AN AUTOMATIC STATEMENT LEVEL BUG LOCALIZATION TECHNIQUE

Shanto Rahman
Master of Science in Software Engineering,
Institute of Information Technology, University of Dhaka
Class Roll: MSSE 0302
Registration Number: 2010-713-303
Session: 2010-2011

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

Master of Science in Software Engineering

Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh

© IIT, University of Dhaka, 2016

AN AUTOMATIC STATEMENT LEVEL BUG LOCALIZATION TECHNIQUE

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

Shanto Rahman

To my parents.

Acknowledgements

I would like to thank my supervisor Associate Professor Dr. Kazi M. Sakib for his support and guidance during my candidature. His constant encouragement was one of the main driving forces that kept me motivated. I would like to thank him for all his help with the research directions and experimentation. All his comments and suggestions were invaluable. His very demanding nature of supervising as well as pushing me to the limits have helped me grow as a researcher.

I would like to thank my thesis reviewer Dr. Muhammad Masroor Ali as well for her involvement in excellent feedbacks and suggestions to enhance this work further.

I am also thankful to my committee members for serving even at hardship. I also want to thank them for letting my defense be an enjoyable moment and for their brilliant comments and suggestions.

Also I would like to thank other faculty members for their participation and constructive feedback in the production of the thesis.

I am thankful to all other staffs at IIT, University of Dhaka for creating a pleasant work environment.

I am expressing ever gratefulness to all my fellow classmates whose advice, feedback and cooperation is truly incomparable.

I am also thankful to Ministry of Posts, Telecommunications and IT, Government of the People's Republic of Bangladesh for granting me ICT Fellowship 2015-16.

Finally, I am indebted to my parents for the many years of hard work and sacrifices they have made to support me. Without them, this thesis would never have started.

Abstract

Manual bug localization is a time consuming and challenging task, as error can propagate from a large variety of source code. To reduce the manual interventions, automatic bug localization technique is adopted. To do so, some bug localization techniques suggest source code files or methods as buggy. However, these suggestions of class level or method level requires more inspections into the source code to find the actual buggy statements. So, a fine grained such as statement level bug localization technique is needed.

In this thesis, a Statement level Bug Localization (SBL) technique is proposed where statements are extracted from the method level bug localization. This research is based on two groups; one for identifying accurate buggy methods by source code search space minimization (termed as MBuM) and another for statement level bug localization by using those buggy methods.

In MBuM, dynamic analysis has been applied to discard the irrelevant source code. Then two sets of corpora, one for code and another for bug report are generated. Since method size has a relation for being buggy, a modified Vector Space Model (mVSM) is proposed to incorporate the size of a method with VSM. Afterwards, the similarity scores between both sets of corpora are calculated for a list of buggy methods.

In the second phase, for each suggested method from MBuM, a Method Statement Dependency Graph (MSDG) is constructed. A relationship is established between each node and it's predecessor nodes of MSDG by constructing a Node Predecessor-node Dependency Graph (NPDG). Afterwards,

the bug report and each node of NPDG are processed to measure the similarity between those using VSM. Based on the similarity scores, statements are ranked as buggy.

Experiments have been conducted in two phases; one for the accuracy of MBuM and another for SBL. To measure the accuracy of MBuM, both theoretical justification and experimentation have been performed. In the experiment, two open source projects namely Eclipse and Mozilla are used. Here, for both cases 60% bugs are localized at the first position and other 40% bugs are suggested at the 2nd position while other existing techniques cannot do. Three open source projects named as Eclipse, SWT and PasswordProtector are used to measure the effectiveness of SBL. The results show that 28%, 37.5% and 40% bugs are localized at the top for Eclipse, SWT and PasswordProtector respectively. These show that SBL can effectively localize the buggy statements.

Publications

1. "An Appropriate Method Ranking Approach for Localizing Bugs using Minimized Search Space" in 11th *International Conference on Evaluation of Novel Approaches to Software Engineering*, Rome, Italy, IEEE, 2016.

2. "An Improved Bug Localization using Structured Information Retrieval and Version History" in *18th International Conference on Computer and Information Technology (ICCIT)*, IEEE, 2015.

3. "A Statement Level Bug Localization Technique using Statement Dependency Graph" in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, IEEE, 2016. (Submitted)

Contents

SignaturePage ii

Dedication iii

Acknowledgements iv

Abstract v

Publications vii

List of Tables xii

List of Figures xiii

1 Introduction **1**

1.1 Motivation 2

1.2 Research Question 4

1.3 Contribution of this Research 7

1.4 Thesis Organization 8

2 Background Study **10**

2.1 An Overview of Software Bugs 11

2.1.1 What is Bug? 11

2.1.2 Impact of Bugs 11

2.1.3 Bug Report 15
2.1.4 Software Bug Localization 17
2.1.5 Bug Fixing 17
2.1.6 Program Analysis for Software Bug Localization 18
2.2 General Procedures of Software Bug Localization 23
2.2.1 Text Retrieval Based Bug Localization 23
2.2.2 Code Corpora Creation 26
2.2.3 Indexing 29
2.2.4 Query Construction 31
2.2.5 Ranking and Retrieval 32
2.3 Summary 34

3 Literature Review 36
3.1 Existing Research Work 37
3.2 Class Level Bug Localization 38
3.2.1 Source Code Static Analysis Based Techniques 38
3.2.2 Source Code Dynamic Analysis Based Techniques 48
3.3 Method Level Bug Localization 51
3.3.1 Source Code Static Analysis Based Techniques 51
3.3.2 Source Code Dynamic Analysis Based Techniques 55
3.4 Summary 57

4 A Method Ranking Approach for Localizing Software Bugs 58
4.1 Introduction 59

4.2 Model of Proposed System 63
4.3 Minimized Solution Search Space for Improving Bug Localization Accuracy 64
4.4 Methodology of Method level Bug localization using Minimized search space 69
4.4.1 Code Corpora Creation 70
4.4.2 Indexing 73
4.4.3 Bug Report Corpora Creation 75
4.4.4 Retrieval and Ranking of Buggy Methods 75
4.5 Case Study 77
4.5.1 Experimental Setup 78
4.5.2 Objectives of the Case Studies 78
4.5.3 Elements of the Case Studies 79
4.5.4 Case Study: Searching from Eclipse Help 79
4.5.5 Research Questions 82
4.6 Discussion 91
4.7 Summary 92

5 A Statement Level Bug Localization Technique 93
5.1 Introduction 94
5.2 Methodology of Statement Level Bug Localization 97
5.2.1 Generation of Method Statement Dependency Graph (MSDG) 98

5.2.2 Generation of Node Predecessor-node Dependency Graph (NPDG) for Similarity Score Calculation 102
5.3 Result Analysis 106
5.3.1 Data Collection 107
5.3.2 Evaluation Metrics 109
5.3.3 Research Questions and Evaluation 111
5.4 Summary 115

6 Discussion and Conclusion 118
6.1 Search Space Minimization for Enhancing Bug Localization Accuracy 119
6.2 A Statement Level Bug Localization Technique 120
6.3 Threat to Validity 121
6.4 Future Direction 122
6.4.1 Future Direction Related to this Thesis 122
6.4.2 Future Direction Related to the Domain 123

7 Appendix 124
7.0.3 Stop Words 124
7.0.4 Language Specific Keywords 127

Bibliography 128

List of Tables

4.1 Actual buggy methods for chosen bugs in Eclipse 81

4.2 Scenarios and queries used for ECLIPSE bugs 84

4.3 The 1st relevant method against five bugs in Eclipse 85

4.4 The suggestion of buggy methods using different bug localization techniques in Eclipse 85

4.5 Actual buggy methods for chosen bugs in Mozilla 86

4.6 Scenarios and queries used for Mozilla bugs 88

4.7 The 1st relevant methods in Mozilla 89

4.8 The suggestion of buggy methods using different bug localization techniques in Mozilla 89

5.1 The volume of considered bugs 108

5.2 The performance of SBL by considering 100 bugs 112

List of Figures

2.1 Security Bug 13

2.2 Performance Bug 13

2.3 Functional Bug 14

2.4 Deliberate Errors 14

2.5 Masked Errors 15

2.6 Infeasible situation 15

2.7 Bug report 16

2.8 Abstract Syntax Tree 19

2.9 Bug report and its relevant source code 20

2.10 Method call graph of source code 22

2.11 Shared terms (%) between the patched classes and bug report [1] 25

2.12 Steps of Lucen indexing 30

2.13 Vector Space Model Representation for documents and queries . 33

2.14 The representation of Latent Dirichlet Allocation (LDA) 34

3.1 Co-change Matrix 43

3.2 Overall Framework of Wang et al. [2] 45

3.3 Some stereotypes of source code method used in [3] 54

4.1 Functional block diagram of MBuM 71

4.2 Example of source code indexing 74

4.3 Relevant methods retrieved by dynamic execution tracing for bug ♯74149 80

4.4 Ranking provided by different bug localization techniques in Eclipse 90

4.5 Ranking provided by different bug localization techniques in Mozilla 91

5.1 Source code 99

5.2 MSDG for *calculateProfit* 100

5.3 Representation of MSDG containing the details of a node 103

5.4 Node Predecessor-node Dependency Graph (NPDG) corresponding to Figure 5.2 104

5.5 Bug report Id 31779 109

5.6 Patch of Eclipse bug report Id 31779 110

5.7 Demonstrating the effects (in case of 100 bugs) of considering predecessor nodes using Top N rank in Eclipse 114

5.8 Demonstrating the effect (100 bugs) of predecessor nodes based on Top N rank of SWT 115

5.9 Demonstrating the effect (100 bugs) of predecessor nodes based on Top N rank PasswordProtector 116

5.10 Analysis of the impact of the predecessor node MRR 116

5.11 Analysis of the impact of the predecessor node MAP 117

Chapter 1

Introduction

Automatic software bug localization is a technique by which the buggy locations of the source code can be identified with minimum manual interventions. An accurate identification of buggy locations can reduce the developer effort significantly by providing an actual buggy location which is needed to fix a bug. Existing techniques suggest classes or methods as buggy which consume lots of developer effort to find the actual buggy statements. Hence, an accurate statement level bug localization technique is needed.

In software development, errors and mistakes commonly occur due to the programming fault which produce unexpected results. After having that unexpected behavior, users report against the error or bug so that the bug can be fixed as soon as possible. Through bug tracking systems (such as Bugzilla [4], TFS [5]), developers receive the bug reports. After that, developers localize the bugs in the source code. If the developers have to localize the bugs manually, it would take a considerable amount of time. As a result,

the time of bug fixing increased significantly. To ameliorate this situation, an automatic bug localization technique is needed to reduce the overall bug fixing effort by identifying the buggy locations within a short period of time.

Bug localization can be divided into two types such as coarse and fine grained. Examples of coarse grained bug localizations are class [6] and method [7] level bug localization whereas, statement level can be considered as fine grained bug localization. The main objective of bug localization is to identify as much granular level buggy locations as possible so that the time consumption for localizing bugs can be reduced at a large rate. Based on this concept, this chapter first introduces the motivation of software bug localization at granular level. Later, the research questions and a brief discussion of the contribution are given. Finally, the organization of the thesis is provided.

1.1 Motivation

Software bugs have been a fact of life for programmers which are generated from the beginning of a program. When a bug occurs, developers fix that bug and in the phase of bug fixing, at first developers identify the locations of the bug manually. Software bug localization is a technique by which the identification of the buggy locations into the source code is performed. However, this task is a daunting one because a program often consists of more than million of source code statements (for example, Eclipse 3.0.2 contains

1,86,772 statements). Bug localization is also difficult due to the several reasons [7], [6] such as

- Software Aging: Like human, software may be aged. An aged software is commonly written using old technology which may be failed when new technologies have been emerged. Due to software aging, it is hard to find actual buggy locations manually which increases the cost of the software [8].

- Documentation Deficiency: In many software industry, proper documentation is not always maintained. As a result, the developers do not aware of the required functionalities of the bugs which increases the software maintenance effort [9].

- Developer Mobility: A developer may change job at any time from one industry to another. As a result, industries recruit new developers who may not handle the software properly because of the lacking of enough knowledge about the bug [10].

- Others: Sometimes the Quality Assurance team (termed as QA) are not enough skilled. As a result, the quality of the bug reports become very poor which may mislead the developers to localize the software bugs [11]. Unstructured and inappropriate naming convention of the source code is another reason because of failing to identify buggy locations. In real life, inexperienced developers may use inappropriate naming convention in the source code which violates the appropriate design

standard by incorporating code smells [12]. As a result, it is difficult to manually find the actual buggy locations from this huge source code [11]. (These may also hamper the automatic bug localization techniques as well.)

From the statistics, it is found that localizing software bugs takes 60-70% effort of bug fixing [11]. Meanwhile, large effort may lead to increase the software cost. However, an automated localization technique can reduce the software cost by minimizing developer effort during bug fixing.

Automatic software bug localization identifies the buggy locations without conducting any manual interaction. In literature, bugs are localized at the class level or method level. Class level suggestion provides a list of classes where the bug can stay. This provides a large solution search space where a bug may stay and after getting the probable buggy classes, developers have to traverse each statement of the suggested buggy classes and from that, buggy statements are identified. This class level suggestion consumes lots of time to traverse the source code class because a class often contains a large number of statements.

As an example, if 10 classes are suggested as buggy while each class contains on an average 800 statements, and the actual buggy statement is located at the $6th$ class of its suggested list, for finding actual buggy statement, around $800 \times 6 = 4800$ source code statements should be traversed.

In method level suggestion, a list of probable buggy methods are identified where also manual search into the source code is needed to find the actual buggy statements. As a result, developers effort have been increased.

A fine grained buggy location can be identified by statement level suggestion and this can minimize the software cost than other two levels (that is, class and method level). This is because by only traversing the list of suggested statements, developers can find the actual buggy statements. Unfortunately, there does not exist any statement level bug localization technique. Hence, an accurate statement level bug localization is yet to be proposed.

1.2 Research Question

Considering the above mentioned issues, the objective of this research is to develop a fine grained bug localization technique such as statement level bug localization. To suggest a statement level bug localization, obviously coarse grained techniques should be the primary input of the proposed approach. However, existing class level or method level bug localization techniques consider whole source code as their input which hampers the accuracy of bug localization. This is because, unnecessary information may create biasness. Therefore, in this research first the accuracy of the method level bug localization is increased by accurate suggestions which may discard the irrelevant source code. After that, this more accurate suggested methods are used as the input of statement level bug localization. To do so, the following research questions need to be answered:

How to suggest buggy locations at more granular level (that is, statement level)?

This question can be answered by the following sub-research questions.

1. How to improve the accuracy of bug localization techniques by minimizing solution search space?

   Existing software bug localization techniques commonly follow either source code static analysis or dynamic analysis or the combination of both. However, none of those minimize the search space rather these consider the irrelevant information of the source code during static analysis. Hence, here search space reduction is proposed which introduces two further research questions.

   - How to remove irrelevant information from the source code?

     For removing irrelevant information from the large solution search space, only bug specific source code can be taken by considering source code execution traces.

   - How to process the bug report and source code to measure the similarity between these?

     To find the buggy locations, the similarity between bug reports and source code are measured by proposing modified Vector Space Model (mVSM). To do so, bug report and source code texts are processed using several text retrieval based techniques such as stop words removal, programming language specific keywords removal,

multi-word identification, semantic meaning extraction and stemming.

2. How an effective statement level bug localization technique can be developed?

   In this phase, the suggested buggy methods returned by answering the previous question can be used as the input. Since a statement contains very few information about a bug, the information of the probable buggy methods are taken. Here, the way of increasing valid information of a statement is another question that needs to be answered. Apart from these, the way of measuring similarity between each statement of the source code and bug report is also proposed. This also introduces two further research questions and those are given below.

   - How to increase the valid information of a statement?

     A single statement often contains very few information about a bug. For example, Bug Id- $31,779$ is related to line 122[1] (that is, $child = createChildNodeFromFileSystem\,(node,\ parentLocalLocation,\ localName);$). However, line 122 contains very few information about the bug. So, to increase the valid information, each statement needs to be combined with its predecessor nodes (that is, the statement which affects another statement). For example, $y = 5;\ a = 2;\ x = y + a$. Here, x depends on the y and a. So,

[1] https://bugs.eclipse.org/bugs/attachment.cgi?id=3472&action=diff

the information of $x = y + a$ can be increased by combining $y = 5$ and $a = 2$.

- How to measure the similarity with the bug reports?

  To measure the similarity between the bug reports and each statement of the source code, Vector Space Model (VSM) can be used. Finally, for each statement, similarity for a bug report is measured and based on the descending order of the score, the buggy statements are ranked.

1.3 Contribution of this Research

To answer the above research questions, this research contributes to a statement level bug localization technique in an accurate manner. These contributions are summarized as follows.

At first, the proposed technique identifies only the problematic region by executing the source code with suitable test suites where irrelevant source code has been discarded. This source code execution trace does not provide any contents of the methods except method call graph, so static analysis has been applied to extract the contents of the methods. Later, the contents of the source code and bug report have been processed by applying multiple text retrieval techniques such as stop word removal, multi-word identification, programming language specific keywords removal, semantic meaning extraction and stemming. Finally, the similarity is measured between each method of the source code and bug report, and based on this similarity score,

probable source code buggy methods are ranked. This approach answers the first sub-research question which is stated in Section 1.2.

For the experiment, three case studies have been performed on Eclipse and Mozilla open source projects. The experiment shows that the proposed technique locates 60% bugs at the first position, while other techniques rank no more than one (20%) bugs at the top.

In this phase, a statement level bug localization technique has been proposed. Here, the probable buggy methods which are suggested by answering the previous research question are used as minimized search space. For each suggested buggy method, a statement dependency graph has been prepared resulting a Method Statement Dependency Graph (MSDG). Afterwards, each statement and its immediate predecessor nodes are combined to increase the valid information for each node. As a result, a Node Predecessor-node Dependency Graph (NPDG) has been generated. For each node of NPDG, text of each node are processed using stop word removal, programming language specific keywords removal, multi-word splitting, semantic meaning extraction and stemming. The text of each node are compared with the bug report using information retrieval based technique (that is, VSM) to measure the similarity. Finally, a list of probable buggy statements are ranked which depend on the similarity scores.

To show the result of the statement level bug localization technique, Eclipse, SWT and PasswordProtector projects were used. The procured results demonstrate that the proposed system has at least 45.8% Mean Re-

ciprocal Rank (MRR) and 38.5% Mean Average Precision (MAP) for Eclipse, and maximum 66% MRR and 61% MAP for PasswordProtector respectively.

1.4 Thesis Organization

The rest of this thesis answer the above mentioned research questions in more details. The organizations of this thesis are described as follows.

- **Chapter 2: Background Study**

  To understand the proposed technique, the necessary things will be described here. In this chapter, the preliminary knowledge about the software bug localization are described which may help to understand the proposed technique. So, the overview of the software bugs, its classification, types of program analysis for localizing software bugs are introduced. Along with this, general procedures of text retrieval based bug localization will also be discussed.

- **Chapter 3: Literature Review**

  In this chapter, existing state of the art bug localization techniques which suggest both class or method level buggy locations will be presented. From that, the limitations of existing bug localization techniques will also be described which may lead to propose a new bug localization technique.

- **Chapter 4: Search Space Minimization for Enhancing Bug Localization Accuracy**

Here, source code search space will be minimized by ranking probable buggy methods. This will be performed by measuring the similarity between the buggy methods and bug report.

- **Chapter 5: A Statement Level Bug Localization Technique**

  In this chapter, a novel statement level bug localization technique will be proposed by only considering the suggested probable buggy methods returned in Chapter 4.

- **Chapter 6: Discussion and Conclusion**

  This chapter will be presented the overall discussion of the thesis. Later, threat to validity of the proposed technique is described. Finally, the whole thesis will be concluded by finding future research direction.

Chapter 2

Background Study

"Failure is not an option. It comes bundled with the software." - Stalingraad [13]. A software often contains series of errors which requires more maintenance effort to fix these bugs or errors. For fixing a bug, at first developers manually identify the buggy locations and this consumes lots of time. It is noted that bug fixing consumes 50% to 80% of the total development and maintenance effort [14], [15]. Among these, more than half of the effort is needed to manually find the actual buggy locations in the source code [16]. On the other hand, an automatic software bug localization technique can provide the buggy locations without any human interventions which can minimize the software maintenance effort. For a better understanding regarding bug localization, this chapter first introduces the overview and classification of the software bugs. Later, the categories of source code analysis have been performed because program analysis is one of the key process in automatic

bug localization. Finally, a general procedure of software bug localization is presented.

2.1 An Overview of Software Bugs

In the software evolution, bugs are very common. To fix a bug, several processes are performed such as finding bugs into the source code, bug report generation from the buggy scenario, bug localization and bug fixing. Due to the presence of bugs, a software may be standstill. The below discussion holds the details of bugs followed by the impact analysis of those bugs. Besides, a brief overview of software bug localization and bug fixing techniques are described.

2.1.1 What is Bug?

Bug is a jargon used for describing an inexplicable defect of the source code. The term of "bug" is first introduced by the word "bugge" which is derived from "bugbear" and "bugaboo" [17].

Software bug is an error, mistake, defect or fault that leads the program to crash or produce invalid output [18]. More specifically, when a software fails to satisfy its users expectation, it generates bug. Most of the bugs arise from the mistakes or errors made by the developers into a program's source code. To fix these bugs, the cost of the software is increased at a large rate by also consuming developers effort.

2.1.2 Impact of Bugs

The bugs may have several kinds of effects on the software. Some bugs may have severe impacts or some may have very little effects on the users. So, based on the effects of bugs, two types are identified and those are:

- High impact bugs
- Low impact bugs

The high impact bugs represent that the availability of these bugs may hamper the users greatly. After surveying existing literature, six types of high impacted bugs are found [19]. The details of these are given below.

- Surprise bugs: Surprise bugs hamper the workflow of developers, since it appears in unexpected time (for example, during post-release) and unexpected locations (for example, files that rarely change in pre-release) [20].
- Dormant bugs: Dormant bugs refer to a bug that is introduced in one version of a system and not reported until the next immediate version is emerged [21].
- Blocking bugs: This type of bugs block other bugs from being fixed [22]. If several kinds of bugs are related to one another, it may happen that by fixing one bug, other bugs may be appeared one by one. Here, the first bug is known as the blocking bug.

```
public class SecuirtyBug {
public class AllHosts implements HostnameVerifier {
public boolean verify(final String hostname, final SSLSession session)
    {
        return true;
    }
}
```

Figure 2.1: Security Bug

```
public void main(String args[])
{
    for (int i=0; i<=100; i++)
    {
        calculateProfit(10);
        x=x+10;
    }
}
```

Figure 2.2: Performance Bug

- Security bugs: A security bug impacts on the uses of software products directly [23]. Through the availability of this type of bug, an attacker can gain the information about the system. Figure 2.1 presents an example of security bug.

- Performance bugs: Performance bugs occur when the programming errors create significant performance degradation [24]. Inefficient iteration, repeated execution of same things, inefficient data copy, inappropriate API usage, etc. are the results of performance bugs. Figure 2.2 is an example of occurring a performance bug. In this figure, within the loop, *calculateProfit* method is called with a static parameter (that is, 10). Due to having this method call within the loop, this method will be executed 100 times which may create a significant performance issue.

```
String userName= userNameAmount.getKey();
Double savedAmount=userNameAmount.getValue();
if(savedAmount<=withdrawAmount)
{
      currentMoney=savedAmount +withdrawAmount;
      updateAccount(accountNumber,accountType,
      userName, currentMoney);
}
else
System.out.println("Invalid Amount");
```

Figure 2.3: Functional Bug

- Functional bugs: The bugs which deviate a system from executing the desired functionalities are known as functional bugs. The functional bug is introduced when a new feature has been implemented or a bug has been fixed [25]. Several kinds of functional bugs are available such as Arithmetic scenario bugs, Logic bugs, Syntax bugs, etc. Arithmetic bugs are caused when unexpected arithmetic expression are used. For example, division by zero. Logical bugs occur when the source code contains an error in the development of logical operation. For example, if a scenario is actually related to add some money and if the code multiplies some money, that will be the logical errors. Figure 2.3 presents a logical bug. The main motivation of this code is to withdraw money which basically needs to be subtracted from the original amount. However, instead of the subtraction of the withdrawal money, this source code adds the withdrawal amount with the current money which is a logical or functional error.

```
if(layout.charAT(i++)!='[')
{
      layout.charAT(-i);// Error occurs
}
```

Figure 2.4: Deliberate Errors

```
String newName="";
if((x[i]<32)11 (x[i]>128))
{
      newName +=",";
}
else {
      newName += new String(x,i,1);
}
```

Figure 2.5: Masked Errors

Besides this high impact bugs, some of the low impact bugs are also available [26]. The details of few low impact bugs are given below.

- Deliberate errors: This type of errors seem to be correct. However, runtime exception may be the result of deliberate errors. Figure 2.4 represents an example of a deliberate error. Here, error occurs when an invalid memory address is tried to be accessed.

- Masked errors: Masked error occurs into a source code if the code contains error, but does not provide any warning against that code. Figure 2.5 is an example of masked errors where the condition holds an error.

- Infeasible statement, branch, or situation: This type of errors occur when the developer fails to infer easily by direct examination of the

```
  String Name=null;
  try{
      Name=expextedTypeCode.member_name(i);
  }catch(BadKind badkind){// Impossible
}
```

Figure 2.6: Infeasible situation

source code. Figure 2.6 presents an infeasible situation which may be occurred in the source code.

