A Review on Software Bug Localization Techniques using a Motivational Example

Amr Mansour Mohsen[1], Hesham Hassan[2], Ramadan Moawad[3], Soha Makady[4]
Computer Science Department, Faculty of Computers and Information Technology, Future University in Egypt, Cairo, Egypt[1, 3]
Computer Science Department, Faculty of Computers and Artificial Intelligence, Cairo University, Cairo, Egypt[2, 4]

***Abstract*—Software bug localization is an essential step within the software maintenance activity, consuming about 70% of the time and cost of the software development life cycle. Therefore, the need to enhance the automation process of software bug localization is important. This paper surveys various software bug localization techniques. Furthermore, a running motivational example is utilized throughout the paper. Such motivational example illustrates the surveyed bug localization techniques, while highlighting their pros and cons. The motivational example utilizes different software artifacts that get created throughout the software development lifecycle, and sheds light on those software artifacts that remain poorly utilized within existing bug localization techniques, regardless of the rich wealth of knowledge embedded within them. This research thus presents guidance on what artifacts should future bug localization techniques focus, to enhance the accuracy of bug localization, and speedup the software maintenance process.**

Keywords—Bug localization; bug localization artifacts; information retrieval; program spectrum

I. Introduction

Software maintenance is considered a continuous process in software projects. However, software maintenance is one of the most expensive stages in the software development life cycle [1]. According to Erlikh [2], maintenance consumes 70% and maybe up to 90% of the time of any product's life cycle. In addition to that, Hunt et al. [3] presented that the maintenance process takes above 50% of the software life cycle. Also, Lientz and Swanson [4] claimed that software maintenance spending from 20% to 70% of the efforts exerting on maintenance. Software Maintenance is defined by Sommerville [5] as "the modification of a software product after delivery to correct faults, to improve performance or other attributes". Software maintenance must be applied to improve the design, implement enhancements, and interface with other legacy software [6] to build a new one with some updates or solve bugs.

A different view of software maintenance [7] defines it as "error, flaw, or fault in a computer program or system that produces unexpected results or behavior". Once the bug occurs, the bug triaging and localization process is applied to solve the bug [7]. The process involves: (i) understanding the bug, (ii) assigning a maintainer, and (iii) bug localization within the source code, and (iv) bug fixing. The bug localization process is the action of determining the location of the bug in the software program [8]. However, locating the bug manually could be time consuming, cost consuming, and infeasible [10].

Several techniques have been utilized to localize bugs automatically, including: information retrieval [9], machine learning. [10], program spectrum [11], and program slicing [12]. Those techniques use different software artifacts like bug reports, stack traces, source code files. However, such techniques do not necessarily benefit from all the information present within those artifacts. For instance, techniques that utilize source code do not use the structural relationships between source code elements to locate bugs, although such information could improve the accuracy of bug localization. Hence, a review is conducted to identify the different artifacts utilized by bug localization techniques, and how well such artifacts' information gets utilized.

Furthermore, a motivational example is introduced. Within such motivational example, we present a running example that includes different software artifacts and a set of injected bugs. We applied various existing bug localization techniques that utilized subsets of the included artifacts, to locate the injected bugs within such example, and assessed various bug localization techniques on those bugs. What difference in this review that the process of motivational example helps in identifying the limitations of those bug localization. Besides it gives perspective for better utilization of the different software artifacts to increase the quality of the results of such bug localization techniques.

The rest of this paper is structured as follows. Section II will present the related works to software bug localization techniques. Section III presents the motivational example and its software artifacts. Three categories of bug localization techniques: information retrieval, machine learning, and program spectrum will be explained, and applied to the motivation example within Sections IV, V, and VI respectively. Section VII discusses the findings and concludes the review.

II. Related Work

A. Related Work on Information Retrieval

An information retrieval technique called (BLIA) bug localization using integrated analysis [9] proposed by Klaus Changsun et al. to illustrate the technique besides showing limitations. Such work utilizes different software artifacts like stack traces, comments, bug reports, and the history of code modifications are features utilized in the work.

Klaus Changsun et al. evaluated their work on three open-source projects: Aspect-oriented extension to Java (AspectJ), Widget toolkit for Java (SWT), and Barcode image processing library (Zxing). The number of bugs and source files that they worked on are as follows: AspectJ (284 bugs, 5188 source files), SWT (98 bugs, 738 source files), and Zxing (20 bugs, 391 source files). Five steps were followed to complete their approach. First, an information retrieval technique is used for measuring the similarity between the bug reports text and source code files called rVSM [13]. Then the structured data in the bug reports like bug description, bug summary and other stated before in the bug report artifact are analyzed and integrated with the above data. After that, if stack traces appeared in the bug report, they would be analyzed to extract the beneficial information to improve the results of retrieval. Moreover, the historical data for code modifications which is extracted from version control systems to predict the affected files and methods. Finally, similarity measurements were applied between the accumulative data from the above steps and the source code files. The results will be ranked by scores to the files of the code which is mainly expected to have the error. The proposed approach resulted in an enhancement in the mean precision over some other approaches like BugLocator with 54%, BLUiR with 42%, and BRTracer with 30%, and Amalgam with 25%.

Wen et al. proposed an information retrieval technique to localize bugs called FineLocator. FineLocator recommends the position of bugs based on method level [14]. It means that not only recommend the source file that contains bug but also the method contains bug. The proposed architecture consists of three main components are method extraction, method expansion, and method retrieval. The method extraction process is applied by extracting the methods names and their bodies using the abstract syntax tree for the code. Additionally, the timestamp for the methods is also extracted from version history systems and the dependence information for each method is also extracted. The first sub-component of method expansion is the semantic similarity measurement between methods. This step will be applied first by generating a numeric vector for each method by generating a bag of words and among all methods of the code. Then the scores are calculated between every two methods to know the similarity score between them. Then the call dependency is applied among the class level and the method level to enhance the similarity scores between methods. Besides, another score is calculated which is temporal proximity measure which calculates the difference in time of edit between the methods as the methods that edited in time near each other will be more probable to be near to each other. Then all the above scores are combined to one value which is the method augmentation value. They test their work on ArgoUML, Maven, Kylin, Ant, and AspectJ and enhance the performance of the method level by 20% MRR.

