -1s, we select the test case with the value of -1 at the statement appearing latest in the code. We then find the test case that shares the same statement having -1 with the last selected test case (line 9). That is, we group together the test cases that go through the same statement and also fail. If we do not have such test case, then we repeat the process of looking for another failing test case (i.e., with -1). In Fig. 2b, the test case 9 is selected as the first one with only one -1 (marked with a star) at the line 13 (latest statement). We search for the next test case that has a -1 at the latest. The test case 33 is chosen at the second column.

If we do not have any failing test case left, we select the test case that has the most 1s (line 13). Next, we select the next test case that shares the most number of the same statements having the values of 1s with the last selected test case. This helps move the values of 1 closer. We repeat this step to select a new test case compared with the previously selected one until all the test cases were ordered. We stop this step if no test case has the same statements with 1s as the last selected test case (column). If two test cases are tie, we select the one with the

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Test Case Ordering Algorithm

```
 1: function ORDERINGTESTCASES(S : testSet)
 2:     List = []
 3:     while (S <> ∅) do
 4:         if HAVETESTCASESWITHMINUSONE(S) then
 5:             selT = FINDTESTCASEWITHMINUSONEWITHHIGHESTINDEX(S)
 6:             S.remove(selT)
 7:             List.append(selT)
 8:             while HAVETESTCASESAMESTMTWITHMINUSONE(selT, S) do
 9:                 selT = FINDTESTCASESAMESTMTWITHMINUSONE(selT, S)
10:                 S.remove(selT)
11:                 List.append(selT)
12:         else
13:             selT = FINDTESTCASEWITHMOSTONE(S)
14:             S.remove(selT)
15:             List.append(selT)
16:             while HAVETESTCASEWITHSAMESTMTSWITHONE(selT, S) do
17:                 selT = FINDTESTWITHMOSTSAMESTMTSWITHONE(selT, S)
18:                 S.remove(selT)
19:                 List.append(selT)
20:     return List
```

last value of 1 at a statement appearing latter. The rationale is that such a test case covers more statements than the other. If they are still tie, the selection of either of them will result in similar visual effects locally at that row. In brief, in any cases of ties, the visual effects around the statements are similar.

In addition to the spectrum-based matrices, we also apply the same enhancements, *identifying error-exhibiting lines* and *ordering text cases* to mutation-based code coverage matrices.

VI. STATEMENT-DEPENDENCY REPRESENTATION

We aim to model the *execution orders* and *data dependencies* among the statements of the method under study.

1) **Execution Order Representation***:* We obtain the execution path (e-path) as each test case was executed. We only consider the relations among statements within a method. Since an e-path is a sequence of statements, we apply *word2vec* [32] on all execution paths of test cases to learn the vectors that encode the relations among statements. Thus, *each statement has a word2vec-generated vector*.

2) **Data Dependency Representation***:* Using execution paths is not sufficient due to the following. First, the statements in a loop may repeat multiple times in an e-path, thus, they may dominate vector learning using *word2vec* and weaken the relations between the statements inside and outside of a loop, which is also crucial in FL. Second, interdependent statements might not be nearby in an e-path, yet are useful in detecting the buggy line (Observation 2). To address those, we also use the data-flow graph (DFG) for the statements in a method.

We use WALA [45] to generate DFGs in which a node represents a statement and an edge represents a data flow between two nodes. If A connects to B, we assign the weight of 1. If there is no edge from B to A, we create that edge but assign the weight of -1. This makes *node2vec* [18], a network embedding technique, applicable to our graph. The value of -1 helps distinguish between the artificial edges and the real ones. After this step, some statements (nodes) with data dependencies have *node2vec*-generated vectors.

3) **Vectors for Statements with Dependencies***:* The *word2vec* vector for a statement s in the execution order and the *node2vec* vector for s in program dependencies among the statements are combined via Hadamard product to represent s. Finally, the output vector is **a statement-dependency vector for a statement**, modeling the statement with the dependencies and/or execution orders among statements.

4) **Combining Statement Dependencies and ECC Matrices***:* To further enrich the ECC matrix (a spectrum-/mutation-based matrix), we incorporate the dependencies among the statements in a method under study into that matrix. In the enhanced matrix, we have the i-th statement (S_i) of a method under test with the test cases, $T = \{T_1, \ldots, T_j \ldots, T_n\}$, where j indicates the j-th test case, $1 \leq j \leq n$, and n is the number of test cases. The statement S_i under a test case T_j has a cell value v_{ij} that can be either {1, 0, or -1}. Thus, the statement S_i can be represented as a vector $\vec{S_i} = \{v_{i1}, \ldots, v_{ij}, \ldots, v_{in}\}$. Each statement ($S_i$) has a statement-dependency vector ($\vec{S}_i^{sd}$). We multiply each v_{ij} with $\vec{S}_i^{sd}$, to obtain $v_{ij} * \vec{S}_i^{sd}$, for each cell of S_i and T_j in the enhanced matrix. Thus, the statement S_i can be represented as a new 2-dimensional vector $\vec{S}_i^{2d}$ = $< v_{i1} * \vec{S}_i^{sd}, \ldots, v_{ij} * \vec{S}_i^{sd}, \ldots, v_{in} * \vec{S}_i^{sd} >$. Any vector $\vec{S}_i^{sd}$ multiplied by a $v_{ij} = 0$ results in a vector with all 0s.

