IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 1, 2020, accepted November 6, 2020, date of publication November 10, 2020,
date of current version November 30, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3037235

Mulr4FL: Effective Fault Localization of Evolution Software Based on Multivariate Logistic Regression Model

XIAOLIN JU[1,2], (Member, IEEE), JIE QIAN[1], ZHIHUA CHEN[1], CHUNYU ZHAO[1], AND JUNYAN QIAN[3]

[1]School of Information Science and Technology, Nantong University, Nantong 226019, China
[2]School of Computer Science and Technology, Nanjing University, Nanjing 210046, China
[3]School of Computer Science and Information Engineering, Guangxi Normal University, Guilin 541004, China

Corresponding author: Xiaolin Ju (ju.xl@ntu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61502497, Grant 61602267, and Grant 61673384; in part by the Guangxi Key Laboratory of Trusted Software Research Plan under Grant KX201530 and Grant KX201532; and in part by the National Natural Science Foundation of Guanxi under Grant 2015GXNSFDA139038 and Grant 2018GXNSFDA138003.

ABSTRACT Fault localization is indeed tedious and costly work during software maintenance. Studies indicate that combining both structural features and behavior characteristics of programs can be beneficial for improving the efficiency of fault locating. In this paper, we proposed a framework, called Mulr4FL, for fault localization using a multivariate logistic regression model that combined both static and dynamic features collected from the program under debugging. Firstly, the hybrid metrics data set, with both program structural features and behavior characteristics combined, is constructed by static program analyzing and dynamically tracing that runs with a designed metrics set. Meanwhile, the fault information of the legacy program is also obtained from the bug tracking system. Secondly, Bivariate logistic analysis is performed to filter the metrics that are significantly related to faults, and then with the selected metrics and their measurements, a multivariate logistic regression model was constructed and trained. Finally, based on the trained logistic model, we conduct the multivariate logistic analysis on the features of the evolved software and predict the buggy class methods. An empirical study was conducted based on a set of benchmarks that are used widely in program debugging research. The results indicate that the Mulr4FL can significantly improve the effectiveness of locating faults in contrast to 5 baseline techniques.

INDEX TERMS Software testing, debugging, fault localization, logistic regression analysis.

I. INTRODUCTION

Software is a complex artifact during which the life cycle often undergoes multiple version evolution due to the change of requirement or software operating environment. Influenced by the constraints of human intellectual activities, flaws are often inevitably introduced into new versions of the software during its evolution. These potential defects often cause failures or troubles to end-users. Reducing software failures and improving its quality is essential in software maintenance.

Usually, programers detect and locate software bugs by static analying or dynamic testing [1]. Studies have shown that some bugs are caused by complicated program structures, such as recursive calls and complex loops [2]. According to whether to use software runtime information and how to use this information, the existing fault localization approaches can be classified into three categories. The first category is the statistical fault localization based on the program spectrum (SFL, Spectrum based fault localization) [3], [4]. These methods collect and count the coverage information of the program entities (e.g., statements, predicates, execution paths, or functions) in the process of program testing, then analyze the relationship between the entities and the execution results, and calculate the suspiciousness with a specific heuristic evaluation formula to estimate the buggy probability of each program entity, and finally, the program entities are examined in descending order of suspiciousness until the all faults are identified. The second category of fault localization approaches are based on program slicing technology [5], [6].

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

Program slicing uses dependency analysis to identify the set of statements in the program that affect specific program entities [7]. Slicing technology can effectively separate fault-related sentences, so it becomes an effective tool to assist fault localization [8]. The third category is based on static analysis methods. These approaches first analyze and obtain the static structure information of the program, and then combine the syntax and semantic constraints of the programming language to detect program violations [9], or use symbolic execution [10], formal proof [11] and other technologies to locate program faults.

However, the existing fault localization approaches usually utilize program information singly and simply. The former two categories of these approaches apply the dynamic information of program runs for evaluating suspiciousness of program entities, and the third category applies the static information of program structure in suspiciousness computations. As we know, the static information of the program can fully reveal the structural characteristics of the program; on the other hand, the dynamic information during the program testing can expose the behavior characteristics of the program under specific test input. Therefore, making full use of static information combined with dynamic information has the potential benefits for improving the efficiency of fault localization.

Furthermore, most of the existing fault localization approaches are based on the analyzing and testing of the current version to locate faults, and they rarely utilize the information of the legacy software. Recent research indicates that combining static and dynamic features can improve the efficiency of fault localization [12]. The multivariate logistic regression model, known as a powerful tool to reveal the relationship between independent variables and dependent variables, is becoming used widely in social statistical analysis, data mining, and other fields. Basili *et al.* [13] first applied it to the defects prediction for object-oriented programs, and Briand *et al.* [14], [15] also empirically studied the accuracy of the defects prediction of the logistic model and the MARS model in cross-version based on object-oriented metrics. Their empirical study shows that the prediction model of the system before the evolution can better predict the fault-proneness of the system after the evolution. However, their model is limited to using static information measured of object-oriented programs. We hypothesize that it will be beneficial to combine both measurements of static and dynamic information of the program, and then to build an effective multivariate logistic model for fault localization Therefore, the multivariate logistic regression model can be used to combine different behavior of the software to predict the fault-proneness entities and improve the performance of fault localization.

Besides, most fault localization approaches are proposed and demonstrated their effectiveness based on the assumption of perfect fault detection. i.e., the programmers can always determine whether the current review statement is wrong immediately, and then effectively correct the fault even without the context of this statement during code review [16]. However, this assumption, called perfect fault detection, is not always practically hold during the real-world program debugging process. This restricts the application of statement-level fault localization methods in real-world program debugging. In practice, considering the class method level as the fault locating granularity, i.e., method-level fault localization, is more effective according to the statements context in the class methods.

In this paper, we proposed an efficient fault localization framework (called Mulr4FL) using a multivariate logistic regression model in the maintenance of evolution software. The rationale behind Mulr4FLis that our model can learn the history knowledge of the software evolution by training the model with the combination of both historical and current versions of the software under debugging. Specifically, we first design a metrics set which covers software structural features and behavioral features, and we construct a feature data set using static analysis and software testing track; Second, combining with faults of the old version of the software, we conduct a Bivariate logistic analysis to filter metrics which are corresponded with faults significantly, and we construct and train a multivariate logistic regression model applying the filtered metrics and measured data of the new version; Finally, with the feature data set of the new version of the program constructed, we can apply the trained multivariate logistic model for fault predicting and locating. We also conducted an empirical study on the proposed fault localization model on a set of benchmarks.

To the best of our knowledge, the main contributions of this study can be summarized as follows:

- We first proposed a method based on bivariate logistic regression analysis for fault-related metrics selection. With the measured structural features and the detected fault details of the old version of the program, a bivariate logistic regression analysis is applied to select out the metrics that are significantly related to faults. Those selected fault-related metrics will be applied to the subsequent multivariate logistic regression analysis.
- We constructed a multivariate logistic regression model for evaluating the suspiciousness of class methods of the object-oriented program under debugging. Our model conducts regression analysis based on the selected fault-related metrics, which describe either static features of class methods or runtime features of an object-oriented program.
- We proposed a three-stage framework (called Mulr4FL) for fault localization in the class methods level, which applies the multivariate logistic regression model.
- We also evaluated the performance of our approach on a set of benchmarks. The experimental results show that Mulr4FL is more effective than the compared five state-of-the-art baseline techniques.

The rest of the article is organized as follows. Section II briefly presents the preliminary information of our study. Section III outlines the framework of our proposed method

Mulr4FL and details of key components in our method. Section IV lists the experimental settings and discussion of the experiment results. Section V states the potential threats to the validity of our empirical study. Section VI summarized the related work of fault localization. Finally, Section VII concludes this article and shows potential future directions for our study.