Yaojing et al. [15] proposed an approach with three main considerations which are 1) the fix history relationships with old bug reports, 2) word co-occurrence in the bug reports and source files, 3) The long source files. The proposed model consists of a supervised topic modeling technique called LDA for classifying the old bug reports and bug reports with special topic w. Then the word co-occurrence with words from bug reports that appear in the bug reports. In addition to the creation of the long source files and stack traces in bug reports. They test their work on 10-fold cross-validation. Also, the proposed model was applied to three main projects PDE with 3900 bug reports and 2319 source files, the platform with 3954 bug reports and 3696 source files, and JDT with 6267 bug reports and 7153 source files.

Mills et al. [16] constructed an approach trying to enhance the process of text retrieval bug localization by studying the most important elements of a bug report. A genetic algorithm is applied to find the optimal query to retrieve the true results from source files. Yu Zhou et al. construct an approach [17] that consists of three steps to classify bug reports: Classifying the summary part of each bug into (high, middle, and low) using a machine learner. It will help to increase the accuracy of bug localization systems. Then some structured features are used from the bug reports using a machine learner.

Additionally, the results are merged from the above steps and other machine learning algorithms are used. The authors manually classify the bug reports into six categories (BUG, RFE, IMPR, DOC, REFAC, other). Additionally, a voting is applied [18] between different developers to classify each bug report to label them. They need to classify either the bug report is a bug or not. They answer the question of that a given report is a corrective bug or not by using different fields in the bug report. Also, the proposed approach Combines text mining and a data mining approach to solve the problem. The approach evaluated using 3200 random reports from large projects like Mozilla, Eclipse, JBoss, firefox, and OpenFOAM. The Use Bugzilla as the bug tracking system. They use the only reports that are tagged by resolved or closed to analyze them. They consider multiple fields of the bug report like (textual summary, severity, priority, component, assignee, and reporter.

Alessandro Murgia et al… Tonelli [19] tried to make bug tracking systems linked with CVS to enhance the bug fixing and relations between different versions of the software and the bugs and also the end-users. Each commit component consists of (author when it was done, modified files, and commit messages). The work was stressed on fixing-issue commits. They manually labeled the data of commits to training their classifier through one author and this is a drawback as the author may do not know enough the data in the commits then maybe the classifier is biased to their labeling. Preprocessing steps from natural language processing are used like stemming and stop words removal to enhance the classifier. Additionally, some regular expressions are used to filter commits that relate to specific bugs. The features used to feed the classifier are the words extracted from the commits. They applied their experiments to Netbeans and Eclipse projects. The machine learning classifier got a precision of 99.9% for classifying fix issue and non-fix issue commits. The dataset used has not appeared as they didn't use a benchmark dataset. Besides, they identify the main terms used for bug-fixing issues like the fix, for, and bug. The support vector machines are classified with accuracy up to 99.9%.

B. *Related Work on Machine Learning*

In [52], the authors produced an approach for localizing the bugs automatically using ranking. The source code files are ranked to the most probably that contains the bug reported. Different features will be used as a bag of words used from source code files and bug reports. The similarity that is measured between bug reports and source code files using cosine similarity. Also, the API information is used to enhance the features. Another feature is collaborative filtering which is applied between similar bug reports. Additionally, the class names and the bug fixing frequency considered to be featured. They apply their experiment on AspectJ, Eclipse UI, JDT, SWT, and Tomcat. The average accuracy of 70% achieved all over the top 10 ranked files.

In [20], an approach proposed using deep learning with rVSM to enhance the process of bug localization. The revised vector space model (rVSM) is utilized to set up the features that are used in measuring the similarity between bug documents and source code files. The DNN is used to measure the relevancy of the term between the terms in bug reports and source code files. Also, another type of feature rather than terms is the metadata feature about source code files, it seems like logs about the file. The inputs are text similarity, metadata about source code files. They used DNN to learn all the features. They applied on different datasets like AspectJ, Birt, Eclipse UI, JDT, SWT, and Tomcat. They got an average precision of 0.52 using the tomcat dataset.

Dongsun Kim, Sunghun Kim, and Andreas Zeller proposed a model [21] with two phases to predict the files to be fixed. The bug report in many cases as mentioned by the authors may not contain sufficient information to help in predicting the files needed to be fixed. A machine learning approach is applied to classify the bug reports as predictable which means contain useful information or not predictable. The Features extracted from the bug reports are the summary, platform, operating system, severity, priority, and reporter. Then the model is trained using the specified machine learning and tested. Then in phase two, the predictable bug reports to be fixed are then entering a multi-class classification model to know the exact files to be fixed. The recommended model was evaluated using 70 percent of the dataset for training and 30 percent for testing. They achieved an average accuracy for predicting files to be fixed with 70 percent.

ERIC et al. [22] proposed a neural networks technique based on the code coverage data as a feature. This coverage data comes from applying virtual test cases to each line in the code. Then they feed them to a neural network. The technique was tested on four different benchmark datasets (Siemens, UNIX, Grep, and Gzip). They enhance the performance of examining lines of code than [23].

In [24], [25] a deep learning model are applied in order to localize bugs using source code files and bug reports. They got accuracy of applying on different benchmark datasets.

Liang et al. [10], proposed a deep learning system to localize bugs. Bug reports text terms are utilized besides the terms of source code files. The works are evaluated on four datasets (AspectJ, SWT, JDT, and Tomcat) with the following MAP (0.439, 0.457, 0.482, and 0.561).

C. Related Work on Program Spectrum

Jeongho et al. proposed a spectrum-based technique that localizes bugs based on the variables that are most probably suspicious [11] to rank the lines most probably contain bugs. A limitation discussed in this paper about previous work considering program spectrum that if there is an else block as an example and the block contains many lines. The outcome of the ranking of lines contains code will not be accurate and maybe the cause of the error be directly before the block. To overcome the above limitation, the variable-based technique proposed to keep track of mainly the information about the suspicious variables and their coverage in the code. First, the variable spectra are created by using the test cases as an input in addition to the execution trace data for each variable. Then the suspicious ratios are calculated by substituting the variable spectra with the coefficient's similarity. The final step is applied by rank the most variables that are most suspicious in descending order to the bug solver. The work was evaluated using the Exam score evaluation metric.