2.1.3 Bug Report

To fix the above mentioned categories of software bugs or errors, Quality Assurance (QA) team reports the faulty scenarios. It is noteworthy that most of the cases QA are unaware of the source code rather QA just check whether the system behaves according to its specification or not. On the other hand, developers are fully aware of the source code rather than the behavior of the problematic scenario. It is needed to build up a bridge between the QA and developers so that the developers can easily understand the problem which QA faced. Bug report builds such kind of bridges between the QA and developers of the programs.

A bug report contains bug summary, descriptions, bug reproducible approach, severity of that bug, etc. Bug summary holds the title of the bug whereas description contains the details of a bug. The reported bugs are widely known as bug report. Using the bug reports, developers can fix these bugs by incorporating his/her domain knowledge about that buggy scenario.

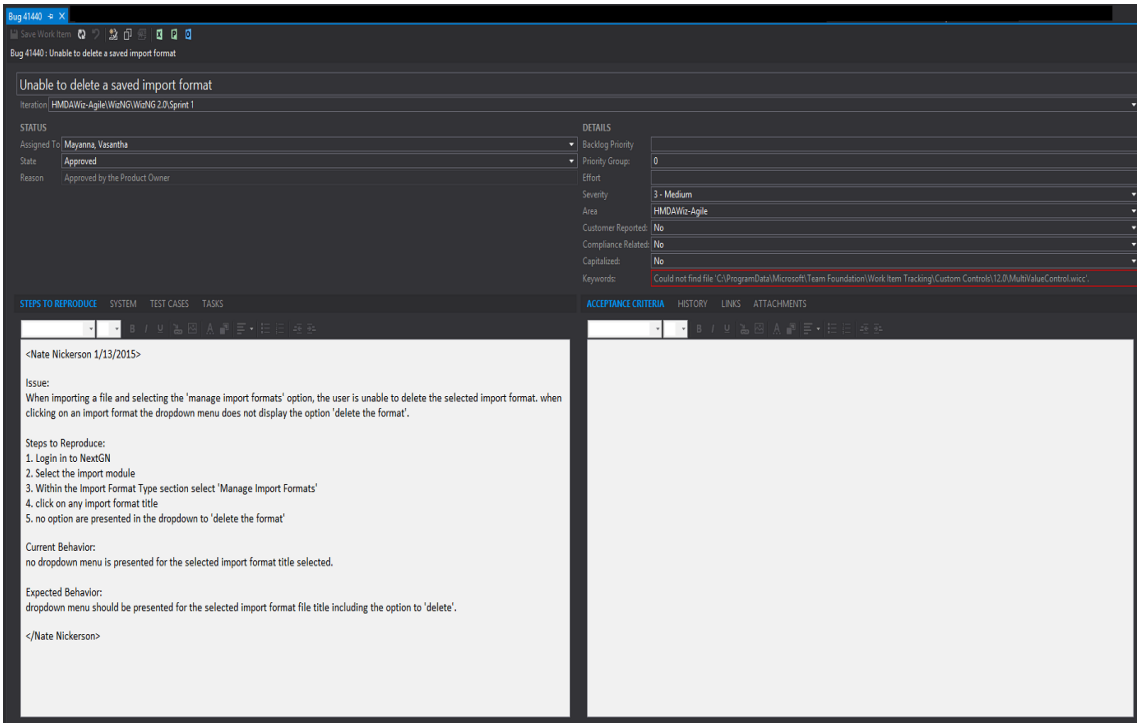

Figure 2.7: Bug report

An example of a reported bug is shown in Figure 2.7 which is taken from a software industry named as Orion Informatics LTD[1]. In Figure 2.7, bug title, steps to reproduce, current behavior, expected behavior of the bug are available to help the developers for fixing the bug.

2.1.4 Software Bug Localization

Bug localization is a technique by which developers can identify the buggy locations into the source code. Localizing bugs is not an easy task because most often the volume of a software is too large in terms of the total num-

[1]http://www.orion-informatics.com/

ber of statements and functionalities. Furthermore, bug localization may be affected, due to the software aging, documentation deficiency, developer mobility, information unavailability of bug report, unstructured and inappropriate naming convention of the source code, etc. [15], [7].

Automatic bug localization basically minimizes the huge time consumption in the software maintenance phase. As a result, the fixing time of bug can be faster. Most of the cases, localizing these bugs consume half or more time of its total maintenance effort [16]. The details of bug localization is also discussed in later sections.

2.1.5 Bug Fixing

Bug fixing refers to a change into the software product intended to permanently resolve a bug. At the beginning of software bug fixing, Quality Assurance (QA) teams or users report against a faulty scenario. When developers find a bug report, they try to fix the reported bug using his/her domain knowledge and bug report information. In software bug fixing, two steps are commonly followed. Among these, the first one identifies the buggy locations and the second one is to fix the bug.

From the above discussion, it is clear that each element (such as bug report, localization of bugs into the source code and bug fixing) is connected with others. Among these, localizing software bugs is one of the vital process in software maintenance phase because this consumes huge effort of developers. Hence, an automatic identification of buggy locations are needed. To

localize bugs automatically, program analysis is commonly performed which helps to find the buggy locations easily. Below discussion holds the general procedures of program analysis.

2.1.6 Program Analysis for Software Bug Localization

The program which contains bugs is known as buggy code. The objective of maintaining a software project is to fix the bugs from the source code. So, at first buggy locations have to be identified and this can be done by performing program analysis. Program analysis is a process by which the source code is reviewed, inspected and analyzed. The structure of the source code and the more detail information of a software can be identified by the program analysis. As software bug localization is performed by establishing the relationships between the bug report and source code, program analysis is needed [11], [27] and [16]. Source code static and dynamic analysis are commonly done in program analysis. The details of these two analysis techniques are described below.

Static Analysis of the Source Code

Source code static analysis is a technique where all possible paths of the source code are explored. Static analysis depends on the program syntactic and semantic information of the source code [28] instead of considering the runtime information of a software [29], [30], [6]. Static analysis of the source code can be performed using Model Checking [31], Data-Flow Analysis and

Abstract Interpretation [32]. This can be performed using some of the tools such as BLAST [33], Clang [34], Microsoft's SLAM Toolkit [35], Abstract Syntax Tree (AST).

Source code static analysis identifies the structure and details of the program. The structures of the source code classes, methods and statements are preserved using AST. Figure 2.8 presents an example of AST where each statement prepares a branch of the tree and the parent node is the program super node such as package name. This figure contains a portion of a source code and that is why *program* has been written as a parent of the tree. In AST, a new branch has been developed for each statement and with that branch each element of the statement has been asserted.

As an example, a statement containing '$x := a + b$', creates a branch under the *program* node. For each operator, a new branch is generated and the operands are inserted. That is why '=' creates a branch where one side contains 'x' and another side contains '$a + b$'. Due to obtaining an operator '+' in '$a + b$', a new branch is created where one side contains 'a' and the other side contains 'b'.

Source code static analysis provides a list of source code words which are used to identify the buggy locations. The source code which has large textual similarity to a bug report provides that the code is more liable for generating that bug. Thus bug localization is performed by establishing a relationship between the source code and bug report. To find the relationships between the bug reports and source code, Information Retrieval (IR) techniques get

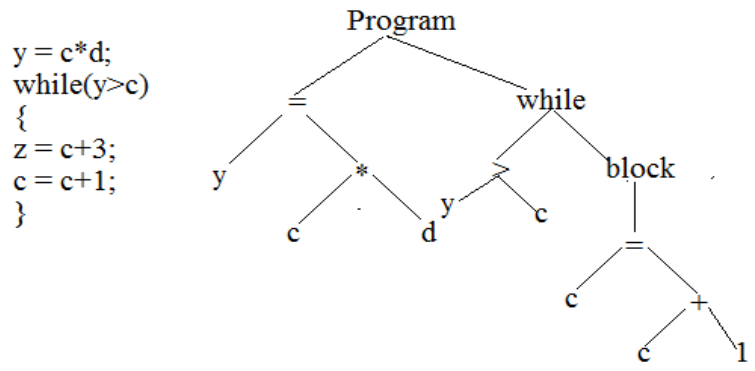

Figure 2.8: Abstract Syntax Tree

much popularity. As an example, the effects of IR techniques are described. Figure 2.9 presents a bug report and corresponding buggy source code where the upper part contains a bug report[2] and the lower part holds corresponding source code for Eclipse (version 3.1).

In Figure 2.9, bug summary and descriptions are available in bug report where several keywords such as *Pinned*, *console*, *views*, *Pin*, etc. exist. Similarly, the corresponding buggy source code named as *ConsoleView.java* contains multiple keywords such as *ConsoleView*, *fPinned*, etc. From this figure, it is clear that the concept of the bug report and source code is similar, the only difference exist in the selection of words. Hence, by applying text processing techniques, the similarity may be found between the source code and bug reports.

Now the bug report and source code can be considered as two text documents to identify the similarity between those. Based on this textual simi-

[2]Bug report ID- 80720- https://bugs.eclipse.org/bugs/

Bug ID: 80720
Summary: **Pinned console** does not remain on top
Description:
*Open two **console views**, ... **Pin** one **console.** Launch another program that produces output. Both **consoles display** the last launch. The **pinned console** should remain **pinned.***

Source code file: **ConsoleView.java**

```
public class ConsoleView extends PageBookView
    implements IConsoleView, IConsoleListener {...
      public void display(IConsole console) {
          if (fPinned && fActiveConsole != null) { return;}
      } ...
      public void pin(IConsole console) {
            if (console == null) {  setPinned(false);
            } else {
                if (isPinned()) { setPinned(false); }
                display(console);
                setPinned(true);
            }
      }
}
```

Figure 2.9: Bug report and its relevant source code

larity, buggy classes or methods even buggy statements can be ranked. The ranking depends on the similarity scores. Large similarity scores represent higher ranking of the buggy locations.

However, static analysis may hamper the accuracy of bug localization techniques by considering unnecessary details (such as whole source code textual information) for a bug. This is because for generating a bug, usually whole source code is not used which may lead the degradation of accuracy. As bug localization follows a probabilistic approach, so the more the unnecessary information is considered, the more the biasness is introduced. Additionally,

it fails to identify run time bugs due to taking only the static information of the software.

Dynamic Analysis of the Source Code

Source code dynamic analysis is another technique by which bug localization can be performed. In dynamic approach, buggy locations are identified using system run time information along with suitable test suites [36], [37], [38]. It reduces irrelevant search space and discovers relevant source code by running the software project. Dynamic analysis usually provides the call graph of a class or a method. Figure 2.10 shows a method call graph where one method calls other methods. In this figure, two classes have been considered and those are *class*1 and *class*2. From this figure, it is clear that at first the '*main*' method has been executed. Then the *function*1 and *function*2 of *class*1 and *class*2 have been executed respectively. Afterwards, *function*1 of *class*1 calls *function*2 of that class. *function*2 of *class*1 calls *function*3 of *class*1 and *function*1 of *class*2 and so on.

Dynamic analysis is used in statistical debugging approaches [39], [40], [41]. This approach collects program statistical information by taking run-time information. These collected information are further used for predicting faulty code. Jones et al. [42] identifies the execution of each statement using passing and failing test cases. These help developers to inspect a program's suspicious statements. Cleve and Zeller visualize the causes of failures by comparing the states of program using failing and passing tests [43]. However, these approaches depend on the quality of the test suites.

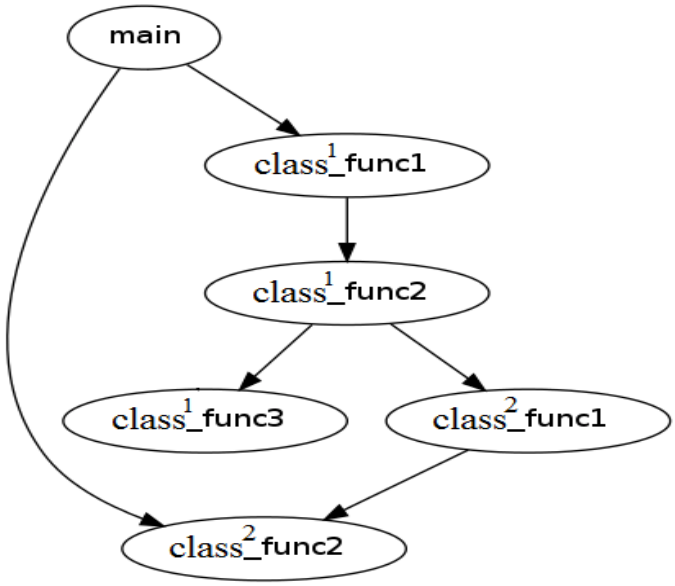

Figure 2.10: Method call graph of source code

Dynamic analysis is comparatively more complex than static analysis of the source code. Dynamic analysis minimizes the solution search space by discarding irrelevant source code which may improve the accuracy of bug localization. However, collecting proper execution trace depends on the quality of the test suits. Besides this, dynamic analysis fails to take the contents of classes or methods, as it only provides call sequences.

2.2 General Procedures of Software Bug Localization

In software bug localization, valid information are retrieved from the source code and bug report. After retrieving text, those are needed to be processed.

These text are compared with each other and based on these the similarity is measured.

2.2.1 Text Retrieval Based Bug Localization

Text retrieval based bug localization represents a technique by which the text are retrieved from the source code and bug report. These techniques are widely used to localize software bugs. During bug localization, developers often formulate queries using the bug report information. The implicit assumption is that the bug descriptions and the relevant buggy source code files share some important keywords [1]. Figure 2.9 also refer this situation where both bug report and source code share some common terms. In this section, the relationships between the software bug report and source code have been described by Moreno et al.

Moreno et al. conducts an empirical study that explores the usual relationship between the bug report and source code [1]. Here, authors find that the bug reports share more terms with the patched classes than with the other classes in the system. Furthermore, it is also visualized that the class names are more likely to share some terms with the bug descriptions than other code locations. According to their findings, the shared terms may be the better predictors for finding buggy locations.

To find more accurate relation between source code and bug report, authors use bugs which are described in [44], [45]. The authors also use six open source projects such as:

- ADempiere[3]
- Art of Illusion[4]
- aTunes[5]
- Eclipse[6] (version 2.0)
- Eclipse (version 3.5)
- JEdit[7]

Authors identify the relationships between large bug report with the unique terms using Spearman coefficient. In their experiments, they find that Spearman coefficient is 0.96 which represents that with the increment of the volume of a bug report, the number of unique terms have been increased. Also Mann-Whitney test [46] is conducted to find the number of average shared terms between bug report and patched classes. In this case, authors also find that on an average, bug reports' terms are stayed in patched classes at a large rate as shown in Figure 2.11.

This information motivates to propose text retrieval based bug localization techniques.

In software bug localization, source code and bug report textual similarity has been measured. To do so, bug report and source code are processed so

[3]Details About Adempiere- http://adempiere.org/
[4]http://www.artofillusion.org/downloads
[5]http://www.atunes.org/
[6]http://archive.eclipse.org/eclipse/downloads/
[7]http://jedit.org/index.php?page=download

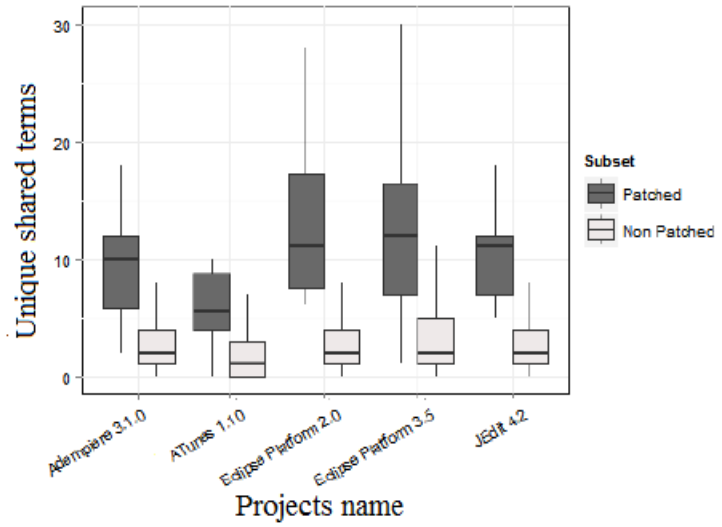

Figure 2.11: Shared terms (%) between the patched classes and bug report [1]

that the maximum similarity can be achieved. Generally, four phases are followed in bug localization and those are

- Code Corpora Generation,
- Query Construction,
- Indexing,
- Ranking and Retrieval.

Each of the above described phases have also series of steps. The details of each step is given below.

2.2.2 Code Corpora Creation

Code corpora are the bag of words generated from the source code. In this step, lexical analysis has been performed for each source code file and pre-

pares a vector of lexical tokens [11]. In this steps, several kinds of pre-processing techniques have been performed and those are:

- Stop words removal,
- Programming language specific keywords removal,
- Multi words identifiers,
- Stemming.

Each step of the pre-processing is described in the followings.

Stop words removal

Stop words usually refer to the most commonly used words in a language which do not provide any scenario specific information. Rather, these stop words increase unnecessary words which may affect the ranking significantly. So, these words should be removed to support phrase search. As bug localization is a textual similarity based approach, initially stop words are removed.

"a", "a's", "able", "about", "above", "according", "accordingly", "across", "actually" are the examples of few stop words. The details are described in Appendix A.

Programming language specific keywords removal

When developers write a program, a large amount of programming language specific keywords such as public, static, private, etc are commonly used.

These keywords have been used to support the program in that programming language. Among this keywords, type declaration plays a vital role. In type declaration, the types of variables are defined using multiple keywords (for example, int, string, char). Many other keywords are also used as programming language keywords, for example, “public”, “static”, “void”, “abstract”, “continue”, “for”, “new”, “switch”, “assert”, “default”, “goto”, etc. Details are described in Appendix A.

Since these programming language specific keywords are only used to run the program, these information do not provide any bug specific information. As a result, the consideration of these programming language specific keywords provide irrelevant information for a bug. Hence, these keywords are removed from the search space.

Multi words identifiers

At the time of software development, a large number of variables are defined by the concatenation of multiple words. For example, a variable name is “startDate” is composed of two words and those are ‘start’ and ‘Date’. So, this composite token should be split into multiple individual tokens to increase the scenario specific keywords.

To identify multi words, the convention of writing variable names have been considered. For example, in object oriented programming language, conjugate variable names are defined using lower Camel case (for example, isCompatible). So, in this case, using the lower Camel case identification process, words are split. Besides this, Multiwords may be created using

multiple separators. For example, the words may be conjugated by (dot, hyphen, underscore, Camel case, etc). Hence, based on this properties, words are split into multiple keywords. As a result, the volume of accurate keywords are increased.

Stemming

The last commonly used pre-processing step is stemming. It is used in linguistic morphology and information retrieval to extract the base or root form of the words. The main goal of stemming is to reduce inflectional forms of a word. For instance,

give, *given*, *giving* $\Rightarrow$ *give*

am, *is*, *are* $\Rightarrow$ *be*

Stemming usually refers to a heuristic process that chops off the ends of words to extract the root word, and often removes the derivational affixes. However, here two types of errors may be occurred such as understemming and overstemming. Understemming fails to produce same stem for all the words in the same concept. For example, "alumnus" $\rightarrow$ "alumnu", "alumni" $\rightarrow$ "alumni". As a result, number of returned relevant results of a search may be failed. On the other hand, overstemming occurs when the stemmer provides same stem for multiple different concepts. For example, the stem of 'business' may be 'busy' and 'university' may be 'universe' which is not desired [47].

To mitigate this problem, several kinds of stemming are available and those are Lovins stemmer [48], Porter stemmer [49], Paice stemmer [50].

Among these, Porter's algorithm is used commonly for its popularity. Porter stemming is consisted of five different phases of word reductions. In each phase, several rules are followed to reduce the words' extensions. In the first phase, following rules have been applied.

Rules:

SSES → SS

IES → I

SS → SS

S →

Based on this above described rules, the words extensions have been deducted. For example, if a word contains *SSES* at the last of that word such as *caresses*. After applying Porter stemming, the word *caresses* will be converted into *caress*.

2.2.3 Indexing

After the generation of code corpus, these corpus are needed to be indexed. In the phase of Indexing, code corpus are stored within that source code class and method respectively. It preserves the structure of the source code. Indexing helps to easily identify the actual buggy classes or methods of the source code. Several kinds of indexing are already established and those are

- Lucene Indexing
- Latent Semantic Indexing

The details of each indexing are described in the followings.

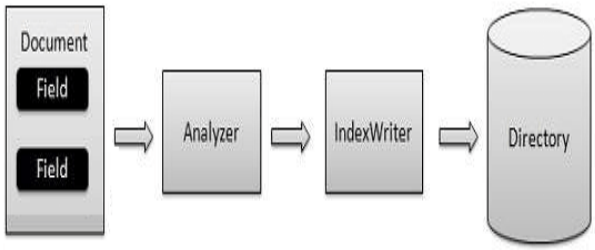

Figure 2.12: Steps of Lucen indexing

Lucen Indexing

Apache Lucene is a free open-source information retrieval based software library which is written in Java. Lucene achieves fast search responses because it searches an index instead of searching the text directly. Here, inverted index is used, because multiple documents may be indexed against a key. So, by only considering keyword instead of traversing the whole book, the desired results may be produced.

Figure 2.12 illustrates the indexing process. At first, documents are given with multiple fields. To create a document, key value pairs have been generated containing keys as names and values as contents to be indexed. The fields are analyzed by Analyzer. After analyzing the documents, IndexWriter creates index or update the index. Among this four steps, Indexwriter plays a vital role in the process of indexing.

Latent Semantic Indexing

Latent Semantic Indexing (LSI) is an indexing and retrieval technique where Singular Value Decomposition (SVD) is used to identify the relationships

between the terms and concepts contained in an unstructured documents [51]. LSI focuses on the similar meanings of the words that are used in the same contexts. The main advantages of LSI is its ability to extract the conceptual contents of a document by developing a relationship between those terms that occur in similar contexts [52]. LSI considers both synonymous and polysemous words. To apply LSI, following steps are considered.

- Each document is converted into a vector of words. Among the words, stop words have been removed. The number of vector dimensions should be equal to the number of unique words.
- The frequency of each term has been measured. Noteworthy that, the terms which have similar meanings are also considered during the measurement of frequency.
- Term-document matrix is developed where the frequency of each term within a document is stored.
- Then Singular Value Decomposition (SVD) is applied. Set all but the k highest singular values to 0.
- Recombine the terms to form the original matrix.
- Break this reduced rank term-document matrix back into column vectors and this is the latent semantic index.

2.2.4 Query Construction

Bug localization considers bug report for constructing the query by which the source code is queried and buggy locations are identified. This query is needed to find the similarity with the source code. Proper query construction leads the achievement of accuracy in bug localization. For constructing a query for a bug, the bug report is processed using text processing techniques such as stop words removal, programming language specific keywords removal, multi-words identifiers, stemming, etc.

After applying this text processing techniques, a list of corpus has been generated which are used as the query for bug localization. These generate a bag of words which are known as the bug corpus.

2.2.5 Ranking and Retrieval

Ranking and Retrieval is used for measuring the similarity between the source code corpora and bug corpora generated from the query construction phase. Several kinds of ranking algorithms are available such as Vector Space Model (VSM), Latent Semantics Analysis (LSA), Latent Dirichlet Allocation (LDA), etc. VSM is widely used for retrieval and ranking methods. As VSM is widely used, the details of VSM is described below.

Vector Space Model

Vector space model is an algebraic model used for representing text documents as vectors of identifiers, such as index terms. Let, d and q are the

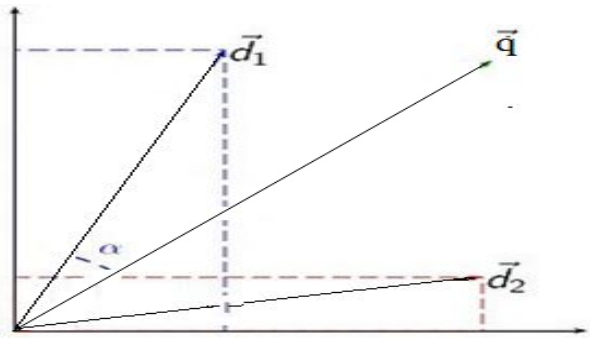

Figure 2.13: Vector Space Model Representation for documents and queries

document and query vectors respectively. The similarity between these two documents (that is, d and q) can be measured by the cosine similarity of the VSM where the angle represents the similarity between the vectors (that is, d and q). Small angle refers that the document and query are very similar, while large angle between source code and query represents that the source code has a very few similarity with the bug report. Equation 2.1 presents the cosine similarity between the bug report and source code.

$$cos\theta = \frac{\overrightarrow{d} \, . \, \overrightarrow{q}}{\mid \overrightarrow{d} \mid\mid \overrightarrow{q} \mid} \tag{2.1}$$

Here, $\overrightarrow{d} \, . \, \overrightarrow{q}$ is the intersection of the document (that is, source code) and query (termed as, bug report) vectors. $\mid \overrightarrow{d} \mid$ and $\mid \overrightarrow{q} \mid$ is the norm of vector d and q respectively.

Figure 2.13 presents a vector representation of the query and document. If a document is more relevant against a query, the angle between the query and

document will be very small. The larger the angle is, the large dissimilarity exists between these.

Latent Dirichlet Allocation

Latent Dirichlet allocation (LDA) is a generative statistical model which observes set of data to explain why some parts of the data are similar. LDA is commonly used in natural language processing to find the text similarity where each document is a combination of a small number of topics and each word is related to one of the document's topics if observations are conducted in the collection of documents. LDA is an example of a topic model and was first presented by David Blei et al. [53].