A method often has multiple statements $\{S_1, \ldots, S_i, \ldots S_m\}$, where i indicates the i-th statement, $1 \leq i \leq m$, and m is the number of statements. Thus, *a method is presented as a 3-D matrix, i.e., a list of 2-D statement vectors*.

The same steps are used to enhance and combine statement dependencies into a mutation-based matrix. A statement S_i in a mutation-based matrix is represented as a set of mutated statements and each mutated statement is represented as a 2-D vector. Thus, in this case, the statement S_i is represented as a 3-D matrix. After enhancing the ECC matrix and combining statement-dependencies as explained, we obtain the following:

- In a spectrum-based matrix (SBM), a statement is represented as a 2-D vector and a method as a 3-D matrix;
- In mutation-based matrices (MBM), a statement is represented as a 3-D matrix and a method as a 4-D matrix.

5) **Encoding Code Coverage Matrices with a CNN Model***:* After obtaining those representations for statements and methods, we apply the Convolution Neural Network (CNN) [23] to learn features. We use a typical CNN with the following layers: a convolutional layer, a pooling layer and a fully connected layer. We feed the followings into the CNN model separately to detect a buggy statement/method:
i) For spectrum-based matrices (SBM), we fed a 2-D vector representing for a statement and a 3-D matrix for a method,
ii) For mutation-based matrices (MBM), we fed a 3-D matrix representing for a statement and a 4-D matrix for a method.

We apply a fully connected layer before CNN on the method in a mutation-based matrix (i.e., represented as a 4-D matrix) to reduce an 4-D matrix into an 3-D matrix.

The outputs of the CNN include the vectors for a statement or a method in spectrum-based or mutation-based matrices:
i) V_{ss}, 1-D vector for a statement in SBM,
ii) V_{sm}, 1-D vector for a method in SBM,
iii) V_{ms}, 1-D vector for a statement in MBM, and
iv) V_{mm}, 1-D vector for a method in MBM.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

VII. Source Code Representation Learning

Let us explain how we capture the usual suspicious source code via code representation learning.

For a statement, we tokenize it and treat each token in the statement as a word and the entire statement as a sentence. We use *word2vec* [32] on all the statements of a project to compute a token vector for each token. After having the vectors for all the tokens, for a statement, we have a matrix [Token-Vector$_1$, Token-Vector$_2$, $\ldots$, Token-Vector$_m$]. To obtain a unified vector to represent a statement instead of a matrix, we apply a fully connected layer to reduce the matrix into 1-D vector. Thus, we have one vector for each statement.

At the method level, we used two existing code representation learning techniques *code2vec* [29] and ASTNN [53] for a method. In *code2vec*, we use *long paths* over the AST. A long path is a path that starts from a leaf node, ends at another leaf node, and passes through the root node of the AST. The AST structure can be represented via the paths with certain lengths across the AST nodes. Specifically, we regard a long path as a sequence and apply *word2vec* on all the long paths of a method to generate a vector representation for each AST node. Now, each path is represented as an ordered list of node vectors (the order is based on the appearance order of the nodes in a path), and each method is represented as a bag of paths, i.e., ordered lists of node vectors. Essentially, a method is represented by a matrix. We use a fully connected layer to transform the matrix into 1-D vector for a method.

At the method level, we also used tree-based representation ASTNN [53]. ASTNN splits the AST of a method into small subtrees at the statement level and applies a Recursive Neural Network (RNN) [42] to learn vector representations for statements. The ASTNN exploits the bidirectional Gated Recurrent Unit (GRU) [44] to model the statements using the sequences of sub-tree vectors. After obtaining the long-path-based vector and the tree-based vector for a method, we apply a fully connected layer as the one in CNN [23] to combine these two vectors into one unified vector for a method.

VIII. Fault Localization with CNN Model

A. Statement-level Fault Localization

After all the previous steps, each statement has 3 vectors:
1) $\vec{V}_{ss}$, a SBM-based statement vector (Section VI-5);
2) $\vec{V}_{ms}$, a MBM-based statement vector (Section VI-5); and
3) $\vec{V}_{cs}$, a source code-based statement vector (Section VII).
The vectors are combined via Hadamard Product [19]:

$$M_s = [len(\vec{V}_{ss}), 1, 1], M_m = [1, len(\vec{V}_{ms}), 1], M_c = [1, 1, len(\vec{V}_{cs})]$$

$$M_{combined} = broadcast(M_s) \circ broadcast(M_m) \circ broadcast(M_c)$$

M is the matrix which is expanded from v by keeping one dimension as v and adding two more dimensions with the size of 1. $broadcast()$ is the operation to copy a dimension into multiple times to expand the matrix to the suitable size for Hadamard product. The rationale is that all three vectors from three different aspects should be fully integrated. The resulting matrix is of the size $[len(\vec{V}_{ss}), len(\vec{V}_{ms}), len(\vec{V}_{cs})]$. Next, we use the trained CNN model with a softmax on the matrix to classify a statement into faulty or non-faulty. The output of the softmax is standardized to be between 0 to 1. To train the model, the same combined matrix for a statement is used at the input layer and the corresponding classification (faulty or not) is used at the output layer of the CNN model.