II. PRELIMINARIES

Mostly, the fault localization approaches are achieved by calculating the probability of fault-prone of program entities(e.g., statements, branch, predicts, basic blocks, class methods, or files). In this paper, we took the class method as the basic component for fault localization. For each class method in a program, it is assumed that whether the fault can be detected is an independent event(denoted as Y=1), and the probability of this class method containing faults can be denoted as *prob*(Y=1). For a given class method, if *prob*(Y=1)> ω, the class method contains a fault, otherwise there is no fault (ω is the specified threshold). Therefore, we can locate faults by calculating the probability of the event ''Y=1''. Previous studies indicate that the logistic regression model is widely used to describe the relationship between a binary response variable and predictors [17]. In view of the fact that different metrics can be conducted on measuring the software under test and obtain a set of variables, we believe that the use of the multivariate logistic regression model has great advantages for fault localization. In this section, we first introduce the definition of Bivariate analysis which is used for the selection of fault-related metrics, then we give the definition of logistic regression, and we finally give the definition of multivariate logistic regression which is used for multivariate analysis and prediction.

Definition 1: Bivariate analysis investigates two variables (denoted as X, Y) to determine the empirical relationship between them [18].

Bivariate analysis can test simple hypotheses of association. It determines to what extent to know and predict a value for one dependent variable if the value of the independent variable is known. Given the independent variable x_i and the dependent variable y, the bivariate analysis model $y = f_R(x_i)$ is used to analyze the statistical correlation between the variables x_i and y.

In fault localization scenarios, the independent variable x_i is the measurement of a certain metric of the class method (such as the number of lines of code), and the dependent variable y is the fault information of the class method tracked by the bug tracking system. Here, function $f_R(x_i)$ can be a linear regression model or logistic regression model, etc. To determine the variables used in the multivariate regression analysis, it is necessary to first analyze the statistical correlation between a single measurement and fault, and then select those metrics that are significantly related to one fault. Applied bivariate analysis, the metrics that are significantly related to the fault are identified.

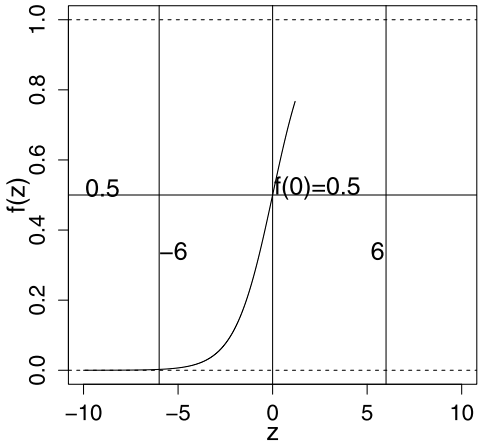


FIGURE 1. The standard logistic function $f(z)$.

Definition 2: Logistic regression model [19] is a statistical model which uses a logistic function to model a binary dependent variable. A logistic (sigmoid) function is defined as follows:

$$f(z) = \frac{e^z}{e^z + 1} = \frac{1}{1 + e^{-z}}. \tag{1}$$

where independent variable(''predictors'') z can each be a binary variable (two classes, coded by an indicator variable) or a continuous variable (any real value). The standard logistic function takes any real input $z(z \in \Re)$, and outputs values between 0 and 1. The curve of the standard logistic function is shown in Figure 1.

As shown in Figure 1, the return value of the logistic function $f(z)$ is from 0 to 1. Therefore, $f(z)$ can be used to denote the probability of the event 'Y=z'. Similarly, we denote z as an event that a class method contains a fault, then we can denote $f(z)$ as the probability of the class method to be a buggy class method. Suppose there are m variables(i.e., metrics) that affect the class method to run failure, then the variable z can be represented by a linear combination of these m variables, as shown in the Equation 2.

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m. \tag{2}$$

For a fault localization task, variables (x_i) denote the factors that affect the calculation of the suspiciousness of a class method to be buggy, and regression coefficients (β_i) denote the weights of the each variable x_i in equation 2. β_i needs to be trained with legacy data. Therefore, with a set of metrics (x_i) which are measured by the metrics on the program under debugging and weights (β_i) which are trained with the data of legacy systems, we can predict the faulty probability of each class method applying Equation 1 and Equation 2. Finally, all the class methods can be examined and identified in descending order of buggy probability.

Definition 3: multivariate analysis addresses the situations that multiple measurements and the relations among these measurements are important in each experiment based on the principles of multivariate statistics. [20]. Typically, the

multivariate analysis involves observation and analysis of more than two statistical variables at a time.

Multivariate regression analysis refers to the use of regression analysis model $y = f_R(x_1, x_2, \cdots, x_m)$ to analyze the correlation between a set of independent variables $x_1, x_2, \cdots, x_m$ and dependent variables y.

In this paper, the metrics $x_1, x_2, \cdots, x_m$ are the metrics significantly related to the fault which are filtered out by bivariate analysis, and y is the fault information of the class method tracked by the bug tracking system. Function f_R uses a multivariate logistic regression model as follows:

$$\pi(x_1, x_2, \cdots, x_m) = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_m x_m}}{1 + e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_m x_m}}. \quad (3)$$

Logistic regression analysis is a standard technique based on maximum likelihood estimation. The multivariate logistic regression model, defined in Equation 3, should be trained firstly with the traced fault information and the measurements of the class methods. After the parameters $(\beta_0, \beta_1, \cdots, \beta_m)$ of the regression model are determined, the dynamic information of the evolved program running and its class method structure measurement are applied to the model to calculate the suspiciousness of each class method. That is, the fault probability of the class method is obtained for fault locating with the trained multivariate logistic regression model. Looking at the Equation 3, we find it is satisfied with the maximum-entropy (MAXENT), which is proved to be optimal in problems where we have prior information about multiplicities [21].

III. OUR APPROACH

In the process of software evolution, logistic regression analysis can be used to locate the fault in the evolved version of the program with the class method level granularity of fault localization. Supposing that there are n class methods contained in the software under debugging, the task of fault localization is to identify each class method that contains faults. Firstly, we design a set of characteristic metrics to measure the structural features and behavior features of the class method. For example, the number of lines of code and cyclomatic complexity is used to describe the static characteristics of a class method, and the number of times the class method is covered by a successful test and a failed test to represent its behavioral characteristics. Secondly, with the help of the measurement metrics and faults information of the legacy versions, bivariate analysis is used to filter the measurement indicators which are significantly correlated with faults, and then the multivariate logistic model is constructed and trained. Finally, based on the measured information of the new version of the software, the previously trained multivariate logistic model is used to predict the buggy class methods.

A. FRAMEWORK

Figure 2 visualizes the overall framework of our appraoch. Generally, the framework mainly consists of three major stages: Pre-processing stage, multivariate logistic regression model constructing stage, and fault locating stage.

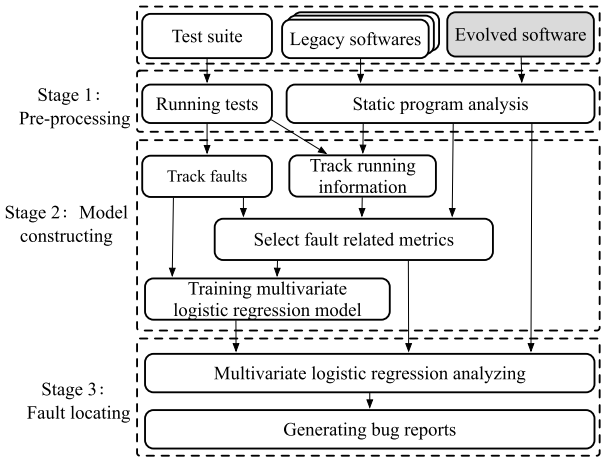


FIGURE 2. Fault localization framework based on logistic regression analysis.