On the other side, Henrique et al. constructed a spectrum-based fault localization tool called Jaguar which stands for Java coverage fault localization ranking [26]. An architecture was formulated for the tool consists of two main components which are Jaguar Runner and Jaguar Viewer. The java runner component gathers the data for control flow spectra and data the data flow using different unit tests. After the data collection steps applied, then a metric score calculated using one of past known calculations Metric like [27]. After that, the mixed scores between data and control flow matrices are normalized for the suspicious parts of the code. Then the jaguar viewer colors the suspicious entities of the code according to their score with for different colors according to their danger. They assessed t their work based on the Defects4J dataset.

A new method that depends on the level of predicates not all the lines of the code was constructed by B´ela that utilizing the data from test cases and code coverage data [28]. This special type of spectrum-based fault localization took into consideration which methods will be hit in the run time of test cases to use these data in ranking the most suspicious methods. Additionally, different past research metrics for ranking that used for the lines of code as stated in [29] will be used at the method level. The pre-step to the algorithm is the building of the coverage matrix between the methods and the test cases. A graph will be generated from the coverage as the nodes of the graph represent the methods and the tests. The edges that will link different nodes with each other represent that a node that may be a test case will hit a node which is a method. Besides, the failed test cases will be marked in the graph. The first step was to calculate the edge weights by summing up the total methods that hit a failed test case to all methods. Then the values will be updated by calculating the average value of methods that cover failed test cases. The next step is to aggregate the values of edges to the method nodes. Finally, the nodes of methods values will be updated by calculating the resulted values concerning the number of test cases. They evaluated their work based on the Defects4J dataset that includes four projects with good results of the ranking.

Abubakar et al. proposed a graph-based technique for the spectrum-based technique based on the execution of the test cases [30]. The technique aims at localizing not only a single bug in the system but also multiple bugs during execution. The exploration of localizing multiple bugs due to dealing only with the bug affects the accuracy of localization as stated by the authors. The graph represented here is undirected where the nodes of the graph represent the program statements and the edges represent the execution between them. Degree centrality is a graph centrality to measure the importance of a node in a network which will indicate that the part of the code will be more probable to contain an error. Another measure in which closeness centrality was used for each node to know the shortest path length between the node and other nodes. The result of this step will affect the process of multiple bug localization. The technique is evaluated on about 5 out of 7 programs from the Siemens dataset (Dset6, Dset6, Dset7, Dset8, Dset9, and Dset11). In the experiment on single fault localization, 99% of the faulty version can be found by exploring only 80% of the code. In the two bug's version, about 99% of bugs found after exploring 70% of the executable code. They evaluated their work based on the exam score evaluation and the incremental Developer Expense (IDE) methods.

Program slicing according to [12] [31] is a debugging technique that formulates a slice of code which are statements that affect a variable. Static slicing is a type of program slicing that generates slices depend on control dependencies in the code. Another type of program slicing is dynamic slicing which works on reducing the amount of space generated by static slicing. Dynamic slicing creates the slice depend on the variable values at run time to reduce the number of statements of the program in the debugging. However, execution slicing as stated by [32] applied data flow tests to formulate the slice or a group of slices (dice) by detecting the most probable statements from the tests to have the bug.

III. Motivational Example

This section presents the software artifacts of the software system explained within the motivational example. These artifacts will be later the input the application of different bug localization techniques in the following sections. The system description will be discussed in subsection "A", a subset of system source code files will be presented in subsection "B", and a subset of the software bug reports are shown in section "C".

A. System Description

Consider a system for online shopping. The aim of the system is to be utilized for online shopping. The customer can browse some products, add them to his shopping cart then process the order. The order will be finalized, and the total amount will be calculated including taxes and the customer payment choice. The customer chooses a payment method and assigns it a profile as it is either cash, or by credit card, and the customer can update such payment method later. The administrator of the shop can add new computer products to the inventory with specific data. The shop has two main types of components: "DesktopLaptop" or "ComputerComponents". Also, the administrator can update taxes for any product, and products of the same type must be updated automatically.

Fig. 1 shows a partial class diagram the 'Online Shopping System' (OSS) including 9 classes. Customer class holds the customer's information and operations does like adding a newproduct to the shopping cart (addProductToShopping () method) and assign a payment (setPayementMethod () method). ShoppingCart class holds information about products selected by the customer. Payment Method class is an interface for the type of payment, and it has two subclasses PaybyCredit and PaybyCache, with specific attributes for payment. ComputerProduct class is a parent class that consists of the basic information of any computer product of the system. DesktopLaptop and ComputerComponents are child classes of ComputerProduct class, each with specific properties. Inventory class manages the inventory through the addProduct () method for adding products with their quantity to the system. A relationship exists between Customer and ShoppingCart classes because each customer must have a shopping cart to add products to it. The relationship exists between Customer and PaymentMethod since each customer must decide his payment method for online shopping. ShoppingCart and Inventory classes are in an aggregation relationship with ComputerProduct class, as both classes consist of computer products. Such software system has a set of software artifacts that are presented within the following subsections.

B. Motivational Example Source Code Files

A subset of source code files for the online shopping system is presented in this section. The source code for the "Shopping Cart" class is presented in Fig. 2. The source code for the "Inventory" class is shown in Fig. 3.

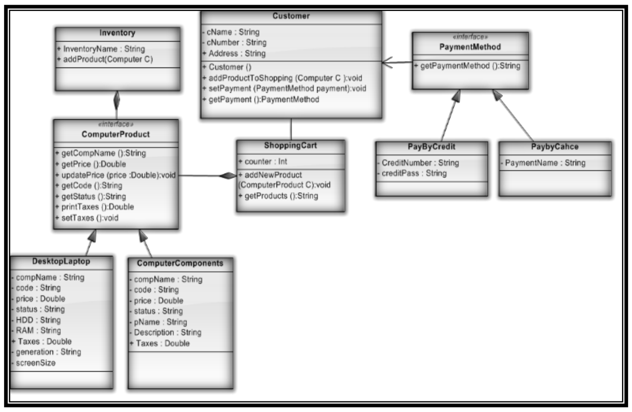
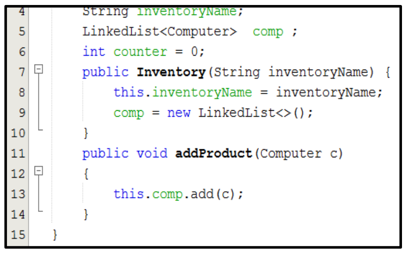
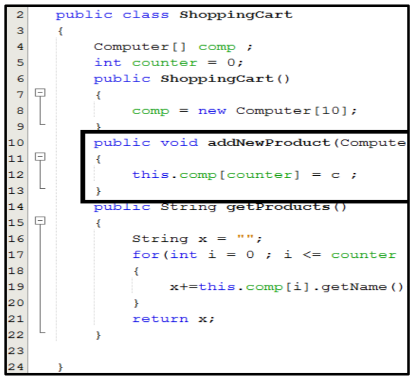