B. Method-level Fault Localization

Similar to statement-level FL, each method has 3 vectors:
1) $\vec{V}_{sm}$, a SBM-based method vector (Section VI-5);
2) $\vec{V}_{mm}$, a MBM-based method vector (Section VI-5); and
3) $\vec{V}_{cm}$, a source code-based method vector (Section VII).

Moreover, we also consider the similarity between the source code and the error messages of the failing test cases as in DeepFL [27]. We first collect 3 types of information from failed tests, including the name of the failed tests, the source code of the failed tests and the complete failure messages (including exception type, message, and stacktrace). Second, we collect 5 types of information from source code, including the full qualified name of the method, accessed classes, method invocations, used variables, and comments. For each combination, we calculate the similarity score between each information from the failed tests and each from the source code using the popular TF-IDF method [27]. We generate 15 similarity scores as 15 features for a method. Thus, a method also has the fourth vector, $\vec{V}_m^{sim}$ with 15 features.

For fault localization, we combine the above method vectors into a matrix by using the Hadamard product as in Section VIII-A, then use the trained CNN model with a softmax to classify a method into faulty or non-faulty. We train the model in the same manner as FL at the statement level.

IX. Empirical Evaluation

A. Research Questions

We seek to answer the following research questions:

RQ1. Statement-level FL Comparison. How well does our tool perform compared with the state-of-the-art *statement-level* FL models?

RQ2. Method-level FL Comparison. How well does our tool perform compared with the existing *method-level* FL models?

RQ3. Impact Analysis of Different Matrix Enhancing Techniques. How do those techniques including test case ordering, and statements dependency affect the accuracy?

RQ4. Impact Analysis of Different Representations Learning. How do different types of information affect the accuracy?

RQ5. Cross-project Analysis. How does DeepRL4FL perform in the cross-project setting?

RQ6. Performance on C Code. How does DeepRL4FL perform in C projects for FL?

B. Experimental Methodology

*1) **Data Set:*** We use the benchmark, Defects4J V1.2.0 [1] with ground truth (Table I). For a bug in project P, Defects4J has a separate copy of P but with only the corresponding test suite revealing the bug. For example, P_1, a version of P, passes a test suite T_1. Later, a bug B_1 in P_1 is identified. After

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Defects4J Dataset

Identifier	Project name	# of bugs
Chart	JFreeChart	26
Closure	Closure compiler	133
Lang	Apache commons-lang	65
Math	Apache commons-math	106
Mockito	Mockito	38
Time	Joda-Time	27

debugging, P_1 has an evolved test suite T_2 detecting the bug. In this case, Defects4J has a separate copy of the buggy P_1 with a single bug, together with the test suite T_2. Similarly, for bug B_2, Defects4J has a copy of P_2 together with T_3 (evolving from T_2), and so on. For within-project setting, we test one bug B_i with test suite $T_{(i+1)}$ by training on all other bugs in P. To reduce the influence of the overfitting problem, we applied L2 regularization and added dropout layers.

*2) **Experiment Metrics**:* Following prior studies [27], [28], we use the following metrics to evaluate an FL model:

Recall at Top-K: is the number of faults with at least one faulty statement that is correctly predicted in the ranked list of K statements. We report Top-1, Top-3, and Top-5.

Mean Average Rank (MAR): We compute the average rank of all faulty elements for each fault. MAR of each project is the mean of the average rank of all of its faults.

Mean First Rank (MFR): For a fault with multiple faulty elements (methods/statements), locating the first one is critical since the others may be located after that. MFR of each project is the mean of the first faulty element's rank for each fault.

*3) **Experiment Setup and Procedure**:*

RQ1: Statement-level Fault Localization Comparison.

Baselines. We compare DEEPRL4FL with the following statement-level FL approaches:

- Two spectrum-based fault localization (SBFL) techniques: **Ochiai** [6] and **Dstar** [48];
- Two recent mutation-based fault localization (MBFL) techniques: **MUSE** [33] and **Metallaxis** [38];
- Two deep-learning based FL approaches: **RBF Neural Network (RBF)** [47] and **DeepFL** [27]. DeepFL [27] works at the *method level* with several features. For comparison, in this RQ1 for the statement level, we can only use DeepFL's spectrum- and mutation-based features applicable to detect faulty statements.

As in FL work [12], [27], [28] using Defects4J, we used the setting of leave-one-out cross validation on the faults for each individual project (i.e., within-project setting). Specifically, we use one bug (i.e., with one buggy statement or method) as testing, and the remaining bugs in a project for training.

Tuning DEEPRL4FL *and the baselines.* We tuned our model with the following key hyper-parameters to obtain the best performance: (1) Epoch size (i.e., 100, 200, 300); (2) Batch size (i.e., 64, 128, 256); (3) Learning rate (i.e., 0.001, 0.003, 0.005, 0.010); (4) Vector length of word representation and its output (i.e., 150, 200, 250, 300); (5) Convolutional core size (i.e., 3×3, 5×5, 7×7, 9×9, and 11×11); (6) The number of convolutional core (3, 5, 7, 9, and 11).