TABLE 1. Metrics measuring and bugs tracking of legacy softwares.

method number	static features x_1	x_2	$\cdots$	dynamic features x_k	$\cdots$	x_m	#fault (y)
1	e_{11}	e_{12}	$\cdots$	e_{1k}	$\cdots$	e_{1m}	f_1
2	e_{21}	e_{22}	$\cdots$	e_{2k}	$\cdots$	e_{2m}	f_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	e_{n1}	e_{n2}	$\cdots$	e_{nk}	$\cdots$	e_{nm}	f_n

Step 1: Mulr4FL mainly completes two tasks: (1) Running the each version of legacy software with the test suite; (2) Collecting the structure information of the class methods. Such as the number of lines of code of the class method, cyclomatic complexity, etc.

Step 2: Mulr4FL identifies metrics that are highly correlated with faults by using bivariate analysis and to train a multivariate logistic regression model with both static and dynamic features.

Step 3: Mulr4FL predicts the faulty prone of each class method of evolved software (shaded in Figure 2) by using the multivariate regression analysis model, that is, the faulty suspiciousness of each class method.

B. MODEL CONSTRCUTING

The main task in this stage is to build a multivariate logistic regression model. The first step is to measure the program structural features and behavioral features as shown in Table 1 with the help of static analyzing and program tracing. The next step in the model construction stage is to select metrics that are significantly related to errors using bivariate analysis. The final step is to use the structural features and behavioral features to train a multivariate logistic regression model.

Table 1 shows metrics measurement with n class methods of a object-oriented program under debugging. The first column represents the numbers of all the class methods of the program; The last column #fault, denoted as $f_i (i \in [1, n])$, represents the number of detected faults contained in each

$$U = \begin{array}{c} \begin{matrix} x_1 & x_2 & \cdots & x_k & \cdots & x_m \end{matrix} \\ \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1k} & \cdots & u_{1m} \\ u_{21} & u_{21} & \cdots & u_{2k} & \cdots & u_{2m} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ u_{n1} & u_{n2} & \cdots & u_{nk} & \cdots & u_{nm} \end{pmatrix} \end{array} \quad V = \begin{array}{c} \text{fault-frequency} \\ \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \end{array}$$

FIGURE 3. Normalized matrix of legacy program metrics and faults.

method obtained by the fault tracking system; The variables, denoted as $e_{ij}(i \in [1, n], j \in [1, m])$ in the middle columns, represent the program structure measurement and dynamic tracking information. In our approach, these metrics, denoted as $x_1, x_2, \cdots, x_m$, respectively represent the static features and dynamic behavior features of each class method in the object-oriented program under debugging. Although several object-oriented program measurement metrics (such as CK set [22], and MOOD set [23]) have been proposed for the past decades, it is still unclear which subset of these metrics are significantly related to software failure. However, we can conduct a bivariate analysis to identify the most fault-related metrics.

Before conducting the bivariate analysis, we would first normalize the variables in Table 1. A preliminary investigation of the value of these metrics shows that the ranges of values are quite different. To improve the accuracy of bivariate analysis and multivariate analysis, we provide a strategy to normalize the variables listed in Table 1 to avoid the fluctuations of these metrics. For each column element e_{ij} in Table 1, the normalization factor is:

$$Z(e_{ij}) = \sum_{i=1}^{n} e_{ij} \tag{4}$$

Similarly, we also normalize the last column in Table 1, and the normalization factor is:

$$Z(f_i) = \sum_{i=1}^{n} f_i \tag{5}$$

After normalizing Table 1, the normalized matrixes (U, V) are obtained as shown in Figure 3.

In Figure 3, $u_{ij} = \dfrac{e_{ij}}{Z(e_{ij})}$, $v_i = \dfrac{f_i}{Z(f_i)}$, $i \in [1, n], j \in [1, m]$. Each column of the matrix U satisfies $\sum_{i=1}^{n} u_{ij} = 1, \forall j \in [1, m]$. Similarly, the matrix V satisfies $\sum_{i=1}^{n} v_i = 1$.

Applying the bivariate analysis with model $\hat{y} = f_R(x)$ to each feature $x_j(j \in [1, m])$, the correlation between the features and the fault can be evalued. The $\hat{y}_i$ is the normalized value that contains faults in each class method, that is, $\hat{y}_i$ belongs to the vector V in Figure 3. The feature x_j corresponds to the static information and dynamic information sub-columns in Table 1 and its value equals to vector U in Figure 3. Here $f_R(x)$ is the linear regression model. A bivariate analysis on the feature x_i and fault-frequency in Figure 3 are performed, and features that are significantly related to the fault (significant level $\alpha = 0.05$) are selected for the training of the multivariate logistic regression

TABLE 2. Metrics measuring and error prone prediction of evolved software.

method number	static features x_1	x_2	$\cdots$	dynamic features x_k	$\cdots$	x_m	fault prob.
1	e'_{11}	e'_{12}	$\cdots$	e'_{1v}	$\cdots$	e'_{1w}	p_1
2	e'_{21}	e'_{22}	$\cdots$	e'_{2v}	$\cdots$	e'_{2w}	p_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	e'_{n1}	e'_{n2}	$\cdots$	e'_{nv}	$\cdots$	e'_{nw}	p_n

$$U' = \begin{array}{c} \begin{matrix} x_1 & x_2 & \cdots & x_v & \cdots & x_w \end{matrix} \\ \begin{pmatrix} u'_{11} & u'_{12} & \cdots & u'_{1v} & \cdots & u'_{1w} \\ u'_{21} & u'_{21} & \cdots & u'_{2v} & \cdots & u'_{2w} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ u'_{n1} & u'_{n2} & \cdots & u'_{nv} & \cdots & u'_{nw} \end{pmatrix} \end{array}$$

FIGURE 4. Normalized metrics matrix of evolved program.

model. Specifically, using the vector $V(v_i)$ and multivariate $x_1, x_2, \cdots, x_m$ in Figure 3 to train multivariate logistic regression analysis model as shown in equation 3, the model parameters $\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2, \cdots, \hat{\beta}_m$ can be determined. Therefore, logistic regression analysis model is obtained as shown in the Equation 6.

$$\pi_0(x_1, x_2, \cdots, x_m) = \frac{e^{\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_m x_m}}{1 + e^{\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_m x_m}} \tag{6}$$

C. FAULT LOCATING

With the trained multivariate logistic regression model, we can conducted fault localization. In this section, we first introduce the multivariate logistic regression analysis on evolved software, then we give the algorithm fo fault localization based on the multivariate logistic regression model.

For a new version of the software, we first analyze the static structure information of its code, then run the program with the test cases to collect the dynamic information of the program. All the collected measures are recorded as sub-columns of static and dynamic features $(x_1, x_2, \cdots, x_m)$ as shown in Table 2.

Next, we normalize the original data of Table 2. The normalization factor is $Z(e'_{ij}) = \sum_{i=1}^{n} e'_{ij}$, $1 \le j \le m$, then we get the normalized matrix U' as shown in Figure 4.

The features $x_j(j \in [1, w])$ in Table 2 are selected from $x_1, x_2, \cdots, x_m$ in Table 1 which are significantly related to the faults. Specifically, different from the last column (f_i) in Table 1, which directly indicates the fault number of class methods, the last column in Table 2 indicates the probabilities of fault in the corresponding class methods.

The calculation of fault probability is to use the model established in phase two, input the traced information of the new version of the program under debugging, and calculate the normalized matrix U', and further carry out logistic regression analysis and prediction of buggy class methods, and record the predicted value for each class method. The fault probability of each class method p_i can be calculated according to the formula 6. Next, each class method is examined in a descending order of fault probability. Algorithm 1

gives a detailed description of each step of fault localization based on logistic regression analysis.