Fig. 1. A Partial Class Diagram of the Online Shopping System (OSS)



```
    String inventoryName;
    LinkedList<Computer>  comp ;
    int counter = 0;
    public Inventory(String inventoryName) {
        this.inventoryName = inventoryName;
        comp = new LinkedList<>();
    }
    public void addProduct(Computer c)
    {
        this.comp.add(c);
    }
}
```

Fig. 2. "ShoppingCart .java" Source Code.



```
public class ShoppingCart
{
    Computer[] comp ;
    int counter = 0;
    public ShoppingCart()
    {
        comp = new Computer[10];
    }
    public void addNewProduct(Compute
    {
        this.comp[counter] = c ;
    }
    public String getProducts()
    {
        String x = "";
        for(int i = 0 ; i <= counter
        {
            x+=this.comp[i].getName()
        }
        return x;
    }

}
```

Fig. 3. "Inventory.java" Source Code File from File from the use Case Example.

C. Motivational Example Bug Reports

A bug report is a terminology that refers to documenting and describing software bugs that appeared while running a software [33]. As stated by [33] a bug report can be submitted by different stakeholders related to the software project such as the tester or the developer or user to the system. They posted their bug reports on a bug tracking system [33] which is used mainly for open-source projects to track different bug-report changes, assigned to solve the bug, or any other discussions.

Four bug reports, for the used motivational scenario, are presented in this section. Three bug reports have the status "resolved fixed" and one new bug report has the status "New".

TABLE I. BUG REPORT 1

Bug Report 1	
Bug ID	1102
Bug Summary	The payment method didn't change
Bug Status	Resolved Fixed
Product	Normal user
Reported	Online Marketing Application
Version	5/5/2020
Bug Description	I purchased pc and two other products then when I proceed to the order, they give me a note the payment will be on the cache given; However, I updated my payment method to pay by credit before.
Stack Trace	-
Fixed Files	Customer.java
Fixed Time	7/5/2020
Test Cases	-

Bug report 1, shown in Table I, shows a user who had previously changed his payment method from using cash to using the credit card. When making a new purchase afterwards, the system still displayed that his payment method will be using cash. Bug report 2, shown in Table II, has the status New" as it will be fixed by our example. The bug appears with the user when adding a new product to purchase to his cart, the program crashed and stopped. Bug report 3 shown in Table III presented a solved bug by adding a new product to the store. When the user of the system adds a new product to the system, a crash occurred. The bug was solved by the maintainers and the source code file "Inventory. Java". Bug report 4 shown in Table IV presented a solved bug with getting an invoice for a purchasing process. It was found that the tax percent is calculated incorrectly however it is calculated before. The solution to the bug is found in the source file "ComputerComponents. Java".

Starting from Section III to Section V, different bug localization techniques will be discussed and applied to the motivational scenario showing how those techniques work and their limitations.

TABLE II. BUG REPORT 2

Bug Report 2	
Bug ID	1104
Bug Summary	Adding a PC to purchase cause an error
Bug Status	New
Product	Online Marketing Application
Reported	5/12/2020
Version	1.2
Bug Description	When trying to add a PC to purchase and browse some other components and added them then adding another pc to the cart it is crashed.
Stack Trace	-
Fixed Files	-
Fixed Time	-
Test Cases	-

TABLE III. BUG REPORT 3

Bug Report 3	
Bug ID	1201
Bug Summary	Error with adding a new product to the store
Bug Status	Resolved Fixed
Product	Online Marketing Application
Reported	5/5/2020
Version	1.1
Bug Description	When I trying to add new product to the store, the program crashed given the following error
Stack Trace	Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded at java.util.LinkedList.linkLast(LinkedList.java:142) at java.util.LinkedList.add(LinkedList.java:338) at Inventory.addProduct(Inventory.java:28) at project. main(project.java:15)
Fixed Files	Inventory.java
Fixed Time	5/9/2020
Test Cases	1201

TABLE IV. BUG REPORT 4

Bug Report 4	
Bug ID	1325
Bug Summary	Error with getting an invoice
Bug Status	Resolved Fixed
Product	Online Marketing Application
Reported	5/5/2020
Version	1.1
Bug Description	When the transaction is going to be fired, it calculates the total invoice wrong with a problem in taxes percent
Stack Trace	-
Fixed Files	ComputerComponents.java
Fixed Time	7/5/2020
Test Cases	-

IV. INFORMATION RETRIEVAL TECHNIQUES

Information retrieval (IR) [34] is finding or extracting beneficial data that may be documents of unstructured nature like text that answers information needs. In the bug localization process, the source code files, bug reports either old or new, stack traces artifacts [35] will be the unstructured text of the system being analyzed. The unstructured data like the natural text in the bug reports, stack traces and source code file terms. This data needs to be retrieved and ranked using specific queries to retrieve the file contains bug [35]. The information retrieval passes through steps from preprocessing and preparing different text sources to similarity measures.

A. *Case Study IR Experiment*

In this subsection, three experiments will be applied. First, historical similar bug reports artifact will be utilized. Then source code artifact will be utilized in the second experiment. Finally, Similar Bug Reports Experiment applied.

1) Similar bug reports application: The first bug localization technique to apply, is an information retrieval technique that uses similarity scores across bugs. Klaus Changsun et al. [9] proposed a technique to localize bugs using an information retrieval technique. The assumption of their work depends on that if there is a new bug report similar in its attributes to one of the old bug reports then the fixed source code file by this old bug report will be the recommended source code file to be fixed with the new bug report. Such technique was applied to calculate similarity scores between the three resolved bug reports and the newly added bug report within the presented motivational example. The first step is to convert each bug report to a text vector as shown in Table V.

Then the Term Frequency Inverse Document Frequency (TF-IDF) measure [36] will be applied to the text of the bug reports. The calculated similarity measure between the new bug report (i.e., bug report 2) and each of the old bug reports resulted in the following scores presented in Table VI: bug report 1 is 0.2, bug report 3 is 0.16, and bug report 4 is 0.11. The experiment resulted in that bug report 1 is the most similar bug report to the new bug report. It means that the fixed file within bug report 1 (Customer .java) in the old bug report 1 is the file that contains the bug.