Each document of LDA is commonly viewed as a mixture of various topics. As an example, two topics of LDA model may be "Man_related" and "Bird_related". A topic has probabilities of generating multiple word, such as mankind, house and human which can be interpreted as "Man_related". Naturally, the word cat itself will have high probability given a topic. The "Bird_related" topic has probabilities of generating each word such as feathers and wing. Figure 2.14 presents an LDA model where D denotes the whole document and T presents the topics of the document which is in between the collection of text documents. W refers to the collection of words where under each topic, multiple words may be available. α and β are the parameters used for representing per-document topic distribution and per-topic word distribution.

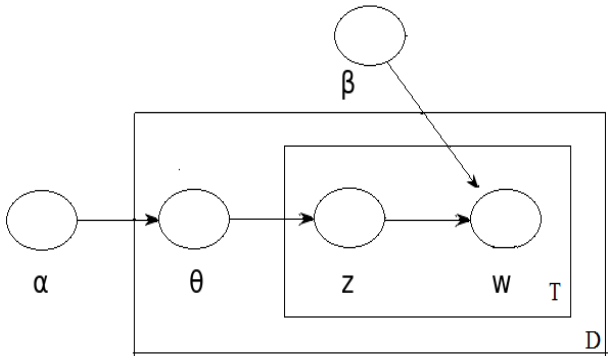

Figure 2.14: The representation of Latent Dirichlet Allocation (LDA)

The above described words are used here to measure the similarity between the newly reported bug and previously defined collection of words. As a result, based on this similarity score, a list of buggy locations can be identified.

2.3 Summary

In this chapter, the basics of software bug localization is discussed in a nutshell. To localize software bugs, the types of program analysis is also described. Not only that, but also the general procedures of bug localization, text processing, indexing, retrieval and ranking are also highlighted. In principle, these highlighted topics will be really insightful to understand the proposed method level and statement level bug localization. The next chapter holds some of the existing automatic software bug localization techniques which improves the accuracy.

Chapter 3

Literature Review

Automatic software bug localization is a process by which buggy locations of the source code can be identified. This automated suggestion of buggy locations can minimize the bug fixing time. As a result, the maintenance effort of the software can be reduced at a large rate. Existing bug localization techniques commonly identify buggy locations at the class level or method level. Hence, this chapter focuses on recent researches which are conducted to localize buggy locations at class or method levels.

In the previous chapter, the background knowledge of bug localization, the overview of software bugs and the general procedures of bug localization techniques have been studied. In this chapter, the existing research are described which are conducted in automated bug localization at class or method level. The target of such research is to identify buggy locations as more granular level (that is, class, methods, statements of the source code) as possible, so this chapter is organized according to the types of granularity of bug local-

ization. At first class level and then method level bug localization techniques are described. Finally, this chapter concludes with the limitations of existing bug localization techniques along with the necessary recommendations.

3.1 Existing Research Work

Manual bug localization is a tedious, time consuming and costly task, so automated bug localization is a prominent research area nowadays. The reduction of maintenance time depends on the suggested levels of buggy locations. If the suggestion occurs at the class level, developers have to manually traverse the whole class of the source code to identify the buggy statements. Similarly, the suggestion of method level bug localization also demands to traverse that method until the actual buggy statements are found. In the literature, only class and method level bug localizations are available, so the following discussion separates bug localization techniques into two groups and those are:

- Class Level Bug Localization,
- Method Level Bug Localization.

The accuracy of suggesting buggy locations depends on the size of the relevant search space and this can be identified by the source code analysis. The above described levels of bug localization can be divided using the types of source code analysis. Generally, source code is analyzed by two methods:

- Static Analysis,

- Dynamic Analysis.

The following discussion holds the details of existing bug localization techniques with their limitations.

3.2 Class Level Bug Localization

Class level bug localization refers to classes as the unit of suggesting buggy locations. This represents that for identifying a bug, a list of classes are suggested which may contain the bug. These types of localization provides developers a large problematic solution search space where a bug may stay. All the suggestions of buggy locations are based on the source code analysis such as source code static analysis and dynamic analysis. Below discussion first holds bug localization with source code static analysis and then some of the dynamic analysis based class level bug localization techniques are presented.

3.2.1 Source Code Static Analysis Based Techniques

Most of the class level bug localization techniques use static analysis of the source code. It refers to the analysis of the source code without executing the programs. This static analysis helps to understand the code structure. Several kinds of static source code analyzers are available such as Fortify

Software[1], GrammaTech[2], FindBugs[3], Jtest[4], etc. Using these tools, static analysis of the source code can be performed which can help to identify the buggy locations.

Zhou et al. propose BugLocator where buggy locations are identified at the class level using static analysis of the source code [11]. To identify classes as buggy, similarity between the source code class and bug report is measured using Information Retrieval (IR) techniques. At first authors process bug report and source code separately, resulting two sets of corpora, one for bug report and another for source code. These corpora are processed using several text processing techniques such as stop words removal, programming language specific keywords removal, multi-words identification and stemming. Later, these two sets of corpora are compared by proposing a revised Vector Space Model (rVSM). It modifies the existing VSM by introducing *tf-idf* formulation as well as giving emphasis on the length of the source code files. Authors find that a file containing large number of statements is more probable to be buggy than other small sized files. Based on this $rVSM$ score, similarity between each source code file and bug report is measured.

The similarity score is also between the newly reported bug and previously fixed bug information. The previously fixed files are collected from version history. Version history is commonly used in class level bug localization which

[1]Fortify Software- http://www.fortifysoftware.com/
[2]Details of GrammaTech Software- http://www.grammatech.com/
[3]FindBugs Software - http://findbugs.sourceforge.net/
[4]Jtest Software- https://www.parasoft.com/product/jtest/

stores historical information including previous bug reports, corresponding patched files, re-opened bugs, date of bug fixing, name of the developers who fix the bug, etc. of a project. If any similarity exists between the newly reported bug and previously fixed bugs, the files which are fixed previously will get more priority to be buggy for the newly generated bugs. This score was named as SimiScore as shown in Equation 3.1. Here, B presents the bug report, P_i refers to the previously fixed bugs that are similar to B. n is the total number of files connected to fix bug P_i.

$$SimiScore = \sum_{All\ P_i} (Similarity(B, P_i)/n) \tag{3.1}$$

The final score of each file is measured with a weighted sum of rVSM score and SimiScore according to Equation 3.2. Here, α is a weighting function which is experimentally set and the authors show that for $\alpha = 0.2$ and $\alpha = 0.3$, BugLocator produces best result.

$$FinalScore = (1 - \alpha) \times rVSMScore + \alpha \times SimiScore \tag{3.2}$$

In their experiments, four open source projects have been considered such as Eclipse (version 3.1), SWT (version 3.1), AspectJ and ZXing. When MRR was considered as a measurement metric, for Eclipse, SWT, AspectJ and ZXing, 41%, 53%, 41% and 50% accuracies are achieved respectively. On the other hand, when MAP was used 30%, 45% 22% and 44% accuracies have been achieved for Eclipse, SWT, AspectJ and ZXing respectively.

Here, authors do not assign special weights on structural information (termed as, class names, method names, variable names and comments), though it is needed due to containing exact structural information in the bug report. BugLocator does not consider semantic meanings of the words. As a result, exact word matching between the source code and bug report may not provide accurate results because one word may have multiple synonyms. For a specific bug, this approach suggests a list of class level buggy locations where developers need to manually investigate the whole source code for finding buggy locations in more granular levels (for example, buggy methods or statements of the source code). Since the whole source code is considered during static analysis, accuracy may be hampered. This is because bug localization follows a probabilistic way, so large unnecessary information creates more biasness in the ranking.

An improved version of BugLocator [11] is addressed by Ripon et al. where the assignment of special weights is performed on structural information including class names, method names, variable names and comments of the source code [27]. This technique is named as BLUiR. At first, BLUiR takes source code files, then Abstract Syntax Tree (AST) is developed where the structurally organized source code information is extracted. By traversing AST, different program structures such as class names, method names, variable names, comments are taken into consideration. All identifiers and comments are tokenized using text processing approaches such as stop words removal, multi-words identifiers and stemming on the source code. It is note-

worthy that, authors do not remove programming language specific keywords such as private, public, static, void, etc. This is because according to [27], some keywords like *String* or *Class* may be used as identifiers and the bug report may also contains these keywords. However, most of the cases the consideration of these programming language specific keywords may generate irrelevant search space.

Using the extracted structural information, a separate search into the bug report is performed, and then the total scores across all the searches are combined with different weights. In their experiments, similar to Zhou et al. [11], four open source projects have been considered such as Eclipse (version 3.1), SWT (version 3.1), AspectJ and ZXing. For MRR, BLUiR achieves 44%, 66%, 43% and 49% accuracies for Eclipse (version 3.1), SWT (version 3.1), AspectJ and ZXing respectively. When MAP was an evaluation metric, 33%, 58%, 25% and 39% accuracies are gained for Eclipse, SWT, AspectJ and ZXing projects accordingly.

Although authors impose different weights for different types of words, due to including programming language specific keywords into the code corpora, the accuracy may be degraded. For example, if a bug report contains "In dashboard the **public** data cannot be viewed" and if a source code contains a large number of occurrences of **public** word as programming language keyword, the results may be adversely affected due to including this **public** keyword. Here, authors consider files as buggy where developers have to spend lots of time for finding more specific buggy locations (that is, source

code buggy methods, statements). Due to considering whole source code information using static analysis of the source code, the accuracy of this technique may be biased because, bug localization depends on the volume of search space.

Later, another class level bug localization technique is proposed by Tantithamthavorn et al. where co-change histories of each fixed bug are considered instead of the consideration of previous bug history of a software [54]. Here, an intrinsic assumption is that when a bug is fixed, a set of classes are changed altogether. Three steps are considered to rank a list of buggy classes and those are:

- Co-change score calculation,
- Create list of co-change files,
- Adjust result from BugLocator.

In the phase of co-change score calculation, how many times two files are changed altogether are considered. To do so, a co-change matrix (C) is developed as shown in Figure 3.1. Here, $C_{(i,j)}$ represents the score of the concurrent modification of two files (that is, i and j). For example, the value of $C_{(i,j)} = 0.75$ represents that the file i and j are changed concurrently in 75% times.

Large number of changes holds high score for those files to be buggy. On the other hand, a file with rare changes has minimum score. As a consequence, that file has minimum probability to be buggy. After finding the

$$
C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{ccccc} 1 & 2 & 3 & \dots & c_n \end{array} \\
\begin{pmatrix}
0 & 0.75 & 0.87 & 0.12 & 0.26 \\
0.75 & 0.11 & 0.05 & 0 & 0.01 \\
0.87 & 0.05 & 0 & 0.10 & 0.21 \\
0.12 & 0 & 0.10 & 0 & 0.15 \\
0.26 & 0.01 & 0.21 & 0.15 & 0.1
\end{pmatrix}
\end{array}
$$

Figure 3.1: Co-change Matrix

co-change score for each file, these results are adjusted with BugLocator. Finally, a ranked list of buggy files is returned to the developers.

Here, authors conduct their experiments on two open source software (OSS) datasets, Eclipse SWT (version 3.1) and Android ZXing projects. The experimental results show that on an average 21.44% accuracy improvement has been performed than BugLocator.

Sisman and Kak propose another IR based bug localization technique by incorporating version histories [55]. For finding bug proneness, associated with the files, prior probability has been measured using version histories. This prior probability has been used to measure the posterior probability of a file to be buggy. Two models such as Modification History based Prior (MHbP) and Defect History based Prior (DHbP) are used for measuring the prior probabilities. In MHbP, the frequencies of file modification have been considered. The Maximum Likelihood (ML) can be expressed using Equation 3.3 where f and C_k represent the source code file and a collection of files respectively.

$$PMHBP(f \mid C_k, R_k) = \frac{\sum_{i=1}^{k} I_m(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} I_m(f', r_i)} \tag{3.3}$$

Here, I_m is an indicator that becomes 1 if r_i implements a bug fix which is measured using Equation 3.4.

$$I_m(f, r_i) = \begin{cases} 1, & f \in r_i \\ 0, & otherwise \end{cases} \tag{3.4}$$

Similar to Equation 3.3, defect probability of DHbP is also measured according to Equation 3.5.

$$DHBP(f \mid C_k, R_k) = \frac{\sum_{i=1}^{k} I_x(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} I_x(f', r_i)} \tag{3.5}$$

Along with this, a time factor is added with the calculation of priors. Here, the time duration between the submission of bug report and fixing is measured. This is needed because the time factor states which files are changed recently and based on this information, the buggy files can easily be predicted.

The previously measured priors are further used to determine the posterior probability of a file. From that posterior probability, a list of classes are reported as buggy according to Equation 3.6. Here, B represents bug report and s denotes the source code file.

$$P(s \mid B, R_k, C_k) = \frac{P(B \mid s, C_k) P(f \mid R_k, C_k)}{P(B)} \tag{3.6}$$

Due to considering irrelevant source code classes into the search space by conducting source code static analysis, the accuracy may be deteriorated significantly.

Wang et al. introduce another bug localization technique, where similar bug reports, version history and structure of the source code are amalgamated [2]. Figure 3.2 represents the overall procedure of this technique. In Figure 3.2, *Version history component* holds the information of a project history, while *Similar report component* uses old reports in bug repository to rank the buggy files. *Structure component* represents the structure of the bug reports and source code files. Each component of this technique produces a suspicious score for each source code file. Using the scores of these three components, final score is measured for each class file.

Here, authors perform experiments on four open source projects such as AspectJ, Eclipse, SWT and ZXing. The results are compared with [55], and in terms of Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR), this technique (termed as, AmaLgam) achieves 33% and 45% accuracies respectively.

This technique also suggests file level buggy locations and so developers have to spend lots of searching time to identify more granular level buggy locations (for example, methods or statements of the source code). Along with this, the accuracy of this bug localization technique may be declined significantly because of the consideration of large and irrelevant information domain.

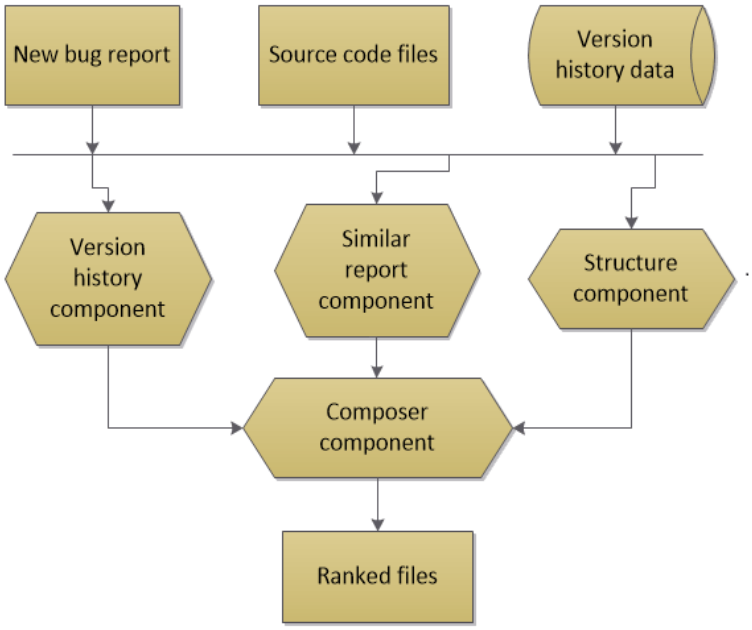

Figure 3.2: Overall Framework of Wang et al. [2]

Recently, Rahman et al. propose another bug localization technique which considers version histories and structural information of the source code to identify buggy files [56]. In that paper, authors identify suspicious score for each class by proposing three components and those are:

- Modified rVSM (MrVSM) component,
- Prioritization component,
- Combination component.

In MrVSM component, the scores of rVSM which is proposed in [11] are combined with the frequently changed file information. The impact of each

frequently changed file is shown in Equation 3.7.

$$f(\mathit{fixedfiles}) = \frac{1}{1 + \exp^{-Norm(\#\mathit{fixedfiles})}} \tag{3.7}$$

Here, $\#\mathit{fixedfiles}$ is the number of occurrences of a fixed file in the previous bug report and $Norm(\mathit{fixedfiles})$ is the normalized value of each fixed file. The final MrVSM score is measured using Equation 3.8.

$$MrVSM_{score} = rVSM_{score} + f(\mathit{fixedfiles}) \tag{3.8}$$

Meanwhile, the prioritization component prioritizes source code files whose structural information (such as class names, method names) are available in the bug report. So, at first, the bug report is processed using text processing techniques such as stop words removal, multi-words identification which generates a list of bug corpora. Afterwards, structural information are extracted from these generated bug corpora by applying some of the word-type identification rules. For example, words started with uppercase or upper camel case are considered as the class name, whereas a word started with a small case letter and having a '(' sign is considered temporarily as a method name. Then these extracted names getting from the bug corpora are compared with the actual class and method names of the source code. If the extracted terms getting from the bug report are matched with the source code class and method names, those classes will get more priority (β) than others. Equation 3.9 presents the score calculation for those files whose

structural information are available in the bug report.

$$FinalScore = MrVSM_{score} + \beta \tag{3.9}$$

The scores of these two components are merged in *Combination component* according to Equation 3.9 and based on that scores, a list of buggy files are suggested.

Three open source projects named as SWT (version 3.1), ZXing and Guava have been considered for the purpose of validation. The results demonstrate that 10%, 5% and 5% accuracy improvements have been achieved than BugLocator for SWT, ZXing and Guava projects respectively.

However, this technique is based on some predefined rules for providing emphasis on source code files. This technique also suggests a coarse level buggy locations (that is, class of source code) which demand more time to find the actual buggy statements. Along with this, the accuracy of the above mentioned techniques may be biased, due to using whole source code in static analysis.

All of the above described techniques suggest source code classes as buggy. Here, static analysis is applied where whole source code has been considered to compare the similarity between the source code and bug report. This may deteriorate the accuracy of bug localization because the measurement of accuracy depends on the volume of source code. As the above mentioned techniques suggest classes as buggy, manual interventions of developers are needed to find the actual buggy statements from the source code. This

manual approach may consume lots of developer efforts which may distract the motivation of automatic software bug localization technique.

3.2.2 Source Code Dynamic Analysis Based Techniques

In dynamic analysis of the source code, a lot of irrelevant information are discarded by only considering run time information of the program. Several kinds of dynamic analysis based techniques are available in automatic bug localization. Among this, machine learning based techniques are also used to identify the buggy locations.

In automatic software bug localization, supervised artificial intelligence based techniques are applied [57], [58]. Here, at first the program is learned using the training data and based on that learning, program behaves when a new scenario is emerged.

Kim et al. propose a machine learning based bug localization technique to suggest a list of buggy classes. Here, two phase recommendation models are proposed. In the first phase, this technique identifies whether the bug report is predictable or deficient using the volume of newly reported bug information [57]. If the first phase identifies the bug report as deficient, the second phase will not work, otherwise buggy files are recommended using static analysis of the source code.

The first phase basically filters out the bug report quality. To do so, the features are extracted from the bug report and using those information the

quality of the bug report is identified. If the bug report is predictable, the system is trained on using those information.

In the second phase, the probability of these predictable files are defined. Here, the bug report information are sent to a prediction model which recommends buggy files to be fixed. At first the model provides probability score for each file of the source code where Naive Bayes classifier is used as a prediction model [59], [60]. Finally, a list of buggy files are ranked according to the probability scores.

To compare the results, Likelihood, Precision and Recall are used as the measurement metrics. The results are compared with the One-phase, Usual suspect and BugScout [61]. The results show that the two phase recommendation technique performs better than the other three techniques in all the cases.

However, the produced results may be biased because the accuracy depends on the classifiers and training data. This technique also suggests classes as buggy which is a coarse level suggestion. As a result, developers have to traverse the source code file to find the buggy statements.

Another genetic algorithm based bug localization technique is addressed by Shaowei et al. which is worked in two steps such as training and deployment [58]. For the purpose of training, here 15 Vector Space Models (VSM) are taken as input along with the previous bug reports. Each VSM has a different *tf-idf* weighting scheme. Using this 15 VSMs and the bug report, a

near optimal composite model is developed that best localizes the bugs. The composite model is defined using Equation 3.10.

$$M_{composite}(b, f) = \sum_{i=1}^{15} w_i \times VSM_i(b, f) \tag{3.10}$$

Here, b and f are the bug report and source code files respectively. w_i is the weight of the ith VSM and $w_i \in [0, 1]$.

Using genetic algorithm, the target is to maximize the objective function (that is, accurate suggestion of buggy files). Generally, genetic algorithm starts with a list of population of chromosomes (that is, initial solutions) and continuously new populations are generated by performing crossover and mutation among the VSMs. Genes (that is, the value of some weights) are randomly swapped from one another. After applying mutation and crossover in multiple times, the best fitness function (that is, accurate similarity measurement function) can be gained.

Finally, in the deployment phase, the composite model which is learned in the training phase is used to localize a new bug. To find the buggy files, this approach assigns similarity scores to the source code files. These scores are used to localize buggy source code classes.

Three open source projects such as AspectJ, Eclipse and SWT are used for evaluating this bug localization technique. 43%, 39% and 63% accuracies are achieved for AspectJ, Eclipse and SWT respectively. Still the accuracy is questionable because in this case accuracy depends on the quality of the classifier and the training data. As this approach suggests classes as buggy,

it consumes lots of searching time for developers to find the actual buggy statements.

In this section, basically some of the machine learning based techniques are described because using source code execution trace buggy classes are not suggested yet. Hence, the accuracy depends on the quality of the classifiers and training data. A good classifier can distinguish multiple things in an accurate manner. Besides this, as classes are suggested as buggy, lots of time is consumed to find the buggy statements.

3.3 Method Level Bug Localization

Since the suggestion of more granular level (that is, methods and statements of the source code) bug localization technique is needed, few techniques are available to suggest methods as buggy. In method level bug localization, methods are identified as the unit of suggestions. Here, also the time consumption may be another important research issue, because a method also consists of numerous number of source code lines. A short description of the method level bug localization techniques are discussed below.

3.3.1 Source Code Static Analysis Based Techniques

Brent D. Nichols propose a method level bug localization technique [62] using source code static analysis. Here, three steps are followed to identify buggy methods and those are:

- Extracting semantic data from source code,
- Adding additional information from previous bugs,
- Querying LSI model.

At first, the semantic meanings of each method is extracted. To do so, several text processing techniques such as stop words removal, separation of identifiers and stemming are applied. This provides semantic meanings of each source code method.

In the second phase, authors add extra information from the previous bug history to the methods. It is noteworthy that the additional bug information are collected from the Bugzilla repository. From that, more information for each bug is taken by analyzing previously fixed bugs and their corresponding patched files where each bug of the patch is mapped to the affected methods.

Finally, when a new bug is arrived, Latent Semantic Indexing (LSI) is applied on the method documents to identify the relationships between the terms of the bug report and the concepts of the method documents. Based on that relationships, a list of buggy methods has been suggested.

Due to depending on the predefined dictionary keywords and inadequate previous bug reports, this greedy approach may fail. Furthermore, the accuracy of this technique may not be satisfactory enough due to considering the whole source code information rather than the bug specific information.

Lukins et al. propose another method level bug localization technique where both source code methods and comments are extracted [7]. Three steps are also performed here and those are:

- Build a document collection from the source code,
- Generate a Latent Dirichlet Allocation (LDA) model,
- Query the LDA Model.

In the step of building document collection from the source code, each method is considered as a document. For each method, authors generate a collection of terms. To do so, the source code is processed using stop words removal, programming language specific keywords removal, multi-words identifiers and stemming. After doing this, a document collection is prepared.

In the next phase, Latent Dirichlet Allocation (LDA) model is generated. To perform an LDA analysis, GibbsLDA++ is used which is an open source software tool [63]. This takes the whole source code as the input. Using this tool, word-topic modeling and topic-document probability distribution are measured. The word-topic distribution refers the probability of a word belongs to a topic. Similarly, the topic-document distribution represents the probability that a given topic belongs to a document.

The final step is to query the LDA model. To do so, a query is generated from the bug report using previously used text pre-processing techniques such as stop words removal, multi-words identification, Porter stemming, etc. Then using these query, LDA model is searched to find the list of buggy methods.

Here, three case studies have been performed on Rhino, Eclipse and Mozilla open source projects. Authors show that in Rhino 23% and 63% bugs are located at the top 1 and top 5 rank respectively.

By addressing multiple levels of abstraction, Duy et al. [64] propose a method level bug localization technique. At first, method corpus of the source code has been prepared by applying multiple pre-processing techniques such as Java keywords removal, stop words removal, Camel case identification, Porter stemming, etc. Here, documents are represented in multiple levels considering topic modeling. To model the topics, authors use a popular topic modeling algorithm named as Latent Dirichlet Allocation (LDA) [53].

Topic modeling maps words to topics by assigning each word to one topic. The fewer the number of topics, the higher the abstraction level, while the larger the number of topics, the lower the number of abstraction levels. Thus topic modeling has been iteratively applied to create multiple abstraction levels. Finally, these abstractions are aggregated to measure the similarity between a bug report and a code unit.