Algorithm 1 Regression Analysis Based Fault Localization

Input:
X; // metric
P, P'; // legacy and evolved system
T, T'; // old and new test suite
Output:
$Prob$; // fault probability of class methods
1: $F = Construct(X, P, T)$; // construct metrics
2: $Y = faultTrace()$; // trace fault-related information
3: $< U, V > = < Normalize(F), Normalize(Y) >$;
4: $X' = \phi$;
5: **for** each feature $x_i \in U$ **do**
6: Perform bivariate analysis using $\hat{y}_i = f_R(x_i)$, $\hat{y}_i \in V$;
7: **if** x_i correlated with $\hat{y}_i$, (p-value < 0.05) **then**
8: $X' = X' \cup \{x_i\}$;
9: **end if**
10: **end for**
11: $Log\_Model = Train(F, X')$; // train logistic model
12: $F' = Construct(X', P', T')$;
13: $U' = Normalize(F')$;
14: $Prob = Predict(U', X', Log\_Model)$; // predict buggy method
15: **return** $Prob$;

The descriptions of Algorithm 1 are as follows: Line 1–2 constructs the feature set F and fault information Y which is listed in Table 1. The function *Construct*() returns the construction feature set by static analyzing and tracking program runs. The function *Normalize*() in line 3 calculates the Normalized value of the input vectors to the range of [0, 1] by applying equation 4 and equation 5. Line 5–10 conduct the bivariate analysis to check out the metrics (X') that are significantly correlated to faults. Line 11 combines the normalized value of feature set F with the data corresponding to the fault-related metrics set X' (line 2), and calculates fault information by the normalized matrix V of Y, then trained the multivariate logistic regression model. Line 12–14 construct and normalize the measurement of selected feature metrics for the evolved program, then apply the trained model to carry out fault localization on the class method level.

IV. EXPERIMENTAL STUDY

In our empirical study, we want to evaluate the performance of our framework (**Mulr4FL**) by answering the following four research questions (RQs):

RQ1: Among the mainstream metrics for object-oriented software, which metrics are more associated with the root cause of software errors?

Motivation. In this RQ, We want to filter out those metrics that are closely associated with the causes of software failures from mainstream object-oriented software metrics by applyinng bivariate analysis technology. With program trace and analysis techniques, the selected metrics (shown in Table 1) are measured and then normalized for multivariate logistic regression analysis.

RQ2: Compared with Mulr4FL that only uses static structure metrics measurement, how does Mulr4FL perform in fault localization using hybrid static structure measurements and runtime information?

Motivation. In this RQ, we want to compare the performance of the multivariate regression model based fault localization using different metrics (i.e., simple static structure measurements, static structure measurements added with runtime information). It is necessary to evaluate the performance of our framework on mixed static and dynamic measurements via simple static measurements.

RQ3: Compared with Mulr4FL that only uses dynamic information, how does Mulr4FL perform in fault localization using hybrid static structure measurements and runtime information?

Motivation. In this RQ, we want to show the effectiveness of the multivariate regression analysis based fault localization using the hybrid static structure measurement and runtime information via simply runtime information. That is to say, we want to compare the multivariate regression model based fault localization using different metrics(i.e., runtime measurements, hybrid static measurements add with runtime information).

RQ4: Compared with the spectrum-based fault localization (SFL), can Mulr4FLperform better while using only runtime information or hybrid static structure measurement with runtime information?

Motivation. In RQ2 and RQ3, we mainly evaluate the performance of Mulr4FL based on the different measurements (i.e., static structure measurements, runtime information, and hybrid static structure measurements and runtime information). Then in RQ4, we want to evaluate the performance of Mulr4FL comparing with those state-of-the-art spectrum-based fault localization (SFL) which only uses dynamic information due to the nature of these SFL approaches. We carry out two comparisons: (1) comparing SFL with the Mulr4FL only uses dynamic information; (2) comparing SFL with the Mulr4FL uses hybrid static and dynamic measurements.

A. STUDIED PROJECTS

In this paper, we investigate eight open-source projects, namely Jtcas, NanoXml1,[1] NanoXml2,[2] Chart, Closure, Math, Time, Lang. All these projects come from two well-known datasets that are widely used in software engineering research, namely Software-artifact Infrastructure Repository (SIR[3]), Defects4J.[4] The former three projects, Jtcas, NanoXml1, NanoXml2, are from SIR [24], and the rest five projects are from Defects4J [25].

[1] the version of NanoXml is V1.6.8
[2] the version of NanoXml is V2.2.1
[3] SIR: https://sir.csc.ncsu.edu
[4] Defects4J: https://github.com/rjust/defects4j

TABLE 3. Characteristics of projects.

Projects	Excutable KLoc	#Methods	#Test cases	#Faults
Jtcas	0.18	9	1608	29
NanoXml1	3.50	118	214	7
NanoXml2	4.00	173	214	6
Chart	37.7	8943	2205	26
Closure	34.3	8257	7927	133
Math	63.0	13436	3602	106
Time	40.1	9655	4130	27
Lang	30.2	5146	2245	65
Total	212.02	45837	22145	399

Table 3 summarized the data from the nine projects used in the experiment. There are enough bugs and class methods in these eight projects. Totally, we study 45837 methods and 399 faults in 212K executable code lines together with 22145 test cases. All these 399 faults can be seeded into source code. Both the code lines and methods number of Jtcas are very small, and two versions of NanoXml have the medium scale of code lines and methods, and the last five projects have a larger scale of code and methods. We want to evaluate the performance of our approach over small, medium, or large scale software. Besides, we treated those projects which provide the training data as the legacy software, and the project which is under review as evolved software. Furthermore, we also investigate the performance of Mulr4FL on Nanoxml2 as the evolution of Nanoxml1.

B. PERFORMANCE MEASURES

Our empirical evaluation uses the metric *expense* which is firstly proposed by Renieris and Reiss [26] and is adopted widely by many researchers [4], [27], [28]. In this paper, the metric, *expense*, indicates the percentage of the number of class methods that the debugger needs to review to identify whether the class method contains fault. The calculation formula is as follows:

$$expense(f) = 100 \cdot \frac{rank(f)}{|M|} \tag{7}$$

In EQ 7, f is a class method which contains a fault, and $rank(f)$ represents the rank of f in the descending order list of all class methods with fault suspiciousness, and $|M|$ represents the number of all class methods in the system. When the suspiciousness of f is the same as some other methods, we adopt the **ML**(Middle Line) strategy [29], i.e., we calculate the average rank to deal with this case.

C. EXPERIMENTAL METHODS

To carry out our empirical study, we first identify the metrics for constructing the multivariate logistic regression model. We applied Understand software[5] (version 5.1), a very efficient tool at collecting metrics about the code, to measure the projects listed in Table 3. For simplicity, we initially selected 7 typical metrics listed in Table 4, which is a subset of Understand's 27 method-level metrics, by removing the metrics that are similar to others or not closely related to faults.

[5]https://scitools.com/

TABLE 4. Static metrics of studied projects.

Abbr. name	Full name	Description
Out	Output	Number of called subprograms plus global variables set.
In	Input	Number of calling subprograms plus global variables read.
StmtExe	CountStmtExe	Executable statements number.
CC	Cyclomatic	Cyclomatic complexity of methods.
Path	Paths	Number of possible paths, not counting abnormal exits or gotos.
Knot	Knots	Measure of overlapping jumps.
CCR	Comment to Code Ratio	Ratio of comment lines to code lines.

The detailed reasons why we set initial metrics in Table 4 are as follows: Firstly, it has been reported that subprograms and global variables often increase the risk of error-prone software. Therefore, we count the number of calling/called subprograms and set/read global variables, denoted as Out and In, respectively. Secondly, many studies have found that the size of programs (typically measures the number of executable statements) is related to the program's faults. Therefore, we take lines of method's code(denoted as StmtExe) as a metric. Thirdly, there are a large number of published studies [30] that demonstrate a positive correlation between cyclomatic complexity and faults: functions and methods that have the highest complexity tend to also contain the most faults. Therefore, we take the cyclomatic complexity of method(denoted as CC) as another metric. Furthermore, the number of executable paths of a program also indicates the complexity of the program, which is intuitively related to program faults. So we take the number of possible paths(denoted as Path) as a metric. Likewise, overlapping jumps may also increase the complexity of the program, and we take the number of overlapping as a metric(denoted as Knot). Finally, comments embedded in the lines of code can improve the readability of the code, thereby reducing the probability of faults. Therefore, we take the ratio of comment lines to code lines(denoted as CCR) as a metric, too. Using the measurement tool (Understand, version 5.1), we obtain the static metrics (listed in Table 4) measurement of all versions of projects (listed in Table 3).