As for *word2vec*, for a method, we consider all tokens in the source code order as a sentence. We tune the following hyper-parameters for DeepFL (using only the features relevant to statements): Epoch number (5, 10, 15, ..., 60), Loss Functions (softmax, pairwise), and learning rate (0.001, 0.005, 0.010).

RQ2: Method-level Fault Localization Comparison.

Baselines: We also compare our approach with the following state-of-the-art approaches that localize faulty methods.

MULTRIC [52] is a learning-based approach to combine different spectrum-based ranking techniques using learning-to-rank for effective fault localization.

FLUCCS [43] is a learn-to-rank based technique using spectrum-based scores and change metrics (e.g., code churn and complexity metrics) to rank program elements.

TraPT [28] is a learn-to-rank technique to combine spectrum-based and mutation-based fault localization.

DeepFL [27] is a DL-based model to learn the existing/latent features from multiple aspects of test cases and a program. We used all the features of DeepFL in this method-level study.

Tuning DEEPRL4FL *and the baselines.* Similar to RQ1, we perform our experiments using leave-one-out cross validation on the faults for each project. We use the same settings in RQ1 to train our model. Note that in DeepFL paper [27], **DeepFL**, **MULTRIC**, **FLUCCS**, and **TraPT** have been evaluated using leave-one-out cross validation and other settings on the same data set of Defects4J V1.2.0. DEEPRL4FL is also evaluated on Defects4J V1.2.0 using the same settings and procedure as DeepFL. Thus, we used the result on the numbers of detected bugs reported in DeepFL [27] for those models.

RQ3: Impact Analysis of Different Matrix Enhancing Techniques. We evaluate the impact of the following techniques on accuracy: (1) test case ordering algorithm utilizing the EE lines (**Order**); (2) statements' dependencies (**StatDep**). We first build a base model by using only the spectrum- and mutation- based matrices in DEEPRL4FL (without using the above techniques), then apply the above techniques on the matrices to build two variants of DEEPRL4FL: {*Base+Order*}, and {*Base+Order+StateDep* (DEEPRL4FL)}. We train each variant using the same settings as in RQ1. Due to space limit, we show only the analysis results obtained in the within-project setting for method-level FL.

RQ4: Impact Analysis of Learning Representations. We have the following representation learning schemes: the enhanced spectrum-based CC matrix (**NewSpecMatrix**) and the enhanced mutation-based CC matrix (**NewMutMatrix**). We also have source code representation (**CodeRep**) and textual similarity between source code and error messages in failing tests (**TextSim**). To test the impact of those representation learning schemes on accuracy, we built a base model using only *NewSpecMatrix*, and three other variants: {*NewSpecMatrix+NewMutMatrix*}, {*NewSpecMatrix+ NewMutMatrix+CodeRep*}, and {*NewSpecMatrix+ NewMutMatrix+CodeRep+TextSim*}. We trained each variant using the same settings as in RQ1. Due to space limit, we show only the results for the within-project setting for method-level FL.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE II: RQ1. Results of comparative study for statement-level fault localization. P% = |Top-1|/{395 Bugs}

Approach	Top-1	Top-3	Top-5	P%	MFR	MAR
Ochiai	17	88	115	4.3%	54.29	71.32
Dstar	19	92	115	4.8%	48.67	69.51
MUSE	26	47	63	6.6%	36.34	48.73
Metallaxis	24	81	108	6.1%	34.59	49.21
RBF	12	37	52	3.0%	22.54	57.47
DeepFL	39	114	129	9.9%	24.09	31.28
DEEPRL4FL	**71**	**128**	**142**	**18.0%**	**20.32**	**28.63**

RQ5: Cross-project Analysis. We also setup the cross-project scenario: testing one bug in a project, but training a model on all of the bugs of other projects. For a project, we test every bug and sum up the total number of bugs in the project that can be localized in the cross-project scenario.

RQ6: DEEPRL4FL's Fault Localization Performance on C Code. We also evaluated DEEPRL4FL on C projects from the benchmark dataset, ManyBugs [3], [25], with 185 bugs from 9 projects. We used the same model in RQ1 for statement-level FL and the model in RQ2 for method-level FL.

C. Experimental Results

*1) RQ1-Results (***Statement-level Fault Localization Comparison***):* As seen in Table II, DEEPRL4FL improves over the state-of-the-art statement-level FL baselines. Specifically, DEEPRL4FL improves Recall at Top-1 by 317.6%, 273.7%, 173.1%, 195.8%, 491.7%, and 82.1% in comparison with Ochiai, Dstar, Muse, Metallaxis, RBF, and DeepFL.

We examined the results and report the following. The key reason for the spectrum-based FL approaches fail to localize the buggy statements is that they give the same suspiciousness score to the statements at the same nested level. For the mutation-based FL approaches, the key reason for not being able to localize the buggy statements/methods is that the fix requires a more sophisticated change than a mutation. Let us take an example. In Fig. 7, the fault is caused by an incorrect variable. To fix it, the variable was changed from *pos* to *pt* at line 14. The state-of-art spectrum-based approaches cannot localize this fault because lines 6, 7, 13, and 14 have the same score (They were executed in both passing and failing test cases). For the mutation-based FL approaches, there is none of mutation operators that changes the variable *pos* into *pt* in a method call at the buggy line 14. Thus, they cannot observe the impact of mutations on the code coverage. As a consequence, they cannot locate the buggy line 14.