Alhindawi et al. [3] introduce another method level feature location based technique by enhancing source code with stereotypes. Stereotypes represent the details of each word which is commonly used in programming. For example, the stereotype named as 'get' means that a method returns a value. Similarly the stereotype 'set' represents that the value of a data member has been set. In this approach, the corpus of the source code is enriched with the combination of stereotypes which describes the abstract role of a method or a function of source code. Figure 3.3 presents some of the stereotype information getting from the source code. These stereotype information are derived automatically from the source code via program analysis. After

Stereotype Category	Stereotype	Description
Structural ***Accessor***	get	Returns a data member.
	predicate	Returns Boolean value which is not a data member.
	property	Returns information about data members.
	void-accessor	Returns information through a parameter.
Structural ***Mutator***	set	Sets a data member.
	command	Performs a complex change to the object's state *(this)*.
	non-void-command	
Degenerate	incidental	Does not read/change the object's state.
	empty	Has no statements.

Figure 3.3: Some stereotypes of source code method used in [3]

adding stereotype information with the source code methods, Information Retrieval (IR) based technique is used to run quires for feature location.

The basic assumption is that adding stereotype information to the source code corpus will improve the results of bug localization. However, the reason behind improving the results using stereotypes is unclear.

This section contains some of the method level bug localization techniques where whole source code is considered during static analysis. Although these techniques provide comparatively fine granularity than class level suggestion, these also demand manual searching into the source code methods to find the actual buggy statements. As a result, time consumption is increased. On the other hand, due to considering whole source code by conducting static analysis, the accuracy may be degraded.

3.3.2 Source Code Dynamic Analysis Based Techniques

In dynamic analysis based techniques, the run time behavior of the source code is obtained using proper test suits. Using source code dynamic analysis, data flow of the execution can be recorded. Using dynamic analysis, the irrelevant source code can be discarded.

The first dynamic analysis based bug localization technique is proposed by Wilde et al. where source code execution traces are analyzed using multiple test cases [65]. Authors consider two types of test cases such as the passing test case and failing test case. Using the passing test cases, desired features are extracted. Similarly, failing test case provides the features which is not desirable. To identify the buggy locations, these two types of test cases are considered which provides a large volume of features.

However, due to using passing test cases, irrelevant features may be included in the domain of search space. As a result, accuracy of bug localization may be hampered.

Eisenbarth et al. propose an improved version of [65] where both dynamic and static analysis of the source code are combined [38]. Here, static analysis identifies the dependencies among the data to locate the features in a program while dynamic analysis collects the source code execution traces for a set of scenarios. These traces are analyzed with a view to categorizing the subroutines based on the degree to a feature. Here, formal concept analysis has been used to relate features altogether. This produces a set of concepts

for the program. However, because of using whole source code during static analysis, accuracy may be degraded.

One of the earliest method level bug localization technique is proposed by Poshyvanyk et al. which is named as PROMESIR. Here, authors use not only static, but also dynamic analysis of the source code. Through dynamic analysis, executed buggy methods are extracted for a bug. Initially, the two analysis techniques produce bug similarity scores differently without interacting with each other. Hence, PROMESIR measures similarity scores for each method separately in dynamic and static analysis. In dynamic analysis, scenario-based probabilistic ranking is performed and in static analysis, buggy methods are ranked using Latent Semantic Indexing (LSI). Finally, these two scores are combined and performed a weighted ranking score using Equation 3.11.

$$f_{PROMESIR}(m) = \lambda\, \beta_{spr} f_{spr}(m) + (1 - \lambda)\beta_{lsi} f_{lsi}(m) \tag{3.11}$$

In this equation, m denotes method and λ is a weighting factor where $\sum_{i=1}^{n} \lambda_i = 1$.

Three case studies are conducted to evaluate PROMISER. Two widely used open source projects such as Eclipse and Mozilla are considered as the subject of case studies. Although this technique uses dynamic information of the source code, it fails to minimize the solution search space. Rather the whole source code is considered during static analysis which may increase the biasness. As a consequence, the bug localization accuracy is deteriorated.

Moreover, this approach suggests methods as buggy which also demands developers investigation to find the exact buggy statements.

The bug localization techniques which are described in this section suggest methods as buggy followed by dynamic analysis of the source code. Here, some of the cases irrelevant information can be discarded by only performing method call graph which contains very few relevant information. On the other hand, some techniques combine both static and dynamic source code analysis based techniques where irrelevant source code is considered in the search space. As a result, the accuracy of bug localization techniques may be declined.

3.4 Summary

From the above discussions, it is clear that the existing bug localization techniques commonly followed static, dynamic or the combination of both analysis of the source code. Here, the accuracy depends on the removal of irrelevant source code by identifying valid information domain. Besides, existing approaches suggest either classes or methods as buggy which also demand manual inspections into the class files or methods to find more granular buggy locations such as buggy statements. As a result, the time of localizing bugs is increased which distracts the motivation of automatic software bug localization. Next chapter describes an appropriate method level bug localization technique by minimizing solution search space. Later, from

that minimized search space, a statement level bug localization technique is proposed.

Chapter 4

A Method Ranking Approach for Localizing Software Bugs

In automatic software bug localization, source code analysis is usually used to localize the buggy code without manual intervention. However, due to considering irrelevant source code, localization accuracy may get biased. In this chapter, for improving the accuracy, a Method level Bug localization using Minimized search space (MBuM) is proposed. To do so, at first this chapter introduces the necessity of search space minimization with the limitations of existing research work. Later, a formal proof and a new methodology of bug localization technique is given followed by a model. Experimental results provide two research questions which provide that the buggy methods must reside in the suggested list and the accuracy of this suggestions. Two case studies are used to answer the above research questions.

4.1 Introduction

In general, bug fixing is initiated when the Quality Assurance (QA) team or user reports against a faulty scenario. When developers receive bug reports, they try to find the buggy locations using those reports and source code information. Developers may inspect the source code by their experience or may debug the source code by using the debugger of an IDE. In the manual debugging approach, after executing each statement, developers have to inspect the result of that executed statement and make a decision either that statement is buggy or not. This approach consumes lots of time because a source code often contains millions of lines (for example, Eclipse 3.0.2 version contains 1,86,772 statements).

In case of automatic software bug localization, developers usually provide the bug reports and buggy projects to an automated tool, which provides a ranked list of buggy locations. The ranking provides the highest probable buggy locations at the top. After getting a list of buggy locations, developers traverse the list from the beginning until they find the actual one. If the automated tool uses whole source code to suggest buggy locations, localization accuracy may be questionable because bug localization is a probabilistic approach and accuracy depends on the size of the search space.

Automatic software bug localization is commonly performed using either static, dynamic or both analysis of the source code by which failure locations of a software can be identified [66], [67], [68], [39], [69]. Most of the bug localization techniques follow static analysis where program analysis and Infor-

mation Retrieval (IR) techniques are used [70], [71], [72], [27], [73], [58], [11]. These techniques provide less accurate results because as mentioned above static analysis uses probabilistic approach, so the more the unnecessary information is considered, the more the biasness is introduced. Additionally, it fails to identify runtime bugs due to considering only textual information of the source code.

Dynamic analysis based techniques analyze the execution trace of the source code with suitable test suites to identify the executed methods for a bug. As dynamic analysis only provides method call sequences except method contents, the solution search space becomes very small. And using this small search space, it is hard to find the actual buggy locations. Hence, the search space should be increased for IR based bug localization techniques.

Although a large number of literature addresses software bug localization with bug report, to the best of the authors knowledge, those do not specifically focus on the minimization of search space with valid information. In recent years, some researches on bug localization have been conducted using static analysis where buggy locations are identified using bug reports and source code [39], [62], [2], [74], [75]. In static analysis, authors create two lists of corpora from the source code and bug report. Then corpora are processed so that proper similarity between source code and bug report can be measured. Finally, IR based techniques are applied for ranking probable buggy files [27], [58], [11]. Zhou et al. [11] propose a static analysis based bug localization technique named as BugLocator where Vector Space Model (VSM) is

modified by proposing tf-idf formulation. As BugLocator only uses static information of the whole source code, this considers irrelevant information for a bug. An extended version of BugLocator is proposed by assigning special weights on structural information (for example, class names, method names, variable names, comments) of the source code which also ranks classes as buggy [27]. Similar to BugLocator, this technique also considers the whole information of the source code, as a consequence, biasness may be raised.

Lukins et al. rank methods as buggy where semantic and syntactic meanings of the source code are extracted [7]. Here, authors propose Latent Dirichlet Allocation (LDA) which provides word-topic modeling and topic-document distribution. These topics are used to measure the similarity between the bug report and methods of the source code. This technique also considers whole source code by static analysis. Several dynamic analysis based bug localization techniques have already been proposed to localize software bugs [38], [76]. Scenario-based Probabilistic Ranking (SPR) is a source code dynamic analysis based technique [77], [78], where the execution trace of the different scenarios (that is, classes and methods of the source code) are considered. Although this technique minimizes solution search space using only dynamic analysis, the volume of considered valid information is very few. As a result, accuracy may be deteriorated because they only use the executed method names to localize software bugs without taking any content of those methods. Poshyvanyk et al. propose PROMISER which suggests methods as buggy by combining both static and dynamic analysis of the source code [76].

Although this technique considers dynamic analysis of the source code, the whole source code is used in static analysis. Since most of the techniques suggest classes as buggy, developers need to manually investigate the source code class to determine more granular buggy locations (for example, methods of the source code). Besides, most of the aforementioned existing techniques consider the whole source code during static analysis, though few techniques incorporate dynamic analysis. As a result, bug localization accuracy may be biased by unnecessary information.

This chapter proposes a novel automatic software bug localization technique, namely Method level Bug localization using Minimized search space (MBuM) where buggy locations of the source code are identified by eliminating irrelevant search space. At first, MBuM identifies a relevant information domain by tracing the execution of the source code for a bug. As dynamic analysis provides a list of executed methods without method contents, static analysis is performed to extract the contents of those methods. After that, several pre-processing techniques are applied on these relevant source code along with the bug report, which produces code and bug corpora. During the creation of bug corpora, pre-processing techniques such as stop words removal, multiwords splitting, semantic meaning extraction and stemming are applied on the bug report. In addition to these, programming language specific keywords removal is applied for generating scenario specific code corpora. Finally, to rank the buggy methods, similarity scores are measured between the code corpora of the methods and bug corpora of the bug report

by applying mVSM. It modifies existing VSM to provide large sized methods more priority to be buggy.

The effects of search space on ranking is formulated as a proposition, which has been proved using formal methods. Besides this, to evaluate the results of the proposed solution, the experiments (that is, Section 4.5) contains three case studies where Eclipse and Mozilla are used as the subject of case studies. Results are compared with four existing bug localization techniques namely PROMISER [76], BugLocator [11], LDA [7] and LSI [79]. In Eclipse, MBuM ranks the actual buggy method at the first position in three (60%) among five bugs, while other techniques rank no more than one (20%) bug at the top. Similarly in case of Mozilla, LDA, LSI and BugLocator rank none of the bugs at the top position whereas MBuM ranks three (60%) and PROMISER ranks two (40%) bugs at the first position. Above results show that, MBuM performs better than other existing state-of-the-art bug localization techniques.

4.2 Model of Proposed System

In this section, a model of Method level Bug localization using Minimized search space (MBuM) has been developed. Here, at first the elements of the model are defined. Source code and bug report act as the input, while a ranked list of buggy methods is the output of the system. The input are processed using a bug localizer and the buggy methods are ranked. During the processing of input, it is assumed that a bug report and source code

share some common information. Since the size of the source code of a software is too large, it is quite difficult to find the actual buggy locations for a bug. The details of the model is described below using Z notations [80].

$$
\begin{array}{|l}
\hline
\textit{MBuM} \\
D : \textit{Dictionary} \\
M : \textit{List of methods} \\
B : \textit{Bug report} \\
S : \textit{whole source code} \\
b_i : \textit{terms of the bug report} \\
m : \textit{accurate set of buggy methods} \\
s_i : \textit{terms of the source code} \\
\hline
B \leftarrow b_i \mid b_i \in D \\
S \leftarrow s_i \mid s_i \in D \\
m \leftarrow \textit{find accurate relation between } b_i \bigwedge s_i \\
\hline
\end{array}
$$

Here, the main objective is to increase accurate number of buggy methods (m). To do so, an accurate relationship should be established between bug report and source code terms. As it is difficult to find exact buggy locations from the whole source code, domain extraction of the source code is needed as much as possible. (The reason for minimizing the domain of the source code is described in Section 4.3.) Moreover, to find the list of buggy methods, a good source code terms and bug report processing technique is needed.

4.3 Minimized Solution Search Space for Improving Bug Localization Accuracy

In this Section, the rational behind minimizing solution search space has been described which is one of the main objectives described in Section 4.2. A new proposition is also developed by which valid search space can be extracted. The proof of the proposed proposition is described in the followings using Z notations [80].

A small and relevant search space can increase the accuracy of bug localization because localization follows a probabilistic way and accuracy depends on the volume of search space. The relevant search space may be obtained by executing the source code for a specific bug. Since only the bug specific source code is considered, it ensures that the actual buggy methods must reside within the relevant extracted domain. After discarding the irrelevant source code, more accurate bug localization can be obtained. The reason behind this hypothesis is described in **Lemma 1** and **Lemma 2**.

Lemma 1: In bug localization, the selection of Relevant Domain data can produce more accurate ranking than considering the Whole-Domain.

For developing a software, a large number of source code files or classes are created. The number of selected source code class can significantly affect the ranking score of bug localization. To show the effects of the number of classes in case of accurate bug localization, a representative ranking model

namely Vector Space Model (VSM) can be used [81]. VSM depends on the inverse document frequency (idf) and idf also depends on the number of documents or source code files which is used to increase the weights of rare terms as Equation 4.1.

$$idf = \log(\frac{\sharp docs}{n_t}) \tag{4.1}$$

Here, $\sharp docs$ and n_t are the total number of documents and the number of documents containing the term t respectively. Equation 4.1 shows that idf increases with the increment of $\sharp docs$. The VSM depends on the idf and the final score of VSM is calculated using Equation 4.2.

$$\begin{aligned} VSM(q,d) = cos(q,d) = (\sum_{t \in q \cap d} (\log f_{tq} + 1) \times (\log f_{td} + 1) \times idf^2) \\ \times \frac{1}{\sqrt{\sum_{t \in q}((\log f_{tq} + 1) \times idf)^2}} \times \frac{1}{\sqrt{\sum_{t \in d}((\log f_{td} + 1) \times idf)^2}} \end{aligned} \tag{4.2}$$

In Equation 4.2, t, q and d represent the term, query and document respectively. f_{tq} and f_{td} are the term frequencies within the query and documents respectively. In this study, inverse method frequency (imf) has been used instead of idf to give more priority to rare terms in methods. So, the consideration of large number of irrelevant methods can deteriorate the ranking scores significantly. The effect of imf is illustrated by the following mathematical model when whole source code is considered for finding buggy locations.

RareTermPriority

$weight(x) : x\ gets\ weight$
$r? : Rare\ Term$
$b? : Bug\ Report$
$t? : Term$
$\upsilon : Buggy\ Methods$
$\psi : Non\ Buggy\ Methods$

$\exists\, t? \in b? : (t? \in r?) \wedge (t? \in \upsilon) \wedge (t? \notin \psi)$
$\bullet\ weight(\upsilon) \Rightarrow rank_{high}(\upsilon) \qquad (a)$
$\exists\, t? \in b? : (t? \in r?) \wedge (t? \notin \upsilon) \wedge (t? \in \psi)$
$\bullet\ weight(\psi) \Rightarrow rank_{high}(\psi) \qquad (b)$

The above model shows the impact of rare terms in bug localization. Here, two scenarios may be occurred such as $RareTermPriority(a)$ and $RareTermPriority(b)$ represent the weight of buggy and non-buggy methods respectively. The detail is described below.

- The methods which hold rare terms and related to a bug, get more priority to be buggy which is desired (shown in $RareTermPriority(a)$),
- Similarly, rare terms exist in methods which have no relation with the occurrence of reported bugs, get more *imf* weights which is repulsive (shown in $RareTermPriority(b)$). When the solution search space is large, the situation of $RareTermPriority(b)$ may be occurred which deteriorates the ranking accuracy significantly. So, if the total number of methods can be restricted by only considering valid and relevant methods, *imf* cannot create large negative impacts on the ranking.

Another problem may be occurred due to considering large solution search space and that is the actual buggy location may be suggested in the Tth position in the worst case where the total number of available methods are T. It is noteworthy that an automated bug localization technique provides a list of buggy locations according to the descending order of ranking scores, where developers traverse from the beginning of the suggested list one by one. If the actual buggy location is suggested in Tth position, the developers need to inspect T number of suggestions to find the actual one. On the other hand, if it can be ensured that the bug is obtained in the targeted domain (for example, $T - \psi$), developers have to inspect only $(T - \psi)$ buggy locations in the worst case. So, minimized solution search space is needed which can be determined using the following mathematical model named as *SearchSpaceMinimization*.

SearchSpaceMinimization

$b? : \mathit{Bug\ Report}$
$m_t? : \mathit{Total\ methods\ in\ source\ code}$
$m_d? : \mathit{Methods\ relevant\ to\ bug}$
$m_u? : \mathit{Methods\ irrelevant\ to\ bug}$
$\vartheta : \mathit{seq\ bugs}$
$\kappa! : \mathit{Minimized\ search\ space}$

$m_d? \in \mathbb{P}\, m_t?$
$m_d? \subseteq m_t?$
$m_d? = m_t? \setminus m_u?$
$m_t? = m_d? \cup m_u?$
$\forall\, b? \in \vartheta \bullet (b? \in m_d?) \wedge (b? \notin m_u?) \Rightarrow \kappa! = m_d?$ (c)

In *SearchSpaceMinimization* model, b is a bug report. m_t, m_d and m_u represent total, relevant and irrelevant methods of source code for a bug respectively. This model states that the relevant methods can be obtained by discarding the irrelevant methods from the source code. From this model, it is clear that the bug must reside into the dynamically traced methods m_d according to *SearchSpaceMinimization*(c).

Lemma 2: Large information domain can increase False Positive Rate of bug localization.

In case of software bug localization, False Positive Rate (FPR) means that the identification of non-buggy methods as buggy which misguides developers to identify buggy methods. The large unnecessary information can produce large FPR. The situation of the increment of FPR with respect to the unnecessary information is illustrated using the following mathematical model.

$$
\begin{array}{|l}
\hline
\textit{FPR} \\
Package : \{p_1, p_2, p_3, ..., p_n\} \\
p? : Package \\
m? : Module \\
b? : Bug\ report \\
Buggy(x) : x\ is\ buggy \\
pr_{rank}(x) : Ranking\ score\ of\ x \\
f(p?) : Feature\ of\ p? \\
Imp(p?) : Implementation\ of\ p? \\
\beta! = Buggy\ package \\
\hline
m? \in f(p_1) \cap f(p_2) \cap f(p_3) \\
b? \in m? \wedge b? \in p_1 \\
(f(p_1(m?)) = f(p_2(m?)) = f(p_3(m?))) \wedge \\
(Imp(p_1)! = Imp(p_2)! = Imp(p_3)) \Rightarrow \\
(pr_{rank}(p_2) \geq pr_{rank}(p_1)) \vee (pr_{rank}(p_3) \geq pr_{rank}(p_1)) \\
\beta! = p_2 \vee p_3 \\
\hline
\end{array}
$$

In this model, m is a module which is common within p_1, p_2 and p_3 packages of a project whose basic functionalities are same but implementations are different. b is a bug associated with m, which is actually related to package p_1. Due to obtaining the same feature in three different packages, it may happen that p_2 and p_3 get more ranking scores than p_1. This situation could be raised when the whole source codes have been considered to locate a single bug because the size (that is, terms) of p_2 and p_3 may be larger than p_1. And according to VSM theory, the small but relevant document will get more priority than others which leads to the following theorem.

Theorem 1: A small but relevant search space can increase the accuracy of bug localization.

Lemma 1 illustrates that *VSM* may incur negative impacts in case of large

solution search space and **Lemma 2** shows the effects of using whole source code in case of FPR. So from **Lemma 1** and **Lemma 2**, it can be derived that a small number of relevant search space can increase the accuracy of bug localization.

4.4 Methodology of Method level Bug localization using Minimized search space

In this section, a model described in Section 4.2 has been implemented by devising a methodology which increases the ranking accuracy because of considering only relevant search space from the source code for a bug. The methodology of proposed bug localization technique is described in the following subsections.

The overall process of improving the bug localization accuracy is briefly discussed as follows. From Section 4.3, it is observed that if relevant information domain can be extracted by ignoring irrelevant methods from the large solution space, the accuracy of bug localization can be increased dramatically. At the beginning of bug localization, source code dynamic analysis is performed to minimize the solution search space which extracts only the related methods for generating a specific bug. Static analysis is done with a view to getting the contents of extracted methods. After applying dynamic and static source code analysis, valid and relevant information can be obtained. The contents of extracted methods are processed to create code corpora using

static analysis. Since bug report only contains textual information related to a specific buggy scenario, again static analysis is performed to process the bug report. Finally, generated corpora from the bug report and source code are matched with each other and based on that, source code methods are ranked. The whole process for localizing bugs can be divided into four steps and those are:

- Code corpora creation,
- Indexing,
- Bug corpora creation,
- Retrieval and ranking.

Each of the steps follow series of tasks as shown in Figure 4.1.

4.4.1 Code Corpora Creation

Code corpora are the collection of source code words which are used to check the similarity with bug report corpora [2], [11], [27], [62]. So, the more accurate the code corpora generation is, the more accurate matching can be obtained which may increase the accuracy of bug localization. For generating valid code corpora, two approaches are conducted and those are dynamic and static analysis [57]. Dynamic analysis produces relevant search space by considering source code execution trace (for example, source code methods) for a specific bug by despising the codes which are not responsible for

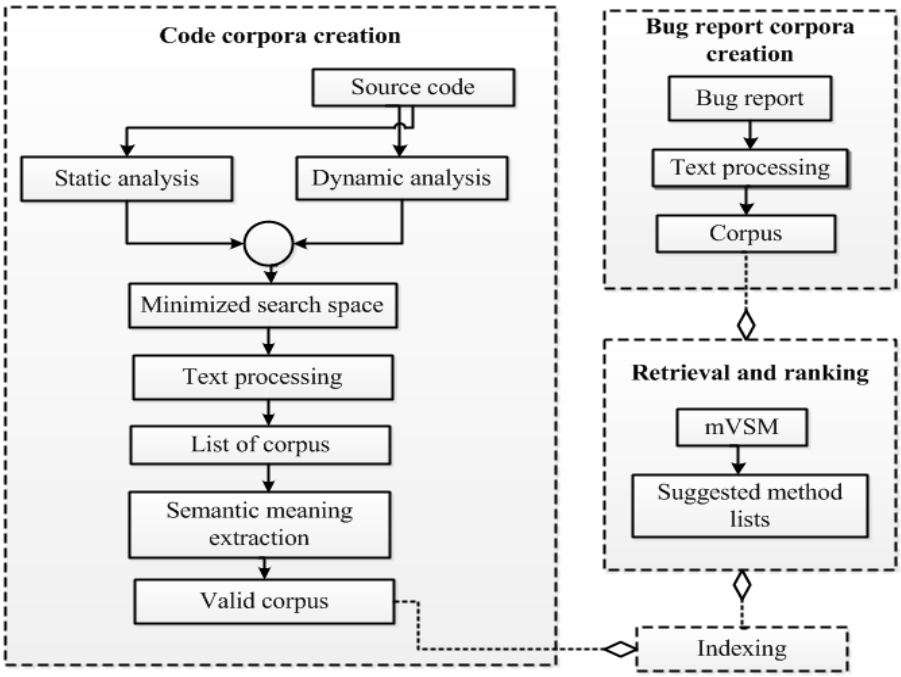

Figure 4.1: Functional block diagram of MBuM

generating the bug. Although dynamic analysis provides relationships between methods or classes, it cannot provide method or class contents. On the other hand, static analysis is related to the code analysis which considers whole source code information and extracts method contents. So, to get the method contents for only relevant methods, the output from dynamic and static analysis are combined and the common methods of those analysis are considered. The code corpora creation can be divided into multiple granular levels which are described below.

For the purpose of dynamic analysis, initially developers need to reproduce the bug after getting the information from the bug report's title, summary and description. When developers reproduce the bug, execution traces

are recorded and analyzed to extract executed methods. From these traces, method call graphs are generated and parsed to obtain the structure of the executed source code. It is noteworthy that the method call graph does not contain the method contents rather it stores the sequentially executed method names. Hence using static analysis, source code is parsed by maintaining the code structures. This is done by traversing the Abstract Syntax Tree (AST)[1] to extract different program structures such as package, class, method and variable names. This static analysis is conducive to dynamic analysis because from this, contents of dynamically traced methods are derived. In practice, irrelevant information deteriorates the accuracy as mentioned before and inspired from this motivation, the irrelevant information is eliminated from large search space by only considering relevant information.

Contents of the minimized search space are processed to get the relevant code corpora as shown in Figure 4.1. This is needed because buggy locations are identified by measuring similarity between the contents of bug report and minimized search space. So, trade-off needs to be done between bug report and source code contents such as the format of all words should be the same (for example, base form). Minimized source code are pre-processed because source code may contain lots of unnecessary keywords such as programming language specific keywords (for example, public, static, void, int, string, etc.), stop words (for example, has, is, a, the, etc.) which do not provide any bug

[1]A large lexical database of English, for details - https://eclipse.org/jdt/core/r2.0/dom%20ast/ast.html

specific information rather may create negative impacts on ranking and thus stop words are discarded from the source code corpora.