To illustrate the performance of our approach, we constructed a set of benchmarks, which is consisted of five spectrum-based fault localization approaches, for comparison. These five approaches, namely Naish1, Naish2 [31], Wong [32], Russel&Rao, and Binary, is theoretically proved to be the optimal approaches for fault localization [33]. The risk formulas are as follows:

$$Naish1(f) = \begin{cases} -1, & \text{if } N_{CF}(f) < N_F \\ N_S - N_{CS}(f), & \text{if } N_{CF}(f) = N_F \end{cases} \tag{8}$$

$$Naish2(f) = N_{CF}(f) - \frac{N_{CS}(f)}{N_{CS}(f)+N_{US}(f)+1} \quad (9)$$

$$Wong(f) = N_{CF}(f) \quad (10)$$

$$R\&R(f) = \frac{N_{CF}(s)}{N_{CF}(s)+N_{UF}(s)+N_{CS}(s)+N_{US}(s)} \quad (11)$$

$$Binary(s) = \begin{cases} 0, & \text{if } N_{CF}(s) < N_F \\ 1, & \text{if } N_{CF}(s) = N_F \end{cases} \quad (12)$$

where f A represents a class method, N_{CF} and N_{CS} represents the coverage times by failed and passed test cases, respectively, N_{UF} and N_{US} represents the times uncovered by failed and passed test cases, respectively. Obviously, both of the two equations $N_S = N_{CS} + N_{US}$ and $N_F = N_{CF} + N_{UF}$ hold.

To carry out the logistic regression analysis, we defined the metrics composed of both static metrics and dynamic metrics. In detail, the static metrics include {Out, In, StmtExe, CC, Path, Knot, CCR } (denoted as S_X), and the dynamic metrics include $\{N_{CF}, N_{CS}, N_{UF}, N_{US}\}$ (denoted as D_X). Firstly, we checked out each faulty version of projects and measured the static metrics (S_X) using Understand software (version 5.1). Then we compiled and run test cases on each faulty version of projects. and at the same time, we tracked and collected the dynamic metrics (D_X) of each program running on test cases. Next, we carried out the bivariate analysis with the static metrics (S_X) and faults information of class methods. Our goal is to test the significance of the correlation between the dynamic metrics and program faults. Finally, we carried out the multivariate logistic regression analysis to answer the four research questions proposed at the beginning of Section IV.

D. RESULTS AND DISCUSSION

In this section, we present RQs in detail. We first show the analysis methods we proposed to answer these RQs. Next, we introduce and summarize the experimental results. To simplify our discussion, we give three notations to represent three scienarios, namely hyb-MLR, sta-MLR, and dyn-MLR, whose detailed description are give as follows:

- hyb-MLR means that using our framework (Mulr4FL) with hybrid static metrics and dynamic metrics.
- sta-MLR means that using our framework (Mulr4FL) with only static metrics.
- dyn-MLR means that using our framework (Mulr4FL) with only dynamic metrics.

1) RQ1: FAULTS RELATED METRICS

We first checked out each faulty version of projects listed in Table 3, and measured all the metrics ({Out, In, StmtExe, CC, Path, Knot, CCR}) of each method using Understand software, at the same time, we obtain the fault information. Then we normalized the measurements with the Equation 4 and Equation 5, and we obtain the normalized matrixes of program metrics and faults.

Next, we carried out a bivariate logistic regression analysis on each metric with faults measurement. Furthermore, we also employed the two-side t-test to calculate the p-values when we compare these different metrics to evaluate the significance of the correlations between them. The correlation coefficients and p-value of each analysis are listed in Table 5.

TABLE 5. Metrics correlation coefficients with faults and *p*-values.

No.	metrics	correlation coefficients	p-values
1	Out	0.7371213	0.030105026
2	In	0.6534016	0.148253832
3	StmtExe	0.9273033	0.001053107
4	CC	0.9103156	0.000156001
5	Path	0.8991537	0.018211032
6	Knot	0.8575046	0.038276817
7	CCR	0.7834007	0.108276817

As Table 5 shows, all the correlation coefficients of our metrics are greater than 0 which means there is a correlation between these metrics and faults. However, both the p-values at row 3 and row 8 are greater than 0.05 which means the two metrics(In and CCR) are not correlated with fault significantly though there are correlation coefficients are greater than 0.

Summary for RQ1: The five metrics, denoted as $S_{\hat{X}} = \{$Out, StmtExe, CC, Path, Knot$\}$, are significantly correlated with programs faults among our original metrics S_X. However, the remaining two metrics (In and CCR) are slightly but not significantly correlated with faults.

2) RQ2: HYB-MLR VS. STA-MLR

To evaluate this research question, we checked out each faulty version of all projects and measured the metrics for each faulty program using Understand software. That means, we measured the 5 metrics (i.e., Out, StmtExe, CC, Path, and Knot) of a total of 399 faulty versions. Furthermore, we run all the test cases on these 399 faulty versions of all projects and gathered the dynamic information of programs. That is to say, we gathered the coverage metrics ($N_{CF}, N_{CS}, N_{UF}, N_{US}$) of all the methods of a total of 22145 tests on these projects. We conducted the rest research questions (RQ2, RQ3, and RQ4) by applying all the static metrics and the dynamic information obtained above.

Next, we combined static metrics ($S_{\hat{X}}$) and dynamic metrics (D_X) and obtained a hybrid metrics H_X ($H_X = S_{\hat{X}} \cup D_X$). Then we apply our model (Mulr4FL) on H_X and $S_{\hat{X}}$, respectively. The k-fold cross-validation is applied to our research due to its high accuracy [34]. For the two projects (NanoXml1, NanoXml2), we applied the 5-fold cross-validation because their faulty versions are less than 10. For the rest projects, we applied the 10-fold cross-validation in our empirical studies. We calculated the expense on each project with formula 7. Specially, we use the Boxplot figures to reveal the distributions of hyb-MLR and sta-MLR on each project. Figure 5 shows the comparison of expense distributions on each project applying hyb-MLR and sta-MLR.

Figure 5 reveals the distributions of these two methods (i.e., hyb-MLR and sta-MLR) for the expenses on all eight projects. We observed that the expenses for fault localization

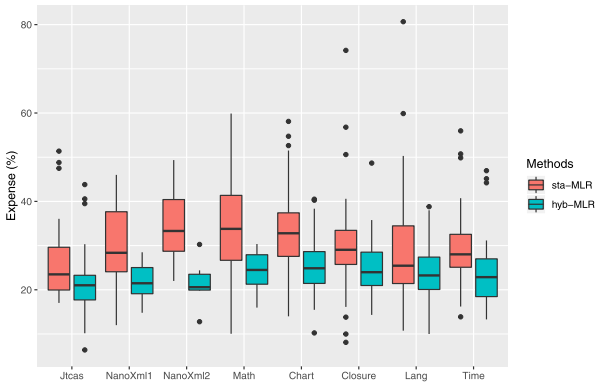


FIGURE 5. Comparison of expense applying hyb-MLR and sta-MLR.

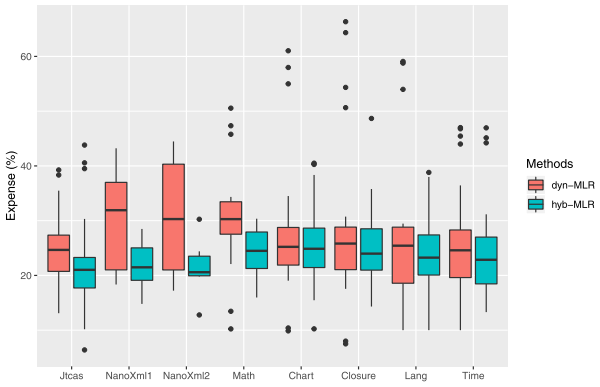


FIGURE 6. Comparison of expense applying hyb-MLR and dyn-MLR.

of hyb-MLR on all projects are lower than those of sta-MLR. Besides, the range of the expense when sta-MLR is applied is wider than that when hyb-MLR is applied to each project, especially on NanaoXml1, NanoXml2, Math, and Lang. This shows that the effectiveness of hyb-MLR is more stable than that of sta-MLR.