To gain insights, we performed a visualization of a feature map for this case. During training, CNN learns the values for small windows, called *filters*. The feature maps of a CNN capture the result of applying the filters to an input matrix. That is, at each layer, the feature map is the output of that layer. In image processing, visualizing a feature map for an input helps gain understanding on whether the model detects some part of our desired object and what features the CNN observes. Fig. 8 shows a feature map for the example in Fig. 7. We can see that around the lines 6–8 and 13–14, the feature

```
1 public void translate(CharSequence input, Writer out)...{
2   ...
3   int pos = 0;
4   int len = input.length();
5   while (pos < len) {
6     int consumed = translate(input, pos, out);
7     if (consumed == 0) {
8       char[] c=Character.toChars(Char...codePointAt(...));
9       out.write(c);
10      pos+= c.length;
11      continue;
12    }
13    for (int pt = 0; pt < consumed; pt++) {
14 -    pos += Char.charCount(Char.codePointAt(input,pos));
15 +    pos += Char.charCount(Char.codePointAt(input, pt));
16    }
17  }
18 }
```

Fig. 7: An Example from Defects4J

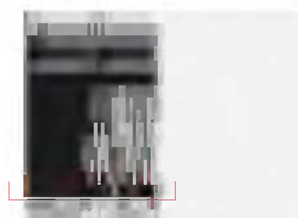

Fig. 8: A Feature Map Produced by CNN for Fig. 7

map is visually dark. Without ordering (i.e., a random order of test cases), the feature map does not exhibit such visualization.

To further study the impacts of the ordering and data dependencies, we modified DEEPRL4FL in the following settings: 1) *No ordering + No dependencies*: the buggy line 14 is ranked at 43th; 2) *No ordering + dependencies*: it is ranked at 29th; 3) *Ordering + No dependencies*: it is ranked at 7th; and 4) *Ordering + dependencies*: it is ranked at the top.

*2) RQ2-Results (***Method-level Fault Localization Comparison***):* As seen in Table III, DEEPRL4FL improves Recall at Top-1 by 206.3%, 53.1%, 57.1%, and 15.0% over MULTRIC, FLUCCS, TraPT, and DeepFL, respectively. DEEPRL4FL's MAR is slightly higher than DeepFL's (3.6% higher). On average, DEEPRL4FL ranks the correct elements higher than DeepFL, as its MFR is lower (10.4% lower).

The spectrum-based and mutation-based FL approaches fall short of DeepFL and DEEPRL4FL. A key reason is that they consider only dynamic information in test cases, while DeepFL and our model use both static and dynamic information. In comparison with DeepFL, we further analyzed the bugs that our tool can locate, but DeepFL missed. We found that the mean first rank of a buggy method in the ranking lists of potential buggy methods returned by DeepFL is 7.08. Without the ordering and statement dependency in our model, the mean first rank is 6.84. With only ordering in our model, the mean first rank is 2.82. With only dependency in our model, the mean first rank is 4.45. With both ordering and dependency, our model can locate the bugs that DeepFL missed.

Let us use an example in Defects4J (Fig. 9) that our model detected but DeepFL missed. The (buggy) method

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE III: RQ2. Results of Comparative Study for Method-level Fault Localization. P% = |Top-1|/{395 Bugs}

Approach	Top-1	Top-3	Top-5	P%	MFR	MAR
MULTRIC	80	163	195	20.3%	37.71	43.68
FLUCCS	160	249	275	40.5%	16.53	21.53
TraPT	156	249	281	39.5%	9.94	12.70
DeepFL	213	282	305	53.9%	6.63	**8.27**
DEEPRL4FL	**245**	**294**	**311**	**62.0%**	**5.94**	8.57

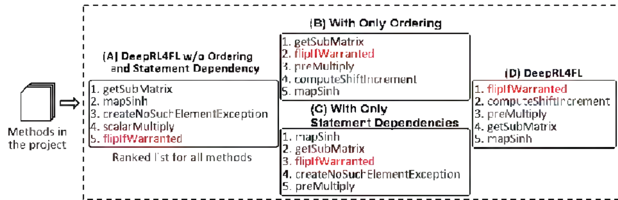

Fig. 9: Ordering and Statement Dependencies Affect Ranking

flipIfWarranted together with the other methods in the project were fed into four variants of our model. As seen, with the setting in which both ordering and statement dependencies are removed, flipIfWarranted is ranked 5th in the list of all methods. For the setting with only ordering, it is ranked at 2nd place. For the setting with only statement dependencies, it is ranked 3rd. With both, our model ranks the buggy method flipIfWarranted at the 1st position. This analysis shows that ordering test cases and statement dependencies are the key drivers that help our model locate more bugs than DeepFL.

*3) RQ3-Results (***Impact Analysis of Different Matrix Enhancing Techniques***):* Table IV shows that our matrix enhancing techniques positively contribute to DEEPRL4FL. Specifically, comparing {*Base*} with {*Base+Order*}, ordering the test cases can improve every metric. *Order* helps localize 53 more bugs (13.4%) using Top-1. It helps improve MFR and MAR by 20.1% and 12.7%, respectively, showing that ordering can help DEEPRL4FL push the faulty methods higher in the ranked list.