Within source code, one word may consist of multiple terms such as '*beginHeader*' consists of 'begin' and 'Header' terms. Therefore, multiword identifiers are also used for creating singular value decomposition. Porter Stemming [82] is applied to get the original form of the word so that 'searching', 'searched' and 'search' are identified as the same word. Moreover, statements are splitted based on some syntax specific separators such as '.', '=', '(', ')', '{', '}', ';', '/', etc. After completing all the aforementioned preprocessing, source code corpora are produced.

The last step for generating code corpora is semantic meaning extraction as shown in Figure 4.1. During this step, semantic information of each word is extracted because one word may have multiple synonyms. For example, to describe a single case, developers and QA teams often use different words. Although the semantic meanings of developers and QA described scenarios are the same, the only difference is in their vocabulary choice. Bug localization usually follows IR based techniques by performing word matching. So, the accuracy depends on the matching of the words. To match the words of bug report accurately with the source code words, semantic word matching is done. For example, 'close' word has many synonyms such as 'terminate', 'stop', etc. To describe a scenario if a developer uses 'close' and QA uses 'terminate', the system cannot identify the similar words without using se-

mantic meanings of those words. Thus semantic meaning extraction plays vital role in accurate ranking of buggy methods.

4.4.2 Indexing

Indexing helps to compare code corpora and bug corpora easily. After the completion of comparison, buggy locations can easily be found by only searching the indexes. Figure 4.2 is an example of a source code index (taken from, Eclipse project). Here, the packages are *org.eclipse.swt.graphics*, *org.eclipse.swt.custom* and *org.eclipse.swt.widget*. Among this, to better understand about the indexing only one package (that is, *org.eclipse.swt.graphics*) contents are expanded. At first, that package is defined and later the class name, method name of the source code are accumulated in Figure 4.2. Here, two classes such as Rectangle and Point are available in package *org.eclipse.swt.graphics*. Among these classes, two methods are stored. As this technique provides a method level ranking, the contents of each method is stored within the method. For each word of the method $< word >$ is generated.

In this step, the source code corpora as well as its semantic meanings are indexed where packages, classes, methods and method contents are stored according to the structural hierarchy of the program. Each method contains multiple words and each word within a method is stored sequentially.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Source_Code>
    <Package name="org.eclipse.swt.graphics">
        <Class name="Rectangle">
            <Method lineNo="66" name="Rectangle">
                <Word content="rectangl"/>
                <Word content="width"/>
                <Word content="width"/>
                <Word content="height"/>
                <Word content="height"/>
                <Word content="rectangl"/>
            </Method>
        </Class>
        <Class name="Point">
            <Method lineNo="53" name="Point">
                <Word content="point"/>
                <Word content="point"/>
            </Method>
        </Class>
    </Package>
    <Package name="org.eclipse.swt.custom">..
    <Package name="org.eclipse.swt.widgets">..
</Source_Code>
```

Figure 4.2: Example of source code indexing

4.4.3 Bug Report Corpora Creation

Software bug report contains the details of error of a program. The bug report is usually prepared by Quality Assurance (QA) team or users. A software bug report contains bug title, summary and description which provide important information about a bug. These information are used for localizing software bugs.

In bug report, the information may contain some common and irrelevant words such as stop words which do not provide any bug specific information rather create biasness for localizing the bugs. Moreover, words of bug report may be in present, past or future tenses. So, bug report needs to be processed to remove these noisy information. At the beginning of bug corpus generation, stop words (for example, am, is, are, etc.) are removed from the bug report. Then multiword splitting (if needed) and porter stemming [82] are applied (as used for code corpora generation) to get the base form of the words. After completing these pre-processing, valid bug corpora are generated which provide only the relevant words.

4.4.4 Retrieval and Ranking of Buggy Methods

At the beginning of this step, each bug corpus is searched in the minimized solution space. For ranking the source code methods, the proposed technique applies modified Vector Space Model ($mVSM$) [11]. $mVSM$ calculates the similarity between each query (bug corpora) and methods as the cosine similarity with their corresponding vector representations according to Equation 4.3.

$$Similarity(q, m) = cos(q, m) = \frac{\overrightarrow{V_q} \times \overrightarrow{V_m}}{\mid \overrightarrow{V_q} \mid \times \mid \overrightarrow{V_m} \mid} \tag{4.3}$$

Here, $\overrightarrow{V_q}$ and $\overrightarrow{V_m}$ are the vectors of terms for the query (q) and method (m) respectively. $\mid \overrightarrow{V_q} \mid \times \mid \overrightarrow{V_m} \mid$ is the inner product of two vectors. Term weight is calculated by multiplying *tf* (term frequency) and *imf* (inverse method frequncy). $mVSM$ uses the logarithm of term frequency of a method.

The *imf* ensures that rare or unique terms in the methods are given more importance. *tf* and *imf* are calculated using Equation 4.4 and Equation 4.5 respectively.

$$tf(t, m) = \log f_{tm} + 1 \tag{4.4}$$

$$imf = \log(\frac{\sharp methods}{n_t}) \tag{4.5}$$

f_{tm} is the number of occurrences of a term (t) in a method (m), $\sharp methods$ refers to the total number of methods in the minimized search space, and n_t refers to the total number of methods containing the term t. MBuM also considers method length because previous studies showed that larger files are more likely to contain bugs due to carrying many features of a software [11]. The function used to model the method length is provided in Equation 4.6.

$$g(terms) = \frac{1}{1 + e^{-Norm(\sharp terms)}} \tag{4.6}$$

Here, $\sharp terms$ is the number of terms in a method and $Norm(\sharp terms)$ is the normalized value of $\sharp terms$. The normalized value of a is calculated using Equation 4.7.

$$Norm(a) = \frac{a - a_{min}}{a_{max} - a_{min}} \tag{4.7}$$

Where, a_{max} and a_{min} are the maximum and minimum value of a. Now this normalized value is multiplied with the cosine similarity score to calculate final $mVSM$ score which is calculated by Equation 4.8.

$$mVSM(q, m) = g(terms) \times cos(q, m) \tag{4.8}$$

After measuring *mVSM* score of each method, a list of buggy methods has been ranked according to the descending order of scores. The method with maximum score is suggested at the top of the ranking.

4.5 Case Study

The effectiveness of MBuM has been evaluated by conducting three case studies. The case studies are similar with the existing bug localization techniques such as PROMISER [76], LSI [79] and LDA [7]. The comparative analysis with different bug localization techniques such as PROMISER, LSI, BugLocator and LDA shows that MBuM provides more accurate ranking than others.

4.5.1 Experimental Setup

The proposed solution has been developed in Java programming language. The equipment which are used to develop MBuM are as follows:

- Eclipse Luna (4.4.1): A Java Integrated Development Environment (IDE) used to develop MBuM,
- Windows 8.1, 4 GB RAM,
- 64 bit Operating System.

4.5.2 Objectives of the Case Studies

Since MBuM performs method level bug localization, methods are chosen as the level of granularity in all the case studies. Large and commonly used open-source projects such as Eclipse and Mozilla are taken into account to show the efficacy of MBuM. Documented bugs with corresponding published patches are used to evaluate the bug localization techniques where each patch specifies which methods were actually changed to fix a specific bug. The considered bugs are well-acquainted and reproducible which meet the following criteria.

- The bug reports do not contain method and class names directly to prevent the bias,
- The bugs having large similarity with multiple scenarios are chosen. For example, a similar feature may be implemented in multiple packages where the implementations are different in every packages. Here, a bug may be occurred in one package by not affecting other packages. This criterion supports basically **Lemma 2**,
- The bugs are chosen with approved and published patches.

4.5.3 Elements of the Case Studies

Here, two well-known open-source projects named as Eclipse and Mozilla are considered as the subject of case study. Eclipse is an Integrated Development Environment (IDE) which is widely used and for experiments, version 2.1.0,

2.1.3, 3.0.1, 3.0.2, and 3.1.1 are taken. The volume of each version is too large. For example, version 2.1.3 contains 7,648 classes with 89,341 methods [76]. Mozilla is another open-source web browser which can be used in most of the hardware and software platforms [76]. The volume of this project is also very large as well. For instance, version 1.5.1 contains 4853 classes and 53,617 methods. Here, Mozilla version 1.5.1, 1.6 and 1.6 (a) are used for experiments.

The actual buggy classes and methods corresponding to the bugs are identified from the published patches. The identified actual buggy methods are presented in Table 4.1 and Table 4.5 for Eclipse and Mozilla respectively. In case of more than one published patches for a bug, the union of the most recent and earlier patches are considered. A brief overview of bug title, description and the generated queries for Eclipse and Mozilla are provided in Table 4.2 and Table 4.6 accordingly.

4.5.4 Case Study: Searching from Eclipse Help

The main focus of this case study is to elaborate the effectiveness of the minimized search space for a specific bug. As shown in Table 4.2, bug report ♯74149 raised an issue about searching from 'Help' in Eclipse titled as "the search words after ' " ' will be ignored". For this case, the following scenario is executed to retrieve the relevant methods.

- Expand the 'Help' menu from Eclipse and click on the search option,

[1] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=74149`

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<source>
<Package name="org.eclipse.help.internal.search">
    <Class name="SearchProgressMonitor">
        <Method name="getProgressMonitor"/>
        <Method name="SearchProgressMonitor"/>
        <Method name="DummySearchQuery"/>
    </Class>
    <Class name="SearchManager">
        <Method name="SearchManager"/>
        <Method name="search"/>
        <Method name="getIndex"/>
        <Method name="getAnalyzer"/>
        <Method name="ensureIndexUpdated"/>
        <Method name="updateIndex"/>
    </Class>
    <Class name="SearchQuery">
        <Method name="getLocale"/>
        <Method name="getSearchWord"/>
        <Method name="getFieldNames"/>
        <Method name="isFieldSearch"/>
    </Class>
    <Class name="AnalyzerDescriptor">
        <Method name="AnalyzerDescriptor"/>
        <Method name="createAnalyzer"/>
        <Method name="getLang"/>
    </Class>
    <Class name="SearchIndexWithIndexingProgress">
        <Method name="SearchIndexWithIndexingProgress"/>
        <Method name="getProgressDistributor"/>
    </Class>
<Class name="SearchIndex">
    <Method name="SearchIndex"/>
    <Method name="tryLock"/>
    <Method name="needsUpdating"/>
    <Method name="getDocPlugins"/>
    <Method name="getIndexedDocs"/>
    <Method name="getTocManager"/>
    <Method name="addDocument"/>
    <Method name="beginAddBatch"/>
    <Method name="endAddBatch"/>
    <Method name="exists"/>
    <Method name="search"/>
    <Method name="registerSearch"/>
    <Method name="openSearcher"/>
</Class>
<Class name="HTMLDocParser">
    <Method name="HTMLDocParser"/>
</Class>
<Class name="ProgressDistributor">
    <Method name="ProgressDistributor"/>
    <Method name="addMonitor"/>
</Class>
 <Class name="IndexingOperation">
    <Method name="IndexingOperation"/>
    <Method name="execute"/>
    <Method name="getRemovedDocuments"/>
    <Method name="getIndexableURL"/>
    <Method name="getAddedDocuments"/>
    <Method name="getAllDocuments"/>
    <Method name="removeDocuments"/>
    <Method name="checkCancelled"/>
    <Method name="addDocuments"/>
</Class>
    <Class name="LazyProgressMonitor">
        <Method name="LazyProgressMonitor"/>
        <Method name="beginTask"/>
    </Class>
    <Class name="QueryBuilder">
        <Method name="QueryBuilder"/>
        <Method name="getLuceneQuery"/>
        <Method name="tokenizeUserQuery"/>
        <Method name="analyzeTokens"/>
        <Method name="buildLuceneQuery"/>
        <Method name="getLuceneQuery"/>
        <Method name="createLuceneQuery"/>
        <Method name="getRequiredQueries"/>
        <Method name="getRequiredQuery"/>
        <Method name="gethighlightTerms"/>
    </Class>
    <Class name="QueryWordsToken">
        <Method name="word"/>
        <Method name="phrase"/>
        <Method name="createLuceneQuery"/>
   </Class>
    <Class name="SearchResults">
        <Method name="addHits"/>
   </Class>
   <Class name="QueryWordsPhrase">
        <Method name="addWord"/>
   </Class>
</Package>
<Package name="org.eclipse.help.internal.base">
    ≈
</Package>
</source>
```

Figure 4.3: Relevant methods retrieved by dynamic execution tracing for bug ♯74149

- Enter a search query within the search field,
- Finally, click on 'Go' button or press enter.

In this case, MBuM finds only 20 classes and 100 methods as relevant to this bug, though Eclipse contains a large number of classes and methods. The retrieved methods using dynamic analysis are shown in Figure 4.3. Here, two source code packages namely *org.eclipse.help.internal.search* and *org.eclipse.help.internal.base* are executed. Within the first package, 14 classes are available and only 6 relevant classes are extracted from *org.eclipse.help.internal.base* package. Each extracted class also obtains one or more methods. For example, The class named as *SearchProgressMonitor*

Table 4.1: Actual buggy methods for chosen bugs in Eclipse

♯Bug	Methods
5138	org.eclipse.jdt.internal.ui.text.data.JavaString.DoubleClickSelector--doubleClicked org.eclipse.jdt.internal.ui.text.data.JavadocDouble.ClickStrategy--doubleClicked org.eclipse.jface.text.TextViewer.mouseDoubleClick org.eclipse.jface.text.TextViewer.mouseUp org.eclipse.jface.text.DefaultTextDoubleClick.Strategy.doubleClicked org.eclipse.jface.text.DefaultTextDoubleClick.Strategy.wordPosition
31779	org.eclipse.core.localstore.UnifiedTree.addChildren org.eclipse.core.localstore.UnifiedTree.addChildren.FromFileSystem org.eclipse.core.localstore.UnifiedTree.createChild.NodeFromFileSystem
74149	org.eclipse.help.internal.search. QueryBuilder.tokenizeUserQuery
83307	org.eclipse.ui.internal.WorkingSet.restoreWorkingSet
91047	org.eclipse.ui.internal.dialogs.AboutDialog.button.Pressed

contains three relevant methods for bug id-74149 whereas class *HTMLDocParser* extracts only one method named as *HTMLDocParser*. After despising the large irrelevant search space, static analysis is applied only on the relevant information (for example, Figure 4.3). If a query contains lots of ambiguous keywords, in that case MBuM may suggest the actual buggy method at most $100th$ position while all other existing bug localization techniques will suggest buggy method in $53,617th$ position in the worse case.

A query is formulated using the bug description which contained 'search query quote token'. The actual buggy method is manually retrieved from the

published patch which is 'org.eclipse.help.internal.search.QueryBuilder.tokenize UserQuery'. MBuM suggests 'tokenizeUserQuery' method at the 1st position of its ranking. Same query is applied on PROMISER, LSI and BugLocator to find the buggy location. PROMISER and LSI rank the actual buggy method at $5th$ and $8th$ position respectively. So, comparing with PROMISER and LSI, the effectiveness of MBuM is 5 and 8 times better respectively. On the other hand, BugLocator suggests buggy class at $12th$ position which shows that MBuM performs m times faster where m represents total number of methods in the suggested 12 buggy classes because BugLocator suggests buggy classes. LDA is not implemented because same bug reports are also used in LDA and that is why the results are taken from that paper [7]. In this case, LDA creates a different query as 'query quote token' which discards the 'search' term from the query due to obtaining 'search' query in multiple scenarios. That is the reason for suggesting the buggy method at the 1st position. Hence, it can be concluded that static analysis followed by dynamic execution trace of the source code reduces the search space which improves the ranking accuracy of bug localization technique.

4.5.5 Research Questions

In MBuM, probable buggy methods are ranked by conducting dynamic and static analysis of the source code. Hence, some research questions have been emerged such as **RQ1** and **RQ2**. **RQ1** is introduced to validate that the actual buggy methods reside in the minimized search space. It is also needed

to prove that the actual buggy methods get large similarity scores for a bug. To validate this, **RQ2** is introduced.

RQ1: Does the suggestion of buggy method list contain the actual buggy methods?

The algorithm of MBuM states that at first the solution search space is minimized by considering source code execution trace followed by static analysis. The contents of minimized search space and the bug report undergo some pre-processing steps. Finally, the similarity between the bug report and each method of the source code is measured. If a method is buggy against a scenario, some keywords should be shared between that method and the bug report. Due to obtaining shared terms, the similarity scores must be greater than zero which indicate that the buggy method is in the suggested list.

RQ2: How much effectively MBuM can suggest buggy methods?

The effectiveness of MBuM can be measured by considering the ranking of buggy methods. If the buggy method is ranked at the 1st position, the effectiveness is 100%. To answer **RQ2**, two case studies are conducted on Eclipse and Mozilla where the ranking of buggy methods provided by MBuM are compared with BugLocator, PROMISER, LSI and LDA. These case studies consider five different bugs which were also studied in PROMISER [76] and LDA [7].

Case Study 1: Bug Localization in Eclipse

In this case study, five different bugs in Eclipse are considered for making a comparison with state-of-the-art bug localization techniques named as Bu-

Table 4.2: Scenarios and queries used for ECLIPSE bugs

♯Bug	Bug Title	Bug description	Query
5138	Double-click-drag to select (multiple words doesn't work)	1. Click once and release the mouse button. 2. Click second time quickly and hold the mouse button down. Drag and select some text. 3. Release the mouse button. The text you have selected is deselected.	mouse double click up down drag release select offset document position
31779	UnifiedTree should ensure file/ folder exists	This problem appears to the user when executing refresh operations. At first refresh, the file and folder is located in the file system but second refresh removed the corresponding folder.	unified tree node file system folder location
74149	the search words after ‘ ” ’ will be ignored	User starts Eclipse, brings the help forward and performs a query leaving the last quoted text unclosed.	search query quoted token
83307	Unable to restore working set item	It looks like we are restoring a working set after a breakpoint is deleted.	restore work set item breakpoint
91047	About dialog buttons seemingly not responsive	Clicking on the ‘Plug-in Details’ or ‘Configuration Details’ buttons in the About dialog are time-consuming operations.	about dialog button press click

Table 4.3: The 1st relevant method against five bugs in Eclipse

♯Bug	Methods
5138	JavaStringDoubleClickSelector.doubleClicked
31779	UnifiedTree.addChildrenFromFileSystem
74149	QueryBuilder.tokenizeUserQuery
83307	WorkingSet.restoreWorkingSet
91047	AboutDialog.buttonPressed

Table 4.4: The suggestion of buggy methods using different bug localization techniques in Eclipse

♯Bug	BugLocator	PROMISER	LSI	LDA	MBuM
5138	7	2	7	2	1
31779	4	1	2	2	1
74149	12	5	8	1	1
83307	6	5	13	7	2
91047	4	6	9	5	3

gLocator, LDA, PROMISER and LSI. The chosen bugs are described as follows.

- Bug ♯74149[2], titled as "The search words after ' " ' will be ignored", exists in the versions 3.0.0, 3.0.1, 3.0.2, and fixed in the version 3.1.1,
- Bug ♯5138[3], titled as "Double-click-drag to select multiple words doesn't work", exists in version 2.1.3 and fixed in the version 3.3,

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=74149
[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=5138

Table 4.5: Actual buggy methods for chosen bugs in Mozilla

♯Bug	Methods
182192	nsAbAddressCollecter::CollectAddress nsAbAddressCollecter::CollectUnicodeAddress
216154	nsMailboxService::NewURI nslmapService::NewURI nsSmtpService::NewURI
225243	nsPostScriptObj::begin_page nsPostScriptObj::begin_document nsPostScriptObj:: setcolor nsPostScriptObj::colorimage nsPostScriptObj::grayimage nsRenderingContextPS::Init nsRenderingContextPS::DrawScaledlmage, nsRenderingContextPS::LockDrowingSurface
209430	nsPlaintextEditor:DeleteSelection
231474	Root::MimeObject_parse_begin

- Bug ♯31779[4], titled as "UnifiedTree should ensure file/ folder exists", presents in version 2.0.0 and fixed in the version 2.1.0,
- Bug ♯83307[5], titled as "Unable to restore working set item", presents in version 3.1.0 and fixed in the version 3.4,
- Bug ♯91047[6], titled as "About dialog buttons seemingly not responsive", exists in version 3.1.0 and fixed in the version 3.4.

[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=31779
[5] https://bugs.eclipse.org/bugs/show_bug.cgi?id=83307
[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=91047

Table 4.2 presents the generated queries from the bug reports such as bug title and description for each bug. From this table, it is clear that bug title contains a very few information about a bug. In most of the cases, the title of the bug contains 6-8 words on an average while bug description contains more information about that bug. The column named as 'Query' of Table 4.2 shows the bug corpus generated from MBuM. Table 4.3 provides the name of the first relevant method against each bug. Table 4.4 presents the ranking of the aforementioned bug localization techniques. It is noteworthy that, BugLocator did not suggest methods rather suggest files or classes. Figure 4.4 represents the ranking provided by different techniques for five different considered bugs in Eclipse. LDA's result is taken from the result provided in their paper [7].

These results show that MBuM ranks the actual buggy methods at the 1st position for three (60%) of the five bugs. Table 4.4 and Figure 4.4 present that for bugs ♯5138, ♯83307 and ♯91047 MBuM performs better than four other techniques. For bug ♯31779, PROMISER only provides equal result as MBuM. In case of bug ♯74149, although LDA ranks equal as MBuM, the term 'search' is omitted from the query which helps for providing better ranking because 'search' is a common term in Eclipse and might create biasness. However, it is not desired because 'search' may be a good candidate for finding buggy locations. From this case study, it can be concluded that static analysis of source code followed by dynamic execution trace of buggy scenario minimizes solution search space which further improves the accuracy

of ranking. Another case study described in the following also validates these findings as well.

Case Study 2: Bug Localization in Mozilla

Similar to the previous case study, here also five mostly used bugs are taken from the Mozilla bug repository and the selected bugs are described as follows.

- Bug ♯182192[7], titled as "quotes (') are not removed from collected e-mail addresses", presents in Mozilla version 1.6 and fixed in the version 1.7,
- Bug ♯216154[8], titled as "Anchors in e-mails are broken - clicking anchor doesn't go to target in an email", exists in version 1.5.1 and patched in the version 1.6,
- Bug ♯225243[9], titled as "Page appears reversed (mirrored) when printed", exists in the version 1.6 (a) and fixed in the version 1.7. This bug does not exist in version the 1.6 rather actually presents in the version 1.6(a) [7],
- Bug ♯209430[10], titled as "Ctrl+Delete and Ctrl+BackSpace delete words in the wrong direction", located in the version 1.5.1 and fixed in the version 1.6,

[7] https://bugzilla.mozilla.org/long_list.cgi?buglist=182192
[8] https://bugzilla.mozilla.org/show_bug.cgi?id=216154
[9] https://bugzilla.mozilla.org/show_bug.cgi?id=225243
[10] https://bugzilla.mozilla.org/show_bug.cgi?id=209430

Table 4.6: Scenarios and queries used for Mozilla bugs

♯Bug	Bug Title	Bug Description	Query
182192	Remove quotes (”) from collected addresses	A user replies to an e-mail. Then, using the mouse, the user, collects the e-mail address of the sender.	collect sender recipient email name address address-book
216154	Anchors in e-mails are broken (Clicking Anchor doesn’t go to target in e-mail)	A user receives an e-mail with internal, anchors and clicks on one anchor to access some text below in the document.	mailbox uri url msg pop3 msgurl service
225243	Page appears reversed (mirrored) when printed	A user generates a postscript file by printing a page containing images.	print page file document image postscript
209430	Ctrl+Delete and Ctrl + BackSpace delete words in the wrong direction	A user accesses the web page associated with a bug in Bugzilla and delete words.	delete word web page action back space
231474	attachments mix contents	A user sends a sample e-mail with many attachments from Bugzilla gets this e-mail using the Mozilla mail reader.	mail attachment encode content send mime

Table 4.7: The 1st relevant methods in Mozilla

♯Bug	Methods
182192	nsAbAddressCollecter::CollectAddress
216154	nsMailboxService::NewURI
225243	nsPostScriptObj::begin_page
209430	nsPlaintextEditor:DeleteSelection
231474	Root::MimeObject_parse_begin

Table 4.8: The suggestion of buggy methods using different bug localization techniques in Mozilla

♯Bug	BugLocator [11]	PROMISER [76]	LSI [79]	LDA [7]	MBuM
182192	4	2	37	3	1
216154	7	6	56	4	2
225243	5	6	24	9	2
209430	6	1	49	9	1
231474	3	1	18	4	1

- Bug ♯231474[11], titled as "Attachments mix contents", presents in the version 1.5.1 and fixed in the version 1.6.