Besides, we investigated the performance of Mur4FL on Nanoxml2 as the evolution of Nanoxml1. In detail, we applied the prediction model, which was trained on NanoXml1, on Nanoxml2. The expenses of predicting 6 faults range from 21.23 to 49.78.

Furthermore, we calculated p-value to detect the differences between hyb-MLR and sta-MLR by Wilcoxon rank-sum test, and corrected the obtained p-values by using Benjamini_Hochberg (BH) method [35]. We set the significance level at 0.05, which means that if the corrected p-values are less than 0.05. The result indicates that there is a significant difference between the compared two methods.

Summary for RQ2: In all case, the hyb-MLR (applied our model Mulr4FL on hybrid metrics) has a significant improvement and stability compared to sta-MLR (applied our model Mulr4FL on static metrics).

3) RQ3: HYB-MLR VS. DYN-MLR

Next, we evaluate the performance of our model (Mulr4FL) on the dynamic information of the program's runtime (denoted as dyn-MLR) by comparing it with that on hybrid metrics (denoted as hyb-MLR). Similarly, we apply our model (Mulr4FL) on H_X and D_X with k-fold cross-validation to calculate the expenses, respectively. In detail, we applied 5-fold cross-validation on two projects (NanoXml1, NanoXml2) and 10-fold cross-validation on the rest of the projects. For each fault, the expense is calculated by Equation 7, and the expense distributions of the faults are group present by each project with the Boxplot (as shown in Figure 6).

Looking at Figure 6, it is apparent that hyb-MLR reported significantly lower expenses than dyn-MLR. All the mean scores of the expenses using hyb-MLR are lower than that of dyn-MLR on the eight projects. Besides, the expenses range using dyn-MLR is changed more widely than using hyb-MLR on each project. That is to say, hyb-MLR is more stable in expense than dyn-MLR for locating faults.

Similarly to RQ2, we also applied the prediction model, trained with NanoXml1, on Nanoxml2. The expenses of predicting 6 faults range from 18.47 to 46.37.

To Further verify the different methods (i.e., hyb-MLR and dyn-MLR) have significant differences in the expense of locating faults, we also used Wilcoxon rank-sum test to calculate the BH-corrected p-values when comparing our model on two different metrics (H_X and D_X). Although the p-value is a little greater than 0.05 on Chart, the p-values calculated on the other seven projects are far less than 0.05. That is to say, in most cases, the expense of hyb-MLR is lower than that of dyn-MLR.

Summary for RQ3: We observed that the hyb-MLR (applied our model Mulr4FL on hybrid metrics) has a significant improvement and stability compared to dyn-MLR (applied our model Mulr4FL on static metrics) in most cases.

4) RQ4: HYB-MLR VS. SFLs

In this section, we conduct comparisons on the expenses of locating faults between hyb-MLR and a set of state-of-the-art SFL (Spectrum based fault localization) techniques. Since previous studies (RQ2 and RQ3) indicate that hyb-MLR, which is applied to our model (Mulr4FL) on hybrid metrics, performs the best than sta-MLR and dyn-MLR, which are based on static and dynamic metrics, respectively. Therefore, we conduct another comparison between dyn-MLR (which is applied to our model (Mulr4FL) on dynamic metrics) and SFLs (which are based on dynamic metrics either). Here, SFL techniques we studied include Naish1, Naish2, Wong, Russel&Rao, and Binary.

For all studied SFL techniques, we also applied 5-fold cross-validation on two projects (NanoXml1, NanoXml2) and 10-fold cross-validation on the rest of the projects.

TABLE 6. Comparison of expense applying hyb-MLR, dyn-MLR and SFL.

Projects	hyb-MLR	dyn-MLR	Naish1	Naish2	Wong	Russel &Rao	Binary
Jtcas	21.02	24.66	30.35	33.97	43.30	33.02	43.68
NanoXml1	21.48	31.91	29.45	34.96	39.92	38.09	46.44
NanoXml2	20.59	30.30	30.55	37.07	39.73	43.69	41.15
Math	24.49	30.29	38.99	39.00	38.08	44.80	42.49
Chart	24.87	25.22	41.24	42.76	40.96	47.93	45.24
Closure	23.98	25.82	42.23	47.30	40.25	44.11	42.56
Lang	23.25	25.43	42.29	44.25	39.73	42.05	48.37
Time	22.86	24.59	42.81	43.43	40.37	43.49	44.89
Avg.	22.82	27.28	37.24	40.34	40.29	42.15	44.35

We statistic the means of expenses of each technique applied on eight projects, and calculate the average expense of each technique on all projects. The results are listed in Table 6.

From Table 6 we can see that the means of the expense of hyb-MLR are much lower than those of SFLs. And the average expense of hyb-MLR is 22.82%, while the averages of expenses of compared SFL techniques ranged from 37.24% to 44.35%. Similarly, the expenses of dyn-MLR are significantly lower than those of SFL, though both dyn-MLR and SFL techniques use only dynamic metrics of runtime information of examined projects. Furthermore, the average expense of dyn-MLR is also much lower than the average expense of other SFL techniques. Among all studied SFL techniques, Naish1 is the best technique than others, and Naish2 and Wong have similar performance, Russel&Rao and Binary have similar performance either.

Next, we further examine the difference between the techniques listed in Table 6 by using the Wilcoxon rank-sum test to calculate the BH-corrected p-values. Although these SFL techniques are proved to be different optimal groups with a theoretical analysis [33], the locating granularity is based on statement-level, not on method-level in their theoretical analysis framework. In our empirical study, we want to evaluate the difference between them on a method-level. The significance level is set at 0.05, which means that if the corrected p-values are less than 0.05, there is a significant difference between the two compared vectors. We compared hyb-MLR with five SFL techniques, all corrected p-values obtained are far less than 0.05 which means that hyb-MLR is significantly different from SFL techniques. Additionally, all corrected p-values obtained are far less than 0.05 when comparing dyn-MLR with SFL techniques. That means dyn-MLR is significantly better than SFL techniques. However, the corrected p-values calculated when comparing two SFL techniques are greater than 0.05, which means that there are no significant differences between these SFL techniques.

Summary for RQ4: We observed that both hyb-MLR (on hybrid metrics) and dyn-MLR (on dynamic metrics) have a significant improvement compared to SFL techniques (on dynamic metrics). And there is no significant difference between studied SFL techniques.

V. THREATS TO VALIDITY

In this section, we will discuss potential threats to the validity of our research, which mainly includes three parts: internal threats, externalthreats, and construct threats.

A. INTERNAL THREATS

In this paper, the proposed model is based on a set of metrics using software measurement. Therefore, the internal threat lies in the metrics used in our model. Although, the optimal static and dynamic metrics for fault locating are still unclear. We selected seven static metrics and four dynamic metrics that are proven to be fault-related by the previous empirical studies [15]. We further separate the fault-related metrics by conducting a bivariate analysis. In the future, we will extend our model to cover more areas of features and present more metrics.

B. EXTERNAL THREATS

There are only eight projects used for evaluating performance in our empirical study, and our experimental results may not be generalizable to all projects. To alleviate this threat, we select the projects from popular repositories, such as SIR and Defects4J, which are used widely in previous studies [36], [37]. Moreover, the scale of these projects from small to large, and the faults include manually seeded and real-world happened. Therefore, it can reduce the impacts in this aspect to a certain extent by using these projects. In the future, we want to verify the effectiveness of our proposed model on projects of different scales or other programming languages (such as Python).