Comparing {*Base+Order*} with {*Base+Order+StateDep*}, we see that modeling dependencies into matrices is useful to improve the performance of DEEPRL4FL. StateDep can improve 8.4%, 9.6%, and 4.5% in Top-1, MFR, and MAR.

To further study *the impact of the ordering*, we visualize the feature maps for the 53 bugs that *Order* can detect and *Base* did not. Those are the cases where ordering helps. Visualizing the feature maps for those inputs allows us to understand what features the CNN detects in both cases of ordering and no-ordering. Moreover, that also allows us to see if ordering can help the CNN model learns better the discriminative features in locating the buggy statements. To do so, for each of those bugs, we used the CNN model as part of *Base* and *Order* to produce two feature maps: one corresponds to *Base* (no ordering) and one to *Order*. We then visualized and compared those feature maps as gray-scale images. The CNN model generates 9 feature maps as the output from 9 convolutional cores.

TABLE IV: RQ3. Ordering (Order) and Adding Dependencies (StateDep) in Method-level FL. P% = |Top-1|/{395 Bugs}

Variants	Top1	P%	MFR	MAR
Base (DEEPRL4FL *w/o* Order,StateDep)	173	43.8%	8.23	10.27
Base + Order	226	57.2%	6.57	8.97
Base + Order + StateDep (**DEEPRL4FL**)	245	62.0%	5.94	8.57

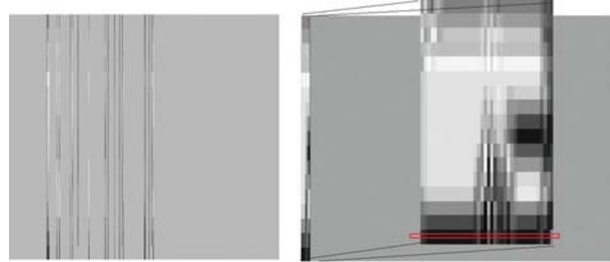

Fig. 10: Visually Darker Lines around Buggy Statement

In all the bugs, we observe the same phenomenon. Let us take an example. Fig. 10 shows two feature maps for one of those bugs. The left image is for *Base* (without ordering), and the right one is for *Order* (with ordering). We zoom out the leftmost columns in the right feature map. The row corresponding to the buggy line is in the red rectangle. With ordering, one of those 9 feature maps has visually darker lines around the buggy statement. In contrast, without ordering, all the feature maps are similar to the one on the left, i.e., do not show any clear visual lines. In brief, with ordering, the CNN model, which focuses on the relations of neighboring cells, can detect the features along the buggy statement.

*4) RQ4-Results (***Impact Analysis of Learning Representations***):* Table V shows that our representation learning has positive contributions. Comparing {*Base*} with {*Base+ NewMutMatrix*}, we can see that mutation-based matrices can help locate 23 more bugs using Top-1 and improve MFR and MAR by 8.2% and 3.5%. By adding code representation learning, we improve DEEPRL4FL to localize 9 more bugs and gain an increase on MFR and MAR by 7.9% and 4.0%, respectively. Furthermore, TextSim also positively contributes to our model. For statement-level FL, code representation is also useful, improving Top-1 from 65 to 71 bugs, i.e., 9.2% (not shown).

*5) RQ5-Results (***Cross-project Analysis***):* As seen in Table VI, DEEPRL4FL achieves better results in the within-project setting than in the cross-project one. This is expected as the training and testing data is from the same project in the within-project setting, thus a model may see similar faults.

In the cross-project setting, DEEPRL4FL correctly detects 217 bugs at Top-1 in comparison with the best result (207 bugs) from the baselines. In the within-project setting, DEEPRL4FL correctly detects 230 bugs at Top-1 in comparison with 80/160/156/213 bugs (not shown) from the baseline models MULTRIC/FLUCCS/TraPT/DeepFL, respectively.

Time Complexity. On average, training time is 350-380 minutes per project in the cross-project setting, and 120-130 minutes per project in the within-project setting. Once the model is trained, the prediction time per fault is 2-7 seconds in both the cross- and within-project settings.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE V: RQ4. Results of Learning Representations in Method-level FL. P% = |Top-1|/{395 Bugs}

Variants	Top-1	P%	MFR	MAR
Base (NewSpecMatrix)	189	47.8%	8.09	9.91
Base+NewMutMatrix	212	53.7%	7.43	9.56
Base+NewMutMatrix+CodeRep	221	55.9%	6.84	9.18
Base+NewMutMatrix+CodeRep+TextSim (**DEEPRL4FL**)	245	62.0%	5.94	8.57

TABLE VI: RQ5. Cross-project versus Within-project Results

Projects	Cross-project				Within-project			
	Top-1	P%	MFR	MAR	Top-1	P%	MFR	MAR
Chart	13	50.0%	3.15	5.62	15	57.7%	2.85	4.65
Time	13	48.1%	9.78	14.70	14	51.9%	8.41	13.33
Math	61	57.5%	3.81	4.88	64	60.4%	2.93	4.83
Closure	71	53.4%	11.70	15.23	73	54.9%	9.38	12.37
Mockito	12	31.6%	11.42	16.42	14	36.8%	9.39	15.11
Lang	47	72.3%	2.13	2.49	50	76.9%	1.97	2.31

*6) RQ6-Results (***Performance on C Code***):* As seen in Table VII, DEEPRL4FL can localize 27 faulty statements and 98 faulty methods with only Top-1 statements and methods. The empirical results show that the performance of DEEPRL4FL on the C projects is consistent with the one on the Java projects. Specifically, at the statement level, the percentages of the total C and Java bugs that can be localized are similar, i.e., 14.6% vs. 18.0%, respectively. At the method level, the percentages of the total C and Java bugs that can be localized are also consistent, i.e., 53.0% vs. 62.0%, respectively.