Table 4.6 holds the considered bugs' title, description and quires and Table 4.7 retains the name of the first relevant method. The results demonstrate that MBuM provides better ranking accuracy over BugLocator, LDA, PROMISER and LSI techniques (shown in Table 4.8). These results show that three (60%) out of five bugs are located at the 1st position and another two are ranked at the 2nd position by MBuM. On the other hand, among the other four techniques only PROMISER suggests two (40%) of five bugs at

[11] `https://bugzilla.mozilla.org/show_bug.cgi?id=231474`

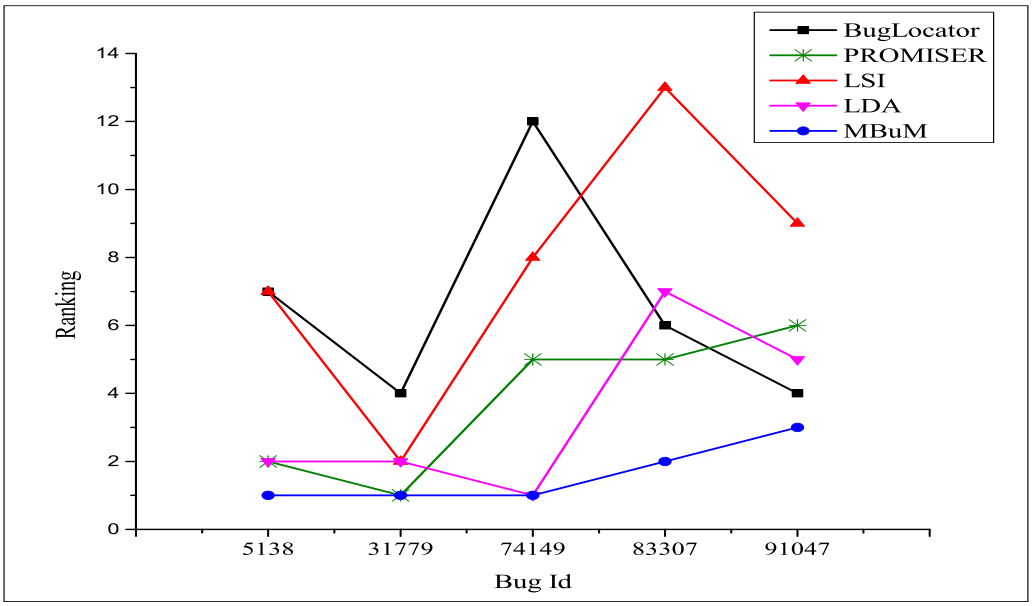

Figure 4.4: Ranking provided by different bug localization techniques in Eclipse

the 1st position and other three technique's results are far away from the 1st position (according to Table 4.8). Figure 4.5 presents the ranking provided by different techniques for five different bugs in Mozilla. Although for bugs ♯209430 and ♯231474, PROMISER provides the same ranking as MBuM, it produces noticeably poor ranking in other three bugs as shown in Table 4.8 and Figure 4.5. In case of ♯182192, ♯216154 and ♯225243, MBuM ranks the actual buggy methods more accurately than other four techniques. This comparative analysis of results also shows the significant improvement of ranking by MBuM.

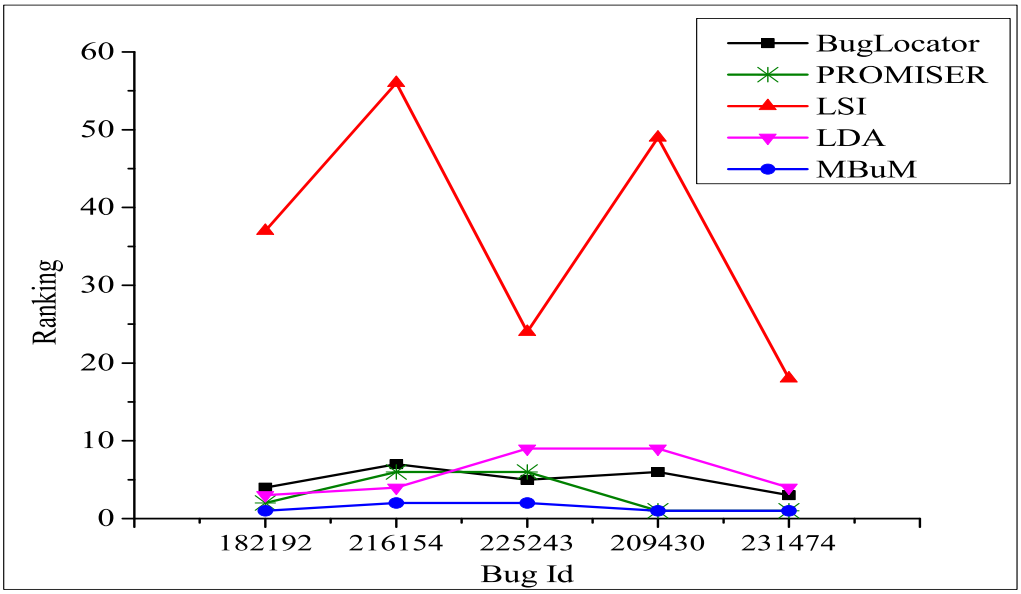

Figure 4.5: Ranking provided by different bug localization techniques in Mozilla

4.6 Discussion

Although the proposed MBuM performs better than the existing bug localization techniques, still the improvement of accuracy may be affected by several hindrances such as source code and bug report quality [83]. Hence, this section describes possible obstacles in bug localization, and presents a comparative result analysis of different techniques with MBuM.

Different issues may affect the results of the case studies such as the quality of queries may vary with the skills of the QA. Besides this, the selected scenarios could be the "best" or "worst" case, because no Eclipse or Mozilla source code experts were involved.

The bug reports which were considered by other techniques for measuring their efficacy, has also been used by MBuM. BugLocator [11], PROMISER [76], LSI [79] and LDA [7] consider the whole source code for finding a specific bug. As a result, the source codes which are not related to a specific bug are unnecessarily included and created biased impacts on methods ranking. Besides this, the existing bug localization techniques may increase the false positive rates severely. Due to extracting irrelevant information, MBuM provides more consistent and accurate results in bug localization. Table 4.4 and Table 4.8 also proves the justification of the proposed MBuM technique.

4.7 Summary

This chapter presents an approach to rank Method level Bug localization using Minimized search space (MBuM). For ranking buggy methods, it discards irrelevant search space by taking the execution trace considering method call sequences of the source code. To retrieve the content of the methods static analysis has been performed. Finally, similarity is measured between the method contents of the source code and bug report which provides a rank list of the methods. Still, the improvement of this technique is possible to suggest more granular level buggy locations. Hence, the next chapter describes statement level bug localization technique using this ranked list of methods.

Chapter 5

A Statement Level Bug Localization Technique

This chapter presents a Statement level Bug Localization (SBL) technique where valid information of each statement is maximized by combining each statement with its connected statements. Additionally, Information Retrieval (IR) based techniques are used to measure the similarity between the bug report and source code. For a better understanding, this chapter first introduces the necessity of SBL by focusing on the limitations of existing bug localization techniques and a brief overview of SBL. Later, the detail methodology of the proposed SBL is discussed. Result analysis is presented by considering two research questions which discuss about how many number of bugs (%) are located at the statement level and whether the consideration of connected statements improves the bug localization accuracy or not. Finally, this chapter concludes its contribution with future remarks.

5.1 Introduction

In automatic software bug localization, finding bugs in more granular levels (that is, class, method or statement of source code) are needed because it reduces software cost by minimizing maintenance effort. On top of this, many bug localization techniques have already been proposed to rank buggy classes [11], [27] or methods [76], [7]. The suggestion of buggy class provides a large problematic solution search space where a bug can stay. Among the suggested class list, it is almost impossible to find buggy statements within a short time because a class is consisted of numerous statements. As an example, if 10 classes are suggested as buggy while each class contains on an average 700 statements, and the actual buggy statement is located in the $5th$ class of its suggested list, for finding actual buggy statement, around $700 \times 5 = 3500$ source code statements should be traversed.

Although suggesting buggy methods are better than suggesting buggy classes, this suggestion also hardly reduces the time consumption because a method may contain large number of statements. As a result, the motivation of automatic bug localization has been distracted by suggesting list of classes or methods.

Fault localization is closely related to the bug localization, however the main difference is, fault localization does not consider bug report whereas bug localization does [42], [11]. Most of the cases, fault localization is performed before the delivery of the software to the users [84], [85]. In real life projects, user or Quality Assurance (QA) team reports against a faulty

scenario, and the bug is fixed using that report. Although several researches address spectrum based statement level fault localization [86], [87], [42], to the best of the knowledge, still no research has been conducted to suggest buggy statements using bug report.

From the above discussions, it is clear that statement level bug localization is a necessary demand. However, there may exist several kinds of limitations such as the availability of large irrelevant information and a low volume of valid information of a statement. Generally, a software project often contains a large number of statements having massive amount of unnecessary information for a bug. As an example, the Bug Id- 75739 of Eclipse is related to *org.eclipse.swt.ole.win*32.*Variant.java* file. Meanwhile, the total number of files in Eclipse is 86206 and except *org.eclipse.swt.ole.win*32.*Variant.java*, other files are irrelevant for this bug. As bug localization using bug report follows probabilistic approach, the consideration of unnecessary information can mislead to rank the actual buggy statements. Therefore, it is needed to discard irrelevant source code files as much as possible. On the other hand, a single statement often contains very few information about a bug. Only using these few information, suggesting a buggy statement is another challenging issue.

Although statement level bug localization is not available, suggesting buggy classes and methods are common in the literature [88], [11], [76]. Zhou et al. [11] propose a class level bug localization technique by only considering static analysis of the source code. Here, previously fixed bugs information

are taken to suggest buggy classes. However, they do not consider any structural information (for example, class names, method names, variable names, comments) of the source code which may improve the accuracy. To mitigate this problem, Saha et al. introduced an extended version of BugLocator by assigning special weights on structural information of the source code [27]. Similar to BugLocator, this technique also suggest classes as buggy which also demands manual inspection into the source code files to find buggy statements.

Lukins et al. rank buggy methods where semantic and syntactic meanings of the source code are extracted [7]. These information are used to measure the similarity between the bug report and methods of the source code. Poshyvanyk et al. propose PROMISER which also suggests methods as buggy by combining both static and dynamic analysis of the source code [76]. Rahman et al. introduce MBuM where irrelevant search space has been discarded by considering execution trace of the source code methods [16]. However, this technique also predicts list of buggy methods. Since no one suggests statements as buggy, developers need to manually investigate the source code to find more granular buggy locations (that is, statements of the source code).

In this chapter, a novel Statement level Bug Localization (SBL) technique is proposed where statements are suggested as buggy. To do so, search space minimization, method statements dependency graph and Information Retrieval (IR) techniques are considered. SBL processes bug report and source code to measure the similarity using IR techniques. SBL suggests statements

as buggy by only considering the source code methods which are suggested in Chapter 4. These methods are considered as minimized search space for finding buggy statements. For each suggested method, relationships have been developed among the statements which is named as Method Statement Dependency Graph (MSDG), and later a Node Predecessor-node Dependency Graph (NPDG) is developed where contents of each node (that is, statements of the source code) and its predecessor nodes of MSDG are combined. This combination of each node with its predecessors increase the valid information of each statement. Later, the similarity between the bug report and each node of NPDG is measured using Vector Space Model (VSM). These statement similarity scores are weighted by the corresponding method similarity score. Finally, a list of buggy statements are suggested using the descending order of the similarity scores.

The effectiveness of SBL has been measured using three benchmark projects named as Eclipse, SWT and PasswordProtector. The experimental section (that is, Section 5.3) contains three benchmark metrics such as Top N Rank, Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP). In all the projects, SBL ranks more than 28% buggy statements at the top 1 and more than 58% at top 5. In case of MAP, 38.5%, 44.2% and 61% accuracy are gained in Eclipse, SWT and PasswordProtector respectively. Besides this, for MRR, 45.8%, 51.3% and 66% accuracy have been achieved in Eclipse, SWT and PasswordProtector respectively. All of the aforementioned results justify the assumptions that SBL made.

5.2 Methodology of Statement Level Bug Localization

This section proposes a statement level bug localization technique namely SBL which is consisted of two major phases such as irrelevant search space minimization but maximizing the relevant data domain, and the ranking of buggy statements from that relevant data domain. Since Chapter 4 provides better ranking of buggy methods than other existing bug localization techniques, that suggested list of buggy methods is used here as minimized search space. Then static analysis is applied on these probable suggested buggy methods to find the actual buggy statements.

For each method, a graph has been generated where each statement acts as a node of the graph and the edge between nodes represents the callee (predecessor) - called (successor) relationship between the statements. This graph is named after Method Statement Dependency Graph (MSDG). For each node, a super node is generated by combining the node and its predecessors, because the result of each statement may depend on its predecessor nodes by which valid information for a statement can be maximized. This graph is mentioned as a Node Predecessor-node Dependency Graph (NPDG). To suggest a list of buggy statements, the similarity score for each statement should be defined. Hence, Node similarity score (N_s) is measured between each node of NPDG and the bug report using Vector Space Model. For each node, N_s score is weighted by the score of the method (M_s) (as mentioned

in Chapter 4) which contains the node. All these steps are described in the followings.

5.2.1 Generation of Method Statement Dependency Graph (MSDG)

The main motivation of SBL is to increase the valid information of each statement. In source code, statements are related with one another and defects propagate from one statement to another. Therefore, a relationship can be established among the statements which can increase the valid information of each statement of the source code. As Chapter 4 provides a list of buggy methods of the source code, here those methods are used to establish the relationships among the statements of each method. To represent the relationships among the statements in a method, a Method Statement Dependency Graph (MSDG) is constructed.

MSDG is developed because one statement may have relationships with other statements (for example, $x = 5;\ \ y = x + z;$ here x and y have a relationship and x is a predecessor of y), and it is very often, defects propagate from one statement to another (for example, one variable generates null value and when the next variable uses that value, a null pointer exception is rendered). Usually, the error propagation is forwarded [89, 90]. Since error propagation is forwarded, the statement which is executed first may affect the statements which use the result of that statement.

```
1  package bank;
2  public class IncomeProfit {
3  double profit= 0.0;
4  public void calculateProfit(int sell1, int buy1){
5        int sell=sell1;
6        int buy=buy1;
7        double operationCost=calculateOperationCost();
8        profit+=sell-buy-operationCost;
9        if(profit>0.0){
10             System.out.println("Yes you gain profit");
11       }
12       else{
13             System.out.println("Sorry you lose");
14       }
15 }
16 private double calculateOperationCost() {
17       double electricityCost=50;
18       double waterCost=10;
19       double gasCost=15;
20       double cost=electricityCost+waterCost+gasCost;
21       return cost;
22 }
```

Figure 5.1: Source code

For a better understanding regarding the generation of MSDG, a *Java* source code class is considered as shown in Figure 5.1. This is a banking software where profit is calculated. Here, within the class of *IncomeProfit*, two methods namely *calculateProfit* and *calculateOperationCost* are available. Both *calculateProfit* and *calculateOperationCost* introduce some local variables, global variables and method call variables. For example, the method *calculateProfit* contains sell, buy, etc. as local variables whereas profit is a global variable. Except this, *calculateProfit* also depends on the result of *calculateOperationCost* method. So, *operationCost* at line 7 is a method call variable.

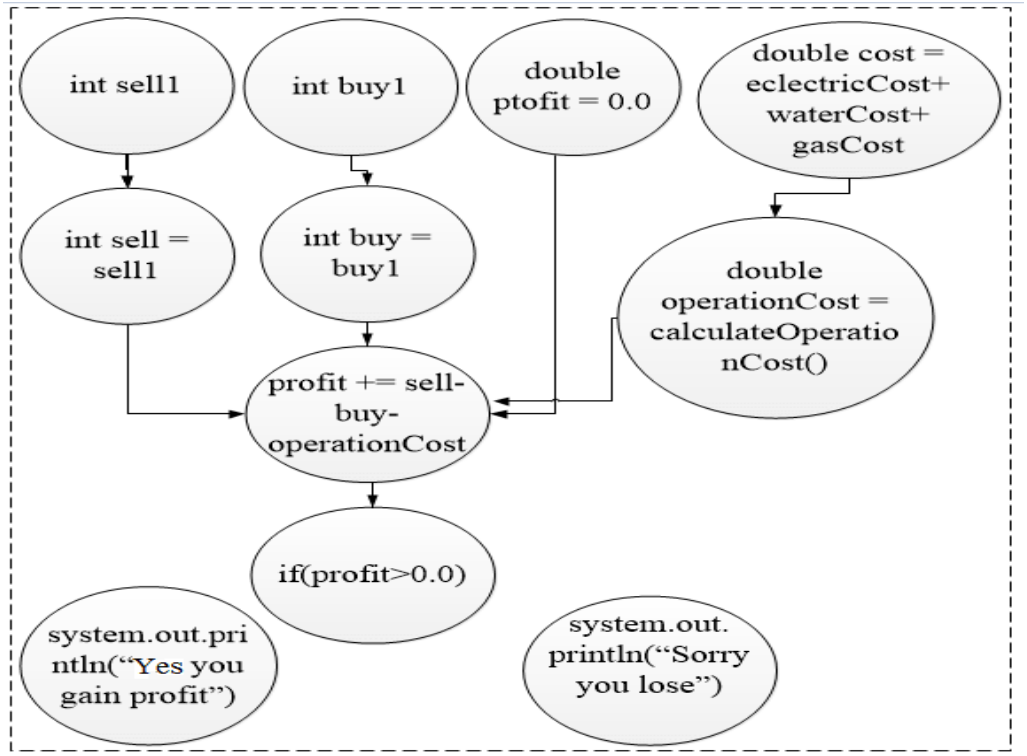

Figure 5.2: MSDG for *calculateProfit*

For a source code method *calculateProfit*, a MSDG is depicted in Figure 5.2 which is a directed graph. Let, $G = (V, E)$ is a graph where V is the set of vertices or statements of the method. Each vertex stores some properties of the source code (that is, package name, class name, method name, method score and line number of the statement). Figure 5.3 shows these properties of node $int\ sell = sell1$ of MSDG in Figure 5.2. E is the set of directed edges between statements which represents the data flow. If there exists an edge (v_1, v_2) from node v_1 to v_2, v_1 is said to be a predecessor of v_2. In source code, the predecessors can be defined as follows: if a statement contains an assignment such as $x = y + z$, the last modified statements of y and

z are the predecessors of x. In case of increment (x++), decrement (x- -) and conditional statements (for example, if ($x > 0$)), statement containing the last modification of x is a predecessor. This is needed with a view to maximizing the valid information of each statement which can improve the bug localization accuracy. Hence, to find the last modification of a variable, three cases are identified based on the types of predecessor variables such as Local Variable Usage (LVU), Global Variable Usage (GVU) and Method Return Variable Call (MRVC).

1. In LVU, the scopes of predecessor variables are bounded to the scope of that method. In *calculateProfit* method of Figure 5.1, *int sell1* affects *sell*, hence *int sell1* is the predecessor of *sell*. Similarly, *int buy1* is the predecessor of *buy* as *int buy1* affects *buy*. Likewise, in line 8, *sell*, *buy*, *operationCost* and *profit* are the predecessors of *profit*. As line 9 depends on the calculation of line 8, it is the predecessor of line 9. These predecessor identification can help to improve the valid information of each statement,

2. In GVU, the predecessor variable is globally declared and multiple methods may use that variable. The declaration of the *double profit* = 0.0 in line 3 affects line 8 where *profit* is calculated using *sell*, *buy*, *operationCost* and *profit*. Therefore, line 3 is the predecessor of line 8,

3. MRVC indicates that a statement calls a method that returns a value. In this case, the last modification of that called method's return statement is the predecessor of the current node. In Figure 5.1, line 7 shows

that *operationCost* calls a method (that is, *calculateOperationCost*), which returns *cost* variable. The last modification of *cost* variable is found at line 20. Therefore, line 20 is the predecessor of the calling node (that is, line 7), as shown in Figure 5.2.

From Figure 5.2, it is found that the last two nodes such as *Yes you gain profit* and *Sorry you lose* have no connection with other statements because these statements do not use any variables. As a result, the above mentioned three cases (that is, LVU, GVU and MRVC) do not satisfy.

Figure 5.3 presents an example with details of a node. Here, *int sell* = *sell*1 is a representative node which contains its package name, class name, method name where the statement is located in, method score, and line number of that statement. Similar to this node, all other nodes also contain this types of information.

5.2.2 Generation of Node Predecessor-node Dependency Graph (NPDG) for Similarity Score Calculation

In this phase, the information is maximized by combining each statement with its predecessors. Here, at first each node of MSDG containing a statement of the source code is processed. In this step, several kinds of preprocessing techniques have been applied. Stop words and programming language specific keywords are also removed because these keywords do not provide any bug specific information rather increase ambiguity and affects

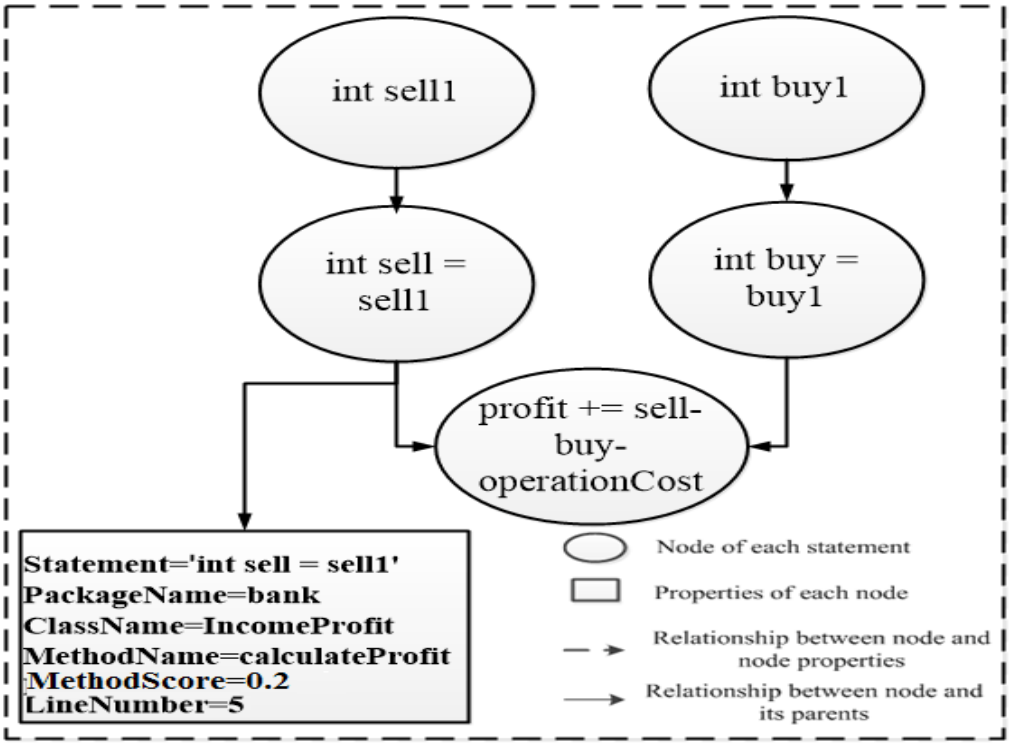

Figure 5.3: Representation of MSDG containing the details of a node

the accuracy of bug localization. Stop words (for example, am, is, are, etc) and programming language specific keywords such as int, HashMap, array, public, etc. are removed. Besides this, multi word identifications are also performed using multiple separators such as Camel case, underscore, dot, etc. These multi word identifications help to increase the valid information about a bug. For example, if a word is *passwordProtector*, using Camel case identification, two terms such as *password* and *protector* can be identified.

The remaining words are transformed into lower case so that all words are in the same format. The semantic meanings of each word are identified using WordNet to increase the valid information domain [91]. Here, the

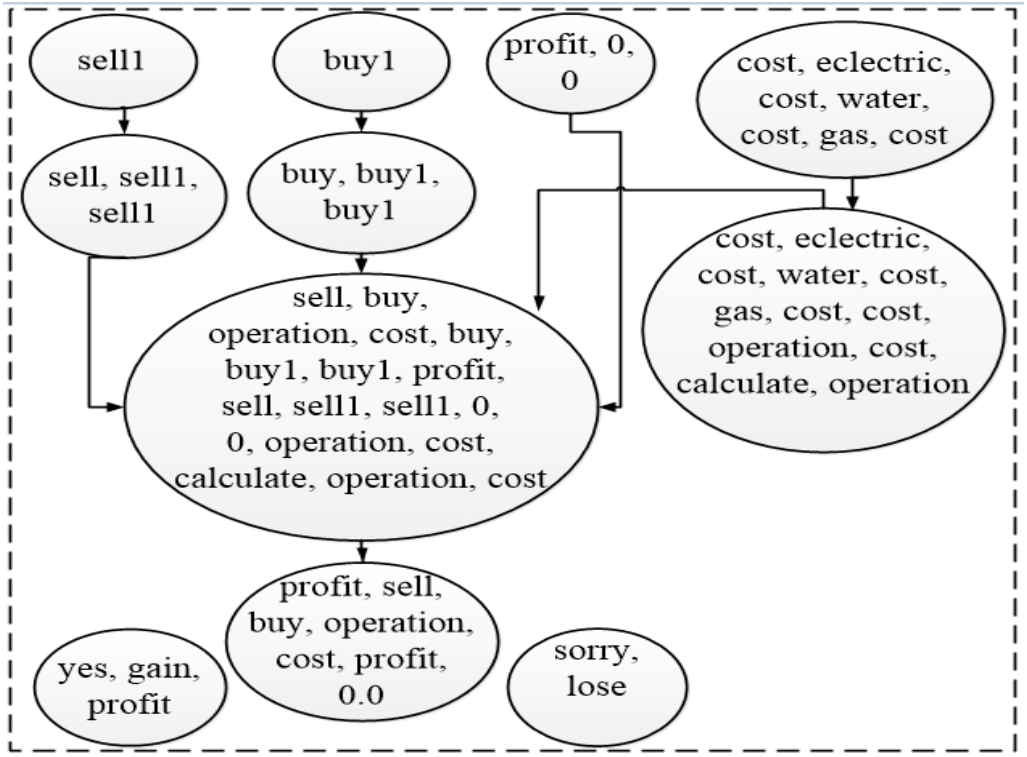

Figure 5.4: Node Predecessor-node Dependency Graph (NPDG) corresponding to Figure 5.2

synset of WordNet [1] containing single word is only considered. Words are processed to identify the basic concept of multiple form of each word because bug localization using textual similarity depends on the close relationships between the words. For example, a developer may use 'contains' keyword whereas QA uses 'containing'. Here, the basic concept or root form is the same (that is, 'contain') though multiple forms of words are used. To get the basic word among multiple words, stemming is commonly used. In SBL, Porter stemming is applied on the remaining textual words to get the root

[1]The details of WordNet- https://wordnet.princeton.edu/

form of the words. After applying these pre-processing techniques, a list of valid code corpus is obtained.