C. CONSTRUCT THREATS

In our RQ2, RQ3, and RQ4, we calculated the expenses using Equation 7 which depends on the order of faults in the suspiciousness rank list. However, the calculation faces a dilemma when a faulty method has the same suspiciousness as other methods. To alleviate this threat, we choose a compromise named ML(Middle Line) strategy which has proven feasible [29] and also be used by other research works.

VI. RELATED WORK

This section briefly introduces the recent studies related to our work from two aspects: spectrum based fault localization and defect prediction.

A. SPECTRUM BASED FAULT LOCALIZATION

For improving the efficiency of program debugging, there have been a large number of researches on automatic locating faults. Among the various fault localization approaches proposed in previous studies, the Spectrum based fault localization(SFL) techniques constitute the main category. SFL techniques locate faults by calculating the suspiciousness of the entities of the program with some risk formulas which need to collect the coverage information of program entities.

Researchers proposed hundreds of formulas accompanied by empirical studies to support the efficiency of their approaches. Typical formulas are: Tarantula [4], Ochiai [38], Naish1, and Naish2 [31], etc. The main difference between these SFL techniques lies in the formulas used for calculating the suspiciousness of program entities and the coverage metrics gathered by trace tools. However, Xie *et al.* [33] proposed a theoretical analysis framework. Based on the framework, they compared the efficiency of 30 formulas and theoretically proved five of them to be optimal. Compared with the five optimal SFL formulas which only use four dynamic features, our approach combines coverage information with static features and dynamic features of the program under debugging, which can improve the efficiency of fault localization.

To further improve the efficiency of SFL techniques, researchers introduced program analysis techniques to fault localization. For example, Liblit *et al.* [39] proposed the CBI approach to isolate bugs by calculating the suspicious entities using their coverages when the predicate is true. Similarly, Liu *et al.* [40] proposed the SOBER to identify buggy predicate by considering the predicates value patterns in both passed and failed runs, respectively. Later, Zhang *et al.* [41] proposed the PRFL, a lightweight technique using the PageRank algorithm to recompute the spectrum by considering the contributions of different tests. Recently, Jiang *et al.* [42] proposed Predicate-based Fault Localization (PREDFL), which combines the spectrum-based fault localization (SFL) and statistical debugging (SD) into a unified model. With the best configuration for each dimension under the unified model, PREDFL can further reduce the cost of fault localization. Besides, a number of SFL techniques improve fault localization efficiency by using a hybrid spectrum of program slice and execution coverage. Such as HSFal [43], IPSETFUL [44], FPA-FL [45], etc. A recent work by Chaleshtari *et al.* [46] combines mutation techniques into fault localization with the help of program slicing, which aims to reduce the number of statements to be mutated. Different from the above approach, our model only needs to collect dynamic coverage and the static metrics of class methods, which lead to lower time and space costs.

B. DEFECT PREDICTION

Defect prediction aims to find out the defects(faults) in software systems. Numerous software metrics and statistical models have been developed for predicting fault. These metrics can be categorized into three types: size and complexity metrics, testing metrics, process quality metrics. An earlier study of defect prediction by Compton *et al.* [47] indicates that troublesome areas could be identified using two groups of specific metrics: predelivery defects and complexity of Ada packages. Later, Basili *et al.* [13] first introduced logistic analysis to defects prediction. They measure a student-developed C++ project used weighted metrics and then predicted the fault-prone classes. From then on, many defect prediction techniques were developed. Similar to Basili's work, Briand *et al.* [14] measure and predict faulty classes using logistic regression analysis on a set of static metrics. And other statistical models Bayesian Belief Networks (BBN) [48], analysis of variance (ANOVA) [49]–[51]), etc.) have been developed to determine if a program entity contains defects. Different from the above methods, our approach applies multivariate logistic regression analysis on both static and dynamic metrics to predict the fault-prone class methods.

Recently, machine learning techniques became popular for defect prediction [52]. Researchers proposed various defect prediction models based on algorithms such as deep learning, representation learning, and transfer learning [53]. Wang *et al.* [54] leverage Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees (ASTs), which bridge the gap between programs' semantics and defect prediction features. Similarly, Li *et al.* [55] proposed Defect Prediction via Convolutional Neural Network (DP-CNN) to leverage deep learning for effective feature generation. DP-CNN used a combination of learned features with traditional hand-crafted features for defect prediction. Recently, learning-based fault localization has been intensively studied. For example, Li *et al.* [56] present DeepFL, a tailored Multi-layer Perceptron (MLP) model, to automatically learn the most effective existing/latent features for precise fault localization. To further improve the efficiency of defect prediction, a series of combined methods are proposed. For example, Manjula *et al.* [57] combine genetic algorithm (GA) with deep neural network (DNN) for defect prediction. In particular, GA is used for feature optimization, and DNN is used for classification. Although machine learning-based techniques achieved great improvement, they are heavy-weighted approaches that usually consume large amounts of resources. Our approach is based on pre-designed program metrics (which can be measured very quickly) and runtime metrics (which can be traced on-the-fly). Furthermore, the metrics used by our approach are extensible according to program language and project types.

VII. CONCLUSION AND FUTURE WORK

Fault localization is a critical task to improve debugging efficiency in software maintenance. Multivariate logistic regression analysis model, by training on knowledge of legacy program debugging, can be used to predict fault location efficiently. In this article, we propose a novel framework Mulr4FL for locating source code faults based on multivariate logistic regression analysis. In particular, we construct an initial metrics-set of static metrics and runtime metrics of class methods. Next, we conduct a correlation analysis to select those fault-related metrics. Then, we combine the static metrics and dynamic metrics to a hybrid matrix to train the multivariate logistic regression model. With the trained model, we can predict the fault location at the class-method level effectively. Furthermore, we conduct empirical studies to verify the effectiveness of our proposed framework Mulr4FL on different metrics (static and dynamic) by

comparing it with the state-of-the-art SFL techniques. The experimental results indicate that multivariate logistic regression analysis can significantly improve the effectiveness of fault localization when static metrics are introduced into the model.

In the future, we first want to design more reasonable metrics and the optimal metrics set for better fault localization using multivariate logistic regression model. We will introduce more metrics into our study, such as KBCs(Key Block Chains) [58]. Second, we will study the orthogonalization of feature metric or construct an orthogonal feature metric set to alleviate the potential impact of non-orthogonal metrics on the fault localization results. Additionally, we consider introducing the weighted metrics in multivariate logistic regression for improving the accuracy of the fault localization. Finally, we also want to refine the multivariate logistic regression model or introduce other models, such as linear mixed model, to improve the accuracy of locating faults based on the obtained fault-related metrics.

REFERENCES

[1] 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.

[2] L. A. Graf, "Suggestion program failure: Causes and remedies," *Personnel J.*, vol. 61, no. 6, pp. 450–454, 1982.

[3] W. E. Wong and V. Debroy, "A survey of software fault localization," Dept. Comput. Sci., Univ. Texas at Dallas, Richardson, TX, USA, Tech. Rep. UTDCS-45-09, 2009.

[4] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2005, pp. 273–282.

[5] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proc. 6th Int. Symp. Automated Anal.-Driven Debugging (AADEBUG)*, New York, NY, USA, 2005, pp. 33–42.

[6] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *Proc. 2nd Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. (ESEC/FSE)*, in Lecture Notes in Computer Science, vol. 1687, O. Nierstrasz and M. Lemoine, Eds. Berlin, Germany: Springer, 1999, pp. 303–321.

[7] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng. (ICSE)*, Los Alamitos, CA, USA, 1981, pp. 439–449.

[8] X. Zhang, N. Gupta, and R. Gupta, "Locating faulty code by multiple points slicing," *Softw., Pract. Exper.*, vol. 37, no. 9, pp. 935–961, 2007.