To evaluate the presented experiment, the new bug report needed to be fixed manually to know the files that contain the bug. The result of the manual investigation that the source code file "ShoppingCart.java”. However, experiment 1 resulted in that the “Customer.java” is the file that contains the bug which means that the experiment 1 result is not true.

To understand why the applied bug localization technique failed to locate the source code file that contained the bug, a closer look is needed at the used bug reports. As per the bug’s description in Section 3.2, bug report 1 was fixed by a change in Customer.java, whereas bug report 3 was fixed by a change in Inventory.java. The similarity score between bug report 1 and the new bug report was higher than the similarity score between bug report 3 and the new bug report. Hence, the applied bug localization technique suggested fixing the same file that was fixed previously by bug report 1. Hence, the applied technique could lead to a wrong location based on the text used within the newly opened bug. Such text is usually written by an end user, who has no knowledge of the inner details of the source code. Hence, relying on the text of the bug report solely is one main drawback of that bug localization technique. Another drawback is the complete reliance of the technique on the presence of historically fixed bug reports to recommend resolutions for the new bugs. Such assumption is not realistic when developing new applications that do not have a repository of previously fixed bug reports.

TABLE V. TEXT VECTORS OF THE BUG REPORTS OF OSS SYSTEM

Bug Report	*Bug Report Text Vector*
Bug Report 1	[Payment, Method, change, Normal, User, Application, purchased, two, products, proceed, order, they, give, note, payment, cache, given, updated, method, pay, credit]
Bug Report 2 (NEW)	[Adding, PC, purchase, cause, error, Online, Marketing, Application, trying, add, browse, some, components, added, adding, another, pc, cart, crashed
Bug Report 3	[Error, with, new, product, store, Online, Application, store, program, given, following, Exception, thread, main, javalangOutOfMemoryError, GC, overhead, limit, exceeded, at, javautilLinkedListlinkLastLinkedListjava142, javautilLinkedListaddLinkedListjava338, InventoryaddProductInventoryjava28, project, mainprojectjava15Inventoryjava]
Bug Report 4	[Error, new, product, store, Resolved, Fixed, transaction, going, fired, calculates, total, invoice, wrong, problem, taxes, percent]

TABLE VI. SIMILARITY SCORES BETWEEN THE NEW AND OLD REPORTS

Old Bug Report	**Similarity Score with the new bug report**
Old Bug Report 1	0.20
Old Bug Report 3	0.16
Old Bug Report 4	0.11

2) Source code experiment: In the second experiment, Similarity Scores between the new Bug report and the source code files are applied [9]. As experiment 1, similarity scores will be calculated. The difference here that text similarity will be applied between the new bug report and project source code files with the TF-IDF technique.

The number of source code files is nine files as listed in Table VII with their text vectors. In the same table, the similarity score between these sources code files and the new bug report is calculated. The similarity results must be sorted in descending order. But in this case, there are no common words between the new bug report and all source files.

TABLE VII. TEXT VECTORS OF THE BUG REPORTS WITH SOURCE FILES TEXT OF OSS SYSTEM

Source Code Files (.java)	**Source File Text Vector**	**Similarity Score**
ShoppingCart	[ShoppingCart, Computer, comp, counter, Computer, addNewGoodsComputer, compcounter, getGoods, xthiscompigetName, compigetPricen]	0
Inventory	[Inventory, inventoryName, Computer, comp, counter, Inventory, inventoryName, addProductComputer, comp, add]	0
Customer	[Customer, cName, cNumber, PaymentMethod, payment, ShoppingCart, PaybyCahce, addProductToShoppingComputer, shaddNewGoodsc, setPaymentPaymentMethod, thispayment, getPaymentMethod]	0
Computer	[Computer, getName, double, getPrice, updatePricedouble, price, getCode, getStatus, printTaxes, setTaxesdouble, taxes]	0
ComputerComponents	[ComputerComponents, implements, Computer, compName, code, price, status, pName, Description, taxes, ComputerComponentsString, thiscompName, code, price, thispName, Description, taxes, Override, getName, return, getPrice, getCode, getStatus, status, printTaxes, setTaxesdouble, updatePricedouble]	0
DesktopLaptop	[DesktopLaptop, implements, Computer, compName, code, price, status, HDD, RAM, generation, screenSize, taxes, DesktopLaptopString, compName, code, price, HDD, RAM, generation, screenSize, taxes, status, Offered, Override, getName, return, getPrice, getCode, getStatus, printTaxes, void, setTaxesdouble, updatePricedouble]	0
PaybyCahce	[public, class, PayByCredit, implements, PaymentMethod, Override, String, getMethod, return, enter, card, number, pass]	0
PayByCredit	[PaybyCahce, implements, PaymentMethod, Override, getMethod, Pay, cache]	0
PaymentMethod	[PaymentMethod, getMethod]	0

Discussion: As per the above similarity calculation, the text of the bug report does not match the naming conventions used within source files. Hence, relying on similarity scores analysis between the source code and bug reports would not result in locating bugs. The absence of such similarity is attributed to the constructing of those bug reports by a normal user who uses terms not related to the developer terms used within the source code files. So, the bug localization system in this state will not resulted in a true source code file. An example comparing the bug report called "NEW BUG REPORT 1" text to source file text as an example "ShoppingCart.java". The similarity scores between all the source code files and the new bug report equal to zero as no common words between them. After computing the same way with all source files, the new bug report got zero similarity score with all of them.

3) Stack traces: Stack traces or execution traces represent the method calls during the execution of the application. When an error occurs during the execution and the program stops working or works in an unexpected way, the current state of the stack trace represents the method calls till the stopping point.

The presence of stack traces, as a part of the bug report, will enhance the accuracy of finding the source code file that contains the error [37]. From an information retrieval perspective, having a stack trace as a part of the bug report will result in higher similarity score between the bug reports and the source code files. Furthermore, stack traces result in faster manual debugging by the developers [38]. For example, Fig. 4 shows represents a bug from eclipse [39] how the file names involved in the error and the corresponding line number are shown appear or the line that contains an error are shown in Fig. 4 that line 13 contains the error in the source file inventory.java. Schroter et al. apply [40] a study on 3940 bug reports. 2,321 bugs reports observed that they are fixed contains stack traces with 60 %. Also, the mean lifetime of the bugs include stack traces is 2.73 Days compared with the remaining bug report does not contain stack traces with mean 4.13 days. So, in our case, bug report 2 with status new will not be solved using stack traces as the bug report does not contain it.