As statement level bug localization depends on the similarity between the bug report and statements of the source code, increasing valid information of a statement is an implicit demand. Hence, a super node is generated by combining each node with its predecessor nodes' corpora because predecessors may affect its successors, resulted in a NPDG (that is, shown in Figure 5.4). Figure 5.4 is a NPDG getting from Figure 5.2 by combining each node with its predecessor nodes. In Figure 5.2, $int\, sell = sell1$ is a node while in NPDG this node is incorporated with its predecessor node (that is, $int\, sell1$). As a result, for that statement the number of corpus is increased. Similar to this, $profit+ = sell - buy - operationCost$ has four predecessor nodes (termed as, *profit*, *sell*, *buy* and *operationCost*) and so for this node, Figure 5.4 accumulates the corpora of that node and corpora of its four predecessor nodes.

After the generation of NPDG, the similarity score is measured between the bug report and each node of NPDG. NPDG is traversed to find the shared terms between the bug corpora and each node corpora of NPDG. By traversing the graph, the frequencies of shared terms are calculated to measure term-frequency (tf) which is used in VSM. As several kinds of tf are already proposed to improve the performance of VSM (for example, logarithm, augmented and boolean variants), logarithm variant performs better than others [92–94]. Hence, $tf(t, n)$ and $tf(t, b)$ are calculated according to

Equation 5.1 and 5.2.

$$tf(t, n) = \log f_{tn} + 1 \tag{5.1}$$

$$tf(t, b) = \log f_{tb} + 1 \tag{5.2}$$

Here, f_{tn}, f_{tb} are the frequency of a term of NPDG node and bug report respectively. These are used for measuring the similarity between bug report and each statement of the source code. The similarity is calculated using VSM as shown in Equation 5.3.

$$N_s(n, b) = VSM(n, b) = cos(n, b) = (\sum_{t \in b \cap n} (\log f_{tn} + 1) \times (\log f_{tb} + 1)) \times \frac{1}{\sqrt{\sum_{t \in n} (\log f_{tn} + 1)^2}} \times \frac{1}{\sqrt{\sum_{t \in b} (\log f_{tb} + 1)^2}} \tag{5.3}$$

Here, $N_s(n, b)$ denotes the similarity score of each node for a bug where f_{tb} and f_{tn} are the term frequency of the bug report and each node respectively. A statement containing large score (N_s) represents that the statement has large similarity with the bug report. Similarly, low score of a statement indicates that the statement has minimum effect on that bug.

Now each statement score is weighted by the score of the method (shown in Equation 5.4). This is because a statement contains a few number of words compared to a method [1]. So, it is more obvious that the large ranking scored

methods' are more likely liable for being buggy.

$$SBL_{score} = N_s \times M_s \tag{5.4}$$

Here, M_s and N_s represent the score of a method containing statement s and the statement similarity score, respectively. SBL_{score} denotes the ranking score of each statement after weighting by the method score, M_s. Finally, statements are ranked based on the descending order of this SBL_{score}.

5.3 Result Analysis

To measure the effectiveness of SBL, several bug reports from three open source projects named as Eclipse, SWT and PasswordProtector are considered. The experiments are conducted for validating the ranking of buggy statements. Top N Rank, Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) are the metrics used to validate the ranking of statements provided by SBL. All of these experiments are discussed in the following.

5.3.1 Data Collection

Eclipse, SWT and PasswordProtector are collected as the subjects of evaluation. The details of each considered projects are given below.

- Eclipse: Eclipse is a widely used open source Integrated Development Environment (IDE). This IDE contains a workspace and an extensible plug-in system. Eclipse is written in Java and for developing Java

applications, Eclipse is commonly used. Besides this, Eclipse may also be used to develop applications in other programming languages such as C, C++, etc. Here, different versions of Eclipse[2] (for example, version 2.1.0, 3.0.1, 3.0.2 and 3.1.0) are chosen which contain large volume of source code. Table 5.1 shows the volume of Eclipse 3.0.2 project. This project contains 12,863 classes and 95,341 methods,

- SWT: SWT is another widely used open source project which is a vital component of Eclipse. Several versions (for example, version 2.1.0, 3.0.1, 3.0.2 and 3.1.0) of SWT[3] are also considered. This project also has a large volume of source code. For example, SWT 3.0.2 contains 489 classes and 18,784 methods as shown in Table 5.1,

- PasswordProtector: PasswordProtector[4] is also an open source project used to encrypt passwords for accessing multiple websites. The encrypted passwords are saved and displayed. Additionally, if password is needed for a specific website then based on some criteria, passwords are generated. After completing the authentication process, if an user wants, he/she can store the password and username. Later by website name, the user can perform searching to see the full information. For generating password, several criteria (for example, underscore or lower case) are considered. The volume of this project is comparatively small

[2] `https://bugs.eclipse.org`
[3] `http://www.eclipse.org/swt/`
[4] `https://raw.githubusercontent.com/shanto-Rahman/SBL/master/PasswordProtector.zip`

Table 5.1: The volume of considered bugs

Project name	**Class**	**Method**
Eclipse 3.0.2	12,863	95,341
SWT 3.0.2	489	18,784
PasswordProtector	26	236

than above mentioned two projects. Table 5.1 shows that 26 classes and 236 methods are available in PasswordProtector.

Large projects such as Eclipse and SWT contain millions of statements. The benchmark open source projects have lack of available patches with statement level bug fixing because still none of the research is conducted on statement level bug localization. Hence, SBL is evaluated using 100 bugs from all these three projects. These bugs are manually collected from Bugzilla[5] for Eclipse and SWT. Along with each bug, corresponding patched files are also collected to validate the results getting from SBL. Figure 5.5 and Figure 5.6 are the instances of collecting a bug report and corresponding patched files respectively which are taken from Bugzilla. Meanwhile, the information of PasswordProtector is collected from the developers of the projects [95]. The collected bugs are chosen using the following criteria.

- The bugs have published and approved patches which are well-known, documented and reproducible,
- Bugs are often categorized as resolved or verified or fixed. So, only the valid bugs that actually have been fixed are considered here,

[5]The details of Bugzilla - https://www.bugzilla.org/docs/2.16/html/whatis.html

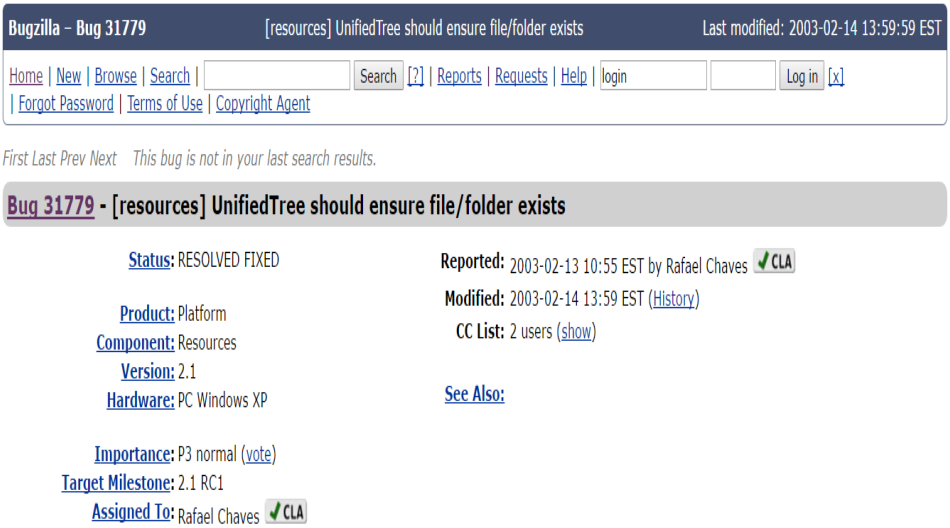

Figure 5.5: Bug report Id 31779

- Bugs at all levels of severity are included,
- The considered bugs have statement level fixed patches which are identified manually by the authors.

The details of each considered bugs are available in Github [95] where bug id, description and actual buggy locations are given.

5.3.2 Evaluation Metrics

To measure the performance of SBL, Top N Rank, MRR (Mean Reciprocal Rank) and MAP (Mean Average Precision) have been used as metrics. These are widely-used metrics for measuring the effectiveness of a retrieval and ranking system [96], [81]. For all of these, the higher the value is, the better the performance will be. Details are discussed below.

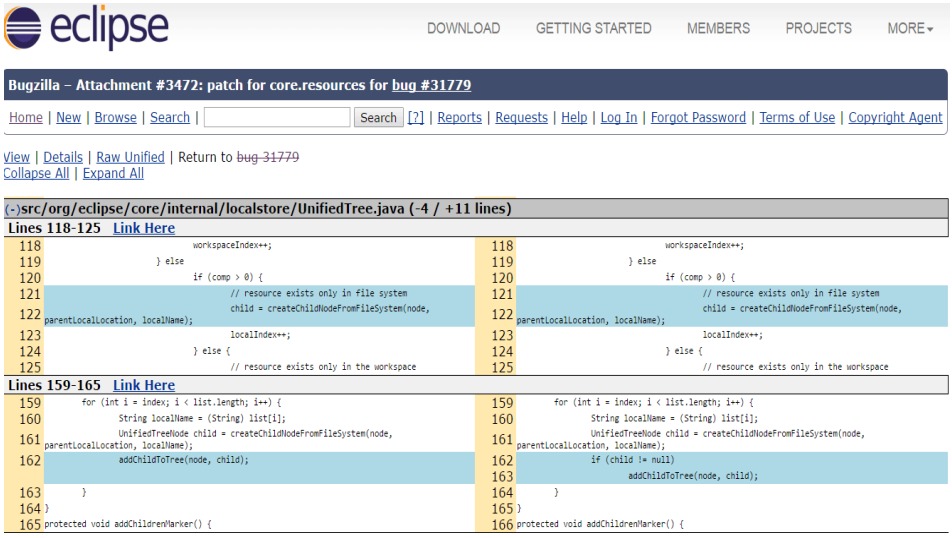

Figure 5.6: Patch of Eclipse bug report Id 31779

1. Top N Rank: This is the number of bugs that are localized in the top N ($N = 1, 5, 10, ...$ for this system) rank. As an example, $N = 5$ means that within top 5 suggestions, the buggy statements are actually obtained. If one of the fixed files of a bug is in the result set, it is marked as localized [56].

2. MRR: A reciprocal rank is the multiplicative inverse of the rank of the first correct result of a query [56]. For example, if a bug is localized in rank position 4, the reciprocal rank is $\frac{1}{4}$. So, the range of MRR will be $0 \leq MRR \leq 1$. MRR is the average of all the ranks of results of a set of query. MRR is calculated using Equation (5.5). Here, n and r_i are the number of queries and rank of a query i, respectively. Higher

MRR represents better accuracy in bug localization.

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{r_i} \tag{5.5}$$

3. MAP: MAP indicates how successfully the system is able to locate all the buggy files, not only the first one as considered in MRR [56]. MAP is the mean of the average precision values of a set of query. The precision of top n files is calculated using Equation (5.6). The higher the value of MAP is, the better the performance of bug localization obtains.

$$P(n) = \frac{\text{number of buggy files localized in top n}}{n} \tag{5.6}$$

Average precision of a query (bug report) is calculated using Equation (5.7).

$$AverageP = \sum_{n=1}^{K}\frac{p(n) \times b(n)}{\text{number of buggy files}} \tag{5.7}$$

Where n is the rank, K is the total number of ranked files respectively. $b(n)$ indicates if n^{th} file contains bug or not and $p(n)$ is the precision of top n files as mentioned before. MAP is calculated by taking mean of these average precision values [81].

5.3.3 Research Questions and Evaluation

In this section, SBL is validated by addressing two research questions. **RQ1** states that how many bugs are successfully located by suggesting buggy statements. Along with this, **RQ2** demonstrates the effects of considering predecessors of each node. The below discussion holds the detail description of each research question along with their evaluation.

RQ1: How many bugs are successfully located at statement level?

To answer this question, three open source projects described in Section 5.3.1 are considered. For each bug, the ranked list of buggy statements suggested by SBL are compared with the actual buggy statements. The actual buggy statements are collected from bug tracking system such as Bugzilla [97]. Figure 5.6 shows an example of collecting actual buggy statements for bug Id - 31779. If the actual buggy statements are ranked at the top 1, top 5, top 10 or top 20, that bug is considered effectively localized. MRR and MAP metrics are also used to prove the efficacy of the proposed SBL (described in Section 5.3.2).

Table 5.2 presents that for Eclipse, SBL locates 28% buggy statements at the top position. This refers that developers need not traverse any more suggested statements for this 28% bugs, because at first position, the actual buggy statements are ranked. Besides, 58%, 66% and 82% bugs are located at top 5, top 10 and top 20 respectively. In case of SWT, SBL locates 37.5%, 62.5%, 72.5% and 85% bugs at the top 1, top 5, top 10 and top 20 which

Table 5.2: The performance of SBL by considering 100 bugs

Project name	**Top 1**	**Top 5**	**Top 10**	**Top 20**	**MRR**	**MAP**
Eclipse	14 (28%)	29 (58%)	33 (66%)	41 (82%)	45.8%	38.5%
SWT	15 (37.5%)	25 (62.5%)	29 (72.5%)	34 (85%)	51.3%	44.2%
PasswordProtector	4 (40%)	9 (90%)	10 (100%)	10 (100%)	66%	61%

are also promising. This represents that 37.5% bugs are localized without consuming any searching time for identifying buggy locations, as 37.5% bugs are localized at the first position. For PasswordProtector, 40% bugs are localized at the top 1, 90% bugs are at top 5 and within top 10 suggestion, 100% bugs are correctly localized. These results indicate that SBL is effective enough for localizing software bugs at the statement level.

The large scores of MRR and MAP indicate better accuracy of bug localization. For Eclipse, SWT and PasswordProtector, MRR of SBL is 45.8%, 51.3% and 66% respectively while MAP is 38.5%, 44.2% and 61% respectively. These results also show that SBL can effectively localize the buggy statements.

RQ2: Does the consideration of predecessor nodes improve the accuracy of bug localization?

SBL creates super node by combining each node (that is, statement) and its predecessor nodes. Hence, it is needed to show that either the consideration

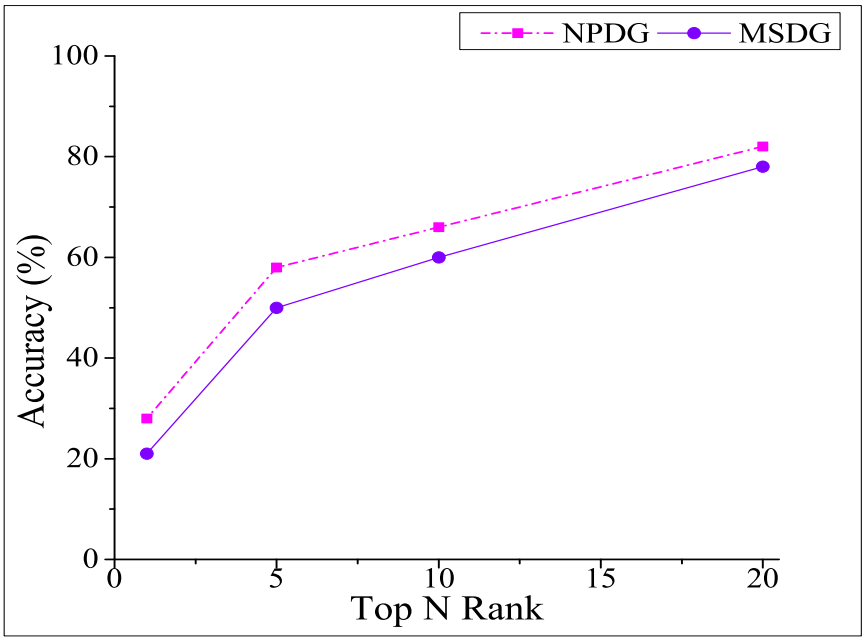

Figure 5.7: Demonstrating the effects (in case of 100 bugs) of considering predecessor nodes using Top N rank in Eclipse

of these predecessor nodes improve the bug localization accuracy or not. To validate this, SBL is compared without considering predecessor (that is, MSDG) and with consideration of predecessors (that is, NPDG).

Figure 5.7, Figure 5.8 and Figure 5.9 show that the accuracy of ranking buggy statements in Top N Rank by MSDG (that is, without predecessor) is comparatively lower than that of NPDG (that is, with predecessor) for all the studied projects. This is because, for some cases buggy statements depend on its previously executed statements. And the consideration of the predecessor nodes is more effective when the bug is not stayed in a statement rather it propagates from one statement to another.

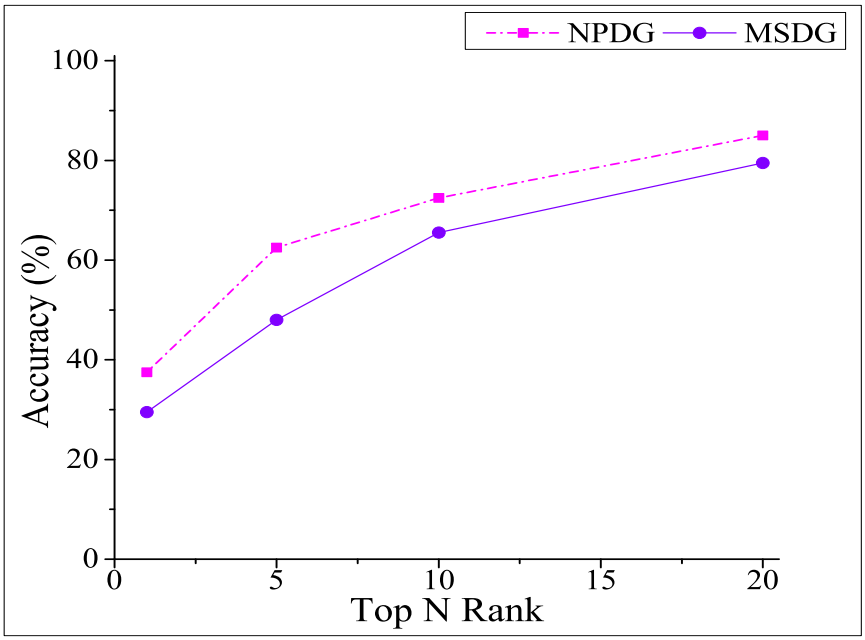

Figure 5.8: Demonstrating the effect (100 bugs) of predecessor nodes based on Top N rank of SWT

Figure 5.7 holds a comparative study between the results of considering NPDG and MSDG in case of Eclipse. The ranking of buggy statements in Eclipse shows that 21% and 28% bugs are located at the 1st position in case of MSDG and NPDG respectively. For SWT, 29% and 37% bugs are located at the 1st position for MSDG and NPDG respectively. Although Password-Protector is comparatively low volume project than other two, here also the consideration of NPDG performs better than MSDG. For PasswordProtector, 30% and 40% bugs are located at the 1st position in case of MSDG and NPDG respectively. Hence, it is clear that NPDG provides superior outcomes to MSDG.

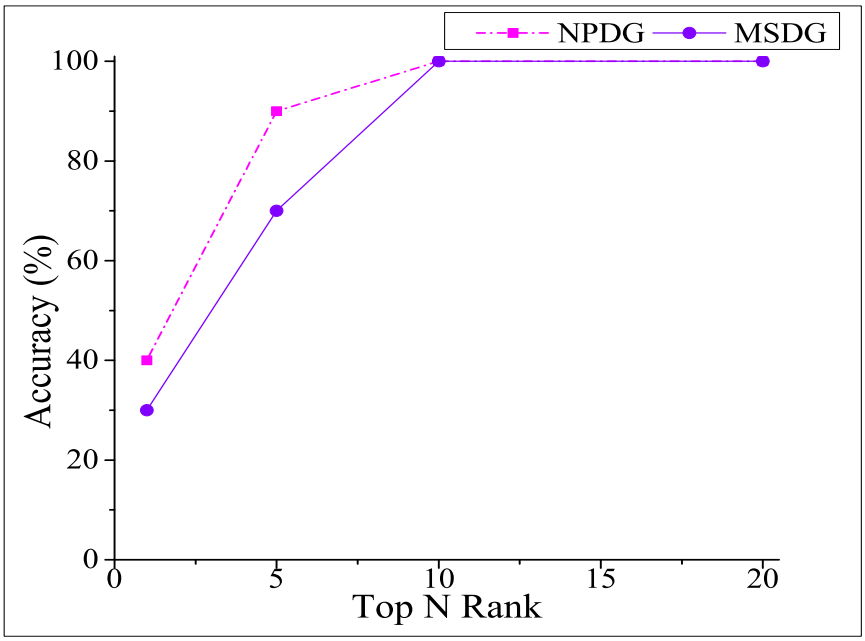

Figure 5.9: Demonstrating the effect (100 bugs) of predecessor nodes based on Top N rank PasswordProtector

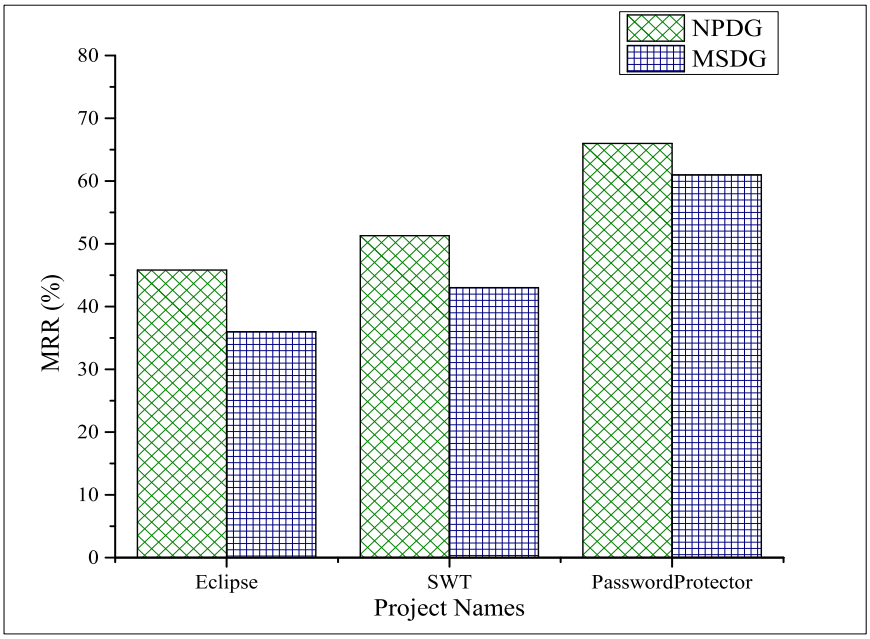

Figure 5.10: Analysis of the impact of the predecessor node MRR

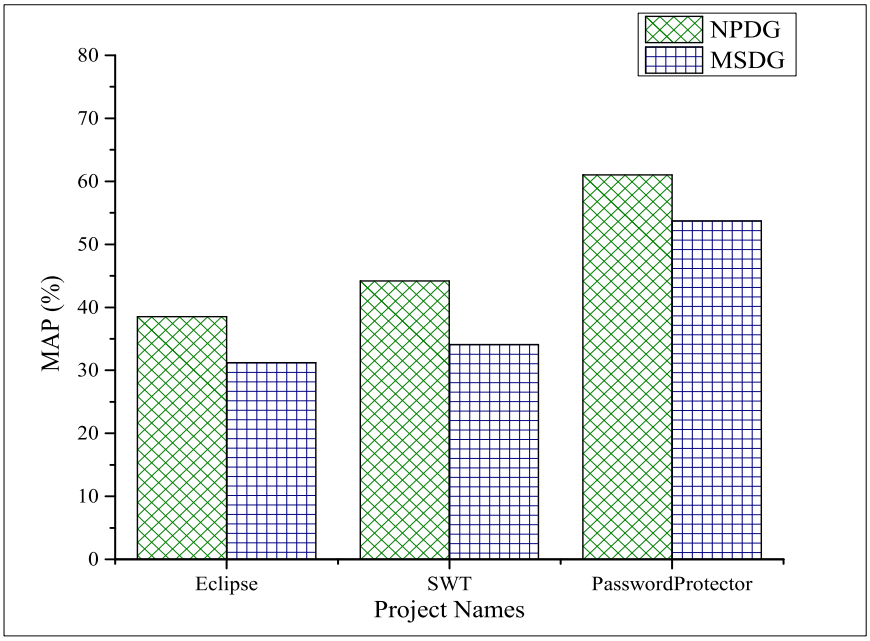

Figure 5.11: Analysis of the impact of the predecessor node MAP

The effects of using predecessors are shown in Figure 5.10 and Figure 5.11 where MRR and MAP are considered as metrics respectively. Here, the consideration of NPDG are also performed higher accuracy than MSDG. In case of MRR, by considering NPDG 9.8%, 8.3% and 5% (see Figure 5.10) improvements are performed than considering MSDG in Eclipse, SWT and PasswordProtector respectively. On the other hand, for MAP 7.3%, 10.1% and 7.3% (see Figure 5.11) accuracy improvements have been found by NPDG over MSDG in Eclipse, SWT and PasswordProtector respectively.