[9] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using FindBugs on production software," in *Proc. Companion to 22nd ACM SIGPLAN Conf. Object Oriented Program. Syst. Appl. Companion (OOPSLA)*, 2007, pp. 805–806.

[10] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, New York, NY, USA, 2002, pp. 234–245.

[12] Y. Kim, S. Mun, S. Yoo, and M. Kim, "Precise learn-to-rank fault localization using dynamic and static features of target programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 1–34, Oct. 2019, doi: 10.1145/3345628.

[13] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[14] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, May 2000.

[15] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 706–720, Jul. 2002.

[16] W. E. Wong and V. Debroy, "Software fault localization," *Encyclopedia Softw. Eng.*, vol. 1, pp. 1147–1156, 2010.

[17] J. M. Landwehr, D. Pregibon, and A. C. Shoemaker, "Graphical methods for assessing logistic regression models," *J. Amer. Stat. Assoc.*, vol. 79, no. 385, pp. 61–71, Mar. 1984.

[18] E. R. Babbie, *The Practice of Social Research*. Boston, MA, USA: Cengage Learning, 2020.

[19] D. W. Hosmer and S. Lemesbow, "Goodness of fit tests for the multiple logistic regression model," *Commun. Statist.-Theory Methods*, vol. 9, no. 10, pp. 1043–1069, 1980.

[20] D. Grey, "Multivariate analysis," *Math. Gazette*, vol. 65, no. 431, pp. 75–76, 1981.

[21] E. T. Jaynes, "On the rationale of maximum-entropy methods," *Proc. IEEE*, vol. 70, no. 9, pp. 939–952, Sep. 1982.

[22] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, Apr. 2003.

[23] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 491–496, Jun. 1998.

[24] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[25] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Test. Anal.*, San Jose, CA, USA, Jul. 2014, pp. 437–440.

[26] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. 18th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, New York, NY, USA, Oct. 2003, pp. 30–39.

[27] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, Los Alamitos, CA, USA, 2005, pp. 342–351.

[28] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[29] S. Ali, J. H. Andrews, T. Dhandapani, and W. T. Wang, "Evaluating the accuracy of fault localization techniques," in *Proc. 24th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Los Alamitos, CA, USA, Nov. 2009, pp. 76–87.

[30] M. Schroeder, "A practical guide to object-oriented metrics," *IT Prof.*, vol. 1, no. 6, pp. 30–36, 1999.

[31] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11–43, 2011.

[32] W. E. Wong, Y. Qi, L. Zhao, and K. Y. Cai, "Effective fault localization using code coverage," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, Los Alamitos, CA, USA, vol. 1, 2007, pp. 449–456.

[33] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 1–40, Oct. 2013.

[34] T. Fushiki, "Estimation of prediction error by using K-fold cross-validation," *Statist. Comput.*, vol. 21, no. 2, pp. 137–146, Apr. 2011.

[35] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 157–168.

[36] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed, and R. A. Rasheed, "Multiple fault localization of software programs: A systematic literature review," *Inf. Softw. Technol.*, vol. 124, Aug. 2020, Art. no. 106312.

[37] A. Zakari, S. Abdullahi, N. M. Shagari, A. B. Tambawal, N. M. Shanono, J. Z. Maitama, R. A. Rasheed, A. Adamu, and S. M. Abdulrahman, "Spectrum-based fault localization techniques application on multiple-fault programs: A review," *Global J. Comput. Sci. Technol.*, vol. 20, no. 2, pp. 41–48, Mar. 2020.

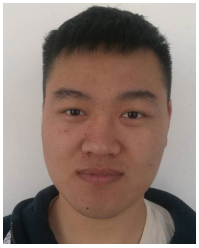
[38] R. Abreu, P. Zoeteweij, and A. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proc. 12th Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, 2006, pp. 39–46.

[39] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, Jun. 2005.

[40] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 286–295, Sep. 2005.
[41] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via PageRank," *IEEE Trans. Softw. Eng.*, early access, Apr. 25, 2019, doi: 10.1109/TSE.2019.2911283.
[42] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining spectrum-based fault localization and statistical debugging: An empirical study," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 502–514.
[43] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao, "HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices," *J. Syst. Softw.*, vol. 90, pp. 3–17, Apr. 2014.
[44] X. Sun, X. Peng, B. Li, B. Li, and W. Wen, "IPSETFUL: An iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra," *Frontiers Comput. Sci.*, vol. 10, no. 5, pp. 812–831, Oct. 2016.
[45] F. Feyzi and S. Parsa, "FPA-FL: Incorporating static fault-proneness analysis into statistical fault localization," *J. Syst. Softw.*, vol. 136, pp. 39–58, Feb. 2018.
[46] N. B. Chaleshtari and S. Parsa, "SMBFL: Slice-based cost reduction of mutation-based fault localization," *Empirical Softw. Eng.*, vol. 25, no. 5, pp. 4282–4314, Sep. 2020.
[47] B. T. Compton and C. Withrow, "Prediction and control of ADA software defects," *J. Syst. Softw.*, vol. 12, no. 3, pp. 199–207, Jul. 1990.
[48] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, Sep. 1999.
[49] T. M. Khoshgoftaar and N. Seliya, "Comparative assessment of software quality classification techniques: An empirical case study," *Empirical Softw. Eng.*, vol. 9, no. 3, pp. 229–257, Sep. 2004.
[50] T. M. Khoshgoftaar, N. Seliya, and N. Sundaresh, "An empirical study of predicting software faults with case-based reasoning," *Softw. Qual. J.*, vol. 14, no. 2, pp. 85–111, Jun. 2006.
[51] R. W. Selby and A. A. Porter, "Learning from examples: Generation and evaluation of decision trees for software resource analysis," *IEEE Trans. Softw. Eng.*, vol. 14, no. 12, pp. 1743–1757, 1988.
[52] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl. Soft Comput.*, vol. 27, pp. 504–518, Feb. 2015.
[53] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Softw.*, vol. 12, no. 3, pp. 161–175, Jun. 2018.
[54] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 297–308.
[55] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2017, pp. 318–328.
[56] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2019, pp. 169–180.
[57] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Comput.*, vol. 22, no. S4, pp. 9847–9863, Jul. 2019.
[58] J. Xu, Z. Zhang, W. K. Chan, T. H. Tse, and S. Li, "A general noise-reduction framework for fault localization of java programs," *Inf. Softw. Technol.*, vol. 55, no. 5, pp. 880–896, May 2013.

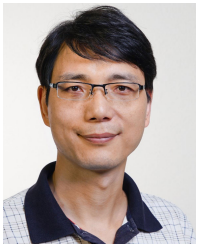
XIAOLIN JU (Member, IEEE) was born in April 1976. He received the B.S. degree in information science from Wuhan University, in 1998, the M.Sc. degree in computer science from Southeast University, in 2004, and the Ph.D. degree in computer science from the Chinese University of Mining Technology, in 2014. He is currently an Associate Professor with the School of Information Science and Technology, Nantong University, Nantong, China. His current research interests include software testing, such as collective intelligence, deep learning testing and optimization, and software defects analysis.

JIE QIAN received the B.S. degree in computer science from the Xuzhou University of Technology. She is currently pursuing the master's degree in computer science with Nantong University, Nantong, China. Her current research interests include software analysis and testing, web application testing, and fault localization technique.



ZHIHUA CHEN received the B.S. degree in computer science from Nantong University, Nantong, China, where he is currently pursuing the master's degree in computer science. His current research interests include software testing and analysis, software measurement, and defects prediction.

CHUNYU ZHAO received the B.S. degree in computer science from Nantong University, Nantong, China, where she is currently pursuing the master's degree in computer science. Her current research interests include machine learning, deep learning testing and optimization, and defects prediction.



JUNYAN QIAN received the Ph.D. degree in computer science from Southeast University, in 2006. He is currently a Professor with the School of Computer Science and Information Engineering, Guangxi Normal University, China. His current research interests include software dependability and reliability, software maintenance, and software development methodology.