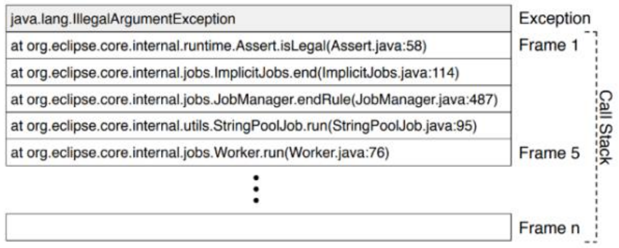

```
java.lang.IllegalArgumentException
at org.eclipse.core.internal.runtime.Assert.isLegal(Assert.java:58)
at org.eclipse.core.internal.jobs.ImplicitJobs.end(ImplicitJobs.java:114)
at org.eclipse.core.internal.jobs.JobManager.endRule(JobManager.java:487)
at org.eclipse.core.internal.utils.StringPoolJob.run(StringPoolJob.java:95)
at org.eclipse.core.internal.jobs.Worker.run(Worker.java:76)
```

Fig. 4. Sample Stack Trace Extracted from ECLIPSE Bug.

V. MACHINE LEARNING TECHNIQUES

Machine learning is a branch of computational algorithms that are designed to emulate human intelligence by learning from the surrounding environment [41]. Different bug localization techniques utilized different machine learning techniques to localize bugs automatically.

A. Case Study Experiment using Machine Learning

The most important step in applying a machine learning algorithm is the preparation of the features. For the motivational scenario, the features will be similarity scores between each old bug report and all the source code files. Each record or row of features represents this similarity scores with the source code files as presented in the following Table IX. Three bug reports will be utilized from the motivational example. Two of them will be used for training and one for testing. Their calculated similarity scores with the source code files to be the features. Also, the result of these features which the source file contains the error with each bug report appeared in the right cell of the row. As these bug reports are solved before and the files contains solved error already known. After preparing the training set with these two bug reports, it will be fed to the machine learning algorithm. Then the testing phase started by preparing the features for a bug report that we know its result before. The new bug report is prepared with its specific features. Then the machine learning algorithm will decide its decision which appeared in the last row in Table VIII. As shown in Table VIII, we have only two training examples with only two results. After running a machine learning algorithm, with feeding the training examples to the machine learning. Then feeding the testing example as to tell us the source code file containing the error. The result will be "ComputerComponents .java".

The file that contains the bug for new bug report 4 will be “ComputerComponents.java”. That means the result of the experiment is not true. Different reasons lead to such a wrong location of the bug. First, machine learning needs a huge training set to learn, otherwise it will not work properly [42]. Second, if the project has no old, solved bugs, machine learning will not be an applicable technique. In such a case, the only alternative would be to take training data from a different software project, like projects similar in nature to make use of their old bug reports. Some challenges will face this work: the language of the project may be different, the type of project as it may be desktop, web application or other.

TABLE VIII. FEATURES PREPARATION FOR MACHINE LEARNING ALGORITHM BUG REPORT

Training example	**Feature 1 (ShoppingCart)**	**Feature 2 (Inventory)**	**Feature N**	**Result (Source File)**
Training example 1 (*Bug report 1*)	0.11	0.1	…..	*Customer*
Training example 2 (*Bug report 3)*	0.3	0.1	0.3	*Inventory*
Testing example (Bug Report 4)	0.1	0.5	0.1	*?*

VI. PROGRAM SPECTRUM TECHNIQUES

Program spectra refer to the program entities that are covered during the execution of the program [29], [43]. Also, the spectrum based get some information executed from the programs as the test cases. There are several types of spectra [29] used in the spectra based fault localization as (program statements, variables, execution trace, execution path, path profile, execution profile, Number of failed test cases cover a statement and not, number of successful test cases that cover a statement and not, the total number of test cases that cover a statement, the total number of test cases that do not cover a statement, the total number of successful test cases, total number of failed test cases and the test case number. Such technique demands the presence of test cases, or the presence of correct program execution traces, to be applied. Furthermore, the technique demands having a large set of test cases to cover the lines of code that most probably contains an error.

A. Case Study Experiment using Program Spectrum

The main inputs to this experiment will be the test cases. Some operations must be applied to test cases to know the lines of code that will most probably have the error. Two test cases related to the motivational example are shown in Tables IX and X. When a new bug is reported, the output of the test cases (i.e., the spectrum) can be utilized with some equations to find the bug [44]. The source code file that the test cases will run on is presented in Fig. 2.

TABLE IX. TEST CASE 1 FOR OSS SYSTEM

Test Case 1	
Test Case Number	1
Test Case Inputs	Create Object of Inventory
Expected Output	Object Created without error
Actual Output	Created Successfully
Test Case Result	Success

TABLE X. TEST CASE 2 FOR OSS SYSTEM

Test Case 2	
Test Case Number	2
Test Case Inputs	Add new Product to Inventory
Expected Output	Added Successfully
Actual Output	Added Successfully
Test Case Result	Success

From the test cases and different inputs, test case number (1) will execute the following lines 6 to 13. And test case number (2) will cover 14 and 16. Then the number of records will be counted for each line to find the bug. The technique applied for spectrum depends on the statistical equation for every line of code covered by test cases. In Table XI, different program spectrum listed in section 2.2.3 as number of successful test cases that execute lines of code are presented with the occurrence of each spectrum with each line of code that are in the vertical rows. The hit of a spectra with a line of code represented by 1 and 0 for nit hitting. The criteria for each line like the number of successful test cases covered, the number of failed test cases, overall test cases in each line. We will consider that we have only two test cases for our bug localization task. Test case one presented in the above figure, it will hit the class source code from line 6 to 13 and the result of this test cases is a success. Test case two will hit the lines of code from 14 to 16. Consider Line 6 as shown in the table for illustration, we must calculate different spectrum for each line from the resulted test case as follows: Number of success test cases (NCS)covered line 6 will be test case number (1) only so the total will be one. The number of failed test cases (N_{CF}) covered in line 6 will be equal to zero as our test cases here are only two and both are successful. The number of test cases covers line 6 is equal to one as we list it before. Several test cases not covered in line 6 are equal to one which is test case number (2). The number of failed test cases not covered (NUF) line 6 is equal to zero as we have only two successful test cases. The last spectrum is the Number of Success Test Cases Not Covered which is one as test case 2 is a successful test case not covered line 6. The total number of failed test cases (NF) =0. The above steps will be calculated to all lines of the code as shown in the table. Then an equation is applied to calculate different spectra in one number for all lines of code then we have to sort these scores in descending order. The highest score will represent the line that contains the error.