5.4 Summary

In this chapter, a statement level bug localization technique is proposed by establishing the relationships among the statements. For ranking buggy statements, at first buggy methods are identified (described in Chapter 4) which are used as relevant search space. Later, for each method of this relevant search space, the relationships among statements are defined by proposing MSDG. As few information is available in each node of MSDG, the information is maximized by incorporating predecessor nodes. This creates a graph named as Node Predecessors-node Dependency Graph (NPDG). Now the similarity between each node of NPDG and bug report is measured. Finally, each statement is weighted by the method score (generated from Chapter 4) which provides more accurate ranking of buggy statements. The next chapter concludes the whole thesis with future research directions.

Chapter 6

Discussion and Conclusion

Developing an accurate statement level bug localization technique is a necessary demand in the success of software development because it may eliminate the manual intervention completely. So, in this thesis, an accurate statement level bug localization technique has been proposed which can minimize the developers effort at a large rate. To do so, at first bugs are localized by discarding irrelevant solution search space. The accuracy is ensured by suggesting a list of probable buggy methods where irrelevant search space has been discarded. These ranked buggy methods are further used as minimized search space to suggest statements as buggy. In this chapter, at first a discussion regarding the solution search space minimization has been introduced and using this relevant search space later the statement level bug localization technique is discussed. Afterwards, several threats to validity are also described. Finally, this chapter concludes with future remarks.

6.1 Search Space Minimization for Enhancing Bug Localization Accuracy

Since existing bug localization techniques commonly use whole source code, the accuracy of bug localization may be affected. This is because bug localization is a probabilistic way, so the more the unnecessary information is available, the more the biasness may be occurred. To improve the accuracy, search space is minimized to suggest methods as buggy. To do so, irrelevant search space is discarded and a relevant one is considered by performing dynamic analysis of the source code and static analysis on the executed methods. Since bug localization from the bug reports is an information retrieval based technique, text of source code and bug reports are processed. Several text processing techniques such as stop words removal, programming language specific keywords removal, multi-words identification, semantic meaning extraction and stemming are applied which generate two lists of corpora one for source code and another for bug report. Finally, a modified Vector Space Model ($mVSM$) is applied on those corpora to rank the source code methods for a specific bug. In $mVSM$, the size of the method modifies the existing VSM.

Search space is minimized by identifying probable buggy methods. The accuracy of search space minimization is evaluated both theoretically and experimentally. Theoretical evaluation is done using formal methods and for the purpose of experimentation, three case studies are conducted using two

large and widely used open-source projects named as Eclipse and Mozilla. The case studies show that MBuM ranks actual buggy methods at the 1st position for three (60%) of the five bugs in both Eclipse and Mozilla, while other existing techniques cannot do the same.

6.2 A Statement Level Bug Localization Technique

In this section, the suggested buggy methods described in Section 6.1 are used as the relevant search space. And using valid information, a novel statement level bug localization technique named as SBL is proposed. For each suggested method, the relationships among the statements have been developed by proposing a Method Statement Dependency Graph (MSDG) because there exist relationships among the statements. It is noteworthy that a statement often contains very few information about a bug. To improve the valid information of each statement, the statement is combined with its predecessor nodes. This is named as a Node Predecessor-node Dependency Graph (NPDG). Then each node of NPDG is processed by applying several text processing techniques such as stop word removal, programming language specific keywords extraction, multi-word identification, semantic meaning extraction and stemming. As a result, each node of NPDG contains a bag of words. The similarity between each node (that is, statement) of NPDG and the bug report is measured using Vector Space Model (VSM) to rank the

buggy statements. Finally, based on the similarity scores, a list of statements are suggested as buggy.

Effectiveness of SBL has been evaluated by three widely used open source projects namely Eclipse, SWT and PasswordProtector. Results show that for Eclipse 45.8% Mean Reciprocal Rank (MRR) and 38.5% Mean Average Precision (MAP) can be achieved, while for PasswordProtector project 66% and 61% MRR and MAP are gained respectively. Hence, these show that SBL can successfully rank the buggy statements.

6.3 Threat to Validity

This section discusses the threats which can affect the validity of the proposed technique. The threats are identified from three perspective - internal threats, external threats and construct threats.

Internal Threats: The internal threats refers threats that affect the validity of the results which depend on the implementation of the technique and the environmental set up of the experimental procedure. The proposed technique as well as the experimental projects are implemented in java programming language. Therefore, the result gained through analyzing the experimental projects may differ when experimented in platforms other than java.

External Threats: This research requires proper quality of the bug reports. As bug report is one of the important means from which the buggy locations can be identified, the quality of the bug report should contain the

bug related information. For example, if a buggy scenario is related to the 'file import' module and the bug report holds another bug modules' information, the quality of the bug report will be significantly deteriorated. In practice, non-informative bug report can delay to fix a bug. Similarly, in automatic localization, if a bug report does not provide enough information, or provides misleading information, the performance of SBL may be adversely affected. The slight modification is handled by the proposed technique using the semantic meaning extraction from WordNet. However, if the source code is not similar with the bug report, the localization may fail, though it is a common problem in all bug localization schemes.

Finally, if the bug report does not contain proper reproducible approach, it may be hard for developers to find the accurate dynamic tracing of the source code.

Another factor is the quality of source code and the accuracy of bug localization depends on the good programming practices in naming variables, methods and classes. If a developer uses meaningless names, the performance of the proposed technique may be affected. However, in most of the well-managed projects, developers follow good naming conventions and programming practices.

Construct Threats: Construct threats are related to the metrics which are used to analyze the effectiveness of the proposed technique. The results are analyzed based on Top N Rank, MRR and MAP. Therefore, analyzing the results with other metrics can affect the generalization of the results.

6.4 Future Direction

In this thesis, a statement level bug localization technique has been proposed where probable buggy methods are considered as relevant search space. There may also some scopes to further improve this research. The necessary directions are described below.

6.4.1 Future Direction Related to this Thesis

The proposed approach discards irrelevant source code by reproducing the buggy scenario using bug report. However, proper reproducible approach of buggy scenario may not be available in bug report in all the cases. So, an alternative approach for ignoring irrelevant search space may be discovered.

As a statement contains very few information about a bug, the information of a statement should be increased. To do so, in this thesis predecessor nodes are combined with each successor node. Although this approach improves the accuracy with satisfactory level, other approaches may be introduced to increase the valid information for a statement.

This statement level bug localization is based on the relationships between statements, however other dimensions such as machine learning based techniques may be followed for improving more accurate bug localization technique.

6.4.2 Future Direction Related to the Domain

Since a statement level bug localization has been proposed using bug report in this thesis, the quality of the bug report may improve.

As statement level bug localization is new, other more accurate statement level bug localization technique may be proposed. Since the proposed technique outperforms other existing state-of-the-art techniques for open source projects, it can be applied in any industry projects.

Chapter 7

Appendix

7.0.3 Stop Words

"a", "a's", "able", "about", "above", "according", "accordingly", "across", "actually", "after", "afterwards", "again", "against", "ain't", "all", "allow", "allows", "almost", "alone", "along", "already", "also", "although", "always", "am", "among", "amongst", "an", "and", "another", "any", "anybody", "anyhow", "anyone", "anything", "anyway", "anyways", "anywhere", "apart", "appear", "appreciate", "appropriate", "are", "aren't", "around", "as", "aside", "ask", "asking", "associated", "at", "available", "away", "awfully", "b", "be", "became", "because", "become", "becomes", "becoming", "been", "before", "beforehand", "behind", "being", "believe", "below", "beside", "besides", "best", "better", "between", "beyond", "both", "brief", "but", "by", "c", "c'mon", "c's", "came", "can", "can't", "cannot", "cant", "cause", "causes", "certain", "certainly", "changes",

“clearly”, “co”, “com”, “come”, “comes”, “concerning”, “consequently”, “consider”, “considering”, “contain”, “containing”, “contains”, “corresponding”, “could”, “couldn’t”, “course”, “currently”, “d”, “definitely”, “described”, “despite”, “did”, “didn’t”, “different”, “do”, “does”, “doesn’t”, “doing”, “don’t”, “done”, “down”, “downwards”, “during”, “e”, “each”, “edu”, “eg”, “eight”, “either”, “else”, “elsewhere”, “enough”, “entirely”, “especially”, “et”, “etc”, “even”, “ever”, “every”, “everybody”, “everyone”, “everything”, “everywhere”, “ex”, “exactly”, “example”, “except”, “f”, “far”, “few”, “fifth”, “first”, “five”, “followed”, “following”, “follows”, “for”, “former”, “formerly”, “forth”, “four”, “from”, “further”, “furthermore”, “g”, “get”, “gets”, “getting”, “given”, “gives”, “h”, “had”, “hadn’t”, “happens”, “hardly”, “has”, “hasn’t”, “have”, “haven’t”, “having”, “he”, “he’s”, “hello”, “help”, “hence”, “her”, “here”, “here’s”, “hereafter”, “hereby”, “herein”, “hereupon”, “hers”, “herself”, “hi”, “him”, “himself”, “his”, “hither”, “hopefully”, “how”, “howbeit”, “however”, “i”, “i’d”, “i’ll”, “i’m”, “i’ve”, “ie”, “if”, “ignored”, “immediate”, “in”, “inasmuch”, “inc”, “indeed”, “indicate”, “indicated”, “indicates”, “inner”, “insofar”, “instead”, “into”, “inward”, “is”, “isn’t”, “it”, “it’d”, “it’ll”, “it’s”, “its”, “itself”, “j”, “just”, “k”, “keep”, “keeps”, “kept”, “know”, “knows”, “known”, “l”, “last”, “lately”, “later”, “latter”, “latterly”, “least”, “less”, “lest”, “let”, “let’s”, “like”, “liked”, “likely”, “little”, “look”, “looking”, “looks”, “ltd”, “m”, “mainly”, “many”, “may”, ”maybe”, ”me”, ”mean”, “meanwhile”,

“merely”, “might”, “more”, “moreover”, “most”, “mostly”, “much”, “must”, “my”, “myself”, “n”, “name”, “namely”, “nd”, “near”, “nearly”,

“necessary”, “need”, “needs”, “neither”, “never”, “nevertheless”, “new”, “next”, “nine”, “no”, “nobody”, “non”, “none”, “noone”, “nor”, “normally”, “not”, “nothing”, “novel”, “now”, “nowhere”, “o”, “obviously”, “of”, “off”, “often”, “oh”, “ok”, “okay”, “old”, “on”, “once”, “one”, “ones”, “only”, “onto”, “or”, “other”, “others”, “otherwise”, “ought”, “our”, “ours”, “ourselves”, “out”, “outside”, “over”, “overall”, “own”, “p”, “particular”, “particularly”, “per”, “perhaps”, “placed”, “please”, “plus”, “possible”, “presumably”, “probably”, “provides”, “q”, “que”, “quite”, “qv”, “rather”, “rd”, “re”, “really”, “reasonably”, “regarding”, “regardless”, “regards”, “relatively”, “respectively”, “right”, “s”, “said”, “same”, “saw”, “say”, “saying”, “says”, “second”, “secondly”, “see”, “seeing”, “seem”, “seemed”, “seeming”, “seems”, “seen”, “self”, “selves”, “sensible”, “sent”, “serious”, “seriously”, “seven”, “several”, “shall”, “she”, “should”, “shouldn’t”, “since”, “six”, “so”, “some”, “somebody”, “somehow”, “someone”, “something”, “sometime”, “sometimes”, “somewhat”, “somewhere”, “soon”, “sorry”, “specified”, “specify”, “specifying”, “still”, “sub”, “such”, “sup”, “sure”, “t”, “t’s”, “take”, “taken”, “tell”, “tends”, “th”, “than”, “thank”, “thanks”, “thanx”, “that”, “that’s”, “thats”, “the”, “their”, “theirs”, “them”, “themselves”, “then”, “thence”, “there”, “there’s”, “thereafter”, “thereby”, “therefore”, “therein”, “theres, ‘thereupon’, ‘these’, ‘they’, ‘they’d’, ‘they’ll’, ‘they’re’, ‘they’ve’, ‘think’, ‘third’, ‘this’, ‘thorough’,

‘thoroughly’, ‘those’, ‘though’, ‘three’, ‘through’, ‘throughout’, ‘thru’, ‘thus’, ‘to’, ‘together’, ‘too’, ‘took’, ‘toward’, ‘towards’, ‘tried’, ‘tries’, ‘truly’, ‘try’, ‘trying’, ‘twice’, ‘two’, ‘u’, ‘un’, ‘under’, ‘unfortunately’, ‘unless’, ‘unlikely’, ‘until’, ‘unto’, ‘up’, ‘upon’, ‘us’, ‘use’, ‘used’, ‘useful’, ‘uses’, ‘using’, ‘usually’, ‘uucp’, ‘v’, ‘value’, ‘various’, ‘very’, ‘via’, ‘viz’, ‘vs’, ‘w’, ‘want’, ‘wants’, ‘was’, ‘wasn’t’, ‘way’, ‘we’, ‘we’d’, ‘we’ll’, ‘we’re’, ‘we’ve’, ‘welcome’, ‘well’, ‘went’, ‘were’, ‘weren’t’, ‘what’, ‘what’s’, ‘whatever’, ‘when’, ‘whence’, ‘whenever’, ‘where’, ‘where’s’, ‘whereafter’, ‘whereas’, ‘whereby’, ‘wherein’, ‘whereupon’, ‘wherever’, ”whether”, ”which”, ”while”, ‘whither’, ‘who’, ‘who’s’, ‘whoever’, ‘whole’, ‘whom’, ‘whose’, ‘why’, ‘will’, ‘willing’, ‘wish’, ‘with’, ‘within’, ‘without’, ‘won’t’, ‘wonder’, ‘would’, ‘would’, ‘wouldn’t’, ‘x’, ‘y’, ‘yes’, ‘yet’, ‘you’, ‘you’d’, ‘you’ll’, ‘you’re’, ‘you’ve’, ‘your’, ‘yours’, ‘yourself’, ‘yourselves’, ‘z’, ‘zero’, ‘quot’, ”

7.0.4 Language Specific Keywords

“public”, “static”, “void”, “abstract”, “continue”, “for”, “new”, “switch”, “assert”, “default”, “goto”, “package”, “synchronized”, “boolean”, “do”, “if”, “private”, “this”, “break”, “double”, “implements”, “protected”, “throw”, “byte”, “else”, “import”, “public”, “throws”, “case”, “enum”, “instanceof”, “return”, “transient”, “catch”, “extends”, “int”, “short”, “try”, “char”, “final”, “interface”, “static”, “void”, “class”, “finally”, “long”, “strictfp”, “volatile”, “const”, “float”, “native”, “super”, “while”, “org”, “eclipse”, “swt”, “string”, “main”, “args”, “null”, “this”, “extends”,

“true”, “out”, “println”, “try”, “catch”, “e”, “exception”, “stack”, “trace”, “Scanner”, “in”, “int”, “double”,

Bibliography

[1] Laura Moreno, Wathsala Bandara, Sonia Haiduc, and Andrian Marcus. On the relationship between the vocabulary of bug reports and source code. In *2013 IEEE International Conference on Software Maintenance*, pages 452–455. IEEE, 2013.

[2] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63. ACM, 2014.

[3] Nouh Alhindawi, Natalia Dragan, Michael L Collard, and Jonathan I Maletic. Improving feature location by enhancing source code with stereotypes. In *2013 IEEE International Conference on Software Maintenance*, pages 300–309. IEEE, 2013.

[4] Bugzilla. `https://www.bugzilla.org/`. Accessed: 2016-04-15.

[5] Team foundation server (tfs). `https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx`. Accessed: 2016-04-15.

[6] Peng Shao, Travis Atkison, Nicholas A Kraft, and Randy K Smith. Combining lexical and structural information for static bug localisation. *International Journal of Computer Applications in Technology*, 44(1):61–71, 2012.

[7] Stacy K Lukins, Nicholas Kraft, Letha H Etzkorn, et al. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of the 15th Working Conference on Reverse Engineering, (WCRE, 2008)*, pages 155–164. IEEE, 2008.

[8] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[9] William M Wilson, Linda H Rosenberg, and Lawrence E Hyatt. Automated analysis of requirement specifications. In *Proceedings of the 19th international conference on Software engineering*, pages 161–171. ACM, 1997.

[10] O Lahav and D Mioduser. Multisensory virtual environment for supporting blind persons acquisition of spatial cognitive mapping, orientation, and mobility skills. In *Proceedings of the Third International Conference on Disability, Virtual Reality and Associated Technologies, ICDVRAT 2000*, pages 53–58, 2000.

[11] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE, 2012)*, pages 14–24. IEEE, 2012.

[12] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010.

[13] `https://projectquotes.wordpress.com/2011/01/11/failure-is-not-an-option-it-comes-bundled-with-the-software`.

[14] Sai Zhang and Congle Zhang. Software bug localization with markov logic. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 424–427. ACM, 2014.

[15] James S Collofello and Scott N Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of systems and software*, 9(3):191–195, 1989.

[16] Shanto Rahman and Sakib Kazi. An appropriate method ranking approach for localizing bugs using minimized search space. In *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), (accepted), 2016*.

[17] Moth in the machine: Debugging the origins of 'bug'. *Computerworld staff*, September 3, 2011.

[18] `https://en.wikipedia.org/wiki/Software_bug`.

[19] Yutaro Kashiwa, Hayato Yoshiyuki, Yusuke Kukita, and Masao Ohira. A pilot study of diversity in high impact bugs. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 536–540. IEEE, 2014.

[20] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 300–310. ACM, 2011.

[21] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 82–91. ACM, 2014.

[22] Harold Valdivia Garcia and Emad Shihab. Characterizing and predicting blocking bugs in open source projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 72–81. ACM, 2014.

[23] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *Mining software repositories (MSR), 2010 7th IEEE working conference on*, pages 11–20. IEEE, 2010.

[24] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 237–246. IEEE Press, 2013.

[25] Emad Shihab, Audris Mockus, Yasutaka Kamei Bram, et al. Highimpact defects: a study of breakage and surprise defects. In *in Proceedings of ESEC/FSE 2011*. Citeseer.

[26] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT*

workshop on Program analysis for software tools and engineering, pages 1–8. ACM, 2007.

[27] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE, 2013) IEEE/ACM*, pages 345–355. IEEE, 2013.

[28] Debjit Pal and Rizwan Mohiuddin. Automated bug localization of software programs: A survey report1.

[29] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 351–360. IEEE, 2009.

[30] David Binkley and Dawn Lawrie. Information retrieval applications in software maintenance and evolution. *Encyclopedia of Software Engineering*, pages 454–463, 2010.

[31] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[32] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[33] Rupak Majumder Thomas A. Henzinger, Ranjit Jhala and Kenneth L. McMillan. Blast. In *http://goto.ucsd.edu/rjhala/blast.html*, 2002.

[34] Chris Lattner. Blast. In *Clang: a C language family frontend for LLVM. http://clang.llvm.org/.*, 2007.

[35] Microsoft research. In *SLAM Toolkit. http://research.microsoft.com/en-us/projects/slam/.*, 2010.

[36] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiene Tahar, and Alf Larsson. Jcharming: A bug reproduction approach using crash traces and directed model checking. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 101–110. IEEE, 2015.

[37] Frank Eichinger and Klemens Bohm. Software-bug localization with graph mining. In *Managing and Mining Graph Data*, pages 515–546. Springer, 2010.

[38] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[39] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[40] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM, 2005.

[41] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. *ACM SIGPLAN Notices*, 38(5):141–154, 2003.

[42] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[43] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351. ACM, 2005.

[44] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.

[45] Giuseppe Scanniello and Andrian Marcus. Clustering support for static concept location in source code. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 1–10. IEEE, 2011.

[46] Graeme D Ruxton. The unequal variance t-test is an underused alternative to student's t-test and the mann–whitney u test. *Behavioral Ecology*, 17(4):688–690, 2006.

[47] Emily Hill, Shivani Rao, and Avinash Kak. On the use of stemming for concern location and bug localization in java. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 184–193. IEEE, 2012.

[48] Julie B Lovins. *Development of a stemming algorithm*. MIT Information Processing Group, Electronic Systems Laboratory Cambridge, 1968.

[49] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[50] Chris D Paice. An evaluation method for stemming algorithms. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 42–50. Springer-Verlag New York, Inc., 1994.

[51] Thomas Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 50–57. ACM, 1999.

[52] Scott Deerwester. Improving information retrieval with latent semantic indexing. 1988.

[53] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[54] Chakkrit Tantithamthavorn, Akinori Ihara, and Ken-ichi Matsumoto. Using co-change histories to improve bug localization performance. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, pages 543–548. IEEE, 2013.

[55] Bunyamin Sisman and Avinash C Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 50–59. IEEE Press, 2012.

[56] Shanto Rahman, Kishan Ganguly, and Sakib Kazi. An improved bug localization using structured information retrieval and version history. In *Proceedings of the 18th International Conference on Computer and Information Technology (ICCIT), (accepted), 2015*.

[57] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, 2013.

[58] Shaowei Wang, Daniel Lo, and Julia Lawall. Compositional vector space models for improved bug localization. In *IEEE International Conference on Software Maintenance and Evolution (ICSME, 2014)*, pages 171–180. IEEE, 2014.

[59] David D Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *Machine learning: ECML-98*, pages 4–15. Springer, 1998.

[60] Alex Smola and SVN Vishwanathan. Introduction to machine learning. *Cambridge University*, pages 32–34, 2008.

[61] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 263–272. IEEE, 2011.

[62] Brent D Nichols. Augmented bug localization using past bug information. In *Proceedings of the 48th Annual Southeast Regional Conference*, page 61. ACM, 2010.

[63] Xuan-Hieu Phan and Cam-Tu Nguyen. Gibbslda++: Ac. *C++ implementation of latent Dirichlet allocation (LDA)*, 2007.

[64] Tien-Duy B Duy, Shaowei Wang, and David Lo. Multi-abstraction concern localization. 2013.

[65] Norman Wilde, Juan A Gomez, Thomas Gust, and Douglas Strasburg. Locating user functionality in old code. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 200–205. IEEE, 1992.

[66] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[67] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[68] Da Huo, Tao Ding, Collin McMillan, and Malcom Gethers. An empirical study of the effects of expert knowledge on bug reports. In *IEEE International Conference on Software Maintenance and Evolution (ICSME, 2014)*, pages 1–10. IEEE, 2014.

[69] Ripon K Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E Perry. On the effectiveness of information retrieval based bug localization for c programs. In *IEEE International Conference on Software Maintenance and Evolution (ICSME, 2014)*, pages 161–170. IEEE, 2014.

[70] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 137–148. IEEE, 2006.

[71] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52. ACM, 2011.

[72] Shivani Rao, Henry Medeiros, and Avinash Kak. Comparing incremental latent semantic analysis algorithms for efficient retrieval from software libraries for bug localization. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–8, 2015.

[73] Bunyamin Sisman and Avinash C Kak. Assisting code search with automatic query reformulation for bug localization. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR, 2013)*, pages 309–318. IEEE, 2013.

[74] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *IEEE International Conference on Software Maintenance and Evolution (ICSME, 2014)*, pages 181–190. IEEE, 2014.

[75] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699. ACM, 2014.

[76] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vá Clav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[77] Giuliano Antoniol and Yann-Gael Gueheneuc. Feature identification: An epidemiological metaphor. *Software Engineering, IEEE Transactions on*, 32(9):627–641, 2006.

[78] Giuliano Antoniol and YannGael Gueheneuc. Feature identification: a novel approach and a case study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM'05*, pages 357–366. IEEE, 2005.

[79] Scott C. Deerwester, Susan T Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JAsIs*, 41(6):391–407, 1990.

[80] Jim Woodcock and Jim Davies. Using z. specification, refinement, and proof. 1996.

[81] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[82] William B Frakes. Stemming algorithms. pages 131–160, 1992.

[83] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 803–814. ACM, 2014.

[84] Manos Renieres and Steven P Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30–39. IEEE, 2003.

[85] Małgorzata Steinder and Adarshpal S Sethi. Probabilistic fault localization in communication systems using belief networks. *Networking, IEEE/ACM Transactions on*, 12(5):809–822, 2004.

[86] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. Provably optimal and human-competitive results in sbse for

spectrum based fault localisation. In *Search Based Software Engineering*, pages 224–238. Springer, 2013.

[87] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[88] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.

[89] Kunrong Chen and Vaclav Rajlich. Ripples: tool for change in legacy software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 230. IEEE Computer Society, 2001.

[90] Vaclav Rajlich. A model for change propagation based on graph rewriting. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 84–91. IEEE, 1997.

[91] Mark Alan Finlayson. Java libraries for accessing the princeton wordnet: Comparison and evaluation. In *Proceedings of the 7th Global Wordnet Conference, Tartu, Estonia*, 2014.

[92] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*, volume 283. Addison-Wesley Reading, 2010.

[93] Susan T Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, 1991.

[94] E Garcia. Description, advantages and limitations of the classic vector space model, 2006.

[95] Shanto Rahman. shanto-rahman/source code: 2016. `https://github.com/shanto-Rahman/SBL`.

[96] Harsh H Pareek and Pradeep K Ravikumar. A representation theory for ranking functions. In *Advances in Neural Information Processing Systems*, pages 361–369, 2014.

[97] Bug repository. `https://bugs.eclipse.org/bugs/`.