7) Threats to Validity: **i) Baseline implementation.** For comparative study, we implemented Ochiai, Dstar, MUSE, Metallaxis, and RBF-neural-network for statement-level FL. We followed the paper [27] to implement MUSE and Metallaxis using PIT-1.1.5. RBF-neural-network approach is built for artificial faults and our real bug dataset cannot match the requirements. **ii) Result generalization.** Our comparisons with the baselines were only carried out on the Defects4J dataset. Further evaluation on other datasets should be done.

8) Limitations: The quality of test cases is important for our approach. If there are only a couple of passing test cases or the crash occurs far apart from the faulty method, DEEPRL4FL does not learn a useful representation matrix to localize the faults. It does not work well on locating the faults that require statement additions to fix (all of the baselines in this paper do not either). Moreover, it does not work well for short methods, as they provide less statement dependencies. It is also hard for our model to localize the uncommon faults. Because it is DL-based, if there is a very uncommon fault that may not be seen in the training dataset, it will not work correctly.

X. RELATED WORK

Fault Localization (FL). The Spectrum-based Fault Localization (SBFL), e.g., [6], [7], [21], [22], [30], [31], [36], [51], [54], has been intensively studied in the literature. Tarantula [20], SBI [30], Ochiai [6] and Jaccard [7], they share the same basic insight, i.e., code elements mainly executed by failed tests are more suspicious. The Mutation-based Fault Localization (MBFL), e.g., [17], [33], [35], [55], [56], aims to additionally consider mutated code in fault localization. The examples of MBFL are Metallaxis [37], [38] and MUSE [33]. Learning-to-Rank (LtR) has been used to improve fault localization [12], [28], [43], [52]. MULTRIC [52] combines different suspiciousness values from SBFL. Some work combines SBFL suspiciousness values with other information, e.g., program invariant [12] and source code complexity information [43], for more effective LtR in FL. TraPT [28] combines suspiciousness values from both SBFL and MBFL. Neural networks have been applied to fault localization [16], [50], [58], [60]. However, they mainly work on the test coverage summarization scores, which has clear limitations (e.g., it cannot distinguish elements covered by both failing and passing test cases) [28], and are usually studied on artificial faults or small programs. DeepFL [27] was shown to improve the method-level FL approach TraPT [28]. DEEPRL4FL is also related to CNN-FL [57], which feeds the original coverage matrix with passing/failing information into a CNN model. CNN-FL is theoretically equivalent to *Base* model in Table IV, without any matrix enhancements, test cases ordering, statement dependencies, and code representations.

TABLE VII: RQ6. ManyBugs (C Projects) versus Defects4J (Java Projects). P% = |Top-1|/{Total Bugs in Datasets}

Level	ManyBugs (C projects)				Defects4J (Java projects)			
	Top-1	P%	MFR	MAR	Top-1	P%	MFR	MAR
Statement	27	14.6%	25.74	31.33	71	18.0%	20.32	28.63
Method	98	53.0%	6.91	9.89	245	62.0%	5.94	8.57

Code Representation Learning (CRL). The recent success in machine learning has lead to much interest in applying machine learning, especially deep learning, to program analysis and software engineering tasks, such as automated correction for syntax errors [14], spreadsheet errors detection [13], [40], fuzz testing [39], program synthesis [11], code clones [46], [41], [26], program summarization [8], [34], code similarity [10], [59], probabilistic model for code [15], and path-based code representation, e.g., Code2Vec [10] and Code2Seq [9]. All the approaches learn code representations using different program properties. However, none of the existing fault localization techniques has performed direct code modeling and learning on code coverage information of the test cases for the FL purpose as in DEEPRL4FL.

XI. CONCLUSION

We propose a deep learning based fault localization (FL) approach, DEEPRL4FL, to improve existing FL approaches. The key ideas include (1) treating the FL problem as the image recognition; (2) enhancing code coverage matrix by modeling the relations among statements and failing test cases; (3) combining code coverage representation learning with statement dependencies, and the code representation learning for usual suspicious code. Our empirical evaluation shows that our model advances the state-of-the-art baseline approaches.

ACKNOWLEDGMENT

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] (2019) The Defects4J data set. [Online]. Available: https://github.com/rjust/defects4j

[2] (2019) Gzoltar. [Online]. Available: http://www.gzoltar.com/

[3] (2019) The ManyBugs data set. [Online]. Available: https://repairbenchmarks.cs.umass.edu/

[4] (2019) Pit. [Online]. Available: https://pitest.org/

[5] (2021) The github repository for this study. [Online]. Available: https://github.com/deeprl4fl2021icse/deeprl4fl-2021-icse

[6] 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)*. IEEE, 2006, pp. 39–46.