The equation performed here that utilized different spectrum used from [44] presented in the (1):

$$Suspiciousness(Ochiai) = \frac{N_{CF}}{\sqrt{N_F*(N_{CF}+N_{CS})}} \qquad (1)$$

The score to line 6 will be equal to NCF = 0, NF=0, NCS=1 by substituting, the result will equal to zero. The above process will be repeated for every line of code then sorted but here all scores equal to zero.

The main limitation of program spectrum is that the many test cases need to be analyzed, to find the bug then many test cases to be tested to find a true solution which affects the time and performance of bug localization process [45]. Unfortunately, the results are equal, also we need to compute many test cases that affect the time to find the source code file contains bug [45].

TABLE XI. DIFFERENT PROGRAM SPECTRA FOR A SOURCE CODE FILE FOR OSS SYSTEM

Different Program Spectra	**Code Line**									
	6	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
Success Test Cases Covered	1	1	1	1	1	1	1	1	1	1
Failed Test Cases Covered	0	0	0	0	0	0	0	0	0	0
Test Cases Covered	1	1	1	1	1	1	1	1	1	1
Test Cases not Covered	1	1	1	1	1	1	1	1	1	1
Failed Test Cases not Covered	0	0	0	0	0	0	0	0	0	0
Success Test Cases not Covered	1	1	1	1	1	1	1	1	1	1
Total Score	0	0	0	0	0	0	0	0	0	0

VII. Discussion and Conclusion

This paper presents a review to explore different software bug localization techniques. The exploration done through presenting different past works. Additionally, a motivational example is applied to show how these techniques are working presenting their limitations; also, the software artifacts that are utilized and which are not utilized.

Two main findings are presented: (1) Some software artifacts are not properly utilized in the process of software bug localization. (2) The current software bug localization techniques suffer from some limitations. We elaborate on those findings as follows.

Finding 1: Many bug localization systems use information from both bug reports and source files. Previous research [10], [46], [47], [48], [49], [50], [51], [15] utilized the natural text of the bug reports with terms of the source files. Method names and the abstract syntax trees are used from source code files [10]. However, the changes that applied to each source code file among different from version control systems used by [49], [15]. Also, application interface descriptions text has been utilized by [49] , [50]. Test cases are also used where successful test cases and failing test cases are used to find the most probable error [22] [11] [26].

However, several artifacts are not utilized in the bug localization process. They are software requirements, use cases, classes' relationships within the source code, software architecture, and different comments between developers or written discussion between them of old bug reports that may affect the process of localization mentioned. If we have bug report text data as stated above. Text data can be linked to requirements text, which can be then, shorten the search with source code files to specific files of a definite module.

Finding 2: Different software bug localization techniques are applied in the process of bug localization (Information Retrieval, Machine Learning, and Program spectrum). These techniques suffer from some limitations and this appeared from applying the motivational example.

Information retrieval techniques suffer from the problem of the dependence on natural unstructured text. Those techniques depend on matching the new bug report text to any of the old bug reports, and hence recommending the fixed file of the old bug report. But such technique may not take us to the true old bug report depending on how the bug report is written, which varies greatly between developers and end users of the system. This issue appears as well if we attempt to measure the text similarities between the new bug report and the source code files. Also, the lack of old bug reports for the same project may prohibit applying the technique altogether.

Machine Learning techniques will not work properly in two situations. In the first situation, that we train the bug localization system on some software projects, and the new bug appears in a different project. Hence, the bug localization system will not give the exact source code that contains the error. The past works [46] [47] [48] [49] [11] are applied on one of the datasets and tested on the same dataset. The second situation when we do not have enough training examples to train the machine learning algorithm.

Test case-based techniques as program spectrum and program slicing are depending on test cases to localize bugs. The main limitation comes with performance and time to test the whole system to find the bug. Also, the huge number of test cases is to be examined to find the bug.

Accordingly, we anticipate that by utilizing additional software artifacts, and additional information from previously utilized software artifacts, we can improve the accuracy of bug localization, and extend its applicability even to projects that do not have historical information about the source code of the fixed bugs.

REFERENCES

[1] S. M. H. Dehaghani and N. Hajrahimi, "Which factors affect software projects maintenance cost more?," Acta Informatica Medica, vol. 21, no. 1, p. 63, 2013.

[2] L. Erlikh, "Leveraging Legacy System Dollars for E-Business," IT Professional , vol. 2, no. 3, pp. 17-23, 2000.

[3] H. B., T. B. and M. K., "Software Maintenance Implications on Cost and Schedule," in 2008 IEEE Aerospace Conference, 2008.

[4] B. P. Lientz and E. B. Swanson, Software maintenance management, Addison-Wesley Longman Publishing Co., 1980.

[5] I. Sommerville, Software Engineering, Addison-wesley, 2007.

[6] A. Kumar and B. S. Gill, "Maintenance vs. reengineering software systems," Global Journal of Computer Science and Technology, vol. 11, no. 23, 2012.

[7] D. Cubrani´c, "Automatic bug triage using text categorization," in the International Conference on Software Engineering & Knowledge Engineering, Alberta, 2004.

[8] W. W. Eric, G. Ruizhi, L. Yihao, R. Abreu and W. Franz, "A survey on software fault localization," IEEE Transactions on Software Engineering, pp. 707-740, 2016.

[9] K. Youm, J. Ahn and E. Lee, "Improved bug localization based on code change histories and bug reports," Information and Software Technology, pp. 177-192, 2017.

[10] S. LU, W. MEILIN and Y. YUXING, "Deep Learning With Customized Abstract Syntax Tree for Bug Localization," IEEE Access 7 , vol. 7, pp. 116309-116320, 2019.

[11] K. Jeongho, K. Jindae and L. Eunseok, "VFL: Variable-based fault localization," Information and Software Technology, p. 179–191, 2019.

[12] T. Frank, " A survey of program slicing techniques," Journal of Programming Languages, vol. 3, pp. 121-189, 1995.