[7] ——, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[8] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[9] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[10] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018. [Online]. Available: http://arxiv.org/abs/1803.09473

[11] M. Amodio, S. Chaudhuri, and T. W. Reps, "Neural attribute machines for program generation," *CoRR*, vol. abs/1705.09231, 2017. [Online]. Available: http://arxiv.org/abs/1705.09231

[12] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 2016, pp. 177–188.

[13] D. W. Barowy, E. D. Berger, and B. Zorn, "Excelint: Automatically finding spreadsheet formula errors," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276518

[14] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *CoRR*, vol. abs/1603.06129, 2016. [Online]. Available: http://arxiv.org/abs/1603.06129

[15] P. Bielik, V. Raychev, and M. Vechev, "Phog: Probabilistic model for code," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2933–2942. [Online]. Available: http://proceedings.mlr.press/v48/bielik16.html

[16] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 2007, pp. 137–146.

[17] T. A. Budd, "Mutation analysis of program test data." 1981.

[18] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," *CoRR*, vol. abs/1607.00653, 2016. [Online]. Available: http://arxiv.org/abs/1607.00653

[19] Hadamard, "Hadamard product," https://en.wikipedia.org/wiki/Hadamard_product_(matrices), last Accessed July 11, 2019.

[20] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, 2002, pp. 467–477.

[21] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE'05)*. ACM, 2005, pp. 273–282.

[22] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*. IEEE, 2017, pp. 114–125.

[23] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[25] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, December 2015.

[26] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A deep learning-based clone detection approach," in *IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, Sep. 2017, pp. 249–260.

[27] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 169–180.

[28] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133916

[29] Y. Li, S. Wang, T. N. Nguyen, and S. V. Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang. 3, OOPSLA, Article 1 (October 2019)*, 2019.

[30] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 15–26. [Online]. Available: https://doi.org/10.1145/1065010.1065014

[31] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[32] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *27th Annual Conference on Neural Information Processing Systems 2013 (NIPS'13)*, 2013, pp. 3111–3119.

[33] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *IEEE International Conference on Software Testing, Verification and Validation*, 2014, pp. 153–162.

[34] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "TBCNN: A tree-based convolutional neural network for programming language processing," *CoRR*, vol. abs/1409.5718, 2014. [Online]. Available: http://arxiv.org/abs/1409.5718

[35] V. Musco, M. Monperrus, and P. Preux, "A large-scale study of call graph-based impact prediction using mutation testing," *Software Quality Journal*, vol. 25, no. 3, pp. 921–950, 2017.

[36] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, p. 11, 2011.

[37] M. Papadakis and Y. Le Traon, "Using mutants to locate "unknown" faults," in *IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 691–700.

[38] ——, "Metallaxis-FL: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[39] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," TUD-CS-2016-14664, TU Darmstadt, Tech. Rep., 2016.

[40] R. Singh, B. Livshits, and B. Zorn, "Melford: Using neural networks to find spreadsheet errors," Microsoft Research, Microsoft Tech Report Number MSR-TR-2017-5, Tech. Rep., 2017.

[41] R. Smith and S. Horwitz, "Detecting and measuring similarity in code clones," 2009.

[42] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng, "Parsing natural scenes and natural language with recursive neural networks," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 129–136.

[43] 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 (ISSTA'17)*. ACM, 2017, pp. 273–283.

[44] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Sep. 2015, pp. 1422–1432. [Online]. Available: https://www.aclweb.org/anthology/D15-1167

[45] WALA, "Wala documentation." http://wala.sourceforge.net/wiki/index.php/Main_Page, last Accessed July 11, 2019.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.

[46] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326

[47] 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, 2011.

[48] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using DStar (D*)," in *6th IEEE International Conference on Software Security and Reliability*. IEEE, 2012, pp. 21–30.

[49] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2521368

[50] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 04, pp. 573–597, 2009.

[51] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1. IEEE, 2007, pp. 449–456.

[52] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 2014, pp. 191–200.

[53] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on Abstract Syntax Tree," in *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE Press, 2019, pp. 783–794.

[54] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*. IEEE, 2011, pp. 23–32.

[55] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *IEEE International Conference on Software Maintenance (ICSM'10)*. IEEE, 2010, pp. 1–10.

[56] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 765–784. [Online]. Available: https://doi.org/10.1145/2509136.2509551

[57] Z. Zhang, Y. Lei, X. Mao, and P. Li, "CNN-FL: An effective approach for localizing faults using convolutional neural networks," in *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, 2019, pp. 445–455.

[58] Z. Zhang, Y. Lei, Q. Tan, X. Mao, P. Zeng, and X. Chang, "Deep learning-based fault localization with contextual information," *Ieice Transactions on Information and Systems*, vol. 100, no. 12, pp. 3027–3031, 2017.

[59] G. Zhao and J. Huang, "Deepsim: Deep learning code functional similarity," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 141–151. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236068

[60] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Mathematical Problems in Engineering*, vol. 2016, 2016.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on January 09,2022 at 01:05:57 UTC from IEEE Xplore. Restrictions apply.