[13] J. Zhou, H. Zhang and D. Lo., "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in Software Engineering (ICSE), 2012 34th International Conference on, IEEE, 2012.

[14] Z. Wen, L. Ziqiang, W. Qing and L. Juan, "FineLocator: A novel approach to method-level fine-grained bug localization by query expansion," Information and Software Technology, pp. 1-15, 2019.

[15] W. Yaojing, Y. Yuan, T. Hanghang, X. Huo, L. Ming, X. Feng and L. Jian, "Bug Localization via Supervised Topic Modeling," in 2018 IEEE International Conference on Data Mining (ICDM)., 2018.

[16] C. Mills, P. Jevgenija, P. Esteban, B. Gabriele and H. Sonia, "Are Bug Reports Enough for Text Retrieval-based Bug Localization," in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018.

[17] Y. Zhou, Y. Tong, R. Gu and H. Gall, "Combining text mining and data mining for bug report classification," Journal of Software: Evolution and Process, vol. 228, no. 3, pp. 150-176, 2016.

[18] M. D'Ambros, M. Lanza and R. Robbes, "An extensive comparison of bug prediction approaches," in Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on. IEEE, 2010.

[19] A. Murgia, G. Concas and M. Marchesi, "A machine learning approach for text categorization of fixing-issue commits on CVS," in the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (p. 6). ACM., 2010.

[20] A. N. Lam, A. T. Nguyen, H. A. Nguyen and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering, 2015.

[21] D. Kim, Y. Tao, S. Kim and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," IEEE transactions on software Engineering, vol. 39, no. 11, pp. 1597-1610, 2013.

[22] W. ERIC and Q. YU, "BP neural network-based effective fault localization," International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 4, pp. 573-593, 2009.

[23] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization," in the 20th IEEE/ACM international Conference on Automated software engineering, 2005.

[24] Z. Ziye, L. Yun, W. Yu, T. Hanghang and W. Yaojing, "A deep multimodal model for bug localization," Data Mining and Knowledge Discovery, vol. 35, no. 4, pp. 1369-1392, 2021.

[25] B. Qi, S. Hailong, Y. Wei, Z. Hongyu and M. Xiangxin, "DreamLoc: A Deep Relevance Matching-Based Framework for bug Localization," IEEE Transactions on Reliability , 2021.

[26] R. Henrique, d. A. Roberto, C. Marcos, S. Higor and K. Fabio, "Jaguar: a spectrum-based fault localization tool for real-world software," in 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018.

[27] A. Rui, Z. Peter, G. Rob and G. Arjan, "A practical evaluation of spectrum-based fault localization," The Journal of Systems and Software, p. 1780–1792, 2009.

[28] V. B´ela, "NFL: Neighbor-Based Fault Localization Technique," in IEEE 1st International Workshop on Intelligent Bug Fixing (IBF), 2019.

[29] S. Higor, C. Marcos and K. Fabio, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges," arXiv preprint arXiv:1607.04347, 2016.

[30] Z. Abubakar, P. ,. Sai and Y. C. Chun, "Simultaneous localization of software faults based on complex network theory," IEEE Access, pp. 23990-24002, 2018.

[31] M. Weiser, "Programmers use slices when debugging," Communications of the ACM, vol. 25, no. 7, pp. 446-452, 1982.

[32] W. Eric and D. Vidroha, "A Survey of Software Fault Localization," Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45 9, Texas, 2009.

[33] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang and H. Mei, "A survey on bug-report analysis," Science China Information Sciences, vol. 58, no. 2, pp. 1-24, 2015.

[34] U. Cambridge, Introduction to information retrieval, 2009.

[35] S. Wang and D. Lo, "Amalgam+: Composing rich information sources for accurate bug localization," Journal of Software: Evolution and Process, pp. 921-942, 2016.

[36] A. Aizawa, "An information-theoretic perspective of tf--idf measures," Information Processing & Management, vol. 39, no. 1, pp. 45-65, 2003.

[37] W. Shaowei and L. David, "Amalgam+: Composing rich information sources for accurate bug localization," Journal of Software: Evolution and Process , vol. 28, p. 921–942, 2016.

[38] N. Bettenburg, R. Premraj, T. Zimmermann and S. Kim, "Extracting structural information from bug reports," in The 2008 international working conference on Mining software repositories, 2008.

[39] S. Adrian, B. Nicolas and P. Rahul, "Do Stack Traces Help Developers Fix Bugs?," in 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), 2010.

[40] S. Adrian, B. Nicolas and P. Rahul, "Do stack traces help developers fix bugs?," in 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), 2010.

[41] M. Sayed, R. K. Salem and A. E. Khder, "A Survey of Arabic Text Classification Approaches," International Journal of Computer Applications in Technology, vol. 95, no. 3, pp. 236-251, 2019.

[42] K. J. Haider and Z. K. Rafiqul, "Methods to Avoid over-Fitting and Under-Fitting in Supervised Machine Learning (Comparative Study)," Computer Science, Communication & Instrumentation Devices, pp. 163-172, 2015.

[43] T. Reps, T. Ball, M. Das and J. Larus, Software Engineering—Esec/Fse'97, Springer, 1997.

[44] A. Rui, Z. Peter and J. Arjan, "An evaluation of similarity coefficients for software fault localization," in 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06), 2006.

[45] V. László and B. Árpád, "Test suite reduction for fault detection and localization: A combined approach.," in 014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering , 2014.

[46] H. Xuan, T. Ferdian, L. Ming, L. David and S. Shu-Ting, "Deep Transfer Bug Localization," IEEE Transactions on Software Engineering, pp. 1-12, 2019.

[47] X. Yan, K. Jacky, M. Qing and B. Kwabena, "Bug Localization with Semantic and Structural Features using Convolutional Neural Network and Cascade Forest," in of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. ACM, 2018.

[48] H. Xuan, L. Ming and Z. Zhi-Hua, "Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code," in IJCAI, 2016.

[49] Y. Xin, B. Razvan and L. Chang, "Learning to Rank Relevant Files for Bug Reports using Domain Knowledge," in In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014.

[50] N,. An, T. N. Anh, A. ,. Hoan and N. N. Tien, "Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports," in 30th IEEE/ACM International Conference on Automated Software Engineering, 2015.

[51] S. Jeongju and Y. Shin, "FLUCCS: Using Code and Change Metrics to Improve Fault Localization," in the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018.